Efficiency and Computational Complexity
What is a “good” algorithm/program?

• Solution is simple but powerful/general
• Easily understood by reader
• Easily modifiable and maintainable
• Correct for clearly defined situations
• Efficient in space and time
• Well documented
  – usable by those who do NOT understand the detailed working
• Portable across computers
• Can be used as a sub-program

Courtesy Prof P. R. Panda, CSE, IIT Delhi
Efficiency of Algorithms

• Algorithms/Programs evaluated in terms of:
  – execution time
  – memory space
Identifying Redundant Computation

- Redundant computation can be moved from loop
  - loop-invariant computation

```python
x = 0
for i in range(10):
    y = (a*a*a+c)*x*x + (b*b)*x + c
    print(y)
    x = x + 0.01
```

```python
x = 0
t1 = (a*a*a + c)
t2 = b*b
for i in range(10):
    y = t1*x*x + t2*x + c
    print(y)
    x = x + 0.01
```
Computational Complexity

• Quantitative measure of algorithm’s performance needed
  – independent of programming language
  – independent of machine

• Performance is characterised in terms of size of the problem being solved
  – if problem size is n (e.g., searching in array of n integers)
  – how many operations are performed by algorithm?
    • as a function of n
    • indirectly measures execution time
  – how much memory is required for storage
    • as a function of n
Rate of growth of functions

- $y = ax + b$
- $y = \log x$
- $y = ax^2 + b$
- $y = 2^x$

Courtesy Prof P. R. Panda, CSE, IIT Delhi
Asymptotic Analysis

• What happens for large $n$?
Order Notation

• Function $g(n)$ is of order $O(f(n))$ if
  – there exists $c$ for which $g(n) \leq cf(n)$
  – for all $n \geq$ some $n_1$

Courtesy Prof P. R. Panda, CSE, IIT Delhi
Rate of Growth of Functions

• Given $f(n)$ and $g(n)$, which grows faster?
  – If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, then $g(n)$ is faster
  – If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$, then $f(n)$ is faster
  – If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{non-zero constant}$, then both grow at the same rate

• Two polynomials of the same degree grow at the same rate

• $O(1)$ means constant time
  – independent of $n$
Why use $O(\ )$ for measuring Complexity?

- Hiding constants
  - crude/approximate
  - easier to compute
  - holds across machines

- How does algorithm scale for increasing $n$?
- Which algorithm is better for large problem size?
Computing the Complexity

- Estimate the number of operations
  - as a function of input size
- One for initialisation
- Loop executes n times
  - Max. of one comparison and one assignment in each iteration
  - Max. 2n operations for loop
- Total operations = 1 + 2n
- Complexity is O(n)

```python
def largest(L):
    lelement = L[0]
    for i in range(len(L)):
        if (L[i] > lelement):
            lelement = L[i]
    return lelement
```

Courtesy Prof P. R. Panda, CSE, IIT Delhi
Complexity of Matrix Multiplication

- **In k-loop**
  - 2 operations: one +, one *
  - n iterations
  - 2n operations
- **In j-loop**
  - 1 assignment
  - 2n operations in k-loop
  - n iterations
  - Total = n * (2n+1) operations
- **In i-loop**
  - n iterations
  - Total = n * n * (2n+1) operations
- **Complexity is O(n^3)**

ALGORITHM MatMult (int n)
BEGIN
for i = 1 to n
  for j = 1 to n
    BEGIN
      C[i][j] = 0
      for k = 1 to n
        C[i][j] = C[i][j] + A[i][k] * B[k][j]
    END
END
END

Courtesy Prof P. R. Panda, CSE, IIT Delhi
Efficient Algorithms

- Problem:
  - Given real number $x$ and integer $n$
  - Write an algorithm to calculate $x^n$
First Algorithm

• Power
  – power(x,n) = 1 for n = 0
  – power(x, n) = x * power(x,n-1) for n>1

```python
def power(x,n):
    if (n==0):
        return 1
    else:
        return x*power(x,n-1)
```
Recurrence Relation

• Power
  – \( T(n) = 1, \) if \( n = 0 \)
  – \( T(n) = T(n-1) + 1, \) otherwise
Solving Recurrence Relations

• By Telescoping
  – substitution

\[ T(n) = T(n-1) + 1 \]
\[ = T(n-2) + 2 \]
\[ = T(n-3) + 3 \]
\[ \vdots \]
\[ = T(2) + n-2 \]
\[ = T(1) + n-1 \]
\[ = T(0) + n \]
\[ = 1 + n \]
\[ = O(n) \]
Fast Algorithm

• Fast Power

def fpower(x,n):
    if (n==0):
        return 1
    else:
        y = fpower(x,int(n/2))
        if (n%2 == 0):
            return y*y
        else:
            return x*y*y

Courtesy Prof P. R. Panda, CSE, IIT Delhi
Recurrence Relation

- **Fast Power**
  - $T(n) = 1$, if $n = 0$
  - $T(n) = 1$, if $n = 1$
  - $T(n) = T(n/2) + c$, otherwise
Solving Recurrence Relations

• By Telescoping
  – substitution

\[
T(n) = T(n/2) + c
= T(n/2^2) + 2c
= T(n/2^3) + 3c
... \\
= T(n/2^{m-1}) + (m-1)c
= T(n/2^m) + mc
= O(m)
= O(\log_2 n)
\]

...where \( m = \log_2 n \)
Binary Search

- “Divide and Conquer” strategy
  - at every stage, we reduce the size of the problem to half the earlier stage
- Strategy: Compare with the middle element of current range, and eliminate half of the range

```python
# Algorithm Binary Search
def binarysearch(ar, l, r, x):
    while l<=r:
        mid = l+(r-l)//2
        if ar[mid] == x:
            return mid
        elif ar[mid] < x:
            l = mid + 1
        else:
            r = mid - 1
    return -1
```

Iterative
### Binary Search

#### Recursive

```
# Algorithm Binary Search
def binarysearch (ar, l, r, x):
    if r >= l:
        mid = l + (r - l)//2
        if ar[mid] == x:
            return mid
        elif ar[mid] > x:
            return binarysearch(ar, l, mid-1, x)
        else:
            return binarysearch(ar, mid+1, r, x)
    else:
        return -1
```
Recurrence Relation

• Binary Search
  – $T(n) = 1$, if $n = 1$
  – $T(n) = T(n/2) + O(1)$, otherwise

– Solution $O(\log_2 n)$
Sorting an Array

• Rearranging array contents in increasing or decreasing order

How do we sort?

Sort in increasing order

Courtesy Prof P R Panda CSE, IIT Delhi
for  in range(len(A)) :
    k = position of min. element
    between A [i] and A [N-1]
    Swap A [i] and A [k]
for in range(len(A)):
    k = position of min. element between A[i] and A[N-1]
    Swap A[i] and A[k]

for j in range(i+1, len(A)):
    if A[min_index] > A[j]:
        min_index = j

t = A[i]
A[i] = A[k]
A[k] = t

Courtesy Prof P R Panda CSE, IIT Delhi
Simple Sorting Algorithm

Find Min for first time n elements: n-1 comparisons
Next time : n-2
.
.
up to 1

Total time = (n-1)+(n-2)+….+1=(n*(n-1))/2
O(n²)

Courtesy Prof P R Panda CSE, IIT Delhi