Algorithm Analysis and Design
- A Contemporary Perspective

Sandeep Sen and Amit Kumar
Department of Computer Science and Engineering,
IIT Delhi, New Delhi 110016, India.
E-mail: ssen, amitk@cse.iitd.ernet.in

April 27, 2017
# Contents

1 Model and Analysis .................................................. 6
   1.1 Computing Fibonacci numbers .................................. 6
   1.2 Fast Multiplication .............................................. 8
   1.3 Model of Computation ........................................... 10
   1.4 Other models ..................................................... 11
      1.4.1 External memory model ................................... 11
      1.4.2 Parallel Model ............................................. 12

2 Warm up problems ..................................................... 15
   2.1 Euclid’s algorithm for GCD ..................................... 15
      2.1.1 Extended Euclid’s algorithm ............................... 16
   2.2 Finding the \( k \)-th element ..................................... 17
      2.2.1 Choosing a random splitter ................................. 18
      2.2.2 Median of medians ......................................... 20
   2.3 Sorting words ..................................................... 20
   2.4 Mergeable heaps .................................................. 22
      2.4.1 Merging Binomial Heaps .................................. 23
   2.5 A simple semi-dynamic dictionary ............................... 23
      2.5.1 Potential method and amortized analysis ................. 24

3 Optimization I:
   Brute force and Greedy strategy .................................. 26
   3.1 Heuristic search approaches .................................... 27
      3.1.1 Game Trees * .................................................. 29
   3.2 A framework for Greedy Algorithms ............................... 32
      3.2.1 Maximum Spanning Tree .................................... 36
      3.2.2 Finding minimum weight subset ............................ 36
      3.2.3 A Scheduling Problem ..................................... 37
   3.3 Efficient data structures for Minimum Spanning Tree algorithms . 38
      3.3.1 A simple data-structure for union-find ..................... 40
3.3.2 A faster scheme ........................................ 41
3.3.3 The slowest growing function? .................... 43
3.3.4 Putting things together .............................. 44
3.3.5 Path compression only .............................. 46
3.4 Greedy in different ways .............................. 46
3.5 Compromising with Greedy ......................... 48
3.6 Gradient Descent ....................................... 50
3.6.1 Applications ........................................ 56
4 Optimization II:
   Dynamic Programming .................................. 61
   4.1 A generic dynamic programming formulation .... 62
   4.2 Illustrative examples ............................... 62
      4.2.1 Context Free Parsing ......................... 62
      4.2.2 Longest monotonic subsequence ............. 63
      4.2.3 Function approximation ....................... 64
      4.2.4 Viterbi’s algorithm for Maximum likelihood estimation .... 66
5 Searching .................................................. 68
   5.1 Skip Lists - a simple dictionary ................ 68
      5.1.1 Construction of Skip-lists ................... 68
      5.1.2 Analysis ...................................... 69
   5.2 Treaps: Randomized Search Trees ................. 71
   5.3 Universal Hashing .................................. 73
      5.3.1 Example of a Universal Hash function ....... 74
   5.4 Perfect Hash function .............................. 75
      5.4.1 Converting expected bound to worst case bound . . . . 76
   5.5 A log log N priority queue ....................... 76
6 Multidimensional Searching and Geometric algorithms .... 79
   6.1 Interval Trees and Range Trees .................. 80
      6.1.1 Two Dimensional Range Queries ............... 80
   6.2 k-d trees .......................................... 82
   6.3 Priority Search Trees .............................. 85
   6.4 Planar Convex Hull ................................. 86
      6.4.1 Jarvis March ................................ 88
      6.4.2 Graham’s Scan ................................ 88
      6.4.3 Sorting and Convex hulls .................... 88
   6.5 A Quickhull Algorithm ............................. 89
      6.5.1 Analysis ...................................... 90
6.5.2 Expected running time ........................................... 92
6.6 Point location using persistent data structure ..................... 93
6.7 Incremental construction ........................................... 96

7 Fast Fourier Transform and Applications .......................... 101
  7.1 Polynomial evaluation and interpolation .......................... 101
  7.2 Cooley-Tukey algorithm ........................................ 102
  7.3 The butterfly network ......................................... 104
  7.4 Schonage and Strassen’s fast multiplication ..................... 105

8 String matching and fingerprinting .................................. 108
  8.1 Rabin Karp fingerprinting ..................................... 108
  8.2 KMP algorithm ................................................ 110
     8.2.1 Analysis of the KMP algorithm ........................... 111
     8.2.2 Pattern Analysis ....................................... 111
  8.3 Generalized String matching ................................... 112
     8.3.1 Convolution based approach .............................. 112

9 Graph Algorithms ..................................................... 114
  9.1 Applications of DFS .......................................... 117
     9.1.1 Strongly Connected Components (SCC) ................... 117
     9.1.2 Biconnected Components ................................. 119
  9.2 Path problems ................................................ 122
     9.2.1 Bellman Ford SSSP Algorithm ........................... 123
     9.2.2 Dijkstra’s SSSP algorithm ............................... 125
     9.2.3 All Pair Shortest Paths algorithm ....................... 126
  9.3 Maximum flows in graphs ...................................... 127
     9.3.1 Max Flow Min Cut ..................................... 128
     9.3.2 Ford and Fulkerson method ............................. 129
     9.3.3 Edmond Karp augmentation strategy ..................... 130
     9.3.4 Monotonicity Lemma and bounding the iterations ....... 130
  9.4 Global MinCut ................................................ 131
     9.4.1 The contraction algorithm ............................... 132
     9.4.2 Probability of mincut ................................... 133
  9.5 Matching ....................................................... 134

10 NP Completeness and Approximation Algorithms .................. 138
  10.1 Classes and reducibility ..................................... 139
  10.2 Cook Levin theorem .......................................... 140
  10.3 Common NP complete problems ................................ 142
A  Recurrences and generating functions  
    A.1  An iterative method - summation  ................................... 204
    A.2  Linear recurrence equations  .......................................... 206
        A.2.1  Homogeneous equations  .......................................... 206
        A.2.2  Inhomogeneous equations  ........................................ 207
    A.3  Generating functions  .................................................. 208
        A.3.1  Binomial theorem  ................................................. 209
    A.4  Exponential generating functions  ................................... 209
    A.5  Recurrences with two variables  ...................................... 210

B  Refresher in discrete probability and probabilistic inequalities  212
    B.1  Probability generating functions  ................................. 213
        B.1.1  Probabilistic inequalities  ....................................... 214

C  Generating Random numbers  217
    C.1  Generating a random variate for an arbitrary distribution .... 217
    C.2  Generating random variates from a sequential file ............ 218
Chapter 1

Model and Analysis

When we make a claim like Algorithm A has running time $O(n^2 \log n)$, we have an underlying computational model where this statement is valid. It may not be true if we change the model. Before we formalize the notion of a computational model, let us consider the example of computing Fibonacci numbers.

1.1 Computing Fibonacci numbers

One of the most popular sequence is the Fibonacci sequence defined by

$$F_i = \begin{cases} 
0 & i = 0 \\
1 & i = 1 \\
F_{i-1} + F_{i-2} & \text{otherwise for } i \geq 2 
\end{cases}$$

Exercise 1.1 How large is $F_n$, the $n$-th Fibonacci number - show that

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \phi'^n) \quad \phi = \frac{1 + \sqrt{5}}{2} \quad \phi' = 1 - \phi$$

Clearly it grows exponentially with $n$. You can also prove that

$$F_n = 1 + \sum_{i=0}^{n-2} F_i$$

Since the closed form solution for $F_n$ involves the golden ratio - an irrational number, we must find out a way to compute it efficiently without incurring numerical errors or approximation as it is an integer.

Method 1
Simply use the recursive formula. Unfortunately, one can easily argue that the number of operations (primarily additions) involved is proportional to the value of \( F_n \) (just unfold the recursion tree where each internal node corresponds to an addition. As we had noted earlier this leads to an exponential time algorithm and we can’t afford it.

**Method 2**

Observe that we only need the last two terms of the series to compute the new term. So by applying the idea of dynamic programming we gradually compute the \( F_n \) starting with \( F_0 = 0 \) and \( F_1 = 1 \).

This takes time that is proportional to approximately \( n \) additions where each addition involves adding (increasingly large) numbers. The size of \( F[n/2] \) is about \( n/2 \) bits so the last \( n/2 \) computations are going to take \( \Omega(n) \) steps \(^1\) culminating in an \( O(n^2) \) algorithm.

Since the \( n \)-th Fibonacci number is at most \( n \) bits, it is reasonable to look for a faster algorithm.

**Method 3**

\[
\begin{pmatrix}
F_i \\
F_{i-1}
\end{pmatrix} = \begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix} \begin{pmatrix}
F_{i-1} \\
F_{i-2}
\end{pmatrix}
\]

By iterating the above equation we obtain

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix} = \begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^{n-1} \begin{pmatrix}
1 \\
0
\end{pmatrix}
\]

To compute \( A^n \), where \( A \) is a square matrix we can extend the following strategy for computing \( x^n \) where \( n \) is an integer.

\[
\begin{cases}
x^{2k} = (x^k)^2 & \text{for even integral powers} \\
x^{2k+1} = x \cdot x^{2k} & \text{for odd integral powers}
\end{cases}
\]

The number of multiplications taken by the above approach to compute \( x^n \) is bounded by \( 2\log n \) (Convince yourself by writing a recurrence). However, the actual running time depends on the time to multiply two numbers which in turn depends on their lengths (number of digits). If we assume that \( M(n) \) is the number of (bit-wise) steps to multiply two \( n \) bit numbers. Therefore the number of steps to implement the above approach must take into account the lengths of numbers that are being multiplied. The following observations will be useful.

The length of \( x^k \) is bounded by \( k \cdot |x| \) where \( |x| \) is the length of \( x \). Therefore, the cost of the the squaring of \( x^k \) is bounded by \( M(k|x|) \). Similarly, the

\(^1\)Adding two \( k \) bit numbers take \( \Theta(k) \)
cost of computing \(x \times x^{2k}\) can also be bound by \(M(2k|x|)\). The overall recurrence for computing \(x^n\) can be written as

\[
T_B(n) \leq T_B([n/2]) + M(n|x|)
\]

where \(T_B(n)\) is the number of bit operations to compute the \(n\)-th power using the previous recurrence. The solution of the above recurrence can be written as the following summation (by unfolding)

\[
\sum_{i=1}^{\log n} M(2^i|x|)
\]

If \(M(2i) > 2M(i)\), then the above summation can be bounded by \(O(M(n|x|))\), i.e. the cost the last squaring operation.

In our case, \(A\) is a \(2 \times 2\) matrix - each squaring operation involves 8 multiplications and 4 additions involving entries of the matrix. Since multiplications are more expensive than additions, let us count the cost of multiplications only. Here, we have to keep track of the lengths of the entries of the matrix. Observe that if the maximum size of an entry is \(|x|\), then the maximum size of an entry after squaring is at most \(2|x| + 1\) (Why?).

**Exercise 1.2** Show that the cost of computing \(A^n\) is \(O(M(n|x|))\) where \(A\) is a \(2 \times 2\) matrix and the maximum length of any entry is \(|x|\).

So the running time of computing \(F_n\) using Method 3 is dependent on the multiplication algorithm. Well, multiplication is multiplication - what can we do about it? Before that let us summarize what we know about it. Multiplying two \(n\) digit numbers using the add-and-shift method takes \(O(n^2)\) steps where each step involves multiplying two single digits (bits in the case of binary representation), and generating and managing carries. For binary representation this takes \(O(n)\) for multiplying with each bit and finally \(n\) shifted summands are added - the whole process takes \(O(n^2)\) steps.

Using such a method of multiplication implies that we cannot do better than \(\Omega(n^2)\) steps to compute \(F_n\). For any significant (asymptotically better) improvement, we must find a way to multiply faster.

### 1.2 Fast Multiplication

**Problem** Given two numbers \(A\) and \(B\) in binary, we want to compute the product \(A \times B\).
Let us assume that the numbers $A$ and $B$ have lengths equal to $n = 2^k$ - this will keep our calculations simpler without affecting the asymptotic analysis.

$$A \times B = (2^{n/2} \cdot A_1 + A_2) \times (2^{n/2} \cdot B_1 + B_2)$$

where $A_1$ ($B_1$) is the leading $n/2$ bits of $A$ ($B$). Likewise $A_2$ is the trailing $n/2$ bits of $A$. We can expand the above product as

$$A_1 \times B_1 \cdot 2^{n/2} + (A_1 \times B_2 + A_2 \times B_1) \cdot 2^{n/2} + A_2 \times B_2$$

Observe that multiplication by $2^k$ can be easily achieved in binary by adding $k$ trailing 0’s (likewise in any radix $r$, multiplying by $r^k$ can be done by adding trailing zeros). So the product of two $n$ bit numbers can be achieved by recursively computing four products of $n/2$ bit numbers.

**Exercise 1.3** What is the time to multiply using the above method - write and solve an appropriate recurrence ?

We can achieve an improvement by reducing it to three recursive calls of multiplying $n/2$ bit numbers by rewriting the coefficient of $2^{n/2}$ as follows

$$A_1 \times B_2 + A_2 \times B_1 = (A_1 + A_2) \times (B_1 + B_2) - (A_1 \times B_1) - (A_2 \times B_2)$$

Although strictly speaking, $A_1 + A_2$ is not $n/2$ bits but at most $n/2 + 1$ bits (Why ?), we can still view this as computing three separate products involving $n/2$ bit numbers recursively and subsequently subtracting appropriate terms to get the required products. Subtraction and additions are identical in modulo arithmetic (2’s complement), so the cost of subtraction can be bounded by $O(n)$. (What is maximum size of the numbers involved in subtraction ?). This gives us the following recurrence

$$T_B(n) \leq 3 \cdot T_B(n/2) + O(n)$$

where the last term accounts for addition, subtractions and shifts.

**Exercise 1.4** With appropriate terminating condition, show that the solution to the recurrence is $O(n^{\log_2 3})$.

The running time is roughly $O(n^{1.7})$ which is asymptotically better than $n^2$ and therefore we have succeeded in designing an algorithm to compute $F_n$ faster than $n^2$.

---

It is possible to multiply much faster using a generalization of the above method in $O(n \log n \log \log n)$ by a method of Schonage and Strassen. However it is quite involved as it uses Discrete Fourier Transform computation over modulo integer rings and has fairly large constants that neutralize the advantage of the asymptotic improvement unless the numbers are a few thousand bits long. It is however conceivable that such methods will become more relevant as we may need to multiply large keys for cryptographic/security requirements.
1.3 Model of Computation

Although there are a few thousand variations of the computer with different architectures and internal organization, it is best to think about them at the level of the assembly language. Despite architectural variations, the assembly level language support is very similar - the major difference being in the number of registers and the word length of the machine. But these parameters are also in a restricted range of a factor of two, and hence asymptotically in the same ball park. In summary, think about any computer as a machine that supports a basic instruction set consisting of arithmetic and logical operations and memory accesses (including indirect addressing). We will avoid cumbersome details of the exact instruction set and assume realistically that any instruction of one machine can be simulated using a constant number of available instruction of another machine. Since analysis of algorithms involves counting the number of operations and not the exact timings (which could differ by an order of magnitude), the above simplification is justified.

The careful reader would have noticed that during our detailed analysis of Method 3 in the previous sections, we were not simply counting the number of arithmetic operations but actually the number of bit-level operations. Therefore the cost of a multiplication or addition was not unity but proportional to the length of the input. Had we only counted the number of multiplications for computing $x^n$, that would only be $O(\log n)$. This would indeed be the analysis in a uniform cost model where only the number of arithmetic (also logical) operations are counted and does not depend on the length of the operands. A very common use of this model is for comparison-based problems like sorting, selection, merging, and many data-structure operations. For these problems, we often count only the number of comparisons (not even other arithmetic operations) without bothering about the length of the operands involved. In other words, we implicitly assume $O(1)$ cost for any comparison. This is not considered unreasonable since the size of the numbers involved in sorting do not increase during the course of the algorithm for majority of the commonly known sorting problems. On the other hand consider the following problem of repeated squaring $n$ times starting with 2. The resultant is a number $2^{2^n}$ which requires $2^n$ bits to be represented. It will be very unreasonable to assume that a number that is exponentially long can be written out (or even stored) in $O(n)$ time. Therefore the uniform cost model will not reflect any realistic setting for this problem.

On the other extreme is the logarithmic cost model where the cost of an operation is proportional to length of the operands. This is very consistent with the physical world and also has close relation with the Turing Machine model which is a favorite of complexity theorists. Our analysis in the previous sections is actually done with this model in mind. It is not only the arithmetic operations but also the cost of memory access is proportional to the length of the address and the operand.
The most commonly used model is something in between. We assume that for an input of size $n$, any operation involving operands of size $\log n$\(^2\) takes $O(1)$ steps. This is justified as follows. All microprocessor chips have specialized hardware circuits for arithmetic operations like multiplication, addition, division etc. that take a fixed number of clock cycles when the operands fit into a word. The reason that $\log n$ is a natural choice for a word is that, even to address an input size $n$, you require $\log n$ bits of address space. The present high end microprocessor chips have typically 2-4 GBytes of RAM and about 64 bits word size - clearly $2^{64}$ exceeds 4 GBytes. We will also use this model, popularly known as Random Access Machine (or RAM in short) except for problems that deal with numbers as inputs like multiplication in the previous section where we will invoke the log cost model. In the beginning, it is desirable that for any algorithm, you get an estimate of the maximum size of the numbers to ensure that operands do not exceed $\Omega(\log n)$ so that it is safe to use the RAM model.

1.4 Other models

There is clear trade-off between the simplicity and the fidelity achieved by an abstract model. One of the obvious (and sometimes serious) drawbacks of the RAM model is the assumption of unbounded number of registers since the memory access cost is uniform. In reality, there is a memory hierarchy comprising of registers, several levels of cache, main memory and finally the disks. We incur a higher access cost as we go from registers towards the disk and for technological reason, the size of the faster memory is limited. There could be a disparity of $10^5$ between the fastest and the slowest memory which makes the RAM model somewhat suspect for larger input sizes. This has been redressed by the external memory model.

1.4.1 External memory model

In this model, the primary concern is the number of disk accesses. Given the rather high cost of a disk access compared to any CPU operation, this model actually ignores all other costs and counts only the number of disk accesses. The disk is accessed as contiguous memory locations called blocks. The blocks have a fixed size $B$ and the simplest model is parameterized by $B$ and the size of the faster memory $M$. In this two level model, the algorithms are only charged for transferring a block between the internal and external memory and all other computation is free. In this model, the cost of sorting $n$ elements is $O\left(\frac{n}{B} \log_{M/B} \frac{M}{B}\right)$ disk accesses and this is also optimal. To see this, analyse $M/B$-way mergesort in this model. Note that, one block from

\(^2\)We can also work with $c \log n$ bits as the asymptotic analysis does not change for a constant $c$. 
each of the $M/B$ sorted streams can fit into the main memory. Using appropriate
data structures, we can generate the next $B$ elements of the output and we can write
an entire block to the output stream. So, the overall number of I-Os per phase is
$O(n/B)$ since each block is read and written exactly once. The algorithm makes
$O\left(\frac{n/B}{M/B}\right)$ passes, yielding the required bound.

There are further refinements to this model that parameterizes multiple levels and
also accounts for internal computation. As the model becomes more complicated,
designing algorithms also becomes more challenging and often more laborious.

1.4.2 Parallel Model

The basic idea of parallel computing is extremely intuitive and a fundamental in-
tellectual pursuit. At the most intuitive level it symbolises what can be achieved
by cooperation among individuals in terms of expediting an activity. It is not in
terms of division of labor (or specialisation), but actually assuming similar capabil-
ities. Putting more labourers clearly speeds up the construction and similarly using
more than one processor is likely to speed up computation. Ideally, by using $p$
processors we would like to obtain a $p$-fold speed up over the conventional algorithms;
however the principle of decreasing marginal utility shows up. One of the intuitive
reasons for this that with more processors (as with more individuals), the commu-
ication requirements tend to dominate after a while. But more surprisingly, there
are algorithmic constraints that pose serious limitations to our objective of obtaining
proportional speed-up.

This is best demonstrated in the model called PRAM (or Parallel Random Ac-
cess Machine) which is the analogue of the RAM. Here $p$ processors are connected to
a shared memory and the communication happens through reading and writing ina
globally shared memory. It is left to the algorithm designer to avoid read and write
conflicts. It is further assumed that all operations are synchronized globally and there
is no cost of synchronization. In this model, there is no extra overhead for communi-
cation as it charged in the same way as a local memory access. Even in this model, it
has been shown that it is not always possible to obtain ideal speed up. As an example
consider the elementary problem of finding the minimum of $n$ elements. It has been
proved that with $n$ processors, the time (parallel time) is at least $\Omega(\log \log n)$. For
certain problems, like depth first search of graphs, it is known that even if we use
any polynomial number of processors, we cannot obtain polylogarithmic time ! So,
clearly not all problems can be parallelized effectively.

A more realistic parallel model is the interconnection network model that has an
underlying communication network, usually a regular topology like a two-dimensional
mesh, hypercube etc. These can be embedded into VLSI chips and can be scaled
according to our needs. To implement any any parallel algorithm, we have to design
efficient schemes for data routing.

A very common model of parallel computation is a hardware circuit comprising of basic logic gates. The signals are transmitted in parallel through different paths and the output is a function of the input. The size of the circuit is the number of gates and the (parallel) time is usually measured in terms of the maximum path length from any input gate to the output gate (each gate contributes to a unit delay). Those familiar with circuits for addition, comparison can analyse them in this framework. The carry-save adder is a low-depth circuit that adds two $n$-bit numbers in about $O(\log n)$ steps which is much faster than a sequential circuit that adds one bit at a time taking $n$ steps.

**An example** Given numbers $x_1, x_2 \ldots x_n$, consider problem of computing the terms $S_i = \sum_{j=1}^{i} x_j$ for all $1 \leq i \leq n$. Each term corresponds to a partial sum. It is trivial to compute all the partial sums in $O(n)$ steps. Computing $S_i$ for each $i$ can be done in parallel using a binary tree of depth $\lceil \log n \rceil$ where the inputs are given at the leaf nodes and each internal node corresponds to a summation operation. All the summations at the same level can be done simultaneously and the final answer is available at the root. Doing this computation independently for each $S_i$ is wasteful since $S_{i+1} = S_i + x_{i+1}$ that will about $O(n^2)$ additions compared to the sequential complexity of $O(n)$.

Instead we use the following idea. Add every odd-even pair of inputs into a single value $y_{i/2} = x_{i-1} + x_i$, for every even $i$ (assume $n$ is a power of two). Now compute the partial sums $S'_1, S'_2 \ldots S'_{n/2}$ recursively. Note that $S'_j = \sum_{k=1}^{2j} x_k = S_{2j}$, i.e., half the terms can be computed this way. To obtain $S_{2j+1}$, $0 \leq j \leq n/2 - 1$, add $x_{2j+1}$ to $S'_j$. This can also be done simultaneously for all terms.

The total parallel time $T^{||}(n) = T^{||}(n/2) + 2$ where the last term corresponds to the two additional operations to combine the terms. This yields $T^{||}(n) = O(\log n)$. The total number of operations used

$$W(n) = W(n/2) + 2n \quad \text{or} \quad W(n) = O(n)$$

which is also optimal. This recursive description can be unfolded to yield a parallel circuit for this computation. This algorithm can be generalized for any arbitrary associative operation and is known as *parallel prefix* or *scan* operation. Using appropriately defined composition function for a semi-adder (adding two bits given a carry), we can construct the carry-save adder circuit.

One of the most fascinating developments is the Quantum Model which is inherently parallel but it is also fundamentally different from the previous models. A breakthrough result in recent years is a polynomial time algorithm for factorization which forms the basis of many cryptographic protocols in the conventional model.

**Biological Computing** models is a very active area of research where scientists
are trying to assemble a machine out of DNA strands. It has potentially many advantages over silicon based devices and is inherently parallel.
Chapter 2

Warm up problems

One of the primary challenges in algorithm design is to come up with provably optimal algorithms. The optimality is with respect to the underlying model. In this chapter, we look closely at some well-known algorithms for basic problems that uses basic properties of the problem domain in conjunction with elementary analytical methods.

2.1 Euclid’s algorithm for GCD

Euclid’s algorithm for computing the greatest common divisor (gcd) of two positive integers is allegedly the earliest known algorithm in a true sense. It is based on two very simple observations that the gcd of numbers $a, b$ satisfies

$$\text{gcd}(a, b) = \text{gcd}(a, a + b)$$

$$\text{gcd}(a, b) = b \text{ if } b \text{ divides } a$$

Exercise 2.1 Prove this rigorously.

The above also implies that $\text{gcd}(a, b) = \text{gcd}(a - b, b)$ for $b < a$ and repeated applications imply that $\text{gcd}(a, b) = \text{gcd}(a \mod b, b)$ where $\mod$ denotes the remainder operation. So we have essentially derived Euclid’s algorithm, described formally as

<table>
<thead>
<tr>
<th>Algorithm Euclid_GCD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Positive integers $a, b$ such that $b \leq a$</td>
</tr>
<tr>
<td><strong>Output:</strong> GCD of $a, b$</td>
</tr>
</tbody>
</table>

  Let $c = a \mod b$.
  If $c = 0$ then return $b$ else
  return Euclid_GCD($b, c$)
Let us now analyze the running time of Euclid’s algorithm in the bit model. Since it depends on integer division, which is a topic in its own right, let us address the number of iterations of Euclid’s algorithm in the worst case.

**Observation 2.1** The number $a \mod b \leq \frac{a}{2}$, i.e. the size of $a \mod b$ is strictly less than $|a|$.

This is a simple case analysis based on $b \leq \frac{a}{2}$ and $b > \frac{a}{2}$. As a consequence of the above observation, it follows that the number of iterations of the Euclid’s algorithm is bounded by $|a|$, or equivalently $O(\log a)$.

**Exercise 2.2** Construct an input for which the number of iterations match this bound.

So, by using the long division method to compute $\mod$, the running time is bounded by $O(n^3)$ where $n = |a| + |b|$.

### 2.1.1 Extended Euclid’s algorithm

If you consider the numbers defined by the linear combinations of $a, b$, namely, $\{xa + yb| x, y \text{ are integers}\}$ it is known that

$$\gcd(a, b) = \min\{xa + yb|x, y \geq 0\}$$

**Proof:** Let $\ell = \min\{xa + yb|x, y \geq 0\}$. Clearly $\gcd(a, b)$ divides $\ell$ and hence $\gcd(a, b) \leq \ell$. We now prove that $\ell$ divides $a$ (also $b$). Let us prove by contradiction that $a = \ell q + r$ where $\ell > r > 0$. Now $r = a - \ell q = (1 - xq)a - (yq)b$ contradicting the minimality of $\ell$. \qed

The above result can be restated as $\ell$ divides $a$ and $b$. For some applications, we are interested in computing $x$ and $y$ corresponding to $\gcd(a, b)$. We can compute them recursively along with the Euclid’s algorithm.

**Exercise 2.3** Let $(x', y')$ correspond to $\gcd(b, a \mod b)$, i.e. $\gcd(b, a \mod b) = x' \cdot b + y' \cdot (a \mod b)$. Then show that $\gcd(a, b) = y' \cdot a + (x' - q)b$ where $q$ is the quotient of the integer division of $a$ by $b$.

One immediate application of the extended Euclid’s algorithm is computing the inverse in a multiplicative prime field $F_q^*$ where $q$ is prime. $F_q^* = \{1, 2 \ldots (q - 1)\}$ where the multiplication is performed modulo $q$. It is known \footnote{since it forms a group} that for every number
There exists $y \in F_q^*$ such that $x \cdot y \equiv 1 \pmod{q}$ which is also called the inverse of $x$. To compute the inverse of $a$ we can use the extended Euclid algorithm to find $s, t$ such that $sa + tq = 1$ since $a$ is relatively prime to $q$. By taking remainder modulo $q$, we see that $s \mod q$ is the required inverse.

**Exercise 2.4** Extend the above result to $\mathbb{Z}_N^*$, i.e., $\{x | x$ is relatively prime to $N\}$. First show that $\mathbb{Z}_N^*$ is closed under multiplication modulo $N$, i.e., $a, b \in \mathbb{Z}_N^* \Rightarrow a \cdot b \mod N \in \mathbb{Z}_N^*$.

### 2.2 Finding the $k$-th element

**Problem** Given a set $S$ of $n$ elements, and an integer $k, 1 \leq k \leq n$, find an element $x \in S$ such that the rank of $x$ is $k$. The rank of an element in a set $S$ is $k$ if $x = x_k$ in the sorted set $x_1, x_2, \ldots, x_n$ where $x_i \in S$. We will denote the rank of $x$ in $S$ by $R(x, S)$.

Note that $k$ is not unique if the value of $x$ is not unique, but the value of the $k$-th element is unique. If $S$ is a multiset, we can (hypothetically) append $\log n$ trailing bits equal to the input index to each element. So an element $x_i$ can be thought of as a pair $(x_i, i)$ so that every pair is unique since the input index is unique. The case $k = 1 \ (k = n)$ corresponds to finding the minimum (maximum) element.

We can easily reduce the selection problem to sorting by first sorting $S$ and then reporting the $k$-th element of the sorted set. But this also implies that we cannot circumvent the lower bound of $\Omega(n \log n)$ for comparison based sorting. If we want a faster algorithm, we cannot afford to sort. For instance, when $k = 1$ or $k = n$, we can easily select the minimum (maximum) element using $n - 1$ comparisons. The basic idea for a faster selection algorithm is based on the following observation.

Given an element $x \in S$, we can answer the following query in $n - 1$ comparisons

Is $x$ the $k$-th element or is $x$ larger than the $k$-th element or is $x$ smaller than the $k$-th element?

This is easily done by comparing $x$ with all elements in $S - \{x\}$ and finding the rank of $x$. Using an arbitrary element $x$ as a filter, we can subsequently confine our search for the $k$-th element to either

(i) $S_> = \{y \in S - \{x\} | y > x\}$ if $R(x, S) < k$ or
(ii) $S_< = \{y \in S - \{x\} | y < x\}$ if $R(x, S) > k$

In the fortuitous situation, $R(x, S) = k$, $x$ is the required element. In case 1, we must find $k'$-th element in $S_>$ where $k' = k - R(x, S)$. 

17
Suppose $T(n)$ is the worst case running time for selecting the $k$-th element for any $k$, then we can write the following recurrence

$$T(n) \leq \max\{T(|S_<|), T(|S>|)\} + O(n)$$

A quick inspection tells us that if we can ensure $\max\{|S_<|, |S>|\} \leq \epsilon n$ for some $1/2 \leq \epsilon < \frac{n-1}{n}$, (Why the bounds?) $T(n)$ is bounded by $O\left(\frac{1}{1-\epsilon} \cdot n\right)$. So it could vary between $\Omega(n)$ and $O(n^2)$ - where a better running time is achieved by ensuring a smaller value of $\epsilon$.

An element $x$ used to divide the set is often called a splitter or a pivot. So, now we will discuss methods to select a good splitter. From our previous discussion, we would like to select a splitter that has a rank in the range $[\epsilon \cdot n, (1 - \epsilon) \cdot n]$ for a fixed fraction $\epsilon$. Typically, $\epsilon$ will be chosen as $1/4$.

### 2.2.1 Choosing a random splitter

Let us analyze the situation where the splitter is chosen uniformly at random from $S$, i.e., any of the $n$ elements is equally likely to be chosen as the splitter. This can be done using standard routines for random number generation in the range $(1, 2, \ldots n)$.

A central observation is

For a randomly chosen element $r \in S$, the probability

$$\Pr\{n/4 \leq R(r, S) \leq 3n/4\} \geq 1/2$$

It is easy to verify if the rank $R(r, S)$ falls in the above range, and if it does not, then we choose another element independently at random. This process is repeated till we find a splitter in the above range - let us call such a splitter a good splitter.

How many times do we need to repeat the process?

To answer this, we have to take a slightly different view. One can argue easily that there is no guarantee that we will terminate after some fixed number of trials, while it is also intuitively clear that it is extremely unlikely that we need to repeat this more than say 10 times. The probability of failing 9 consecutive times, when the success probability of picking a good splitter is $\geq 1/2$ independently is $\leq \frac{1}{2^9}$. More precisely, the expected number of trials is bounded by 2. So, in (expected) two trials, we will find a good splitter that reduces the size of the problem to at most $\frac{3}{4}n$. This argument can be repeated for the recursive calls, namely, the expected number of splitter selection (and verification of its rank) is 2. If $n_i$ is the size of the problem after $i$ recursive calls with $n_0 = n$, then the expected number of comparisons done

\[2\text{Please refer to the Appendix for a quick recap of basic measures of discrete probability}\]
after the $i$-th recursive call is $2n_i$. The total expected number of comparisons $X$ after $t$ calls can be written as $X_0 + X_1 + \ldots X_t$ where $t$ is sufficiently large such that the problem size $n_t \leq C$ for some constant $C$ (you can choose other stopping criteria) and $X_i$ is the number of comparisons done at stage $i$. By taking expectation on both sides

$$E[X] = E[X_1 + X_2 + \ldots X_t] = E[X_1] + E[X_2] + \ldots E[X_t]$$

From the previous discussion $E[X_i] = 2n_i$ and moreover $n_i \leq \frac{3}{4}n_{i-1}$. Therefore the expected number of comparisons is bounded by $4n$.

Let us analyze the original recursive algorithm, where we choose a random splitter and proceed with the relevant subproblem. Let $\bar{T}(n)$ be the expected time for selection of the $k$-th ranked element (for any $k$). Since each element is equally likely to be the splitter, we can do a case analysis based on the rank of the random splitter $x$ compared to $k$.

Case $\text{rank}(x) < k$ Then the subproblem size is $n - \text{rank}(x)$ for each of the $k - 1$ possibilities for $x$.

Case $\text{rank}(x) > k$ The subproblem size is $\text{rank}(x) - 1$ for each of the $n - k - 1$ possibilities for $x$.

As each individual case has probability $\frac{1}{n}$ we can write the recurrence as

$$\bar{T}(n) = \frac{1}{n} \sum_{i=n-1}^{n-1} \bar{T}(i) + \frac{1}{n} \sum_{j=n-1}^{n-1} \bar{T}(j) + O(n)$$

**Exercise 2.5** Verify that this recurrence has the worst case behavior for $k = n/2$ assuming that $\bar{T}(i)$ grows monotonically with $i$.

Then we have to find the solution of

$$\bar{T}(n) = \frac{2}{n} \sum_{i=n/2}^{n-1} \bar{T}(i) + c'n$$

which can be verified as $\bar{T}(n) = cn$ for $c > 4c'$ by induction as follows

$$\bar{T}(n) = \frac{2}{n} \sum_{i=n/2}^{n-1} [c(n/2 + i)] + c'n$$

$$\leq c'[\frac{n}{2} + \frac{n}{4}] + cn/4 = cn$$

(2.2.1)
2.2.2 Median of medians

Partition the elements into groups of 5 and choose the median of each group. Denote the groups by $G_i$ and their medians by $m_i$. Now consider the median of the set $\{m_i\}$ which contains about $n/5$ elements. Denote the median of medians by $M$.

How many elements are guaranteed to be smaller than $M$?

Wlog, assume all elements are distinct and that implies about $n/10^3$ medians that are smaller than $M$. For each such median there are 3 elements that are smaller than $M$, giving a total of at least $n/10 \cdot 3 = 3n/10$ elements smaller than $M$. Likewise, we can argue that there are at least $3n/10$ elements larger than $M$. Therefore we can conclude that $3n/10 \leq R(M, S) \leq 7n/10$ which satisfies the requirement of a good splitter. The next question is how to find $M$ which is the median of medians. Each $m_i$ can be determined in $O(1)$ time because we are dealing with groups of size 5. However, finding the median of $n/5$ elements is like going back to square one! But it is $n/5$ elements instead of $n$ and therefore, we can apply a recursive strategy. We can write a recurrence for running time as follows

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n)$$

where the second recursive call is to find the median of medians (for finding a good splitter). After we find the splitter (by recursively applying the same algorithm), we use it to reduce the original problem size to at most $\frac{7n}{10}$.

**Exercise 2.6** By using an appropriate terminating condition, show that $T(n) \in O(n)$. Try to minimize the leading constant by adjusting the size of the group.

2.3 Sorting words

**Problem** Given $n$ words $w_1, w_2 \ldots w_n$ of lengths $l_1, l_2 \ldots l_n$ respectively, arrange the words in a lexicographic order. A word is an ordered sequence of characters from a given alphabet $\Sigma$.

Recall that lexicographic ordering refers to the dictionary ordering. Let $N = \sum_i l_i$, i.e. the cumulative length of all words. A single word may be very long and we cannot assume that it fits into a single word of the computer. So, we cannot use straightforward comparison sorting. Let us recall some basic results about integer sorting.

**Claim 2.1** $n$ integers in the range $[1..m]$ can be sorted in $O(n + m)$ steps.

---

3Strictly speaking we should be using the floor function but we are avoiding the extra symbols and it does not affect the analysis.
A sorting algorithm is considered \textit{stable} if the relative order of input elements having identical values is preserved in the sorted output.

\textbf{Claim 2.2} Using stable sorting, \(n\) integers in the range \([1..m^k]\) can be sorted in \(O(k(n + m))\) steps.

The above is easily achieved by applying integer sorting in the range \([1..m]\) starting from the least significant digits - note that the maximum number of digits in radix \(m\) representation is \(k\). If we apply the same algorithm for sorting words, then the running time will be \(O(L(n + |\Sigma|))\) where \(L = \max\{l_1, l_2..l_n\}\). This is not satisfactory since \(L \cdot n\) can be much larger than \(N\) (size of input).

The reason that the above method is potentially inefficient is that many words may be much shorter than \(L\) and hence by considering them to be length \(L\) words (by hypothetical trailing blanks), we are increasing the input size asymptotically. When we considered radix sort as a possible solution, the words have to be left-aligned, i.e., all words begin from the same position. To make radix sort efficient and to avoid redundant comparison (of blanks), we should not consider a word until the radix sort reaches the right boundary of the word. The radix sort will take a maximum of \(L\) rounds and a word of length \(l\) will start participating from the \(L - l + 1\) iteration. This can be easily achieved. A bigger challenge is to reduce the range of sorting in each iteration depending on which symbols of the alphabet participate.

Given a word \(w_i = a_{i,1}a_{i,2}...a_{i,l_i}\), where \(a_{i,j} \in \Sigma\), we form the following pairs - \((1, a_{i,1}), (2, a_{i,2})\ldots\). There are \(N\) such pairs from the \(n\) words and we can think of them as length two strings where the first symbol is from the range \([1..L]\) and the second symbol is from \(\Sigma\). We can sort them using radix sort in two rounds in \(O(N + L + |\Sigma|)\) which is \(O(N + |\Sigma|)\) since \(N > L\). From the sorted pairs we know exactly which symbols appear in a given position (between 1 and \(L\)) - let there be \(m_i\) words that have non-blank symbols in position \(i\). We also have the ordering of symbols in position \(i\) which is crucial to implement integer sort in \(O(m_i)\) steps.

Now we go back to sorting the given words using radix sort where we will use the information available from the sorted pairs. When we are sorting position \(i\) from the left, we apply integer sort in the range \([1..m_i]\) where the ordered buckets are also defined by the sorted pairs. We do not move the entire words into the buckets but only the pointers (which could be the input index of the words) associated with the words. For every round, we allocate an array of size \(m_i\) where we place the pointers according to the sorted order of the symbols involved. For same symbols, we maintain the order of the previous round (stable sorting). We must also take care of the new words that start participating in the radix sort - once a word participates, it will participate in all future rounds. (Where should the new words be placed within its symbol group?)
The analysis of this algorithm can be done by looking at the cost of each radix sort which is proportional to \( \sum_{i=1}^{L} O(m_i) \) which can be bounded by \( N \). Therefore overall running time of the algorithm is the sum of sorting the pairs and the radix sort. This is given by \( O(N + |\Sigma|) \). If \( |\Sigma| < N \), then the optimal running time is given by \( O(N) \).

**Exercise 2.7** Work out the details of sorting \( n \) binary strings in \( O(N) \) steps where \( N = \sum_{i} \ell_i \), \( \ell_i \) is the number of bits in the \( i \)-th string.

### 2.4 Mergeable heaps

Heaps\(^4\) are one of the most common implementation of priority queues and are known to support the operations \textit{min}, \textit{delete-min}, \textit{insert}, and \textit{delete} in logarithmic time. A complete binary tree (often implemented as an array) is one of the simplest ways to represent a heap. In many situations, we are interested in an additional operation, namely, combining two heaps into a single heap. A binary tree doesn’t support fast (polylogarithmic) merging and is not suitable for this purpose - instead we use binomial trees.

A binomial tree \( B_i \) of order \( i \) is recursively defined as follows

- \( B_0 \) is a single node
- For \( i \geq 0 \), \( B_{i+1} \) is constructed from two \( B_i \)'s by making the root node of one \( B_i \) a left child of the other \( B_i \).

**Exercise 2.8** Prove the following properties for \( B_i \) by induction

(i) The number of nodes in \( B_i \) equals \( 2^i \).
(ii) The height of \( B_k \) is \( k \) (by definition \( B_0 \) has height 0).
(iii) There are exactly \( \binom{i}{k} \) nodes at depth \( k \) for \( k = 0, 1 \ldots \)
(iv) The children of \( B_i \) are roots of \( B_{i-1}, B_{i-2} \ldots B_0 \).

A **binomial heap** is an ordered set of binomial trees such that for any \( i \) there is at most one \( B_i \).

Let us refer to the above property as the \textit{unique-order} property. We actually maintain list of the root nodes in increasing order of their degrees.

You may think of the above property as a binary representation of a number where the \( i \)-th bit from right is 0 or 1 and in the latter case, its contribution is \( 2^i \) (for LSB \( i = 0 \)). From the above analogue, a Binomial Heap on \( n \) elements has \( \log n \) Binomial trees. Therefore, finding the minimum element can be done in \( O(\log n) \) comparisons by finding the minimum of the \( \log n \) roots.

\(^4\)we are assuming min heaps
2.4.1 Merging Binomial Heaps

Merging two Binomial Heaps amounts to merging the root lists and restoring the unique-order property. First we merge two lists of size at most $\log n$. Subsequently, we walk along the list combining two trees of the same degree whenever we find them - they must be consecutive. We know that by combining two $B_i$ trees, we obtain a $B_{i+1}$ tree which may have to be now combined with the next $B_{i+1}$ tree if there exists one. In this process, if you have three consecutive Binomial trees of order $i$ (can it happen?), merge the second and third instead the first and second - it simplifies the procedure. Combining two binomial trees takes $O(1)$ time, so the running time is proportional to the number of times we combine.

Claim 2.3 Two Binomial heaps can be combined in $O(\log n)$ steps where the total number of nodes in the two trees is $n$.

Every time we combine two trees, the number of binomial trees decreases by one, so there can be at most $2 \log n$ times where we combine trees.

Remark The reader may compare this with the method for summing two numbers in binary representation.

Exercise 2.9 Show that the delete-min operation can be implemented in $O(\log n)$ steps using merging.

Inserting a new element is easy - add a node to the root list and merge. Deletion takes a little thought. Let us first consider an operation decrease-key. This happens when a key value of a node $x$ decreases. Clearly, the min-heap property of the parent node, $\text{parent}(x)$ may not hold. But this can be restored by exchanging the node $x$ with its parent. This operation may have to be repeated at the parent node. This continues until the value of $x$ is greater than its current parent or $x$ doesn’t have a parent, i.e., it is the root node. The cost is the height of a Binomial tree which is $O(\log n)$.

Exercise 2.10 Show how to implement the delete operation in $O(\log n)$ comparisons.

2.5 A simple semi-dynamic dictionary

Balanced binary search trees like AVL trees, Red-black trees etc. support both search and updates in worst case $O(\log n)$ comparisons for $n$ keys. These trees inherently use dynamic structures like pointers which actually slow down memory access. Arrays are inherently superior since it supports direct memory access but are not amenable to inserts and deletes.
Consider the following scheme for storing \( n \) elements in multiple arrays \( A_0, A_1, \ldots, A_k \) such that \( A_i \) has length \( 2^i \). Each \( A_i \), that exists contains \( 2^i \) elements in sorted order - there is no ordering between different arrays. Only those \( A_i \) exists for which the \( i \)-th bit \( b_i \) in the binary representation of \( n \) is non-zero (recall that this representation is unique). Therefore \( \sum_i b_i \cdot |A_i| = n \) and maximum number of occupied arrays is \( \log n \).

For searching we do binary search in all the arrays that takes \( O(\log^2 n) \) steps \( (O(\log n) \) steps for each array). To insert, we compare the binary representations of \( n \) and \( n+1 \). There is a unique smallest suffix (of the binary representation of \( n \)) that changes from 11..1 to 100..0, i.e., \( n \) is \( w011...1 \) and \( n+1 \) is \( w100..0 \). Consequently all the elements of those \( A_i \) for which \( i \)-th bit becomes 0 is merged into an array that corresponds to the bit that becomes 1 (and is also large enough to hold all elements including the new inserted element).

**Exercise 2.11** How would you implement the merging in \( O(2^j) \) steps for merging where the \( j \)-th bit becomes 1 in \( n+1 \) ?

Clearly this could be much larger than \( O(\log n) \), but notice that \( A_j \) will continue to exist for the next \( 2^j \) insertions and therefore the averaging over the total number of insertions gives us a reasonable cost. As an illustration consider a binary counter and let us associate the cost of incrementing the counter as the number of bits that undergo changes. Observe that at most \( \log n \) bits change during a single increment but mostly it is much less. Overall, as the counter is incremented from 0 to \( n - 1 \), bit \( b_i \) changes at most \( n/2^i \) times, \( 1 \leq i \). So roughly there are \( O(n) \) bits that change implying \( O(1) \) changes on the average.

In the case of analysing insertion in arrays, by analogy, the total number of operations needed to carry out the sequence of merging that terminates at \( A_j \) is \( \sum_{s=1}^{j-1} O(2^s) \) which is \( O(2^j) \). Therefore the total number of operations over the course of inserting \( n \) elements can be bounded by \( \sum_{j=1}^{\log n} O(n/2^j \cdot 2^j) \) which is \( O(n \log n) \). In other words, the average cost of insertion is \( O(\log n) \) that matches the tree-based schemes.

To extend this analysis more formally, we introduce the notion of potential based amortized analysis.

### 2.5.1 Potential method and amortized analysis

To accurately analyse the performance of an algorithm, let us denote by \( \Phi(i) \) as a function that captures the state of an algorithm or its associated data structure at any stage \( i \). We define amortized work done at step \( i \) of an algorithm as \( w_i + \Delta_i \), where \( w_i \) is actual number of steps\(^5 \Delta_i = \Phi(i) - \Phi(i - 1) \) which is referred to as the

\(^5\)this may be hard to analyze.
difference in potential. Note that the total work done by an algorithm over $t$ steps is $W = \sum_{i=1}^{t} w_i$. On the other hand, the total amortized work is

$$\sum_{i=1}^{t} (w_i + \Delta_i) = W + \Phi(t) - \Phi(0)$$

If $\Phi(t) - \Phi(0) \geq 0$, amortized work is an upperbound on the actual work.

**Example 2.1** For the counter problem, we define the potential function of the counter as the number of 1’s of the present value. Then the amortised cost for a sequence of 1’s changing to 0 is 0 plus the cost of a 0 changing to 1 resulting in $O(1)$ amortised cost.

**Example 2.2** A stack supports push, pop and empty-stack operations. Define $\Phi()$ as the number of elements in the stack. If we begin from an empty stack, $\Phi(0) = 0$. For a sequence of push, pop and empty-stack operations, we can analyze the amortized cost. Amortized cost of push is 2, for pop it is 0 and for empty stack it is negative. Therefore the bound on amortized cost is $O(1)$ and therefore the cost of $n$ operations is $O(n)$. Note that the worst-case cost of an empty-stack operation can be very high.

**Exercise 2.12** Can you define an appropriate potential function for the search data-structure analysis?
Chapter 3

Optimization I: Brute force and Greedy strategy

Optimization problems are used to model many real life problems. Therefore, solving these problems is one of the most important goals of algorithm design. A general optimization problem can be defined by specifying a set of constraints that defines a subset in some underlying space (like the Euclidean space $\mathbb{R}^n$) called the feasible subset and an objective function that we are trying to maximize or minimize as the case may be over the feasible set. The difficulty of solving such problems typically depends on how “complex” the feasible set and the objective function are. For example, a very important class of optimization problems is “Linear Programming”. Here the feasible subset is specified by a set of linear inequalities (in the Euclidean space), and the objective function is also linear. A more general class of optimization problems is “convex programming”, where the feasible set is a convex subset of a Euclidean space and the objective function is also convex. Convex programs (and hence, linear programs) have a nice property that any local optimum is also a global optimum for the objective function. There are a variety of techniques for solving such problems – all of them try to approach a local optimum (which we know would be a global optimum as well). The more general problem, the so-called “non-convex programs”, where the objective function and the feasible subset could be arbitrary can be very challenging to solve. In particular, “discrete optimization problems”, where the feasible subset could be a (large) discrete subset of points falls under this category.

In this chapter, we first discuss some of the most intuitive approaches for solving such problems. We begin with heuristic search approaches, which try to search for an optimal solution by exploring the feasible subset in some principled manner. Subse-

\[ f(\lambda \cdot x + (1-\lambda) \cdot y) \leq \lambda f(x) + (1-\lambda) f(y). \]
sequently, we introduce the idea of designing algorithms based on the greedy heuristic.

### 3.1 Heuristic search approaches

In heuristic search, we explore the search space in a structured manner. Observe that in general, the size of the feasible set (also called the set of feasible solutions) can be infinite. Even if we consider some discrete approximations to the feasible set (or if the feasible set itself is discrete), the set of feasible solutions can be exponentially large. In such settings, we cannot hope to look at every point in the feasible set. Heuristic search approaches circumvent this problem by pruning out parts of the search space where we are sure that the optimal solution does not lie. These approaches are widely used in practice, and are often considered a general purpose technique for many difficult optimization problems.

We illustrate the ideas behind this technique by considering the 0-1 knapsack problem. The 0-1 Knapsack problem is defined as follows. The input consists of a parameter $C$, which is the capacity of a knapsack, and $n$ objects of volumes $\{w_1, w_2 \ldots w_n\}$ and profits $\{p_1, p_2 \ldots p_n\}$. The objective is to choose a subset of these $n$ objects that fits into the knapsack (i.e., the total volume of these objects should be at most $C$) such that the total profit of these objects is maximized.

We can frame this problem as a discrete optimization problem. For each object $i$, we define a variable $x_i$, which could be either 0 or 1. It should be 1 if the solution selects object $i$ in the knapsack, 0 otherwise. Note that the feasible subset in this optimization problem is the discrete set $\{0, 1\}^n$. The knapsack problem can now be formally stated as

$$\text{Maximize } \sum_{i=0}^{n} x_i \cdot p_i \text{ subject to } \sum_{i=0}^{n} x_i \cdot w_i \leq C, \text{ and } (x_1, \ldots, x_n) \in \{0, 1\}^n$$

Note that the constraint $x_i \in \{0, 1\}$ is not linear, otherwise we could use linear programming\(^2\). A simplistic approach to solving this problem would be to enumerate all subsets of the $n$ objects, and select the one that satisfies the constraints and maximizes the profits. Any solution that satisfies the knapsack capacity constraint is called a feasible solution. The obvious problem with this strategy is the running time which is at least $2^n$ corresponding to the power-set of $n$ objects. Instead of thinking of the search space as the set of all subsets of objects, we now think of it

\(^2\)If we allow the variables $x_i$ to take real values between 0 and 1, the so-called fractional relaxation of the knapsack problem, then this becomes a linear program. It can be solved efficiently, and one can show that there is an optimal solution where all the $x_i$ values, except perhaps for one object, are either 0 or 1. This allows one to get a solution whose profit is at least half of the optimal value.
in a more structured manner. We can imagine that the solution space is generated by a binary tree where we start from the root with an empty set and then move left or right according to selecting the first object (i.e., value of the variable $x_1$). At the second level, we again associate the left and right branches with the choice of $x_2$. Thus each node in the tree corresponds to a partial solution – if it is at depth $j$ from the root, then the values of variables $x_1, \ldots, x_j$ are known at $j$. In this way, the $2^n$ leaf nodes correspond to each possible subset of the power-set which corresponds to a $n$ length 0-1 vector. For example, a vector $000\ldots01$ corresponds to the subset that only contains only the object $n$.

Thus, the simplistic approach just means that we look at every leaf node in this tree, and see whether the objects chosen by the solution at the leaf node fit in the knapsack. Among all such leaves, we pick the best solution. This is just a restatement of the brute force strategy of looking at all possible $2^n$ different solutions. However, we can devise clever ways of reducing the search space. For example, suppose we traverse the tree (in top-down manner), and reach a node $v$. This node corresponds to a partial solution, and suppose the objects which have been picked in this partial solution have total weight more than the knapsack size. At this moment, we know that there is no use exploring the sub-tree below $v$, because the partial solution corresponding to $v$ itself does not fit in the knapsack.

One can devise more intricate strategies for pruning the search space. A very high level idea would be following. We maintain a parameter $T$, which denotes the profit of the best solution we have seen so far while traversing the tree. For each node $v$, let $S(v)$ denote the set of leaves in the sub-tree rooted at $v$. Note that the leaves in $S(v)$ correspond to all possible ways of extending the partial solution at node $v$. For each node $v$ in the tree, we maintain two values, $L(v)$ and $U(v)$, which are supposed to be lower and upper bounds on the best solution among all leaves in $S(v)$. When our algorithm reaches a node $v$, and if it so happens that $T > U(v)$, it need not explore the sub-tree below $v$ at all. Note that the bounds $L(v)$ and $U(v)$ need not be fixed – the algorithm could also update them as it proceeds.

For example, we could keep $L(v)$ as the profit of the partial solution corresponding to $v$ if it fits in the knapsack, and $U(v)$ to be some very high value. If the partial solution at $v$ does not fit in the knapsack, we can make $U(v)$ to 0 – this is the same as the pruning strategy described above. A more sophisticated pruning strategy is as follows.

Consider a node $v$ at level $j$ in the tree. Thus, $v$ corresponds to a partial solution, where we have decided which objects among $1, 2, \ldots, j$ to pick. If this partial solution does not fit in the knapsack, we set $U(v)$ to 0 (it does not matter what $L(v)$ is). Now suppose the partial solution fits in the knapsack and occupies weight $W(v)$ and has profit $P(v)$. We set $L(v)$ to $P(v)$. For $U(v)$, let $\rho$ denote the maximum density of an object among $j + 1, \ldots, n$, where the density of an object is the ratio of its profit.
to its weight. We observe that \( U(v) \) can be set to \( P(v) + (C - W(v)) \cdot \rho \). Indeed, the objects chosen in the partial solution for \( v \) already occupy \( W(v) \) space, and so, we can only add \( C - W(v) \) more weight to it. Any such object added after \( v \) would contribute at most \( \rho \) units of profit per unit weight.

**Example 3.1** Let the capacity of the knapsack be 15 and the weights and profits are respectively

<table>
<thead>
<tr>
<th>Profits</th>
<th>10</th>
<th>10</th>
<th>12</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

We will use the strategy described above for setting \( L(v) \) and \( U(v) \). Observe that the densities of the objects are 5, 2.5, 2 and 2 respectively. Initially, our estimate \( T = 0 \). For the root node \( v \), \( L(v) \) is 0, and \( U(v) \) is \( 5 \times 15 = 75 \). Consider the left child of the root node which corresponds to including the first object. For this node, call it \( w \), the remaining capacity of the knapsack is 13, and \( T \) becomes 10 (because there is a solution of value 10 – just take this object). By proceeding this way, we obtain \( T = 38 \) for the set of objects \{1, 2, 4\}. By exploring further, we come to a stage when we have included object 1 and decided against including object 2. Call this node \( u \). So \( U(u) = 10 \), and residual capacity is 13. Should we explore the subtree regarding \{3, 4\}? Since the densities of both these objects are 2, we get \( U(u) = 2 \times 13 + 10 = 36 < T = 38 \). So we need not search this subtree. By continuing in this fashion, we may be able to prune large portions of the search tree. However, it is not possible to obtain any provable improvements.

This method of pruning search is called *branch and bound* and although it is clear that there it is advantageous to use the strategy, there may not be any provable savings in the worst case.

### 3.1.1 Game Trees *

Game trees represent a game between two players who alternately make moves trying to win the game. For example, consider the game "tic-tac-toe". This game is played on a 3 × 3 board with 9 squares. Let us call the two players \( A \) and \( B \). Initially all 9 squares are empty. The two players make move alternately – player \( A \) writes the symbol × one of the empty squares, whereas the player \( B \) writes the symbol ⃝ on one of the empty squares. The player who first gets 3 of her symbols along a straight line in the board (diagonal, vertical or horizontal) wins.

The set of all strategies in this game can be represented by a giant tree, where each node in the tree corresponds to a configuration of the board. A configuration of the board is labeling of the squares on the board which can be realized during the game. Note that certain labelings of the board are not configurations. For example,
suppose we label 4 of the squares as $\times$ and 1 square as $\Box$. We can never reach such a labeling because players take turns. In this tree, the root node corresponds to the configuration where all squares are empty. Further, player $A$ makes a move – she has 9 choices. Therefore, the root node has 9 children, each corresponding to writing the $\times$ symbol on one of the 9 squares. Consider a child $v$ of the root (which has exactly one $\times$ symbol). At $v$, it is $B$’s turn to move. Player $B$ has 8 choices, and so this node will have 8 children. As we can see, this tree has 9 levels, with odd levels (we denote the top level consisting of the root as level 1) corresponding to player $A$’s turn and even levels for player $B$. Further, a node at which we have 3 symbols of the same kind lying on a straight line corresponds to a win situation for one of the players. Such a node will be a leaf node in the tree. Similarly, a node where all the squares have been labeled will be a leaf node (and may correspond to a scenario where no one wins).

For sake of simplicity, let us consider a two-player game where one of the player always wins when the game stops (at a leaf node). For reasons which will become clear soon, we shall call the nodes corresponding to player $A$’s turn (i.e., those at odd levels) as “OR” nodes, and similarly the nodes at even levels as “AND” nodes. Such a tree is called an $AND-OR$ tree. Let 1 represent a win for player $A$ and 0 represent a loss for player $A$. The numbers are flipped for player $B$. The leaf nodes correspond to the final state of the game and are labelled 1 or 0 corresponding to win or loss for player $A$. We describe a rule for labeling each internal node of the tree (in a bottom-up fashion). An OR node has value 1 if one of the children is 1, and 0 otherwise, and so it is like the boolean function $OR$ . An AND node behaves like the boolean AND function - it is 0 if one of the children is 0, and 1 if all of the children are 1. The interpretation of this 0–1 assignment to nodes is as follows. A label of ”1” at a nodes indicates the fact that there is a winning strategy for player $A$ irrespective of how player $B$ plays; whereas a label of “0” indicates that no matter how $A$ plays, there is always a winning strategy for player $B$. The player $A$ at the root can choose any of the branches that leads to a win. However at the next level, he is at the mercy of the player $B$ – only when all branches for $B$ leads to a win for $A$, player $A$ has a winning strategy, otherwise the player $B$ can inflict a loss.

For concreteness, we will consider game trees where each internal node has two children. So the evaluation of this game tree works as follows. Each leaf node is labelled 0 or 1 and an internal node as AND or OR – these will compute the boolean function of the value of the two child nodes. The value of the game tree is the value available at the root node. This value indicates which player has a winning strategy – note that one of the two players will always have a winning strategy. Consider a game tree of depth $2k$ – it has $2^k = 4^k$ nodes. Thus, it seems that it will take about $O(4^k)$ time to evaluate such a game tree. We now show that with clever use of randomness, one can reduce this time to $O(3^k)$.
The idea can be explained for the special case of a single level AND tree. Suppose we are evaluating the AND node at the root, and assume that it evaluates to 0. Therefore, at least one of the two leaf children happen to be 0. If we happened to look at this child before the other one, we need not evaluate the other child. A priori it is difficult to tell which of the two leaf children is 0 – but if we pick a child randomly, then the expected number of lookups is only $0.5 \times 2 + 0.5 \times 1 = 1.5$, a saving of $4/3$ factor over the naive strategy of probing both the children (and paying for 2 evaluations). For the other case when the AND node evaluates to 1, there is no saving by this strategy. We still have to probe both the children. However any interesting game tree will have at least two levels, one AND and the other OR. Then you can see that for an AND node to be 1, both the child OR nodes must be 1. Now for there OR nodes, we can use the above strategy to save the number of probes. In essence, we are applying the branch-and-bound method to this problem, and we obtain a provable improvement by evaluating the two children at a node in a random order.

Now consider the general case of a tree with depth $2k$ (i.e. $4^k$ leaf nodes) with alternating AND and OR nodes, each type having $k$ levels. We will show that the expected cost of evaluation is $3^k$ by induction on $k$. The base case (for $k = 1$) is left to the reader. Assume that the statement is true for trees of depth $2(k−1)$.

Now consider such a tree of depth $2k$. There are two cases depending on whether the root is labeled OR or AND. Let us consider the case where the root has label OR and hence, its two children, say $y$ and $z$, are labeled AND. The children of $y$ and $z$ are OR nodes with $2(k−1)$ depth. We have the two cases

(i) **The root evaluates to 0:** Since the root is an OR node, both $y$ and $z$ must evaluate to 0. Since these are AND nodes, it must be the case that at least one child of $y$ is 0 (and similarly for $z$). It now follows from the argument above that with probability 1/2, we will end up evaluating only one of the children of $y$ (and similarly for $z$). Using the induction hypothesis for the children of $y$ and $z$, we see that the expected number of evaluations for the sub-tree below $y$ is $1/2 \cdot 3^{k−1} + 1/2 \cdot 2 \cdot 4^{k−1} = 3^k/2$. We get the same expression for the expected number of evaluations below $z$, and therefore, the total expected number of evaluations is $3^k$.

(ii) **The root evaluates to 1:** At least one of the AND nodes $y, z$ must be 1. Assume without loss of generality that the node $y$ evaluates to 1. With probability 1/2, we will probe $y$ first, and then we need not look at $z$. To evaluate $y$, we will have to look at both the children of $y$, which are at depth $2(k−1)$. Applying induction hypothesis on children of $y$, we see that the expected number of evaluations for the sub-tree below $y$ is $2 \cdot 3^{k−1}$. We get the same expression for the sub-tree below $z$. Therefore the expected number of evaluations
is $\frac{1}{2} \cdot 2 \cdot 3^{k-1} + \frac{1}{2} \cdot 4 \cdot 3^{k-1} = 3^k$, where the first term corresponds to the event that we pick $y$ first (and so do not evaluate $z$ at all), and the second term corresponds to the event that we pick $z$ first, and so may evaluate both $y$ and $z$.

In summary, for an OR root node, regardless of the output, the expected number of evaluations is bounded by $3^k$. We can express this in terms of the total number of leaves. Note that if $N$ denotes the number of leaves, then $N = 4^k$, and so, the expected number of evaluations is $N^{\log_4 3} = N^\alpha$ where $\alpha < 0.8$. The case when the root is an AND node is left as an exercise.

### 3.2 A framework for Greedy Algorithms

There are very few algorithmic techniques for which the underlying theory is as precise and clean as what we will discuss here. Let us define the framework. Let $S$ be a set and $M$ be a subset of $2^S$. Then $(S, M)$ is called a **subset system** if it satisfies the following property

For all subsets $T \in M$, for any $T' \subset T$, $T' \in M$.

Note that the empty subset $\emptyset \in M$. The family of subsets $M$ is often referred to as independent subsets and one may think of $M$ as the feasible subsets.

**Example 3.2** Let $G = (V, E)$ be an undirected graph, and consider the subset system $(E, M)$, where $M$ consists of all subsets of $E$ which form a forest (recall that a set of edges form a forest if they do not induce a cycle). It is easy to see that this satisfies the property for a subset system.

Given a subset system, we can define a natural optimization problem as follows. For any weight function $w : S \to \mathbb{R}^+$, we would like to find a subset from $M$ for which the cumulative weight of the elements is maximum among all choices of subsets from $M$. We call such a subset an **optimal** subset. Note that this is a non-trivial problem because the size of $M$ could be exponential in $S$, and we may only have an implicit description of $M$ (as in the example above\(^3\)). In such a case, we cannot look at every subset in $M$ and evaluate the total weight of elements in it. A simple way to find such a subset is the following greedy approach.

---

\(^3\) $M$ is a family of subsets of $S$

\(^4\) The number of spanning trees of a complete graph is $n^{n-2}$
Procedure Greedy($S, M$)

1. **Input** $S = \{e_1, e_2 \ldots e_n\}$ in decreasing order of weights;
2. $T = \phi$.
3. **for** $i = 1$ to $n$ **do**
   
   if $T \cup \{e_i\} \in M$ then
   
   $T \leftarrow T \cup \{e_i\}$

4. **Output** $T$ as the solution

Figure 3.1: Algorithm Gen_Greedy

The running time of the algorithm is dependent mainly on the test for independence which depends on the specific problem. Even if $M$ is not given explicitly, we assume that an implicit characterization of $M$ can be used to perform the test. In the example of forests in a graph, we just need to check if the set $T$ contains a cycle or not.

What seems more important is the following question – Is $T$ the maximum weight subset in $M$? This is answered by the following result.

**Theorem 3.1** The following are equivalent

1. Algorithm Gen_Greedy outputs the optimal subset for any choice of the weight function.

2. **exchange property**

   For any pair of subsets $S_1, S_2 \in M$ where $|S_1| < |S_2|$, there exists an element $e \in S_2 - S_1$ such that $S_1 \cup \{e\} \in M$.

3. **rank property**

   For any $A \subset S$, all maximal independent subsets of $A$ have the same cardinality (we say that a subset $T$ of $A$ is a maximal independent subset if $T \in M$, but $T \cup \{e\} \notin M$ for any $e \in A - T$). This is also called the rank of the subset system.

A subset system satisfying any of the three conditions above is called a matroid. The theorem is used to establish properties 2 or 3 to justify that a greedy approach works for the problem. On the contrary, we can try to prove that one of the properties doesn’t hold (by a suitable counterexample), and then the greedy approach cannot always return an optimal subset.

**Proof:** We will prove it by the following cyclic implications – Property 1 implies Property 2. Then Property 2 implies Property 3 and finally Property 3 implies Property 1.
Property 1 implies Property 2 We prove the contrapositive. Suppose Property 2 does not hold for some subsets $S_1$ and $S_2$. That is, we cannot add any element from $S_2 - S_1$ to $S_1$ and keep it independent. We will show that Property 1 does not hold. Let $p$ denote $S_1$ (and hence, $|S_2| \geq p + 1$). We now define a weight function on the elements of $S$ such that the greedy algorithm fails to output an optimal subset. We define the weight function on the elements of $S$ as follows:

$$w(e) = \begin{cases} 
  p + 2 & \text{if } e \in S_1 \\
  p + 1 & \text{if } e \in S_2 - S_1 \\
  0 & \text{otherwise}
\end{cases}$$

The greedy algorithm will pick up all elements from $S_1$ and then it won’t be able to choose any element from $S_2 - S_1$. Therefore, the solution given by the greedy algorithm has weight $(p + 2)|S_1| = (p + 2) \cdot p$. Now consider the solution consisting of elements of $S_2$. The total weight of elements in $S_2$ is $(p + 1)|S_2 - S_1| + (p + 2)|S_1| > (p + 2) \cdot p$. Thus, the greedy algorithm does not output an optimal subset, i.e., Property 1 does not hold.

Property 2 implies Property 3 Let $S_1$ and $S_2$ be two maximal independent subsets of $A$, and suppose, for the sake of contradiction, that $|S_1| < |S_2|$. Then Property 2 implies that we can add an element $e \in S_2 - S_1$ to $S_1$ and keep it independent. But this contradicts the assumption that $S_1$ is maximal. Therefore, the two sets must have the same size.

Property 3 implies Property 1 Again we will prove the contrapositive. Suppose Property 1 does not hold, i.e., there is a choice of weights $w(e)$ such that the greedy algorithm does not output an optimal subset. Let $e_1, e_2, \ldots, e_n$ be the edges chosen by the greedy algorithm in decreasing order of their weights. Call this set $E_1$. Further, let $e'_1, e'_2, \ldots, e'_m$ be the edges of an optimal solution in decreasing order weights – call this set $E_2$. First observe that the solution $E_1$ is maximal – indeed, if we can add an element $e$ to the greedy solution and keep it independent, then the greedy algorithm should have added $e$ to the set $T$ (as described in Procedure Greedy).

First assume that $m > n$. Let $E'_2$ be a maximal independent subset containing $E_2$. We can always find such a subset by repeatedly adding elements to $E_2$ till we can no longer add any more without violating independence. Now, both $E_1$ and $E'_2$ are maximal, but their sizes are different. So, we see that Property 3 does not hold.

Now assume $m \leq n$. Since the weight of the greedy solution is not maximum, there must a $j \leq m$ such that $w(e_j) < w(e'_j)$. Otherwise the fact that $m \leq n$ implies that weight of $E_1$ is at least that of $E_2$. Let $A = \{e \in S | w(e) \geq w(e'_j)\}$ be the set of elements whose weight is at least $w(e'_j)$. The subset $\{e_1, e_2, \ldots, e_{j-1}\}$ is maximal with respect to $A$ (Why?). All the elements in $\{e'_1, e'_2, \ldots, e'_j\}$ form an independent subset of $A$ that has greater cardinality. This shows that Property 3 does not hold. □
The class of matroid set systems is very rich. We describe some well-known examples of matroids and the corresponding maximum weight independent set problem in this section.

Example 3.3 Half Matching Problem Given a directed graph with non-negative edge weights, we would like to find out the maximum weighted subset of edges such that the in-degree of any node is at most 1. Let us see how to phrase this as a maximum weight independent set problem in a matroid.

The definition of the subset system should be clear – the set $S$ is the set of edges in the directed graph and $M$ is the family of all subsets of edges $E'$ such that no vertex has in-degree more than 1 in the sub-graph induced by $E'$. Let us now show that this subset system is a matroid. We prove Property 2. Considering two subsets $S_p$ and $S_{p+1}$ with $p$ and $p + 1$ edges respectively. Let $V_p$ be the set of vertices which form the head of the edges in $S_p$, i.e., $V_p = \{ u : \exists e = (v, u) \in S + p \}$. By the definition of an independent set, note that $|V_p| = |E_p|$. Define $V_{p+1}$ similarly. Since $|V_{p+1}| > |V_p|$, there is a vertex $u \in V_{p+1} - V_p$. Consider the edge $e$ in $E_{p+1}$ whose head is $u$. Clearly $e \notin E_p$ and adding $e$ to $E_p$ will preserve the independence of this set. Therefore, this subset system is a matroid.

Example 3.4 Maximum Weight Bipartite Matching We now give an example of an important subset system which is not a matroid. Let $G$ be a bipartite graph where edges have weights. A matching in $G$ is a subset of edges which do not share a common vertex. The maximum weight matching problem seeks to find a matching for which the total weight of the edges in it is maximum. As before, we can define a subset system corresponding to matchings in $G$. We define a subset system $(S, M)$, where $S$ is the set of edges in $G$ and $M$ consists of all subsets of edges which form a matching. However, this subset system is not a matroid.

Figure 3.2: The matching $(a, d)$ is a maximal independent set, but $(a, b), (c, d)$ is a larger maximal independent set.
To see why, consider a simple bipartite “zig-zag” graph (shown in Figure 3.4). There are two maximal independent sets here – one with cardinality 2 and the other having only 1 edge. Therefore, Property 3 is violated. In fact, algorithms for finding maximum weight matchings turn out to be much more subtle than simple greedy strategies.

3.2.1 Maximum Spanning Tree

In the maximum spanning tree problem, we are given an undirected graph $G = (V, E)$ with edge weights. Assuming that the graph is connected, we would like to find a spanning tree for which the total weight of edges in it is maximum. It is natural to define a subset system here – $(S, M)$, where $S$ is the set of edges, and $M$ consists of those subsets of $S$ which form a forest (i.e., do not contain a cycle). Note that a maximum independent subset here will be a spanning tree. We know that every spanning tree in a connected graph has $n - 1$ edges, where $n$ is the number of vertices.

3.2.2 Finding minimum weight subset

The greedy algorithm for matroids finds an independent set of maximum total weight. How would you extend it to finding minimum weighted maximal independent subset, for example minimum spanning trees? The well known Kruskal’s algorithm (see Figure Kruskal ) seems identical to the greedy framework except that it chooses the minimum weight element at each stage. Do we need to develop an analogous theory for minimization? Fortunately, we can simply use a reduction to the maximization problem. The idea which just replaces the weight of each element by its negation does not work because the greedy algorithm requires that all weights are non-negative (where?). However, something similar keeps all weights non-negative works.

Suppose the maximum weight of any element in $S$ is $g = \arg \max_{x \in S} \{w(x)\}$. We define another related weight function $w'(x) = g - w(x), \forall x \in S$. Thus $w'(x) \geq 0$. Suppose we now run the Gen-Greedy algorithm with the weight function $w'$. This produces a maximum weight independent subset with respect to the weight function $w'$. Let this subset be $\{y_1, y_2 \ldots y_n\}$ in increasing order of weights $w'(y_i)$, where $n$ equals the size of any maximal independent set in the matroid, i.e., it’s rank (Property 3). In other words among all possible independent subsets in $M$, the maximum weighted subset has weight

$$\sum_{i=1}^{n} w'(y_i) = \sum_{i=1}^{n} (g - w(y_i)) = ng - \sum_{i} w(y_i)$$

This implies that $\sum w(y_i)$ must be the minimum among all maximal independent subsets (else we improve the maximum under $w'$). Moreover, $y_1, y_2 \ldots y_n$ must be in
increasing order of weights under \( w \). This means that if we run the \textit{Gen-Greedy} algorithm by picking the smallest feasible element at each step, we obtain the minimum weighted independent subset. Kruskal’s algorithm is a special case of this fact.

The crucial argument in this reduction is based on the \textit{rank property} of matroid that enabled us to express the weight of the minimum subset in terms of subtraction from the fixed term \( ng \) where \( n \) is the rank of the matroid. If \( n \) was not fixed for all maximal independent subsets, the argument would fail.

### 3.2.3 A Scheduling Problem

We now give another application of the greedy algorithm. Unlike other examples, the construction of the corresponding matroid is not immediately obvious. The scheduling problem is as follows. We are given a set of jobs \( J_1, J_2 \ldots J_n \). Each job \( J_i \) has a deadline \( d_i \) for completion and a corresponding penalties \( p_i \) if it completes its deadline. There is one machine on which each of them needs to be processed (in some order) and each job takes unit amount of time to complete. Our goal is to process the jobs in an order such that the total penalty incurred by the jobs that are not completed before their deadlines is minimized. Stated otherwise, we want to maximize the penalty of the jobs that get completed before their deadlines.

In order to apply the greedy algorithm, we define a subset system \((S, M)\), where \( S \) is the set of all jobs. A set \( A \) of jobs is \textit{independent} if there exists a schedule to complete all jobs in \( A \) without incurring any penalty, i.e., all jobs in \( A \) can be completed within their deadlines. We prove that this set system is a matroid by showing Property 2. Recall that Property 2 states that given any two independent sets \( A, B \) with \( |B| > |A| \), there exists a job \( J \in B - A \) such that \( \{J\} \cup A \) is independent. We prove this by induction on \( |A| \). If \( |A| = 0 \), this is trivial. Now assume Property 2 holds whenever \( |A| = m - 1 \). Now pick two independent sets \( A, B \), with \( |A| = m < n = |B| \).

Consider a feasible schedule \( F_A \) for \( A \), i.e., an ordering of jobs in \( A \) such that each job finishes before its deadline. Note that the \( i^{th} \) job in this ordering finishes at time \( i \) because all jobs are of unit size. Let this ordering be \( A_1, A_2, \ldots, A_m \) (note that \( A_1, \ldots, A_m \) are the jobs in the set \( A \)). Similarly consider a similar schedule \( F_B \) for \( B \), and let the ordering be \( B_1, B_2, \ldots, B_n \).

Note that the deadline of \( B_n \) is at least \( n \) (because it finishes before its deadline in \( F_B \)). If \( B_n \notin A \), then we can add \( B_n \) to \( A \) and schedule it as the last job — this job will finish at time \( m+1 \), whereas it was finishing at time \( n \) in schedule \( B \). Since \( m+1 \leq n \), this job will finish before its deadline. So assume \( B_n \in A \). Now form sets \( A' \) and \( B' \) by removing \( B_n \) from \( A \) and \( B \) respectively. By induction hypothesis, there is a job \( J \in B' - A' \) such that \( A' \cup \{J\} \) is independent. Let \( A'' \) denote \( A' \cup \{J\} \) — we know that there is an ordering \( F_{A''} \) of jobs in \( A'' \) such that every job finishes by its deadline. Now we claim that \( A'' \cup \{B_n\} \), which is same as \( A \cup \{J\} \) is also independent. Indeed,
consider the following schedule – first process the jobs in $A'$ according to $F_{A'}$, and then process $B_n$. Since $|A'| = m$, note that $B_n$ will finish at time $m + 1 \leq n$, which is before its deadline. Thus, we see that $A \cup \{J\}$ is independent. Since $J \in B - A$, Property 2 follows.

Now that our subset system is a matroid, we can use the greedy algorithm to solve the maximization problem. The only remaining detail is how to verify if a set of jobs is independent. For this, we just need to order in increasing order by their deadlines and check if this ordering satisfies all the deadlines (see Exercises).

### 3.3 Efficient data structures for Minimum Spanning Tree algorithms

In this section, we revisit the greedy algorithm for minimum spanning tree in (connected) undirected graphs. This algorithm is also known as Kruskal’s algorithm and was discovered much before the matroid theory was developed. We present it again below without the matroid notation. We first sort the edges in increasing order of weights. The algorithm maintains a set $T$ of edges which will eventually form the desired spanning tree. It considers an edges in this order, and adds an edge $e$ to $T$ only if adding $e$ to $T$ does not create a cycle (i.e., if the set remains independent in the matroid sense).

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Kruskal($G, w$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Input</td>
<td>Graph $G = V, E$, a weight function $w : E \Rightarrow \mathbb{R}$ ;</td>
</tr>
<tr>
<td>2 Sort</td>
<td>$E$ in increasing order of weights. Let ${e_1, e_2 \ldots e_m}$ be the sorted order ;</td>
</tr>
<tr>
<td>3 $T = \emptyset$ ;</td>
<td></td>
</tr>
<tr>
<td>4 for $i = 1$ to $m$ do</td>
<td></td>
</tr>
<tr>
<td>5 if $T \cup {e_i}$ doesn’t contain a cycle in $G$ then</td>
<td></td>
</tr>
<tr>
<td>6 $T \leftarrow T \cup {e_i}$</td>
<td></td>
</tr>
<tr>
<td>7 Output $T$ as MST of $G$.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: Kruskal’s Minimum Spanning Tree Algorithm

The key to an efficient implementation is the cycle test, i.e., how do we quickly determine if adding an edge induces a cycle in $T$? We can view Kruskal’s algorithm as a process that starts with a forest of singleton vertices and gradually connects the graph by adding edges to the set $T$ and hence, growing the trees. In fact, at any point of time, the set $T$ will be a forest. An edge $e$ will be added to $T$ only if the
end-points of \( e \) do not lie in the same connected component (i.e., tree) of \( T \). Adding such an edge will create a cycle. Conversely, if the end-points of such an edge lie in different trees of \( T \), then we can add \( e \) to \( T \) without creating a cycle. When we add such an edge to \( T \), two connected components of \( T \) get replaced by one connected component.

Therefore, we can answer the cycle test query provided we maintain the partition of vertices into trees in \( T \). It should be clear that we need to maintain a data structure that supports the following operations:

**Find** Given a vertex, find out which connected component it belongs to.

**Union** Combine two connected components into one component.

The “Find” operation corresponds to checking if adding an edge creates a cycle. Indeed, we just need to check if the end-points of a vertex belong to the same connected component. The “union” operation is needed to update the set of connected components. When we add an edge, two components in \( T \) merge into one tree.

For obvious reasons, such a data structure is called a “union-find” data-structure. In fact, we can view the data-structure in a more general context. Given a set \( S \), we would like to maintain a family of disjoint subsets of this set \( S \). The data-structure needs to support two operations – given an element of \( S \), find the subset in this family which contains this element, and replace two subsets in this family by their union. We now give details of how to implement this data-structure.
3.3.1 A simple data-structure for union-find

We will use the more general context where we are given a set with \( n \) elements, labeled 1, 2, \ldots, \( n \). We are also given a family of subsets of \( \{1, 2, \ldots, n\} \) – the subsets in this family form a disjoint partition of \( \{1, 2, \ldots, n\} \). Initially, we assume that the family consists of \( n \) singleton sets, one for each element (this corresponds to the case when the set \( T \) is empty in the minimum spanning tree algorithm).

We use an array \( A \) of size \( n \) to represent the sets. For each element \( i \), \( A(i) \) contains the label of the set containing \( i \) – we assign each set in the family a unique label. Initially, we set \( A(i) \) to \( i \). Thus all labels are distinct to begin with. As we merge sets

Figure 3.4: Successive iterations in Kruskal’s greedy algorithm. The dashed edge denotes that the edge could not be selected since it induces a cycle.
(during union operation), we create a new set – we will need to assign a new label. We will ensure that the label of each set remains in the range 1, 2, ..., n at all time. For each set (label), we also have pointers to all its elements, i.e., the indices of the array that belong to the set. Now we perform the two operations as follows:

**Find** This is really simple – for vertex i report A(i). This takes $O(1)$ time.

**Union** To perform union($S_j, S_k$), where $S_j$ and $S_k$ are labels of two sets, we first consider the elements in the two sets and update $A[i]$ value for all such elements i to a unique label. For sake of concreteness, we update the $A[i]$ values of all elements in the set labeled $S_j$ to $S_k$. The time for this operation is proportional to the number of elements in set $S_j$. Note that we had a choice here – we could have changed the labels of all elements in $S_k$ to $S_j$. For obvious reasons, we would change labels of the smaller subset (this is called “union-by-rank” heuristic).

In contexts like the MST algorithm, the time for a single union operation can be quite large. Instead, as in the Kruskal’s algorithm, we often need to analyze the total time taken by a sequence of union operations. Consider a fixed element $x$. The key to the analysis lies in the answer to the following question.

**How many times can the label of $x$ change?**

Every time there is a label change the size of the set containing $x$ increases by a factor of two (Why?). Since the size of any set in our set system is at most $n$, this implies that the maximum number of label changes of $x \log n$. Kruskal’s algorithm involves $|E|$ finds and at most $|V| - 1$ unions; it follows that this can be done in $O(m + n \log n)$ steps using the array daa-structure described above.

### 3.3.2 A faster scheme

The previous data structure gives optimal performance for $m \in \Omega(n \log n)$ – indeed, any algorithm for MST must look at every edge. So theoretically we want to design better schemes for graphs with fewer edges. For this we will explore faster schemes implementing the union-find data-structure.

Instead of arrays, it is easier to visualize the data-structure using trees. We represent every subset using a rooted tree. We will maintain the invariant that every such tree will have as many nodes as the number of elements in the corresponding subset – each node of the tree will be labeled by a unique element of the subset. An

---

5this tree should not be confused with the MST that we are trying to construct
example is given in figure 3.5. We can label the three sets by the labels of their roots, i.e., 6, 12, and 5 respectively.

![Figure 3.5: An example of union find data-structure storing elements \{1, 2, \ldots, 12\}. The three sets are \{6, 4, 10, 1, 9, 7\}, \{12, 8, 3, 11\}, \{5, 2\}.](image)

Initially all trees are singleton nodes (which represent singleton sets). The root of each tree is associated with a label (of the corresponding subset) and a \textit{rank} which denotes the maximum depth of any leaf node in this tree. To perform the operation \texttt{Find}(x), we traverse the tree starting from the node \(x\) till we reach the root and report its label. So the cost of a \texttt{Find} operation is the maximum depth of a node.

To perform \texttt{Union} \((T_1, T_2)\), where \(T_1\) and \(T_2\) are the roots of two trees, we make the root of one tree the child of the root of the other tree. To minimize the depth of a tree, we attach the root of the smaller rank tree to the root of the larger rank tree. This strategy is known as the \textit{union by rank} heuristic. The rank of the resulting tree is determined as follows: if both \(T_1\) and \(T_2\) have the same rank, then the rank of the new tree is one more than the rank of \(T_1\) (or \(T_2\)), otherwise it is equal to the maximum of the ranks of the two trees. Note that once a root node becomes a child of another node, its rank does not change anymore. Clearly, the union operation takes \(O(1)\) steps. We will show in exercises that a tree of rank \(r\) has at least \(2^r\) nodes, and so, the depth of any tree in this data-structure will be bounded above by \(\log n\). It follows that a \texttt{Find} operation will take \(O(\log n)\) time. We have already argued that a \texttt{Union} operation takes \(O(1)\) time. Therefore, Kruskal’s algorithm can be implemented in \(O(m \log n + n)\) time. This seems to be worse than the array implementation mentioned above. Seemingly, we haven’t quite gained anything so let us use the following heuristic.

**Path compression Heuristic.** In this heuristic, we try to reduce the height of a tree even below \(\log n\) by \textit{compressing} any sequence of nodes which lie on a path starting from the root. When we do a \texttt{Find}(x) operation, let \(x_0 = \text{root of } x, x_1, x_2 \ldots x\) be the sequence of nodes visited (in the reverse order starting from \(x\)). In this heuristic, we update \(x_0\) as the parent of \(x_1, x_2, \ldots, x\) (i.e., \(x_0\) has all the other nodes in this path as its children). An example is given in Figure 3.6
Figure 3.6: An example of path compression heuristic. The operation Find(6) causes 6, 4 and 7 to become children of the root node.

Clearly, the motivation is to bring more nodes closer to the root node, so that the time for subsequent Find operations involving these nodes decrease. Note that the time spent during the path compression heuristic is not much – it only doubles the cost of the current Find operation.

While it is intuitively clear that it should give us an advantage, we have to rigorously analyze if it indeed leads to any asymptotic improvement. Before we get into the analysis, we first introduce some slow growing functions which will be used for expressing the heights of such trees.

### 3.3.3 The Slowest Growing Function

Let us look at a very rapidly growing function, namely the *tower of two*. The tower looks like

\[ 2^{2^{2^{\ldots^2}}} \]

which can defined more formally as the following function

\[
B(i) =
\begin{cases}
2^1 & i = 0 \\
2^2 & i = 1 \\
2^{B(i-1)} & \text{otherwise for } i \geq 2
\end{cases}
\]

We now define the operation which applies log iteratively \( i \) times. More formally, define

\[
\log^{(i)} n =
\begin{cases}
n & i = 0 \\
\log(\log^{(i-1)} n) & \text{for } i \geq 1
\end{cases}
\]

The inverse of \( B(i) \) is defined as

\[
\log^* n = \min\{i \geq 0 | \log^{(i)} n \leq 1\}
\]
In other words,

$$\log^* 2^{2^{2^{2^n}}} = n + 1$$

We will use the function $B()$ and $\log^*()$ to analyze the effect of path compression. We will say that two integers $x$ and $y$ are in the same block if $\log^* x = \log^* y$.

Although $\log^*$ appears to slower than anything we can imagine, (for example $\log^* 2^{65536} \leq 5$), there is a closely related family of function called the inverse Ackerman function that is even slower!

Ackerman’s function is defined as

$$A(1, j) = 2^j$$

for $j \geq 1$

$$A(i, 1) = A(i - 1, 2)$$

for $i \geq 2$

$$A(i, j) = A(i - 1, A(i, j - 1))$$

for $i, j \geq 2$

Note that $A(2, j)$ is similar to $B(j)$ defined earlier. The inverse-Ackerman function is given by

$$\alpha(m, n) = \min\{i \geq 1 | A(i, \lfloor \frac{m}{n} \rfloor) > \log n\}$$

To get a feel for how slowly it grows, verify that

$$\alpha(n, n) = 4 \text{ for } n = 2^{2^{2^{2^{2^{16}}}}}$$

### 3.3.4 Putting things together

Clearly the cost of Find holds key to the analysis of this Union Find data structure. Since the rank of any root node is at most $\log n$, we already have an upper bound of $\log n$ for any individual Find operation. We now show that the path compression heuristic further reduces the cost of Find operations to $O(\log^* n)$. Before we do this, recall that the definition of rank function. We state below some simple properties of the rank function. The proof is left to the exercises.

**Lemma 3.1** The rank function has the following properties:

- **Property 1:** The rank of root node is strictly larger than that of any children nodes.

- **Property 2:** There are at most $n/2^r$ nodes of rank $r$.

- **Property 3:** For a node $v$, the rank of its parent node never decreases (note that the parent node could change because of union operations followed by path compression).
• Property 4: If the root node of a tree changes from \( w \) to \( w' \), then rank of \( w' \) is strictly larger than that of \( w \).

We shall use \( \log^*(\text{rank}(v)) \) for a node \( v \) to refer to the block number of \( v \). We will adopt the following strategy for counting the cost of Find operations. Whenever a Find operation visits a node, we refer to this as a charging the node. Clearly, the total cost of all find operations is bounded by the total number of charges. We distinguish between three kinds of charges

**Base Charge** If the parent of \( v \) is the root node (of the tree containing \( v \)), then \( v \) receives a base charge. Clearly, each Find operation incurs at most one base charge.

**Block charge** If the block number of the parent node \( p(v) \) is strictly greater than that of the node \( v \), i.e., \( \log^*(\text{rank}(p(v))) > \log^*(\text{rank}(v)) \), then we assign \( v \) a block charge. Clearly the maximum number of block charges for a single Find operation is \( O(\log^* n) \)

**Path charge** Any charge incurred by a Find operation that is not a block charge.

From our previous observation, we will focus on counting the path charges. Consider a node \( v \). Whenever it gets a path charge, its parent is the root node. For it to incur a path charge again, the root node of this tree needs to change. But then the rank of the root node will go up (Property 4). Consequently, \( v \), whose rank lies in block \( j \) (say), will continue to incur path charge for at most \( B(j) - B(j - 1) \leq B(j) \) Find operations.

Since the number of elements with rank \( r \) is at most \( \frac{n}{2^r} \) (Property 2), the number of elements having ranks in block \( j \) is

\[
\frac{n}{2B(j-1)+1} + \frac{n}{2B(j-1)+2} + \ldots + \frac{n}{2B(j)} = n \left( \frac{1}{2B(j-1)+1} + \frac{1}{2B(j-1)+2} + \ldots \right) \\
\leq 2n \cdot \frac{1}{2B(j-1)+1} = \frac{n}{2B(j-1)}
\]

Therefore the total number of path charges for elements in block \( j \) is at most \( \frac{n}{2B(j-1)} \cdot B(j) \) which is \( O(n) \). For all the \( \log^* n \) blocks the cumulative path charges is \( O(n \log^* n) \). Further, the total number of block and base charges will be \( O(m) \) (the number of Find and Union operations). Therefore, the total time spent by the Find and Union operations is \( O(m \log n) \).
3.3.5 Path compression only

We would now like to gain a better understanding of role of the path compression. If we only use path-compression, without the union-by-rank heuristic, we cannot bound the rank of a node by $\log n$. Nevertheless one can still define rank of a node as before. Note that the ranks of the sequence of nodes starting from any node to the root still increase monotonically. Without the union-by-rank heuristic, the ranks of nodes in this sequence can increase by more than one (in fact arbitrarily) after a union operation. We would now analyse the running time of MST algorithm using path compression heuristic only.

Let us denote the parent of a node $x$ as $p(x)$. Let us define the level of a node $x$, denote by $\ell(x)$, as the integer $i$ such that $2^{i-1} \leq \text{rank}(p(x)) - \text{rank}(x) \leq 2^i$. Therefore $\ell(x) \leq \log n$. Note that $\ell(x)$ is defined for non-root vertices only.

We account for the cost of a Find($x$) operation by charging one unit of cost to all the nodes in the path from $x$ to the root (except the root). The only exception is that for any level $i$, $1 \leq i \leq \log n$, the last node (in the path to the root) in level $i$ is not charged. Instead the cost is charged to the Find operation. Clearly the number of charges to the Find operation is $O(\log n)$. For any other node $y$, we claim that whenever it gets charged by the Find operation, $\ell(y)$ increases by at least one. Since $\ell(y)$ is bounded by $\log n$, this will show that the Find operation charges $y$ at most $\log n$ times.

Let us now see why the claim is correct. Since $y$ is not the last node in its level, there is another node $v$ above $y$ in the path to the root such that $\ell(v) = \ell(y) = i$ (say). By definition of level,

$$\text{rank}(p(v)) - \text{rank}(y) = \text{rank}(p(v)) - \text{rank}(v) + \text{rank}(v) - \text{rank}(y) \geq \text{rank}(p(v)) - \text{rank}(v) + \text{rank}(p(y)) - \text{rank}(y) \geq 2 \cdot 2^i = 2^{i+1}.$$ 

The second last inequality above follows from the fact that $v$ lies above $p(y)$ in the path from $y$ to the root, and so, rank of $v$ will be at least that of $p(y)$. Let $w$ be the parent of $v$ (before this Find operation) and $r$ be the root of this tree. Again, by rank monotonicity, $\text{rank}(r) \geq \text{rank}(w)$. We have shown above that $\text{rank}(w) - \text{rank}(y) \geq 2^{i+1}$, and so, $\text{rank}(r) - \text{rank}(y) \geq 2^{i+1}$ as well. Since $r$ will now be the parent of $y$, it follows that the level of $y$ will be $i + 1$ or higher. This proves our claim.

Therefore, over the course of all the union-find operations, a node can get charged at most $\log n$ times resulting in a total cost of $O(m \log n)$ for all the find operations.

3.4 Greedy in different ways

The matroid structure is closely related to the form of the greedy algorithm described in Figure Greedy. But there may other variations that attempt to choose the next
Procedure Prim($G, w$)

1. **Input** Graph $G = (V, E)$, a weight function $w : E \rightarrow \mathbb{R}$;
2. $T = e_1$ where $e_1 \in E$ is the smallest weighted edge;
3. **for** $|T| \leq n - 1$ **do**
   4. Let $(u, v)$ be the least weight edge in $V_T \times (V - V_T)$;
   5. $T \leftarrow T \cup \{(u, v)\}$;
6. **Output** $T$ as MST of $G$.

Figure 3.7: Prim’s Minimum Spanning Tree Algorithm

best element without necessarily picking them in reverse sorted order of their weights. One such classic algorithm is Prim’s MST algorithm described in Figure Prim. Recall that Kruskal’s algorithm maintains several connected components – at each step, it picks an edge and merges two of these components into one. In Prim’s algorithm, we maintain only one connected component, which is initially just the cheapest edge. At each step, the algorithm finds the least weight edge which can extend this connected component by one more edge.

Although Prim’s algorithm intuitively seems to do the right thing, note that the sequence of edges that it picks could be different from Kruskal’s and therefore it requires a separate proof of correctness. It is clear that it outputs a tree. Indeed, it always picks an edge one of whose endpoints is not in the current component $T$. Therefore the added edge cannot induce a cycle.

Let us first address the running time. The algorithm needs to pick the least weighted edge going out of the current tree $T$. For this we can maintain a label for every vertex denoting its distance from the tree $T$. If $v \in T$ then its label is 0. Let $N(v)$ denote the neighbors of $v$ in $G$. Then label of $v$ is $\ell(v) = \min_{u \in N(v)} w(u, v) + \ell(u)$. (If no neighbor of $v$ belongs to $T$ then its label is $\infty$.) The labels maintain the shortest distance of any vertex to the nearest vertex in $T$. The details of maintaining this data-structure and update time are left to exercises (see also Dijkstra’s algorithm in Chapter 9). We can use a heap data-structure to store these labels such that finding the minimum weight edge will take $O(\log n)$ time. Consequently, the algorithm can be implemented in $O(m \log n)$ time.

To prove correctness we will invoke a useful result, whose proof is left as an exercise. The setting is an undirected weighted graph $G = (V, E)$ whose edges are colored red, blue (or left uncolored) according to the following rules:

(i) **Red rule**: An edge is colored red if it is the heaviest (i.e., highest weight) edge in a cycle$^6$.

$^6$Assume all edges have unique weight.
(ii) **Blue rule**: An edge is colored blue if it is the lightest edge across any cut of the graph. A cut is a partition of the vertices $V$ and an edge across the cut has one endpoint in each partition.

(iii) The two rules can be applied in any order.

**Theorem 3.2 (red-blue rule)** There exists an MST of $G$ that includes all the blue edges and none of the red edges.

The proof is left as an exercise. The applications are quite powerful. Prim’s algorithm can be seen as coloring edges blue where each blue edge is across the cut defined by the tree vertices and the remaining graph.

Kruskal’s algorithm can be viewed as coloring an edge red if the two end points are within the same component and the order of adding the edges ensures that it is the heaviest edge (in the cycle formed by this edge and the edges in the component). On the other hand if an edge connects two components then it must be a cut-edge if the two components are in different partitions (other components can be arbitrarily assigned to either partitions). Moreover, it is the lightest edge among the edges not added, and so it must be colored blue by definition.

A lesser known algorithm called Borůvka’s algorithm is described in Figure Boruvka. The algorithm maintains several connected components at any point of time (as in Kruskal’s algorithm). The set $\mathcal{F}$ denotes the set of these components. At any point of time, it picks a component $C$ in $\mathcal{F}$ and chooses the least weight edge which has exactly one end-point in $C$ – such an edge would have its other end-point in some other component $C'$ in $\mathcal{F}$. The algorithm picks this edge $e$ and replaces $C$ and $C'$ by $C \cup C' \cup \{e\}$. Note that the choice of $C$ is arbitrary. The algorithm terminates when there is one connected component in $\mathcal{F}$.

The correctness of the algorithm follows from use of blue rule and an additional assumption that the edge weights are unique\(^7\). Indeed, whenever we add an edge joining $C$ and $C'$, it is the cheapest edge in the cut formed by $C$ and rest of the vertices. There are several advantages of this algorithm. It is inherently parallel in nature as all the components can simultaneously choose the nearest neighboring vertex. Moreover, the fastest known linear MST algorithm is based on an adaptation of this algorithm with clever use of randomization.

### 3.5 Compromising with Greedy

So far, we have shown that the greedy strategy yields an optimal solution for a large class of problems. However, in many cases, the greedy strategy does not doesn’t

\(^7\)One can add lower significant bits based on edge labels to break ties.
Procedure Borůvka\( (G, w) \)

1. **Input** Graph \( G = (V, E) \), a weight function \( w : E \rightarrow \mathbb{R} \);
2. \( F = \{\{v_1\}, \{v_2\}, \ldots\} \) where \( v_i \in V \) are initial components without any edges;
3. \( T = \emptyset \);
4. **while** \( |F| > 1 \) **do**
   5. Pick a component \( C \) in \( F \);
   6. Let \((v, w) \in E\) be the least weight edge out of component \( C \);
   7. Suppose \( w \) lies in component \( C' \) in \( F \);
   8. Replace \( C \) and \( C' \) by \( C \cup C' \cup \{(v, w)\} \) in \( F \).
9. **Output** the single component in \( F \) as MST of \( G \).

---

Figure 3.8: Borůvka’s Minimum Spanning Tree Algorithm

always pay in terms of achieving an optimal solution. It is still attractive because of its simplicity and efficiency. Therefore, we may relax on our objective of find an optimal but compromise with a near optimal solution. We touch on this aspect of algorithm design in a later chapter more formally so we illustrate this with an example.

Recall the **maximum matching** problem discussed in Example 3.4. Although the example discussed the special case of bipartite graphs, the same definition extends to general graph. More formally, we are given an undirected weighted graph \( G = (V, E) \). We want to find a subset \( E' \subset E \) such that no two edges in \( E' \) share any end-points (the degree of the induced subgraph is exactly 1) and we want to maximize the number of edges in \( E' \). For a weighted graph we want to maximize \( \sum_{e \in E'} w(e) \) where \( w(e) \) is the weight of \( e \). We had shown in Example 3.4 that the subset system corresponding to matchings is not a matroid.

Nevertheless let us persist with greedy strategy for finding a matching and analyse the outcome. Consider the following algorithm: sort the edges in decreasing order of weight. Maintain a solution \( G \) initialized to empty. We look at edges in this order, and add an edge to \( G \) if it does not have a common end-point with any of the edges chosen in \( G \) so far. It is easy to show that this algorithm may not give optimal solution. Still it turns out that the total weight of edges in \( G \) is always at least half of that of an optimal solution. We now prove this claim.

Let \( O \) denote an optimal solution. Let \( w(O) \) and \( w(G) \) denote the total weight of edges in \( O \) and \( G \) respectively. Clearly \( w(O) \geq w(G) \). Consider an edge \( e = (x, y) \in O \setminus G \). When the greedy algorithm had considered \( e \), there must have been an edge \( e' \in G \) which shared a common end-point with \( e \). Further, \( w(e') \geq w(e) \). Thus we get a function \( B \) from \( O \setminus G \) to \( G \setminus O \) (mapping edges \( e \) to \( e' \)). How many edges can
map to an edge $e'$ in $G$ using this mapping $B$? We claim that there can be at most 2 such edges, and both these edges have weight at most $w(e')$. Indeed, $e'$ has two endpoints, and if $B(e) = e'$, then $e$ must have a common end-point with $e'$. The claim now follows from the fact that no two edges in $O$ (and so, in $O \setminus G$) share a common end-point. Therefore, the total weight of edges in $O \setminus G$ is at most twice that of edges in $G \setminus O$. Therefore, $w(O) = w(O \setminus G) + w(O \cap G) \leq 2w(G \setminus O) + w(G \cap O) \leq 2w(G)$.

Thus the greedy strategy can have some provable guarantees even though it does not yield an optimal solution.

### 3.6 Gradient Descent

So far we have used the greedy strategy to solve “discrete” optimization problems, i.e., problems where a decision variable can take finite set of values. For example, in the minimum spanning tree problem, we have one variable with each edge – should we include this edge in the tree solution? This variable is a binary variable, because it can take only two values – true or false. Similarly, in the more general setting of finding maximum weight independent set in a matroid, we have to decide whether to add an element in the independent set or not. We now consider a different application of the greedy strategy where (at least in principle) the variables can have values from a continuous interval.

We are given a continuous (and differentiable) function $f: \mathbb{R}^n \to \mathbb{R}$, where the domain of the function, denoted by $\text{dom}(f)$, is a convex and compact set. We would like to find a minimizer of $f$, i.e., a point $x^* \in \text{dom}(f)$ such that $f(x^*) \leq f(x)$ for all $x \in \text{dom}(f)$. Because the domain is a compact set, we know that there will be at least one such point $x^*$, though it may not be unique. The problem is not very well defined because it may not be possible to express the point $x^*$ using finite precision. Consider, for example, the function $f(x) = x^2 - 2$. Clearly, $x^* = \sqrt{2}$, and it cannot be expressed using finite number of bits. We shall assume that the input instance also provides an error parameter $\varepsilon$, and we are required to find a point $x$ such that $f(x) - f(x^*) \leq \varepsilon$. 


Figure 3.9: A convex function of one variable. The line joining the points \((x, f(x))\) and \((y, f(y))\) stays above the plot of \(f\).

For a general function \(f\), we do not know how to solve this problem in polynomial time (where the polynomial may include terms depending on the diameter of the domain of \(f\) and bounds on the slope of \(f\)). We focus on a special class of functions called convex functions. It turns out that any local minimum for such functions is also a global minima, and so, it suffices to run a greedy algorithm to find a local minimum of such a function. Let \(f\) be a function of one variable, i.e., \(f: \mathbb{R} \rightarrow \mathbb{R}\). We say that \(f\) is convex if for every \(x, y \in \text{dom}(f)\), and parameter \(\lambda\), where \(0 \leq \lambda \leq 1\),

\[
f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y). \tag{3.6.1}
\]

Graphically, it means that if we look at the plot of \(f\), then the line joining the points \((x, f(x))\) and \((y, f(y))\) should lie above the curve \(f\) in the interval \([x_1, x_2]\) (see Figure 3.9). We say that \(f\) is strictly convex if the above inequality is strict. It is not clear how we can use this definition to easily check if a function is convex. Fortunately, if we make some mild assumptions on \(f\), then there are other equivalent definitions which turn out to be easier to work with. It turns out that if \(f\) is differentiable, then it is enough to check that for every pair of points \(x, y \in \text{dom}(f)\),

\[
f(y) \leq f(x) + (y - x) \cdot f'(x), \tag{3.6.2}
\]

where \(f'(x)\) denotes the derivative of \(f\). This result gives another way of thinking about convex functions: if we draw the tangent at any point on the curve corresponding to \(f\), then the entire curve lies above the tangent line. If \(f\) happens to be twice differentiable, there happens to be another intuitive definition of convex functions: the second derivative of \(f\) is always non-negative (see Exercise 3.9).
We now extend these definitions to a function of multiple variables. We say that a function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) is convex if the restriction of \( f \) on any line is a convex function. Recall that a line can be specified by two vectors: a point \( x_0 \) on the line, and a direction \( d \). Any point on this line can be described by a single parameter \( t \): \( x_0 + t \cdot d \). Thus, we can define a function \( h(t) = f(x_0 + t \cdot d) \), and think of \( h \) as the restriction of \( f \) on this line. According to our definition, \( f \) is convex iff every such function \( h \) is convex.

As in the case of functions of one variable, we now want to define convexity in terms of first and second derivatives of \( f \). Let \( \nabla f \) denote the gradient of \( f \). Exercise 3.11 shows the analogous statements. It turns out that a local minimum of a convex function is also a global minimum. More precisely,

**Lemma 3.2** Suppose \( x \in \text{dom}(f) \) and \( x \) is a local minimum of \( f \), i.e., there is a radius \( r > 0 \) such that \( f(x) \leq f(y) \) for all \( y \) satisfying \( ||y - x|| \leq r \). Then \( x \) is also a global minimum of \( f \), i.e., \( f(x) \leq f(y) \) for all \( y \in \text{dom}(f) \).

The intuitive reason for the above statement is as follows. Suppose a convex function has a local minimum at \( x \). Let \( x' \) be any other point in the domain of \( f \). Consider the one-dimensional projection of \( f \) along the line joining \( x \) and \( x' \). Suppose, for the sake of contradiction, that \( f(x') < f(x) \). Then, by convexity of \( f \), the curve corresponding to \( f \) lies below the line joining \( x \) and \( x' \). Therefore, we can find a point \( x'' \) in the vicinity of \( x \) such that \( f(x'') < f(x) \). This contradicts the fact that \( f \) is convex.

Thus, if we want to find the minimum of a convex function, it is enough to find a local minimum of \( f \) – note that in general, a convex function may not have a unique local minimum, but a strictly convex function has a unique local minimum. The gradient descent algorithm is a popular greedy algorithm for minimizing a convex function. Intuitively, it starts from an arbitrary point in the domain of \( f \), and tries to move along “steepest direction” at the current point.

The algorithm starts with an initial guess \( x^{(0)} \), and iteratively moves to points \( x \) which have smaller \( f(x) \) values. The intuition can be described as follows. Suppose we are currently at a point \( x \) and want to make a small step of size \( \eta \) along a direction \( d \), i.e., move to a point \( x + \eta d \), where \( d \) is a unit vector. What should be the best choice for \( d \)? If \( \eta \) is a small quantity, then we can approximate \( f \) by a linear approximation using Taylor expansion:

\[
  f(x + \eta d) \approx f(x) + \eta d^T \nabla f(x).
\]

Now, we know that \( |d^T \nabla f(x)| \leq ||d|| ||\nabla f(x)|| \), with equality if and only if \( d \) is along \( \nabla f(x) \). Therefore, we should pick \( d \) along the negative gradient direction. This motivates the gradient descent algorithm described in Figure Gradient Descent. The parameter \( \eta \), which is often called the “learning rate”, should be chosen carefully: if
it is too small, then the progress towards a local minimum will be slow, whereas if we pick $\eta$ to be large, we may not converge to the desired point. Similarly the time $T$ till which run the algorithm depends on how close we want to come to the optimal solution.

**Procedure** Gradient Descent($f, \eta, x_0$)

1. **Input** Convex function $f$, step size $\eta$ and initial point $x^{(0)}$;
2. **for** $t = 1, \ldots, T$ **do**
   3. $x_t \leftarrow x_{t-1} - \eta \nabla f(x)$
5. **Output** $x_T$.

Let us see some examples. Consider $f(x) = x^2 - 1$. Clearly $x^* = 0$ is the global minimum. Now, if we start at $x = 1$, and set $\eta = 10$, it is easy to see that the successive points will diverge from $x^*$. Therefore, it is important to keep $\eta$ small, preferably much smaller than the distance between the current point and the desired minimum. However, as Exercise 3.14 shows, even a very small value of $\eta$ can lead to this oscillatory behaviour if we do not assume smoothness properties of $f$. The reason for the oscillatory behaviour of gradient descent in this exercise happens because the derivative of the function changes suddenly at $x = 0$. We now assume that the derivative of the function cannot change very fast, i.e., there is a parameter $L$ such that for all $x, y \in \text{dom}(f),

$$||\nabla f(x) - \nabla f(y)|| \leq L \cdot ||x - y||.$$  

Such a convex function is said to be $L$-smooth. One consequence of $L$-smoothness is that a convex function cannot deviate from the tangent line at a point too fast. Let $x$ and $y$ be two points in the domain of $f$. Then,

$$0 \leq f(y) - f(x) - f'(x) \cdot (y - x) \leq \frac{L}{2} \cdot (y - x)^2.$$  

(3.6.3)

The first inequality follows by the definition of convexity. For the other inequality note that

$$f(y) - f(x) - f'(x) \cdot (y - x) = \int_0^1 (f'(x + t(y - x)) - f'(x)) \cdot (y - x)dt = \int_0^1 f'(x + t(y - x)) \cdot (y - x)dt \leq L \cdot \int_0^1 t(y - x)^2 = \frac{L}{2} \cdot (y - x)^2.$$  

53
The following theorem states that gradient descent converges in small number of steps provided we pick $\eta$ suitably. We prove this for a function $f$ of one variable only – the ideas for the more general proof when $f$ is a function of several variables are similar, though the details require a bit more work. We first observe a simple consequence of $L$-smoothness.

**Theorem 3.3** Let $f$ be an $L$-smooth convex function. Let $x^*$ denote a global minimum of $f$. If we run gradient descent with $\eta = 1/L$, then $f(x_t) - f(x^*) \leq \varepsilon$ for $t \geq \frac{LR^2}{\varepsilon}$, where $R$ denotes $|x_0 - x^*|$. To prove this result, we show that the gradient descent algorithm makes some progress at each step – the progress is higher if we are farther from $x^*$ and it slows down as we start approaching $x^*$. From the description of the gradient descent algorithm, we know that $x_s - x_{s+1} = \eta f'(x_s) = \frac{f(x_s)}{L}$. Using this fact and substituting $x = x_s$ and $y = x_{s+1}$ in inequality (3.6.3), we see that

$$f(x_s) - f(x_{s+1}) \geq (x_s - x_{s+1})f'(x_s) = \frac{L}{2}(x_s - x_{s+1})^2 = \frac{1}{2L} \cdot f'(x_s)^2.$$ 

Thus, if $f'(x_s)$ is large, we make more progress, and as we approach $x^*$, $f'(x_s)$ gets closer to 0, and so, our progress also slows down. Now, we show how to make this argument more formal. Assume without loss of generality that $x_0 > x^*$. We will show that $x_s$ will also be at least $x^*$ for all values of $s \geq 1$. Assuming this is the case, it follows that $x_s - x^* \leq x_0 - x^* \leq R$. Now, if $\delta_s$ denotes $f(x_s) - f(x^*)$, then the above observation can be restated as

$$\delta_s - \delta_{s+1} \geq \frac{f'(x_s)^2}{2L}.$$  

Now, convexity of $f$ implies that

$$\delta_s = f(x_s) - f(x^*) \leq f'(x_s)(x_s - x^*) \leq Rf'(x_s).$$

Substituting this in inequality (3.6.4), we get

$$\delta_s - \delta_{s+1} \geq \frac{\delta_s^2}{2LR^2}.$$

It remains to solve this recurrence relation. Observe that

$$\frac{1}{\delta_{s+1}} - \frac{1}{\delta_s} = \frac{\delta_s - \delta_{s+1}}{\delta_s \delta_{s+1}} \geq \frac{\delta_s - \delta_{s+1}}{\delta_s^2} \geq \frac{1}{2LR^2}.$$ 

Adding this for $s = 1, \ldots, T_1$, we see that

$$\frac{1}{\delta_T} - \frac{1}{\delta_1} \geq \frac{T - 1}{2LR^2}.$$
Finally, notice that inequality (3.6.3) implies that

\[ \delta_{1} = f(x_{1}) - f(x^{*}) \leq f'(x^{*})(x_{1} - x^{*}) + \frac{LR^{2}}{2} = \frac{LR^{2}}{2}. \]

Substituting this in the above inequality, we see that \( \delta_{T} \) is \( O(LR^{2}/T) \). This proves the theorem.

It remains to show that \( x_{s} \) always stays at least \( x^{*} \), i.e., the iterates never cross from the right of \( x^{*} \) to the left of \( x^{*} \). This happens because our step sizes are small enough – in fact this illustrates the fact the step size should be large enough to make enough progress but small enough to avoid “overshooting” the desired point. From the definition of \( L \)-smoothness, we know that \( f'(x_{s}) - f'(x^{*}) \leq L(x_{s} - x^{*}) \), and so, \( f'(x_{s}) \leq L(x_{s} - x^{*}) \). Therefore, \( x_{s+1} = x_{s} - f'(x_{s})/L \geq x_{s} - (x_{s} - x^{*}) \geq x^{*} \). This completes the analysis of the greedy algorithm and shows that under mild conditions, it converges to the optimal solution.

Remarks: 1. In practice, the parameter \( \eta \) is chosen in an ad-hoc manner by figuring out the right trade-off between convergence rate and accuracy.

2. The decision to stop the iterations of the gradient descent algorithm can also be based on several criteria: (i) there could be an upper bound on the number of iterations, (ii) the value \( ||x_{t} - x_{t-1}|| \) becomes smaller than a given threshold, (iii) the values \( f(x_{t}) - f(x_{t+1}) \) become smaller than a given threshold.

3. Sometimes the function \( f \) may not be differentiable at the current point \( x_{t} \). Consider for example the function in Figure 3.11 – this function is convex, but not differentiable at the point \( x \). It turns out that one can still use the gradient descent algorithm at \( x \) provided one uses a vector \( v \) instead of the gradient \( \nabla f(x) \) provided the following condition holds for all points \( y \) in the domain of \( f \):

\[ f(y) \geq f(x) + (y - x)^{T}v. \]

Such a vector \( v \) is called a sub-gradient at \( x \) – note that there is no unique choice for \( v \) here.
3.6.1 Applications

Gradient descent is a very popular general purpose algorithm for optimizing a function. In practice, it is used even if the function is not convex – the hope is that one would instead converge to a local optimum. We now give some sample applications.

Locating a point by multiple measurements

Suppose you want to find the location an object \(P\) in the two-dimensional plane. There are 3 observation points \(O_1, O_2, O_3\). For each of the observation points \(O_i\), you can measure the distance \(r_i\) between \(P\) and \(O_i\). As shown in Figure ??, you can find the location of \(P\) by finding the common intersection point of the circles of radii \(r_1, r_2, r_3\) centered at \(O_1, O_2, O_3\) respectively. But the measurements incur some error, and so you only know approximations to \(r_1, r_2, r_3\) – call these \(\tilde{r}_1, \tilde{r}_2, \tilde{r}_3\). Given these three approximate values, you would like to find the best possible estimate for the location of \(P\) (see Figure 3.12).

Such problems are often solved by solving an appropriate optimization problem. Suppose the coordinates of \(O_i\) are \((a_i, b_i), i = 1, 2, 3\). Let \((x, y)\) be the coordinates of \(P\). Note that \((a_i, b_i)\) are known quantities, whereas we would like to find \((x, y)\). Assuming the errors in the measurements are small, one way of framing this problem would be find the values of \((x, y)\) such that the overall error is as small as possible. In other words, let \(f_i(x, y)\) denote \((\tilde{r}_i - \sqrt{(a_i - x)^2 + (b_i - y)^2})^2\). Observe that \(f_i(x, y)\) denotes (square of) the error in the measurement of \(r_i\). And so we would like to find the value of \((x, y)\) such that \(f(x, y) = \sum_{i=1}^{3} f_i(x, y)\) is minimized. We can solve this by the gradient descent algorithm. It is easy to write down the gradient of \(f(x, y)\) and so, one can run the gradient descent algorithm till the values converge.
Figure 3.12: The point $P$ should ideally lie on the intersection of the three circles, but there are some measurement errors.

Perceptron Algorithm

A neuron is often modelled as a unit with a threshold $w_0$. When the input to the neuron (in terms of exceeds $w_0$, it outputs 1. Otherwise it outputs 0. Consider the situation shown in Figure 3.13. There are $n$ input variables $x_1, x_2, \ldots, x_n$, and weights $w_1, w_2, \ldots, w_n$ (shown on the “input” edges in the figure). Therefore the input to the neuron is $w_1x_1 + \ldots + w_nx_n$ – if this exceeds $w_0$, output is 1; otherwise output is -1. In other words (replacing $w_0$ by $-w_0$), the output is determined by the sign of $w_0 + w_1x_1 + \ldots + w_nx_n$.

The perceptron algorithm takes as input several pairs $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$, where each $x^{(j)}$ is a vector $(x^{(j)}_1, \ldots, x^{(j)}_n)$, and $y^{(j)}$ is either -1 or 1. Given such input pairs, we would like to find the values $w_0, w_1, \ldots, w_n$. One way of thinking about this problem is as follows: consider the hyperplane $w_0 + w_1x_1 + \ldots + w_nx_n = 0$ in $n$ dimensions (where the coordinates are given by $(x_1, x_2, \ldots, x_n)$). The points $x^{(j)}$ for which $y^{(j)}$ is 1 lie on one side of this hyperplane, whereas the rest lie on the other side. Thus, we can frame this problem as follows: given a set of points where each point is labelled either ‘+’ or ‘-’ (depending on whether the $y$-coordinate is 1 or 0), find a hyperplane separating the points.

---

8The inputs and outputs are electrical signals in nature. Also note that this is an ideal model of a neuron. In reality, there will be a “grey” area where it outputs something between 0 and 1.
Figure 3.13: A perceptron with inputs $x_1, x_2, \ldots, x_n$ and output determined by the sign of $w_0 + w_1 x_1 + \ldots + w_n x_n$.

We can express this as an optimization problem as follows. Let $\mathbf{w}$ denote the vector $(w_0, w_1, \ldots, w_n)$. Given such a solution $\mathbf{w}$, we can count the number of misclassified inputs – an input $(x^{(j)}, y^{(j)})$ is mis-classified if the sign of $\mathbf{w}^T x^{(j)}$ is positive if and only if $y^{(j)}$ is -1. More formally, define a function $\text{sgn}$ as $\text{sgn}(z)$ to be 1 if $z < 0$, 0 otherwise. Notice that the quantity $\text{sgn}(y^{(j)} \cdot \mathbf{w}^T x^{(j)})$ is 1 if and only if we make a mistake for the input $(x^{(j)}, y^{(j)})$. Therefore, we can state this problem as minimizing $f(\mathbf{w}) := \sum_j \text{sgn}(y^{(j)} \cdot \mathbf{w}^T x^{(j)})$. However, the function $f$ is not convex – it is easy to see that the function $\text{sgn}$ (as a function of one variable) is not convex. Instead, we replace $f$ by another function which is convex and approximates the $\text{sgn}$ function. There are various ways to do this. We will use a simple approach which replaces $\text{sgn}$ by the following function $g$: $g(x) = x$ if $x < 0$, otherwise it is 0. This function is shown in Figure 3.14. Note that the function $g$ is convex, but it is not differentiable at $x = 0$. Recall from the discussion on the gradient descent algorithm that we can run it on a non-differentiable convex function as long as we can define the notion of a sub-gradient. It is easy to see that we can define any number between 0 and $-1$ as the sub-gradient at $x = 0$ – we shall define it to be 0 here. If $x < 0$, the derivative of $g$ is $-1$. If $x \geq 0$, the derivative is 0. Now, we can define the function $f$ as $f(\mathbf{w}) := \sum_j g(y^{(j)} \cdot \mathbf{w}^T x^{(j)})$. Observe that if $\mathbf{w}$ indeed represents a separating hyperplane, then $f$ would be 0. In this sense, we are justified in replacing $\text{sgn}$ by the function $g$. It is now easy to write down the derivative of $f$ at a point $\mathbf{w}$. Let $N(\mathbf{w})$ be the set of input points which are incorrectly classified by $\mathbf{w}$, i.e., $y^{(j)} \cdot \mathbf{w}^T x^{(j)} < 0$. Then, the derivative is given by $-\sum_{j \in N(\mathbf{w})} y^{(j)} x^{(j)}$. Thus, we get the following simple rule for finding the separating hyperplane: if $\mathbf{w}^t$ is the estimate for $\mathbf{w}$ at iteration $t$, it
then:

\[ w^{t+1} = w^t - \eta \sum_{j \in w^t} y^{(j)} x^{(j)}. \]

Geometrically, this rule is saying that we \textit{tilt} the vector \( w^t \) based on the points which are getting mis-classified.

![Figure 3.14: A plot of the function \( g \) as an approximation to the \texttt{sgn} function.](image)

**Exercises**

**Exercise 3.1** Construct an instance of a knapsack problem that visits every leaf node, even if you use branch and bound. You can choose any well defined way if pruning the search space.

**Exercise 3.2** Consider the special case of \( k = 1 \) in the analysis of the AND-OR tree. Show that the expected number of evaluations is 3. (You must consider all cases of output and take the worst, since we are not assuming any distribution on input or output).

**Exercise 3.3** Complete the analysis of the AND-OR tree when the root is an AND node.

**Exercise 3.4** Consider the implementation of union-find data-structure where we use the path compression heuristic, but may not follow the union-by-rank heuristic. Can you prove a better bound than \( \log n \) for this heuristic?

**Exercise 3.5** Prove that Borůvka’s algorithm outputs an MST correctly.

**Exercise 3.6** Given an undirected graph \( G = (V, E) \), consider the subset system \( (E, M) \), where \( M \) consists of those subsets of edges which induce a subgraph of \( G \) with at most one cycle. Prove that this subset system is a matroid.
Exercise 3.7 Suppose you are given an MST for a graph. Now suppose we increase the weight of one of the edges $e$ in this MST from $w_e$ to $w'_e$. Give a linear time algorithm to find the new MST.

Exercise 3.8 Suppose $f : \mathbb{R} \to \mathbb{R}$ is differentiable. Prove that $f$ is convex if and only if for every pair of points $x, y \in \text{dom}(f)$,
\[ f(y) \leq f(x) + (y - x) \cdot f'(x), \]  
(3.6.5)
where $f'(x)$ denotes the derivative of $f$.

Exercise 3.9 Suppose $f : \mathbb{R} \to \mathbb{R}$ is twice differentiable. Prove that $f$ is convex if and only if $f''(x) \geq 0$ for all $x \in \text{dom}(f)$. Use this to prove that the functions $x^2, e^x$ and $e^{x^2}$ are convex.

Exercise 3.10 Consider the following functions on $n$ variables $x_1, \ldots, x_n$: (i) $a_1 x_1 + \ldots + a_n x_n$, where $a_1, \ldots, a_n$ are constants, (ii) $\log(e^{x_1} + \ldots + e^{x_n})$, (iii) $x_1^2 + x_2^2 + \ldots + x_n^2$. Prove that these functions are convex.

Exercise 3.11 Let $f$ and $h$ be defined as above. Prove that $h'(t) = d^T \nabla f(x_0 + td)$. Conclude that a differentiable function $f$ is convex if and only if for all points $x, y \in \text{dom}(f)$,
\[ f(y) \geq f(x) + (y - x)^T \nabla f(x) \]  
(3.6.6)
Suppose the second derivative, i.e., the Hessian of $f$, denoted by $H(x)$ exists. Show that $f$ is convex if and only if the matrix $H(x)$ is positive semi-definite at all points $x$ in the domain of $f$.

Exercise 3.12 Show that a strictly convex function has a unique local minimum in its domain.

Exercise 3.13 Consider the convex function $f(x) = x^2 - 1$. Starting from $x_0 = 1$, run the gradient descent algorithm with $\eta = 0.1$ for 10 steps (you may need to use a calculator). How close is the final estimate to the minimizer of $f$?

Exercise 3.14 Consider the function $f(x) = |x|$ and $\eta = 0.1$ with starting point 3.05. Show that the gradient descent algorithm will never converge to the minimum.

---

Suppose the second derivative, i.e., the Hessian of $f$, denoted by $H(x)$ exists. Show that $f$ is convex if and only if the matrix $H(x)$ is positive semi-definite at all points $x$ in the domain of $f$. An $m \times m$ matrix $H$ is said to be positive semi-definite if $x^T H x \geq 0$ for all vectors $x$. 

---

60
Chapter 4

Optimization II:
Dynamic Programming

Let us try to solve the knapsack problem using a somewhat different strategy. Let $F_i(y)$ denote the optimal solution for a knapsack capacity $y$ and using only the objects in $\{x_1, x_2 \ldots x_i\}$. Under this notation, $F_n(M)$ is the final solution to the knapsack problem with $n$ objects and capacity $M$. Let us further assume that all the weights are integral as also $M$. We can write the following equation

$$F_i(y) = \max\{F_{i-1}(y), F_{i-1}(y - w_i) + p_i\}$$

where the two terms correspond to inclusion or exclusion of object $i$ in the optimal solution. Also note that, once we decide about the choice of $x_i$, the remaining choices must be optimal with respect to the remaining objects and the residual capacity of the knapsack.

We can represent the above solution in a tabular form, where the rows correspond to the residual capacity from 1 to $M$ and the column $i$ represents the choice of objects restricted to the subset $\{1, 2 \ldots i\}$.

The first column corresponds to the base case of the subset containing only object $\{x_1\}$ and varying the capacity from 1 to $M$. Since the weight of the object is $w_1$, for all $i < w_1$, $F_i(i) = 0$ and $p_1$ otherwise. From the recurrence, it is clear that the $i$-th column can be filled up from the $(i - 1)$-st column and therefore after having computed the entries of column 1, we can successively fill up all the columns (till $n$). The value of $F_n(M)$ is readily obtained from the last column.

The overall time required to fill up the table is proportional to the size of the table multiplied by the time to compute each entry. Each entry is a function of two previously computed terms and therefore the total running time is $O(n \cdot M)$.

**Comment** The running time ($nM$) should be examined carefully. $M$ is the capacity of knapsack, for which $\log M$ bits are necessary for its representation. For the
remaining input data about \( n \) objects, let us assume that we need \( b \cdot n \) bits where \( b \) is the number of bits per object. This makes the input size \( N = b \cdot n + \log M \) so if \( \log M = N/2 \) then the running time is clearly exponential (\( M = 2^{N/2} \)).

### 4.1 A generic dynamic programming formulation

We begin with a recurrence (or an inductive) relation. In a typical recurrence, you may find repeated subproblems as we unfold the recurrence relation. There is an interesting property that the dynamic programming problems satisfy. The overall optimal solution can be described in terms of optimal solution of subproblems. This is sometimes known as **optimal substructure** property. This is what enables us to write an appropriate recurrence for the optimal solution.

Following this, we describe a table that contains the solutions to the various subproblem. Each entry of the table \( T \) must be computable using only the previously computed entries. This sequencing is very critical to carry the computation forward. The running time is proportional to

\[
\sum_{s \in T} t(s) \quad \text{where } t(s) \text{ is the time time to compute an entry } s
\]

In the knapsack problem \( t(s) = O(1) \). The space bound is proportional to part of table that must be retained to compute the remaining entries. This is where we can make substantial savings by sequencing the computation cleverly. Dynamic programming is often seen as a trade-off between space and running time, where we are reducing the running time at the expense of extra space. By storing the solutions of the repeated subproblems, we save the time for recomputation. For the knapsack problem, we only need to store the previous column - so instead of \( M \cdot n \) space, we can do with \( O(n) \) space.

### 4.2 Illustrative examples

#### 4.2.1 Context Free Parsing

Given a context free grammar \( G \) in a Chomsky Normal Form (CNF) and a string \( X = x_1 x_2 \ldots x_n \) over some alphabet \( \Sigma \), we want to determine if \( X \) can be derived from the grammar \( G \).

A grammar in CNF has the following production rules

\[
A \rightarrow BC \quad A \rightarrow a
\]
where $A, B, C$ are non-terminals and $a$ is a terminal (symbol of the alphabet). All derivations must start from a special non-terminal $S$ which is the start symbol. We will use the notation $S \Rightarrow \alpha$ to denote that $S$ can derive the sentence $\alpha$ in finite number of steps by applying production rules of the grammar.

The basis of our algorithm is the following observation

**Observation 4.1** $A \Rightarrow x_ix_{i+1}...x_k$ iff $A \Rightarrow BC$ and there exists a $i < j < k$ such that $B \Rightarrow x_ix_{i+1}...x_j$ and $C \Rightarrow x_{j+1}...x_k$.

There are $k - 1$ possible partitions of the string and we must check for all partitions if the above condition is satisfied. More generally, for the given string $x_1x_2...x_n$, we consider all substrings $X_{i,k} = x_ix_{i+1}...x_k$ where $1 \leq i < k \leq n$ - there are $O(n^2)$ such substrings. For each substring, we try to determine the set of non-terminals $A$ that can derive this substring. To determine this, we use the the previous observation. Note that both $B$ and $C$ derive substrings that are strictly smaller than $X_{i,j}$. For substrings of length one, it is easy to check which non-terminals derive them, so these serve as base cases.

We define a two dimensional table $T$ such that the entry $T(s, t)$ corresponds to all non-terminals that derive the substring starting at $x_s$ of length $t$. For a fixed $t$, the possible values of $s$ are $1, 2, ... n - t + 1$ which makes the table triangular. Each entry in the table can be filled up in $O(t)$ time for column $t$. That yields a total running time of $\sum_{t=1}^{n} O((n - t) \cdot t)$ which is $O(n^3)$. The space required is the size of the table which is $O(n^2)$. This algorithm is known as CYK (Cocke-Young-Kassimi) after the discoverers.

### 4.2.2 Longest monotonic subsequence

**Problem** Given a sequence $S$ of numbers $x_1, x_2...x_n$ a subsequence $x_{i_1}, x_{i_2}...x_{i_k}$ where $i_{j+1} > i_j$ is monotonic if $x_{i_{j+1}} \geq x_{i_j}$. We want to find the longest (there may be more than one) monotonic subsequence.

**Exercise 4.1** For any sequence of length $n$ prove that either the longest increasing monotonic subsequence or the longest decreasing subsequence has length at least $\lceil \sqrt{n} \rceil$. This is known as the Erdos-Szekeres theorem.

The previous result is only an existential result but here we would like to find the actual sequence. Let us define the longest monotonic subsequence in $x_1, x_2...x_i$ ending at $x_i$ (i.e. $x_i$ must be the last element of the subsequence) as $S_i$. Clearly we are looking for $\max_{i=1}^{n} S_i$. With a little thought we can write a recurrence for $S_i$ as

$$S_i = \max_{j<i} \{S_j + 1 | x_j \leq x_i \}$$

\(^1\)We are using $S_i$ for both the length and the sequence interchangeably.

63
Computing $S_i$ takes $O(i)$ time and therefore, we can compute the longest monotonic subsequence in $O(n^2)$ steps. The space required is $O(n)$.

Can we improve the running time? For this, we will actually address a more general problem, namely for each $j$, we will compute a monotonic subsequence of length $j$ (if it exists). For each $i \leq n$, let $M_{i,j}$ denote a monotonic subsequence of length $j$ in $x_1 x_2 \ldots x_i$. Clearly, if $M_{i,j}$ exists then $M_{i,j-1}$ exists and the maximum length subsequence is

$$\max_j \{M_{n,j} | M_{n,j} \text{ exists} \}$$

Among all subsequences of length $j$, we will compute an $M_{i,j}'$ which has the minimum terminating value among all $M_{i,j}$. For example, among the subsequences 2,4,5,9 and 1,4,5,8 (both length 4), we will choose the second one, since 8 < 9.

Let $\ell_{i,j}$ be the last element of $M_{i,j}'$. Here is a simple property of the $\ell_{i,j}$'s that can be proved by contradiction.

**Observation 4.2** The $\ell_{i,j}$'s form a non-decreasing sequence in $j$ for any fixed $i$.

By convention we will implicitly initialise all $\ell_{i,j} = \infty$. We can write a recurrence for $\ell_{i,j}$ as follows

$$\ell_{i+1,j} = \begin{cases} x_{i+1} & \text{if } \ell_{i,j} - 1 \leq x_{i+1} < \ell_{i,j} \\ \ell_{i,j} & \text{otherwise} \end{cases}$$

This follows, since, $M_{i+1,j}'$ is either $M_{i,j}'$ or $x_{i+1}$ must be the last element of $M_{i+1,j}'$ and in the latter case, it must satisfy the previous observation. This paves the way for updating and maintaining the information about $\ell_{i,j}$ in a compact manner, namely by maintaining a sorted sequence of $\ell_{i,j}$ such that when we scan $x_{i+1}$, we can quickly identify $\ell_{i,k}$ such that $\ell_{i,k-1} \leq x_{i+1} < \ell_{i,k}$. Note that this becomes $M_{i+1,k}'$ whereas for all $j \neq k$, $M_{i+1,j}' = M_{i,j}'$ (Why?). We can maintain reconstruct $M_{i,j}'$s by maintaining predecessor information. We can easily maintain the $\ell_{i,j}$'s in a dynamic dictionary data structure (like AVL tree) $O(\log n)$ time. Therefore, the total running time reduces to $O(n \log n)$.

**Remark:** If you could design a data structure that would return the maximum value of $S_j$ for all $x_j \leq x_i$ in $O(\log n)$ time then the first approach itself would be as good as the second one. Note that this data structure must support insertion of new points as we scan from left to right. You may want to refer to Section 6.3 for such a data structure.

### 4.2.3 Function approximation

Consider an integer valued function $h(i)$ on integers $\{1, 2 \ldots n\}$. We want to define another function $g(i)$ with a maximum of $k$ steps $k \leq n$ such that the difference
between \(g(i)\) and \(h(i)\), \(\Delta(g, h)\) is minimized according to some measure. One of
the most common measures is the sum of the squares of the differences of the two
functions that we will denote by \(L_2^2\).

Let \(g_{i,j}^*\) denote the optimal \(i \leq k\)-step function for this problem restricted to the
points \(1, \ldots, j\) - we are interested in computing \(g_{k,n}^*\). Note that \(g_{i,j}^*\) for \(i \geq j\) is identical
to \(h\) restricted to points \(1, \ldots, j\).

**Exercise 4.2** Show that \(g_{1,j}^* = \frac{1}{n} \sum_{i=1}^j h(i)\), i.e., it is a constant function equal to
the mean.

Further show that \(L_2^2(h, g_{1}^* - \delta) = L_2^2(h, g_{1}^*) + \delta^2 \cdot n\), i.e., for \(\delta = 0\), the sum of squares
of deviation is minimized.

We can now write a recurrence for the \(g_{i,j}^*\) as follows - let \(t(i, j)\) denote the smallest \(s \leq j\) such that \(g_{i,j}^*\) is constant for values \(\geq s\), viz.,
\(t(i, j)\) is the last step of \(g_{i,j}^*\). Then

\[
t(i, j) = \min_{s<j} \{L_2^2(h, g_{i-1}^* - \delta) + D_{s,j}\}
\]

where \(D_{s,j}\) denotes the sum of squares of deviation of \(h()\) from its mean value in the
interval \([s, j]\) and the domain of \(h\) is restricted to \(1, \ldots, s\). We can now write

\[
g_{i,j}^*(s) = \begin{cases} g_{i-1}^*(t(i, \ell)) & s < t(i, \ell) \\ A_{t(i, \ell)} & \text{otherwise} \end{cases}
\]

where \(A_{t,j}\) denotes the from the mean value of \(h\) in the interval \([i, j]\).

The recurrence captures the property that an optimal \(k\) step approximation can be
expressed as an optimal \(k - 1\) step approximation till an intermediate point followed
by the best 1 step approximation of the remaining interval (which is the mean value
in this interval from our previous observation). Assuming that \(D_{j,\ell}\) are precomputed
for all \(1 \leq j < \ell \leq n\), we can compute the \(g_{i,j}^*\) for all \(1 \leq i \leq k\) and \(1 \leq j \leq n\) in a
table of size \(kn\). The entries can be computed in increasing order of \(i\) and thereafter
in increasing order of \(j\)'s. The base case of \(i = 1\) can be computed directly from the
result of the previous exercise. We simultaneously compute \(t(i, j)\) and the quantity
\(L_2^2(h, g_{i,j}^*)\). Each entry can be computed from \(j - 1\) previously computed entries
yielding a total time of

\[
\sum_{i=1}^{i=k} \sum_{j=1}^{n} O(j) = O(k \cdot n^2)
\]

The space required is proportional to the previous row (i.e. we need to keep track of
the previous value of \(i\)), given that \(D_{j,\ell}\) can be stored/computed quickly. Note that a
\(i\)-step function can be stored as an \(i\)-tuple, so the space in each row is \(O(k \cdot n)\), since
\(i \leq k\).
Exercise 4.3 Complete the analysis of the above algorithm considering the computation of the $D_{i,j}$.

Exercise 4.4 Instead of partitioning $g_{i,j}^*$ in terms of an optimal $i - 1$ step approximation and a $1$ step (constant) approximation, you can also partition as $i'$ and $i - i'$ step functions for any $i - 1 \geq i' \geq 1$.

Can you analyze the algorithm for an arbitrary $i'$?

4.2.4 Viterbi’s algorithm for Maximum likelihood estimation

In this problem we have a weighted directed graph $G = (V, E)$ where the weights are related to probabilities and the sum of the probabilities on outgoing edges from any given vertex is 1. Further, the edges are labelled with symbols from an alphabet $\Sigma$ - note that more than one edge can share the same label. Given a string $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ over $\Sigma$, find the most probable path in the graph starting at $v_o$ with label equal to $\sigma$. The label of a path is the concatenation of labels associated with the edges. To find the most probable path, we can actually find the path that achieves the maximum probability with label $\sigma$. By assuming independence between successive edges, we want to choose a path that maximizes the product of the probabilities. Taking the log of this objective function, we can instead maximize the sum of the probabilities. So, if the weights are negative logarithms of the probability - the objective is to minimize the sum of the weights of edges along a path (note that log of probabilities are negative numbers).

We can write a recurrence based on the following observation.

The optimal least-weight path $x_1, x_2 \ldots x_n$ starting at vertex $x_1$ with label $\sigma_1\sigma_2\ldots\sigma_n$ is such that the path $x_2x_3 \ldots x_n$ is optimal with respect to the label $\sigma_2, \sigma_3\ldots\sigma_n$. For paths of length one, it is easy to find the optimal labelled path. Let $P_{i,j}(v)$ denote the optimal labelled path for the labels $\sigma_i\sigma_{i+1}\ldots\sigma_j$ starting at vertex $v$. We are interested in $P_{1,n}(v_o)$.

$$P_{i,j}(v) = \min_w \{P_{i+1,j}(w) | \text{label of } (v, w) = \sigma_i\}$$

Starting from the base case of length one paths, we build length 2 paths from each vertex and so on. Note that the length $i + 1$ paths from a vertex $v$ can be built from length $i$ paths from $w$ (computed for all vertices $w \in V$). The paths that we compute are of the form $P_{i,n}$ for all $1 \leq i \leq n$. Therefore we can compute the entries of the table starting from $i = n - 1$. From the previous recurrence, we can now compute the entries of the $P_{n-2,n}$ etc. by comparing at most $|V|$ entries (more specifically the outdegree ) for each starting vertex $v$.

Given that the size of table is $n \cdot |V|$, the total complexity is $O(n \cdot |V|)$ steps. You can argue that each iteration takes $O(|E|)$ steps where $|E|$ is the number of edges.
time required to compute all the entries is $O(n \cdot |V|^2)$. However, the space requirement can be reduced to $O(|V|)$ from the observation that only the $(i - 1)$ length paths are required to compute the optimal $i$ length paths.
Chapter 5
Searching

5.1 Skip Lists - a simple dictionary

Skip-list is a data structure introduced by Pugh\(^1\) as an alternative to balanced binary search trees for handling dictionary operations on ordered lists. The underlying idea is to substitute complex book-keeping information used for maintaining balance conditions for binary trees by random sampling techniques. It has been shown that, given access to random bits, the expected search time in a skip-list of \( n \) elements is \( O(\log n) \) \(^2\) which compares very favourably with balanced binary trees. Moreover, the procedures for insertion and deletion are very simple which makes this data-structure a very attractive alternative to the balanced binary trees.

Since the search time is a stochastic variable (because of the use of randomization), it is of considerable interest to determine the bounds on the tails of its distribution. Often, it is crucial to know the behavior for any individual access rather than a chain of operations since it is more closely related to the real-time response.

5.1.1 Construction of Skip-lists

This data-structure is maintained as a hierarchy of sorted linked-lists. The bottom-most level is the entire set of keys \( S \). We denote the linked list at level \( i \) from the bottom as \( L_i \) and let \( |L_i| = N_i \). By definition \( L_0 = S \) and \( |L_0| = n \). For all \( 0 \leq i, L_i \subset L_{i-1} \) and the topmost level, say level \( k \) has constant number of elements. Moreover, correspondences are maintained between common elements of lists \( L_i \) and \( L_{i-1} \). For a key with value \( E \), for each level \( i \), we denote by \( T_i \) a tuple \((l_i, r_i)\) such that \( l_i \leq E \leq r_i \) and \( l_i, r_i \in L_i \). We call this tuple straddling pair (of \( E \)) in level \( i \).

\(^1\)William Pugh. Skip list a probabilistic alternative to balanced trees. CACM June 1990 Vol33 Num6 668-676, 1990
\(^2\)Note that all logarithms are to base 2 unless otherwise mentioned.
The search begins from the topmost level $L_k$ where $T_k$ can be determined in constant time. If $l_k = E$ or $r_k = E$ then the search is successful else we recursively search among the elements $[l_k, r_k] \cap L_0$. Here $[l_k, r_k]$ denotes the closed interval bound by $l_k$ and $r_k$. This is done by searching the elements of $L_{k-1}$ which are bounded by $l_k$ and $r_k$. Since both $l_k, r_k \in L_{k-1}$, the descendence from level $k$ to $k - 1$ is easily achieved in $O(1)$ time. In general, at any level $i$ we determine the tuple $T_i$ by walking through a portion of the list $L_i$. If $l_i$ or $r_i$ equals $E$ then we are done else we repeat this procedure by descending to level $i - 1$.

In other words, we refine the search progressively until we find an element in $S$ equal to $E$ or we terminate when we have determined $(l_0, r_0)$. This procedure can also be viewed as searching in a tree that has variable degree (not necessarily two as in binary tree).

Of course, to be able to analyze this algorithm, one has to specify how the lists $L_i$ are constructed and how they are dynamically maintained under deletions and additions. Very roughly, the idea is to have elements in $i$-th level point to approximately $2^i$ nodes ahead (in $S$) so that the number of levels is approximately $O(\log n)$. The time spent at each level $i$ depends on $[l_{i+1}, r_{i+1}] \cap L_i$ and hence the objective is to keep this small. To achieve these conditions on-line, we use the following intuitive method. The nodes from the bottom-most layer (level 0) are chosen with probability $p$ (for the purpose of our discussion we shall assume $p = 0.5$) to be in the first level. Subsequently at any level $i$, the nodes of level $i$ are chosen to be in level $i + 1$ independently with probability $p$ and at any level we maintain a simple linked list where the elements are in sorted order. If $p = 0.5$, then it is not difficult to verify that for a list of size $n$, the expected number of elements in level $i$ is approximately $n/2^i$ and are spaced about $2^i$ elements apart. The expected number of levels is clearly $O(\log n)$, (when we have just a trivial length list) and the expected space requirement is $O(n)$.

To insert an element, we first locate its position using the search strategy described previously. Note that a byproduct of the search algorithm are all the $T_i$’s. At level 0, we choose it with probability $p$ to be in level $L_1$. If it is selected, we insert it in the proper position (which can be trivially done from the knowledge of $T_1$), update the pointers and repeat this process from the present level. Deletion is very similar and it can be readily verified that deletion and insertion have the same asymptotic run time as the search operation. So we shall focus on this operation.

### 5.1.2 Analysis

To analyze the run-time of the search procedure, we look at it backwards, i.e., retrace the path from level 0. The search time is clearly the length of the path (number of links) traversed over all the levels. So one can count the number of links one traverses before climbing up a level. In other words the expected search time can be expressed

\[69\]
in the following recurrence

\[ C(k) = (1 - p)(1 + C(k)) + p(1 + C(k - 1)) \]

where \( C(k) \) is the expected cost for climbing \( k \) levels. From the boundary condition \( C(0) = 0 \), one readily obtains \( C(k) = k/p \). For \( k = O(\log n) \), this is \( O(\log n) \). The recurrence captures the crux of the method in the following manner. At any node of a given level, we climb up if this node has been chosen to be in the next level or else we add one to the cost of the present level. The probability of this event (climbing up a level) is \( p \) which we consider to be a success event. Now the entire search procedure can be viewed in the following alternate manner. We are tossing a coin which turns up heads with probability \( p \) - how many times should we toss to come up with \( O(\log n) \) heads? Each head corresponds to the event of climbing up one level in the data structure and the total number of tosses is the cost of the search algorithm. We are done when we have climbed up \( O(\log n) \) levels (there is some technicality about the number of levels being \( O(\log n) \) but that will be addressed later). The number of heads obtained by tossing a coin \( N \) times is given by a Binomial random variable \( X \) with parameters \( N \) and \( p \). Using Chernoff bounds (see Appendix, equation B.1.5), for \( N = 15 \log n \) and \( p = 0.5 \), \( \Pr[X \leq 1.5 \log n] \leq 1/n^2 \) (using \( \epsilon = 9/10 \) in equation 1). Using appropriate constants, we can get rapidly decreasing probabilities of the form \( \Pr[X \leq c \log n] \leq 1/n^\alpha \) for \( c, \alpha > 0 \) and \( \alpha \) increases with \( c \). These constants can be fine tuned although we shall not bother with such an exercise here.

We thus state the following lemma.

**Lemma 5.1** The probability that access time for a fixed element in a skip-list data structure of length \( n \) exceeds \( c \log n \) steps is less than \( O(1/n^2) \) for an appropriate constant \( c > 1 \).

**Proof** We compute the probability of obtaining fewer than \( k \) (the number of levels in the data-structure) heads when we toss a fair coin \( p = 1/2 \) \( c \log n \) times for some fixed constant \( c > 1 \). That is, we compute the probability that our search procedure exceeds \( c \log n \) steps. Recall that each head is equivalent to climbing up one level and we are done when we have climbed \( k \) levels. To bound the number of levels, it is easy to see that the probability that any element of \( S \) appears in level \( i \) is at most \( 1/2^i \), i.e. it has turned up \( i \) consecutive heads. So the probability that any fixed element appears in level \( 3 \log n \) is at most \( 1/n^3 \). The probability that \( k > 3 \log n \) is the probability that at least one element of \( S \) appears in \( L_{3 \log n} \). This is clearly at most \( n \) times the probability that any fixed element survives and hence probability of \( k \) exceeding \( 3 \log n \) is less than \( 1/n^2 \).

Given that \( k \leq 3 \log n \) we choose a value of \( c \), say \( c_0 \) (to be plugged into equation B.1.6 of Chernoff bounds) such that the probability of obtaining fewer than \( 3 \log n \)
heads in \(c_0 \log n\) tosses is less than \(1/n^2\). The search algorithm for a fixed key exceeds \(c_0 \log n\) steps if one of the above events fail; either the number of levels exceeds \(3 \log n\) or we get fewer than \(3 \log n\) heads from \(c_0 \log n\) tosses. This is clearly the summation of the failure probabilities of the individual events which is \(O(1/n^2)\).

\[\square\]

**Theorem 5.1** The probability that the access time for any arbitrary element in skip-list exceeds \(O(\log n)\) is less than \(1/n^\alpha\) for any fixed \(\alpha > 0\).

**Proof:** A list of \(n\) elements induces \(n + 1\) intervals. From the previous lemma, the probability \(P\) that the search time for a fixed element exceeding \(c \log n\) is less than \(1/n^2\). Note that all elements in a fixed interval \([l_0, r_0]\) follow the same path in the data-structure. It follows that for any interval the probability of the access time exceeding \(O(\log n)\) is \(n\) times \(P\). As mentioned before, the constants can be chosen appropriately to achieve this.

It is possible to obtain even tighter bounds on the space requirement for a skip list of \(n\) elements. We can show that the expected space is \(O(n)\) since the expected number of times a node survives is 2.

**Exercise 5.1** Prove the following stronger bound on space using Chernoff bounds - For any constant \(\alpha > 0\), the probability of the space exceeding \(2n + \alpha \cdot n\) is less than \(\exp^{\Omega(-\alpha^2n)}\).

### 5.2 Treaps: Randomized Search Trees

The class of binary (dynamic) search trees is perhaps the first introduction to non-trivial data-structure in computer science. However, the update operations, although asymptotically very fast are not the easiest to remember. The rules for rotations and the double-rotations of the AVL trees, the splitting/joining in B-trees and the color-changes of red-black trees are often complex, as well as their correctness proofs. The Randomized Search Trees (also known as randomized treaps) provide a practical alternative to the Balanced BST. We still rely on rotations, but no explicit balancing rules are used. Instead we rely on the magical properties of random numbers.

The Randomized Search Tree (RST) is a binary tree that has the keys in an in-order ordering. In addition, each element is assigned a priority (Wlog, the priorities are unique) and the nodes of the tree are heap-ordered based on the priorities. Simultaneously, the key values follow in-order numbering.

**Exercise 5.2** For a given assignment of priorities, show that there is a unique treap.

If the priorities are assigned randomly in the range \([1, N]\) for \(N\) nodes, the expected height of the tree is small. This is the crux of the following analysis of the performance of the RSTs.
Let us first look at the search time using a technique known as backward analysis. For that we (hypothetically) insert the \( N \) elements in a decreasing order of their priorities and then count the number of elements that an element \( Q \) can see during the course of their insertions. \( Q \) can see an element \( N_i \) if there are no previously inserted elements in between. This method (of assigning the random numbers on-line) makes arguments easier and the reader must convince himself that it doesn’t affect the final results.

**Claim 5.1** The tree constructed by inserting the nodes in order of their priorities (highest priority is the root) is the same as the tree constructed on-line.

This follows from the uniqueness of the treap.

**Exercise 5.3** Show that the number of nodes \( Q \) sees during the insertion sequence is exactly the number of comparisons performed for searching \( Q \). In fact, the order in which it sees corresponds to the search path of \( Q \).

**Theorem 5.2** The expected length of search path in RST is \( O(H_N) \) where \( H_N \) is the \( N \)-th harmonic number.

In the spirit of backward analysis, we pretend that the tree-construction is being reversed, i.e. nodes are being deleted starting from the last node. In the forward direction, we would count the expected number of nodes that \( Q \) sees. In the reverse direction, it is the number of times \( Q \)’s visibility changes (convince yourself that these notions are identical). Let \( X_i \) be a Bernoulli rv that is 1 if \( Q \) sees \( N_i \) (in the forward direction) or conversely \( Q \)’s visibility changes when \( N_i \) is deleted in the reverse sequence. Let \( X \) be the length of the search path. Then

\[
X = \sum X_i \quad \text{and} \quad E[X] = E[\sum X_i] = \sum E[X_i]
\]

We claim that \( E[X_i] = \frac{2}{i} \). Note that the expectation of a Bernoulli variable is the probability that it is 1. We are computing this probability over all permutations of \( N \) elements being equally likely. In other words, if we consider a prefix of length \( i \), all subsets of size \( i \) are equally likely. Let us find the probability that \( X_i = 1 \), conditioned on a fixed subset \( N^i \subset N \) consisting of \( i \) elements. Since, for a fixed \( N^i \), all \( N^{i-1} \) are equally likely, the probability that \( X_i = 1 \) is the probability that one of the (maximum two) neighboring elements was removed in the reverse direction. The probability of that is less than \( \frac{2}{i} \) which is independent of any specific \( N^i \). So, the unconditional probability is the same as conditional probability - hence \( E[X_i] = \frac{2}{i} \). The theorem follows as

\[
\sum E[X_i] = 2 \sum \frac{1}{i} = O(H_N).
\]

The \( X_i \)’s defined in the previous proof are nearly independent but not identical. We can obtain tail-estimates for deviation from the expected bound using a technique.
similar to Chernoff bounds. The proof is omitted but the interested reader may want to devise a proof based on Lemma B.3 in the Appendix.

**Theorem 5.3** The probability that the search time exceeds $2 \log n$ comparisons in a randomized trie is less than $O(1/n)$.

Insertions and deletions require changes in the tree to maintain the heap property and rotations are used to push up or push down some elements as per this need. A similar technique can be used for counting the number of rotations required for RST during insertion and deletions. Backward analysis is a very elegant technique for analyzing randomized algorithms, in particular in a paradigm called randomized incremental construction.

## 5.3 Universal Hashing

Hashing is often used as a technique to achieve $O(1)$ search time by fixing the location where a key is assigned. For the simple reason that the number of possible key values is much larger than the table size, it is inevitable that more than one key is mapped to the same location. The number of conflicts increase the search time. If the keys are randomly chosen, then it is known that the expected number of conflicts is $O(1)$. However this may be an unrealistic assumption, so we must design a scheme to handle any arbitrary subset of keys. We begin with some useful notations

- **Universe**: $\mathcal{U}$, Let the elements be $0, 1, 2, \ldots, N - 1$
- **Set of elements**: $S$ also $|S| = n$
- **Hash locations**: $\{0, 1, \ldots, m - 1\}$ usually, $n \geq m$

**Collision** If $x, y \in \mathcal{U}$ are mapped to the same location by a hash function $h$.

$$
\delta_h(x, y) = \begin{cases} 
1 : & h(x) = h(y), x \neq y \\
0 : & \text{otherwise}
\end{cases}
$$

$$
\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y)
$$

**Hash by chaining**: The more the collision the worse the performance. Look at a sequence $O_1(x_2), O_2(x_2), \ldots, O_n(x_n)$ where $O_i \in \{\text{Insert, Delete, Search}\}$ and $x_i \in \mathcal{U}$

Let us make the following assumptions

1. $|h^{-1}(i)| = |h^{-1}(i')|$ where $i, i' \in \{0, 1, \ldots, m - 1\}$
2. In the sequence, $x_i$ can be any element of $\mathcal{U}$ with equal probability.
Claim: Total expected cost = \( O((1 + \beta)n) \) where \( \beta = \frac{n}{m} \) (load factor).

Proof: Expected cost of \((k + 1)\)th operation = expected number of elements in location \( \leq 1 + k(\frac{1}{m}) \) assuming all the previous operations were Insert.

So total expected cost \( \leq \sum_{k=1}^{n} 1 + \frac{k}{m} = n + \frac{n(n+1)}{2m} = (1 + \frac{\beta}{2})n \). This is worst case over operations but not over elements. \( \square \)

Universal Hash Functions

Definition 5.1 A collection \( H \subset \{ h| h : [0...N-1] \rightarrow [0...m-1] \} \) is c-universal if for all \( x, y \in [0...N-1] \) \( x \neq y \),

\[
|\{ h | h \in H \text{ and } h(x) = h(y) \}| \leq c \frac{|H|}{m}
\]

for some small constant \( c \). Roughly \( \sum_{h} \delta_{h}(x, y) \leq c \frac{|H|}{m} \).

Claim: \[
\frac{1}{|H|} \sum_{h \in H} 1 + \delta_{h}(x, S) \leq 1 + \frac{c}{m} n
\]

where \( |S| = n \).

Proof: Working from the LHS, we obtain

\[
= \frac{1}{|H|} \sum_{h \in H} 1 + \frac{1}{|H|} \sum_{h} \sum_{y \in S} \delta_{h}(x, y)
\]

\[
= 1 + \frac{1}{|H|} \sum_{y} \sum_{h} \delta_{h}(x, y)
\]

\[
\leq 1 + \frac{1}{|H|} \sum_{y} c \frac{|H|}{m}
\]

\[
= 1 + \frac{c}{m} n
\]

So expected cost of \( n \) operation = \( \sum (1 + \frac{\delta_{i}}{m}) \leq (1 + c\beta)n \) \( \square \)

5.3.1 Example of a Universal Hash function

\( H' : h_{a,b}; h_{ab}(x) \rightarrow ((ax + b) \mod N) \mod m \) where \( a, b \in 0...N - 1 \) (\( N \) is prime).
If \( h_{ab}(x) = h_{ab}(y) \) then for some \( q \in [0...m-1] \) and \( r, s \in [0...N-1] \):

\[
ax + b = (q + rm) \mod N \\
ay + b = (q + sm) \mod N
\]

This is a unique solution for \( a, b \) once \( q, r, s \) are fixed. So there are a total of \( m(\frac{N^2}{m}) \) solutions = \( \frac{N^2}{m} \). Also, since \( |H'| = N^2 \), therefore \( H' \) is "1" universal.

**Exercise 5.4** Consider the hash function \( h(x) = a \cdot x \mod N \mod m \) for \( a \in \{0, 1\ldots(N-1)\} \). Show that it satisfies the properties of a universal family when \( m \) is prime.

*Hint:* For \( x \neq y \), \((x - y)\) has a unique inverse modulo \( m \).

### 5.4 Perfect Hash function

Universal hashing is very useful method but may not be acceptable in a situation, where we don’t want any conflicts. **Open addressing** is method that achieves this at the expense of increased search time. In case of conflicts, we define a sequence of probes that is guaranteed to find an empty location (if there exists one).

We will extend the scheme of universal hashing to one where there is no collision without increasing the expected search time. Recall that the probability that an element \( x \) collides with another element \( y \) is less than \( \frac{c}{m} \) for some constant \( c \). Therefore, the expected number of collisions in a subset of size \( n \) by considering all pairs is \( f = (\frac{n}{2}) \cdot \frac{c}{m} \). By Markov inequality, the probability that the number of collisions exceeds \( 2f \) is less than \( \frac{1}{2} \). For \( c = 2 \) and \( m \geq 4n^2 \), the value of \( 2f \) is less than \( \frac{1}{2} \), i.e. there are no collisions.

We use a two level hashing scheme. In the first level, we hash it to locations \( 1, 2 \ldots m \). If there are \( n_i \) keys that get mapped to location \( i \), we subsequently map them to \( 4n_i^2 \) locations. From our previous discussion, we know that we can avoid collisions with probability at least 1/2. So we may have to repeat the second level hashing a number of times (expected value is 2) before we achieve zero collision for the \( n_i \) keys. So the search time is \( O(1) \) total expected for both levels.

The space bound is \( 4 \sum n_i^2 \). We can write

\[
n_i^2 = 2 \sum_{x,y| h(x)=h(y)=i} 1 + n_i.
\]
\[ \sum_{i} n_i^2 = \sum_{i} n_i + 2 \left( \sum_{x,y \mid h(x)=h(y)=i} 1 \right) \]

\[ = 2 \sum_{i} n_i + 2 \sum_{i \mid x,y \mid h(x)=h(y)=i} 1 \]

\[ = 2n + \sum_{x,y} \delta(x,y) \]

Taking expectation on both sides (with respect to choice of a random hash function), the R.H.S. is \(2E[\sum_{x,y\in S} \delta(x,y)] + n.\) This equals \(2 \cdot \binom{n}{2} \cdot \frac{n}{m}\) since \(E[\delta(x,y)] = \Pr[h(x) = h(y)] \leq \frac{c}{m}.\) Therefore the total expected space required is only \(O(n)\) for \(m \in O(n)\).

### 5.4.1 Converting expected bound to worst case bound

We can convert the expected space bound to worst case space bound in the following manner. In the first level, we repeatedly choose a hash function until \(\sum_{i} n_i^2\) is \(O(n)\). We need to repeat this twice in the expected sense. Subsequently at the second stage, for each \(i\), we repeat it till there are no collisions in mapping \(n_i\) elements in \(O(n_i^2)\) locations. Again, the expected number of trials for each \(i\) is two that takes overall \(O(n)\) time for \(n\) keys. Note that this method makes the space worst case \(O(n)\) at the expense of making the time expected \(O(n).\) But once the hash table is created, for any future query the time is worst case \(O(1).\)

For practical implementation, the \(n\) keys will be stored in a single array of size \(O(n)\) where the first level table locations will contain the starting positions of keys with value \(i\) and the hash function used in level 2 hash table.

### 5.5 A \(\log \log N\) priority queue

Searching in bounded universe is faster by use of hashing. Can we achieve similar improvements for other data structures? Here we consider maintaining a priority queue for elements drawn from universe \(U\) and let \(|U| = N.\) The operations supported are insert, minimum and delete.

Imagine a complete binary tree on \(N\) leaf nodes that correspond to the \(N\) integers of the universe - this tree has depth \(\log N.\) Let us colour the leaf nodes of the tree black if the corresponding integer is present in the set \(S \subset U\) where \(|S| = n.\) Let us also imagine that if a leaf node is coloured then its half-ancestor (halfway from the node to the root) is also coloured and is labelled with the smallest and the largest integer in its subtree. Denote the set of the minimum elements by \(TOP\) and we recursively build a data structure on the elements of \(TOP\) which has size at most
We will denote the immediate predecessor of an element $x$ by $PRED(x)$ and the successor of an element by $SUCC(x)$. The reason we are interested in $PRED$ and $SUCC$ is that when the smallest element is deleted, we must find its immediate successor in set $S$. Likewise, when we insert an element, we must know its immediate predecessor. Henceforth we will focus on the operations $PRED$ and $SUCC$ as these will be used to support the priority queue operations.

For a given element $x$, we will check if its ancestor at depth $\log \frac{N}{2}$ (halfway up the tree) is colored. If so, then we search $PRED(x)$ within the subtree of size $\sqrt{N}$. Otherwise, we search for $PRED(x)$ among the elements of $TOP$. Note that either we search within the subtree or in the set $TOP$ but not both. Suitable terminating conditions can be defined. The search time can be captured by the following recurrence

$$T(N) = T(\sqrt{N}) + O(1)$$

which yields $T(N) = O(\log \log N)$. The $TOP$ data structure is built on keys of length $\log \frac{N}{2}$ higher order bits and the search structure is built on the lower $\log \frac{N}{2}$ bits. The space complexity of the data structure satisfies the recurrence

$$S(m) = (\sqrt{m} + 1)S(\sqrt{m}) + O(\sqrt{m})$$

which yields $S(N) = O(N \log \log N)$. The additive term is the storage of the elements of $TOP$.

**Exercise 5.5** Propose a method to decrease the space bound to $O(N)$ - you may want to prune the lower levels of the tree.

For the actual implementation, the tree structure is not explicitly built - only the relevant locations are stored. To insert an element $x$ in this data-structure, we first find its predecessor and successor. If it is the first element in the subtree then the ancestor at level $\log \frac{N}{2}$ is appropriately initialized. $SUCC(PRED(x)) \leftarrow x$ where $PRED(x)$ is in a different subtree. Notice that we do not have to process within the subtree any further. Otherwise, we find its $PRED$ within the subtree. It may appear that the time for insertion is $(\log \log^2 N)$ since it satisfies the recurrence

$$I(N) = I(\sqrt{N}) + O(\log \log N) \text{ or } I(N) \in \log \log^2 N$$

To match the $O(\log \log N)$ time bound for searching, we will actually do the search for $PRED$ simultaneously with the insert operation.

**Exercise 5.6** Show how to implement delete operation in $O(\log \log N)$ steps.
One of the most serious drawbacks of this data structure is the high space requirement, which is proportional to the size of the universe. Note that, as long as \( \log \log N \in o(\log n) \), this is faster than the conventional heap. For example, when \( N \leq 2^{\log n / \log \log n} \), this holds an advantage, but the space is exponential. However, we can reduce the space to \( O(n) \) by using the techniques mentioned in this chapter to store only those nodes that are coloured black.
Chapter 6
Multidimensional Searching and Geometric algorithms

Searching in a dictionary is one of the most primitive kind of search problem and it is relatively simple because of the property that the elements can be ordered. Instead, suppose that the points are from $d$ dimensional space $\mathbb{R}^d$ - for searching we can build a data-structure based on lexicographic ordering. That is, given two $d$ dimensional points $p$ and $p'$, there is a total ordering defined by the relation

$$p < p' \iff \exists j \leq d : x_1(p) = x_1(p'), x_2(p) = x_2(p') \ldots x_{j-1}(p) = x_{j-1}(p')x_j(p) < x_j(p')$$

where $x_i(p)$ is the $i$-th coordinate of point $p$. For example, if $p = (2.1, 5.7, 3.1)$ and $p' = (2.1, 5.7, 4)$ then $p < p'$. If we denote a $d$ dimensional point by $(x_0, x_1 \ldots x_d)$ then, given a set of $n$ d-dimensional points, the immediate predecessor of a query point can be determined in $O(d \cdot \log n)$ comparisons using a straightforward adaption of the binary search. With a little more thought we can try to improve as follows.

When two $d$-tuples are compared, we can keep track of the maximum prefix length (the index $j$) which is identical in the two tuples. If we create a BST to support binary search, we also keep track of the largest common prefixes between the parent and the children nodes, so that we can find the common prefixes between the query tuple and the internal nodes in a way that we do not have to repeatedly scan the same coordinates of the query tuple. For example, if the root node has a common prefix of length 10 with the left child and the query tuple has 7 fields common with the root node (and is smaller), it is clearly smaller than the left child also. The reader is encouraged to solve Exercise 6.1 at the end of this chapter.

Queries can be far more sophisticated than just point-queries but we will focus on the simpler case.
6.1 Interval Trees and Range Trees

One Dimensional Range Searching

Given a set $S$ of $n$ points on a line (wlog, say the $x$-axis), we have to build a data-structure to report the points inside a given query interval $[x_\ell : x_u]$. The counting version of range query only reports the number of points in this interval instead of the points themselves.

Let $S = \{p_1, p_2, \ldots, p_n\}$ be the given set of points on the real line. We can solve the one-dimensional range searching problem using a balanced binary search tree $T$ in a straightforward manner. The leaves of $T$ store the points of $S$ and the internal nodes of $T$ store splitters to guide the search. If the splitter-value at node $v$ is $x_v$, the left subtree $L(v)$ of a node $v$ contains all the points smaller than or equal to $x_v$ and right subtree $R(v)$ contains all the points strictly greater than $x_v$. It is easy to see that we can build a balanced tree using median as the splitter value.

To report the points in the range query $[x_\ell : x_u]$ we search with $x_\ell$ and $x_u$ in the tree $T$. Let $\ell_1$ and $\ell_2$ be the leaves where the searches end. Then the points in the interval $[x_\ell : x_u]$ are the points stored between the leaves $\ell_1$ and $\ell_2$. Another way to view the set of points is the union of the leaves of some subtrees of $T$. If you examine the search path of $x_\ell$ and $x_u$, they share a common path from root to some vertex (may be the root itself), where the paths fork to the left and right - let us call this the fork node. The leaf nodes correspond to the union of the right subtrees of the left path and the left subtrees of the right path. This can be formally shown to be the union of at most $2 \log n$ subtrees (Exercise 6.2).

Complexity The tree uses $O(n)$ space and it can be constructed in $O(n \log n)$ time. Each query takes $O(\log n + k)$ time, where $k$ is the number of points in the interval, i.e., the output size. The counting query takes $O(\log n)$ time. This is clearly the best we can hope for.

You can hypothetically associate a half-open interval with each internal node $x$, $(l(x) : r(x))$ where $l(x)$ ($r(x)$) is the value of the leftmost (rightmost) leaf node in the subtree rooted at $x$.

6.1.1 Two Dimensional Range Queries

Each point has two attributes: its $x$ coordinate and its $y$ coordinate - the two dimensional range query is a Cartesian product of two one-dimensional intervals. Given a query $[x_\ell : x_u] \times [y_\ell : y_u]$ (a two dimensional rectangle), we want to build a data structure to report the points inside the rectangular region (or alternately, count the points in the region.)

We extend the previous one dimensional solution by first considering the vertical
slab \([x_l : x_u]\) \(^1\). Let us build the one-dimensional range tree identical to the previous scheme (by ignoring the \(y\) coordinates of points). Therefore we can obtain the answer to the slab-query. As we had observed every internal node represents an interval in the one dimensional case and analogously the corresponding vertical slab in the two dimensional case. The answer to the original query \([x_l : x_u] \times [y_l : y_u]\) is a subset of \([x_l : x_u] \times [-\infty : +\infty]\). Since our objective is to obtain a time bound proportional to the final output, we cannot afford to list out all the points of the vertical slab. However, if we had the one dimensional data structure available for this slab, we can quickly find out the final points by doing a range query with \([y_l : y_u]\). A naive scheme will build the data structure for all possible vertical slabs but we can do much better using the following observation.

**Observation 6.1** Each vertical slab is a union of \(2\log n\) canonical slabs.

It follows from the same argument as any interval. Each canonical slab corresponds to the vertical slab (the corresponding \([x_l : x_u]\)) spanned by an internal node. We can therefore build a one-dimensional range tree for all the points spanned by corresponding vertical slab - this time in the \(y\)-direction. So the final answer to the two dimensional range query is the union of at most \(2\log n\) one-dimensional range query, giving a total query time of \(\sum_{i=1}^{t} O(\log n + k_i)\) where \(k_i\) is the number of output points in slab \(i\) among \(t\) slabs and \(\sum_i k_i = k\). This results in a query time of \(O(t \log n + k)\) where \(t\) is bounded by \(2\log n\).

The space is bounded by \(O(n \log n)\) since in a given level of the tree \(T\), a point is stored exactly once. The natural extension of this scheme leads us to \(d\)-dimensional range search trees with the following performance parameters.

\[
Q(d) \leq \begin{cases} 
2 \log n \cdot Q(d-1) & \text{for } d \geq 2 \\
O(\log n) & \text{for } d = 1
\end{cases} \tag{6.1.1}
\]

where \(Q(d)\) is the query time in \(d\) dimensions for \(n\) points. This yields \(Q(d) = O(2^d \cdot \log^d n)\). A more precise recurrence can be written in terms of \(n, d\).

\[
Q(n, d) \leq \begin{cases} 
2 \sum_{i} Q\left(\frac{n}{2^i}, d-1\right) & \text{for } d \geq 2 \\
O(\log n) & \text{for } d = 1
\end{cases} \tag{6.1.2}
\]

since the number of points in a node at distance \(i\) from the root has at most \(\frac{n}{2^i}\) points.

The reader may want to find a tight solution of the above recurrence (Exercise 6.6). The number of output points \(k\) can be simply added to \(Q(n, d)\) since the subproblems output disjoint subsets.

\(^1\)You can think about the \([y_l : y_u]\) as \([-\infty : +\infty]\)
Figure 6.1: The rectangle is the union of the slabs represented by the darkened nodes plus an overhanging left segment containing $p_6$. The sorted list of points in $y$ direction is indicated next to the nodes - not all the lists are shown. The number inside the node of a tree indicates the splitting coordinate of the interval that defines the left and right subintervals corresponding to the left and right children.

### 6.2 k-d trees

A serious drawback of range trees is that both the space and the query time increases exponentially with dimensions. Even for two dimensions, the space is super-linear. For many applications, we cannot afford to have such a large blow-up in space (for a million records $\log n = 20$).

We do a divide-and-conquer on the set of points - we partition the space into regions that contain a subset of the given set of points. The input rectangle is tested against all the regions of the partition. If it doesn’t intersect a region $U$ then we do not search further. If $U$ is completely contained within the rectangle then we report all the points associated with $U$ otherwise we search recursively in $U$. We may have to search in more than one region - we define a search tree where each region is associated with a node of the tree. The leaf nodes correspond to the original point
Procedure Search($Q, v$)

1. if $R(v) \subset Q$ then
2. report all points in $R(v)$
else
3. Let $R(u)$ and $R(w)$ be rectangles associated with the children $u, w$;
4. if $Q \cap R(u)$ is non-empty then
   Search($Q, u$)
5. if $R \cap R(w)$ is non-empty then
   Search($Q, w$)

Figure 6.2: Rectangular range query using in a $k - d$ tree

set. In general, this strategy will work for other (than rectangular) kinds of regions also.

For the rectangular query, we split on $x$-coordinate and next on $y$-coordinate, then alternately on each coordinate. We partition with a vertical line at nodes whose depth is even and we split with a horizontal line at nodes whose depth is odd. The time to build the 2-D tree is as follows.

The region $R(v)$ corresponding to a node $v$ is a rectangle which is bounded by horizontal and vertical lines and it is a subset of the parent node. The root of a tree is associated with a (bounded) rectangle that contains all the $n$ points. We search a subtree rooted at $v$ iff the query rectangle intersects the associated with node $v$. This involves testing if two rectangles (the query rectangle and $R(v)$) overlap that can be done in $O(1)$ time. We traverse the 2-D tree, but visit only nodes whose region is intersected by the query rectangle. When a region is fully contained in the query rectangle, we can report all the points stored in its subtree. When the traversal reaches a leaf, we have to check whether the point stored at the leaf is contained in the query region and, if so, report it.

Since a point is stored exactly once and the description of a region corresponding to a node takes $O(1)$ space, the total space taken up the search tree is $O(n)$. Figure 6.3 illustrates $k - d$ tree data structure.
Figure 6.3: Each rectangular subdivision corresponds to a node in the \( k \)-\( d \) tree and is labelled by the splitting axis - either vertical or horizontal. The shaded nodes indicate the nodes visited due to the query rectangle. The leaf nodes are represented by the black squares - a leaf node is visited if the parent was visited and is associated exactly one of the given points.

**Query Time** Let \( Q(i) \) be the number of nodes at distance \( i \) from the root that are visited in the worst case by a rectangular query. Since a vertical segment of \( Q \) intersects only horizontal partitioning edges, we can write a recurrence for \( Q(i) \) by observing that the number of nodes can increase by a factor 2 by descending 2 levels.
Hence $Q(i)$ satisfies the recurrence

$$Q(i + 2) \leq 2Q(i)$$

which one can verify to be $Q(i) \in O(2^{i/2})$ or total number of nodes visited in the last level is $O(\sqrt{n})$.

### 6.3 Priority Search Trees

The combination of BST with heap property resulted in a simple strategy for maintaining balanced search trees called treaps. The heap property was useful to keep a check on the expected height of the tree within $O(\log n)$. What if we want to maintain a heap explicitly on set of parameters (say the $y$ coordinates) along with a total ordering required for binary search on the $x$ coordinates. Such a data structure would be useful to support a *three sided* range query in linear space.

A *three sided* query is a rectangle $[x_\ell : x_u] \times [y_\ell : \infty]$, i.e. a half-infinite vertical slab.

If we had a data structure that is a BST on $x$ coordinates, we can first locate the two points $x_\ell$ and $x_u$ to determine (at most) $2\log n$ subtrees whose union contains the points in the interval $[x_\ell : x_u]$. Say, these are $T_1, T_2 \ldots T_k$. Within each such tree $T_i$, we want to find the points whose $y$ coordinates are larger than $y_i$. If $T_i$ forms a max-heap on the $y$ coordinates then we can output the points as follows -

#### Procedure Search($v$)

1. Let $v_y$ denote the $y$ coordinate associated with a node $v$;
2. if $v_y < y_\ell$ or $v$ is a leaf node then
3. terminate search
4. else
5. if $v_x \geq x_\ell$ then
6. Output the point associated with $v$;
7. Search($u$) where $u$ is the left child of $v$;
8. Search ($w$) where $w$ is the right child of $v$;

Since $v$ is a root of max-heap, if $v_y < y_\ell$, then all the descendents of $v$ and therefore we do not need to search any further. This establishes correctness of the search procedure. Let us mark all the nodes that are visited by the procedure in the second phase. When we visit a node in the second phase, we either output a point or terminate the search. For the nodes that are output, we can charge it to the output size. For the nodes that are not output, let us add a charge to its parent - the maximum charge to a node is two because of its two children. The first phase takes
$O(\log n)$ time to determine the canonical sub-intervals and so the total search time is $O(\log n + k)$ where $k$ is number of output points$^2$.

Until now, we assumed that such a dual-purpose data structure exists. How do we construct one?

First we can build a leaf based BST on the $x$ coordinates. Next, we promote the points according to the heap ordering. If a node is empty, we inspect its two children and the pull up the larger value. We terminate when no value moves up. Alternately, we can construct the tree as follows

**Procedure Build Priority Search Tree($S$)**

1. **Input** A set $S$ of $n$ points in plane. **Output** A priority search tree;
2. Let $p \in S$ be the largest $y$ coordinate. Store $y$ in the root $r$;
3. **if** $S-p$ is non-empty **then**
   4. Let $L$ (respectively $R$) be the left (respectively right) half of the points in $S-p$ with separating $x$ coordinate $m(L,R)$;
   5. Set $X(r) = m(L,R)$ in root $r$;
   6. Build Priority Search Tree ($L$) ;
   7. Build Priority Search Tree ($R$) ;
4. **Comment**: The left (right) subtree will be searched iff the query interval extends to the left (right) of $X(r)$

Since the number of points reduce by half in every subtree, the height of this tree is clearly $O(\log n)$. This combo data structure is known as priority search trees that takes only $O(n)$ space and supports $O(\log n)$ time three sided query.

### 6.4 Planar Convex Hull

**Problem** Given a set $P$ of $n$ points in the plane, we want to compute the smallest convex polygon containing the points.

A polygon is convex if for any two given points $a, b$ inside the polygon, the line segment $a, b$ is completely inside the polygon.

A planar hull is usually represented by an ordering of the extreme points - a point is extreme iff it cannot be expressed as a convex linear combination$^3$ of three other points in the convex hull. We make a few observations about the convex hull $CH(P)$.

**Observation 6.2** $CH(P)$ can be described by an ordered subset $x_1, x_2 \ldots$ of $P$, such that it is the intersection of the half-planes supported by $(x_i, x_{i+1})$.

$^2$This kind of analysis where we are amortizing the cost on the output points is called filtering search.

$^3$inside the triangle
Figure 6.4: The query is the semi-infinite upper slab supported by the two bottom points (0, 4.5) and (10, 4.5). The points corresponding to each node are indicated as well as the separating x-coordinate after the ":". The points corresponding to the shaded points are output. The lightly shaded nodes are visited but not output.

We know that the entire segment $(x_i, x_{i+1})$ should be inside the hull, so if all the points of $P$ (other than $x_i, x_{i+1}$ lie to one side, then $CH(P) \subset \text{half-plane supported by } x_i, x_{i+1}$. Therefore $CH(P)$ is a sequence of extreme points and the edges joining those points and clearly there cannot be a smaller convex set containing $P$ since any point in the intersection must belong to the convex hull.

For building the hull, we divide the points by a diagonal joining the leftmost and the rightmost point - the points above the diagonal form the upper hull and the points below form the lower hull. We also rotate the hull so that the diagonal is parallel to x-axis. We will describe algorithms to compute the upper hull - computing the lower hull is analogous.

The planar convex hull is a two dimensional problem and it cannot be done using a simple comparison model. While building the hull, we will need to test whether three points $(x_0, y_0), (x_1, y_1), \text{ and } (x_2, y_2)$ are clockwise (counter-clockwise) oriented. Since the x-coordinates of all the points are ordered, all we need to do is test whether the middle point is above or below the line segment formed by the other two. A triple of points $(p_0, p_1, p_2)$ is said to form a right turn iff the determinant

$$
\begin{vmatrix}
  x_0 & y_0 & 1 \\
  x_1 & y_1 & 1 \\
  x_2 & y_2 & 1
\end{vmatrix} < 0
$$

87
where \((x_1, y_1)\) are the co-ordinates of \(p_1\). If the determinant is positive, then the triple points form a left turn. If the determinant is 0, the points are collinear.

### 6.4.1 Jarvis March

A very intuitive algorithm for computing convex hulls which simply simulates \((or\ gift\ wrapping)\). It starts with any extreme point and repeatedly finds the successive points in clockwise direction by choosing the point with the least polar angle with respect to the positive horizontal ray from the first vertex. The algorithm runs in \(O(nh)\) time where \(h\) is the number of extreme points in \(CH(P)\). Note that we actually never compute angles; instead we rely on the determinant method to compare the angle between two points, to see which is smaller. To the extent possible, we only rely on algebraic functions when we are solving problems in \(\mathbb{R}^d\). Computing angles require inverse trigonometric functions that we avoid. Jarvis march starts by computing the leftmost point \(\ell\), i.e., the point whose \(x\)-coordinate is smallest which takes linear time.

When \(h\) is \(o(\log n)\), Jarvis march is asymptotically faster than Graham’s scan.

### 6.4.2 Graham’s Scan

Using this algorithm each point in the set \(P\) is first sorted using their \(x\)-coordinate in \(O(n \log n)\) time and then inductively constructs a convex chain of extreme points. For the upper hull it can be seen easily that a convex chain is formed by successive right-turns as we proceed in the clockwise direction from the left-most point. When we consider the next point (in increasing \(x\)-coordinates), we test if the last three points form a a convex sub-chain, i.e. they make a right turn. If so, we push it into the stack. Otherwise the middle point among the triple is discarded (Why ?) and the last three points (as stored in the stack) are tested for right-turn. It stops when the convex sub-chain property is satisfied.

**Time bound**: Since each point can be inserted and deleted at most once, the running time is linear after the initial sorting.

### 6.4.3 Sorting and Convex hulls

There is a very close relationship between sorting and convex hulls and we can reduce sorting to convex hull in the following manner. Suppose all the input points are on a parabola (i.e. all are extreme points). Then the ordered output of extreme points is a sorted output.

So it is hardly surprising that almost all sorting algorithms have a counterpart in the world of convex hull algorithms.
An algorithm based on divide-and-conquer paradigm which works by arbitrary partitioning is called **merge hull**. After creating arbitrary partition, we construct convex hulls of each partition. Finally, merge the two partitions in \( O(n) \) steps.

\[
T(n) = 2T\left(\frac{n}{2}\right) + O(n)
\]

The key step here is to merge the two upper hulls. Note that the two upper hulls are not necessarily separated by a vertical line \( L \). The merge step computes the common tangent, called bridge over line \( L \), of these two upper hulls.

### 6.5 A Quickhull Algorithm

Let \( S \) be a set of \( n \) points whose convex hull has to be constructed. We compute the convex hull of \( S \) by constructing the upper and the lower hull of \( S \). Let \( p_l \) and \( p_r \) be the extreme points of \( S \) in the x direction. Let \( S_u \) ( \( S_d \) ) be the subset of \( S \) which lie above ( below ) of the line through \( p_l \) and \( p_r \). As we had noted previously, \( S_u \cup \{p_l, p_r\} \) and \( S_d \cup \{p_l, p_r\} \) determine the upper and the lower convex hulls. We will describe the algorithm **Quickhull** to determine the upper hull using \( S_u \cup \{p_l, p_r\} \). As the name suggests, this algorithm shares many features with its namesake quicksort.

The slope of the line joining \( p \) and \( q \) is denoted by \( \text{slope}(pq) \). The predicate \( \text{left-turn}(x, y, z) \) is true if the sequence \( x, y, z \) has a counter-clockwise orientation, or equivalently the area of the triangle has a positive sign.
Algorithm Quickhull$(S_a, p_l, p_r)$

Input: Given $S_a = \{p_1, p_2, \ldots, p_n\}$ and the leftmost extreme point $p_l$ and the rightmost extreme point $p_r$. All points of $S_a$ lie above the line $l_{p_l p_r}$.

Output: Extreme points of the upper hull of $S_a \cup \{p_l, p_r\}$ in clockwise order.

Step 1. If $S_a = \{p\}$, then return the extreme point $\{p\}$.

Step 2. Select randomly a pair $\{p_{2i-1}, p_{2i}\}$ from the the pairs $\{p_{2j-1}, p_{2j}\}$, $j = 1, 2, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$.

Step 3. Select the point $p_m$ of $S_a$ which supports a line with slope $(p_{2i-1}p_{2i})$. (If there are two or more points on this line then choose a $p_m$ that is distinct from $p_l$ and $p_r$). Assign $S_a(l) = S_a(r) = \emptyset$.

Step 4. For each pair $\{p_{2j-1}, p_{2j}\}$, $j = 1, 2, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$ do the following

Case 1: $x[p_{2j}] < x[p_m]$
- if left-turn $(p_m, p_{2j}, p_{2j-1})$ then $S_a(l) = S_a(l) \cup \{p_{2j-1}, p_{2j}\}$
- else $S_a(l) = S_a(l) \cup \{p_{2j-1}\}$.

Case 2: $x[p_m] < x[p_{2j-1}]$
- if left-turn $(p_m, p_{2j-1}, p_{2j})$ then $S_a(r) = S_a(r) \cup \{p_{2j}\}$
- else $S_a(r) = S_a(r) \cup \{p_{2j-1}, p_{2j}\}$.

Case 3: $x[p_{2j-1}] < x[p_m] < x[p_{2j}]$
- $S_a(l) = S_a(l) \cup \{p_{2j-1}\}$;
- $S_a(r) = S_a(r) \cup \{p_{2j}\}$.

Step 5. (i) Eliminate points from $S_a(l)$ which lie below the line joining $p_l$ and $p_m$.
(ii) Eliminate points from $S_a(r)$ which lie below the line joining $p_m$ and $p_r$.

Step 6. If $S_a(l) \neq \emptyset$ then Quickhull$(S_a(l), p_l, p_m)$.
- Output $p_m$.
- If $S_a(r) \neq \emptyset$ then Quickhull$(S_a(r), p_m, p_r)$.

Remark In step 3, show that if the pair $\{p_{2i-1}, p_{2i}\}$ satisfies the property that the line containing $\overrightarrow{p_{2i-1}p_{2i}}$ does not intersect the line segment $\overrightarrow{p_{2i-1}p_{2i}}$, then it guarantees that $p_{2i-1}$ or $p_{2i}$ does not lie inside the triangle $\triangle p_l p_{2i} p_r$ or $\triangle p_l p_{2i-1} p_r$, respectively.

This could improve the algorithm in practice by eliminating all points within the quadrilateral $p_l, p_{2i-1}, p_{2i}, p_r$.

6.5.1 Analysis

To get a feel for the convergence of the algorithm Quickhull we must argue that in each recursive call, some progress is achieved. This is complicated by the possibility that one of the end-points can be repeatedly chosen as $p_m$. However, if $p_m$ is $p_l$, then at least one point is eliminated from the pairs whose slopes are larger than the
Figure 6.5: \( \text{Left-turn}(p_m, p_{2j-1}, p_{2j}) \) is true but \( \text{slope}(p_{2j-1}p_{2j}) \) is less than the median slope given by \( L \).

supporting line \( L \) through \( p_l \). If \( L \) has the largest slope, then there are no other points on the line supporting \( p_m \) (Step 3 of algorithm). Then for the pair \( (p_{2j-1}, p_{2j}) \), whose slope equals that of \( L \), \text{left-turn} \( (p_m, p_{2j}, p_{2j-1}) \) is true, so \( p_{2j-1} \) will be eliminated. Hence it follows that the number of recursive calls is \( O(n + h) \), since at each call, either with an output vertex or it leads to elimination of at least one point.

Let \( N \) represent the set of slopes \( (p_{2j-1}p_{2j}) \), \( j = 1, 2, \ldots \lfloor \frac{n}{2} \rfloor \). Let \( k \) be the rank of the slope \( (p_{2i-1}p_{2i}) \), selected uniformly at random from \( N \). Let \( n_l \) and \( n_r \) be the sizes of the subproblems determined by the extreme point supporting the line with slope \( (p_{2i-1}p_{2i}) \). We can show that

**Observation 6.3** \( \max(n_l, n_r) \leq n - \min(\lfloor \frac{n}{2} \rfloor - k, k) \).

Without loss of generality, let us bound the size of the right sub-problem. There are \( \lfloor \frac{n}{2} \rfloor - k \) pairs with slopes greater than or equal to \( \text{slope}(p_{2i-1}p_{2i}) \). At most one point out of every such pair can be an output point to the right of \( p_m \).

If we choose the median slope, i.e., \( k = \frac{n}{4} \), then \( n_l, n_r \leq \frac{3}{4} n \). Let \( h \) be the number of extreme points of the convex hull and \( h_l(h_r) \) be the extreme points of the left (right) subproblem. We can write the following recurrence for the running time.

\[
T(n, h) \leq T(n_l, h_l) + T(n_r, h_r) + O(n)
\]

where \( n_l + n_r \leq n \), \( h_l + h_r \leq h - 1 \). Exercise 6.10 requires you to show that the solution of the above recurrence relation is \( O(n \log n) \). Therefore this achieves the right balance between Jarvis march and Graham scan as it scales with the output size at least as well as Jarvis march and is \( O(n \log n) \) in the worst case.
6.5.2 Expected running time *

Let $T(n, h)$ be the expected running time of the algorithm randomized Quickhull to compute $h$ extreme upper hull vertices of a set of $n$ points, given the extreme points $p_l$ and $p_r$. So the $h$ points are in addition to $p_l$ and $p_r$, which can be identified using $\frac{3}{2} \cdot n$ comparisons initially. Let $p(n_l, n_r)$ be the probability that the algorithm recurses on two smaller size problems of sizes $n_l$ and $n_r$ containing $h_l$ and $h_r$ extreme vertices respectively. Therefore we can write

$$T(n, h) \leq \sum_{n_l, n_r \geq 0} p(n_l, n_r)(T(n_l, h_l) + T(n_r, h_r)) + bn \quad (6.5.3)$$

where $n_l, n_r \leq n - 1$ and $n_l + n_r \leq n$, and $h_l, h_r \leq h - 1$ and $h_l + h_r \leq h$ and $b > 0$ is a constant. Here we are assuming that the extreme point $p_m$ is not $p_l$ or $p_r$. Although, in the Quickhull algorithm, we have not explicitly used any safeguards against such a possibility, we can analyze the algorithm without any loss of efficiency.

**Lemma 6.1** $T(n, h) \in O(n \log h)$.

**Proof:** We will use the inductive hypothesis that for $h' < h$ and for all $n'$, there is a fixed constant $c$, such that $T(n', h') \leq cn' \log h'$. For the case that $p_m$ is not $p_l$ or $p_r$, from Eq. 6.5.3 we get

$$T(n, h) \leq \sum_{n_l, n_r \geq 0} p(n_l, n_r)(cn_l \log h_l + cn_r \log h_r) + bn.$$

Since $n_l + n_r \leq n$ and $h_l, h_r \leq h - 1$,

$$n_l \log h_l + n_r \log h_r \leq n \log(h - 1) \quad (6.5.4)$$

Let $\mathcal{E}$ denote the event that $\max(n_l, n_r) \leq \frac{7}{8} n$ and $p$ denote the probability of $\mathcal{E}$. Note that $p \geq \frac{1}{2}$.

From the law of conditional expectation, we have

$$T(n, h) \leq p \cdot [T(n_l, h_l | \mathcal{E}) + T(n_r, h_r | \mathcal{E})] + (1 - p) \cdot [T(n_l, h_l + |\bar{\mathcal{E}}| + T(n_r, h_r | \bar{\mathcal{E}})] + bn$$

where $\bar{\mathcal{E}}$ represents the complement of $\mathcal{E}$.

When $\max(n_l, n_r) \leq \frac{7}{8} n$, and $h_l \geq h_r$,

$$n_l \log h_l + n_r \log h_r \leq \frac{7}{8} n \log h_l + \frac{1}{8} n \log h_r \quad (6.5.5)$$

The right hand side of 6.5.5 is maximized when $h_l = \frac{7}{8}(h - 1)$ and $h_r = \frac{1}{8}(h - 1)$. Therefore,
\[ n_l \log h_l + n_r \log h_r \leq n \log (h - 1) - tn \]

where \( t = \log 8 - \frac{7}{8} \log 7 \geq 0.55 \). We get the same bounds when \( \max(n_l, n_r) \leq \frac{7}{8} n \) and \( h_r \geq h_l \). Therefore

\[
T(n, h) \leq p(cn \log(h - 1) - tcn) + (1 - p)cn \log(h - 1) + bn \]
\[
= pcn \log(h - 1) - ptcn + (1 - p)cn \log(h - 1) + bn \]
\[
\leq cn \log h - ptcn + bn
\]

Therefore from induction, \( T(n, h) \leq cn \log h \) for \( c \geq \frac{b}{t} \).

In case \( p_m \) is an extreme point (say \( p_l \)), then we cannot apply Eq. 6.5.3 directly, but some points will still be eliminated according to Observation 6.3. This can happen a number of times, say \( r \geq 1 \), at which point, Eq. 6.5.3 can be applied. We will show that this is actually a better situation, that is, the expected time bound will be less and hence the previous case dominates the solution of the recurrence.

The rank \( k \) of \( \text{slope}(p_{2i-1}p_{2i}) \) is uniformly distributed in \([1, \frac{n}{2}]\), so the number of points eliminated is also uniformly distributed in the range \([1, \frac{n}{2}]\) from Observation 6.3. (We are ignoring the floor in \( \frac{n}{2} \) to avoid special cases for odd values of \( n \) - the same bounds can be derived even without this simplification). Let \( n_1, n_2 \ldots n_r \) be the \( r \) random variables that represent the sizes of subproblems in the \( r \) consecutive times that \( p_m \) is an extreme point. It can be verified by induction, that \( E[\sum_{i=1}^{r} n_i] \leq 4n \) and \( E[n_r] \leq (3/4)^r n \) where \( E[\cdot] \) represents the expectation of a random variable. Note that \( \sum_{i=1}^{r} b \cdot n_i \) is the expected work done in the \( r \) divide steps. Since \( cn \log h \geq 4nb + c(3/4)^r \cdot n \log h \) for \( r \geq 1 \) (and \( \log h \geq 4 \)), the previous case dominates. \( \square \)

### 6.6 Point location using persistent data structure

The point location problem involves an input planar partition (a planar graph with an embedding on the plane), for which we build a data structure, such that given a point, we want to report the region containing the point. This fundamental problem has numerous applications including cartography, GIS, Computer Vision etc.

The one dimensional variant of the problem has a natural solution based on binary search - in \( O(\log n) \) time, we can find the interval containing the query point. In two dimensions, we can also consider a closely related problem called ray shooting, in which we shoot a vertical ray in the horizontal direction and report the first segment that it hits. Observe that every segment borders two regions and we can report the region below the segment as a solution to the point location problem (Exercise problem 6.11). Consider a vertical slab which is cris-crossed by \( n \) line segments such that no pair of segments intersect within the slab. Given a query point, we can easily solve the binary search to answer a ray shooting query in \( O(\log n) \) primitives of the following kind - Is the point below/above a line segment. This strategy works since the line segments are totally ordered within the slab (they mat intersect outside).
For the planar partition, imagine a vertical line $V$ being swept from left to right and let $V(x)$ represent the intersection of $V$ with the planar partition at an X-coordinate value $x$. For simplicity let us assume that no segment is vertical. Further let us order the line-segments according to $V(x)$ and denote it by $S(x)$.

**Observation 6.4** Between two consecutive (in X direction) end-points of the planar partition, $S(x)$ remains unchanged.

Moreover the region between two consecutive end-points is a situation similar to vertical slab discussed before. So once we determine which vertical slab contains the query point, in an additional $O(\log n)$ steps, we can solve the ray shooting problem. Finding the vertical slab is a one dimensional problem and can be answered in $O(\log n)$ steps involving a binary search. Therefore the total query time is $O(\log n)$ but the space bound is not nearly as desirable. If we treat the $(2n - 1)$ vertical slabs corresponding to the $2n$ end-points, we are required to build $\Omega(n)$ data structures, each of which involves $\Omega(n)$ segments. Figure 6.6 depicts a worst-case scenario in terms of space. A crucial observation is that the two consecutive vertical slabs have almost all the segments in common except for the one whose end-points separate the region.

*Can we exploit the similarity between two ordered lists of segments and support binary search on both lists efficiently?*

In particular, can we avoid storing the duplicate segments and still support $\log n$ steps binary searches. Here is the intuitive idea. Wlog, let us assume that an element is inserted and we would like to maintain both versions of the tree (before and after insertion). Let us also assume that the storage is leaf based.

**Path copying strategy** If a node changes then make a new copy of its parent and also copy the pointers to its children.

Once a parent is copied, it will lead to copying its parent, etc, until the entire root-leaf path is copied. At the root, create a label for the new root. Once we know which root node to start the binary search, we only follow pointers and the search proceeds in the normal way that is completely oblivious to fact that there are actually two implicit search trees. The search time also remains unchanged at $O(\log n)$. The same strategy works for any number of versions except that to start searching at the correct root node, we may require an additional data structure. In the context of planar point location, we can build a binary search tree that supports one dimensional search.

The space required is $(\text{path length}) \cdot (\text{number of slabs}) + n$ which is $O(n \log n)$. This is much smaller than the $O(n^2)$ scheme that stores each tree explicitly.
Figure 6.6: An example depicting $\Omega(n^2)$ space complexity for $n$ segments. The search tree corresponds to the slab $s_5$ and each node corresponds to an above-below test corresponding to the segment.

Figure 6.7: Path copying technique on adjacent slabs $s_5$ and $s_6$. 
6.7 Incremental construction

Given a set $S = \{p_1, p_2 \ldots p_n\}$ of $n$ points on a plane, we want to find a pair of points $q, r \in S$ such that $d(p, q) = \min_{p_i, p_j \in S} d(p_i, p_j)$ where $d()$ computes the Euclidean distance between two points. The pair $(q, r)$ is known as the closest pair and it may not be unique. Moreover, the closest pair has distance zero if the points are not distinct.

In one dimension, it is easy to compute the closest pair by first sorting the points and choosing an adjacent pair which has the minimum separation. A trivial algorithm is to compute all the $\binom{n}{2}$ pairs and choose the minimum separated pair, so we would like to design a significantly faster algorithm.

A general approach to many similar problems is given in Figure 6.8. The idea is to maintain the closest pair in an incremental fashion, so that in the end we have the required result.

**Algorithm 1: Closest pair($S$)**

1. Input $P = \{p_1, p_2 \ldots p_n\}$
2. $S = \{p_1, p_2\}; C = d(p_1, p_2); j = 2$
3. while $j \leq n$ do
4.   if $d(p_j, S) < C$ then
5.     $C = d(p_j, q)$ where $q = \arg \min_{p \in S} d(p_j, p)$
6.     $P \leftarrow P \cup \{p_j\}; j \leftarrow j + 1$
7. Output $C$ as closest pair distance.

Figure 6.8: Incremental Algorithm for closest pair computation

While the correctness of the algorithm is obvious, the analysis depends on the test in line 3 and the update time in line 5. For simplicity, let us analyze the running time for points in one dimension. Suppose the distances $d(p_{j+1}, S_j)$ are decreasing where $S_j = \{p_1, p_2 \ldots p_j\}$. Then the closest pair distance $C$ is updated in every step. To find the closest point from $p_{j+1}$ to $S_j$, we can maintain $S_j$ as a sorted set and we can find the closest point from $p_{j+1}$ using a binary search in $O(\log j) = O(\log n)$ time. Overall, the algorithm takes $O(n \log n)$ time which is the same as presorting the points.

For points on a plane, we have to design a data structure to efficiently perform the test in line 3 and update in line 5. Trivially it can be done in $O(n)$ steps leading to an $O(n^2)$ time algorithm. Instead we analyze the algorithm for a random ordering of the points

---

4So the lower bound for element distinctness would hold for the closest pair problem.
points in $S$. This will potentially reduce the number of updates required significantly from the worst case bound of $n-2$ updates. Let $q_i$ denote the probability that point $p_i$ causes an update when the points are inserted in a random order. A random ordering corresponds to a random permutation of points in $P$. To avoid extra notations let us assume that $p_1, p_2 \ldots p_n$ is according to a randomly chosen permutation.

We can restate our problem as

When $p_1, p_2 \ldots p_i$ is a random ordering of the set of points $P = \{p_1, p_2 \ldots p_i\}$ what is the probability that $p_i$ defines the closest pair?

Suppose the closest pair is unique, i.e., $C = d(r, s)$ for some $r, s \in \{p_1, p_2 \ldots p_i\}$. Then, this probability is the same as the event that $p_i = \{r, s\}$. The total number of permutations of $i$ objects is $i!$ and the total number of permutations with $r$ or $s$ as the last element is $2(i - 1)!$. So the probability that $p_i$ defines $C$ equals $\frac{2(i-1)!}{i!} = \frac{2}{i}$.

In a random permutation of $n$ elements, the previous argument holds for a fixed set of $i$ points. The law of total probability states that

$$\Pr[A] = \Pr[A|B_1] \cdot \Pr[B_1] + \Pr[A|B_2] \cdot \Pr[B_2] + \ldots \Pr[A|B_k] \cdot \Pr[B_k]$$

for disjoint events $B_1, B_2 \ldots$ (6.7.6)

In the above situation $B_i$ represent each of the $\binom{n}{i}$ possible choice of $i$ elements as the first $i$ elements and by symmetry the probabilities are equal as well as $\sum_i \Pr[B_i] = 1$. Since $\Pr[A|B_i] = \frac{2}{i}$, the unconditional probability of update in the $i$-th step is $\frac{2}{i} \cdot U(i)$ where $U(i)$ is the cost of updating the data structure in the $i$-th step. Therefore even for $U(i) = O(i \log i)$, the expected update time is $O(\log i) = O(\log n)$.

The situation for the test in line 3 is somewhat different since we will execute this step regardless of whether update is necessary. Given $S$ and a new point $p_i$, we have find the closest point from $p_i$ and $S$ (and update if necessary). Suppose the closest pair distance in $S$ is $D$, then consider a $D \times D$ grid of the plane and each point of $S$ is hashed to the appropriate cell. Given the new point $p_i = (x_i, y_i)$, we can compute the cell as $\lceil \frac{x_i}{D} \rceil \lceil \frac{y_i}{D} \rceil$. It can be seen (Figure 6.9) that the closest point to $p_i$ is within distance $D$ then it must lie in one of the neighboring grid cells, including the one containing $p_i$. We can exhaustively search each of the nine cells.

**Claim 6.1** None of the cells can contain more than 4 points.

This implies that we need to do at most $O(1)$ computations. These neighboring cells can be stored in some appropriate search data structure (Exercise ??) so that it can be accessed in $O(\log i)$ steps. In line 4, this data structure can be rebuilt in $O(i \log i)$ time which results in an expected update time of $O(\log i)$. So, the overall expected running time for the randomized incremental construction is $O(n \log n)$.  

97
Figure 6.9: Maximum number of $D$-separated points per cell is 4 and the shaded area is the region within which a point can lie with distance less than $D$ from $p$

Exercise Problems

Exercise 6.1 Given a set of $n$ $d$-dimensional points, the immediate predecessor of a query point can be determined in $O(d + \log n)$ comparisons.

Exercise 6.2 Show that for any query interval, the points belong to at most $2\log n$ subtrees.

Exercise 6.3 Show how to use threading to solve the range query in a BST without having leaf based storage.

Exercise 6.4 Analyze the space bound for $d$ dimensional orthogonal range trees?

Exercise 6.5 How would you modify the data structure for the counting version of orthogonal range trees?

Exercise 6.6 Find a solution rigorously for the recurrence given in Equation 6.1.2.

Exercise 6.7 Work out the details of performing a three-sided query using a Priority Search tree and also analyse the running time. If the given set of points is sorted by $y$-coordinates, show that the priority search trees can be constructed in $O(n)$ time.

Exercise 6.8 Given a set of $n$ line segments, design a data structure such that for any query rectangle, the set of line segments intersecting (including fully contained) can be reported in $O(\log^2 n + k)$ time. Here $k$ represents the size of output.
Exercise 6.9 Given a set of \( n \) horizontal and vertical segments, design an algorithm to identify all the connected components in \( O(n \text{polylog}(n)) \) time. A connected component is a set of line segments is defined as follows: two segments that intersect are connected. A segment intersecting any segment of a connected component belongs to the connected component.

Exercise 6.10 Complete the solution of the recurrence for running time of quickhull algorithm to show that \( T(n, h) \) is \( O(n \log h) \).

Exercise 6.11 Design an efficient solution to the ray shooting problem using by extending the interval trees.

Exercise 6.12 Given a set of \( n \) horizontal line segments, design a data structure that reports all intersections with a query vertical segment. Hint: Use segment trees.

Exercise 6.13 Analyse the performance of range trees for reporting orthogonal range queries for dimensions \( d \geq 3 \). In particular what are the preprocessing space and query time?

Exercise 6.14 If we allow for insertion and deletion of points, how does the performance of range trees get affected? In particular what are the time bounds for orthogonal range query, insertion and deletion of points? Discuss the data structure in details.

Exercise 6.15 Design efficient algorithms to construct union and intersection of two convex hulls.

Exercise 6.16 Given a set \( S \) of \( n \) points in a plane, design a data structure to
(i) Find the closes point form a query line.
(ii) Find the closest point to the convex hull of \( S \) from a query point (if the point is inside the hull then distance is 0).

Exercise 6.17 A point \( p_1 \succ p_2 \) (\( p_1 \) dominates \( p_2 \)) iff all the coordinates of \( p_1 \) are greater than or equal to \( p_2 \). A point \( p \) is maximal if it is not dominated by any other point in an input set \( S \).
(i) Design an \( O(n \log n) \) time algorithm to find all maximal points in an input set \( S \) of \( n \) distinct points (no two points have all coordinates same).
(ii) Design an \( O(n \log n) \) algorithm for the three dimensional version of the DOMINANCE problem.

Exercise 6.18 Design an \( O(n \log n) \) algorithm for finding all \( h \) maximal points among a set of \( n \) points on a plane.
Exercise 6.19  Point-line duality  Consider a transform \( D(a,b) = \ell : y = 2ax - b \) which maps a point \( p = (a,b) \) to a line \( \ell : y = 2ax - b \). Moreover \( D \) is a 1-1 mapping, i.e., \( D(\ell : y = mx + c) = (m/2, -c) \), such that \( D(D(p)) = p \). Note that \( D \) is not defined for \( m = \infty \), i.e., vertical lines.

Prove the following properties of \( D \) where \( p \) is a point and \( \ell \) is a (non-vertical) line.

1. If \( p \) is incident on \( \ell \) then \( D(\ell) \) is incident on \( D(p) \).

2. If \( p \) lies below \( \ell \), then \( D(\ell) \) lies above \( D(p) \) and vice versa.

3. Let \( p \) be the intersection point of lines \( \ell_1 \) and \( \ell_2 \). Then the line \( D(p) \) passes through the points \( D(\ell_1) \) and \( D(\ell_2) \).

Exercise 6.20  Intersection of half-planes  Given a set of \( n \) half-planes \( h_i : y = m_i \cdot x + c_i \), we want to compute the intersection of \( h_i \)s. Note that the intersection of half-spaces in a convex region which may be bounded, unbounded or even empty.

1. If the intersection is non-empty show that the boundary is the intersection of two convex chains \( C^+ \) and \( C^- \) where
   \( (i) \ C^+ = \bigcap_{h_i \in H^+} H_i \quad (ii) \ C^- = \bigcap_{h_i \in H^-} H_i \)
   where \( H^+ \) (respectively \( H^- \)) denotes the set of positive (negative) half-planes, i.e., the planes that contain \((0, \infty)\) and \((0, -\infty)\).

2. Design an \( O(n \log n) \) algorithm for constructing the intersection of \( n \) half-planes by exploiting the dual transform described in the previous problem.

   \textbf{Hint}  There is a 1-1 correspondence between the boundary of the intersection of half-planes and the boundary of the convex hull of \( D(h_i) \). If a point belongs to the lower hull then there exists a line through the point that contains all the remaining points on the positive half-plane. Likewise if a half-plane forms the boundary of \( C^+ \), then a point on the boundary satisfies \( \bigcap_{h_i \in H^+} H_i \).
Chapter 7

Fast Fourier Transform and Applications

7.1 Polynomial evaluation and interpolation

A polynomial $P(x)$ of degree $n - 1$ in indeterminate $x$ is a power series with maximum degree $n - 1$ and has the general form $a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x + a_0$, where $a_i$ are coefficients over some field, typically the complex numbers $\mathbb{C}$. Some of the most common problems involving polynomials are

**evaluation** Given a value for the indeterminate $x$, say $x'$, we want to compute $\sum_{i=0}^{n-1} a_i \cdot x'^i$.

By Horner’s rule, the most efficient way to evaluate a polynomial is given by the formula

$$(((a_{n-1}x' + a_{n-2})x' + a_{n-3})x' + \ldots + a_0$$

We are interested in the more general problem of evaluating a polynomial at multiple (distinct) points, say $x_0, x_1 \ldots x_{n-1}$. If we apply Horner’s rule then it will take $\Omega(n^2)$ operations, but we will be able to do it much faster.

**interpolation** Given $n$ values (not necessarily distinct), say $y_0, y_1 \ldots y_{n-1}$, there is a unique polynomial of degree $n - 1$ such that $P(x_i) = y_i$ $x_i$ are distinct.

This follows from the fundamental theorem of algebra which states that a polynomial of degree $d$ has at most $d$ roots. Note that a polynomial is characterized by its coefficients $a_i$ $0 \leq i \leq n - 1$. A popular method for interpolation is the Lagrange’s formula.

$$P(x) = \sum_{k=0}^{n-1} y_k \cdot \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)}$$
Exercise 7.1  Show that Lagrange’s formula can be used to compute the coefficients $a_i$’s in $O(n^2)$ operations.

One of the consequences of the interpolation is an alternate representation of polynomials as $\{(x_0, y_0), (x_1, y_1) \ldots (x_{n−1}, y_{n−1})\}$ from where the coefficients can be computed. We will call this representation as the point-value representation.

**multiplication**  The product of two polynomials can be easily computed in $O(n^2)$ steps by clubbing the coefficients of the powers of $x$. This is assuming that the polynomials are described by their coefficients. If the polynomials are given by their point-value, then the problem is considerably simpler since

$$P(x) = P_1(x) \cdot P_2(x) \text{ where } P \text{ is the product of } P_1 \text{ and } P_2$$

A closely related problem is that of convolution where we have to perform computations of the kind $c_i = \sum_{l+p=i} a_l \cdot b_p$ for $1 \leq i \leq n$.

The efficiency of many polynomial related problems depends on how quickly we can perform transformations between the two representations.

### 7.2 Cooley-Tukey algorithm

We will solve a restricted version of the evaluation problem where we will carefully choose the points $x_0, x_1 \ldots x_{n−1}$ to reduce the total number of computations. Let $n$ be a power of 2 and let us choose $x_{n/2} = −x_0, x_{n/2+1} = −x_1, \ldots x_{n−1} = −x_{n/2−1}$. You can verify that $P(x) = P_E(x^2) + xP_O(x^2)$ where

$$P_E = a_0 + a_2 x + \ldots + a_{n−2} x^{n−2−1}$$
$$P_O = a_1 + a_3 x + \ldots + a_{n−1} x^{n−2−1}$$

corresponding to the even and odd coefficients and $P_E, P_O$ are polynomials of degree $n/2 − 1$.

$$P(x_{n/2}) = P_E(x_{n/2}^2) + x_{n/2} P_O(x_{n/2}^2) = P_E(x_0^2) − x_0 P_O(x_0^2)$$

since $x_{n/2} = −x_0$. More generally

$$P(x_{n/2+i}) = P_E(x_{n/2+i}^2) + x_{n/2+i} P_O(x_{n/2+i}^2) = P_E(x_i^2) − x_i P_O(x_i^2), \quad 0 \leq i \leq n/2 − 1$$

since $x_{n/2+i} = −x_i$. Therefore we have reduced the problem of evaluating a degree $n − 1$ polynomial in $n$ points to that of evaluating two degree $n/2 − 1$ polynomials at $n/2$ points $x_0^2, x_1^2 \ldots x_{n/2−1}^2$. This will also involve $O(n)$ multiplications and additions to compute the values at the original points. To continue this reduction, we have to choose points such that $x_0^2 = −x_{n/4}^2$ or equivalently $x_{n/4} = \sqrt{-1} \cdot x_0$. This involves complex numbers if we started with coefficients in $\mathbb{R}^1$. If we continue with this strategy

\[\footnote{Depending on our choice of the field $F$, we can define $\omega$ such that $\omega^2 = -1$.} \]
of choosing points, at the \( j \)-th level of recursion, we require

\[ x_i^{2^{j-1}} = -x_i^{2^{j-1}} \quad 0 \leq i \leq \frac{n}{2^j} - 1 \]

This yields \( x_1^{2^{\log n - 1}} = -x_0^{2^{\log n - 1}} \), i.e., if we choose \( \omega^{n/2} = -1 \) then \( x_i = \omega x_{i-1} \). By setting \( x_0 = 1 \), the points of evaluation work out to be \( 1, \omega, \omega^2 \ldots \omega^{n/2} \ldots \omega^{n-1} \) which are usually referred to as the principal \( n \)-th roots of unity.

### Analysis

Let \( \mathcal{P}(x)_{a_0, a_1 \ldots a_{n-1}} \) denote the evaluation of \( \mathcal{P}(x) \) with coefficients \( a_0, a_1 \ldots a_{n-1} \) at points \( z_1, z_2 \ldots z_n \). Then we can write the recurrence

\[
\mathcal{P}(x)_{a_0, a_1 \ldots a_{n-1}} = \mathcal{P}(x)_{a_0, a_1 \ldots a_{n/2-1}} + \mathcal{P}(x)_{a_{n/2}, a_{n/2+1} \ldots a_{n-1}} + O(n) \text{ multiplications and additions}
\]

This immediately yields \( O(n \log n) \) operations for the FFT computation.

For the inverse problem, i.e., interpolation of polynomials given the values at \( 1, \omega, \omega^2 \ldots \omega^{n-1} \), let us view the process of evaluation as a matrix vector product.

\[
\begin{bmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\
\vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
\vdots \\
a_{n-1}
\end{bmatrix}
= \begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
\vdots \\
y_{n-1}
\end{bmatrix}
\]

Let us denote this by the matrix equation \( A \cdot \mathbf{a} = \mathbf{y} \). In this setting, the interpolation problem can be viewed as computing the \( \mathbf{a} = A^{-1} \cdot \mathbf{y} \). Even if we had \( A^{-1} \) available, we still have to compute the product which could take \( \Omega(n^2) \) steps. However the good news is that the inverse of \( A^{-1} \) is

\[
\frac{1}{n} \begin{bmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & \frac{1}{\omega} & \frac{1}{\omega} & \ldots & \frac{1}{\omega^{n-1}} \\
1 & \frac{1}{\omega^2} & \frac{1}{\omega^2} & \ldots & \frac{1}{\omega^{2(n-1)}} \\
\vdots \\
1 & \frac{1}{\omega^{n-1}} & \frac{1}{\omega^{2(n-1)}} & \ldots & \frac{1}{\omega^{(n-1)(n-1)}}
\end{bmatrix}
\]

which can be verified by multiplication with \( A \). Recall that

\[
1 + \omega^i + \omega^{2i} + \omega^{3i} + \ldots + \omega^{i(n-1)} = 0
\]

(Use the identity \( \sum_j \omega^{ji} = \frac{\omega^{j(n-1)} - 1}{\omega^j - 1} = 0 \) for \( \omega^i \neq 1 \).) Moreover \( \omega^{-1}, \omega^{-2}, \ldots \omega^{-(n-1)} \) also satisfy the properties of \( n \)-th roots of unity. This enables us to use the same algorithm as FFT itself that runs in \( O(n \log n) \) operations.
7.3 The butterfly network

If you unroll the recursion of an 8 point FFT, then it looks like the Figure 7.1. Let us work through some successive recursive calls.

\[
P_{0,1,\ldots,7}(\omega_0) = P_{0,2,4,6}(\omega_0^2) + \omega_0 P_{1,3,5,7}(\omega_0^2)
\]

\[
P_{0,1,\ldots,7}(\omega_4) = P_{0,2,4,6}(\omega_0^2) - \omega_0 P_{1,3,5,7}(\omega_0^2)
\]

Subsequently, \( P_{0,2,4,6}(\omega_0^2) = P_{0,4}(\omega_0^4) + \omega_0^2 P_{2,6}(\omega_0^4) \) and

\[
P_{0,2,4,6}(\omega_0^2) = P_{0,4}(\omega_0^4) - \omega_0^2 P_{2,6}(\omega_0^4)
\]

To calculate \( P_{0,4}(\omega_0^4) \) and \( P_{0,4}(\omega_4^4) \) we compute \( P_{0,4}(\omega_0^4) = P_0(\omega_0^8) + \omega_0^4 P_4(\omega_0^8) \) and

\[
P_{0,4}(\omega_4^4) = P_0(\omega_0^8) - \omega_0^4 P_4(\omega_0^8)
\]

Since \( P_i \) denotes \( a_i \), we do not recurse any further. Notice that in the above figure \( a_0 \) and \( a_4 \) are the multipliers on the left-hand side. Note that the indices of the \( a_i \) on the input side correspond to the mirror image of the binary representation of \( i \). A butterfly operation corresponds to the gadget \( \triangleleft \triangleright \) that corresponds to a pair of recursive calls. The black circles correspond to “+” and “-” operations and the appropriate multipliers are indicated on the edges (to avoid cluttering only a couple of them are indicated).
One advantage of using a network is that, the computation in each stage can be carried out in parallel, leading to a total of \( \log n \) parallel stages. Thus FFT is inherently parallel and the butterfly network manages to capture the parallelism in a natural manner.

### 7.4 Schonage and Strassen’s fast multiplication

In our analysis of the FFT algorithm, we obtained a time bound with respect to multiplication and additions in the appropriate field - implicitly we assumed \( \mathbb{C} \), the complex field. This is not consistent with the boolean model of computation and we should be more careful in specifying the precision used in our computation. This is a topic in itself and somewhat out of the scope of the discussion here. In reality, the FFT computations are done using limited precision and operations like rounding that inherently result in numerical errors.

In other kinds of applications, like integer multiplication, we choose an appropriate field where we can do exact arithmetic. However, we must ensure that the field contains \( n \)-th roots of unity. Modular arithmetic, where computations are done modulo a prime number is consistent with the arithmetic done in hardware.

**Observation 7.1** In \( \mathbb{Z}_m \) where \( m = 2^{m/2} + 1 \) and \( n \) is a power of 2, we can use \( \omega = 2^t \).

Since \( n \) and \( m \) are relatively prime, \( n \) has a unique inverse in \( \mathbb{Z}_m \) (recall extended Euclid’s algorithm). Also

\[ \omega^n = \omega^{n/2} \cdot \omega^{n/2} = (2^t)^{n/2} \cdot (2^t)^{n/2} \equiv (m-1) \cdot (m-1) \mod m \equiv (-1) \cdot (-1) \mod m \equiv 1 \mod m \]

**Claim 7.1** If the maximum size of a coefficient is \( b \) bits, the FFT and its inverse can be computed in time proportional to \( O(bn \log n) \).

Note that addition of two \( b \) bit numbers take \( O(b) \) steps and the multiplications with powers of \( \omega \) are multiplications by powers of two which can also be done in \( O(b) \) steps. The basic idea of the algorithm is to extend the idea of polynomial multiplication. Recall, that in Chapter 2, we had divided each number into two parts and subsequently recursively computed by computing product of smaller numbers. By extending this strategy, we divide the numbers \( a \) and \( b \) into \( k \) parts \( a_{k-1}, a_{k-2}, \ldots, a_0 \) and \( b_{k-1}, b_{k-2}, \ldots, b_0 \).

\[
a \times b = (a_{k-1} \cdot x^{k-1} + a_{k-2} \cdot x^{k-2} + \ldots, a_0) \times (b_{k-1} \cdot x^{k-1} + b_{k-2} \cdot x^{k-2} + \ldots, b_0)
\]
where \( x = 2^{n/k} \) - for simplicity assume \( n \) is divisible by \( k \). By multiplying the RHS, and clubbing the coefficients of \( x^i \), we obtain

\[
a \times b = a_{k-1} b_{k-1} x^{2(k-1)} + (a_{k-2} b_1 + b_{k-2} a_1) x^{2(k-3)} + \ldots + a_0 b_0
\]

Although in the final product, \( x = 2^{n/k} \), we can compute the coefficients using any method and perform the necessary multiplications by an appropriate power of two (which is just adding trailing 0’s). This is polynomial multiplication and each term is a convolution, so we can invoke FFT-based methods to compute the coefficients. The following recurrence captures the running time

\[
T(n) \leq P(k, n/k) + O(n)
\]

where \( P(k, n/k) \) is the time for polynomial multiplication of two degree \( k-1 \) polynomials involving coefficients of size \( n/k \). (In a model where the coefficients are not too large, we could have used \( O(k \log k) \) as the complexity of polynomial multiplication.) We will have to do exact computations for the FFT and for that we can use modular arithmetic. The modulo value must be chosen carefully so that

(i) It must be larger than the maximum value of the numbers involved, so that there is no loss of information

(ii) Should not be too large, otherwise, operations will be expensive.

Moreover, the polynomial multiplication itself consists of three distinct phases

(i) Forward FFT transform. This takes \( O(bk \log k) \) using \( b \) bits.
(ii) Pairwise product of the values of the polynomials at the roots of unity. This will be done recursively with cost \( 2k \cdot T(b) \) where \( b \geq n/k \). The factor two accounts for the number of coefficients of the product of two polynomials of degree \( k - 1 \).
(iii) Reverse FFT, to extract the actual coefficients. This step also takes \( O(bk \log k) \) where \( b \) is the number of bits in each operand.

So the previous recurrence can be expanded to

\[
T(n) \leq r \cdot T(b) + O(bk \log k)
\]

where \( r \cdot b \geq n \) and we must choose an appropriate value of \( b \). For coefficients of size \( s \), we can argue that the maximum size of numbers during the FFT computation is \( 2s + \log r \) bits (sum of \( r \) numbers of pairwise multiplication of \( s \) bit numbers). If we choose \( r \) to be roughly \( \sqrt{n/\log n} \), then \( b = \sqrt{n \log n} \) and we can rewrite the recurrence as

\[
T(n) \leq 2 \sqrt{\frac{n}{\log n}} \cdot T(2 \sqrt{n \log n} + \log n) + O(n \log n)
\]

which gives

\[
T(n) \leq \frac{n^{0.5+0.01}}{\log^{0.5} n}
\]
Exercise 7.2 With appropriate terminating condition, say the $O(n^{\log_3 4})$ time multiplication algorithm, verify that $T(n) \in O(n \log^2 n \log \log n)$.

An underlying assumption in writing the recurrence is that all the expressions are integral. This can actually be ensured by choosing $n = 2^\ell$ and carefully choosing $\sqrt{n}$ for even and odd values of $\ell$. Using the technique of wrapped convolution, one can save a factor of two in the degree of the polynomial, yielding the best known $O(n \log n \log \log n)$ algorithm for multiplication.
Chapter 8

String matching and finger printing

8.1 Rabin Karp fingerprinting

Notations If $Y$ is a string of length $m$, $Y_j$ represents the $j$th character and $Y(j)$ represents the substring of $n$ symbols beginning at $Y_j$.

In terms of the above notation, we define the string matching problem as:
Given $X$ (the pattern), find the first index $i$ such that $X = Y(i)$

The obvious way of doing this is brute-force comparision of each $Y(i)$ with $X$ that could result in $\Omega(n^2)$ comparisons. Alternatively, consider the following Idea: Compare $F(X)$ with $F(Y(1)), F(Y(2))$ etc. for some function $F()$ that maps strings of lengths $n$ to relatively shorter strings. The claim is if $F(X) \neq F(Y(i))$ for any $i$ then $X \neq Y(i)$ else there is a chance that $X = Y(i)$ whenever $F(X) = F(Y(i))$.

The function $F$ is known as the fingerprinting function\(^1\) and may be defined according to the application. In this case let

$$F(X) = x \mod p$$

Here $X$ is assumed to be a binary pattern (of 0 and 1) and $x$ is its integer value.

\(^1\)It is called a hash function.
Theorem 8.1 (Chinese Remaindering Theorem) For \( k \) numbers \( n_1, n_2, \ldots, n_k \), relatively prime to each other,

\[ x \equiv y \mod n_i \text{ for all } i \iff x \equiv y \mod n_1n_2\ldots n_k = M \]

Moreover,

\[ y \equiv \sum_{i=1}^{R} c_i d_i y_i \]

where \( c_i d_i \equiv 1 \mod n_i \),
\( d_i = \prod n_1, n_2 \ldots n_{i-1}n_{i+1}, \ldots n_k \) and \( y_i = x \mod n_i \)

Let \( k \) be such that \( 2^m < M = 2 \times 3 \times \ldots \times p_k \) i.e. the first \( k \) primes. From CRT, if \( X \neq Y(i) \) then for some \( p \) in \{2, 3, \ldots, p_k\},

\[ F_p(X) \neq F_p(Y(i)) \]

Enlarge the set somewhat, i.e. \{2, 3, \ldots, p_{2k}\}

\[ P[F_p(X) = F_p(Y(i))|X \neq Y(i)] \leq \frac{1}{2} \]

Otherwise it would violate CRT. So, if we take \{2, 3, \ldots, p_{2k}\} then the probability of false match at fixed position \( Y(i) \) \leq \frac{1}{t^2}. So for any \( i \in \{1, \ldots, t\} \leq t^{\frac{1}{2}} = \frac{1}{7}. \) Thus \( k = n \) will suffice.

Exercise 8.1 Can you prove a similar result without using CRT?

Hint: \( X \equiv Y \mod p \) implies that \( (X - Y) \equiv 0 \mod p \). How many prime factors are there of \( X - Y \) among prime numbers in the range \( X - Y \)?

Size of numbers involved: The product of \( n \) primes \( 2.3\ldots p_n = M > 2^n \). Also from prime number density, \( \frac{U}{\ln U} \leq \pi(U) \leq 1.26 \frac{U}{\ln U} \), where \( \pi(x) \) represent number of primes less than or equal to \( x \).

\[ \frac{U}{\ln U} > t^2n \Rightarrow U = O(t^2n \log(t^2n)) \]

So \( |p| \leq 2 \log t + \log n \), where \( |p| \) is the bit length.
**Updating Fingerprints:** \( Y(i+1) = 2(Y(i) - Y_i \cdot 2^n) + Y_{i+n} \). So \( Y(i+1) \mod p = [2(Y(i) \mod p) - 2^n \mod p \cdot Y_i + Y_{i+n} \mod p] \mod p \). This is all modulo \( p \). So as long as \( |p| = \Omega(\log t) \) = wordsize, it is constant update time. Therefore, the expected cost of a step = \( O(1) + \frac{n}{t^2} \). The \( O(1) \) is due to the cost to update fingerprint function and the term \( \frac{n}{t^2} \) is the probability of false-match multiplied by the cost of verification.

So expected cost =

\[
\sum_{i=1}^{m} O(1) + \frac{n}{t^2} = O(m) + \frac{n \cdot m}{t^2}
\]

By choosing \( t \geq m \), it is \( O(m) \).

### 8.2 KMP algorithm

String matching can be done in linear time with a custom-made DFA (Deterministic Finite Automaton) for the pattern that we are trying to find. At any stage the state of the DFA corresponds to the extent of partial match - it is in state \( i \), if the previous \( i \) symbols of the text has matched the first \( i \) symbols of the pattern. It reaches the final stage iff it has found a match. Given this DFA, we can find all occurrences of an \( n \) symbol pattern in an \( m \) symbol text in \( O(m) \) steps, where there is a transition for every input symbol of the text. The size of the DFA is \( O(n|\Sigma|) \) where \( \Sigma \) is the alphabet which is optimal if \( \Sigma \) is of constant size.

With some additional ideas, the previous method can be made to run in \( O(n + m) \) steps without dependence on the alphabet size. Let us introduce some useful notations.

Let \( X(i) \) denote the first \( i \) symbols of the pattern, i.e. a prefix of length \( i \).
Let \( \alpha \sqsubseteq \beta \) denote \( \alpha \) is a suffix of \( \beta \).

We mimic the DFA in the sense that in case of mismatch of the input alphabet and the pattern, we want to find the largest overlap of the pattern and the part of the text scanned. If we have matched upto \( i \) symbols before a mismatch, we want to find the largest \( j \) \( |j < i \) such that \( X(j) \sqsubseteq X(i) \). Following this, we try matching \( X_{j+1} \) with the next element of the text \( Y \).

The *failure* function of a string is defined as

\[
f(i) = \max_j \{X(j) \sqsubseteq X(i)\}\) otherwise 0, if no such \( X(j) \) exists
\]

With this definition, the strategy is as follows.
Let $Y_k$ denote the $k$-th symbol of the text for which we have a partial match up to $i$ symbols of the pattern, we then try to match $X_{i+1}$ with $Y_{k+1}$-st position of the text. In case of a match, we increase the partial match and if it is $n$ then we have found a match. Otherwise (if $X_{i+1}$ doesn’t match $Y_{k+1}$), we try to match $X_{f(i)+1}$ with $Y_{k+1}$ and again if there no match, we try $X_{f(f(i)+1)}$ with $Y_{k+1}$ and so on till the partial match becomes 0.

Let us postpone the method for computing the failure function and assume that we have the failure function available. The analysis requires us to look at a situation, where the pattern string keeps sliding to the right (till it cannot). We can analyze it in many ways - here we will use the technique of potential function.

### 8.2.1 Analysis of the KMP algorithm

During the algorithm, we may be comparing any given element of the text, a number of times, depending on the failure function. Let us define the potential function as the extent of partial match. Then

**Case: match** The amortised cost of a match is 2 (actual cost is one and the increase in potential is also one).

**Case mismatch** The amortised cost is $\leq 0$, since the potential is decreasing.

So the total amortized cost is $O(m)$.

### 8.2.2 Pattern Analysis

The preprocessing of the pattern involves constructing the failure function $f(i)$.

**Observation 8.1** If the failure function $f(i) = j$, $j < i$, it must be true that $X(j-1) \subseteq X(i-1)$ and $X_i = X_j$.

This shows that the computation of the failure function is very similar to the KMP algorithm itself and we compute the $f(i)$ incrementally with increasing values of $i$.

**Exercise 8.2** Using the potential function method, show that the failure function can be computed in $O(|X|)$ steps.

Therefore the total running time of KMP algorithm is $O(|X| + |Y|)$.  

111
8.3 Generalized String matching

Very often, we encounter string matching problems where the strings are not represented explicitly. This feature lends versatility to many applications. It gives us a way of compactly representing a set of strings and also deal with situations when we do not have complete information about strings\(^2\). One of the fundamental applications is parsing, where we have a compact representation of (possibly infinite) strings in form of a grammar and given a query string, we would like to know if the string belongs to the set described by the grammar.

In one of the simpler cases, we have to deal with wild-card symbols. For example, there is a match between the strings \(acb^*d\) and \(a^*bed\) by setting the first wild card to \(e\) and the second one as \(c\). Here a wild-card is a placeholder for exactly one symbol.

In other applications, the wild-card may represent an entire substring of arbitrary length. Unfortunately none of the previous string matching algorithms are able to handle wild-cards.

**Exercise 8.3** Give an example to argue why KMP algorithm cannot handle wild-cards. You may want to extend the definition of failure function to handle wild-cards. 

**Hint:** Find all occurrences of the pattern \(aba^*a\) in the text \(ababaababa...\)

8.3.1 Convolution based approach

For a start, assume that we are only dealing with binary strings. Given a pattern \(A = a_0a_1a_2...a_{n-1}\) and a text \(B = b_0b_1b_2...b_{m-1}\) where \(x_i, y_i \in \{0, 1\}\), let us view them as coefficients of polynomials. More specifically, let 
\[
\mathcal{P}_A(x) = a_0x^{n-1} + a_1x^{n-2} + a_2x^{n-3} + ... + a_{n-1}
\]
\[
\mathcal{P}_B(x) = b_0 + b_1x + b_2x^2 + ... + b_{m-1}x^{m-1}
\]

The product of \(\mathcal{P}_A\) and \(\mathcal{P}_B\) can be written as 
\[
\sum_{i=0}^{m+n-2} c_i x^i
\]

Note that 
\[
c_{n-1+j} = a_0 \cdot b_j + a_1 \cdot b_{1+j} + a_2 \cdot b_{2+j} + ... + a_{n-1} \cdot b_{n-1+j} \quad 0 \leq j \leq m + n - 2
\]

which can be interpreted as the dot product of \(X\) and \(Y(j)\) \(0 \leq j \leq n - 1\).

If we replace the \(\{0, 1\}\) with \(\{-1, +1\}\), then we can make the following claim.

**Observation 8.2** There is a match in position \(j\) iff \(c_{n-1+j} = n\).

**Exercise 8.4** Prove it rigorously.

The above convolution can be easily done using FFT computation in \(O(m \log m)\) steps.\(^3\) When wildcard characters are present in the pattern, we can assign them the

\(^2\)Typical situation in many biological experiments dealing with genetic sequences

\(^3\)The number involved are small enough so that we can do exact computation using \(O(\log n)\) bit integers.
value 0. If there are \( w \) such characters then we can modify the previous observation by looking for terms that have value \( n - w \) (Why). However, the same may not work if we have wildcards in the text also - try to construct a counterexample.

**Wildcard in Pattern and Text**

Assume that the alphabet is \( \{1, 2 \ldots s\} \) (zero is not included). We will reserve zero for the wildcard. For every position \( i \) of the pattern that (assume there are no wildcards), we will associate a random number \( r_i \) from the set \( \{1, 2, \ldots N\} \) for a sufficiently large \( N \) that we will choose later. Let \( t = \sum_i r_i X_i \). Here \( r_j \)s are are random multipliers such that

**Observation 8.3** *For any vector \( v_1, v_2 \ldots v_n \), suppose there exists some \( i \) for which \( X_i \neq v_i \). Then the probability that \( \sum_i v_i \cdot r_i = t \) is less than \( \frac{1}{N} \).*

We can imagine that the random numbers are chosen sequentially, so that after fixing the first \( n - 1 \) numbers, there is only one choice for which the equation is satisfied. By choosing \( N \geq n^2 \), the probability of a false match in any of the possible positions is \( n \cdot 1/N \leq 1/n \).

Clearly, if the vector \( v_1, v_2 \ldots v_n \) is the same as \( X \), then \( \sum_i v_i \cdot r_i = t \). So this forms the basis for a randomized string matching algorithm. In the presence of wildcards in the pattern \( X \), we assign \( r_i = 0 \) iff \( X_i = * \) (instead of a random non-zero number) and the same result holds for positions that do not correspond to wildcards (these are precisely the positions that are blanked out by 0). The number \( t \) acts like a fingerprint or a hash function for the pattern.

When the text has wildcard, then the fingerprint cannot be fixed and will vary according to the wildcards in the text. The fingerprint \( t_k \) at position \( k \) of the text can be defined as

\[
t_k = \sum_{j=1}^{n} \delta_{j+k-1} \cdot r_j \cdot X_j
\]

where \( \delta_i = 0 \) if \( Y_i = * \) and 1 otherwise. Recall that \( r_j = 0 \) if \( X_j = * \).

Now we can replace * with 0 in the text and run the basic convolution based algorithm with the \( \{-1, +1\} \) alphabet. The probability of error (false match) is calculated along similar lines. To calculate \( t_j \), we perform another convolution using FFT (which uses the \( \{0, 1\} \) alphabet depending on the wildcard characters). Overall, the string matching in the presence of wildcards can be done in \( O(m \log m) \) operations.

**Remark** The use of FFT for pattern matching is due to Fisher and Patterson. However, because of superlinear running time, it is not the preferred method for simple string matching for which KMP and Karp-Rabin are more efficient.

\[\text{4} \text{We do the arithmetic modulo } N\]
Chapter 9

Graph Algorithms

Consider the natural problem of sequencing a set of tasks. Given jobs $J_1, J_2, \ldots, J_n$ and some precedence constraints $J_i \prec J_k$ which denotes that $J_i$ must be completed before $J_k$, we want to find a feasible sequence of performing the tasks or determine that it is not possible.

**Example 9.1** Set of jobs: $J_a, J_b, J_c, J_d$.

Precedence constraints: $J_a \prec J_b$, $J_a \prec J_d$, $J_d \prec J_c$, $J_c \prec J_b$.

One possible sequencing is $J_a, J_d, J_c, J_b$ that satisfies all the precedence constraints.

When does a sequence exist? What happens if we change the precedence from $J_a \prec J_b$ to $J_b \prec J_a$? Let us use a graph $G = (V, E)$ to model the above problem.

The set of vertices correspond to the set of jobs and a directed edge $(v_i, v_k)$ denotes that $J_i \prec J_k$. More formally, we want to define a function $f : V \rightarrow \{1, 2, \ldots, n\}$ such that $\forall i, j \ J_i \prec J_k \iff f(i) < f(k)$.

**Observation 9.1** There is a feasible schedule if and only if there is no directed cycle in the graph.

Clearly, there cannot be a consistent ordering if there is a cycle. Suppose there is no cycle. Then if we start traversing the graph then, we will get stuck at some vertex (in absence of cycle no vertex can repeat). That is, we are guaranteed to find a vertex with outdegree 0 which is called a sink. Number this vertex $n$ which is the largest index. This cannot violate any precedence relation as there is no edge going out. This vertex along with the incoming edges are deleted and the same argument can be applied to the remaining graph. This actually gives an algorithm for generating a feasible sequence but it can be done more efficiently.

A number of graph problems use Depth First Search (DFS) as the starting point. Since it runs in linear time, it is efficient as well.
Procedure Depth First Search of a directed graph(G)

Input A directed graph $G = (V, E)$ where $|V| = n$, $|E| = m$;
Output Starting and Finishing times for every vertex $v \in V$;
Initially all vertices are unmarked. A global counter $c = 1$;
while Some vertex $x$ is unmarked do
  start($x$) \leftarrow c ; Increment c ;
  DFS ($x$)
if there is an unmarked neighbour $y$ of $v$ then
  start($y$) \leftarrow c ; Increment c ;
  DFS ($y$)
else
  finish($v$) \leftarrow c ; Increment c

Figure 9.1 illustrates the result of running DFS on the given graph.
If \( u \leadsto v \), then if the DFS reaches \( u \) before \( v \) then \( \text{start}(u) < \text{start}(v) \). If the DFS reaches \( v \) before \( u \) (suppose it starts from \( v \)), then \( \text{start}(v) < \text{start}(u) \). Clearly starting times alone cannot be used to determine if \( u \leadsto v \).

**Observation 9.2** For \( u, v \in V \), either \( \text{start}(u) < \text{start}(v) < \text{finish}(v) < \text{finish}(u) \) or \( \text{start}(u) < \text{finish}(u) < \text{start}(v) < \text{finish}(v) \). This is called the bracketing property.

If \( u \leadsto v \) and \( v \leadsto u \) then there is a cycle. A directed graph without cycles is called a Directed Acyclic Graph (DAG). We had earlier observed that every directed graph without a cycle must have a sink; by reversing the directions of every edge, we can also argue that there must be at least one vertex with indegree zero.

**Observation 9.3** Every DAG has at least one vertex with indegree 0 (source) and a vertex with outdegree 0 (sink).

Given a DAG, \( G = (V, E) \), we want to define a a function \( f : V \to \{1, 2, \ldots, n\} \) such that for any directed edge \((u, v)\), \( f(v) > f(u) \). This is also called a topological
sorting of the vertices. By using a counter to keep track of the starting and finishing
times of vertices, say \( \text{start}(v) \) and \( \text{finish}(v) \), we can determine reachability between
vertices \( u, v \in V \), denoted by \( u \sim v \). The starting times are also known as the DFS
numbering of the vertices.

**Observation 9.4** The starting and finishing times correspond to preorder and pos-
torder numbering respectively of a DFS tree (if we set the starting time of the starting
vertex as 1).

Note that the preorder and postorder numbering are in the range \([1..n]\), so we will
need two distinct counters for starting time and finishing times which are incremented
appropriately. Alternately, the counts from the global counter can be mapped to the
range \([1,2\ldots n]\) using integer sorting.

### 9.1 Applications of DFS

A DFS on a directed graph \( G = (V,E) \) yields a wealth of information about the
structure of the graph.

The set of edges used to visit a vertex for the first time is a directed tree. Note
that a DFS could produce a set of directed trees. An edge is called *forward* edge, if it
is directed from a lower number visited vertex to a higher number vertex (according
to starting times) in the same tree. An edge between two different trees is called a *cross*
edge and is directed from a higher number to a lower number vertex.

The previous discussion leads to an efficient algorithm for topological sort using
DFS.

**Observation 9.5** If there is a path from \( u \sim v \), then \( \text{post}(u) > \text{post}(v) \).

If the DFS reaches \( u \) before \( v \), then clearly it is true. If \( v \) is reached before \( u \), then
the DFS of \( v \) is completed before it starts at \( u \) since there is no path from \( v \) to \( u \).

**Claim 9.1** The vertices in reverse order of the post-order numbering gives a topolog-
ical sorting.

**Exercise 9.1** The above observations lead to an efficient algorithm for topological
sorting (Exercise 9.7).

### 9.1.1 Strongly Connected Components (SCC)

In a directed graph \( G = (V,E) \), two vertices \( u, v \in V \) are in the same SCC iff \( u \sim v \)
and \( v \sim u \). It is easy to verify that this is an equivalence relation on the vertices and
the equivalence classes correspond to the SCCs of the given graph. Let us define a
graph $G = (V', E')$ as follows - $V'$ corresponds to the SCCs of $G$ and $(c_1, c_2) \in E'$ if
there is an edge from some vertex in $c_1$ to some vertex of $c_2$. Here $c_1, c_2 \in V'$, i.e.,
they are SCCs of $G$. An SCC may consist of a singleton vertex as there is an implicit
edge to itself.

**Exercise 9.2** Prove that $G$ is a DAG.

![Graph](image)

**Figure 9.2**: The pair of numbers associated with the vertices represent the start and
finish time of the DFS procedure. The SCC component DAG has four components.

To determine the SCCs of $G$, notice that if we start a DFS from a vertex of a *sink*
component $c'$ of $G$ then precisely all the vertices of $c'$ can be reached. Since $G$ is not
explicitly available, we will use the following strategy to determine a sink component
of $G$. First, reverse the edges of $G$ - call it $G^R$. The SCCs of $G^R$ is the same as $G$ but
the sink components and source components of $G$ are interchanged. We observe that
If we do a DFS in $G^R$, then then the vertex with the largest postorder numbering is
in a sink component of $G$. This follows from Observation 9.5 and the fact that a sink
component in $G$ is a source component of $G^R$.  

118
This enables us to output the SCC corresponding to the sink component of $G$ using a DFS in $G$ where the vertices are ordered according to the decreasing postorder numbering of $G^R$ \(^1\). Once this component (a sink component) is deleted (where the vertices and the induced edges are deleted), we can apply the same strategy to the remaining graph, i.e., start with the next highest postorder number.

Algorithm 2: Finding SCC of $(G)$

<table>
<thead>
<tr>
<th>Input</th>
<th>A directed graph $G = (V, E)$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>The strong connected components of $G$;</td>
</tr>
<tr>
<td>1</td>
<td>Let $G^R$ be the reversal of $G$ and $p^r: V \to {1, 2 \ldots n}$ be the postorder numbering obtained by doing DFS on $G^R$;</td>
</tr>
<tr>
<td>2</td>
<td>$W \leftarrow V$ and $G(W)$ be subgraph of $G$ induced by $W$;</td>
</tr>
<tr>
<td>3</td>
<td>While $W$ is not empty do</td>
</tr>
<tr>
<td>4</td>
<td>Choose $v = \arg \max_{w \in W} {p^r(w)}$;</td>
</tr>
<tr>
<td>5</td>
<td>Let $V'$ denote the vertices of $W$ reachable from $v$ by DFS($v$) in $G(W)$;</td>
</tr>
<tr>
<td>6</td>
<td>Output $V'$ as an SCC of $G$;</td>
</tr>
<tr>
<td>7</td>
<td>$W \leftarrow W - V'$</td>
</tr>
</tbody>
</table>

Figure 9.3: Finding Strongly Connected Components using two DFS

The correctness of the algorithm is based on the following argument. Suppose $v \sim u$ in $G$ (equivalently $u \sim v$ in $G^R$) where $p^r(v)$ had the largest value in $G^R$. Since $p^r(v) > p^r(u)$, in $G^R$, either $v \sim u$ or $u \not\sim v$. The latter does not hold from our initial premise. So $u \sim v$ in $G$ implying that $u, v$ belong to the same SCC of $G$. This observation can be converted into an induction proof that the vertices output in line 7 indeed form an SCC (Exercise 9.8).

Example 9.2 Let us revisit the graph given in Figure 9.2. For convenience, we will assume that the DFS numbering corresponds to $G^R$, i.e., the original graph has all the edges reversed. The component SCC of the original graph must have the direction of the edges reversed. Since $b$ has the highest value of finish time, we begin DFS from $b$ (in the reverse of the graph). The reachable vertices are $a, e$ which correctly corresponds to a sink component.

9.1.2 Biconnected Components

Given an undirected graph $G = (V, E)$, Biconnected graphs can be defined on the basis of vertex connectivity as well as equivalence classes on edges. Graphs that

\(^1\)This numbering is used to start the DFS from the next unexplored vertex.
cannot be disconnected by removing one vertex. (along with the incident edges) are biconnected. If the removal of a vertex disconnects a graph then such a vertex is called an articulation point. This implies that between any pair of vertices, there are at least two vertex disjoint paths. The generalization of this observation is known as Whitney’s theorem.

**Theorem 9.1 (Whitney)** A graph is $k$-connected if and only if there are $k$ vertex disjoint paths between any pair of vertices.

It is clear that if there are $k$ vertex disjoint paths then at least $k$ vertices must be removed to disconnect the graph. However the proof of the converse is non-trivial and is not included here. We are interested in 2-connectivity in this section.

One obvious procedure to check biconnectivity is to test if there is an articulation point in a graph. For every vertex $v \in V$, check if $G - \{v\}$ is connected. This takes $O(n \cdot (m + n))$ time which we will try to improve using an alternate characterization. Moreover we will also determine the biconnected components if the graph is not biconnected.

Since the two vertex disjoint paths form a simple cycle, the biconnected components also contain a lot of information on the cyclic structure. This also leads to the alternate definition of BCC.

**Definition 9.1** Two edges belong to the same BCC iff they belong to a common (simple) cycle.

A singleton edge is considered as a biconnected component.

![Diagram](image)

Figure 9.4: (a) If there is a simple cycle containing pairs of edges $(e_1, e_2)$ and $(e_2, e_3)$, then there is a common cycle containing $e_1$ and $e_3$ indicated by the dotted line. (b) A Biconnected component Tree with components $A, B, C, D$ and articulation points $x, y$. If there is a cycle induced by $z$, then $A, C, D$ can be merged into a bigger biconnected component.
The DFS on an undirected graph $G = (V, E)$ partitions the edges into $T$ (tree edges) and $B$ (back edges). Based on the DFS numbering (pre-order numbering) of the vertices, we can direct the edges of $T$ from a lower to a higher number and the edges in $B$ from a higher to a lower number. Let us denote the DFS numbering by a function $d(v)$, $v \in V$. Analogous to the notion of component tree in the context of SCC, we can also define a component tree on the BCC. Here the graph $G$ has the biconnected components (denoted by $B$) and the articulation points (denoted by $A$) as the set of vertices. We have an edge between $a \in A$ and $b \in B$ if $a \in B$. As illustrated in Figure ?? the component graph cannot contain cycles and must be a tree.

The basic idea behind the BCC algorithm is to detect articulation points. If there are no articulation points then the graph is biconnected. Simultaneously, we also determine the BCC. The DFS numbering $d(v)$ helps us in this objective based on the following intuitive observation.

**Observation 9.6** If there are no back-edges out of some subtree of the DFS tree $T_u$ rooted at a vertex $u$ that leads to a vertex $w$ with $d(w) < d(u)$, then $u$ is an articulation point.

This follows because all paths from the subtree to the remaining graph must pass through $u$ making $u$ an articulation point. To detect this condition, we define an additional numbering of the vertices based on DFS numbering. Let $h(v)$ denote the minimum of the $d(u)$ where $(v, u)$ is a back edge. Then

$$LOW(v) = \min_{w | (v, w) \in T} \{LOW(w), h(v)\}$$

Note that the $h(v)$ and $LOW(v)$ can be easily computed if $v$ is a leaf node of the DFS tree. Using this as a base case, we can compute the $h(v)$s and the $LOW$ numbers simultaneously while doing the DFS (Exercise 9.10). Once the $LOW$ numbers are known, we can check if $h(u) \geq d(v)$ for any child $u$ of $v$. If so, the removal of $v$ would disconnect all the vertices in the subtree rooted at $v$ from the remaining graph and therefore $v$ is an articulation point. A special case happens if $v$ is the root of the DFS tree, since $v$ does not have any predecessor. In this case, $v$ is an articulation vertex if it has more than one child in the DFS tree since all paths between the children in the subtrees must go through $v$.

The computation of $LOW$ numbers results in an efficient algorithm for testing biconnectivity but it does not yield the biconnected components directly. For this, let us consider the component graph $G$ (recall that this does not contain any cycles). The biconnected component that corresponds to a leaf node of $G$ should be output as we back-up from a subtree $w$ of $v$ such that $LOW(w)$ is not smaller than $d(v)$ ($v$ is an articulation point). After deleting this component from $G$, we consider the
leaf-component in the remaining $G$. The edges of a BCC can be kept in stack starting from $(v, w)$ that will be popped out till we reach the edge $(v, w)$.

The DFS can start from an arbitrary BCC (biconnected component), this component will be last to be output. Only when the DFS enters a leaf component through an articulation point, this will be only way out of that component and all its edges will be output. For any other non-leaf component, it will be output after all the neighboring components are output, except for the one through which it was first arrived. In other words the DFS on the component tree has a traversal property similar to the DFS on the vertices of a tree. Formalize the above argument into an efficient algorithm that runs in $O(|V| + |E|)$ steps (Exercise 9.14).

9.2 Path problems

We are given a directed graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}$ (may have negative weights also). The natural versions of the shortest path problem are as follows

**distance between a pair** Given vertices $x, y \in V$, find the least weighted path starting at $x$ and ending at $y$.

**Single source shortest path** (SSSP) Given a vertex $s \in V$, find the least weighted path from $s$ to all vertices in $V - \{s\}$.

**All pairs shortest paths** (APSP) For every pair of vertices $x, y \in V$, find least weighted paths from $x$ to $y$.

Although the first problem often arises in practice, there is no specialized algorithm for it. The first problem easily reduces to the SSSP problem. Intuitively, to find the shortest path from $x$ to $y$, it is difficult to avoid any vertex $z$ since there may be a shorter path from $z$ to $y$. Indeed, one of the most basic operations used by shortest path algorithms is the relaxation step. It is defined as follows -

\[
\text{Relax}(u, v) : (u, v) \in E, \quad \text{if } \Delta(v) > \Delta(u) + w(u, v) \text{ then } \Delta(v) = \Delta(v) + w(u, v)
\]

For any vertex $v$, $\Delta(v)$ is an upperbound on the shortest path. Initially it is set to $\infty$ but gradually its value decreases till it becomes equal to $\delta(v)$ which is the actual shortest path distance (from a designated source vertex).

The other property that is exploited by all algorithms is
Observation 9.7 subpath optimality
Let \( s = v_0, v_1, v_2 \ldots v_i \ldots v_j \ldots v_\ell \) be a shortest path from \( v_0 \) to \( v_\ell \). Then for any intermediate vertices, \( v_i, v_j \), the subpath \( v_i, v_{i+2} \ldots v_j \) is also a shortest path between \( v_i \) and \( v_j \).

This follows from a simple argument by contradiction, that otherwise the original path is not the shortest path. Moreover, any shortest path algorithm using the contraction step would compute the shortest path to \( v_j \) before \( v_j \) for \( j > i \). In particular, once the shortest path to \( v_j \) is successfully computed, viz., \( \delta(v_j) = \Delta(v_j) \), then \( \delta(v_{j+1}) = \Delta(v_{j+1}) \) the next time edge \((v_j, v_{j+1})\) is contracted.

9.2.1 Bellman Ford SSSP Algorithm
The Bellman Ford algorithm is essentially based on the following recurrence
\[
\delta(v) = \min_{u \in In(v)} \{ \delta(u) + w(u, v) \}
\]
where \( In(v) \) denotes the set of vertices \( u \in V \) such that \((u, v) \in E \). The shortest path to \( v \) must have one of the incoming edges into \( v \) as the last edge. The algorithm actually maintains upperbounds \( \Delta(v) \) on the distance from the source vertex \( s \) - initially \( \Delta(v) = \infty \) for all \( v \in V - \{s\} \) and \( \Delta(s) = 0 = \delta(s) \). The previous recurrence is recast in terms of \( \Delta \)
\[
\Delta(v) = \min_{u \in In(v)} \{ \Delta(u) + w(u, v) \}
\]
that follows from a similar reasoning. Note that if \( D(u) = \delta(u) \) for any \( u \in In(v) \), then after applying \( relax(u, v) \), \( \Delta(v) = \delta(v) \). The underlying technique is dynamic programming as many vertices may have common predecessors in the shortest path recurrence.

**Algorithm 3:** SSSP \(((V, E), s)\)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textbf{for} ( i = 1 ) to (</td>
</tr>
<tr>
<td>2</td>
<td>\quad \textbf{for} ( \text{all} \ e \in E ) \textbf{do}</td>
</tr>
<tr>
<td>3</td>
<td>\quad \quad \texttt{Relax} \ (e)</td>
</tr>
<tr>
<td>4</td>
<td>\textbf{Output} \ \delta(v) = \Delta(v) \text{ for all } v \in V.</td>
</tr>
</tbody>
</table>

Figure 9.5: Bellman Ford single-source shortest path problem
The correctness of the algorithm follows from the previous discussion and the following critical observation.

**Observation 9.8** After $i$ iterations, all vertices whose shortest paths consist of $i$ edges, have $\Delta(v) = \delta(v)$.

It follows from a straightforward induction on the number of edges in the path with the base case $\delta(s) = 0$ and the definition of relax step.

So, the algorithm finds all shortest paths consisting of at most $n - 1$ edges with $n - 1$ iterations. However, if there is a negative cycle in the graph, then you may require more iterations and in fact, the problem is not well defined any more. We can specify that we will output simple paths (without repeated vertices) but this version is not easy to handle. However, we can use the Bellman-Ford algorithm to detect negative cycles in the given graph (Exercise refCh9.5). Since each iteration involves $O(|E|)$ relax operations - one for every edge, the total running time is bounded by $O(|V| \cdot |E|)$.

To actually compute the shortest path, we keep track of the predecessor of a vertex which is determined by the relaxation step. The shortest path can be constructed by following the predecessor links (Exercise 9.12).

---

2This is equivalent to the longest path problem which is known to be intractable
### 9.2.2 Dijkstra’s SSSP algorithm

If the graph doesn’t have negative weight edges, then we can exploit this feature to design a faster algorithm. When we have only non-negative weights, we can actually determine which vertex has the its $\Delta(v) = \delta(v)$. In the case of Bellman Ford algorithm, at every iteration, at least one vertex had its shortest path computed but we couldn’t identify them. We maintain a partition $U$ and $V - U$ such $s \in U$ where $U$ is the set of vertices $v \in V$ for which $\Delta(v) = \delta(v)$. On the other hand, for non-negative weights, we can make the following claim.

**Observation 9.9** The vertices $v \in V - U$ for which $\Delta(v)$ is minimum, satisfies the property that $\Delta(v) = \delta(v)$.

Suppose for some vertex $v$ that has the minimum label after some iteration, $\Delta(v) > \delta(v)$. Consider a shortest path $s \leadsto x \leadsto y \leadsto v$, where $y \notin U$ and all the earlier vertices in the path $s \leadsto x$ are in $U$. Since $x \in U$, $\Delta(y) \leq \delta(x) + w(x, y) = \delta(y)$. Since all edge weights are non-negative, $\delta(y) \leq \delta(v) < \Delta(v)$ and therefore $\Delta(y) = \delta(y)$ is strictly less than $\Delta(v)$ which contradicts the minimality of $\Delta(v)$.

#### Algorithm 4: SSSP $(V, E, s)$

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$U \leftarrow {s}$ ;</td>
</tr>
<tr>
<td>2</td>
<td><strong>while</strong> $V - U \neq \phi$ <strong>do</strong></td>
</tr>
<tr>
<td>3</td>
<td>$x = \text{arg} \min_{w \in V - U} \Delta(w)$ ;</td>
</tr>
<tr>
<td>4</td>
<td>$\delta(x) = \Delta(x)$ ; Move $x$ to $U$ ;</td>
</tr>
<tr>
<td>5</td>
<td>Relax $(x, y)$ for all edges $(x, y)$ ;</td>
</tr>
</tbody>
</table>

Figure 9.7: Dijkstra’s single source shortest path algorithm

Run the algorithm on the graph given in Figure 9.6 and convince yourself that it doesn’t work. Then convert all the weights to non-negative and try again.

A crucial property exploited by Dijkstra’s algorithm is that along any shortest path $s \leadsto u$, the shortest path-lengths are non-decreasing because of non-negative edge weights. Along similar lines, we can also assert the following

**Observation 9.10** Starting with $s$, the vertices are inserted into $U$ is non-decreasing order of their shortest path lengths.

We can prove it by induction starting with $s - \delta(s) = 0$. Suppose it is true upto iteration $i$, i.e., all vertices $v \in U$ are such that $\delta(v) \leq \delta(x), x \in V - U$. Let $\Delta(u) \in V - U$ be minimum, then we claim that $\Delta(u) = \delta(u)$ (from the previous
This yields a running time of \((\varepsilon, \delta)\) are relaxed - however because of relax operation the steps. Each edge is relaxed exactly once since only the edges incident on vertices in \(\Delta(v)\) of \(\delta(v)\). Therefore yields a running time of \(\mathcal{O}(|V| + |E|)\log |V|\).

### 9.2.3 All Pair Shortest Paths algorithm

We now consider the version of the shortest path problem where for all pairs \(u, w \in V\), we would like to compute \(\delta(u, w)\). We can compute this by using the previous algorithm on each of the possible source vertices. The running time will be \(O(|V| \cdot |V| \cdot |E|)\), i.e., \(O(|V|^3 \cdot |E|)\) steps. For a dense graph having close to \(|V|^2\) edges, this results in \(O(|V|^4)\) running time. We will try to improve this performance significantly.

We start by numbering the vertices arbitrarily with integers \(\{1, 2, \ldots |V| = n\}\). Let us assume that the graph \(G\) is represented using an adjacency matrix \(A_G\) where \(A_G[i, j] = w(i, j)\) which is the weight of the edge \((i, j)\). If there is no edge then \(w(i, j) = \infty\). We define \(D^0_{i,j}\) as the length of the shortest path between vertex \(i\) to vertex \(j\) that does not use any intermediate vertex numbered higher than \(k\) (i.e., \(i, j\) are not included among the intermediate vertices). This restricts the paths for a given value of \(k\) but since all vertices are numbered \([1..n]\), \(\delta(i, j) = D^n_{i,j}\). Moreover, we define \(D^0_{i,j} = w(i, j)\). The following recurrence leads to an efficient dynamic programming based algorithm for all \(i, j \in \{1, 2, \ldots n\}\).

\[
D^k_{i,j} = \begin{cases} 
  w(i, j) & \text{if } k = 0 \\
  \min\{D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}\} & \text{if } 1 \leq k \leq n
\end{cases}
\] (9.2.1)

The reasoning is based on comparing \(D^k_{i,j}\) and \(D^{k-1}_{i,j}\). If the former does not use any vertex numbered \(k\) then \(D^k_{i,j} = D^{k-1}_{i,j}\). Otherwise, the shortest path containing \(k\) comprises of two subpaths - one from \(i\) to \(k\) (that does not use vertices numbered \(k\)) and the remaining path from \(k\) to \(j\) (again that does not use vertex \(k\)). These paths correspond to \(D^{k-1}_{i,k}\) and \(D^{k-1}_{k,j}\) respectively. The reader may also ponder about why \(k\) cannot be visited multiple times in a shortest path between \(i\) to \(j\).

The remaining details of refining the recurrence to an algorithm is left as an exercise (Exercise 9.13). We would like to draw attention to computation of the actual paths. Since each path can be of length \(|V| - 1\) \(^3\) the total length of the paths

\(^3\text{this is again related to the absence of negative cycles}\)
can add up to $\Omega(n^3)$. The reader is encouraged to design such a graph.

Instead we can exploit the subpath optimality to reduce the storage. We will only store the first edge of the path $P_{i,j}$ (the sequence of vertices in the shortest path from $i$ to $j$). Suppose this is $i_1$, then we can find the next vertex with the entry $P_{i_1,j}$. If the shortest path $P_{i,j} = i = i_0, i_1, i_2 \ldots i_m = k$, then we will look up the matrix $m-1$ times which is optimal in terms of the path length.

### 9.3 Maximum flows in graphs

Given a directed graph $G = (V,E)$ and a capacity function $C : E \to \mathbb{R}^+$, and two designated vertices $s$ and $t$, we want to compute a flow function $f : E \to \mathbb{R}^+$ such that

1. **Capacity constraint**
   
   $$ f(e) \leq C(e) \quad \forall e \in E $$

2. **Flow conservation**

   $$ \forall v \in V - \{s,t\}, \sum_{e \in \text{in}(v)} f(e) = \sum_{e \in \text{out}(v)} f(e) $$

   where $\text{in}(v)$ are the edges directed into vertex $v$ and $\text{out}(v)$ are the edges directed out of $v$.

The vertices $s$ and $t$ are often called the source and the sink and the flow is directed out of $s$ and into $t$.

The **outflow** of a vertex $v$ is defined as $\sum_{e \in \text{out}(v)} f(e)$ and the **inflow** into $v$ is given by $\sum_{e \in \text{in}(v)} f(e)$. The net flow is defined as outflow minus inflow $= \sum_{e \in \text{out}(v)} f(e) - \sum_{e \in \text{in}(v)} f(e)$. From the property of flow conservation, net flow is zero for all vertices except $s, t$. For vertex $s$ which is the source, the net flow is positive and for $t$, the net flow is negative.

**Observation 9.11** The net flow at $s$ and the net flow at $t$ are equal in magnitude.

From the flow conservation constraint

$$ \sum_{v \in V - \{s,t\}} \left( \sum_{e \in \text{out}(v)} f(e) - \sum_{e \in \text{in}(v)} f(e) \right) = 0 $$

Let $E'$ be edges that are not incident on $s, t$ (either incoming or outgoing). Then

$$ = \sum_{e \in E'} (f(e) - f(e)) + \left( \sum_{e \in \text{out}(s)} f(e) - \sum_{e \in \text{in}(s)} f(e) \right) + \left( \sum_{e \in \text{out}(t)} f(e) - \sum_{e \in \text{in}(t)} f(e) \right) = 0 $$

127
For an edge $e \in E'$, $f(e)$ is counted once as incoming and once as outgoing which cancel each other. So

$$\sum_{e \in \text{out}(s)} f(e) - \sum_{e \in \text{in}(s)} f(e) = \sum_{e \in \text{in}(t)} f(e) - \sum_{e \in \text{out}(t)} f(e)$$

So the outflow at $s$ equals the inflow at $t$.

Let us denote the net flow at $s$ by $F$ and the maximum flow $f^*$ from $s$ to $t$ is the maximum value of $F$.

Computing maxflow is one of the classic problems in combinatorial optimization with numerous applications. Therefore designing efficient algorithms for maxflow has been pursued by many researchers for many years. Since the constraints and the objective function are linear, we can pose it as a linear program (LP) and use some of the efficient (polynomial time) algorithms for LP. However the algorithms for LP are not known to be strongly polynomial, we will explore more efficient algorithms.

bf Flow augmentation
Consider any $s - t$ path in the graph ignoring the edge direction. If the forward edges are not fully saturated and all the backward edges have non-zero flows, we can increase the flow in the following manner. Let $\Delta$ be the minimum of the residual capacity of the forward edges and the flows of the backward edges. The edge(s) that determines $\Delta$ is called a bottleneck edge of the augmenting path.

By increasing the flow in the forward edges by $\Delta$ and decreasing the backward edges by $\Delta$, we can preserve both the capacity constraints and the flow conservation constraints. In this process the is now increased by $\Delta$. Such a path is called an augmentation path. It is clear that the flow is not maximum if there is an augmenting path. The converse is not immediate, i.e., if there is no augmenting path then the flow is maximum. We will establish this as a consequence of a more general result.

9.3.1 Max Flow Min Cut
An $(S,T)$ cut is defined as a partition of $V$ such that $s \in S$ and $t \in T$. The size of a cut is defined as $\sum_{u \in S, v \in T} C(u,v)$. Note that only the capacities of the forward edges are counted.

**Theorem 9.2 (maxflow-mincut)** The value of the $s - t$ maxflow = $s - t$ mincut.

Consider a flow $f$ such that there is no augmenting path. Let $S^*$ be the set of vertices such that there is an augmenting path from $s$ to $u \in S^*$. By definition, $s \in S^*$ and $t \notin S^*$ and $T^* = V - S^*$.

**Observation 9.12** The forward edges from $S^*$ to $T^*$ are saturated and the backward arcs from $T^*$ to $S^*$ are full.
Otherwise it will contradict our definition of $S^*$ and $T^*$. The net flow from $S^*$ to $T^*$ is

$$
\sum_{e \in \text{out}(S^*)} f(e) = \sum_{e \in \text{in}(T^*)} C(e) = C(S^*, T^*)
$$

This implies that

$$
f^* \geq f = C(S^*, T^*) \text{ mincut}
$$

For any cut $(S, T)$ and any flow $f$, consider

$$
\sum_{v \in S} \left( \sum_{e \in \text{out}(v)} f(e) - \sum_{e \in \text{in}(v)} f(e) \right)
$$

= $\sum_{e \in \text{out}(s)} f(e) - \sum_{e \in \text{in}(s)} f(e)$ since the flow conservation holds at every other vertex in $S$ - this is the net flow out of $s$. By rewriting the the first summation over two sets of edges $E$ and $E'$ corresponding to the cases when both endpoints of $e$ are in $S$ or exactly one end-point is in $S$ (the other is in $T$), we obtain the following

$$
\sum_{e \in E} (f(e) - f(e)) + \sum_{e \in \text{out}(S)} f(e) - \sum_{e \in \text{in}(S)} f(e)
$$

The first term is 0 and the second term equals $C(S, T)$. Since the third term is negative, we can upperbound the expression by $C(S, T)$ for any cut and in particular the mincut, i.e., the maxflow is less than or equal to the mincut. As $f$ is any flow, it implies

$$
\text{maxflow} \leq \text{mincut} \leq C(S^*, T^*)
$$

In conjunction with Equation 9.3.2, we find that $f^* = C(S^*, T^*) = \text{mincut}$.

Since the maxflow corresponds to the situation where no augmenting path is possible, we have proved that

The flow is maximum iff there is no augmenting path.

### 9.3.2 Ford and Fulkerson method

The Ford and Fulkerson strategy for maxflow is directly based on the above result, i.e., we successively find augmenting paths till we can’t find any such path.

How do we find an augmenting path? A residual graph $G(f)$ corresponding to a flow $f$ has the same set of nodes as the original graph and the edges are defined as follows. For an edge $e$ with capacity $C(e)$ and flow $f(e)$ such that $f(e) < C(e)$ generates two edges $e'$ in forward direction and $e''$ in backward direction with capacities
$C(e) - f(e)$ and $f(e)$ respectively. If $f(e) = C(e)$ then only the backward edge is present with capacity $C(e)$. Any s-t path in $G(f)$ is an augmenting path that can be found using a BFS or DFS. If $t$ is disconnected from $s$, then there is no augmenting path. Note that the capacities do not play any role in the path except that zero capacity edges are not present.

Although the Ford Fulkerson method converges since the flow increases monotonically, we do not have a bound on the maximum number of iterations that it takes to converge to the maxflow. Bad examples (taking exponential time) can be easily constructed and actually for irrational capacities, it converge only in the limit!

### 9.3.3 Edmond Karp augmentation strategy

It turns out that if we augment flow along the shortest path (in the unweighted residual network using BFS) between $s$ and $t$, we can prove much superior bounds. The basic property that that enables us to obtain a reasonable bound is the following result.

**Claim 9.2** A fixed edge can become bottleneck in at most $n/2$ iterations.

We will prove the claim shortly. The claim implies that the total number of iterations is $m \cdot n/2$ or $O(|V| \cdot |E|)$ which is polynomial in the input size. Each iteration involves a BFS, yielding an overall running time of $O(n \cdot m^2)$.

### 9.3.4 Monotonicity Lemma and bounding the iterations

Let $s_i^k$ and $t_i^k$ denote the minimum number of edges in a shortest path from $s$ to vertex $i$ after $k$ iterations. We claim that

$$s_i^{k+1} \geq s_i^k \quad \text{and} \quad t_i^{k+1} \geq t_i.$$

We will prove it by contradiction. Suppose $s_i^{k+1} < s_i^k$ for some $k$ and among all such vertices let $s \sim v$ have minimum path length (after $k + 1$ iterations). Consider the last edge in the path, say $(u, v)$. Then

$$s_v^{k+1} = s_u^{k+1} + 1 \quad (9.3.3)$$

since $u$ is on the shortest path. By assumption on minimality of violation,

$$s_u^{k+1} \geq s_u^k \quad (9.3.4)$$

From 9.3.3, it follows that

$$s_v^{k+1} \geq s_u^k + 1 \quad (9.3.5)$$

Consider the flow $f(u, v)$ after $k$ iterations.
Case 1: \( f(u, v) < C(u, v) \) Then there is a forward edge \( u \rightarrow v \) and hence \( s^k_v \leq s^k_u + 1 \)
From Equation 9.3.5 \( s^{k+1}_v \geq s^k_v \). that contradicts our assumption.

Case 2: \( f(u, v) = C(u, v) \) Then \( u \leftarrow v \) is a backward arc. After \( k+1 \) iterations, we have a forward edge \( u \rightarrow v \), so the shortest augmenting path must have passed through the edge \( v \rightarrow u \) (after \( k \) iteration) implying that

\[
s^k_u = s^k_v + 1
\]

Combining with inequality 9.3.5, we obtain \( s^{k+1}_v = s^k_v + 2 \) that contradicts our assumption.

Let us now bound the number of times edge \( (u, v) \) can be a bottleneck for augmentations passing through the edge \( (u, v) \) in either direction. If \( (u, v) \) is critical after \( k \)-iteration in the forward direction then \( s^k_v = s^k_u + 1 \). From monotonicity property \( s^\ell_v \geq s^k_v \), so

\[
s^\ell_v \geq s^k_u + 1 \quad (9.3.6)
\]

Let \( \ell \geq k + 1 \) be the next iteration when an augmenting path passes through \( (u, v) \). Then \( (u, v) \) must be a backward edge and therefore

\[
s^\ell_u = s^\ell_v + 1 \geq s^k_u + 1 + 1 = s^k_v + 2
\]

using inequality 9.3.6. Therefore we can conclude that distance from \( u \) to \( s \) increases by at least 2 every time \( (u, v) \) becomes bottleneck and hence it can become bottleneck for at most \( |V|/2 \) augmentations.

### 9.4 Global Mincut

A cut of a given (connected) graph \( G = (V, E) \) is set of edges which when removed disconnects the graph. An \( s-t \) cut must have the property that the designated vertices \( s \) and \( t \) should be in separate components. A mincut is the minimum number of edges that disconnects a graph and is sometimes referred to as global mincut to distinguish is from \( s-t \) mincut. The weighted version of the mincut problem is the natural analogue when the edges have non-negative associated weights. A cut can also be represented by a set of vertices \( S \) where the cut-edges are the edges connecting \( S \) and \( V - S \).

It was believed for a long time that the mincut is a harder problem to solve than the \( s-t \) mincut - in fact the earlier algorithms for mincuts determined the \( s-t \) mincuts for all pairs \( s, t \in V \). The \( s-t \) mincut can be determined from the \( s-t \)

\(^4\)it may not be a bottleneck edge.
maxflow flow algorithms and over the years, there have been improved reductions of
the global mincut problem to the $s - t$ flow problem, such that it can now be solved
in one computation of $s - t$ flow.

In a remarkable departure from this line of work, first Karger, followed by Karger
and Stein developed faster algorithms (than maxflow) to compute the mincut with
high probability. The algorithms produce a cut that is very likely the mincut.

9.4.1 The contraction algorithm

The basis of the algorithm is the procedure contraction described below. The funda-
mental operation $\text{contract}(v_1, v_2)$ replaces vertices $v_1$ and $v_2$ by a new vertex $v$ and
assigns the set of edges incident on $v$ by the union of the edges incident on $v_1$ and $v_2$.
We do not merge edges from $v_1$ and $v_2$ with the same end-point but retain them as
multiple edges. Notice that by definition, the edges between $v_1$ and $v_2$ disappear.

\begin{verbatim}
Procedure Partition (t)
Input: A multigraph $G = (V, E)$
Output: A $t$ partition of $V$

Repeat until $t$ vertices remain
    choose an edge $(v_1, v_2)$ at random
    $\text{contract}(v_1, v_2)$

$\text{contract}(u, v)$: Merge vertices $u$ and $v$ into $w$ such that
all neighbours of $u$ and $v$ are now neighbours of $w$.
\end{verbatim}

Procedure $\text{Partition}(2)$ produces a 2-partition of $V$ which defines a cut. If it is a
mincut then we are done. There are two issues that must be examined carefully.

1. How likely is it that the cut is a mincut?
2. How do we know that it is a mincut?

The second question addresses a more general question, namely, how does one verify
the correctness of a Monte Carlo randomized algorithm? In most cases there are no
efficient verification procedures and we can only claim the correctness in a probabilis-
tic sense. In our context, we will show that the contraction algorithm will produce
a mincut with probability $p$, so that, if we run the algorithm $\frac{1}{p}$ times we expect to
see the mincut at least once. Among all the cuts that are output in $O(\frac{1}{p})$ runs of the
algorithm, we choose the one with the minimum cut value. If the minimum cut had
been produced in any of the independent runs, we will obtain the mincut.
9.4.2 Probability of mincut

Using the observation that, in an \(n\)-vertex graph with a mincut value \(k\), the minimum degree of a vertex is \(k\), the probability that one of the mincut edge is contracted is \(\leq \frac{k}{kn/2} = \frac{2}{n}\).

Given a specific mincut \(C\), we estimate the probability that \(C\) is not outputted. If \(C\) is output, then it means that none of the edges of \(C\) has ever been contracted.

Let \(A(i)\) denote the event that an edge of \(C\) is contracted in the \(i^{th}\) iteration and let \(E(i)\) denote the event that no edge of \(C\) is contracted in any of the first \(i\) iterations. If \(n_i\) is the number of vertices after \(i\) iterations (initially \(n_0 = n\)) we have \(n_i = n - i\).

We have seen that \(\Pr[\overline{A(1)}] \geq 1 - 2/n\) and similarly, \(\Pr[\overline{A(i)}|E(i-1)] \geq 1 - 2/n_{i-1}\).

Then, using the property of conditional probability

\[
\Pr[E(i)] = \Pr[\overline{A(i)} \cap E(i-1)] = \Pr[\overline{A(i)}|E(i-1)] \cdot \Pr[E(i-1)]
\]

where \(\overline{A}\) denotes the complement of event \(A\). We can use the above equation inductively obtain

\[
\Pr[E(i)] \geq \prod_{i=1}^{n-i} (1 - 2/n_{i-1}) \\
= \prod_{i=1}^{n-i} (1 - \frac{2}{n_{i+1}}) \\
\geq \frac{t(t-1)\ldots(t-n+1)}{n(n-1)\ldots(n-i)}
\]

Claim 9.3 The probability that a specific mincut \(C\) survives at the end of Partition\((t)\) is at least \(\frac{t(t-1)\ldots(t-n+1)}{n(n-1)\ldots(n-i)}\).

Therefore Partition (2) produces a mincut with probability \(\Omega\left(\frac{1}{n^2}\right)\). Repeating the above algorithm \(O(n^2)\) times would ensure that the min cut is expected to be the output at least once. If each contraction can be performed in \(t(n)\) time then the expected running time is \(O(t(n) \cdot n \cdot n^2)\).

Exercise 9.3 By using an adjacency matrix representation, show that the contraction operation can be performed in \(O(n)\) steps.

We now address the problem of choosing a random edge using the above data structure.

Claim 9.4 An edge \(E\) can chosen uniformly at random at any stage of the algorithm in \(O(n)\) steps.

We first present a method to pick an edge randomly in \(O(n)\) time. The selection works as follows.

\(\text{We will prove it only for the unweighted version but the proof can be extended using multiset arguments.}\)
• Select a vertex $v$ at random with probability $\frac{\deg(v)}{\sum_{u \in V} \deg(u)} = \frac{\deg(v)}{2|E|}$

• Select an edge $(v, w)$ at random with probability $\frac{\#E(v, w)}{\sum_{z \in N(v)} \#E(v, z)} = \frac{\#E(v, w)}{\deg(v)}$

where $\#E(u, v)$ denotes the number of edges between $u$ and $v$ and $N(v)$ is the set of neighbours of $v$.

Hence, the probability of choosing any edge $(v, w)$ is given by

$$\frac{\#E(v, w)}{\deg(v)} \cdot \frac{\deg(v)}{2|E|} + \frac{\#E(w, v)}{\deg(w)} \cdot \frac{\deg(w)}{2|E|} = \frac{\#E(v, w)}{|E|}$$

Thus, the above method picks edges with probability that is proportional to the number of edges between $v$ and $w$. When there are no multiple edges, all edges are equally likely to be picked. For the case of integer weights, the above derivation works directly for weighted sampling. By using an adjacency matrix $M$ for storing the graph where $M_{v, w}$ denotes the number of edges between $v$ and $w$ allows us to merge vertices $v$ and $w$ in $O(n)$ time.

**Exercise 9.4** Describe a method to implement Partition(2) in $O(m \log n)$ steps. This will be faster for sparse graphs.

Hint: Can you use union-find?

### 9.5 Matching

Matching is a classical combinatorial problem in graphs and can be related to a number of natural problems in real life. Given a graph $G = (V, E)$, a matching $\mathcal{M} \subset E$ is a subset of edges that do not have any common end-points in $V$. A maximal matching $\mathcal{M}'$ is such that there is no $e \in E - \mathcal{M}'$ such that $\mathcal{M}' \cup \{e\}$ is a matching, i.e., $\mathcal{M}'$ cannot be augmented. It is easy to see that a maximal matching can be easily constructed using a greedy approach.

**Exercise 9.5** Design a linear time algorithm for maximal matching.

A maximum matching is far more challenging problem that can be expressed as the following optimization problem.

$$\max \sum_{e \in E} c_e \cdot x_e \quad s.t. A^{V\times|E|}X \leq [1, 1..]^T$$
where \( x_e \in \{0, 1\} \) correspond to inclusion of each edge and \( c_e \) is the weight of each edge. \( A \) denotes the edge-vertex incidence matrix and the objective function maximizes the sum of weights of the matched edges. When \( c_e = 1 \), is called the maximum cardinality matching.

For the special case of a bipartite graph, the edge-vertex matrix has a very nice property known as unimodularity\(^6\) that enables one to use linear programming as an effective method to solve this problem.

There are however direct combinatorial algorithm for solving the matching problem that are more efficient. The notion of augmenting paths can be extended to the problem of matching (more naturally to the bipartite graphs) and polynomial time algorithms can be designed. An augmenting path begins from an unmatched vertex and traces an alternating path of matched and unmatched edges ending with an unmatched vertex. Therefore, an augmenting path has odd number of edges and increases the size of matching by one by including all the unmatched edges and removing the matched edges of the path. The following claim analogous to the flow problem forms the basis of all matching algorithms.

**Claim 9.5** A matching is maximum (cardinality) iff there is no augmenting path.

The necessary part of the claim is obvious. For the sufficiency, the following notion of symmetric difference of two matchings \( M \) and \( M' \) is useful. Define \( M' \oplus M = (M' - M) \cup (M - M') \).

**Exercise 9.6** Prove that \( M' \oplus M \) consists of disjoint alternating cycles and paths.

If \( M' \) is maximum and \( M \) is not, then using the above result, argue that there must be some augmenting path in \( M \).

It is not difficult to prove that any maximal matching is at least half the size of a maximum cardinality matching. There is a useful generalization of this observation using the notion of augmenting paths.

**Claim 9.6** Let \( M \) be a matching such that there is no augmenting path of length \( \leq 2k - 1 \). If \( M' \) is a maximum matching then

\[
|M| \geq |M'| \cdot \frac{k}{k + 1}
\]

From our previous observation, the symmetric difference \( M \oplus M' \) consists of a set \( \mathcal{P} \) of disjoint alternating paths and cycles (alternating between edges of \( M \) and \( M' \)) such that each path has about half the edges from \( M \). If the shortest augmenting path is of length \( 2k + 1 \) (it must have odd length starting and ending with edges in

\(^6\) Roughly speaking, the polytope of feasible solution has integral coordinates
$M'$), then there are at least $k$ edges of $M$ in each such augmenting path. It follows that $|M' - M| \leq |M' - M| \leq |M' \oplus M| \leq |\mathcal{P}|$. Therefore $|M'| \leq |M| + |M'|/k$ implying the claim.\footnote{A maximal matching has no length 1 augmenting path and hence it is within factor 2 of maximum matching.}

**Exercise Problems**

**Exercise 9.7** Based on suitable modifications of the DFS algorithm, design a linear time algorithm for topological sorting or conclude that the given graph is not a DAG.

**Exercise 9.8** Prove rigorously the correctness of the algorithm given in Figure 9.3.

**Exercise 9.9** Show that the above relation defines an equivalence relation on edges. Moreover, the equivalence classes are precisely the BCC (as defined by the vertex connectivity).

**Exercise 9.10** Show how to compute the $\text{LOW}(v)$ $v \in V$ along with the DFS numbering in linear time using the recursive definition of $\text{LOW}$. This will be done simultaneously with the depth first search.

**Exercise 9.11** Describe an efficient algorithm to detect negative cycle in graph.

**Exercise 9.12** If there is no negative cycle, show that the predecessors form a tree (which is called the shortest-path tree).

**Exercise 9.13** (a) Using the recurrence in Equation 9.2.1 design and analyze the dynamic programming for computing shortest paths. (b) How would you modify the algorithm to report the shortest paths?

**Exercise 9.14** Show that the BCC of an undirected graph can be computed in $O(|V| + |E|)$ steps using DFS.

**Exercise 9.15** Graph theory

1. Show that in any graph there are at least two vertices of the same degree.

2. Given a degree sequence $d_1, D_2 \ldots d_n$ such that $\sum_i d_i = 2n - 2$, construct a tree whose vertices have the above degrees.
3. Show that in a complete graph of six vertices where edges are colored red or blue, there is either a red or a blue triangle.

**Exercise 9.16** Given a directed acyclic graph, that has maximal path length \( k \), design an efficient algorithm that partitions the vertices into \( k + 1 \) sets such that there is no path between any pair of vertices in a set.

**Exercise 9.17** Given an undirected graph, describe an algorithm to determine if it contains an even-length cycle. Can you do the same for odd-length cycle?

**Exercise 9.18** Given an undirected connected graph \( G \), define the Biconnected Component Graph \( H \) as follows. For each BCC of \( G \) and articulation point of \( G \), there is a vertex in \( H \). There is an edge between vertices \( x \) and \( y \) in \( H \) if \( x \) is articulation point in the BCC \( y \).

1. Prove that \( H \) is a tree.
2. Using \( H \) (or otherwise), design an efficient algorithm that adds the minimal number of edge to \( G \) to make it biconnected.

**Exercise 9.19** A directed graph is Eulerian if the in degree equals out degree for every vertex. Show that an Eulerian graph admits a tour where every edge is visited exactly once. Design an efficient (linear time) algorithm to find such a tour.

**Exercise 9.20** An \( n \)-vertex undirected graph is a scorpion if it has a vertex of degree 1 (the string) connected to a vertex of degree 2 (the tail) connected to a vertex of degree \( n - 2 \) (the body) which is connected to the remaining \( n - 3 \) vertices (the feet). Some of the feet may be connected among themselves. Give an \( O(n) \) algorithm to check if a given \( n \times n \) adjacency matrix represents a scorpion.

**Exercise 9.21** Instead of a DFS tree, starting from an arbitrary spanning tree, redesign the bi-connectivity algorithm. Your algorithm should run in linear time.

**Exercise 9.22** Given an undirected graph, orient the edges so that the resulting graph is strongly connected. When is it possible? Design a linear time algorithm for this problem.

**Exercise 9.23** Find a maximum subgraph of \( G = (V, E) \) that has degrees of each vertex is at least \( k \).

**Exercise 9.24** Describe an efficient algorithm to find the girth of a given undirected graph. The girth is defined as the length of the smallest cycle.
Chapter 10

NP Completeness and Approximation Algorithms

Let $\mathcal{C}()$ be a class of problems defined by some property. We are interested in characterizing the hardest problems in the class, so that if we can find an efficient algorithm for these, it would imply fast algorithms for all the problems in $\mathcal{C}$. The class that is of great interest to computer scientists is the class $\mathcal{P}$ that is the set of problems for which we can design polynomial time algorithms. A related class is $\mathcal{NP}$, the class of problems for which non-deterministic\(^1\) polynomial time algorithms can be designed.

More formally,

$\mathcal{P} = \bigcup_{i \geq 1} \mathcal{C}(T^\mathcal{P}(n^i))$

where $\mathcal{C}(T^\mathcal{P}(n^i))$ denotes problems for which $O(n^i)$ time algorithms can be designed.

$\mathcal{NP} = \bigcup_{i \geq 1} \mathcal{C}(T^{NP}(n^i))$

where $T^{NP}()$ represent non-deterministic time. Below we formalize the notion of hardest problems and what is known about the hardest problems. It may be noted that the theory developed in the context of $\mathcal{P}$ and $\mathcal{NP}$ is mostly confined to decision problems, i.e., those that have a Yes/No answer. So we can think about a problem $P$ as a subset of integers as all inputs can be mapped to integers and hence we are solving the membership problem for a given set.

**Exercise 10.1** Prove the following
(i) If $P \in \mathcal{P}$ then complement of $P$ is also in $\mathcal{P}$.
(ii) If $P_1, P_2 \in \mathcal{P}$ then $P_1 \cup P_2 \in \mathcal{P}$ and $P_1 \cap P_2 \in \mathcal{P}$.

\(^1\)We will define it more formally later. These algorithms have a choice of more than one possible transitions at any step that does not depend on any deterministic factor.
10.1 Classes and reducibility

The intuitive notion of reducibility between two problems is that if we can solve one we can also solve the other. Reducibility is actually an asymmetric relation and also entails some details about the cost of reduction. We will use the notation $P_1 \leq_R P_2$ to denote that problem $P_1$ is reducible to $P_2$ using resource (time or space as the case may be) to problem $P_2$. Note that it is not necessary that $P_2 \leq_R P_1$.

In the context of decision problems, a problem $P_1$ is many-one reducible to $P_2$ if there is a many-to-one function $g()$ that maps an instance $I_1 \in P_1$ to an instance $I_2 \in P_2$ such that the answer to $I_2$ is YES iff the answer to $I_1$ is YES.

In other words, the many-to-one reducibility maps YES instances to YES instances and NO instances to NO instances. Note that the mapping need not be 1-1 and therefore reducibility is not a symmetric relation.

Further, if the mapping function $g()$ can be computed in polynomial time then we say that $P_1$ is polynomial-time reducible to $P_2$ and is denoted by $P_1 \leq_{poly} P_2$.

The other important kind of reduction is logspace reduction and is denoted by

$$P_1 \leq_{log} P_2.$$

**Claim 10.1** If $P_1 \leq_{log} P_2$ then $P_1 \leq_{poly} P_2$.

This follows from a more general result that any finite computational process that uses space $S$ has a running time bounded by $2^S$. A rigorous proof is not difficult but beyond the scope of this discussion.

**Claim 10.2** The relation $\leq_{poly}$ is transitive, i.e., if $P_1 \leq_{poly} P_2$ and $P_2 \leq_{poly} P_3$ then $P_1 \leq_{poly} P_3$.

From the first assertion there must exist polynomial time computable reduction functions, say $g()$ and $g'()$ corresponding to the first and second reductions. So we can define a function $g'(g)$ which is a composition of the two functions and we claim that it satisfies the property of a polynomial time reduction function from $P_1$ to $P_3$. Let $x$ be an input to $P_1$, then $g(x) \in P_2$ iff $x \in P_1$. Similarly $g'(g(x)) \in P_3$ iff $g(x) \in P_2$ implying $g'(g(x)) \in P_3$ iff $x \in P_1$. Moreover the composition of two polynomials is a polynomial, so $g'(g(x))$ is polynomial time computable.

A similar result on transitivity also holds for log-space reduction, although the proof is more subtle.

---

2This is a short form of saying that $g(x)$ is an YES instance.
Claim 10.3 If $\Pi_1 \leq_{\text{poly}} \Pi_2$ then

(i) If there is a polynomial time algorithm for $\Pi_2$ then there is a polynomial time algorithm for $\Pi_1$.

(ii) If there is no polynomial time algorithm for $\Pi_1$, then there cannot be a polynomial time algorithm for $\Pi_2$.

Part (ii) is easily proved by contradiction. For part (i), if $p_1(n)$ is the running time of $\Pi_1$ and $p_2$ is the time of the reduction function, then there is an algorithm for $P_{\Pi_1}$ that takes $p_1(p_2(n))$ steps where $n$ is the input length for $\Pi_1$.

A problem $\Pi$ is called NP-hard under polynomial reduction if for any problem $\Pi' \in \mathcal{NP}$, $\Pi' \leq_{\text{poly}} \Pi$.

A problem is NP-complete (NPC) if it is NP-hard and $\Pi \in \mathcal{NP}$.

Therefore these are problems that are hardest within the class $\mathcal{NP}$.

Exercise 10.2 If problems $A$ and $B$ are NPC, then $A \leq_{\text{poly}} B$ and $B \leq_{\text{poly}} A$.

From the previous exercise, these problems form a kind of equivalent class with respect to polynomial time reductions. However, a crucial question that emerges at this juncture is: Do NPC problems actually exist? A positive answer to this question led to the development of one of the most fascinating areas of Theoretical Computer Science and will be addressed in the next section.

So far, we have only discussed many-one reducibility that hinges on the existence of a many-one polynomial time reduction function. There is another very useful and perhaps more intuitive notion of reducibility, namely, Turing reducibility. The many-to-one reduction may be thought of as using one subroutine call of $P_2$ to solve $P_1$ (when $P_1 \leq_{\text{poly}} P_2$) in polynomial time, if $P_2$ has a polynomial time algorithm. Clearly, we can afford a polynomial number of subroutine calls to the algorithm for $P_2$ and still get a polynomial time algorithms for $P_1$. In other words, we say that $P_1$ is Turing-reducible to $P_2$ if a polynomial time algorithm for $P_2$ implies a polynomial time algorithm for $P_1$. Moreover, we do not require that $P_1, P_2$ be decision problems. Although, this may seem to be the more natural notion of reducibility, we will rely on the more restrictive definition to derive the results.

10.2 Cook Levin theorem

Given a boolean formula in boolean variables, the satisfiability problem is an assignment of the truth values of the boolean variables that can make the formula evaluate to TRUE (if it is possible). If the formula is in a conjunctive normal form (CNF)\(^3\),

\(^3\)A formula, that looks like $(x_1 \lor x_2..) \land (x_i \lor x_j \lor ..) \land \ldots (x_\ell \lor \ldots x_n)$
then the problem is known as CNF Satisfiability. Further, if we restrict the number of variables in each clause to be exactly $k$ then it is known as the $k$-CNF Satisfiability problem. A remarkable result attributed to Cook and Levin says the following

**Theorem 10.1** The CNF Satisfiability problem is NP Complete under polynomial time reductions.

To appreciate this result, you must realize that there are potentially infinite number of problems in the class $NP$, so we cannot explicitly design a reduction function. Other than the definition of $NP$ we have very little to rely on for a proof of the above result. A detailed technical proof requires that we define the computing model very precisely - it is beyond the scope of this discussion. Instead we sketch an intuition behind the proof.

Given an arbitrary problem $\Pi \in NP$, we want to show that $\Pi \leq_{poly} CNF - SAT$. In other words, given any instance of $\Pi$, say $I_\Pi$, we would like to define a boolean formula $B(I_\Pi)$ which has a satisfiable assignment iff $I_\Pi$ is a YES instance. Moreover the length of $B(I_\Pi)$ should be polynomial time constructable (as a function of the length of $I_\Pi$).

A computing machine is a transition system where we have

(i) An initial configuration
(ii) A final configuration that indicates whether or not the input is a YES or a NO instance
(iii) A sequence of intermediate configuration $S_i$ where $S_{i+1}$ follows from $S_i$ using a valid transition. In a non-deterministic system, there can be more than one possible transition from a configuration. A non-deterministic machine *accepts* a given input iff there is some valid sequence of configurations that verifies that the input is a YES instance.

All the above properties can be expressed in propositional logic, i.e., by an unquantified boolean formula in a CNF. Using the fact that the number of transitions is polynomial, we can bound the size of this formula by a polynomial. The details can be quite messy and the interested reader can consult a formal proof in the context of Turing Machine model. Just to give the reader a glimpse of the kind of formalism used, consider a situation where we want to write a propositional formula to assert that a machine is in exactly one of the $k$ states at any given time $1 \leq i \leq T$. Let us use boolean variables $x_{1,i}, x_{2,i} \ldots x_{k,i}$ where $x_{j,i} = 1$ iff the machine is in state $j$ at time $i$. We must write a formula that will be a conjunction of two two conditions

(i) At least one variable is true at any time $i$:

$$ (x_{1,i} \lor x_{2,i} \ldots x_{k,i}) $$
At most one variable is true:

\[(x_{1,i} \Rightarrow \bar{x}_{2,i} \land \bar{x}_{3,i} \ldots \land \bar{x}_{k,i}) \land (x_{2,i} \Rightarrow \bar{x}_{1,i} \land \bar{x}_{3,i} \ldots \land \bar{x}_{k,i}) \ldots \land (x_{k,i} \Rightarrow \bar{x}_{1,i} \land \bar{x}_{2,i} \ldots \land \bar{x}_{k-1,i})\]

where the implication \(a \Rightarrow b\) is equivalent to \(\bar{a} \lor b\).

A conjunction of the above formula over all \(1 \leq i \leq T\) has a satisfiable assignment of \(x_{j,i}\) iff the machine is in exactly one state (not necessarily the same state) at each of the time instances. The other condition should capture which states can succeed a given state.

We have argued that \(CNF-SAT\) is NP-hard. Since we can guess an assignment and verify the truth value of the Boolean formula, in linear time, we can claim that \(CNF-SAT\) is in \(NP\).

10.3 Common NP complete problems

To prove that a given problem \(P\) is NPC, the standard procedure is to establish that

(i) \(P \in NP\) : This is usually the easier part.

(ii) \(CNF-SAT \leq_{poly} P\). We already know that any \(P' \in NP\), \(P' \leq_{poly} CNF-SAT\). So by transitivity, \(P' \leq_{poly} P\) and therefore \(P\) is NPC.

The second step can be served by reducing any known NPC to \(P\). Some of the earliest problems that were proved NPC include (besides CNF-SAT)

- 3D Matching
- Three colouring of graphs
- Equal partition of integers
- Maximum Clique /Independent Set
- Hamilton cycle problem
- Minimum set cover

10.3.1 Other important complexity classes

While the classes \(P\) and \(NP\) hogs the maximum limelight in complexity theory, there are many other related classes in their own right.
• $\text{co-}NP$ A problem whose complement is in $\mathcal{NP}$ belongs to this class. If the problem is in $\mathcal{P}$, then the complement of the problem is also in $\mathcal{P}$ and hence in $\mathcal{NP}$. In general we can’t say much about the relation between $\mathcal{P}$ and $\text{co-}\mathcal{NP}$. In general, we can’t even design an NP algorithm for a problem in $\text{co-}\mathcal{NP}$, i.e. these problems are not efficiently verifiable. For instance how would you verify that a boolean formula is unsatisfiable (all assignments make it false)?

**Exercise 10.3** Show that the complement of an NPC problem is complete for the class $\text{co-}\mathcal{NP}$ under polynomial time reduction.

**Exercise 10.4** What would it imply if an NPC problem and its complement are polynomial time reducible to eachother?

• $\mathcal{PSPACE}$ The problems that run in polynomial space (but not necessarily polynomial time). The satisfiability of Quantified Boolean Formula (QBF) is a complete problem for this class.

• **Randomized classes** Depending on the type of randomized algorithms (mainly Las Vegas or Monte Carlo), we have the following important classes
  
  – $\mathcal{RP}$ : Randomized Polynomial class of problems are characterized by (Monte Carlo) randomized algorithms $A$ such that
    
    If $x \in L \Rightarrow \Pr[A\text{ accepts } x] \geq 1/2$
    
    If $x \notin L \Rightarrow \Pr[A\text{ accepts } x] = 0$
    
    These algorithms can err on one side.

  – $\mathcal{BPP}$ When a randomized algorithm is allowed to err on both sides
    
    If $x \in L \Rightarrow \Pr[A\text{ accepts } x] \geq 1/2 + \varepsilon$
    
    If $x \notin L \Rightarrow \Pr[A\text{ accepts } x] \leq 1/2 - \varepsilon$
    
    where $\varepsilon$ is a fixed non zero constant.

  – $\mathcal{ZPP}$ Zero Error Probabilistic Polynomial Time : These are the Las Vegas kind that do not have any errors in the answer but the running time is expected polynomial time.

One of the celebrated problems, involving randomized algorithms is

$$\mathcal{BPP} \subset \mathcal{NP}?$$

143
10.4 Combating hardness with approximation

Since the discovery of NPC problems in early 70’s, algorithm designers have been wary of spending efforts on designing algorithms for these problems as it is considered to be a rather hopeless situation without a definite resolution of the $P = NP$ question. Unfortunately, a large number of interesting problems fall under this category and so ignoring these problems is also not an acceptable attitude. Many researchers have pursued non-exact methods based on heuristics to tackle these problems based on heuristics and empirical results. Some of the well known heuristics are simulated annealing, neural network based learning methods, genetic algorithms. You will have to be an optimist to use these techniques for any critical application.

The accepted paradigm over the last decade has been to design polynomial time algorithms that guarantee near-optimal solution to an optimization problem. For a maximization problem, we would like to obtain a solution that is at least $f \cdot OPT$ where $OPT$ is the value of the optimal solution and $f \leq 1$ is the approximation factor for the worst case input. Likewise, for minimization problem we would like a solution no more than a factor $f \geq 1$ larger than $OPT$. Clearly the closer $f$ is to 1, the better is the algorithm. Such algorithm are referred to as Approximation algorithms and there exists a complexity theory of approximation. It is mainly about the extent of approximation attainable for a certain problem.

For example, if $f = 1 + \varepsilon$ where $\varepsilon$ is any user defined constant, then we way that the problem has a Polynomial Time Approximable Scheme (PTAS). Further, if the algorithm is polynomial in $1/\varepsilon$ then it is called FPTAS (Fully PTAS). The theory of hardness of approximation has yielded lower bounds (for minimization and upper bounds for maximization problems) on the approximations factors for many important optimization problems. A typical kind of result is that Unless $P = NP$ we cannot approximate the set cover problem better than $\log n$ in polynomial time.

In this section, we give several illustrative approximation algorithms. One of the main challenges in the analysis is that even without the explicit knowledge of the optimum solutions, we can still prove guarantees about the quality of the solution of the algorithm.

10.4.1 Equal partition

Given $n$ integers $S = \{z_1, z_2, \ldots, z_n\}$, we want to find a partition $S_1, S - S_1$, such that $| (\sum_{x \in S_1} x) - (\sum_{x \in S - S_1} x) |$ is minimized. A partition is balanced if the above difference is zero.

---

4The reader must realize that our inability to compute the actual solutions makes it difficult to evaluate these methods in in a general situation.
Let $B = \sum_i z_i$ and consider the following a generalization of the problem, namely, the subset sum problem. For a given integer $K \leq B$, is there a subset $R \subset S$ such that the elements in $R$ sum up to $K$.

Let $S(j, r)$ denote a subset of $\{z_1, z_2 \ldots z_j\}$ that sums to $r$ - if no such subset exists then we define it as $\phi$ (empty subset). We can write the following recurrence

$$S(j, r) = S(j-1, r-z_j) \cup z_j \text{ if } z_j \text{ is included or } S(j-1, r) \text{ if } z_j \text{ is not included or } \phi \text{ not possible}$$

Using the above dynamic programming formulation we can compute $S(j, r)$ for $1 \leq j \leq n$ and $r \leq B$. You can easily argue that the running time is $O(n \cdot B)$ which may not be polynomial as $B$ can be very large.

Suppose, we are given an approximation factor $\varepsilon$ and let $A = \lceil \frac{n}{\varepsilon} \rceil$ so that $\frac{1}{\varepsilon} \leq \frac{1}{\varepsilon}/n$. Then we define a new scaled problem with the integers scaled as $z'_i = \lceil \frac{z_i}{A} \rceil$ and let $r' = \lfloor \frac{r}{z/A} \rfloor$ where $z$ is the maximum value of an integer that can participate in the solution $^5$.

Let us solve the problem for $\{z'_1, z'_2 \ldots z'_n\}$ and $r'$ using the previous dynamic programming strategy and let $S'_o$ denote the optimal solution for the scaled problem and let $S'_o$ be the solution for the original problem. Further let $C$ and $C'$ denote the cost function for the original and the scaled problems respectively. The running time of the algorithm is $O(n \cdot r')$ which is $O(\frac{1}{\varepsilon}n^2)$. We would like to show that the cost of $C(S'_o)$ is $\geq (1 - \varepsilon)C(S_o)$. For any $S'' \subset S$

$$C(S'') \cdot \frac{n}{\varepsilon z} \geq C'(S'') \geq C(S'') \cdot \frac{n}{\varepsilon z} - |S''|$$

So

$$C(S'_o) \geq C'(S'_o) \cdot \frac{\varepsilon z}{n} \geq C'(S_o) \cdot \frac{\varepsilon z}{n} \geq \left(C(S_o) \cdot \frac{n}{\varepsilon z} - |S_o| \right) \cdot \frac{\varepsilon z}{n} = C(S_o) - |S_o| \cdot \frac{\varepsilon z}{n}$$

The first and the third inequality follows from the previous bound and the second inequality follows from the optimality of $S'_o$ wrt $C'$. Since $C(S_o) \geq \frac{z|S_o|}{n}$ and so

$$C(S'_o) \geq (1 - \varepsilon)C(S_o)$$

### 10.4.2 Greedy set cover

Given a ground set $S = \{x_1, x_2 \ldots x_n\}$ and a family of subsets $S_1, S_2 \ldots S_m$, $S_i \subset S$, we want to find a minimum number of subsets from the family that covers all elements of $S$. If $S_i$ have associated weights $C()$, then we try to minimize the total weight of the set-cover.

---

$^5$It is a lower bound to the optimal solution - for the balanced partition, it is the maximum integer less than $B/2$. 

145
In the greedy algorithm, we pick up a subset that is most cost-effective in terms of the cost per unchosen element. The cost-effectiveness of a set $U$ is defined by 
\[
\frac{C(U)}{|U-V|}
\] 
where $V \subset S$ is the set of elements already covered. We do this repeatedly till all elements are covered.

Let us number the elements of $S$ in the order they were covered by the greedy algorithm (wlog, we can renumber such that they are $x_1, x_2 \ldots$). We will apportion the cost of covering an element $e \in S$ as 
\[
w(e) = \frac{C(U)}{|U-V|}
\] 
where $e$ is covered for the first time by $U$. The total cost of the cover is 
\[= \sum_i w(x_i)\]

**Claim 10.4**
\[w(x_i) \leq \frac{C_o}{n-i+1}\]
where $C_o$ is the cost of an optimum cover.

In the iteration when $x_i$ is considered, the number of uncovered elements is at least $n-i+1$. The greedy choice is more cost effective than any left over set of the optimal cover. Suppose the cost-effectiveness of the best set in the optimal cover is 
\[C'/U' = \min \left\{ \frac{C(S_{i_1})}{S_{i_1} - S'} \cdots \frac{C(S_{i_k})}{S_{i_k} - S'} \right\}\] 
where $S_{i_1}, S_{i_2} \ldots S_{i_k}$ forms a minimum set cover and $S'$ is the set of covered elements in iteration $i$. Since 
\[C'/U' \leq \frac{C(S_{i_1}) + \ldots + C(S_{i_k})}{(S_{i_1} - S') + \ldots + (S_{i_k} - S')} \leq \frac{C_o}{n-i+1}\]
and the numerator is bounded by $C_o$ and the denominator is more than $n-i+1$, it follows that 
\[w(x_i) \leq \frac{C_o}{n-i+1}\]

Thus the cost of the greedy cover is 
\[= \sum_i \frac{C_o}{n-i+1}\] 
which is bounded by $C_o \cdot H_n$. Here 
\[H_n = \frac{1}{n} + \frac{1}{n-1} + \ldots + 1\]

**Exercise 10.5** Formulate the Vertex cover problem as an instance of set cover problem.

Analyze the approximation factor achieved by the following algorithm. Construct a maximal matching of the given graph and consider the union $C$ of the end-points of the matched edges. Prove that $C$ is a vertex cover and the size of the optimal cover is at least $C/2$. So the approximation factor achieved is better than the general set cover.

**10.4.3 The metric TSP problem**

If the edges of the graph satisfies triangle inequality, i.e., for any three vertices $u, v, w \quad C(u,v) \leq C(u,w) + C(w,v)$, then we can design an approximation algorithm for the TSP problem as follows.

---

146
Metric TSP on graphs

Input: A graph \( G = (V, E) \) with weights on edges that satisfy triangle inequality.

1. Find a Minimum Spanning Tree \( T \) of \( G \).
2. Double every edge - call the resulting graph \( E' \) and construct an Euler tour \( T' \).
3. In this tour, try to take shortcuts if we have visited a vertex before.

Claim 10.5 The length of this tour no more than twice that of the optimal tour.

\( MST \leq TSP \), therefore \( 2 \cdot MST \leq 2 \cdot TSP \). Since shortcuts can only decrease the tour length (because of the triangle inequality), the tour length is no more than twice that of the optimal tour.

10.4.4 Three colouring

We will rely on the following simple observation. If a graph is three colourable, its neighbourhood must be triangle free (or else the graph will contain a four-clique) i.e., it must be 2 colourable,

Observation 10.1 A graph that has maximum degree \( \Delta \) can be coloured using \( \Delta + 1 \) colours using a greedy strategy.

Use a colour that is different from its already coloured neighbours.

Given a 3-colourable graph \( G = (V, E) \), separate out vertices that have degrees \( \geq \sqrt{n} \) - call this set \( H \). Remove the set \( H \) and its incident edges and denote this graph by \( G' = (V', E') \). Note that \( G' \) is 3-colourable and all vertices have degrees less than \( \sqrt{n} \) and so from our previous observation, we can easily colour using \( \sqrt{n} + 1 \) colours. Now, reinsert the the vertices of \( H \) and use an extra \( |H| \) colours to complete the colouring. Since \( |H| \leq \sqrt{n} \), we have used at most \( 2\sqrt{n} \) colours.

It is a rather poor approximation since we have used significantly more colours than three. However, it is known that unless \( P = NP \), any polynomial time colouring algorithm will use \( \Omega(n^\epsilon) \) colours for some fixed \( \epsilon > 0 \).

10.4.5 Maxcut

Problem Given a graph \( G = (V, E) \), we want to partition the vertices into sets \( U, V - U \) such that the number of edges across \( U \) and \( V - U \) is maximized. There is a
corresponding weighted version for a weighted graph with a weight function \( w : E \to \mathbb{R} \).

We have designed a polynomial time algorithm for mincut but the maxcut is an NP-hard problem. Let us explore a simple idea of randomly assigning the vertices to one of the partitions. For any fixed edge \((u, v) \in E\), it either belongs to the optimal maxcut or not depending on whether \(u, v\) belong to different partitions of the optimal maxcut \(M_o\). The probability that we have chosen the right partitions is at least half. Let \(X_e\) be a random 0-1 variable (also called indicator random variables) that is 1 iff the algorithm chooses it consistently with the maxcut. The expected size of the cut produced by the algorithm is

\[
E\left[\sum_e w(e) \cdot X_e\right] = \sum_e w(e) \cdot E[X_e] \geq M_o/2
\]

Therefore we have a simple randomized algorithm that attains a \(\frac{1}{2}\) approximation.

**Exercise 10.6** For an unweighted graph show that a simple greedy strategy leads to a \(\frac{1}{2}\) approximation algorithm.
Chapter 11

Dimensionality Reduction

There are many applications where we deal with points lying in a very high dimensional Euclidean space. Storing $n$ points in a $d$-dimensional space takes $O(nd)$ space, and even a linear time algorithm for processing such an input can be impractical. Many algorithms depend only on the pair-wise distance between these points. For example, the nearest-neighbour problem seeks to find the closest input point (in terms of Euclidean distance) to a query point. There is a trivial linear time algorithm to solve this problem, which just looks at every input point and computes its distance to the query point. Since the solution to this problem only depends on the distance of the query point to these $n$ points, we ask the following question: can the points be mapped to a low dimensional space which preserves all pair-wise distances? It is clear that $d$ can be made at most $n$ (just restrict to the affine space spanned by the $n$ points), and in general one cannot do better.

Exercise 11.1 Consider the 3 vertices of an equilateral triangle in the plane, each at distance 1 from the other two vertices. Show that it is not possible to map these 3 points to a line such that all pair-wise distances are 1.

The above example shows that even in trivial settings, it is not possible to reduce the dimensionality of a set of points without distorting pair-wise distances. What if we are willing to incur a small amount of distortion in the pair-wise distances? This is often an acceptable option because in many practical applications, the actual embedding of points in $d$ dimensions is based on some rough estimates. Since the data already has some inherent noise, it should be acceptable to distort the pair-wise distances slightly.

Let us make these ideas more formal. We are given a set $V$ of $n$ points in a $d$-dimensional Euclidean space. Let $f$ be a mapping of these points to a $k$-dimensional Euclidean space. We say that this mapping (or embedding) has distortion $\alpha > 1$ if