Lecture Notes for Algorithm Analysis and Design

Sandeep Sen[1]

November 6, 2012

[1]Department of Computer Science and Engineering, IIT Delhi, New Delhi 110016, India.
E-mail:ssen@cse.iitd.ernet.in

Contents

Preface

This write-up is a rough chronological sequence of topics that I have covered in the past in postgraduate and undergraduate courses on *Design and Analysis of Algorithms* in IIT Delhi. A quick browse will reveal that these topics are covered by many standard textbooks in Algorithms like AHU, HS, CLRS, and more recent ones like Kleinberg-Tardos and Dasgupta-Papadimitrou-Vazirani.

What motivated me to write these notes are
(i) As a teacher, I feel that the sequence in which the topics are exposed has a significant impact on the appreciation and understanding of the subject.
(ii) Most textbooks have far too much material for one semester and often intimidate an average student. Even when I pick and choose topics, the ones not covered have a distracting effect on the reader.
(iii) Not prescribing a textbook in a course (that I have done in the past) creates insecurity among many students who are not adept at writing down notes as well as participating in class discussions so important for a course like algorithms. (As a corollary, this may make it easier for some of the students to skip some lectures.)
(iv) Gives me some flexibility about asking students to fill up some of the gaps left deliberately or inadvertently in my lectures.
(v) Gives my students some idea of the level of formalism I expect in the assignments and exams - this is somewhat instructor dependent.
(vi) The students are not uniformly mature so that I can demand formal scribing.

I am sure that many of my colleagues have felt about one or more of the above reasons before they took the initiative at some point as is evidenced by the availability of many excellent notes that are accessible via internet. This is a first draft that I am making available in the beginning of the semester and I am hoping to refine and fill up some of the incomplete parts by the middle of this semester. The notes are likely to contain errors, in particular, typographic. I am also relying on the power of collective scrutiny of some of the brightest students to point these out and I will endeavour to update this every week.

I have also promised myself that I will not get carried away in adding to the present version beyond 1xx pages.

Sandeep Sen
July 2007

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 - derive a closed form.*

It is not difficult to argue that 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 $F\lceil n/2 \rceil$ 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 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

$$T_B(n) \leq T_B(\lfloor n/2 \rfloor) + M(n|x|)$$

[1] Adding two k bit numbers take $\Theta(k)$

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 2×2 matrix - each squaring operation involves 8 multiplication 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 2×2 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 us 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) drawback 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 techological reason, the size of the faster memory is limited. There could be a disparity of 10^5 between the fastest andthe 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. It turns out that in this model, the cost of sorting n elements is $O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$ disk accesses and this is also optimal.

[2] We can also work with $c\log n$ bits as the asymptotic analysis does not change for a constant c.

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 intellectual 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 capabilities. 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 intuititive reasons for this that with more processors (as with more individuals), the communication 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 Access 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 synchorized globally and there is no cost of synchronization. In this model, there is no extra overhead for communication 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.

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 protocals 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

$$gcd(a, b) = gcd(a, a + b)$$

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

Exercise 2.1 *Prove this rigorously.*

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

Algorithm Euclid_GCD

Input: Positive integers a, b such that $b \leq a$
Output GCD of a, b

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 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, $\{xa + yb|\ x, y$are integers it is known that

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

Proof: Let $\ell = \min\{xa + yb | xa + yb > 0\}$. Clearly $gcd(a, b)$ divides ℓ and hence $gcd(a, b) \leq \ell$. We now prove that ℓ 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 ℓ. □

The above result can be restated as ℓ 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 [1] that for every number

[1] since it forms a group

$x \in F_q^*$, there exists $y \in F_q^*$ such that $x \cdot y \equiv 1 \mod q$ which is also called the inverse of x. To compute the inverse of a we can use 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* $Z_N^* = \{x | x$ *is relatively prime to* $N\}$. *First show that* Z_N^* *is closed under multiplication modulo* N, *i.e.,* $a, b \in Z_N^* \ a \cdot b \mod N \in Z_N^*$.

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 in 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 $k' = k - R(x, S)$.

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 $O(\frac{1}{1-\epsilon} \cdot n)$. So it could vary between $\Omega(n)$ and $O(n^2)$ - where a better running time is achieved by ensuring a smaller value of ϵ.

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 ϵ. Typically, ϵ 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* [2] 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] 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 $4n$.

2.2.2 Median of medians

Partition the elements into groups of 5 and choose the median of each group. Denote the groups by $\mathcal{G}_i$ 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 2.5 *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.*

[3]Strictly speaking we should be using the floor function but we are avoiding the extra symbols and it does not affect the analysis.

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 Σ.

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.*

A sorting algorithm is considered *stable* if the relative order of input elements having identical values is preserved in the sorted output.

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 $w_i = a_{i,1}a_{i,2} \ldots a_{i,l_i}$, 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 Σ. We can sort them using radix sort in two rounds in time proportional to $O(N + L + |\Sigma|)$ 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 2.6 *Work out the details of sorting n binary strings in $O(N)$ steps where $N = \sum_i \ell_i$, ℓ_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 *min , delete-min , insert , 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.

[4] we are assuming min heaps

Exercise 2.7 *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 *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.

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 2.8 *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, $parent(x)$ 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 2.9 *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 2.10 *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 $\Phi()$ 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 difference in potential. Note that the total work done by an algorithm over t steps is $W = \sum_{i=1}^{i=t} w_i$. 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 funcion 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 2.11 *Can you define an appropriate potential function for the search data-structure analysis ?*

[5]this may be hard to analyze

Chapter 3

Optimization I : Brute force and Greedy strategy

A generic definition of an optimization problem involves 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. A very important common optimization problem is Linear Programming where there is a finite set of linear constraints and the objective function is also linear - the underlying space is Euclidean. A convex function f satisfies $f(\lambda \cdot x + (1 - \lambda) \cdot y) \leq \lambda f(x) + (1 - \lambda) f(y)$ where $0 < \lambda < 1$. A *convex* programming problem is one where the objective function is convex and so is the feasible set. Convex programming problem over Euclidean real space have a nice property that local optima equals the global optimum. Linear programming falls under this category and there are provably efficient algorithms for linear programming.

Many important real life problems can be formulated as optimization problems and therefore solving them efficiently is one of the most important area of algorithm design.

3.1 Heuristic search approaches

In this section, we will use the *knapsack* problem as the running example to expose some of the algorithmic ideas. The 0-1 Knapsack problem is defined as follows.

Given a knapsack of capacity C 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 n objects that fits into the knapsack and that maximizes the total profit.

In more formal terms, let x_i be 1 if object i is present in the subset and 0 otherwise.

The knapsack problem can be stated as

$$\text{Maximize} \sum_{i=0}^{n} x_i \cdot p_i \;\; \text{subject to} \;\; \sum_{i=0}^{n} x_i \cdot w_i \leq C$$

Note that the constraint $x_i \in \{0, 1\}$ is not linear. A simplistic approach will be to enumerate all subsets and select the one that satisfies the constraints and maximizes the profits. Any solution that satisfies the 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.
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 x_1. At the second level, we again associate the left and right branches with the choice of x_2. 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 $000 \ldots 1$ corresponds to the subset that only contains x_n.

Any intermediate node at level j from the root corresponds to partial choice among the objects $x_1, x_2 \ldots x_j$. As we traverse the tree, we keep track of the *best* feasible solution among the nodes visited - let us denote this by T. At a node v, let $S(v)$ denote the subtree rooted at v. If we can estimate the maximum profit $P(v)$ among the leaf nodes of $S(v)$, we may be able to prune the search. Suppose $L(v)$ and $U(v)$ are the lower and upperbounds of $P(v)$, i.e. $L(v) \leq P(v) \leq U(v)$. If $U(v) < T$, then there is no need to explore $S(v)$ as we cannot improve the current best solution, viz., T. In fact, it is enough to work with only the upper-bound of the estimates and $L(v)$ is essentially the current partial solution. As we traverse the tree, we also update $U(v)$ and if it is less than T, we do not search the subtree any further. 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.

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 estimation.*

Example 3.1

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

Profits	10	10	12	18
Volume	2	4	6	9

We will use the ratio of profit per volume as an estimation for upperbound. For the above objects the ratios are 5, 2.5, 2 and 2. Initially, $T = 0$ and $U = 5 \times 15 = 75$. After including x_1, the residual capacity is 13 and $T = 10$. By proceeding this way, we

obtain $T = 38$ for $\{x_1, x_2, x_4\}$. By exploring further, we come to a stage when we have included x_1 and decided against including x_2 so that $L(v) = 10$, and residual capacity is 13. Should we explore the subtree regarding $\{x_3, x_4\}$? Since profit per volume of x_3 is 2, we can obtain $U(v) = 2 \times 13 + 10 = 36 < L = 38$. 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.

Exercise 3.2 *Show that if we use a greedy strategy based on profit/volume, i.e., choose the elements in decreasing order of this ratio, then the profit as at least half of the optimal solution. For this claim, you need to make one change, namely, if x_k is the last object chosen, such that $x_1, x_2 \ldots x_k$ in decreasing order of their ratios that can fit in the knapsack, then eventually choose $\max\{\sum_{i=1}^{i=k} p_i, p_{k+1}\}$. Note that x_{k+1} is such that $\sum_{i=1}^{i=k} w_i \leq C < \sum_{i=1}^{i=k+1} w_i$.*

3.1.1 Game Trees *

A *minmax* tree is used to represent a game between two players who alternately make moves trying to win the game. We will focus on a special class of minmax trees called *AND-OR* trees where alternate levels of trees are labelled as OR nodes starting with the root node and the remaining are labelled AND nodes. Let 1 represent a win for player 1 and 0 represent a loss for player 1 who makes the first move at the root - the numbers are flipped for player 2. The leaf nodes correspond to the final state of the game and are labelled 1 or 0 corresponding to win or loss for player 1. We want to compute the values of the intermediate nodes using the following criteria. An OR node has value 1 if one of the children is 1, and 0 otherwise - so it is like the boolean function *OR* . An AND node behaves like a boolean AND function - it is 0 if one of the children is 0. The interpretation is as follows - the player 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 other player - only when both branches for the other player leads to a win (for the root), the root will win, otherwise the other player 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.

First consider a single level AND tree that evaluates to 0. If we use a fixed order to inspect the leaf nodes, in the worst case, both leaf nodes may have to be searched if there is one 0 and it is the second branch. On the other hand, if we randomly choose the order, for an answer 0, with probability 1/2, we will end up searching only one node (the one labelled 0) and we do not have to evaluate the other one. Therefore

the expected number of look-ups is 3/2 for an AND node with answer 0 and the same holds for an OR node with answer 1. For the other cases, there is no saving. 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 which is the good case for OR.

In essence, we are applying the branch-and-bound method to this problem, and we obtain a provable improvement in the following way. The two children are evaluated in a *random* order.

Consider 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.

Exercise 3.3 *Show that for $k = 1$, 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).*

Let us consider a root with label OR and its two AND children, say y and z, whose children are OR nodes with $2(k-1)$ depth. We have the two cases

output is 0 at the root Both y and z must evaluate to 0. Since these are AND nodes, again with probability 1/2, we will end up evaluating only one of the children (of y, z) that requires expected $\frac{1}{2} \cdot (1+2) \cdot 3^{k-1} = \frac{3}{2} \cdot 3^{k-1}$ steps for y as well as z from Induction hypothesis. This adds upto a total of expected $2 \cdot \frac{3}{2} \cdot 3^{k-1} = 3^k$ steps for y and z.

Output is 1 at the root At least one of the AND nodes y, z must be 1. With probability 1/2 this will be chosen first and this can be evaluated using the expected cost of evaluating two OR nodes with output 1. By induction hypothesis this is $2 \cdot 3^{k-1}$.

The other possibility (also with probability 1/2) is that the first AND node (say y) is 0 and the second AND node is 1. The expected cost of the first AND node with 0 output is $1/2 \cdot 3^{k-1} + 1/2 \cdot (3^{k-1} + 3^{k-1})$ - the first term corresponds to the scenario that the first child evaluates to 0 and the second term corresponds to evaluating both children of y are evaluated. The expected cost of evaluating y having value 0 is $3/2 \cdot 3^{k-1}$.
The expected number of evaluation for second AND node z with output 1 is $2 \cdot 3^{k-1}$ since both children must be evaluated.

So the total expected cost is $1/2 \cdot 3^{k-1}(2 + 3/2 + 2) = 2.75 \cdot 3^{k-1} < 3^k$.

In summary, for an OR root node, regardless of the output, the expected number of evaluations is bounded by 3^k.

Exercise 3.4 *Establish a similar result for the AND root node.*

If N the number of leaf nodes, then the expected number of evaluations is $N^{\log_4 3} = N^{\alpha}$ where $\alpha < 0.8$.

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 [1] 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 $\Phi \in M$. 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 *For the maximal spanning tree problem on a graph $G = (V, E)$, (E, F) is a matroid where F is the set of all subgraphs without cycles (i.e. all the* forests*).*

For any weight function $w : S \to \mathbb{R}^+$, the optimization problem is defined as finding a subset from M for which the cumulative weight of the elements is maximum among all choices of subsets from M. A simple way to construct a subset is the following greedy approach.

Algorithm Gen_Greedy

Let $e_1, e_2 \ldots e_n$ be the elements of S in decreasing order of weights. Initialize $T = \Phi$.
For $i = 1$ to n do

In the i-th stage
If $T \cup \{e_i\} \in M$, then $T \leftarrow T \cup \{e_i\}$

Output T as the solution

The running time of the algorithm is dependent mainly on the test for *independence* which depends on the specific problem. M is not given explicitly as it may be very

[1] M is a family of subsets of S

large (even exponential[2]). Instead, a characterization of M is used to perform the test.

What seems more important is the question - Is T the maximum weight subset ? This is answered by the next result

Theorem 3.1 *The following are equivalent*

1. *Algorithm Gen_Greedy outputs the optimal subset for any choice of the weight function. Note that in this case the subset system is called a* matroid.

2. **exchange property**
   For any $s_1, s_2 \in M$ where $|s_1| < |s_2|$, then there exists $e \in s_2 - s_1$ such that $s_1 \cup \{e\} \in M$.

3. *For any $A \subset S$, all maximal subsets of A have the same cardinality. A maximal subset T of A implies that there is no element $e \in A - T$ such that $T \cup \{e\} \in M$.*

The obvious use of the theorem is 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), then greedy cannot always return the optimum subset.

Proof: We will prove it in 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 will prove it by contradiction. Suppose Property 2 doesn't 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. Further, wlog, let $|s_2| = p+1$ and $|s_1| = p$. Let us define a 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 approach will pick up all elements from s_1 and then it won't be able to choose any element from $s_2 - s_1$. The greedy solution has weight $(p+2)|s_1| = (p+2) \cdot p$. By choosing all elements of s_2, the solution has cost $(p+1) \cdot (p+1)$ which has a higher cost than greedy and hence it is a contradiction of Property 1 that is assumed to be true.

Property 2 implies Property 3 If two maximal subsets of a set A have different cardinality, it is a violation of Property 2. Since both of these sets are independent, we should be able augment the set s_1 with an element from s_2.

[2]The number of spanning trees of a complete graph is n^{n-2}

Property 3 implies Property 1 Again we will prove by contradiction. Let $e_1e_2 \dots e_i \dots e_n$ be the edges chosen by the greedy algorithm in decreasing order of their weights. Further, let $e'_1e'_2 \dots e'_i \dots e'_m$ be the edges of an optimal solution in decreasing order - (Is $m = n$?). Since the weight of the greedy solution is not optimal, there must a $j \leq n$ such that $e_j < e'_j$. Let $A = \{e \in S | w(e) \geq w(e'_j)\}$. The subset $\{e_1, e_2 \dots e_{j-1}\}$ is maximal with respect to A (Why ?). All the elements in $\{e'_1, e'_2 \dots e'_j\}$ form an independent subset of A that has greater cardinality. This contradicts Property 3. □

Example 3.3 Half Matching Problem *Given a directed graph with non-negative edge weights, find out the maximum weighted subset of edges such that the in-degree of any node is at most 1.*

The problem defines a subset system where S is the set of edges and M is the family of all subsets of edges such that no two incoming edges share a vertex. Let us verify Property 2 by considering two subsets S_p and S_{p+1} with p and $p+1$ edges respectively. S_{p+1} must have at least $p+1$ distinct vertices incident on the $p+1$ incoming edges and there must be at least one vertex is not part of S_p's vertex set incident to S_p's incoming edges. Clearly, we can add this edge to S_p without affecting independence.

Example 3.4 Weighted Bipartite Matching

Consider a simple graph with a zig-zag. There are two maximal independent sets (set of edges that do not share an end-point), one with cardinality 2 and the other having only 1 edge. There Property 3 is violated.

3.2.1 Maximal Spanning Tree

Let us try to verify the exchange property. Let F_1 and F_2 be two forests such that F_2 has one edge less than F_1. We want to show that for some $e \in F_1 - F_2$, $F_2 \cup \{e\}$ is a forest. There are two cases

Case 1 F_1 has a vertex that is not present in F_2, i.e. one of the end-points of an edge, say $e' \in F_1$ is not present in F_2. Then e' cannot induce a cycle in F_2.

Case 2 The set of vertices in F_1 is the same. The set of end-points of F_1 may be a proper subset of F_2. Even then we can restrict our arguments to the end-points of F_1 as F_2. Since there are more edges in F_1, the number of connected components in F_2 is more than F_1. Recall that if v_F, e_F, c_F represent the the number of vertices, edges and connected components in a forest F then

$v_F - c_F = e_F$ (Starting from a tree removal of an edge increases components by 1)

We shall show that there is an edge (u', v') in F_1 such that u' and v' are in different connected components of F_2 and therefore $(u', v') \cup F_2$ cannot contain a cycle.

If you imagine coloring the vertices of F_1 according to the components in F_2, at least one component (tree) will have vertices with more than one color from pigeon hole principle. Therefore there will be at least one edge that will have its end-points colored differently. Trace the path starting from one color vertex to a vertex with a different color - we will cross an edge with different colors on its end-point. This edge can be added to F_2 that connects two components.

Exercise 3.5 *The matroid theory is about maximizing the total weight of a subset. How would you extend it to finding minimum weighted subset - for example Minimal Spanning Trees ?*

3.2.2 A Scheduling Problem

We are given a set of jobs $J_1, J_2 \ldots J_n$, their corresponding deadlines d_i for completion and the corresponding penalties p_i if a job completes after deadlines. The jobs have unit processing time on a single available machine. We want to minimize the total penalty incurred by the jobs that are not completed before their deadlines.
Stated otherwise, we want to maximize the penalty of the jobs that get completed before their deadlines.

A set A of jobs is *independent* if there exists a schedule to complete all jobs in A without incurring any penalty. We will try to verify Property 2. Let A, B be two independent sets of jobs with $|B| > |A|$. We would like to show that for some job $J \in B$, $\{J\} \cup A$ is independent. Let $|A| = m < n = |B|$. Start with any feasible schedules for A and B and compress them, i.e. remove any idle time between the jobs by transforming the schedule where there is no gap between the finish time of a job and start time of the next. This *shifting to left* does not affect independence. Let us denote the (ordered) jobs in A by $A_1, A_2 \ldots A_m$ and the times for scheduling of jobs in A be $d_1, d_2 \ldots d_m$ respectively. Likewise, let the jobs in B be $B_1, B_2 \ldots B_n$ and their scheduling times $d'_1, d'_2 \ldots d'_n$.

If $B_n \notin A$, then we can add B_n to A and schedule it as the last job. If $B_n = A_j$, then move A_j to the same position as B_n (this doesn't violate its deadline) creating a gap at the j-th position in A. We can now shift-to-left the jobs in $A - A_j$ and now by ignoring the jobs $B_n = A_j$, we have one less job in A and B. We can renumber the jobs and are in a similar position as before. By applying this strategy inductively, either we succeed in adding a job from B to A without conflict or we are in a situation where A is empty and B is not so that we can now add without conflict.

3.3 Efficient data structures for MST algorithms

The greedy algorithm described in the previous section is known as Kruskal's algorithm that was discovered much before the matroid theory was developed. In the usual implementation of Kruskal's algorithm, the edges are sorted in increasing order of their weights.

Algorithm Kruskal_MST

input Graph $G = V, E)$ and a weight function on edges.
output A minimum Spanning Tree of G
Let $e_1, e_2 \ldots e_m$ be the elements of E in increasing order of weights. Initialize $T = \Phi$.
For $i = 1$ to m do

In the i-th stage
If $T \cup \{e_i\}$ doesn't contain a cycle, then $T \leftarrow T \cup \{e_i\}$

Output T.

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 and growing the trees. The algorithm adds edges that connect distinct trees (connected components) - an edge whose endpoints are within the same tree may not be added since it induces cycles. Therefore we can maintain a data structure that supports the following operations

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

Union Combine two connected components.

For obvious reasons, such a data structure is called a union-find structure and can be seen in a more general context where we have a family of subsets and for any given element we can find the subset it belongs to and we can merge two subsets into one. The subsets are assumed to be *disjont*.

3.3.1 A simple data structure for union-find

Let us try to use arrays to represent the sets, viz., in an array A, let $A(i)$ contain the label of vertex i. The labels are also in the range $1, 2 \ldots n$. Initially all the labels

are distinct. For each set (label), we also have pointers to all its elements, i.e., the indices of the array that belong to the set.

Find Is really simple - for vertex i report $A(i)$. This takes $O(1)$ time.

Union To do $union(S_j, S_k)$, we change the labels of all elements pointed to by j and link them with k. Thus after union, we have labelled all the elements in the union as k. The time for this operation is proportional to the number of elements in set j. For obvious reasons, we would change labels of the smaller subset.

Although the time for a single union operation can be quite large, in the context of MST, we will analyze a sequence of union operations - there are at most $n-1$ union operations in Kruskal's algorithm. Consider a fixed element $x \in S$. 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 a set is $\leq n$, this implies that the maximum number of label changes is $\log n$). Kruskal's algorithm involves $|E|$ finds and at most $|V|-1$ unions; from the previous discussion this can be done in $O(m + n \log n)$ steps using the array data-structure.

3.3.2 A faster scheme

The previous data structure gives optimal performance for $m \in \Omega(n \log n)$ so theoretically we want to design better schemes for graphs with fewer edges. For this we will explore faster schemes for union-find.

Instead of arrays, let us use trees[3] to represent subsets. Initially all trees are singleton nodes. The root of each tree is associated with a label (of the subset) and a *rank* which denotes the maximum depth of any leaf node. To perform 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 Find operation is the maximum depth of a node.
To perform Union (T_1, T_2), 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 *union by rank* heuristic. The rank of a tree increases by one only when T_1 and T_2 have the same ranks. Once a root node becomes a child of another node, the rank doesn't change (by convention). The union operation takes $O(1)$ steps.

[3]this tree should not be confused with the MST that we are trying to construct

Exercise 3.6 *Prove that*
(i) The number of nodes in tree of rank r is at least 2^r if we use the union by rank heuristic.
(ii) The maximum depth of a tree (or equivalently the value of any rank) is at most $\log n$.
(iii) There are at most $\frac{n}{2^r}$ nodes of rank r.
(iv) The ranks of nodes along any path from a node to the root are increasing monotonically.

So we are in a situation where Find takes $O(\log n)$ and Union operation takes $O(1)$. Seemingly, we haven't quite gained anything so let us use the following heuristic.
Path compression

When we do a Find(x) operation, let $x_0 =$ root of $x, x_1, x_2 \ldots x$ be the sequence of nodes visited. Then we make the subtrees rooted at x_i (minus the subtree rooted at x_{i+1}) the children of the root node. Clearly, the motivation is to bring more nodes closer to the root node, so that the time for the Find operation decreases. And Path compression does not increase the asymptotic cost of the current Find operation (it is factor of two).

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.

3.3.3 The slowest growing function ?

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

$$2^{2^{2^{\cdot^{\cdot^{2}}}}\}i}$$

which can defined more formally as a function

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

Let

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

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

$$\log^* n = \min\{i \geq 0 | \log^{(i)} n \leq 1\}$$

In other words,

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

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

$$\begin{array}{lll} A(1,j) = & 2^j & \text{for } j \geq 1 \\ A(i,1) = & A(i-1,2) & \text{for } i \geq 2 \\ A(i,j) = & A(i-1, A(i,j-1)) & \text{for } i,j \geq 2 \end{array}$$

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

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

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 is less than $\log n$, we already have an upperbound of $(\log n)$ for any individual Find operation. We will adopt the following strategy for counting the cost of Find operations. We will associate a hypothetical counter with each node that we will increment whenever it is visited by some Find operation. Finally, by summing the counts of all the nodes, we can bound the cost of all the find operations. We will refer to the incremental cost of Find operation as a *charge*.

We further distinguish between two kinds of *charges*

Block charge If the block number of the parent node $p(v)$ is strictly greater than a node v, i.e., $B^{-1}(rank(p(v))) > B^{-1}(rank(v))$, then we assign 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.

Observation 3.1 *Once the rank of a node and its parent are in different blocks, they continue to be in different blocks, i.e. once a node incurs block charges, it will never incur any more path charges.*

The parent of a node may change because of path compression preceded by one or more union operations, but the new parent will have a rank higher than the previous parent. Consequently, a node in block j can incur path charges at most $B(j) - B(j-1) \leq B(j)$ times. Since the number of elements with rank r is at most $\frac{n}{2^r}$, the number of elements having ranks in block i is

$$\frac{n}{2^{B(i-1)+1}} + \frac{n}{2^{B(i-1)+2}} + \cdots \frac{n}{2^{B(i)}} = n\left(\frac{1}{2^{B(i-1)+1}} + \frac{1}{2^{B(i-1)+2}} + \ldots\right) \leq 2n\frac{1}{2^{B(i-1)+1}} = \frac{n}{2^{B(i-1)}}$$

Therefore the total number of path charges for elements in block i is at most $\frac{n}{2^{B(i-1)}} \cdot B(i)$ which is $O(n)$. For all the $\log^* n$ blocks the cumulative path charges is $O(n \log^* n)$ to which we have to add $O(m \log^* n)$ block charges.
Remark For some technical reason, in a Find operation, the child of the root node always incurs a block charge (Why ?)

3.3.5 Path compression only

If we only use path-compression, without the union-by-rank heuristic, we cannot bound the rank of a node by $\log n$. Since the union-by-rank heuristic does not change the asymptotic bound of the union operation, it is essentially to gain a better understanding of role of the path compression.

Note that the ranks of the nodes from any node to the root still increases monotonically - without the union-by-rank heuristic, the rank of node can increase by more than one (in fact arbitrarily) after a union operation. Let us denote parent of a node x as $p(x)$, parent of parent of x as $p^2(x)$ and likewise. Let us define the *level* of a node x by $\ell(x)$ as an integer i such that $2^{i-1} \leq rank(p(x)) - rank(x) \leq 2^i$. Therefore $\ell(x) \leq \log n$.

We account for the cost of a find(x) operation by charging a cost one to all the nodes in the path from x to 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, note that the $\ell(y)$ increases at least by one from the monotonicity of ranks and the fact that it is not the last node of its level, i.e., $rank(p(v_1)) - rank(v_1) + rank(p(v_2)) - rank(v_2) \geq 2(rank(p(v_1)) - rank(v_1))$ where v_1 and v_2 have same levels. 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.

Exercise 3.7 *Using a similar analysis, can you prove a better bound than* $\log n$ *for the union heuristic ?*

3.4 Compromising with Greedy

Although greedy doesn't always pay in terms of achieving an optimal solution, it is 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.

Example 3.5 Matching *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*.*

Exercise 3.8 *Show that the subset system corresponding to matching is not a matroid.*

Nevertheless let us persist with greedy, and analyse what we can achieve. Note that the edges are chosen in decreasing order of weights such that the end-points have not been chosen previously. Let us define the subset of edges chosen by greedy as G and let us denote the optimal solution by O. Clarly $w(O) \geq w(G)$. Consider an edge $(x, y) \in O - G$. Clearly, by the time the turn of (x, y) came, some edge(s) were already chosen by greedy that was incident on either x or y. Let e' be such an edge and clearly $w(e') \geq w(x, y)$ since it was chosen earlier. If there were two separate edges that were incident on x and y chosen earlier, let e' have the larger weight. We can therefore claim the following
For every edge $(x, y) \in O - G$, there is an edge, $e(x, y) \in G - O$ whose weight is greater than $w(x, y)$. Suppose $e(x, y) = (x, u)$ where $u \neq y$. Note that there may be another edge $(v, u) \in O - G$ which could not be chosen by greedy because of $e(x, y)$. In summary, an edge $e \in G - O$ can *block* atmost two edges in O to be chosen and its weight is greater than or equal to both the blocked edges. This implies that the total weight of the edges in G is at least *half* that of the optimal weight.

So greedy in this case has some guarantee about the solution even though it is not optimal.

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 $i < w_1$, $F_1(i) = 0$ 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 $\mathcal{T}$ 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

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

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 Σ, we want to determine if X can be derived from the grammar G.

A grammar in CNF has the following production rules

$$A \to BC \quad A \to a$$

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 $S \stackrel{*}{\Rightarrow} \alpha$ to denote that S can derive the sentence α in finite number of steps by applying production rules of the grammar.

The basis of our algorithm is the following observation

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

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 $x_1 x_2 \dots x_n$, we consider all substrings $X_{i,k} = x_i x_{i+1} \dots x_k$ 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 $1, 2, \dots n-t+1$ 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 $x_1, x_2 \dots x_n$ a subsequence $x_{i_1}, x_{i_2} \dots x_{i_k}$ 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 $x_1, x_2 \dots x_i$ ending at x_i (i.e. x_i must be the last element of the subsequence) as S_i. Clearly we are looking for $\max_{i=1}^{n} S_i$ [1]. 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.

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 $M_{i,j}$ $j \leq i$ 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 $1..j$.

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 $g^*_{i,\ell}$ 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

$$t(i,j) = \min_{s<j}\{L_2^2(h, g^*_{i-1,s}) + D_{s,j}\}$$

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 $1..s$. We can now write

$$g^*_{i,\ell}(s) = \begin{cases} g^*_{i-1,t(i,\ell)}(s) & s < t(i,\ell) \\ A_{t(i,\ell),\ell} & \text{otherwise} \end{cases}$$

where $A_{i,j}$ 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

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

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) aproximation, 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 Σ - note that more than one edge can share the same label. Given a string $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ over Σ, find the most probable path in the graph starting at v_o with label equal to σ. 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 σ. 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 lengths 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[2]. Given that the size of table is $n \cdot |V|$, the total

[2]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 bottommost 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 probablistic 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 $[l_k, r_k] \bigcap L_0$. 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 $[l_{i+1}, r_{i+1}] \bigcap L_i$ 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

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 α 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 $O(1/n^2)$. $\Box$.

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. $\Box$

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 $\exp^{\Omega(-\alpha^2 n)}$.*

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.

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

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

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 ommitted 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 triep 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 : $\{0, 1, ..., m-1\}$ usually, $n \geq m$

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

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

Hash by chaining: The more the collision the worse the performance. Look at a sequence $O_1(x_2), O_2(x_2), ..., O_n(x_n)$ 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 $i, i' \in \{0, 1, ..., m-1\}$
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*. □

Universal Hash Functions

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

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

for some small constant c. Roughly $\sum_h \delta_h(x,y) \leq c\frac{|H|}{m}$

Claim:

$$\frac{1}{|H|}\sum_{h \in H} 1 + \delta_h(x, S) \leq 1 + c\frac{n}{m}$$

where $|S| = n$.
Proof: Working from the LHS, we obtain

$$\begin{aligned}
&= \frac{1}{|H|}\sum_{h \in H} 1 + \frac{1}{|H|}\sum_h \sum_{y \in S} \delta_h(x,y) \\
&= 1 + \frac{1}{|H|}\sum_y \sum_h \delta_h(x,y) \\
&\leq 1 + \frac{1}{|H|}\sum_y c\frac{|H|}{m} \\
&= 1 + \frac{c}{m}n
\end{aligned}$$

So expected cost of n operation $= \sum(1 + \frac{ci}{m}) \leq (1 + c\beta)n$ □

5.3.1 Example of a Universal Hash function

$H' : h_{a,b}; h_{ab}(x) \to ((ax+b) \bmod N) \bmod m$ 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 $r, s \in [0...\frac{N-1}{m}]$

$$\begin{aligned} ax + b &= (q + rm) \bmod N \\ ay + b &= (q + sm) \bmod N \end{aligned}$$

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 $h(x) = a{\cdot}x \bmod N \bmod m$ 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 $f = \binom{n}{2} \cdot \frac{c}{m}$. By Markov inequality, the probability that the number of collisions exceeds $2f$ is less than $1/2$. 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 probablity 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 $4\sum_i n_i^2$. We can write

$$n_i^2 = 2 \sum_{x,y|h(x)=h(y)=i} 1 \quad + n_i.$$

$$\begin{aligned}\sum_i n_i^2 &= \sum_i n_i + 2\left(\sum_{x,y|h(x)=h(y)=i} 1\right) \\ &= 2\sum_i n_i + 2\sum_i \sum_{x,y|h(x)=h(y)=i} 1 \\ &= 2n + \sum_{x,y} \delta(x,y)\end{aligned}$$

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 $2\binom{n}{2}\cdot\frac{c}{m}$ 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 $\mathcal{U}$ and let $|\mathcal{U}| = N$. 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 $S \subset \mathcal{U}$ 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

$\sqrt{N}$. 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 $\log N/2$ (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 $\log N/2$ higher order bits and the search structure is built on the lower $\log N/2$ 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 $\log N/2$ 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

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

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 $N \leq 2^{2^{\log n / \log \log n}}$, 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. If we denote a d dimensional point by $(x_0, x_1 \dots x_d)$ then show the following.

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

Queries can be far more sophisticated than just point-queries but we will address the more restricted kind.

6.1 Interval Trees and Range Trees

One Dimensional Range Searching

Given a 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 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 $S = \{p_1, p_2, \dots, p_n\}$ 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_ℓ and x_u in the tree T. Let ℓ_1 and ℓ_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 ℓ_1 and ℓ_2. Another way to view the set of points is the union of the leaves of some subtrees of T.

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

Proof Sketch If you examine the search path of x_ℓ 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.
Complexity The tree takes $O(n)$ space and $O(n \log n)$ time in preprocessing. 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.

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

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 $[x_\ell : x_u]$ [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 $[x_\ell : x_u] \times [y_\ell : y_u]$ is a subset of $[x_\ell : x_u] \times [-\infty : +\infty]$. Since our objective is to obtain a time bound proportional

[1] You can think about the $[y_\ell : y_u]$ as $[-\infty : +\infty]$

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 $[y_\ell : y_u]$. 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 $[x_\ell : x_u]$) 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.

$Q(d) \leq 2\log n \cdot Q(d-1)$ $Q(1) = O(\log n + k)$ $Q(d)$ is the query time in d dimensions for n points

which yields $Q(d) = O(2^d \cdot \log^d n)$ [2].

Exercise 6.4 *What is the space bound for d dimensional range trees ?*

Exercise 6.5 *How would you modify the data structure for counting range queries ?*

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

[2]Strictly speaking $Q()$ is a variable of two parameters n and d

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 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.

Search(Q, v)

If $R(v) \subset Q$, then report all points in $R(v)$
Else

Let $R(u)$ and $R(w)$ be rectangles associated with the children u, w.
If $Q \cap R(u)$ is non-empty Search(Q, u)
If $R \cap R(w)$ is non-empty, Search $(Q, w\ )$

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 $O(n)$.
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 $Q(i) \in O(2^{\lfloor i/2 \rfloor})$ 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_ℓ 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_l. If T_i forms a *max-heap* on the y coordinates then we can output the points as follows -

> Let v_y denote the y coordinate associated with a node v.
> If $v_y < y_\ell$ or if v is a leaf node, then terminate search else
>
> > Output the point associated with v. Search u where u is the left child of v.
> > 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[3].

Until now, we assmed 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

[3]This kind of analysis where we are amortizing the cost on the output points is called *filtering* search.

Input A set S of points in plane.
Output A search tree.

- The point p with the largest y coordinate in S is the root r.
- If $S-p$ is non-empty, let L (respectively R) be the left (respectively right) half of the points in $S-p$. Let m be a separating x coordinate between L and R.
- Recursively construct the search trees on L and R and label the root r with $X(r) = m$.
  Comment : The left (right) subtree will be searched iff the query interval extends to the left (right) of $X(r)$.

The height of this tree is clearly $O(\log n)$.

Exercise 6.6 *Work out the details of performing a three-sided query and also analyse the running time.*

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 $\overline{a, b}$ 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[4] 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}).*

We know that the entire segment $\overline{(x_i, x_{i+1})}$ should be inside the hull, so if all the points of P (other than x_i, x_{i+1} lie to one side, then $CH(P) \subset$ half-plane supported by $\overline{x_i, x_{i+1}}$. 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.

[4]inside the triangle

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 $(x_0, y_0), (x_1, y_1)$, 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

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

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 ℓ, 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.

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

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_a (S_b) be the subset of S which lie above (below) of the line through p_l and p_r. As we had noted previously, $S_a \cup \{p_l, p_r\}$ and $S_b \cup \{p_l, p_r\}$ determine the upper and the lower convex hulls. We will describe the algorithm *Quickhull.* to determine the upper hull using $S_a \cup \{p_l, p_r\}$. 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 $slope(pq)$. The predicate *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 $\overline{p_l p_r}$.
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 $\{p_{2j-1}, p_{2j}\}$, $j = 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$.
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 $\{p_{2j-1}, p_{2j}\}$, $j = 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$ do the following (assuming $x[p_{2j-1}] < x[p_{2j}]$) 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 $S_a(r) = S_b(r) \cup \{p_{2j-1}, p_{2j}\}$. 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*$(S_a(r), p_m, p_r)$.

Exercise 6.7 *In step 3, show that if the pair $\{p_{2i-1}, p_{2i}\}$ satisfies the property that the line containing $\overline{p_{2i-1}p_{2i}}$ does not intersect the line segment $\overline{p_l p_r}$, 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.*

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

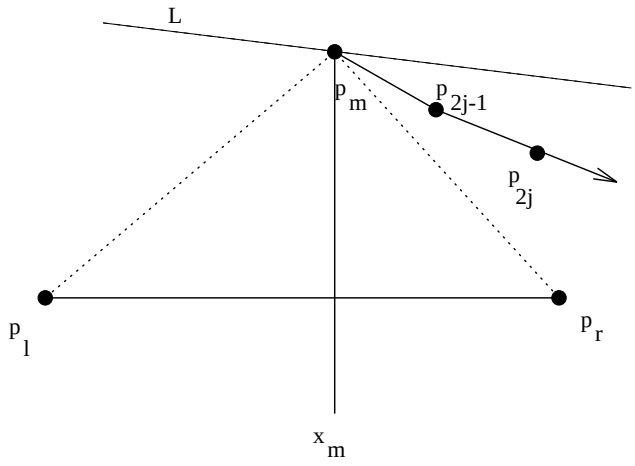

Figure 6.1: *Left-turn*(p_m, p_{2j-1}, p_{2j}) is true but *slope*$(\overline{p_{2j-1}p_{2j}})$ is less than the median slope given by L.

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, *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 *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 $n_l + n_r \leq n, \quad h_l + h_r \leq h - 1$.

Exercise 6.8 *Show that* $T(n, h)$ *is* $O(n \log h)$.

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

$$T(n,h) \leq \sum_{\forall n_l,n_r\geq 0} p(n_l,n_r)(T(n_l,h_l)+T(n_r,h_r))+bn \tag{6.5.1}$$

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. 6.5.1 we get

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

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

$$n_l\log h_l + n_r\log h_r \leq n\log(h-1) \tag{6.5.2}$$

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 $h_l\geq h_r$,

$$n_l\log h_l + n_r\log h_r \leq \frac{7}{8}n\log h_l + \frac{1}{8}n\log h_r \tag{6.5.3}$$

The right hand side of 6.5.3 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

$$\begin{aligned} T(n,h) &\leq p(cn \log(h-1) - tcn) + (1-p)cn \log(h-1) + bn \\ &= pcn \log(h-1) - ptcn + (1-p)cn \log(h-1) + bn \\ &\leq cn \log h - ptcn + bn \end{aligned}$$

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

In case p_m is an extreme point (say p_l), then we cannot apply Eq. 6.5.1 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. 6.5.1 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 $slope(p_{2i-1}p_{2i})$ 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. □

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 6.9 *Design an efficient solution to the ray shooting problem using by extending the interval trees.*

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 is 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.

Exercise 6.10 *Construct an example that depicts the worst case.*

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 list 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 as 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 $path - length \cdot \#versions + n$ which is $O(n \log n)$. This is much smaller than the $O(n^2)$ scheme tat stores each tree explicitly. With some additional ideas, it is possible to improve it to $O(n)$ space which will be optimal.

Chapter 7

Fast Fourier Transform and Applications

7.1 Polynomial evaluation and interpolation

A polynomial $\mathcal{P}(x)$ of degree $n-1$ in indeterminate x is a power series with maximum degree $n-1$ and has the general form $a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots a_1 x + a_0$, 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 $\mathcal{P}(x_i) = y_i$ 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.*

$$\mathcal{P}(x) = \sum_{k=0}^{n-1} y_k \cdot \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)}$$

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 $\mathcal{P}(x) = \mathcal{P}_E(x^2) + xP_o(x^2)$ where

$$\mathcal{P}_E = a_0 + a_2 x + \ldots a_{n-2} x^{n/2-1}$$

$$\mathcal{P}_0 = a_1 + a_3 x + \ldots a_{n-1} x^{n/2-1}$$

corresponding to the even and odd coefficients and $\mathcal{P}_E, \mathcal{P}_O$ are polynomials of degree $n/2 - 1$.

$$\mathcal{P}(x_{n/2}) = \mathcal{P}_E(x_{n/2}^2) + x_{n/2}\mathcal{P}_O(x_{n/2}^2) = \mathcal{P}_E(x_0^2) - x_0\mathcal{P}_O(x_0^2)$$

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

$$\mathcal{P}(x_{n/2+i}) = \mathcal{P}_E(x_{n/2+i}^2) + x_{n/2+i}\mathcal{P}_O(x_{n/2+i}^2) = \mathcal{P}_E(x_i^2) - x_i\mathcal{P}_O(x_i^2), \quad 0 \leq 0 \leq n/2 - 1$$

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 $x_{n/2} = \sqrt{-1} = \omega$. This involves complex numbers if we started with coefficients in $\mathbb{R}$[1]. If we continue with

[1]Depending on our choice of the field F, we can define ω such that $\omega^2 = -1$.

this strategy of choosing points, at the j-th level of recursion, we require

$$x_i^{2^{j-1}} = -x_{\frac{n}{2^j}+i}^{2^{j-1}} \quad 0 \leq i \leq \frac{n}{2^j} - 1$$

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

$$\mathcal{P}(x)_{a_0,a_1\ldots a_{n-1}}^{1,\omega,\omega^2\ldots\omega^{n-1}} = \mathcal{P}(x)_{a_0,a_2\ldots a_{n/2-2}}^{1,w\ldots\omega^{n/2-1}} + \mathcal{P}(x)_{a_1,a_3\ldots a_{n/2-1}}^{1,w\ldots\omega^{n/2-1}} + O(n) \text{ multiplications and additions}$$

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.

$$\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \ldots & \omega^{3(n-1)} \\ \vdots & & & & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Let us denote this by the matrix equation $A \cdot \overline{a} = \overline{y}$. In this setting, the interpolation problem can be viewed as computing the $\overline{a} = A^{-1} \cdot \overline{y}$. 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

$$\frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \frac{1}{\omega^2} & \frac{1}{\omega^4} & \ldots & \frac{1}{\omega^{2(n-1)}} \\ 1 & \frac{1}{\omega^3} & \frac{1}{\omega^6} & \ldots & \frac{1}{\omega^{3(n-1)}} \\ \vdots & & & & \\ 1 & \frac{1}{\omega^{n-1}} & \frac{1}{\omega^{2(n-1)}} & \ldots & \frac{1}{\omega^{(n-1)(n-1)}} \end{bmatrix}$$

which can be verified by multiplication with A. Recall that

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

(Use the identity $\sum_j \omega^{ji} = \frac{\omega^{in}-1}{\omega^i-1} = 0$ for $\omega^i \neq 1$.)
Moreover $\omega^{-1}, \omega^{-2}, \ldots w^{-(n-1)}$ 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

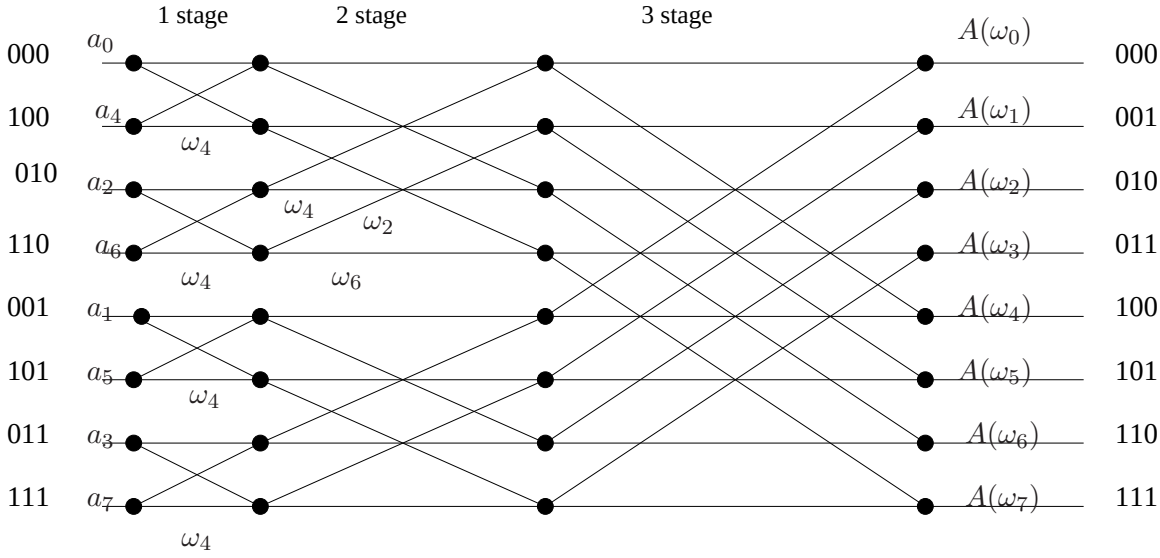

Figure 7.1: Computing an eight point FFT using a butterfly network

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.

$$\mathcal{P}_{0,1,..7}(\omega_0) = \mathcal{P}_{0,2,4,6}(\omega_0^2) + \omega_0 \mathcal{P}_{1,3,5,7}(\omega_0^2)$$

$$\mathcal{P}_{0,1,..7}(\omega_4) = \mathcal{P}_{0,2,4,6}(\omega_0^2) - \omega_0 \mathcal{P}_{1,3,5,7}(\omega_0^2)$$

Subsequently, $\mathcal{P}_{0,2,4,6}(\omega_0^2) = \mathcal{P}_{0,4}(\omega_0^4) + w_0^2 \mathcal{P}_{2,6}(\omega_0^4)$ and

$$\mathcal{P}_{0,2,4,6}(\omega_2^2) = \mathcal{P}_{0,4}(\omega_0^4) - w_0^2 \mathcal{P}_{2,6}(\omega_0^4)$$

To calculate $\mathcal{P}_{0,4}(\omega_0^4)$ and $\mathcal{P}_{0,4}(\omega_1^4)$ we compute $\mathcal{P}_{0,4}(\omega_0^4) = \mathcal{P}_0(\omega_0^8) + \omega_0^4 \mathcal{P}_4(\omega_0^8)$ and

$$\mathcal{P}_{0,4}(\omega_1^4) = \mathcal{P}_0(\omega_0^8) - \omega_0^4 \mathcal{P}_4(\omega_0^8)$$

Since $\mathcal{P}_i$ 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 $\bowtie$ 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 ina 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 $m = 2^{tn/2} + 1$ 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

$$\omega^n = \omega^{n/2} \cdot \omega^{n/2} = (2^t)^{n/2} \cdot (2^t)^{n/2} \equiv (m-1) \cdot (m-1) \bmod m \equiv (-1) \cdot (-1) \bmod m \equiv 1 \bmod m$$

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 ω 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 $b_{k-1}, b_{k-2}, \ldots b_0$.

$$a \times b = \left(a_{k-1} \cdot x^{k-1} + a_{k-2} \cdot x^{k-2} + \ldots a_0\right) \times \left(b_{k-1} \cdot x^{k-1} + b_{k-2} \cdot x^{k-2} + \ldots b_0\right)$$

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

$$a \times b = a_{k-1}b_{k-1}x^{2(k-1)} + (a_{k-2}b_1 + b_{k-2}a_1)x^{2k-3} + \ldots a_0b_0$$

Although in the final product, $x = 2^{n/k}$, we can compute the coefficients using *any* method and perform the necessary multiplcations 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) Paiwise 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

$$T(n) \leq 2\sqrt{\frac{n}{\log n}} \cdot T(2\sqrt{n \log n} + \log n) + O(n \log n) \tag{7.4.1}$$

Exercise 7.2 *With appropriate teminating condition, say the $O(n^{\log_2 3})$ 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 ℓ. 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 jth 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

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

. 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 $n_1, n_2, ..., n_k$, relatively prime to each other,*

$$x \equiv y \bmod n_i \text{ for all } i \Leftrightarrow x \equiv y \bmod n_1 n_2 n_3 ... n_k = M$$

Moreover,

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

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

Let k be such that $2^m < M = 2 \times 3 \times ... \times p_k$ 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 $\{2, 3, ..., p_{t^2 k}\}$ then the probability of false match at fixed position $Y(i) \leq \frac{1}{t^2}$. So for any $i \in \{1, ..., t\} \leq t\frac{1}{t^2} = \frac{1}{t}$. 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 $2.3....p_n = M > 2^n$. 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.

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

So $|p| \leq 2 \log t + \log n$, where $|p|$ is the bit length.

Updating Fingerprints: $Y(i+1) = 2(Y(i) - Y_i \cdot 2^{n-1}) + Y_{i+n}$. So $Y(i+1) \bmod p = [2(Y(i) \bmod p) - 2^n \bmod p . Y_i + Y_{i+n} \bmod p] \bmod p$. 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 =

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

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 Σ is the alphabet which is optimal if Σ 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 dome useful notations.

Let $X(i)$ denote the first i symbols of the pattern, i.e. a *prefix* of length i.
Let $\alpha \sqsubset \beta$ denote α is a *suffix* of β.

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 $X(j) \sqsubset X(i)$. 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

$$f(i) = \max_j \{X(j) \sqsubset X(i)\} \text{ otherwise 0, if no such X(j) exists}$$

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 upto 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 ≤ 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 $X(j-1) \sqsubset X(i-1)$ 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 $O(|X| + |Y|)$.

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 the none of the previous string matching algorithms are able to handle wild-cards.

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

8.3.1 Convolution based approach

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

$$c_{n-1+j} = a_0 \cdot b_j + a_1 \cdot b_{1+j} + a_2 \cdot b_{2+j} + \ldots a_{n-1} \cdot b_{n-1+j} \quad 0 \leq j \leq m+n-2$$

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_is 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 [4]. 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.

[4] We do the arithmetic modulo N

Chapter 9

Graph Algorithms

A number of graph problems use Depth First Search as the starting point. Since it runs in linear time, it is efficient as well.

9.1 Applications of DFS

Directed Graphs

Consider a Directed Acyclic Graph (DAG), $G = (V, E)$. We want to define a a function $f : V \Rightarrow \{1, 2, \ldots n\}$ such that for any directed edge (u, v), $f(v) > f(u)$.

Observation 9.1 *Every DAG has at least one vertex with indegree 0 (source) and a vertex with outdegree 0 (sink).*

This is also called a *topological* sorting of the vertices. Consider a DFS numbering $pre(v),\ v \in V$ and also a post-order numbering $post(v)$.

Observation 9.2 *If there is a path from $u \rightsquigarrow v$, then $post(u) > post(v)$.*

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 topological sorting.*

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 $u \rightsquigarrow v$ and $v \rightsquigarrow u$. 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 $\mathcal{G} = (V', E')$ 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.

Exercise 9.1 *Prove that $\mathcal{G}$ is a DAG.*

To determine the SCCs of G, notice that if we start a DFS from a vertex of a *sink* component c' of $\mathcal{G}$ then precisely all the vertices of c' can be reached. Since $\mathcal{G}$ is not explicitly available, we will use the following strategy to determine a sink component of $\mathcal{G}$. 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 $\mathcal{G}$ are interchanged.

Exercise 9.2 *If we do a DFS in G^R, then show that the vertex with the largest postorder numbering is in a sink component of $\mathcal{G}$.*

This enables us to output the SCC corresponding to the sink component of $\mathcal{G}$ using a simple DFS. Once this component is deleted (delete the vertices and the induced edges), we can apply the same strategy to the remaining graph.

Exercise 9.3 *Show that the SCCs can be determined using two DFS - one in G and the other in G^R.*

Hint Suppose $v \rightsquigarrow u$ in G where $f(v)$ had the largest value in G^R. Since $f(v) > f(u)$, in G^R, either $v \rightsquigarrow u$ or $u \not\rightsquigarrow v$. The latter does not hold from our first assumption. So $u \rightsquigarrow v$ in G implying that u, v belong to the same SCC of G.

9.1.2 Biconncted Components

Biconnected graphs (undirected) can be defined on the basis of vertex connectivity as well as equivalence classes on edges. Graphs that cannot be disconnected by removing one vertex[1] (along with the incident edges) are *biconnected*. This implies that between any pair of vertices, there are at least two vertex disjoint paths. 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.*

Exercise 9.4 *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).*

[1] If the removal of a vertex disconnects a graph then such a vertex is called an *articulation* point.

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 $\mathcal{G}$ 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 $a \in B$.

Exercise 9.5 *Show that $\mathcal{G}$ is a tree, i.e., it cannot contain cycles.*

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 observation.

Observation 9.3 *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 implies that 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

$$LOW(v) = \min_{w|(v,w)\in T} \{LOW(w), h(v)\}$$

Exercise 9.6 *How would you compute the $LOW(v)$ $v \in V$ along with the DFS numbering ?*

The computation of LOW numbers results in an efficient algorithm for testing biconnectivity but it does not yield the biconnected components. For this, let us consider the component graph $\mathcal{G}$. The biconnected component that corresponds to a leaf node of $\mathcal{G}$ 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 $\mathcal{G}$, we consider the next leaf-component. 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).

Exercise 9.7 *Formalize the above argument into an efficient algorithm that runs in $O(|V| + |E|)$ steps.*

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 -

$\mathbf{R}elax(u, v) : (u, v) \in E,$
if $\Delta(v) > \Delta(u) + w(u, v)$ 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 ∞ 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 9.4 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 by a simple argument by contradiction, that otherwise the original path is not the shortest path.

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 Δ

$$\Delta(v) = \min_{u \in In(v)} \{\Delta(u) + w(u, v)\}$$

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.

Bellman Ford SSSP

Initialize $\Delta(s) = 0$, $\Delta(v) = \infty$ $v \in V - \{s\}$.
Repeat $n - 1$ times

$relax(e)$ for all $e \in E$

Output $\delta(v) = \Delta(v)$ for all $v \in V$.

The correctness of the algorithm follows from the previous discussion and the following critical observation.

Observation 9.5 *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. [2]

Exercise 9.8 *Describe an efficient algorithm to detect negative cycle in graph.*

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.

[2] This is equivalent to the longest path problem which is known to be intractable

Exercise 9.9 *If there is no negative cycle, show that the predecessors form a tree (which is called the shortest-path tree).*

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 $\Delta(v) = \delta(v)$. One the other hand, for non-negative weights, we can make the following claim.

Observation 9.6 *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 $s \rightsquigarrow x \rightarrow y \rightsquigarrow v$, where $y \notin U$ and all the earlier vertices in the path $s \rightsquigarrow x$ 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 $\Delta(v)$.

A crucial property exploited by Dijkstra's algorithm is that along any shortest path $s \rightsquigarrow u$, the shortest path-lengths are non-decreasing because of non-negative edge weights. Along similar lines, we can also assert the following

Observation 9.7 *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 observation) and $\Delta(u) \leq \Delta(x), x \in V - U$. Suppose $\delta(x) < \delta(u)$, then by an extension of the previous argument , let y be the earliest vertex in $s \rightsquigarrow x$ that is not in U. Then $\Delta(y) = \delta(y) \leq \delta(x) < \delta(u) \leq \Delta(u)$, thereby violating the minimality of $\Delta(u)$.

To implement this algorithm efficiently, we maintain a priority queue on the values of $\Delta(v)$ for $v \in V - U$, so that we can choose the one with the smallest value in $O(\log n)$ steps. Each edge is relaxed exactly once since only the edges incident on vertices in U are relaxed - however because of relax operation the $\Delta()$ of some vertices may change. This yields a running time of $((|V| + |E|) \log |V|)$.

9.2.3 Floyd-Warshall APSP algorithm

Consider the adjacency matrix A_G of the given graph $G = (V, E)$ where the entry $A_G(u, v)$ contains the weight $w(u, v)$. Let us define the matrix product of $A_G \cdot A_G$ in a way where the multiplication and addition operators are replaced with addition and max operator.

Claim 9.2 *Define the k-th iterate of A_G, namely A_G^k for $n-1 \geq k \geq 2$ as $\min\{A_G^{k-1}, A_G^{k-1} \cdot A_G\}$ where the* min *is the entrywise minimum. Then $A_G^k(u, v)$ contains the shortest path distance between u and v consisting of at most k edges.*

We can prove it by induction on k. Note that a path between u and v of at most k edges consists of a path of at most $k-1$ edges to one neighbours of v followed by the last edge.

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**
$$\forall v \in V - \{s, t\}, \sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$$
where $in(v)$ are the edges directed into vertex v and $out(v)$ 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 $\sum_{e \in out(v)} f(e)$ and the *inflow* into v is given by $\sum_{e \in in(v)} f(e)$. The *net flow* is defined as outflow minus inflow $= \sum_{e \in out(v)} f(e) - \sum_{e \in in(v)} f(e)$. 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 9.8 *The net flow at s and the net flow at t are equal in magnitude.*

From the flow conservation constraint

$$\sum_{v \in V - \{s,t\}} \left(\sum_{e \in out(v)} f(e) - \sum_{e \in in(v)} f(e) \right) = 0$$

Let E' be edges that are **not** incident on s, t (either incoming or outgoing). Then

$$= \sum_{e \in E'} (f(e) - f(e)) + \left(\sum_{e \in out(s)} f(e) - \sum_{e \in in(s)} f(e) \right) + \left(\sum_{e \in out(t)} f(e) - \sum_{e \in in(t)} f(e) \right) = 0$$

For an an edge $e \in E'$, $f(e)$ is counted once as incoming and once as outgoing which cancel eachother. So

$$\sum_{e \in out(s)} f(e) - \sum_{e \in in(s)} f(e) = \sum_{e \in in(t)} f(e) - \sum_{e \in out(t)} f(e)$$

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 Δ be the minimum of the residual capacity of the forward edges and the flows of the backward edges. The edge(s) that determines Δ is called a *bottleneck* edge of the augmenting path.

By increasing the flow in the forward edges by Δ and decreasing the backward edges by Δ, we can preserve both the capacity constraints and the flow conservation constraints. In this process the is now increased by Δ. Such a path is called a *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 $\sum_{\substack{u \in S, \\ v \in T}} C(u, v)$. Note that only the capacities of the forward edges are counted.

Theorem 9.1 (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 9.9 *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

$$\sum_{\substack{e \in out(S^*) \\ e \in in(T^*)}} f(e) = \sum_{\substack{e \in out(S^*) \\ e \in in(T^*)}} C(e) = C(S^*, T^*)$$

This implies that

$$f* \geq f = C(S^*, T^*) \text{ mincut} \tag{9.3.1}$$

For any cut (S, T) and any flow f, consider

$$\sum_{v \in S} \left(\sum_{e \in out(v)} f(e) - \sum_{e \in in(v)} f(e) \right)$$

$= \sum_{e \in out(s)} f(e) - \sum_{e \in in(s)} f(e)$ 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 $\mathcal{E}$ and $\mathcal{E}'$ 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

$$\sum_{e \in \mathcal{E}} (f(e) - f(e)) + \sum_{e \in out(S)} f(e) - \sum_{e \in in(S)} f(e)$$

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 $C(S, T)$. Therefore $f \leq C(S, T)$ 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 9.3.1 , we find that $f^* = C(S^*, T^*) =$ 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 9.3 *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

$$s_i^{k+1} \geq s_i^k \text{ and } t_i^{k+1} \geq t_i.$$

We will prove it by contradiction. Suppose $s_i^{k+1} < s_i^k$ for some k and among all such vertices let $s \rightsquigarrow v$ have minimum path length (after $k+1$ iterations). Consider the last edge in the path, say (u, v). Then

$$s_v^{k+1} = s_u^{k+1} + 1 \tag{9.3.2}$$

since u is on the shortest path. By assumption on minimality of violation,

$$s_u^{k+1} \geq s_u^k \tag{9.3.3}$$

From 9.3.2 , it follows that

$$s_v^{k+1} \geq s_u^k + 1 \tag{9.3.4}$$

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 $s_v^k \leq s_u^k + 1$ From Equation 9.3.4 $s_v^{k+1} \geq s_v^k$. 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

$$s_u^k = s_v^k + 1$$

Combining with inequality 9.3.4 , we obtain $s_v^{k+1} = s_v^k + 2$ 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 $s_v^k = s_u^k + 1$. From monotonicity property $s_v^\ell \geq s_v^k$, so

$$s_v^\ell \geq s_u^k + 1 \tag{9.3.5}$$

Let $\ell (\geq k+1)$ be the next iteration when an augmenting path passes through (u, v)[3]. Then (u, v) must be a backward edge and therefore

$$s_u^\ell = s_v^\ell + 1 \geq s_u^k + 1 + 1 = s_u^k + 2$$

using inequality 9.3.5 . 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.

[3]it may not be a bottleneck edge.

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 $s - t$ 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 fundamental operation $contract(v_1, v_2)$ 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.

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
contract(v_1, v_2)

contract(u, v): Merge vertices u and v into w such that all neighbours of u and v are now neighbours of w.

Procedure *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 probabilistic 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 $\leq \frac{k}{kn/2} = \frac{2}{n}$[4].

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 $\mathcal{E}(i)$ 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 $\Pr[\bar{A}(1)] \geq 1 - 2/n$ and similarly, $\Pr[\bar{A}(i)|\mathcal{E}(i-1)] \geq 1 - 2/n_{i-1}$. Then, using the property of conditional probability

$$\Pr[\mathcal{E}(i)] = \Pr[\bar{A}(i) \cap \mathcal{E}(i-1)] = \Pr[\bar{A}(i)|\mathcal{E}(i-1)] \cdot \Pr[\mathcal{E}(i-1)]$$

where $\bar{A}$ denotes the complement of event A. We can use the above equation inductively obtain

$$\begin{aligned} \Pr[\mathcal{E}(i)] &\geq \prod_{i=1}^{n-t} (1 - 2/n_{i-1}) \\ &= \prod_{i=1}^{n-t} \left(1 - \tfrac{2}{n-i+1}\right) \\ &\geq \tfrac{t(t-1)}{n(n-1)} \end{aligned}$$

Claim 9.4 *The probability that a specific mincut $\mathcal{C}$ survives at the end of Partition(t) is at least $\frac{t(t-1)}{n(n-1)}$.*

[4]We will prove it only for the unweighted version but the proof can be extended using multiset arguments.

Therefore Partition (2) produces a mincut with probability $\Omega(\frac{1}{n^2})$. 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 9.10 *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 9.5 *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.

- 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

$$= \frac{\#E(v,w)}{deg(v)} \cdot \frac{deg(v)}{2|E|} + \frac{\#E(w,v)}{deg(w)} \frac{deg(w)}{2|E|}$$
$$= \frac{\#E(v,w)}{|E|}$$

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 9.11 *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 M' is such that there is no $e \in E - M'$ such that $M' \cup \{e\}$ is a matching, i.e., M' cannot be augmented. It is easy to see that a maximal matching can be easily constructed using a greedy approach.

Exercise 9.12 *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.

$$\max \sum_{x_e \in E} c_e \cdot x_e \quad s.t. A^{|V| \times |E|} X \leq [1, 1..]^T$$

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*[5] 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 9.6 *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 9.13 *Prove that $M' \oplus M$ consists of disjoint alternating cycles and paths.*

[5]Roughly speaking, the polytope of feasible solution has integral coordinates

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 9.7 *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 M'), then there are at 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 [6].

[6] A maximal matching has no length 1 augmenting path and hence it is within factor 2 of maximum matching

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,

$$\mathcal{P} = \cup_{i \geq 1} \mathcal{C}(T^{\mathcal{P}}(n^i))$$

where $\mathcal{C}(T^{\mathcal{P}}(n^i))$ denotes problems for which $O(n^i)$ time algorithms can be designed.

$$\mathcal{NP} = \cup_{i \geq 1} \mathcal{C}(T^{\mathcal{NP}}(n^i))$$

where $T^{\mathcal{NP}}()$ 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 $P_1, P_2 \mathcal{P}$ 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 $\mathcal{I}_1 \in P_1$ to an instance $\mathcal{I}_2 \in P_2$ such that the answer to $\mathcal{I}_2$ is *YES* iff the answer to $\mathcal{I}_1$ 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 $g(x) \in P_2$ [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.

[2] This is a short form of saying that $g(x)$ is an YES instance.

Claim 10.3 *If* $\Pi_1 \leq_{poly} \Pi_2$ *then*
(i) If there is a polynomial time algorithm for Π_2 *then there is a polynomial time algorithm for* Π_1.
(ii) If there is no polynomial time algorithm for Π_1*, then there cannot be a polynomial time algorithm for* Π_2.

Part (ii) is easily proved by contradiction. For part (i), if $p_1(n)$ is the running time of Π_1 and p_2 is the time of the reduction function, then there is an algorithm for Pi_1 that takes $p_1(p_2(n))$ steps where n is the input length for Π_1.

A problem Π is called *NP-hard* under polynomial reduction if for any problem $\Pi' \in \mathcal{NP}$, $\Pi' \leq_{poly} \Pi$.

A problem is Π 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* $A \leq_{poly} B$ *and* $B \leq_{poly} A$.

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 $P_1 \leq_{poly} P_2$) 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 $(x_1 \vee x_2..) \wedge (x_i \vee x_j \vee ..) \wedge \ldots (x_\ell \vee \ldots x_n)$

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 $\mathcal{NP}$, so we cannot explicitly design a reduction function. Other than the definition of $\mathcal{NP}$ 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 $\Pi \in \mathcal{NP}$, we want to show that $\Pi \leq_{poly} CNF-SAT$. In other words, given any instance of Π, say I_Π, we would like to define a boolean formula $B(I_\Pi)$ which has a satisfiable assignment iff I_Π is a YES instance. Moreover the length of $B(I_\Pi)$ should be polynomial time constructable (as a function of the length of I_Π).

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:

$$(x_{1,i} \vee x_{2,i} \ldots x_{k,i})$$

(ii) At most one variable is true :

$$(x_{1,i} \Rightarrow \bar{x}_{2,i} \wedge \bar{x}_{3,i} \ldots \wedge \bar{x}_{k,i}) \wedge (x_{2,i} \Rightarrow \bar{x}_{1,i} \wedge \bar{x}_{3,i} \ldots \wedge \bar{x}_{k,i}) \ldots \wedge (x_{k,i} \Rightarrow \bar{x}_{1,i} \wedge \bar{x}_{2,i} \ldots \wedge \bar{x}_{k-1,i})$$

where the implication $a \Rightarrow b$ is equivalent to $\bar{a} \vee b$.

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 $\mathcal{NP}$.

10.3 Common NP complete problems

To prove that a given problem P is NPC, the standard procedure is to establish that

(i) $P \in \mathcal{NP}$: This is usually the easier part.
(ii) $CNF - SAT \leq_{poly} P$. We already know that any $P' \in \mathcal{NP}$, $P' \leq_{poly} CNF - SAT$. 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 $\mathcal{P}$ and $\mathcal{NP}$ hogs the maximum limelight in complexity theory, there are many other related classes in their own right.

- $co-\mathcal{NP}$ 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 $co-\mathcal{NP}$. In general, we can't even design an NP algorithm for a problem in $co-\mathcal{NP}$, 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 $co-\mathcal{NP}$ 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

    $$\text{If } x \in L \Rightarrow \Pr[A \text{ accepts } x] \geq 1/2$$
    $$\text{If } x \notin L \Rightarrow \Pr[A \text{ accepts } x] = 0$$

    These algorithms can err on one side.

  - $\mathcal{BPP}$ When a randomized algorithm is allowed to err on both sides

    $$\text{If } x \in L \Rightarrow \Pr[A \text{ accepts } x] \geq 1/2 + \varepsilon$$
    $$\text{If } x \notin L \Rightarrow \Pr[A \text{ accepts } x] \leq 1/2 - \varepsilon$$

    where ε 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

  $$\mathcal{BPP} \subset \mathcal{NP}?$$

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 $\mathcal{P} = \mathcal{NP}$ 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 [4]. 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 ε 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* $\mathcal{P} = \mathcal{NP}$ *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 $\mid \left(\sum_{x \in S_1} x\right) - \left(\sum_{x \in S - S_1} x\right) \mid$ is minimized. A partition is *balanced* if the above difference is zero.

[4] The 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 $\{z_1, z_2 \dots z_j\}$ that sums to r - if no such subset exists then we define it as ϕ (empty subset). We can write the following recurrence

$S(j, r) = S(j-1, r-z_j) \cup z_j$ if z_j is included or $S(j-1, r)$ if z_j is not included or ϕ 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 ε and let $A = \lceil \frac{n}{\varepsilon} \rceil$ so that $\frac{1}{A} \leq \varepsilon/n$. Then we define a new scaled problem with the integers scaled as $z'_i = \lfloor \frac{z_i}{z/A} \rfloor$ 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 $\{z'_1, z'_2 \dots z'_n\}$ 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

$$C(S'_o) \geq C'(S'_o) \frac{\varepsilon z}{n} \geq C'(S_o) \frac{\varepsilon z}{n} \geq \left(C(S_o) \frac{n}{\varepsilon z} - |S_o| \right) \frac{\varepsilon z}{n} = C(S_o) - |S_o| \frac{\varepsilon z}{n}$$

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 $S = \{x_1, x_2 \dots x_n\}$ and a family of subsets $S_1, S_2 \dots S_m$ $S_i \subset S$, 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 $B/2$.

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', i.e. $C'/U' = \min\left\{\frac{C(S_{i_1})}{S_{i_1}-S'}, \frac{C(S_{i_2})}{S_{i_2}-S'} \ldots \frac{C(S_{i_k})}{S_{i_k}-S'}\right\}$ 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

$$C'/U' \leq \frac{C(S_{i_1}) + C(S_{i_1}) + \ldots C(S_{i_1})}{(S_{i_1}-S') + (S_{i_2}-S') + \ldots (S_{i_k}-S')} \leq \frac{C_o}{n-i+1}$$

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.

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 $\mathcal{T}$.
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 Δ 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 $\mathcal{P} = \mathcal{NP}$, 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 $w : E \rightarrow \mathbb{R}$.

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 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 choses it consistently with the maxcut. The expected size of the cut produced by the algorithm is

$$E[\sum_e w(e) \cdot X_e] = \sum_e w(e) \cdot E[X_e] \geq M_o/2$$

Therefore we have a simple randomized algorithm that attains a $\frac{1}{2}$ apprximation.

Exercise 10.6 *For an unweighted graph show that a simple greedy strategy leads to a $\frac{1}{2}$ approximation algorithm.*

Appendix A

Recurrences and generating functions

Given a sequence $a_1, a_2 \ldots a_n$ (i.e. a function with the domain as integers), a compact way of representing it is an equation in terms of itself, a recurrence relation. One of the most common examples is the Fibonacci sequence specified as $a_n = a_{n-1} + a_{n-2}$ for $n \geq 2$ and $a_0 = 0,\ a_1 = 1$. The values a_0, a_1 are known as the *boundary conditions*. Given this and the recurrence, we can compute the sequence step by step, or better still we can write a computer program. Sometimes, we would like to find the general term of the sequence. Very often, the running time of an algorithm is expressed as a recurrence and we would like to know the explicit function for the running time to make any predictions and comparisons. A typical recurrence arising from a *divide-and-conquer* algorithm is

$$a_{2n} = 2a_n + cn$$

which has a solution $a_n \leq 2cn\lceil \log_2 n \rceil$. In the context of algorithm analysis, we are often satisfied with an upper-bound. However, to the extent possible, it is desirable to obtain an exact expression.

Unfortunately, there is no general method for solving all recurrence relations. In this chapter, we discuss solutions to some important classes of recurrence equations. In the second part we discuss an important technique based on *generating functions* which are also important in their own right.

A.1 An iterative method - summation

As starters, some of the recurrence relations can be solved by summation or *guessing* and verifying by induction.

Example A.1 *The number of moves required to solve the* Tower of Hanoi *problem with n discs can be written as*

$$a_n = 2a_{n-1} + 1$$

By substituting for a_{n-1} this becomes

$$a_n = 2^2 a_{n-2} + 2 + 1$$

By expanding this till a_1, we obtain

$$a_n = 2^{n-1}a_1 + 2^{n-2} + \ldots.. + 1$$

This gives $a_n = 2^n - 1$ by using the formula for geometric series and $a_1 = 1$.

Example A.2 *For the recurrence*

$$a_{2n} = 2a_n + cn$$

we can use the same technique to show that $a_{2n} = \sum_{i=0} \log_2 n 2^i n/2^i \cdot c + 2na_1$.
Remark *We made an assumption that n is a power of 2. In the general case, this may present some technical complication but the nature of the answer remains unchanged. Consider the recurrence*

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Suppose $T(x) = cx \log_2 x$ for some constant $c > 0$ for all $x < n$. Then $T(n) = 2c\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + n$. Then $T(n) \leq cn \log_2(n/2) + n \leq cn \log_2 n - (cn) + n \leq cn \log_2 n$ for $c \geq 1$.

A very frequent recurrence equation that comes up in the context of divide-and-conquer algorithms (like mergesort) has the form

$T(n) = aT(n/b) + f(n)$ a, b are constants and $f(n)$ a positive monotonic function

Theorem A.1 *For the following different cases, the above recurrence has the following solutions*

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant ϵ, then $T(n)$ is $\Theta(n^{\log_b a})$.*
- *If $f(n) = O(n^{\log_b a})$ then $T(n)$ is $\Theta(n^{\log_b a} \log n)$.*
- *If $f(n) = O(n^{\log_b a + \epsilon})$ for some constant ϵ, and if $af(n/b)$ is $O(f(n))$ then $T(n)$ is $\Theta(f(n))$.*

Example A.3 *What is the maximum number of regions induced by n lines in the plane ? If we let L_n represent the number of regions, then we can write the following recurrence*

$$L_n \leq L_{n-1} + n \quad L_0 = 1$$

Again by the method of summation, we can arrive at the answer $L_n = \frac{n(n+1)}{2} + 1$.

Example A.4 *Let us try to solve the recurrence for Fibonacci, namely*

$$F_n = F_{n-1} + F_{n-2} \qquad F_0 = 0, \quad F_1 = 1$$

If we try to expand this in the way that we have done previously, it becomes unwieldy very quickly. Instead we "guess" the following solution

$$F_n = \frac{1}{\sqrt{5}} \left(\phi^n - \bar{\phi}^n \right)$$

where $\phi = \frac{(1+\sqrt{5})}{2}$ and $\bar{\phi} = \frac{(1-\sqrt{5})}{2}$. The above solution can be verified by induction. Of course it is far from clear how one can magically guess the right solution. We shall address this later in the chapter.

A.2 Linear recurrence equations

A recurrence of the form

$$c_0 a_r + c_1 a_{r-1} + c_2 a_{r-2} \ldots + c_k a_{r-k} = f(r)$$

where c_i are constants is called a *linear recurrence equation* of order k. Most of the above examples fall under this class. If $f(r) = 0$ then it is *homogeneous linear recurrence.*

A.2.1 Homogeneous equations

We will first outline the solution for the homogeneous class and then extend it to the general linear recurrence. Let us first determine the number of solutions. It appears that we must know the values of $a_1, a_2 \ldots a_k$ to compute the values of the sequence according to the recurrence. In absence of this there can be different solutions based on different boundary conditions. Given the k boundary conditions, we can *uniquely* determine the values of the sequence. Note that this is not true for a *non-linear* recurrence like

$$a_r{}^2 + a_{r-1} = 5 \text{ with } a_0 = 1$$

This observation (of unique solution) makes it somewhat easier for us to guess some solution and verify.

Let us guess a solution of the form $a_r = A\alpha^r$ where A is some constant. This may be justified from the solution of Example A.1. By substituting this in the homogeneous linear recurrence and simplification, we obtain the following equation

$$c_0\alpha^k + c_1\alpha^{k-1} \ldots + c_k = 0$$

This is called the *characteristic equation* of the recurrence relation and this degree k equation has k roots, say $\alpha_1, \alpha_2 \ldots \alpha_k$. If these are all distinct then the following is a solution to the recurrence

$$a_r{}^{(h)} = A_1\alpha_1^r + A_2\alpha_2^r + \ldots A_k\alpha_k^r$$

which is also called the *homogeneous solution to linear recurrence*. The values of $A_1, A_2 \ldots A_k$ can be determined from the k boundary conditions (by solving k simultaneous equations).

When the roots are not unique, i.e. some roots have multiplicity then for multiplicity m, $\alpha^n, n\alpha^n, n^2\alpha^n \ldots n^{m-1}\alpha^n$ are the associated solutions. This follows from the fact that if α is a multiple root of the characteristic equation, then it is also the root of the derivative of the equation.

A.2.2 Inhomogeneous equations

If $f(n) \neq 0$, then there is no general methodology. Solutions are known for some particular cases, known as *particular* solutions. Let $a_n^{(h)}$ be the solution by ignoring $f(n)$ and let $a_n^{(p)}$ be a particular solution then it can be verified that $a_n = a_n^{(h)} + a_n^{(p)}$ is a solution to the non-homogeneous recurrence.

The following is a table of some particular solutions

d a constant	B
dn	$B_1n + B_0$
dn^2	$B_2n^2 + B_1n + B_0$
ed^n, e, d are constants	Bd^n

Here B, B_0, B_1, B_2 are constants to be determined from initial conditions. When $f(n) = f_1(n) + f_2(n)$ is a sum of the above functions then we solve the equation for $f_1(n)$ and $f_2(n)$ separately and then add them in the end to obtain a particular solution for the f(n).

A.3 Generating functions

An alternative representation for a sequence $a_1, a_2 \ldots a_i$ is a polynomial function $a_1x+a_2x^2+\ldots a_ix^i$. Polynomials are very useful objects in mathematics, in particular as "placeholders." For example if we know that two polynomials are equal (i.e. they evaluate to the same value for all x), then all the corresponding coefficients must be equal. This follows from the well known property that a degree d polynomial has no more than d distinct roots (unless it is the zero polynomial). The issue of convergence is not important at this stage but will be relevant when we use the method of differentiation.

Example A.5 *Consider the problem of changing a Rs 100 note using notes of the following denomination - 50, 20, 10, 5 and 1. Suppose we have an infinite supply of each denomination then we can represent each of these using the following polynomials where the coefficient corresponding to x^i is non-zero if we can obtain a certain sum using the given denomination.*

$$P_1(x) = x^0 + x^1 + x^2 + \ldots$$
$$P_5(x) = x^0 + x^5 + x^{10} + x^{15} + \ldots$$
$$P_{10}(x) = x^0 + x^{10} + x^{20} + x^{30} + \ldots$$
$$P_{20}(x) = x^0 + x^{20} + x^{40} + x^{60} + \ldots$$
$$P_{50}(x) = x^0 + x^{50} + x^{100} + x^{150} + \ldots$$

For example, we cannot have 51 to 99 using Rs 50,so all those coefficients are zero.

By multiplying these polynomials we obtain

$$P(x) = E_0 + E_1x + E_2x^2 + \ldots E_{100}x^{100} + \ldots E_ix^i$$

where E_i is the number of ways the terms of the polynomials can combine such that the sum of the exponents is 100. Convince yourself that this is precisely what we are looking for. However we must still obtain a formula for E_{100} or more generally E_i, which the number of ways of changing a sum of i.

Note that for the polynomials $P_1, P_5 \ldots P_{50}$, the following holds

$$P_k(1 - x^k) = 1 \quad \textit{for } k = 1, 5, ..50 \textit{ giving}$$
$$P(x) = \frac{1}{(1-x)(1-x^5)(1-x^{10})(1-x^{20})(1-x^{50})}$$

We can now use the observations that $\frac{1}{1-x} = 1 + x^2 + x^3 \ldots$ and $\frac{1-x^5}{(1-x)(1-x^5)} = 1+x^2+x^3 \ldots$. So the corresponding coefficients are related by $B_n = A_n + B_{n-5}$ where A and B are the coefficients of the polynomials $\frac{1}{1-x}$ and $\frac{1}{(1-x)(1-x^5)}$. Since $A_n = 1$, this is a linear recurrence. Find the final answer by extending these observations.

Let us try the method of generating function on the Fibonacci sequence.

Example A.6 *Let the generating function be* $G(z) = F_0 + F_1 x + F_2 x^2 \ldots F_n x^n$ *where* F_i *is the* i*-th Fibonacci number. Then* $G(z) - zG(z) - z^2 G(z)$ *can be written as the infinite sequence*

$$F_0 + (F_1 - F_2)z + (F_2 - F_1 - F_0)z^2 + \ldots (F_{i+2} - F_{i+1} - F_i)z^{i+2} + \ldots = z$$

for $F_0 = 0, F_1 = 1$*. Therefore* $G(z) = \frac{z}{1-z-z^2}$*. This can be worked out to be*

$$G(z) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\bar{\phi} z}\right)$$

where $\bar{\phi} = 1 - \phi = \frac{1}{2}\left(1 - \sqrt{5}\right)$.

A.3.1 Binomial theorem

The use of generating functions necessitates computation of the coefficients of power series of the form $(1+x)^\alpha$ for $|x| < 1$ and *any* α. For that the following result is very useful - the coefficient of x^k is given by

$$C(\alpha, k) = \frac{\alpha \cdot (\alpha - 1) \ldots (\alpha - k + 1)}{k \cdot (k-1) \ldots 1}$$

This can be seen from an application of Taylor's series. Let $f(x) = (1+x)^\alpha$. Then from Taylor's theorem, expanding around 0 for some z,

$$f(z) = f(0) + zf'(0) + \alpha \cdot z + z^2 \frac{f''(0)}{2!} + \ldots z^k \frac{f^{(k)}(0)}{k!} \ldots$$

$$= f(0) + 1 + z^2 \frac{\alpha(\alpha-1)}{2!} + \ldots C(\alpha, k) + \ldots$$

Therefore $(1+z)^\alpha = \sum_{i=0}^{\infty} C(\alpha, i) z^i$ which is known as the binomial theorem.

A.4 Exponential generating functions

If the terms of a sequence is growing too rapidly, i.e. the n-th term exceeds x^n for any $0 < x < 1$, then it may not converge. It is known that a sequence converges iff the sequence $|a_n|^{1/n}$ is bounded. Then it makes sense to divide the coefficients by a rapidly growing function like $n!$. For example, if we consider the generating function for the number of permutations of n *identical* objects

$$G(z) = 1 + \frac{p_1}{1!} z + \frac{p_2}{2!} z^2 \ldots \frac{p_i}{i!} z^i$$

where $p_i = P(i,i)$. Then $G(z) = e^z$. The number of permutations of r objects when selected out of (an infinite supply of) n kinds of objects is given by the exponential generating function (EGF)

$$\left(1 + \frac{p_1}{1!}z + \frac{p_2}{2!}z^2 \ldots\right)^n = e^{nx} = \sum_{r=0}^{\infty} \frac{n^r}{r!}x^r$$

Example A.7 *Let D_n denote the number of derangements of n objects. Then it can be shown that $D_n = (n-1)(D_{n-1} + D_{n-2})$. This can be rewritten as $D_n - nD_{n-1} = -(D_{n-1} - (n-2)D_{n-2}$. Iterating this, we obtain $D_n - nD_{n-1} = (-1)^{n-2}(D_2 - 2D_1)$. Using $D_2 = 1, D_1 = 0$, we obtain*

$$D_n - nD_{n-1} = (-1)^{n-2} = (-1)^n.$$

Multiplying both sides by $\frac{x^n}{n!}$, and summing from $n = 2$ to ∞, we obtain

$$\sum_{n=2}^{\infty} \frac{D_n}{n!}x^n - \sum_{n=2}^{\infty} \frac{nD_{n-1}}{n!}x^n = \sum_{n=2}^{\infty} \frac{(-1)^n}{n!}x^n$$

If we let $D(x)$ represent the exponential generating function for derangements, after simplification, we get

$$D(x) - D_1 x - D_0 - x(D(x) - D_0) = e^{-x} - (1-x)$$

or $D(x) = \frac{e^{-x}}{1-x}$.

A.5 Recurrences with two variables

For selecting r out of n distinct objects, we can write the familiar recurrence

$$C(n,r) = C(n-1, r-1) + C(n-1, r)$$

with boundary conditions $C(n,0) = 1$ and $C(n,1) = n$.

The general form of a linear recurrence with constant coefficients that has two indices is

$$C_{n,r}a_{n,r} + C_{n,r-1}a_{n,r-1} + \ldots C_{n-k,r}a_{n-k,r} \ldots C_{0,r}a_{0,r} + \ldots = f(n,r)$$

where $C_{i,j}$ are constants. We will use the technique of generating functions to extend the one variable method. Let

$$A_0(x) = a_{0,0} + a_{0,1}x + \ldots a_{0,r}x^r$$

$$A_1(x) = a_{1,0} + a_{1,1}x + \ldots a_{1,r}x^r$$
$$A_n(x) = a_{n,0} + a_{n,1}x + \ldots a_{n,r}x^r$$

Then we can define a generating function with $A_0(x), A_1(x)A_3(x)\ldots$ as the sequence - the new indeterminate can be chosen as y.

$$A_y(x) = A_0(x) + A_1(x)y + A_2(x)y^2 \ldots A_n(x)y^n$$

For the above example, we have

$$F_n(x) = C(n,0) + C(n,1)x + C(n,2)x^2 + \ldots C(n,r)x^r + \ldots$$

$$\sum_{r=0}^{\infty} C(n,r)x^r = \sum_{r=1}^{\infty} C(n-1,r-1)x^r + \sum_{r=0}^{\infty} C(n-1,r)x^r$$
$$F_n(x) - C(n,0) = xF_{n-1}(x) + F_{n-1}(x) - C(n-1,0)$$
$$F_n(x) = (1+x)F_{n-1}(x)$$

or $F_n(x) = (1+x)^n C(0,0) = (1+x)^n$ as expected.

Appendix B

Refresher in discrete probability and probabilistic inequalities

The sample space Ω may be infinite with infinite elements that are called *elementary events*. For example consider the experiment where we must toss a coin until a head comes up for the first time. A *probability space* consists of a sample space with a *probability measure* associated with the elementary events. The probability measure Pr is a real valued function on events of the sample space and satisfies the following

1. For all $A \subset \Omega$, $0 \leq \Pr[A] \leq 1$
2. $\Pr[\Omega] = 1$
3. For mutually disjoint events $E_1, E_2 \ldots$, $\Pr[\cup_i E_i] = \sum_i \Pr[E_i]$

Sometimes we are only interested in a certain collection of events (rather the entire sample space)a, say F. If F is closed under union and complementation, then the above properties can be modified in a way as if $F = \Omega$.

The principle of Inclusion-Exclusion has its counterpart in the probabilistic world, namely

Lemma B.1

$$\Pr[\cup_i E_i] = \sum_i \Pr[E_i] - \sum_{i<j} \Pr[E_i \cap E_j] + \sum_{i<j<k} \Pr[E_i \cap E_j \cap E_k] \ldots$$

Definition B.1 *A* random variable *(r.v.) X is a real-valued function over the sample space, $X : \Omega \to \mathbb{R}$. A* discrete random variable *is a random variable whose range is finite or a countable finite subset of* $\mathbb{R}$.
The distribution function $F_X : \mathbb{R} \to (0, 1]$ *for a random variable X is defined as* $F_X(x) \leq \Pr[X = x]$. *The* probability density function *of a discrete r.v. X, f_X is*

given by $f_X(x) = \Pr[X = x]$.
The expectation *of a r.v.* X*, denoted by* $E[X] = \sum_x x \cdot \Pr[X = x]$.

A very useful property of expectation, called the *linearity property* can be stated as follows

Lemma B.2 *If* X *and* Y *are random variables, then*

$$E[X + Y] = E[X] + E[Y]$$

Remark Note that X and Y do not have to be independent !

Definition B.2 *The* conditional probability *of* E_1 given E_2 *is denoted by* $\Pr[E_1|E_2]$ *and is given by*

$$\frac{\Pr[E_1 \cap E_2]}{\Pr[E_2]}$$

assuming $\Pr[E_2] > 0$.

Definition B.3 *A collection of events* $\{E_i | i \in I\}$ *is* independent *if for all subsets* $S \subset I$

$$\Pr[\cap_{i \in S} E_i] = \Pi_{i \in S} \Pr[E_i]$$

Remark E_1 and E_2 are independent if $\Pr[E_1|E_2] = \Pr[E_2]$.

The *conditional probability* of a random variable X with respect to another random variable Y is denoted by $\Pr[X = x|Y = y]$ is similar to the previous definition with events E_1, E_2 as $X = x$ and $Y = y$ respectively. The *conditional expectation* is defined as

$$E[X|Y = y] = \sum_x \Pr x \cdot [X = x|Y = y]$$

The **theorem of total expectation** that can be proved easily states that

$$E[X] = \sum_y E[X|Y = y]$$

B.1 Probability generating functions

The notion of generating functions have useful applications in the context of probability calculations also. Given a non-negative integer-valued discrete random variable X with $\Pr[X = k] = p_k$, the probability generating function (PGF) of X is given by

$$G_X(z) = \sum_{i=0}^{\infty} p_i z^i = p_0 + p_1 z + \ldots p_i z^i \ldots$$

This is also known as the z-transform of X and it is easily seen that $G_X(1) = 1 = \sum_i p_i$. The convergence of the PGF is an important issue for some calculations involving differentiation of the PGF. For example,

$$E[X] = \frac{dG_X(z)}{dz}|z = 1$$

The notion of *expectation of random variable* can be extended to function $f(X)$ of random variable X in the following way

$$E[f(X)] = \sum_i p_i f(X = i)$$

Therefore, PGF of X is the same as $E[z^X]$. A particularly useful quantity for a number of probabilistic calculations is the **Moment Generating Function** (MGF) defined as

$$M_X(\theta) = E[e^{X\theta}]$$

Since

$$e^{X\theta} = 1 + X\theta + \frac{X^2\theta^2}{2!} + \ldots \frac{X^k\theta^k}{k!} \ldots$$

$$M_X(\theta) = 1 + E[X]\theta + \ldots \frac{E[X^k]\theta^k}{k!} \ldots$$

from which $E[X^k]$ also known as *higher moments* can be calculated. There is also a very useful theorem known for *independent* random variables $Y_1, Y_2 \ldots Y_t$. If $Y = Y_1 + Y_2 + \ldots Y_t$, then

$$M_Y(\theta) = M_{Y_1}(\theta) \cdot M_{Y_2}(\theta) \cdot \ldots M_{Y_t}(\theta)$$

i.e., the MGF of the sum of independent random variables is the product of the individual MGF's.

B.1.1 Probabilistic inequalities

In many applications, especially in the analysis of randomized algorithms, we want to guarantee correctness or running time. Suppose we have a bound on the expectation. Then the following inequality known as Markov's inequality can be used.

Markov's inequality

$$\Pr[X \geq kE[X]] \leq \frac{1}{k} \tag{B.1.1}$$

Unfortunately there is no symmetric result.

If we have knowledge of the second moment, then the following gives a stronger result

Chebychev's inequality

$$\Pr[(X - E[X])^2 \geq t] \leq \frac{\sigma^2}{t} \tag{B.1.2}$$

where σ is the variance, i.e. $E^2[X] - E[X^2]$.

With knowledge of higher moments, then we have the following inequality. If $X = \sum_i^n x_i$ is the sum of n mutually independent random variables where x_i is uniformly distributed in {-1 , +1 }, then for any $\delta > 0$,

Chernoff bounds

$$\Pr[X \geq \Delta] \leq e^{-\lambda\Delta} E[e^{\lambda X}] \tag{B.1.3}$$

If we choose $\lambda = \Delta/n$, the RHS becomes $e^{-\Delta^2/2n}$ using a result that $\frac{e^{-\lambda}+e^{\lambda}}{2} = coshh(\lambda) \leq e^{\lambda^2/2}$.

A more useful form of the above inequality is for a situation where a random variable X is the sum of n independent 0-1 valued Poisson trials with a success probability of p_i in each trial. If $\sum_i p_i = np$, the following equations give us concentration bounds of deviation of X from the expected value of np. The first equation is more useful for large deviations whereas the other two are useful for small deviations from a large expected value.

$$Prob(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np} \tag{B.1.4}$$

$$Prob(X \leq (1-\epsilon)pn) \leq exp(-\epsilon^2 np/2) \tag{B.1.5}$$

$$Prob(X \geq (1+\epsilon)np) \leq exp(-\epsilon^2 np/3) \tag{B.1.6}$$

for all $0 < \epsilon < 1$.

A special case of non-independent random variables

Consider n 0-1 random variables $y_1, y_2, \ldots y_n$ such that $\Pr[y_i = 1 | y_1, y_2 \ldots y_{i-1}] \leq p_i$ and $\sum p_i = np$. The random variables are not known to be independent but only bounded in the way described above. In such a case, we will not be able to directly invoke the previous Chernoff bounds directly but we will show the following

Lemma B.3 *Let $Y = \sum_i y_i$ and let $X = \sum_i x_i$ where x_i are* independent *Poisson trials with $x_i = p_i$. Then*

$$\Pr[Y \geq k] \leq [X \geq k] \forall k,\ 0 \leq k \leq n$$

In this case, the random variable X is said to *stochastically dominate* Y.

Therefore we can invoke the Chernoff bounds on X to obtain a bound on Y. We will prove the above property by induction on i (number of variables). For $i = 1$, (for all k) this is true by definition. Suppose this is true for $i < t$ (for all $k \leq i$) and let $i = t$. Let $X_i = x_1 + x_2 \ldots x_i$ and $Y_i = y_1 + y_2 \ldots y_i$. Then

$$\Pr[X_t \geq k] = \Pr[X_{t-1} \geq k] + \Pr[X_{t-1} = k - 1 \cap x_t = 1]$$

Since x_i's are independent, we can rewrite the above equation as

$$\Pr[X_t \geq k] = (p_t + 1 - p_t) \Pr[X_{t-1} \geq k] + \Pr[X_{t-1} = k - 1] \cdot p_t$$

$$= p_t(\Pr[X_{t-1} \geq k] + \Pr[X_{t-1} = k - 1]) + (1 - p_t) \cdot \Pr[X_{t-1} \geq k]$$

$$= p_t \Pr[X_{t-1} \geq k - 1] + (1 - p_t) \cdot \Pr[X_{t-1} \geq k]$$

Similarly

$$\Pr[Y_t \geq k] \leq p_t(\Pr[Y_{t-1} \geq k - 1] + (1 - p_t) \cdot \Pr[Y_{t-1} \geq k]$$

where the inequality exists because $\Pr[Y_{t-1} = k - 1 \cap y_t = 1] = \Pr[y_t = 1 | Y_{t-1} = k - 1] \cdot \Pr[Y_{t-1} = k - 1] \leq p_t \cdot \Pr[Y_{t-1} = k - 1]$. By comparing the two equations term by term and invoking induction hypothesis, the result follows.

An alternate approach to the above proof is using the moment generating function and works in a more general scenario.

$$\mathbb{E}[e^{sY}] = \mathbb{E}[e^{s(y_1 + y_2 \ldots)}] = \mathbb{E}[e^{sy_n} \cdot e^{s(y_1 + y_2 \ldots y_{n-1})}]$$

$$= \mathbb{E}[\mathbb{E}[e^{sy_n} \cdot e^{s(\alpha_1 + \alpha_2 \ldots \alpha_{n-1})} | y_1 = \alpha_1, y_2 = \alpha_2 \ldots]]$$

where $\alpha_i \in \{0, 1\}$ and the outer expectation is over all the values of $y_i, 1 \leq i \leq n - 1$. Thus the above expression can be rewritten as

$$\mathbb{E}[e^{s(y_1 + y_2 + \ldots y_{n-1})}] \cdot \mathbb{E}[e^{sy_n} | y_1 = \alpha_1, y_2 = \alpha_2 \ldots]$$

Since $\Pr[y_i = 1 | y_1, y_2 \ldots y_{i-1}] \leq p_i$, it follows that $\mathbb{E}[e^{sy_n} | y_1 = \alpha_1, y_2 = \alpha_2 \ldots] \leq (1 - p_n) + p_n \cdot e^s$. So from induction it follows that the MGF of Y can be bounded by the MGF of Binomial distribution for which the Chernoff bounds can be applied.

Appendix C

Generating Random numbers

The performance of any randomized algorithm is closely dependent on the underlying random number generator (RNG) in terms of efficiency. A common underlying assumption is availability of a RNG that generates a number uniformly in some range $[0, 1]$ in unit time or alternately $\log N$ independent random *bits* in the discrete case for the interval $[0..N]$. This primitive is available in all standard programming language - we will refer to this RNG as $\mathcal{U}$. We will need to adapt this to various scenarios that we describe below.

C.1 Generating a random variate for an arbitrary distribution

Let a distrubition $\mathcal{D}$ be described in terms of cumulative distribution function (CDF)$F_{\mathcal{D}}(s)$, $0 \leq s \leq N$ and a corresponding distribution function $f(s)$ $\sum_{i=1}^{t} f(i) = F_{\mathcal{D}}(t)$. Moreover $F(N) = 1$ (the subscript $\mathcal{D}$ is ommitted as it is clear from the context) . It follows that $f(s) = F(s) - F(s-1)$. We would like to generate a random variate according to $\mathcal{D}$. The distribution $\mathcal{D}$ can be thought of generating a random variable X with weight $w_i = f(i)$ where $\sum_i w_i = 1$. We can divide the interval $[0, 1]$ into consecutive subintervals $I_1, I_2 \ldots$ such that $I_j = w_j$. Note that $F_{\mathcal{D}}(i) = \sum_{j=0}^{i} w_j$. Using the RNG of the uniform distribution $\mathcal{U}$, the probability that the uniformly distributed random number X falls in the subinterval $I_j = w_j/(\sum_i w_i) = w_j$. So if we output a random number Y such that $Y = j$, then Y will have the desired distribution $\mathcal{D}$. In other words, we need to compute the corresponding j from the value of X such that $F_{\mathcal{D}}(j-1) < X \leq F_{\mathcal{D}}(j)$.

If we have an explicit formula for $F_{\mathcal{D}}$ then we can use binary search to find the corresponding j in $O(\log N)$ computations since F is monotonic.

C.2 Generating random variates from a sequential file

Suppose a file contains N records from which we would like to sample subset of n records uniformly at random. There are several approaches to thi basic problem

- *Sampling with replacement* We can use $\mathcal{U}$ to repeatedly sample an element from the file. This could lead to *duplicates*.

- *Sampling without replacement* We can use the previous method to choose the next sample but we will reject duplicates. The result is an uniform sample but the efficientcy may suffer. In particular, the expected number of times we need to invoke the RNG for the k-th sample is $\frac{N}{N-k}$.

- *Sampling in a sequential order* Here we want to pick the samples $S_1, S_2 \ldots S_n$ in an increasing order from the file, i.e., $S_i \in [1..N]$ and $S_i < S_{i+1}$. This has applications to processes where we can scan the records exactly once and we cannot retrace.

  Here we can find out the distribution of the S_{i+1} given that we have generated m ($< n$) samples from the t records that we have scanned. Here we have two cases 1. We can access each record - then the next record will be chosen with probability $\frac{n-m}{N-t}$. This is the easy case but less efficient since we need to use $\mathcal{U}$ roughly N times.

  2. The distribution of $S_{i+1} - S_i$ is given by

$$F(s) = 1 - \frac{\binom{(N-t-s)}{(n-m)}}{\binom{(N-t)}{(n-m)}}$$

- *Sampling in a sequential order from an arbitrarily large file*, i.e., without the knowledge of N. This is the scenario in a streaming algorithm.

  In this case, we always maintain the following invariant
  Among the i records that we have scanned so far, we have a sample of n elements chosen uniformly at random

  For this, the first n records must be chosen in the sample. When we scan the next record (which may not happen if the file has ended), we want to restore this invariant for the $i + 1$ records. Clearly the $i + 1$-th record could be in the sample with some probability, say p_{i+1} and if picked, one of the previous sampled record must be replaced.

Note that $p_{i+1} = \frac{n}{i+1}$ - this follows from the equation that $\binom{i+1}{n} = \binom{i}{n-1} + \binom{i}{n}$. The first term corresponds to the case where the $i+1$-th element is chosen in the sample and $\frac{\binom{i}{n-1}}{\binom{i+1}{n}} = n/(i+1)$.

If the (i+1)-th record is indeed chosen, we drop one of the previously chosen n samples with equal probability. To see this, notice that the invariant guarantees that the $S_{n,i}$, the sample from the first i records is a uniformly chosen sample of n elements. We claim that dropping one of the samples uniformly at random gives us $S_{n-1,i}$, i.e., a uniform $n-1$ sample. The probability that a specific subset of $n-1$ elements, say S^* is chosen is the probability that $S^* \cup x$ was chosen, $(x \notin S^*)$ and x was dropped. You can verify that

$$\frac{1}{n} \cdot (i - n + 1) \cdot \frac{1}{\binom{i}{n}} = \frac{1}{\binom{i}{n-1}}$$

The RHS is the uniform probability of an $n-1$ sample.