# Algorithms

## Method of Analysis

**Aposterior** (Platform dependent)

Adv: give exact values

Disadv: difficult for comparision

→ Difficult to determine exact times

**Apriori** (Platform independent)

Algo → Ram model → (Time, space)

→ Here we use metric i.e. No of fundamental operations

Adv: Easy for comparision

2 methods: Step Count, Order of magnitude.

→ Input class: certain ordering of on i/p

→ Classification of i/p class: Best, worst, Average

Avg case TC: $\sum_{i=1}^{k} t_i * P_i$

$k$ → no of i/p class

$t_i$ → time for i/p class $i$

$P_i$ → probability.

## Asymptotic Notations:

(i) Big-Oh ($O$): $f(x)$ is $O(g(x))$ iff

$f(x) \le c \cdot g(x)$ for some $c$, $\forall x > k$

(ii) Big-Omega ($\Omega$): $f(x)$ is $\Omega(g(x))$ iff

$f(x) \ge c \cdot g(x)$ for some $c$, $\forall x > k$

(iii) Theta ($\theta$): $f(x)$ is $\theta(g(x))$ iff

$c_1 \cdot g(x) \le f(x) \le c_2 \cdot g(x)$, for some $c$, $\forall x > k$

(iv) Small Oh ($o$): $f(x)$ is $o(g(x))$ iff

$f(x) < c \cdot g(x)$ for all $c$, for all $x > k$

(v) Small omega ($\omega$): $f(x)$ is $\omega(g(x))$ iff

$f(x) > c \cdot g(x)$ $\forall c$, $\forall x > k$

* $f(x) = \theta(g(x)) \iff f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))$
* $f(x) = O(g(x)) \iff g(x) = \Omega(f(x))$
* $f(x) = o(g(x)) \iff g(x) = \omega(f(x))$

* $f(x) = o(g(x)) \Rightarrow f(x) = O(g(x))$
* $f(x) = \omega(g(x)) \Rightarrow f(x) = \Omega(g(x))$
* Discrete properties: reflexive, symmetric, transitive

Eg. $f(n) = O(f(n))$ ; $f(n) = \theta(g(n))$, $g(n) = \theta(f(n))$

Transpose symmetry: $(O, \Omega)$, $(o, \omega)$

* If $f(n) = O(g(n))$ & $d(n) = O(e(n))$ then

$f(n) + d(n) = O[\max(g(n), e(n))]$

$f(n) * d(n) = O(g(n) \cdot e(n))$

* Tricotomy property is not satisfied by functions.

**Note:**

→ $\sum_{i=1}^{n} i \cdot 2^i = 2^{n+1}(n-1) + 2$

→ $n! = O(n^n)$

$n! = \Omega(a^n)$

→ $(\log n)^x = O(n^y)$

→ $n^x = O(a^n)$

$a^{\log_c b} = b^{\log_c a}$

$\log_a b = \frac{1}{\log_b a}$

$\log_a b = \frac{\log_x b}{\log_x a}$

**Note:**

```
Algo(n)
{
  if(n>0)
    algo(n-1);
}
```

$T(n) = a + T(n-1)$

Time for comparision must be included.

## Space Complexity:

* It is space req for computation.
* Space complexity, $S(n) = C + S_p$

$C$: fixed (Instn & fixed mem)

$S_p$: depends on i/p (temporary data, recursion stack)

* $S(n) = O(T(n))$
* If $S(n)$ is $O(1)$ or $O(\log n)$ for rec algo then algo is said to be space efficient.

# Divide & Conquer

* TC: $\begin{cases} f(n) & , n \text{ is small} \\ aT(n/b)+g(n), & n \text{ is large} \end{cases}$

$a \rightarrow$ no of subproblems

$n/b \rightarrow$ size of each subproblem

$a \geq 1;\ b > 1$

**Maxi-min**: no of comp be $T(n)$

$$T(n) = \begin{cases} 0, & \text{if } n=1 \\ 1, & n=2 \\ 2T(n/2)+2 & n>2 \end{cases}$$

$\Rightarrow T(n) = \frac{3n}{2} - 2$

$S(n) = O(\log n)$

**Merge Sort:**

Merge process:

no of comp $\begin{cases} \text{best case: } \min(m,n) \\ \text{worst case: } m+n-1 \end{cases}$

$\therefore TC: O(n+m)$

algo: $mid \leftarrow (l+h)/2;$

MS(a, l, mid)

MS(a, mid+1, h)

merge(a, l, mid, h)

$\therefore TC = \begin{cases} c; & n=1 \\ 2T(n/2)+bn; & n>1\ (b>0) \end{cases}$

$SC = O(n)+O(\log n) = O(n)$

- 2-way | bottomup merge sort.

**Binary Search:**

$$T(n) = \begin{cases} c, & n=1 \\ T(n/2)+a, & n>1 \end{cases}$$

$T(n) = O(\log n)$

$S(n) = O(\log n)$

**Linear Search:**

Avg no of comp: $\frac{n+1}{2}$

**Matrix Multiplication:**

In naive approach & normal D&C approach TC is $O(n^3)$

In normal D&C

$TC: 8T(n/2)+\underbrace{bn^2}_{\text{addition}}, n>2$

SC: $O(\log n)$

**Strassens Matrix Multiplications:**

It reduces no of multiplications from 8 to 7.

$\therefore TC = O(n^{\log_2 7}) = O(n^{2.81})$

$SC: O(n^2)$

**Quick Sort:**

Partitioning:

→ choose leftmost as pivot (index l)

→ $i \leftarrow l+1;\ j \leftarrow h+1;$

→ loop

```
{
    inc i until A[i] ≥ pivot
    dec j until A[j] ≤ pivot
    if (i<j) swap(A[i],A[j]);
    else break;
}
```

→ $A[l] = A[j];\ A[j] = \text{pivot}$

$A[h+1]$ is set to $\infty$ and this useful when the pivot is the largest element. (This helps avoid inf loop)

→ no of comp in partitioning: $n+1$

$\therefore TC: O(n)$

QS Algo:

```
if(l<h)
{
    m ← partition(a,l,h);
    QS(a,l,m-1);
    QS(a,m+1,h);
}
```

TC: $\begin{cases} \text{best case: } 2T(n/2)+O(n) = O(n\log n) \\ \text{worst case (sorted): } T(n-1)+n = O(n^2) \end{cases}$

avg case: $O(n\log n)$

SC: $\begin{cases} \text{best case: } O(\log n) \\ \text{worst case: } O(n) \end{cases}$

**Long Integer Multiplication.**

Conventional approach: $O(n^2)$

D&C approach: $u, v$ be $n$ digit numbers

$m \leftarrow n/2;$

$u = w \times 10^m + x$ ; $w = u/10^m$, $x = u \% 10^m$

$v = y * 10^m + z$ ; $y = v/10^m$, $z = v \% 10^m$

$uv = (w \times 10^m + x)(y * 10^m + z)$

$T(n) = 4T(n/2) + \underbrace{bn}_{\text{addition}}$

$\Rightarrow T(n) = O(n^2)$

Scanned with CamScanner

## Anatolli Karatsuba's optimization:

let $t = (\omega+x)(y+z)$
$= \omega y + \omega z + xy + xz$
$\Rightarrow \omega z + xy = t - \omega y - xz$

now compute $P_1 = \omega y$; $P_2 = xz$; $P_3 = (\omega+x)(y+z)$

$uv = (P_1 \times 10^{2m}) + (P_3 - (P_1+P_2))10^m + P_2$

$\Rightarrow T(n) = 3T(n/2) + O(n)$
$T(n) = O(n^{\log_2 3}) = O(n^{1.58})$

## multi-way split for LIM:

D & C : $k^2 T(n/k) + O(n) = O(n^2)$
Anatolli : $(k^2-1)T(n/k) + O(n) = O(n^{\log_k (k^2-1)})$
Toom & cook : $(2k-1)T(n/k) + O(n) = O(n^{\log_k (2k-1)})$

## Master Theorem:

$T(n) = aT(n/b) + f(n)$ , $n > d$ ; $a \geq 1, b > 1$
$= c$ , $n \leq d$

$f(n)$ is +ve

Case(i): $f(n) = O(n^{\log_b a - \epsilon})$ , $\epsilon > 0$
$T(n) = \Theta(n^{\log_b a})$

Case(ii): $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$

$k \geq 0 \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
$k = -1 \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log\log n)$
$k < -1 \Rightarrow T(n) = \Theta(n^{\log_b a})$

Case(iii): $f(n) = \Omega(n^{\log_b a + \epsilon})$ , $\epsilon > 0$ and
$af(n/b) \leq \delta f(n)$ , for some $\delta < 1$
$T(n) = \Theta(f(n))$

→ Solve $T(n) = 2T(\sqrt{n}) + \log n$ (use transformation)
(Refer notes if any doubt).

→ $\frac{n}{\log n} \neq O(n^{1-\epsilon})$

→ Refer recursion tree method in notes.

Problems < decision / optimization

Problem defn: Constraints ; soln space
Constraints < implicit / explicit
soln space ↓ satisfying explicit constraint.

Feasible soln; objective func; optimal soln.
↳ doesn't exist for decision problems

# Greedy Method:

## fractional/real greedy knapsack:

given $p_i$ & $w_i$ find $x_i$ such that
$\Sigma x_i p_i$ is maximized subjected to $\Sigma x_i w_i \leq M$

→ Think its soln. TC: $O(n\log n)$ → including sort

## Job sequencing with Deadlines (JSD):

→ n-jobs; all arrive at 0; deadline $d_i$; profit $p_i$
→ soln : arrange profits in desc.
pick each job and schedule at max possible time.
TC: $O(n^2)$

## Optimal Merge Pattern:

At any point choose two records with least weight merge them and put them in list. Continue this untill all are merged.

→ If two files have n & m records then no of record movements $= n+m$
$= O(n+m)$

→ Total record movements
= weighted external path length
= sum of internal nodes

TC: < Sorted (unsorted array/list : $O(n^2)$ / Heap : $O(n\log n)$

SC : $O(n)$ (for any implementation)

## Huffman Coding: application of omp

→ non-uniform coding.
→ Build tree & assign prefix code.
→ avg no of bits needed = weighted external path length
= sum of internal nodes.
→ no of bits req ≤ no of bits in uniform encoding.

## Spanning Trees

→ Any graph algo with adj list has min TC of $O(n+e)$ and with adj. matrix has min TC of $O(n^2)$.

→ max no of spanning tree for complete graph = $n^{n-2}$

→ Refer notes for method of calculating no of spanning trees (pg: 8)

### Prims Algo:

(Prims actually starts choosing a single vertex)

→ obtain least cost edge <k, l>

→ Compute near values and set near(k) = near(l) = 0

→ for i:=2 to n-1

{
  get j such that near(j) ≠ 0 and cost(j, near(j)) is minimum.
  near(j) ← 0
  recompute near
}

TC:
- adj. matrix : $O(n^2)$ {sc: O(n)} (near)
- Heap : $O((n+e)\log n)$ (sc: O(e+n)) (heap, near)

### Kruskal's Algo:

→ Construct heap out of edge — O(e)

→ i ← 0

while i < n-1 and heap not empty — O(e)

{
  get min cost edge <u, v> — O(log e)
  j ← find(u); k ← find(v)
  if j ≠ k
  {
    i ← i+1
    Add edge to MCST
    union(j, k)
  } — O(1)
}

if i ≠ n-1 then print "no spanning tree"

TC: — using heap — $O(e \log e)$

SC: $O(n)$ — for parent array

### Dijkstra's Algo:

Randomly choose an edge and add it. If cycle is formed remove the largest edge of the cycle. Continue this until all the edges are added.

**Note:**

Cost of MCST generated by prims & kruskal is same. But trees generated by both algorithms may differ when there are multiple edges of same weight.

→ If all the edges cost are distinct then trees produce will also be same.

### Single Source Shortest Path: Dijkstra's Algorithm

→ Matrix method & spanning tree method (↳ shows path also)

→ Choose vertex closest so far and recompute distances. Do this process n-1 time.

→ Implementation is similar to prims

∴ TC:
- adj. matrix : $O(n^2)$ — SC: O(n)
- Heap : $O[(n+e)\log n]$ — SC: O(n+e)

**Note:**

→ Min cost edge of a cycle may or may not be in the MCST.

→ Max cost edge of a cycle never exists in MCST.

→ max cost edge is present in MCST iff it is a bridge.

| Problem | TC | SC |
|---|---|---|
| Max-Min | $O(n)$ | $O(\log n)$ |
| Merge Sort | $O(n \log n)$ | $O(n) + O(\log n)$ |
| Merge process | $O(n)$ | - |
| Binary search | $O(\log n)$ | - |
| Quick sort | $O(n \log n)$ (avg, best)<br>$O(n^2)$ | $O(\log n)$ (avg, best)<br>$O(n)$ |
| Strassen's | $O(n^{2.81})$ | $O(n^2)$ |
| real knapsack | $O(n \log n)$ including sort | $O(1)$ |
| JSD | $O(n^2)$ | - |
| OMP | $O(n^2)$ (list)<br>$O(n \log n)$ (Heap) | $O(n)$ |
| Huffman | $O(n^2)$ (list)<br>$O(n \log n)$ (Heap) | $O(n)$ |

| Problem | TC | SC |
|---|---|---|
| Prim's | $O(n^2)$<br>$O((n+e)\log n)$ | $O(n)$<br>$O(n+e)$ |
| Kruskal | $O(e\log e)$<br>$= O(e\log n)$ | $O(n+e)$ |
| Dijkstra's | $O(n^2)$<br>$O[(n+e)\log n]$ | $O(n)$<br>$O(n+e)$ |

# Dynamic Programming

DP approach
- Top Down (Memoization) (Recursive)
- Bottom Up (Non-recursive) → More efficient

## Multistage graph:

$COST(i,j) = \min\{COST(i+1,x) + c(j,x)\}$

$i$ → stage, $j$ → vertex

$COST(l-1,x) = c(x,t)$

$l$ → no of stages

$D(i,j) = x$ (used for path computation)

TC: $O(n^2)$ , $O(n+e)$
↓ adj. matrix, ↓ adj list

SC: $O(n)$
↳ dist values

## Travelling Salesman Problem:

$g(i,S) = \min\{c(i,x) + g(x, S-\{x\})\}$

$g(i,\phi) = c(i,v_0)$
↳ home city

$J(i,S) = x$

TC: $O(n^2 \cdot 2^n)$ . SC: $O(n \cdot 2^n)$

## Floyd Warshall's Algorithm (All pair Shortest)

$A^k(i,j)$ represent cost of shortest path from $i$ to $j$ with intermediate vertex not greater than $k$.

$A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}$

$A^0(i,j) = C(i,j)$

TC: $O(n^3)$ S: $O(n^2)$

→ This algo can be used to find transitive closure & reflexive transitive closure in $O(n^3)$

→ Floyd warshall's works with -ve weighted edge graph too.

## Bellman Ford (Single Source Shortest Path):

$d^l[x]$ represent cost of path from source $s$ to vertex $x$ with atmost '$l$' edges.

$d^l[x] = \min\{d^{l-1}[x], \min_{\forall k \in V}\{d^{l-1}[k] + c[k,x]\}\}$

$d^1[x] = C[s,x]$

TC:
- $O(n^3)$ : adj. matrix
- $O(ne)$ : adj list

| Algo | TC | SC |
|---|---|---|
| Multistage graph | $O(n^2)$<br>$O(n+e)$ | $O(n)$ |
| TSP | $O(n^2 \cdot 2^n)$ | $O(n \cdot 2^n)$ |
| Floyd warshall's | $O(n^3)$ | $O(n^2)$ |
| Bellman ford | $O(n^3)$<br>$O(n \cdot e)$ | |
| 0/1 knapsack | $O(n \cdot m)$<br>$O(n \cdot 2^n)$ brute | $O(n \cdot m)$ |
| LCS | $O(nm)$<br>$O(n \cdot 2^m)$ brute<br>$O(2^n \cdot m)$ brute | $O(nm)$ |
| Matrix Chain prod | $O(n^3)$ | $O(n^2)$ |
| OBST | $O(n^3)$ | |
| Realiable system | | |

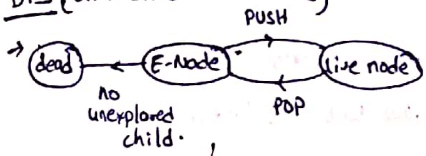
# Graph Techniques :

→ Tree traversal is unique, graph traversal need not be unique.

**Status of node :**

- Enode : node that is currently under exploration
- live node : nodes that are not fully explored (stack in DFS, Q in BFS store these)
- dead node : fully explored node.

**DFS (on undirected connected)**




Discovery time : time at which node is 1st visited

finishing time : time at which node becomes dead.

→ Before any node is explored, it must be popped.

→ A graph is connected

⟺ ~~at~~ finishing time of 1st node is highest

⟺ traversal finishes when 1st node becomes dead.

**DFS on undirected disconnected graph :**

→ ~~Here~~ depth first search spanning forest is formed.

→ no of connected components = no of spanning trees.

**DFS on directed graph :**

DFS on a directed graph produces 4 types of edges.

(i) Tree edge : part of depth first tree

(ii) Forward edge : parent to non-child descendant

(iii) Back~~word~~ edge : node to non-parent ancestor.
Self loop are back edges.

(iv) Cross edge : node to neither descendant nor ancester.
Cross edge may even be b/w vertices of different DFTs

**Paranthesis theorem :**

let $(u,v)$ be a directed edge.

$d(u) < d(v)\ f(v) > f(u) \Rightarrow$ tree edge | frwd edge

$d(v) < d(u)\ f(u) > f(v) \Rightarrow$ back edge

$[d(v) < f(v)] < [d(u) < f(u)] \Rightarrow$ Cross edge

**DFS on DAG :**

→ source vertex : no incoming edges
sink vertex : no outgoing edges.

~~→ Reverse of~~

→ Descending order of finishing time gives topological sort.

**Breadth First Search :**

**FIFO BFS :**

→ 1st node never ~~d~~ becomes live node.
i.e., 1st node never gets on the Q.

→ All live~~s~~ nodes are stored in the Q.
(trace parent so that drawing BFT will be easier)

→ A node is marked visited ~~on~~ as soon as it is pushed into Q.

→ BFS can be used to find shortest cycle containing given vertex.

**LIFO BFS (or) Dsearch :**

→ Here live nodes in the Q are made E-node in LIFO order.

→ can be implemented using stack.

**Priority Queue :**

→ pushing into Q is done normally.

→ But removing is done based on some criteria.

TC of BFS, DFS
- $O(n^2)$ : adj matrix
- $O(n+e)$ : adj list.

**Note :**

→ DFS and BFS can be used to determine presence of cycle (using backedge)

→ DFS and BFS can be used to determine connectivity of graph.

→ BFS is optimal algo for find shortest path in an undirected unweighted graph.

→ DFS is used to find components & articulation pts.
- connected comp
- strongly connected comp
- Biconnected comp.

**Connected Components**

→ Maximal subgraph that is connected is called connected component (applies only for undirected graphs)

## Strongly Connected Components (only for directed graph)

→ 2 vertices are strongly connected ⟺ there exists directed path from u to v & v to u.

Strongly connected comp: Maximal subgraph in which there is directed path ~~btwn~~ from any vertex to any vertex

Metagraph: subgraph with all the vertices

Note:

→ Every directed graph is DAG of it ~~con~~ strongly connected components.

→ $C_1$ & $C_2$ be two strongly conneted components such that there is an edge from vertex in $C_1$ to vertex in $C_2$. then finishing time of $C_1$ will be greater than finishing time of any vertex in $C_2$.

If there edge from $C_1$ to $C_2$ then there will be no edge from $C_2$ to $C_1$.

## Articulation Point & Biconnected Components

→ Graph without AP is called biconnected.

→ Biconnected component: Maximal subgraph that is biconnected

## SORTING TECHNIQUES

→ stable vs unstable.

→ internal vs external

→ inplace vs not inplace

→ TC of comparision based sort $= O(\max\{\text{comp}, \text{swaps}\})$

### Bubble sort:

no of comp $= (n-1)+(n-2)+\dots+(1) = \frac{n(n-1)}{2} = O(n^2)$

no of swaps = no of inversions

### Selection Sort:

In each pass $i$, select $i^{th}$ smallest and place it in the correct position.

→ no of comp $= \frac{n(n-1)}{2}$

→ no of swap $= n-1$

### Insertion sort:

→ Take an element (starting one) from unsorted list and place in its correct position in ~~unsort~~ sorted list.

i.e. In each pass $i$, list will be sorted till index $i$.

→ Insertion acts like external sorting but it is not.

TC: Best case: $O(n)$

worst case: $O(n^2)$

→ If $d$ is no of inversions,

TC of insertion sort $= O(n+d)$

### Non-Comparision Based Sorting

Radix sort:

→ If base is $b$, take $b$ buckets.

→ from LSB to MSB

for each digit, distribute into buckets an reorder.

TC: $O(d(n+b)) = O(nd)$ ($\because b$ is const.)

$d \to$ max no of digits

$b \to$ base

→ If $x$ is maximum number, then

$TC = O(n \log_b x)$

if $x = O(n^c)$

$\Rightarrow TC = O(n \log n)$

$\therefore$ Radix sort is good when $b \geq n$.

$\Rightarrow TC = O(n)$

→ Radix sort need special implementation for sorting negative numbers.

→ sorting fractional numbers is not possible.