GATE CSE NOTES

by

Joyoshish Saha

Graph Algorithms.

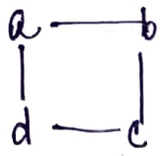
1. BFS/BFT, DFS/DFT
2. Graph Cycle
   - Detect cycle
   - Union-Find Algorithm

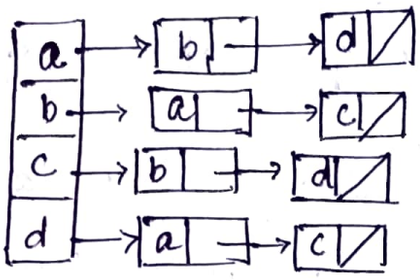
   | Finding connected components using Disjoint Set DS
3. Topological Sorting
4. MST (Prim's, Kruskal's, Boruvka's)
5. Backtracking (m-coloring problem, Hamiltonian cycle)
6. Shortest Path Algo (Dijkstra's (SSSP), Bellman-Ford, Floyd Warshall (APSP))
7. Connectivity (Connected components, Eulerian path-circuit)
8. Maximum flow problem.

Graphs

* Representation:

1. Adjacency matrix $O(n^2)$ or $O(V^2)$
2. Linked list (Adjacency list).





Undirected graph

$O(V+E)$

Dense

$E = O(V^2)$

↓

Adj. matrix.

Sparse

$E = O(V)$

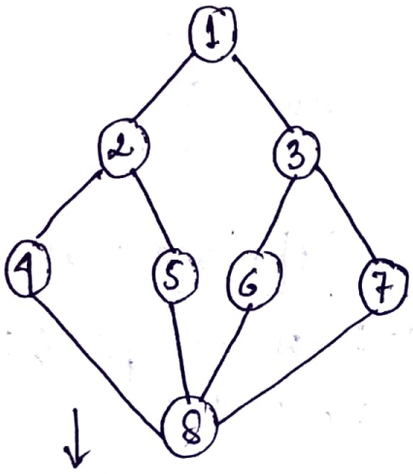
↓

Adjacency list.

কম সংখ্যক edges থাকলে adj. list

বেশি সংখ্যক edges থাকলে adj. matrix

* Graph traversal/search



BFS Queue | DFS Stack

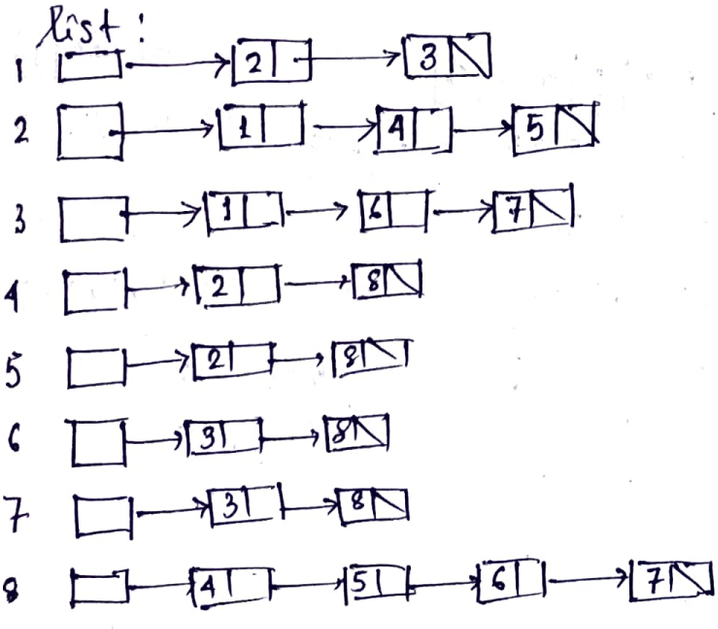
Space $O(n)$ | space $O(n)$

BFS : 1 2 3 4 5 6 7 8

DFS : 1 2 4 8 5 6 3 7

↓

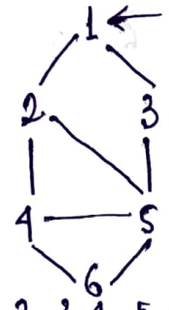
Adj. list:



* BFS.

```
Algorithm BFS(v)
// visited[] array global, initialised to zeros
{
    u := v ;
    visited[v] := 1 ;
    // q is a queue of unexplored vertices
    repeat {
        for all vertices w adjacent from u do {
            if (visited[w] = 0) then {
                Add w to q ; // w unexplored
                visited[w] := 1 ;
            }
        }
        if q is empty then return ; // no unexplored vertex
        Delete next elem., u, from q ; // get 1st unexp. vertex
    } until (false) ;
}
```

eg.



Visited array V[] Queue Q

1 2 3 4 5 6

V | 0 | 0 | 0 | 0 | 0 | 0 |

Q :

① → ② V | 1 | 0 | 0 | 0 | 0 | 0 |

u : 1 | w : {2, 3}

o/p : 1

② → ③ V | 1 | 1 | 0 | 0 | 0 | 0 |

u = 1 w : {2, 3} Q : | 2 |

o/p : 1, 2

④ V | 1 | 1 | 1 | 0 | 0 | 0 |

u = 1 w : {2, 3} Q : | 2 | 3 |

o/p : 1, 2, 3

⑤ V | 1 | 1 | 1 | 0 | 0 | 0 |

→ u = 2 w : {1, 5, 4}

Q : | 3 |

o/p : 1, 2, 3

⑥ V | 1 | 1 | 1 | 0 | 1 | 0 |

u = 2 w : {1, 5, 4}

Q : | 3 | 5 |

o/p : 1, 2, 3, 5

⑦ V | 1 | 1 | 1 | 1 | 1 | 0 |

u = 2 w : {1, 5, 4} →

Q | 3 | 5 | 4 |

o/p : 1, 2, 3, 5, 4

⑧ V | 1 | 1 | 1 | 1 | 1 | 0 |

u = 3 w : {1, 5}

Q | 5 | 4 |

(9) V | 1 | 1 | 1 | 1 | 1 | 0 |

u = 5 W : {3, 2, 4, 6}

Q | 4 |

O/p 1, 2, 3, 5, 4

(10) V | 1 | 1 | 1 | 1 | 1 | 1 |

u = 5 W : {3, 2, 4, 6}

Q | 4 | 6 |

O/p 1, 2, 3, 5, 4, 6

(11) V | 1 | 1 | 1 | 1 | 1 | 1 |

u = 4 W : {2, 5, 6}

Q | 6 |

O/p 1, 2, 3, 5, 4, 6

(12) V | 1 | 1 | 1 | 1 | 1 | 1 |

u = 6 W : {4, 5}

Q

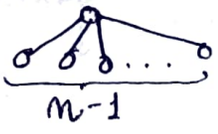
O/p | 1, 2, 3, 5, 4, 6 |

Analysis in case of adj. list rep^n :

Space complexity : $O(n)$ for visited array

$O(n-1)$ for queue (Worst case)

$\Rightarrow O(n)$ or $O(V)$



Time complexity : $O(E+V)$

E → exploration

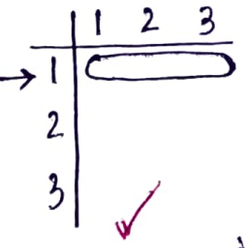
V → initialisation

As adj. list is used, all vertices adjacent to u can be determined in time $d(u)$, $d(u)$ being the degree of u. Since, each vertex can be explored at most once, total time for exploration $O(\Sigma d(u)) = O(2E) = O(E)$. To init. visited[i] it takes $O(V)$ time. Total time = $O(V+E)$.

1. Each node visited at most once.
2. For each visited node, # adjacent vertices $\approx O(d(u))$
We explore each adjacent node.
Hence, for exploration, $O(\Sigma d(u))$

BFS analysis in case of adjacency matrix:

Space complexity: $O(V)$, analysis same as before.



Time complexity: It takes $\theta(V)$ time to determine vertices adj. to u 1 time in total is $O(n^2)$ or $O(V^2)$. Initialisation - $O(V)$

So, TC is $O(V^2)$.

→ We can check graph connectivity using BFS. Starting with a vertex u, if at the end of algo. visited array becomes all 1 then connected.

→ Search techniques are used for connected graphs, as it may not explore all nodes if the graph is not connected. (Search - visit nodes that are reachable)

But, when the search necessarily involves the examination of every vertex in the object being searched, it is called a traversal.

Breadth First Traversal :

```
Algorithm BFT (G, n) {
    for i := 1 to n do
        visited [i] := 0;
    for i := 1 to n do
        if (! visited [i]) then BFS (i);
}
```

BFS is called for each component of graph once.

	Adj. list	Adj. Matrix
SC	$O(V)$	$O(V)$
TC	$O(V+E)$	$O(V^2)$

- Pseudocode BFS.

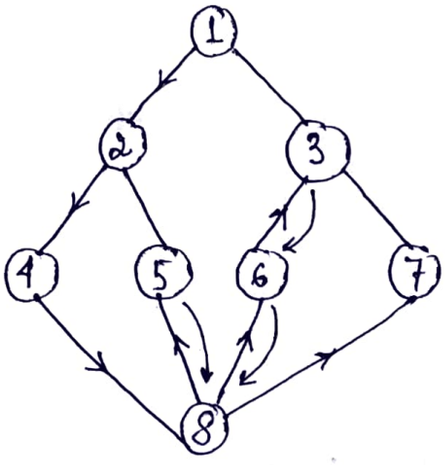
Set all nodes to "not visited"

```
Void BFS () {
    int v;
    for (v=0; v<n; v++)
        visited [v] = 0
    printf ("Enter start vertex: ");
    scanf ("%d", &v);
    int i;
    push_queue (v);
    while (! is_Empty_Queue ()) {
        v = pop_queue ();
        if (visited [v])
            continue;
        printf ("%d ", v);
        visited [v] = 1;
        for (i=0; i<n; i++) {
            if (adj [v][i] == 1 && visited [i] == 0)
                push_queue (i);
        }
        printf ("\n");
}
```

* DFS.

```
Algorithm DFS (v)
// array visited [] set to 0
{
    visited [v] := 1;
    for each vertex w adjacent to v do
    {
        if (visited [w] = 0) then DFS (w);
    }
}
```

- **Example :**



visited

0	0	0	0	0	0	0	0

Stack

v = 1	v = 2	v = 4	v = 8	v = 5	v = 6	v = 3	v = 7
w = {2,3}	w = {1,4,5}	w = {2,8}	w = {1,5,6,7}	w = {2,8}	w = {3,8}	w = {1,6,7}	w = {3,8}
				X		X	X

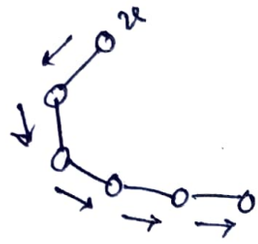
O/p : 1, 2, 4, 8, 5, 6, 3, 7.

- **Code :**

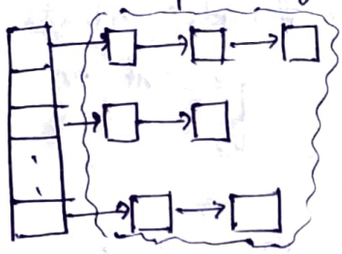
```
void DFS ( struct Graph* graph, int vertex) {
    struct node * ptr = graph → adjLists [vertex] ;
    graph → visited [vertex] = 1;
    printf (" %d ", vertex);
    while (ptr) {
        int adj-vertex = ptr → vertex;
        if ( graph → visited [ adj-vertex] == 0)
                DFS ( graph, adj-vertex);
        ptr = ptr → next ;
    }
}
```

- **Analysis of DFS.**

SC: O(V) for visited array. Also, in the worst case (a chain graph) the stack can grow upto O(V). So, SC is O(V).



TC: In case of adj. lists.



= 2E

We call DFS for each node in the list. for a node all adjacent vertices are visited once.

So, for that O(2E) = O(E)

Also, the initialization : O(V)

⇒ O(V+E)

In case of adj. matrix : $O(V^2)$.

- **DFT :**

```
Algorithm DFT {
    for i := 1 to n do
        visited [i] := 0 ;
    for i := 1 to n do
        if (! visited [i]) then DFS (i) ;
}
```

SC : O(V)

TC : O(V+E) (list) O(V²) (matr.)

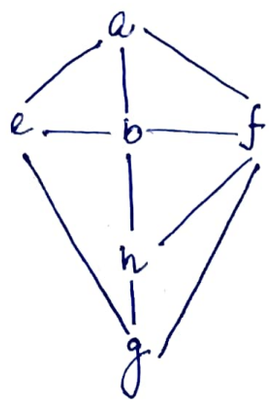
Q's Sequences possible if run DFS

i) a b e g h f

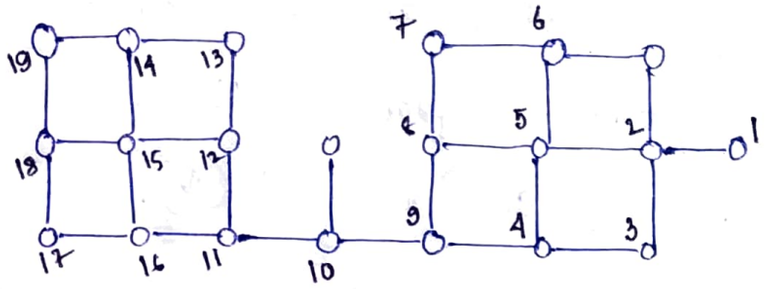
ii) a b f e h g

iii) a b f h g e

iv) a f g h b e



Q'14 Deepest stack possible (longest path) for this (np-hard)



(use intuition to find longest path starting from any node)

Ans: 19 max # nodes that can be present in the stack

Q'06 IT → DFT

Consider a DFS of an undirected graph with 3 vertices P, Q, R. Let discovery time d(u) represent the time instant when the vertex u is first visited, & finish time f(u) represent the time instance when the vertex u is last visited. Given that:

d(P) = 5	d(R) = 14	f(Q) = 10
d(Q) = 6	f(P) = 12	f(R) = 18

What kind of graph this is.?

→ P first → Q first → Q last → P last → R first → R last (DFT).

not DFS

There are 2 connected components & P & Q are connected.

* If a graph ~~(has no loop)~~ and all edges are unweighted or of the same weight, then we can use BFS to find shortest path.

We use modified version of BFS in which we keep storing the predecessor while performing the algo.

* Using BFS, we can record levels
* Using BFS/DFS, we can find connected components.
* BFS tree : Edges explored by BFS form a tree.
  Acyclic graph = connected, with n-1 edges
  (Non-BFS tree edges form cycle)
* DFS tree : Non-tree edges form cycle.
* Directed cycles : Tree edge, Forward edge, Back edge, Cross edge

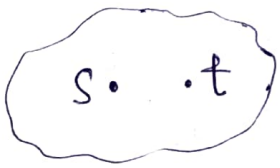
A directed graph has cycle only if its DFS reveals a back edge.

BFS - DFS Appln's

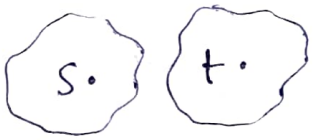
* BFS applications.

1. Finding set of connected components.

- Maximal connected subgraph.



case I



case II

Applications.

i) Network connectivity

ii) Clustering

- For any 2 nodes s & t in G, their connected components are either identical or disjoint.

- BFS of node i discovers the connected component containing i.

```
connComp = 0
Discovered[i] = false  ∀ i : 1 to n

for i = 1 to n
    if Discovered[i] = false
        connComp ++;
        BFS(i);
```

O(V+E)

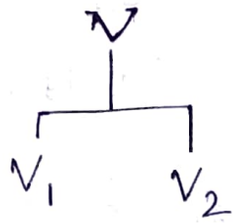
initialisation ↑ edge processing ↑

- One edge can belong to only one connected component.

2. The path found by the BFS to any node is the shortest path to that node from a single source (smallest # edges in unweighted graph).

3. Testing Bipartiteness.

$G = (V, E)$



$\forall e \in E$ s.t. $u \in X$, $v \in Y$

$e = (u, v)$

- Odd length cycles are not bipartite.

1, 2, 3, ..., 2K, 2K+1

R B R B R

⇒ Graph containing odd len cycle is not bipartite.

- Absence of odd cycles proves bipartiteness (can be proved).

- Procedure (assume G is connected, if not check for every component)

Pick any node s - color it red.

neighbor(s) - blue

neighbor(neighbor(s)) - red

⋮

until all nodes are colored.

O(V+E)

Does every edge has ends of opp. colors?

Yes ↙ Bipartite

No ↘ Not bipartite

Bipartite Graph Test using DFS:

```
bool dfs (int v, int c) {
    visited [v] = 1;
    color [v] = c;
    for (int child : ar[v]) {
        if (visited [child] == 0) {
            if (dfs (child, c ^ 1) == false)
                return false;
        }
        else {
            if (color[v] == color [child])
                return false;
        }
    }
    return true;
}
```

(c ^ 1 : XOR)

---

4. Finding the shortest path in a graph with weights 0 or 1 (0-1 BFS).

(In each step of BFS, we check the optimal distance condition. Here, we use doubly ended queue to store the node. If edge weight is 0, then push it at the front, otherwise at the rear.)

5. Finding shortest cycle in a directed unweighted graph (as soon as we go from current vertex to the source vertex, we have found a cycle containing source vertex. Find all such cycles, choose shortest.)

- BFS tree.

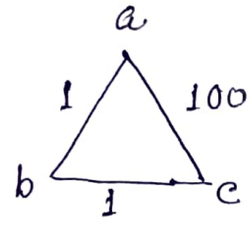
Tree formed by BFS.

- Let x & y be nodes in Layer$_i$ & Layer$_j$ of the BFS tree, & let (x,y) be an edge of G. Then, i & j differ by at most 1 (& nodes in adjacent layers).

- BFS tree gives the shortest path from source to any node (for unweighted graphs).

⇒ Layer L_j is the set of all nodes at distance exactly j from s.

⇒ For weighted graphs, doesn't give shortest path.



6. Finding diameter of graph.

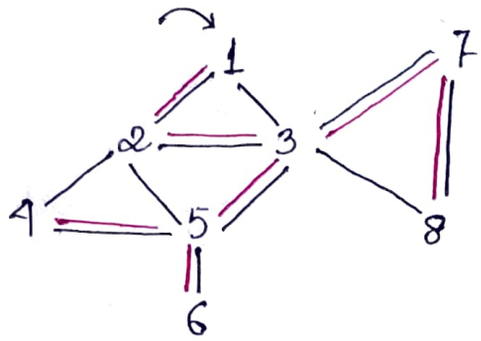
(Max. distance b/w 2 vertices in graph)

- Perform BFS on all nodes.
- Output the max # levels in any BFS tree.

* DFS applications.

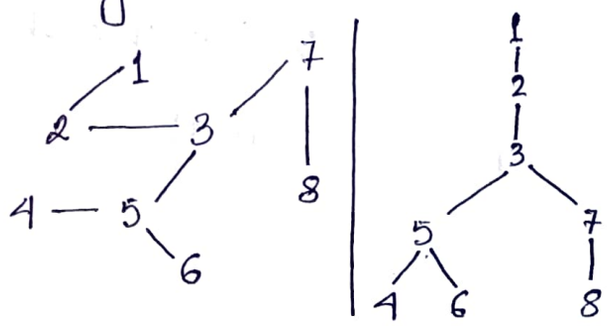
- Maintain 2 timestamps for each node while processing DFS : 'discovery' & 'processing done'. Helps to know how we visited the nodes, what was the order.

	discovery	proc. done
1	0	15
2	1	14
3	2	13
4	4	5
5	3	8
6	6	7
7	9	12
8	10	11



- DFS-tree not unique.

- n-1 edges in DFS-tree



- discovery time (ancestor) < discovery (descendant) < finish time (desc.) < finish time (anc.)

Used in top-sort. (DAG)

if $(u, v) \in E$,

$u < v \quad \forall e \in E$.

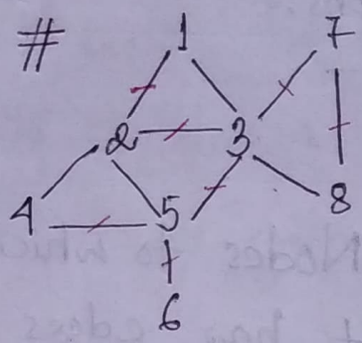
i) Perform DFS.

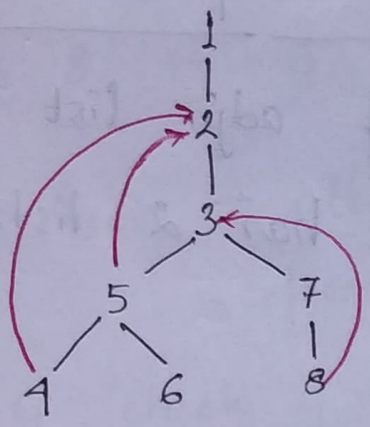
ii) Arrange nodes in decreasing order of finish times.

For above graph, 1, 2, 3, 7, 8, 5, 6, 4.

DFS TREE

#

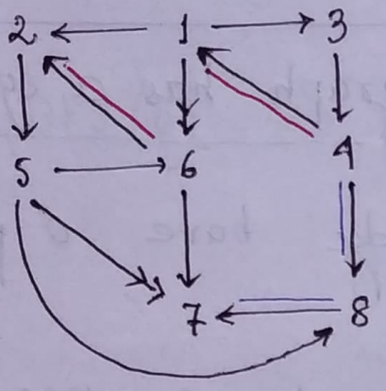


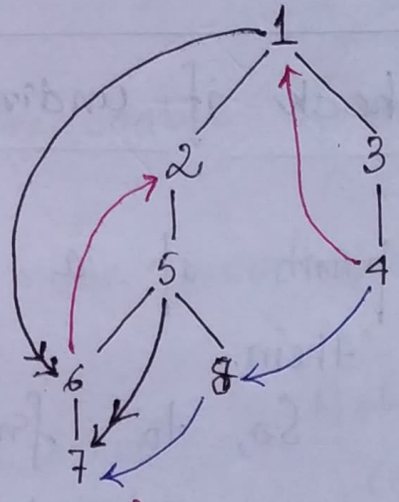


Back edges

Undirected – only tree & back edges

#





Directed graph →

→ Tree edges

→ Back edges (form cycle)

→ Cross edges →→ Forward edges

↓

always from right to left (?)

Tree edge (u,v) discover(u) < disc(v) ... < finish(v) < fin(u)

fwd edge (u,v) disc(u) < disc(v) < fin(v) < fin(u)

Back edge (u,v) disc(u) > disc(v) and fin(u) < fin(v)

Cross edge (u,v) disc(v) < fin(v) < ... < disc(u) < fin(u)

1. Application 1 : Connected components

```
for i = 1 to n
    if discovered[i] = false
        DFS(i)
```

* Representation of directed graphs.

Variant of adj. list

Each node has 2 lists : a) Nodes to which it has edges

b) Nodes from which it has edges

---

2. Check if undirected graph has a cycle.

End points of a back edge have a path b/w them.

So, to find a cycle, we need to detect back edge.

Back edge ⇒ cycle

No Back edge ⇒ No cycle

3. Check if directed graph has a cycle

By checking if there's back edge.

4. Given a directed graph, and a node s find set of nodes having path to s.

→ $G_{rev} = (V, E_{rev})$

E_{rev} with dir^{n}s reversed

A node has a path from s in G_{rev} iff it has a path to s in G.

⇒ DFS(s) in G_{rev} → all nodes reachable from s in G_{rev}

5. Given a directed G, check if it is strongly connected (2 nodes mutually reachable)

DFS(u) → nodes reachable from u

↓

Is every node reachable from u?

↓ Yes

u has path to every other vertex

↓

repeat for all vertices

↘ No

Not strongly connected

If DFS performed on every node yields the vertex set V,

⇒ each node is reachable from every other node

⇒ any 2 nodes are mutually reachable.

So, strongly connected.

TC $O(V(V+E))$

$\rightarrow O(VE)$

O(V+E) algorithm

DFS(u) : all nodes reachable from u?

↙ Yes ↘ No ⇒ no!

$G_{rev} = (V, E_{rev})$

↓

DFS(u)

all nodes reachable from u? → No → No!

↓ Yes

all nodes have a path to u in G.

$u \rightsquigarrow s$ $s \rightsquigarrow u$. $\forall s \in V$

for $v_1, v_2 \in V$, u & v_1, u & v_2 are mutually reachable.

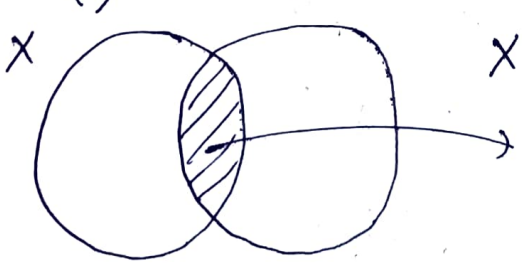
⇒ v_1 & v_2 are mutually reachable.

So, strongly connected!

6. Strongly connected components in directed graphs:

Strong component containing node s

DFS(s) in G DFS(s) in G_{rev}



nodes mutually reachable with s,

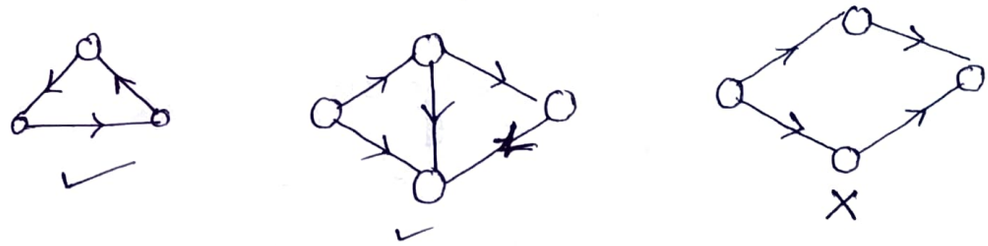
$X \cap X'$

⇓

strong component of s

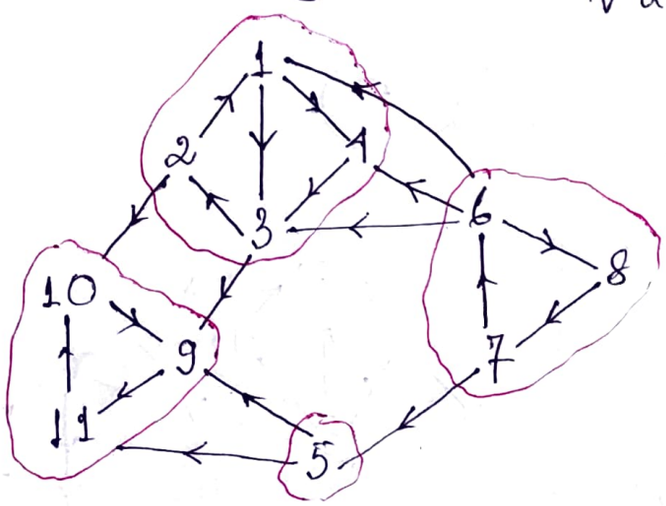
- For any 2 nodes s & t in a directed graph, their strong components are either identical or disjoint.

- Strongly connected example.



- Strong components example

$\forall u, v \; \exists \; u \rightsquigarrow v, \; v \rightsquigarrow u$ | maximal components.



(10, 9, 11)
(1, 4, 3, 2)
(5)
(7, 6, 8)

7. Finding strong components. (2)

Start DFS from a node in a component that has no out-edges.

→ Strong components of G & G_{rev} would be the same.

→

in G_{rev} $C_i \rightarrow C_j$

$\Downarrow$

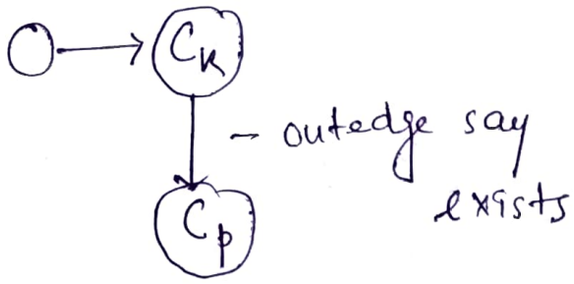
$fin(c_i) > fin(c_j)$

C – component

but in G, $C_j \rightarrow C_i$

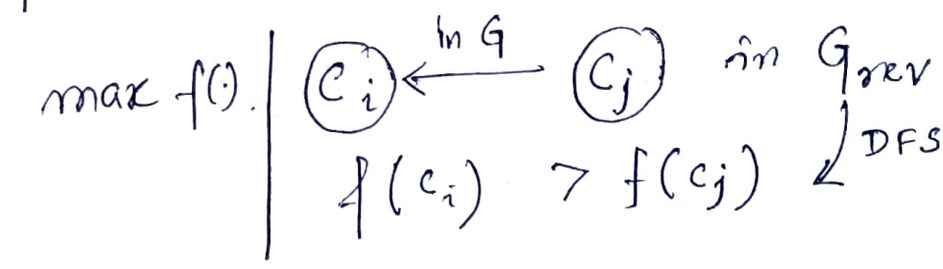
The maximum finish (·) would be for a sink strong component (that has no out-edges).

Say, C_K has the max f(·).



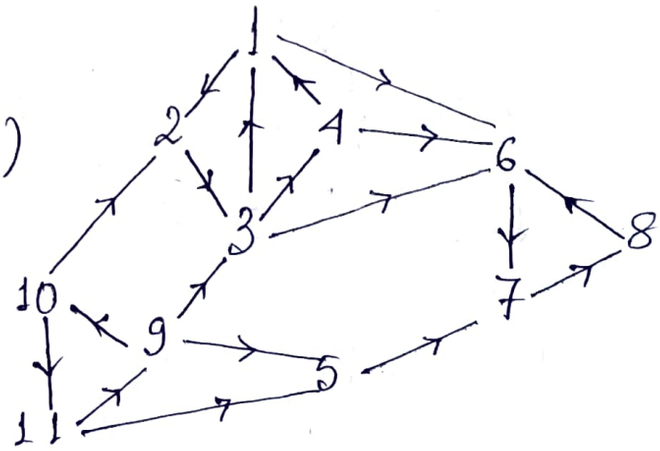
$f(C_p) > f(C_K)$. contradiction!

$\Rightarrow$ So, no out-edges.



Example (graph on the back 5)

↳ G_{rev} of G -



Start DFS on any node (in G_{rev})

DFS(1)

1 → 2 → 3 → 4 → 6 → 7 → 8

0/13 1/12 2/11 3/10 4/9 5/8 6/7

DFS(10)

10 → 11 → 9 → 5

14/21 15/20 16/19 17/18

Finish time in dec. order.

10	11	9	5	1	2	3	4	6	7	8
1	2	3	4	5	6	7	8	9	10	11

node no.

Now take node 10 & DFS(10) in G.

(10 → 9 → 11) this is a strong component.

Ignore 11, 9 from array now.

Pick, 5 · DFS(5) in G. (5) strong comp. ignore 5 now.

DFS(1)
(1, 4, 3, 2)
DFS(6) (6, 8, 7)

Algo (Strong Component)

1. Let $G_{rev} = G$ with directions of edges reversed.

2. Run DFS - loop on G_{rev} (determine finish time of all nodes).

3. Run DFS - loop on G in decreasing order of finish times determined in (2).

4. Output the nodes in each tree in the DFS forest obtained in (3) as a separate strongly connected component.

8. Finding lexicographical first path in the graph from source to all vertices.

9. Check if a vertex in a tree is an ancestor of some other vertex.

v_i is an ancestor of v_j iff

discovery (i) < discovery (j)

finish (i) > finish (j)

v_i
↓
v_j

10. Finding lowest common ancestor (LCA) of 2 vertices.

11. Finding bridges in an undirected graph:

First, convert the given graph into a directed graph by running a series of DFS & making each edge directed as we go through it, in the direction we went. Second, find the strong components in the directed graph. Bridges are the edges whose ends belong to different strong components.

Shortest Path Algorithms.

* In a shortest path problem, we are given weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights. The weight of path $p = \langle v_0, v_2, \ldots, v_k \rangle$ is the sum of the weighted of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the shortest-path weight from u to v by

$$\delta(u,v) = \begin{cases} \min \{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there's path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u,v)$.

* Shortest path algorithms typically rely on the property that a shortest path between 2 vertices contains other shortest paths within it.

* Types of shortest path problem.

i) Single source shortest path problem:

Shortest path from a given a source vertex to all other remaining vertices is computed. Dijkstra's algorithm (Greedy approach) & Bellman Ford algorithm (Dynamic programming) are the famous algorithms used for solving single-source shortest path problem. Many other problems can be solved by the algorithm for the single-source problem, including other 3 variants of shortest path problem.

ii) Single-pair shortest path problem.

Shortest path between a given pair of vertices is computed. A* search algorithm is a famous algorithm used for solving this. No algorithms for this problem are known that run asymptotically faster than the best single source algorithms in the worst case.

iii) Single-destination shortest path problem

Shortest path from all the vertices to a single destination vertex. By reversing the direction of the edges in the graph, we can reduce this problem to a single-source problem.

iv) All pairs shortest path problem.

Shortest path for every pair of vertices u & v. Floyd-Warshall algorithm & Johnson's algorithm are famous for solving this problem.

→ Optimal substructure of a shortest path:

Subpaths of shortest paths are shortest paths.

* Negative Weight Edges:

If the graph $G(V, E)$ contains no negative weight cycles reachable from the source s, then for all $v \in V$, the shortest path weight $\delta(s, v)$ remains well-defined, even if it has a negative valued weight.

If the graph contains a negative-weight cycle reachable from s, however shortest-path weights are not well-defined. No path from s to a vertex on the cycle can be a shortest path; we can always find a path with lower weight by following the proposed shortest path & then traversing the negative-weight cycle. If there is a negative-weight cycle on some path from s to v, we define $\delta(s,v) = -\infty$. (CLRS 582)

NB 1. Dijkstra's algorithm assumes that all edges' weights in the input graph are nonnegative.

*2. Bellman-Ford algo. allows ~~non~~-negative weight edges in the input graph & produce a correct answer as long as no negative-weight cycles are reachable from the source. If there's such a negative weight cycle, the algorithm can detect & report its existence.

* Relaxation.

Dijkstra's & Bellman-Ford use the technique relaxation.

For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a shortest path estimate. We initialize the shortest path estimates & predecessors by the following $\Theta(V)$ time procedure —

Initialize_single_source (G, s) $\Theta(V)$

```
for each vertex v ∈ V[G]
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0.
```

$\pi[v]$ - predecessor of node v if followed the best path

After initialization $\pi[v] = NIL$ for all $v \in V$, $d[s] = 0$ & $d[v] = \infty$ for $v \in V - \{s\}$.

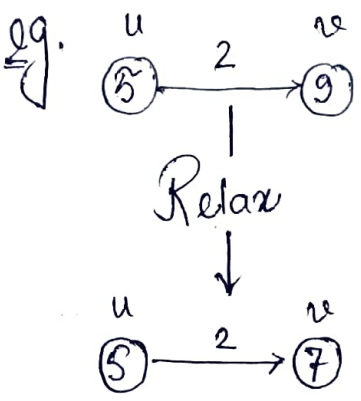
The process of relaxing an edge (u,v) consists of testing whether we can improve the shortest path to v found so far by going through u & , if so, updating $d[v]$ and $\pi[v]$.

Following performs a relaxation step on edge (u,v) in $O(1)$ time.

Relax (u,v,w) $O(1)$

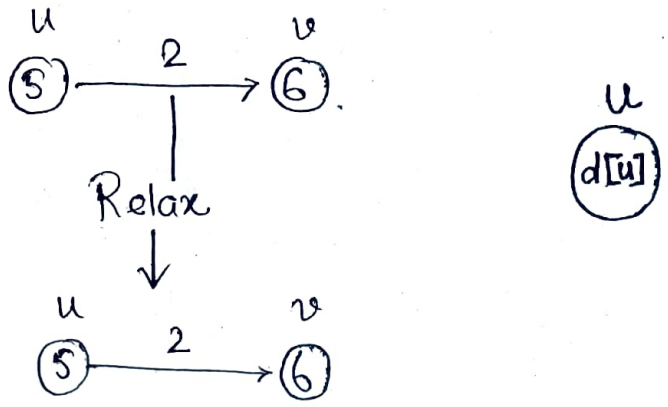
```
if d[v] > d[u] + w(u,v)
   then d[v] ← d[u] + w(u,v)
        π[v] ← u.        // setting predecessor of
                         // v to u.
```

Eg.



$d[v] > d[u] + w(u,v)$

$d[v]$ value decreases



$d[v] \leq d[u] + w(u,v)$

* Both D's & BF algos call Initialize_single-source & repeatedly relaxes edges. Moreover, relaxation is the only means by which shortest path estimates d[v] and predecessors π[v] change. In D's algo, each edge is relaxed only (exactly) once. In BF algo, each edge is relaxed many times.

* Dijkstra's Algorithm. - Greedy (GV)

Solves the single source shortest path problem on a weighted, directed graph G(V,E) for the case in which all edge weights are nonnegative. We assume $w(u,v) \geq 0$ for each edge $(u,v) \in E$.

Dijkstra's algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V-S$ or Q with the minimum shortest path estimate, adds u to S, and relaxes all edges leaving u. We use a min-priority queue Q of vertices, keyed by their d values.

Dijkstra's algorithm is like Prim's algo in that both algos use a min-priority queue to find the lightest vertex outside a given set (the set S in D's algo & the tree being grown in Prim's).

Dijkstra (G, w, s)

TC:
avg $O(|E| \log |V|)$
worst $O(|V|^2)$

SC:
worst $O(|V|+|E|)$

```
1.  Initialize_single_source (G, s)
2.  S ← ϕ
3.  Q ← V[G]
4.  while Q ≠ ϕ
5.      do u ← Extract_min (Q)
6.         S ← S ∪ {u}
7.         for each vertex v ∈ Adj [u]
8.             do Relax (u, v, w).
```

Line 3 initializes the min-priority queue Q to contain all the vertices in V.

Line 5 extracts a vertex u from $Q = V - S$ & line 6 adds it to set S. Lines 7-8 relax each edge (u,v) leaving u, thus updating the estimate $d[v]$ & the predecessor $\pi[v]$ if the shortest path to v can be improved by going through u.

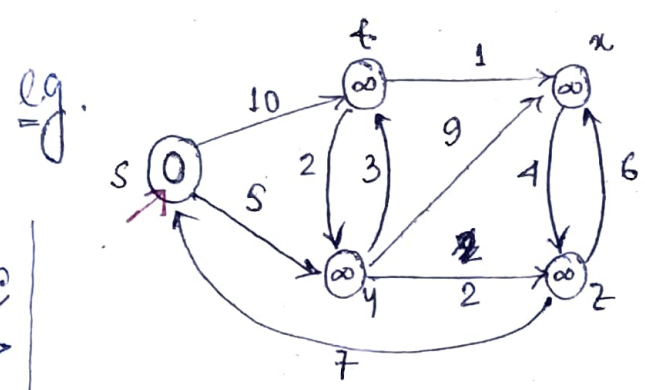
Because Dijkstra's algorithm always chooses the closest vertex, in $V-S$ to add to set S, we say that it uses a greedy strategy. Greedy strategies don't always yield optimal results in general, but Dijkstra's algorithm does indeed compute shortest paths. Each time a vertex u is added to set S, we have $d[u] = \delta(s,u)$.

- Keys in min-heap: nodes in $V-S$

  for $v \in V-S$, key [v] = smallest value of greedy parameters $(d_u + l(u,v))$ of an edge (u,v) s.t. $u \in S$; if no such edge then $d'(v) = +\infty$

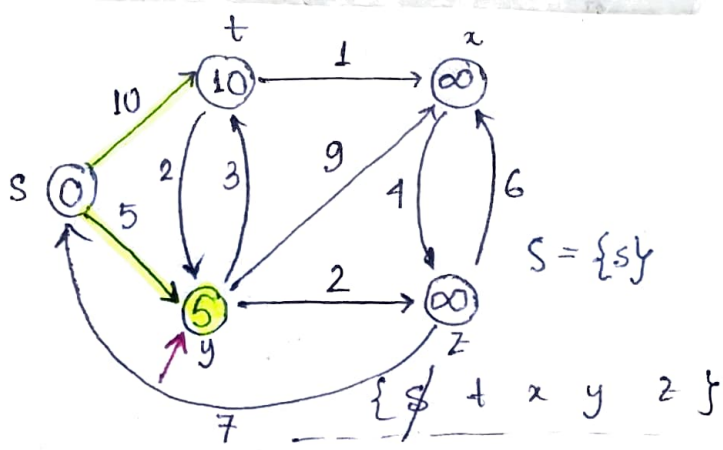
  $d'(\cdot)$

grow the set S by adding $\min(d_u + l(u,v))$ $u \in S, v \in V \setminus S$



(s,t)
(s,y)

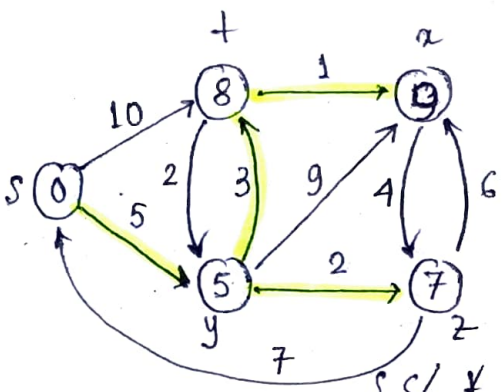
$S = \{s\ t\ x\ y\ z\}$ $S = \phi$



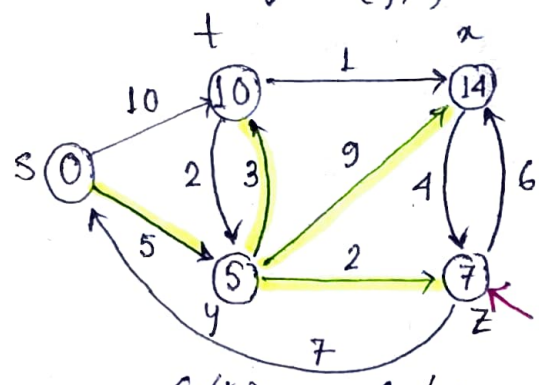
$S = \{s\}$

$\{s\ t\ x\ y\ z\}$

(y,t) (y,z)
(y,x)



$\{s, t\ x\ y\ z\}$

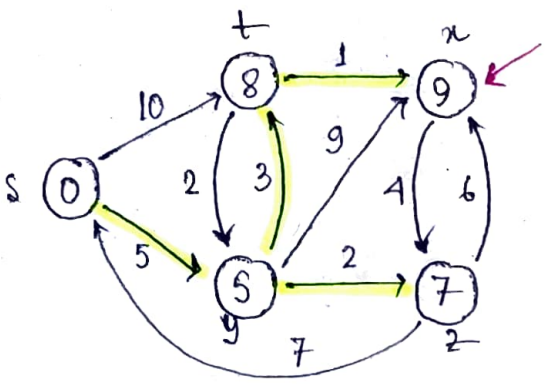


$S = \{s, y\}$

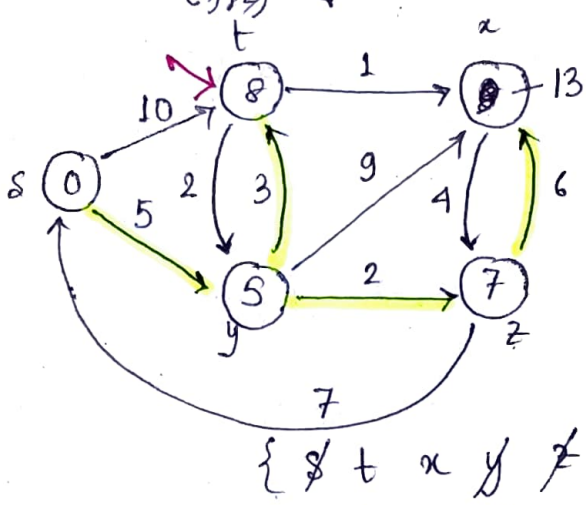
$\{s\ t\ x\ y\ z\}$

$S = \{s,y,z,t,x\}$

(z,x)



(t,x).



$\{s\ t\ x\ y\ z\}$

$S = \{s,y,z,t\}$

$\{s\ t\ x\ y\ z\}$

$S = \{s,y,z\}$

Analysis. (how min-PQ is implemented)

Decrease_key is implicit in Relax. Insert is invoked once per vertex, as in Extract_min. Because, each vertex $v \in V$ is added to set S exactly once, each edge in the adjacency list adj[v] is examined in the for loop of lines 7-8 exactly once during the course of the algo. Since, the total number of edges in all the adjacency lists is $|E|$, there are a total of $|E|$ iterations of this for loop, & thus a total of at most $|E|$ Decrease_key operations.

1. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$. We simply store $d[v]$ in the vth entry of an array.

a) Each Insert & Decrease_key takes $O(1)$.

b) Each Extract_min takes $O(V)$ time.

c) Total time $O(V^2 + E) = O(V^2)$.

2. If we use Binary heap to maintain min-priority queue,

a) Each Extract_min takes $O(\log V)$.

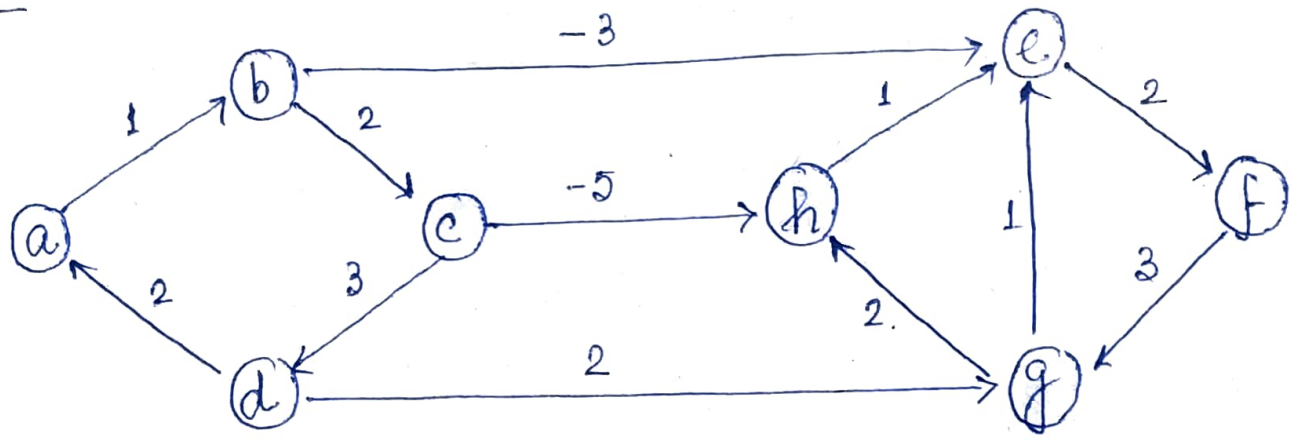
There are $|V|$ such operations. Time to build the binary min-heap is $O(V)$.

b) Each Decrease_key operation takes $O(\log V)$ time & there are still at most $|E|$ such operations.

c) Total running time is $O(V \log V + E \log V)$ which is $O(E \log V)$ if all vertices are reachable from the source.

3. If we use Fibonacci heap, Extract_min op^n takes $O(\log V)$ & Decrease_key $O(1)$, total TC will be $O(V \log V + E)$.

S. G'08.



Dijkstra's single source shortest path algorithm when run from vertex a, computes the correct shortest path distance to

a) Only vertex a

b) Only vertices a, e, f, g, h

c) Only vertices a, b, c, d

✓ d) All the vertices.

→ Just by simulating D's, it works for this graph though D's is not guaranteed to work for graphs with -ve weight edges.

- D's works only for connected graphs.
  D's works for directed as well as undirected graphs.

* Bellman-Ford Algorithm.

Solves the single-source shortest-path problem in the general case in which edge weights may be negative.

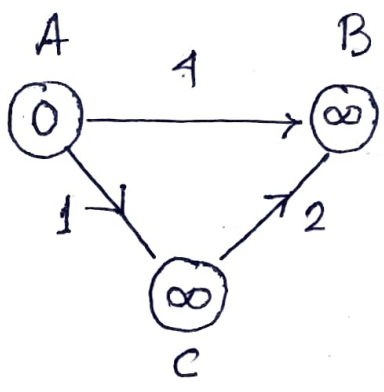
Given a weighted, directed graph $G(V, E)$ with source s and weight function $w : E \rightarrow \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no sol^n exists. If there is no such cycle, the algorithm produces the shortest paths & their weights.

Time complexity of Dijkstra's algo is better than BFA.

- Given n nodes in a graph, the shortest path between 2 nodes will have at most n-1 edges. This is why, in BFA, we relax the edges in the graph n-1 times. Relaxing edges n-1 times implies we are finding the shortest path ~~length with~~ weights when the path length is at most n-1. After n-1 times relaxation, we relax the edges

one more time to check if there is any -ve edge cycle. If after nth relaxation, the weights get decreased, it means there must be some -ve weight loop.

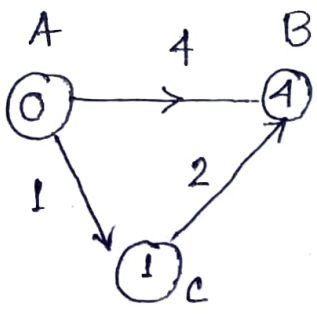
eg.



3 nodes

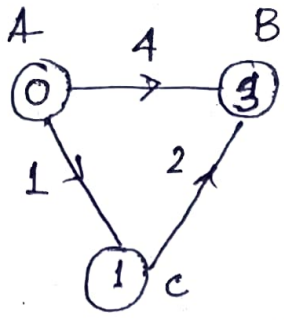
2 relaxations.

1st relaxation



Finding shortest path weights of length 1.

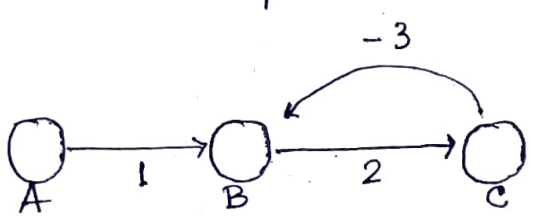
2nd relaxation.

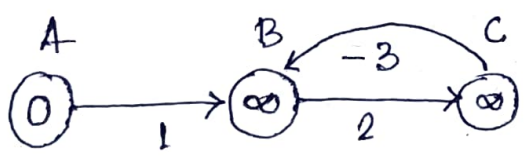


Finding shortest path weights of length 2.

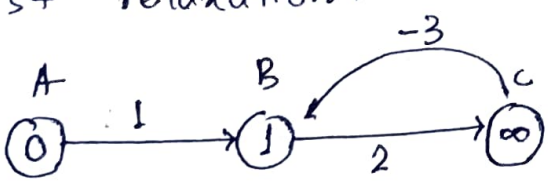
In the 3rd relaxation weights in the node remain same as no -ve edge cycle is present.

eg.

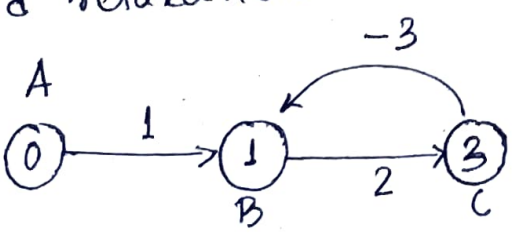




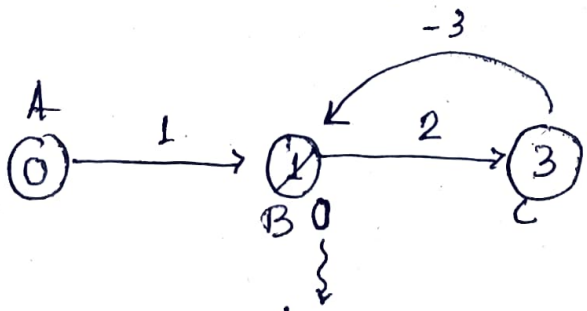
1st relaxation.



2nd relaxation



3rd relaxation to confirm about -ve edge cycle.



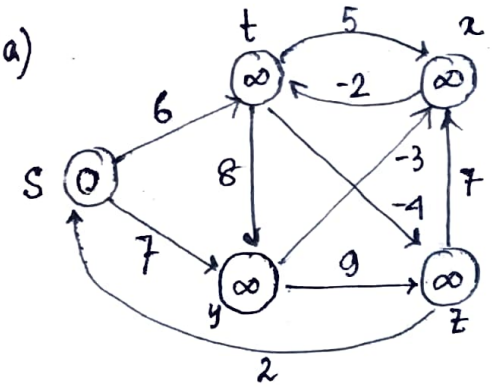
decrease

⇒ -ve edge cycle is present.

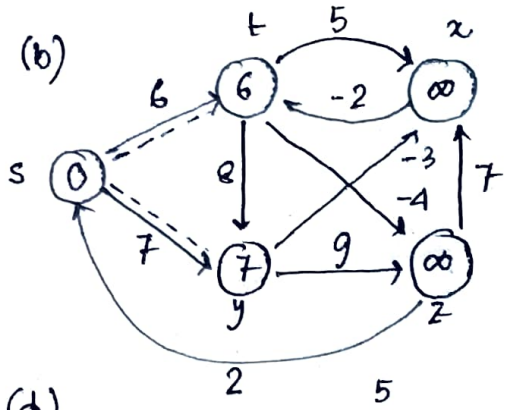
BFA runs for $|V| \times |E|$ relaxations. So, time complexity of BFA is $O(VE)$.

→ Execution of BFA.

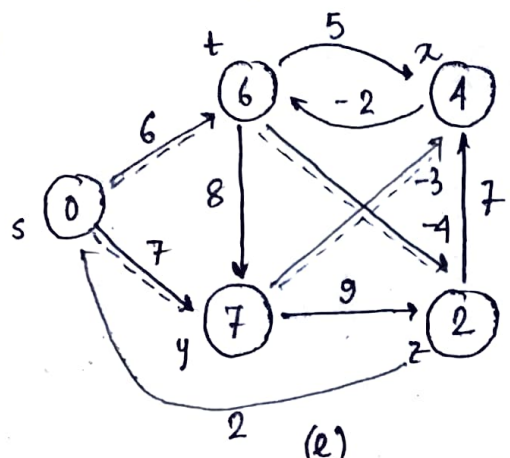
(a)



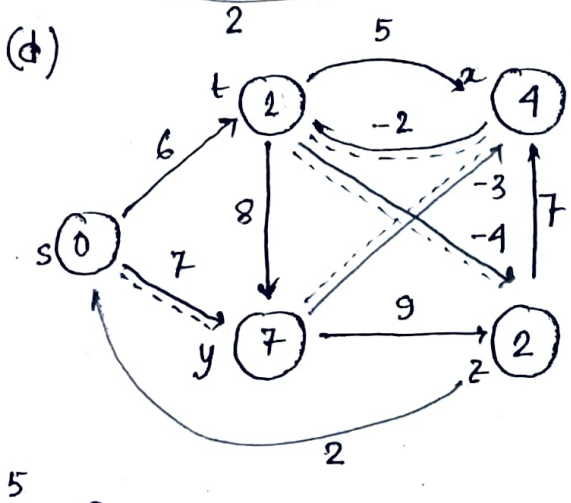
(b)



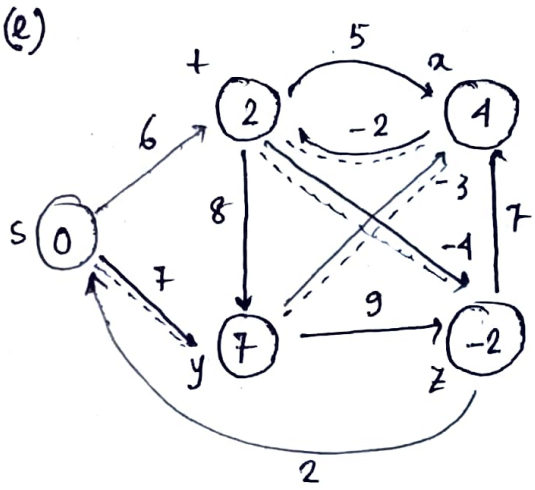
(c)



(d)



(e)



final value of d & π.

Each pass relaxes the edges in the order —

(t,x) (t,y) (t,z) (x,t) (y,x) (y,z) (z,x) (z,s) (s,t) (s,y)

Shortest paths from (e) figure

s to t - s y x t (2)

s to x - s y x (4)

s to y - s y (7)

s to z - s y x t z (-2).

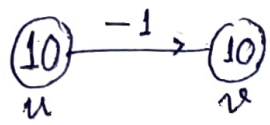
- Algorithm.

The algorithm relaxes edges, progressively decreasing an estimate ~~d~~ $d[v]$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest path weight $\delta(s, v)$.

TC:
avg, worst $O(|E||V|)$
SC: $O(|V|)$

BELLMAN_FORD (G, w, s)

1. Initialize_single_source (G, s).
2. for $i = 1$ to $|G.v| - 1$
3. for each edge $(u, v) \in G.E$
4. RELAX (u, v, w) — $O(1)$ as we take an array.
5. for each edge $(u, v) \in G.E$
6. if $v.d > u.d + w(u, v)$
7. return False
8. return True.



Line 1 initializes the d and π for all vertices.
Line 2 for loop iterates $|V| - 1$ times. Each pass is one iteration of the for loop of lines 2-4 & consists of relaxing each edge of ~~deg~~ the graph once. After that, lines 5-8 check for a -ve weight cycle & return the appropriate boolean value.

(TC)

Initialization on line 1 takes $\Theta(V)$ time, each of the $|V|-1$ passes over the edges on lines 2-4 takes $\Theta(E)$ time & the for loop of lines 5-7 takes $O(E)$ time. So, time complexity is $O(VE)$.

* Shortest path in DAG (Directed Acyclic Graph)

- Single source shortest path
- By relaxing the edges of a weighted DAG $G(V,E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\Theta(V+E)$ time.
- Shortest paths are always well defined in DAG, since even if there are -ve weight edges, no -ve weight cycles can exist.
- The algorithm starts by topologically sorting the DAG to impose a linear ordering on the vertices. If the DAG contains a path from u to v, then u precedes v in the topological sort. We make just one pass over the vertices in the topologically sorted order. As we process each vertex, we relax each edge that leaves the vertex.

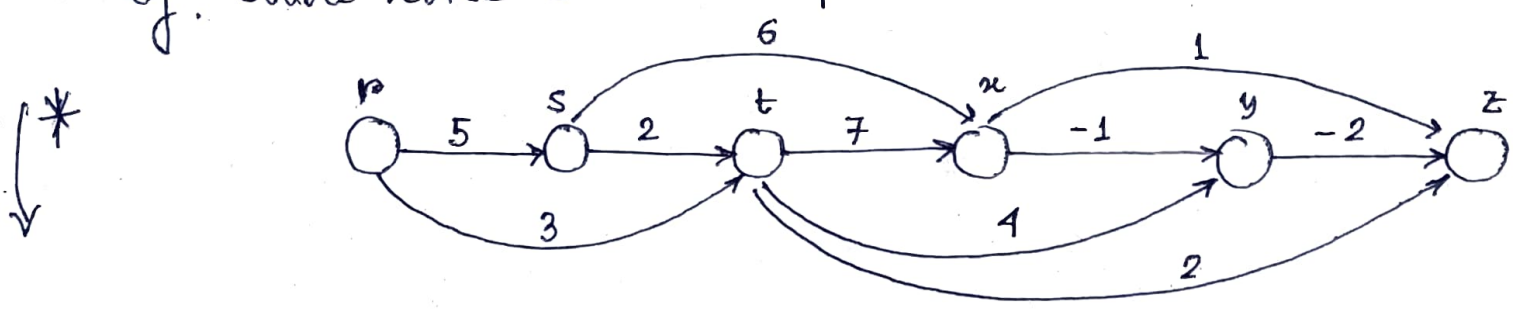
- DAG_SHORTEST_PATHS

```
1  topologically sort the vertices of G
2  Initialize_single_source (G, s)
3  for each vertex u, taken in topologically sorted or
4  |    for each vertex v ∈ G.Adj [u]
5  |    |  RELAX (u, v, w)
```

- Topological sort of line 1 takes $\Theta(V+E)$ time. Line 2 takes $\Theta(V)$ time. for loop of 3-5 makes one iteration per vertex. Altogether, the for loop of 4-5

relaxes each edge exactly once. Because each iteration of the inner loop takes $\Theta(1)$ time, the total running time is $\Theta(V+E)$, which is linear in the size of an adjacency-list representation of the graph.

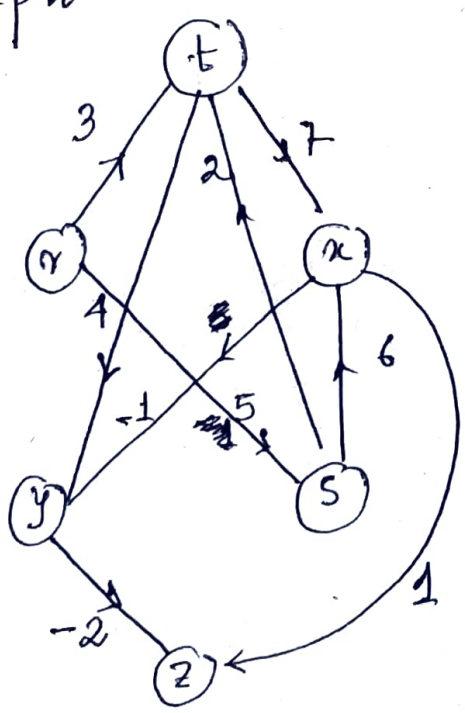
eg. Source vertex s. top-ordered



	r	s	t	x	y	z
	∞	0	∞	∞	∞	∞
Relaxing for r	∞	0	∞	∞	∞	∞
for s	∞	0	2	6	∞	∞
for t	∞	0	2	6	6	4
for x	∞	0	2	6	5	4
for y	∞	0	2	6	5	3
for z	∞	0	2	6	5	3 (final)

---

* Actual graph

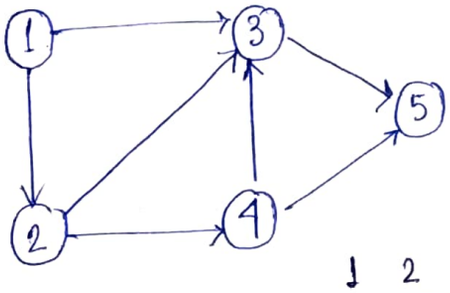


⇒ top. ordering

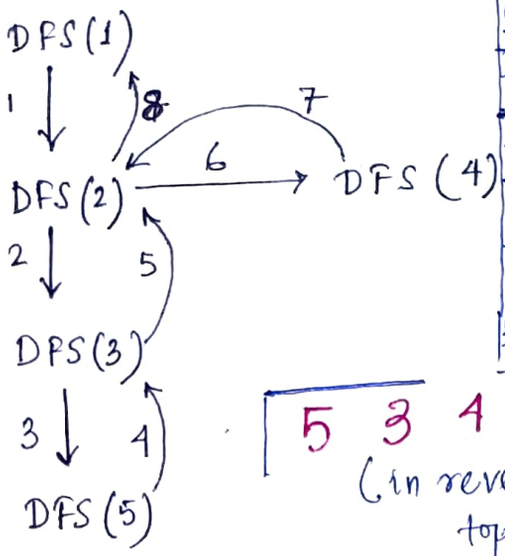
r, s, t, x, y, z

- Topological Sort. (for DAG) $O(V+E)$

Use finishing time for DFS



	1	2	3	4	5
visited	✓	✓	✓	✓	✓



	Entry	Exit
1	0	9
2	1	8
3	2	5
4	6	7
5	3	4

Decreasing order of exit time of nodes

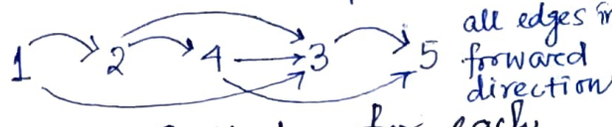
Based on the last finishing time of DFS.

1, 2, 4, 3, 5

— topologically sorted

5 3 4 2 1

(in reverse order - top. sorted)



- Defn: Linear ordering of vertices such that for each directed edge uv for vertex u to v, u comes before v in the ordering. Every DAG has at least one top. sorting order.

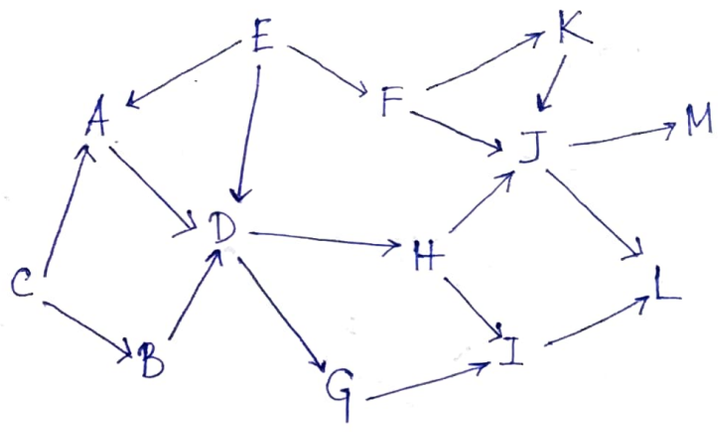
- Algo: a) Pick an unvisited node.

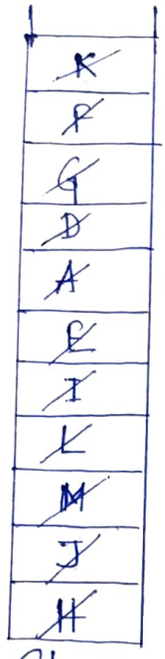
b) Beginning w the selected node, do a DFS exploring only unvisited nodes.

c) On the recursive callback of the DFS, add the current node to the topological ordering in reverse order.

```
function dfs(at, V, visitedNodes, graph):
    V[at] = true
    edges = graph.getEdgesOutFromNode(at)
    for edge in edges:
        if V[edge.to] == false:
            dfs(edge.to, V, visitedNodes, graph)
    visitedNodes.add(at)
```

eg.





Pick unvisited node — H 1st
E 2nd
C 3rd

Top. ordering (in reverse) ↓

M - L - J - I - H - G - D - A - K - F - E - B - C.

Visited.
H J M L I E A D G F K C B

* Pseudocode. Graph stored as adj. list

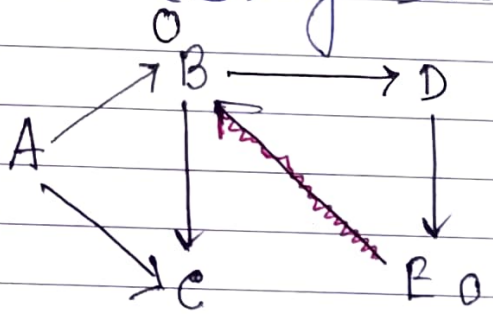
```
function topsort (graph):
    N = graph.numberOfNodes();
    V = [false, ..., false]   # length N
    ordering = [0, ..., 0]    # length N
    i = N-1 ;  # index for ordering array
    for (at = 0 ; at < N ; at++):
        if V[at] == false:
            visitedNodes = []
            dfs (at, V, visitedNodes, graph)
            for nodeId in visitedNodes:
                ordering[i] = nodeId
                i = i-1
return ordering.
```

dfs on back

233-132
34th Week

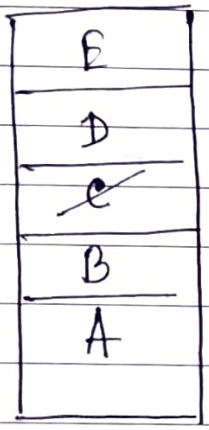
Thursday 21

✱ Detect cycle in Directed Graph (Using DFS).



flag : -1 unvisited
0 visited, in stk
1 visited, popped from stk

Stack



Visited set

A B C D E

Parent map

Vertex	Parent
A	-
B	A
C	B
D	B
E	D

If any vertex finds its adj. vertex to have flag 0, then there's cycle.

Cycle : E → B, D → E, B → D

checking parent of E

checking parent of D

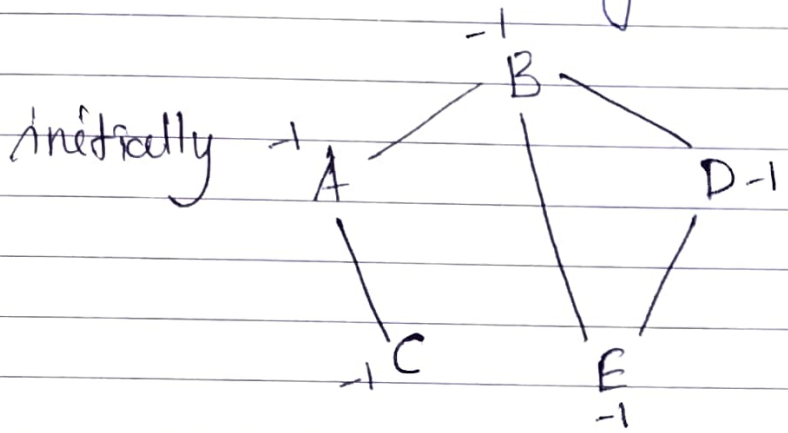
B → D → E

Essential | Job to do | Phone No.

235-130
34th Week

Saturday 23

※ Detect cycle in Undirected Graph (Using BFS)

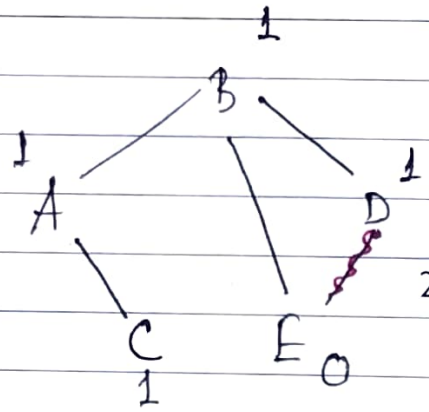


flag:

-1 unvisited
0 in queue
1 traversed

Queue

A	B	C	D	E



24 Sunday

Visited
A B C D

If found a vertex, whose adj. vertex with flag 0, then contains cycle.

Essential	Job to do	Phone No.

Dynamic Programming

- Solve the problem by combining the solutions to smaller subproblems.
- In DP, the subproblems overlap & the solutions to inner problems are stored in memory. This avoids the work of repeatedly solving the innermost problem.
- DP is most often used to solve combinatorial optimization problems, where we are looking for the best possible input to some function chosen from an exponentially large search space.
- DP is applicable when the subproblems are not independent. Subproblems share ~~subs~~ subproblems. Divide & conquer does more work than necessary repeatedly solving common subproblems.
- 2 methods of storing the result in memory -

1. Memoization (Top-down)

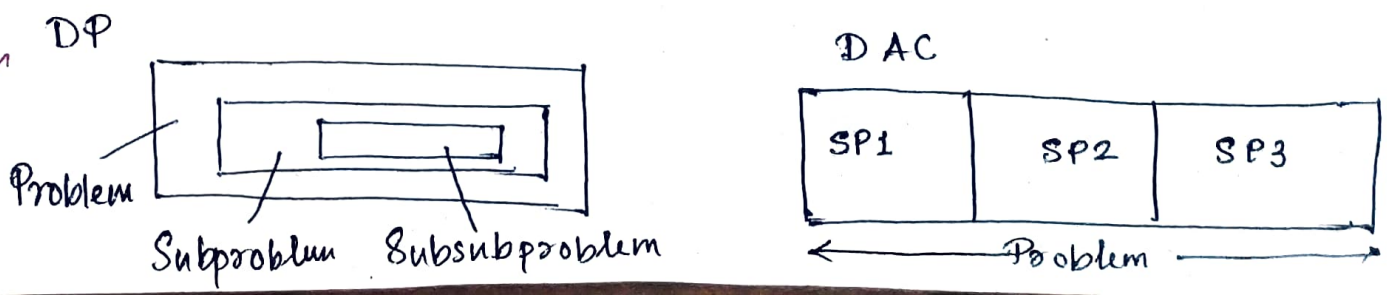
Whenever we solve a problem first time we store it & reuse it next time (make memo).

2. Tabulation (Bottom-up)

We precompute the solutions in linear fashion & later use them.

Overlapping subproblems
Optimal substructure

- In Divide & Conquer, subproblems are disjoint.



• The key feature that a problem must have in order to be amenable to DP is that of optimal substructure : the optimal solution to the problem must contain optimal solutions to subproblems.

- Overlapping subproblems.

- Longest path problem doesn't have optimal substructure property.

* Example with n^{th} Fibonacci number.

$$F(n) = F(n-1) + F(n-2)$$

$$F(1) = F(0) = 1.$$

Naive recursive approach is -

```
int fib (int n).
{
    if (n < 2)
        return 1 ;
    else
        return fib (n-1) + fib (n-2) ;
}
```

$$T(n) = T(n-1) + T(n-2) + \theta(1) = \Theta(a^n)$$

a is golden ratio (1.618033...)

Problem is we keep recomputing values of fib that we've already computed. We avoid this by memoization, where we wrap our solution to a memoizer that stores previously computed solutions in a hash table.

STEPS IN DP

Optimal Substructure
↓
Recursive Eqn
↓
Recursion Tree
↓
Overlapping subproblems ?
↓ Yes
Memoise / Tabulate

```
int memoFib (int n) {
    int ret ;
    if ( hashContains (FibHash, n)) {
        return hashGet (FibHash, n) ;
    } else {
        ret = memoFib (n-1) + memoFib (n-2) ;
        hashPut ( FibHash, n, ret) ;
        return ret;
    }
}
```

Bottom up DP. ~

```
int fib2 (int n) {
    int *a ;
    int i, ret ;
    if (n < 2) {
        return 1;
    } else {
        a = malloc (sizeof (*a) * (n+1)) ;
        assert (a) ;
        a[1] = a[2] = 1 ;
        for (i = 3 ; i <= n ; i++) {
            a[i] = a[i-1] + a[i-2] ;
        }
    }
    ret = a[n] ;
    free (a) ;
    return ret ;
}
```

$\Theta(n)$.

* Matrix Chain Multiplication.

Two matrices of size $p \times q$ & $q \times r$ when multiplied, we need to do $(p \times r) \times q = p \times q \times r$ no. of scalar multiplications.

– Given a sequence of n matrices $\langle A_1, A_2, \ldots, A_n \rangle$ to be multiplied, & we wish to compute the product $A_1 A_2 \cdots A_n$.

– If the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$ then we can fully paranthesize the product in 5 distinct ways.

$(A_1 \ (A_2 \ (A_3 A_4)))$.	$((A_1 \ (A_2 A_3)) A_4)$.
$(A_1 \ ((A_2 A_3) A_4))$	$(((A_1 A_2) A_3) A_4)$.
$((A_1 A_2) \ (A_3 A_4))$	

– We state the matrix chain multiplication problem as follows : given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully paranthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

– Counting no. of parenthesizations. :

Total no. of ways = $(n-1)^{th}$ Catalan number

$$= \binom{2(n-1)}{(n-1)} \Big/ n$$

$$= \frac{(2n)!}{(n+1)!\ n!}$$

For $n = 5$ matrices, # ways $= \frac{(2 \times 4)!}{5!\ 4!} = 14$.

- No. of alternative paranthesizations of a sequence of n matrices be $P(n)$. When $n = 1$, we have just one matrix. When $n \geq 2$, a fully paranthesized matrix product is the product of two fully paranthesized matrix subproducts, & the split between the two subproducts may occur between the k^{th} & the $(k+1)^{st}$ matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 \end{cases}$$

Soln to this recurrence being $\Omega(2^n)$, brute force method of exhaustive search makes for a poor strategy when determining how to optimally paranthesize a matrix chain.

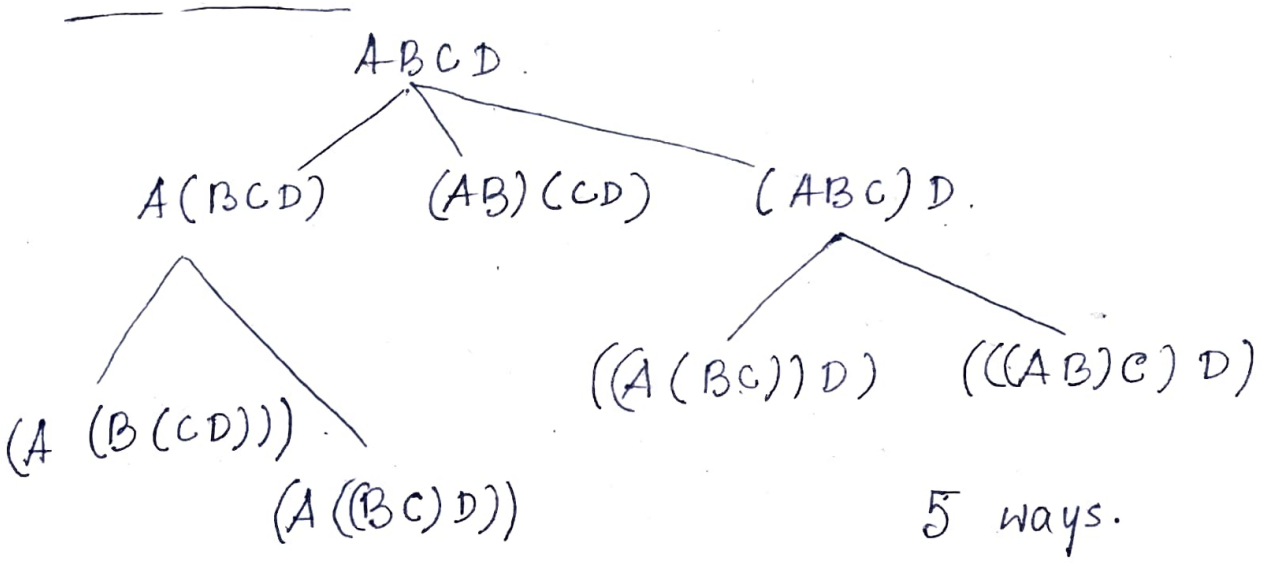
- eg. Paranthesization -

3 matrices

ABC

A(BC) (AB)C 2 ways.

4 matrices



5 ways.

# of scalar multiplications -

A	2 x 1
B	1 x 2
C	2 x 3
D	3 x 1.

(A(B(CD))) → (2 x 3 x 1) +
(1 x 2 x 1) +
(2 x 1 x 1)
= 6 + 2 + 2 = (10)

(A((BC)D)) → (1 x 2 x 3) +
(1 x 3 x 1) +
(2 x 1 x 1)
= 6 + 3 + 2 = (11)

(AB)(CD) → (2 x 1 x 2) +
(2 x 3 x 1) +
(2 x 2 x 1)
= 4 + 6 + 4 = (14)

((A(BC))D) → (1 x 2 x 3) +
(2 x 1 x 3) +
(2 x 3 x 1)
= 6 + 6 + 6 = (18)

(((AB)C)D) → (2 x 1 x 2) +
(2 x 2 x 3) +
(2 x 3 x 1)
= 4 + 12 + 6 = (22).

So, (A(B(CD))) is the optimal parenthesization way to minimise # of scalar multiplications.

→ A recursive solution.

Find the optimal substructure & then use it to construct an optimal solution to the problem from optimal solutions to subproblems.

Let, $A_{i..j}$ $(i \le j)$ is the product $A_i A_{i+1} \cdots A_j$

If $i < j$, then to parenthesize the product, we must split the product between A_k & A_{k+1} for some integer k in the range $i \le k < j$. For some value of k, we first compute the matrices $A_{i..k}$ & $A_{k+1..j}$ & then multiply them together to produce the final product $A_{i..j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$ plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

✓ Optimal substructure : the way we parenthesize the prefix subchain $A_i A_{i+1} \cdots A_k$ & suffix subchain $A_{k+1} A_{k+2} \cdots A_j$ within the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$ & $A_k A_{k+1} \cdots A_j$ respectively.

We can build an optimal solution to an instance of the MCM problem by splitting

the problem into two subproblems (optimally parenthesizing $A_i \cdots A_k$ & $A_{k+1} \cdots A_j$), finding optimal solutions to subproblem instances, & then combining these optimal subproblem solutions.

- Now, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. Subproblem: determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \leq i \leq j \leq n$. Let, $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest cost way to compute $A_{1..n}$ would be $m[1,n]$.

When $i = j$, $m[i,j] = 0 = m[i,i]$ for $i = 1, 2, \ldots, n$ (only 1 element).

When $i < j$, we assume we split the product $A_i A_{i+1} \cdots A_j$ at k (between A_k & A_{k+1}), where $i \leq k < j$. Now, as each matrix A_i is of ~~dimension~~ size $p_{i-1} \times p_i$, computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications.

$(i-1) \times k$ | $k \times j$

Thus,

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

$i < j$

$k = i, i+1, \cdots, j-1$ [$j-i$ possible values for k]

✓ We need to check for all of the k to find the best for optimal parenthesization.

✓✓
$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

Now, let $s[i,j]$ be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization.

✓ $m[i,j] = m[i, s[i,j]] + m[[s[i,j]+1], j] + p_{i-1} p_{s[i,j]} p_j$

eg. M_1 M_2 M_3

$p_0 \times p_1$ $p_1 \times p_2$ $p_2 \times p_3$.

2 ways of full parenthesizations

$(M_1 (M_2 M_3))$

$s[i,j] = 1$

$= M_1 \times [M_{23}]$

$p_0 \times p_1$ $p_1 \times p_3$

$= [M_{123}]_{p_0 \times p_3}$

$m[i,j] = m[1,3]$

$= m[1,1] + m[2,3] + p_0 p_1 p_3$

$((M_1 M_2) M_3)$

$s[i,j] = 2$

$= [M_{12}] \times M_3$

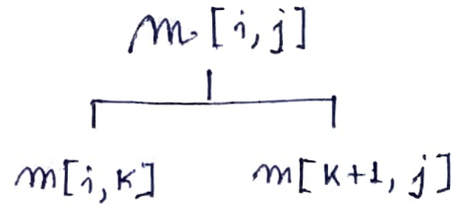
$p_0 \times p_2$ $p_2 \times p_3$

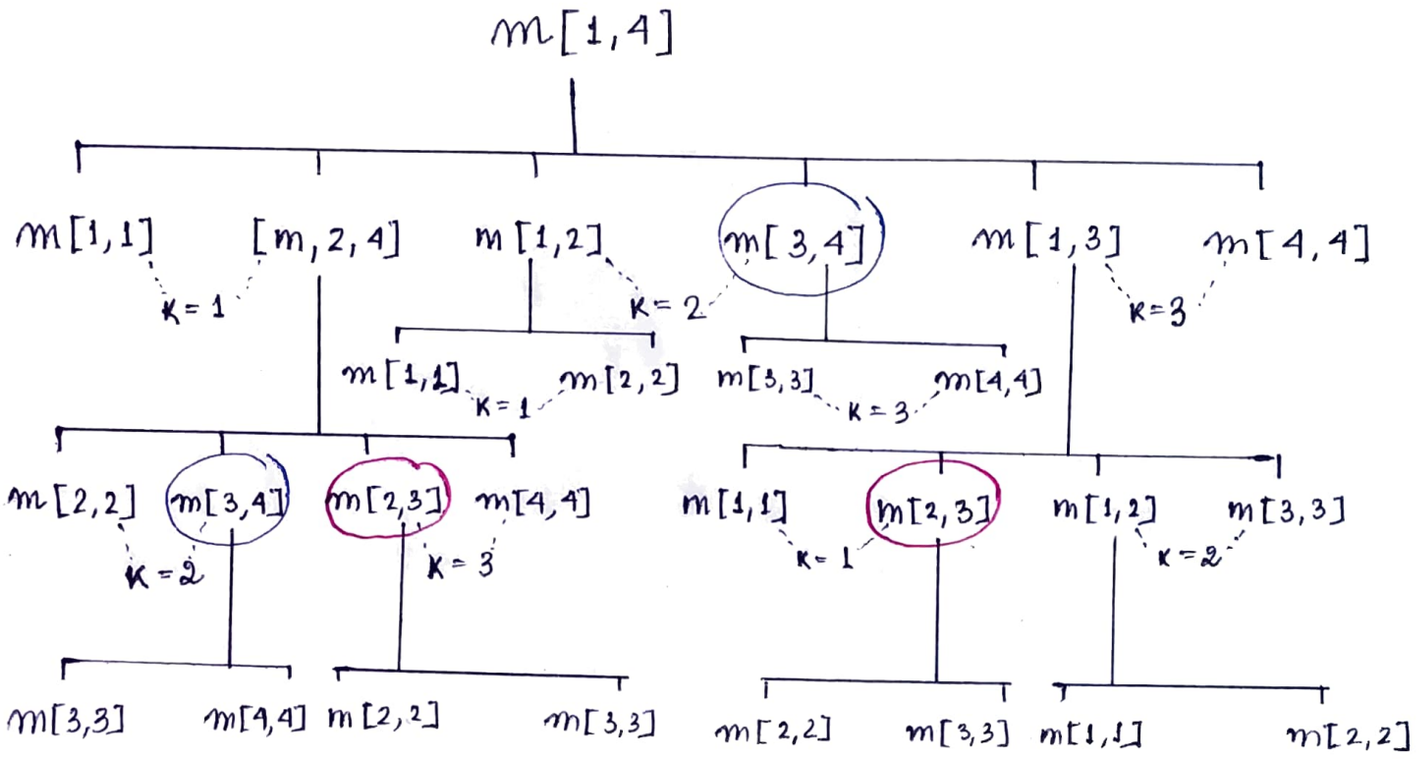
$= [M_{123}]_{p_0 \times p_3}$

$m[i,j] = m[1,3] = m[1,2] + m[3,3] + p_0 p_2 p_3$

※ MCM Recursion tree.

4 matrices A_1, A_2, A_3, A_4.

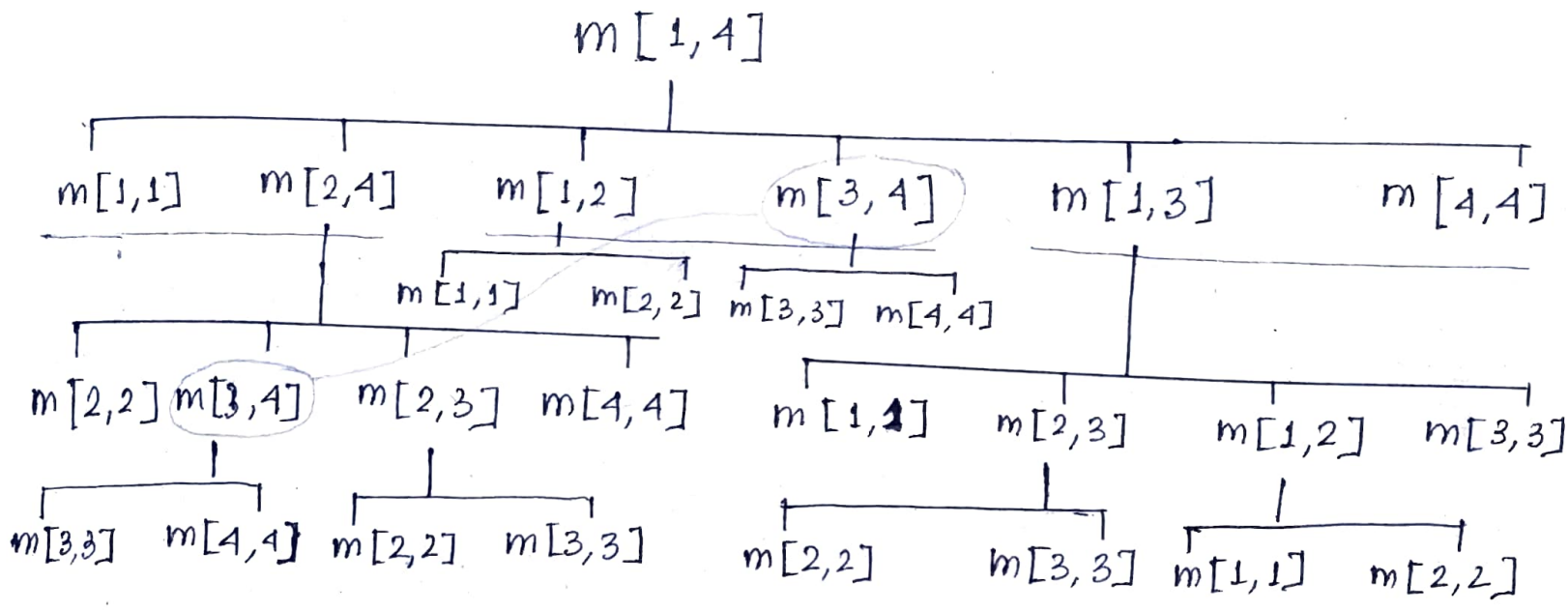




Overlapping subproblems exist.

eg. Recursion tree.

$A_1 A_2 A_3 A_4$.



Max. depth 4. $O(n)$.

Overlapping subproblems: several subproblems called multiple times. We can use DP for memoization.

- Total no. of subproblems:

size 1	(1,1) (2,2) (3,3) (4,4)	4
size 2	(1,2) (2,3) (3,4)	3
size 3	(1,3) (2,4)	2
size 4	(1,4).	1
	+	10.

Generally, for size $1, 2, \ldots, n$

$$n + (n-1) + \ldots + 1. = \frac{n(n+1)}{2}.$$

subproblems (unique)

eg. Computing optimal costs.

A_1	A_2	A_3	A_4
2×5	5×4	4×2	2×3

p_0 2
p_1 5
p_2 4
p_3 2
p_4 3

for $i = j$, 0
for $i > j$, NA

i \ j	1	2	3	4
1	0	40	56	(68)
2	x	0	40	70
3	x	x	0	24.
4	x	x	x	0

$$m[1,2] = m[1,1] + m[2,2] + p_0 p_1 p_2$$

$$= 0 + 0 + 2 \times 5 \times 4 = 40$$

$$m[1,3] = \min_{1 \le k < 3} \{ m[1,k] + m[k+1,3] + p_0 p_k p_3 \}$$

$$= \min \begin{Bmatrix} m[1,1] + m[2,3] + 2 \times 5 \times 2, \\ m[1,2] + m[3,3] + 2 \times 4 \times 2 \end{Bmatrix}$$

$$= \min \begin{Bmatrix} 0 + 40 + 20, \\ 40 + 0 + 16 \end{Bmatrix} = 56$$

(Note: split at $k = 2$)

$$m[2,3] = m[2,2] + m[3,3] + p_1 p_2 p_3 = 5 \times 4 \times 2 = 40$$

$m[3,4] = 24.$ $\quad m[2,4] = 70$

$m[1,4] = (68)$

So, 68 scalar multiplications are required to multiply A_1, A_2, A_3, A_4 optimally.

- So, we have to compute value of the optimal solution in a bottom-up fashion.

2D DP memo table $m[1..n, 1..n]$

$m[i,j]$ only defined for $i \le j$.

While using the equation

$$m[i,j] = \min_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right),$$

to calculate $m[i,j]$ we must have already evaluated $m[i,k]$ & $m[k+1,j]$. For both cases, the corresponding length of the matrix chain are both less than $j-i+1$. Hence, the algorithm should fill the table in increasing order of the length of the matrix chain.

We calculate in the order — (Bottom-up)

$m[1,2], m[2,3], m[3,4], \ldots, m[n-2,n-1], m[n-1,n]$

$m[1,3], m[2,4], \ldots, m[n-3,n-1], m[n-2,n]$

$\vdots$

$m[1,n-1], m[2,n]$

$m[1,n]$.

- We have $\frac{n(n+1)}{2} = O(n^2)$ subproblems & each subproblem is split in $k = j-i$ ways $(O(n))$

So, overall time complexity is $O(n^3)$.

Space complexity = Space taken by the table

$= O(n^2)$

– Algorithm.

$A_1 A_2 \cdots A_n \mid p_{i-1} \times p_i$

Input: sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.length = n+1$.

⊕ Auxiliary table $m[1..n, 1..n]$ for storing the $m[i,j]$ costs & another $s[1..n-1, 2..n]$ that records which index of k achieves the optimal cost in computing $m[i,j]$.

MATRIX_CHAIN (p). (Tabulation). Bottom-up

```
n = p.length - 1
let m[1..n, 1..n] & s[1..n-1, 2..n] be new tables.
for i = 1 to n
    m[i,i] = 0
for l = 2 to n      // l being the chain length
    for i = 1 to n-l+1
        j = i+l-1
        m[i,j] = ∞
        for k = i to j-1
            q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
            if q < m[i,j]
                m[i,j] = q
                s[i,j] = k
return m and s.
```

{ TC $O(n^3)$
{ SC $O(n^2)$

⊕ Memoized version - Top-down approach.

MEMOIZED_MATRIX_CHAIN (p).

```
n = p.length - 1
let m[1..n, 1..n] be a new table.
for i = 1 to n
    for j = 1 to n                    { TC O(n^3)
        m[i, j] = ∞
return LOOKUP_CHAIN (m, p, 1, n).
```

LOOKUP_CHAIN (m, p, i, j)

```
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j - 1
    q = LOOKUP_CHAIN (m, p, i, k) +
        LOOKUP_CHAIN (m, p, k+1, j) +
        p_{i-1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Using recursion calls.

Actual space taken little more due to recursion stack.

$$SC = O(n^2) + \underbrace{O(n)}_{\text{Stack}}$$

$$= O(n^2).$$

⊕ Memoized version - Top-down approach.

MEMOIZED_MATRIX_CHAIN (p).

```
n = p.length - 1
let m[1..n, 1..n] be a new table.
for i = 1 to n
    for j = 1 to n                    { TC O(n^3)
        m[i, j] = ∞
return LOOKUP_CHAIN (m, p, 1, n).
```

LOOKUP_CHAIN (m, p, i, j)

```
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j - 1
    q = LOOKUP_CHAIN (m, p, i, k) +
        LOOKUP_CHAIN (m, p, k+1, j) +
        p_{i-1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Using recursion calls.

Actual space taken little more due to recursion stack.

$$SC = O(n^2) + \underbrace{O(n)}_{\text{Stack}}$$

$$= O(n^2).$$

G'11 4 matrices M_1, M_2, M_3, M_4 of dimensions $p \times q$, $q \times r$, $r \times s$ & $s \times t$ respectively can be multiplied in several ways with different number of total scalar multiplications. If $p = 10$, $q = 100$, $r = 20$, $s = 5$ & $t = 80$, then the min. no. of scalar multiplications needed is — 19000.

i \ j	2	3	4
1	$p_0 p_1 p_2$ = 20000	min (10000 + $p_0 p_1 p_3$, 20000 + $p_0 p_2 p_3$) = 15000 (s = 1)	min (18000 + $p_0 p_1 p_4$, 20000 + 8000 + $p_0 p_2 p_4$, 15000 + $p_0 p_3 p_4$) = 19000 (s = 3)
2	0	$p_1 p_2 p_3$ = 10000	min (10000 + $p_2 p_3 p_4$, $p_1 p_3 p_4$) = 18000. (s = 3)
3	x	0	$p_2 p_3 p_4$ = 8000

Ordering

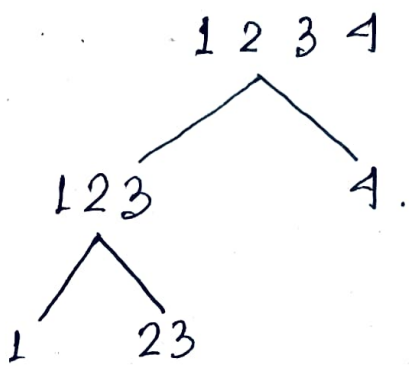
$((M_1 (M_2 M_3)) M_4)$

23

123

1234.

s - split location



* Longest common subsequence.

Given 2 sequences, $X = \langle x_1, \ldots, x_m \rangle$, and $Y = \langle y_1, \ldots, y_n \rangle$. find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

eg. s p r i n g t i m e

p i o n e e r pine

→ Brute force algorithm.

For every subsequence of X, check whether it's a subsequence of Y.

2^m subsequences of X to check. Each subsequence takes $\Theta(n)$ time to check: Scan Y for first letter, from there scan for second & so on. Overall time : $\Theta(n2^m)$.

	m	n
	a	a
	b	c
	c	b
	d	c
	e	m
$\Theta(n)$	x	

→ Optimal substructure of an LCS.

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the i^{th} prefix of X for $i = 0, 1, \ldots, m$ as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. For example, if $X = \langle A, B, C, B, D, A, B \rangle$ then $X_4 = \langle A, B, C, B \rangle$ & X_0 is the empty sequence.

Let $lcs(i, j)$ denote the length of the LCS of X_i & Y_j.

X_i = prefix $\langle x_1, \ldots, x_i \rangle$

Y_i = prefix $\langle y_1, \ldots, y_i \rangle$

$X = \langle x_1, \ldots, x_m \rangle$

$Y = \langle y_1, \ldots, y_n \rangle$

Let Z be any LCS of X & Y.

$Z = \langle z_1, z_2, \ldots, z_k \rangle$

$X_m \equiv X$

$Y_n \equiv Y$

1. If $x_m = y_n$, then $z_k = x_m = y_n$ &

Z_{k-1} is an LCS of X_{m-1} & Y_{n-1}.

2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow$

Z is an LCS of X_{m-1} & Y.

3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow$

Z is an LCS of X & Y_{n-1}

→ $X_5 = \langle ABRAC \rangle$

$Y_6 = \langle YABBAD \rangle$

LCS = $\langle ABA \rangle$

lcs (5,6) = 3

Optimal substructure
↓
Recursive equation
↓
Recursion tree
↓
Overlapping subprob
↓ Yes
Memoise / Tabulate

→ Recursive formulation for computing lcs (i, j)

Basis If either sequence is empty, then the LCS is empty. Therefore,

✓ lcs (i, 0) = lcs (0, j) = 0.

Last characters match $x_i = y_j$

For example, $X_i = \langle ABCA \rangle$ & $Y_j = \langle DACA \rangle$.

Since, both end in 'A', LCS must end in A too.

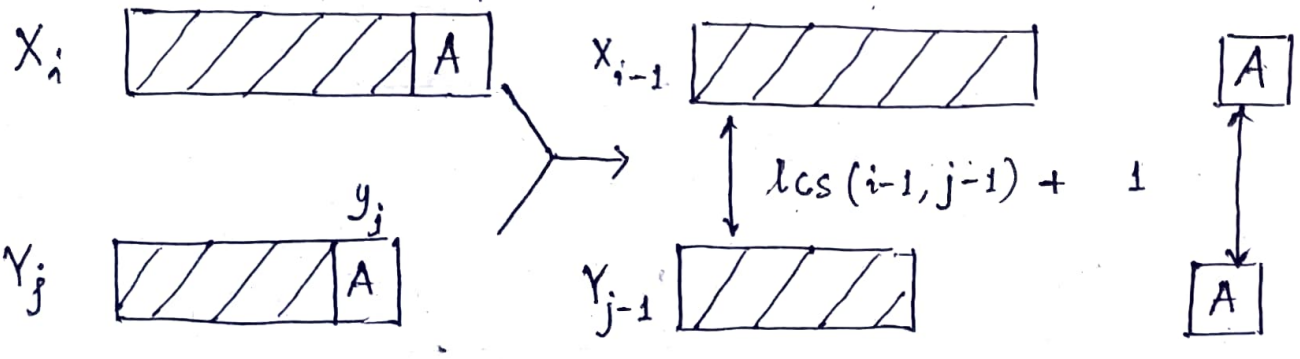
So, we may find the overall LCS by ① removing last 'A' from both sequences, ② taking the LCS of $X_{i-1} = \langle ABC \rangle$ and $Y_{j-1} = \langle DAC \rangle$ which is $\langle AC \rangle$ & ③ adding 'A' to the end. This yields $\langle ACA \rangle$ as the LCS. Therefore, the length of the final LCS is the length

$$\frac{lcs\,(X_{i-1}, Y_{j-1})}{AC} + \frac{1}{A}$$

. So,

{ if $(x_i = y_j)$ then

$$lcs\,(i, j) = lcs\,(i-1, j-1) + 1.$$



Last characters do not match. $x_i \neq y_j$

x_i & y_j cannot both be in the LCS. Thus, either x_i is not part of the LCS, or y_j is not part of the LCS (& possibly both are not the part of the LCS.).

The DP selection principle – When given a set of feasible options to choose from, try them all & take the best. Now, considering

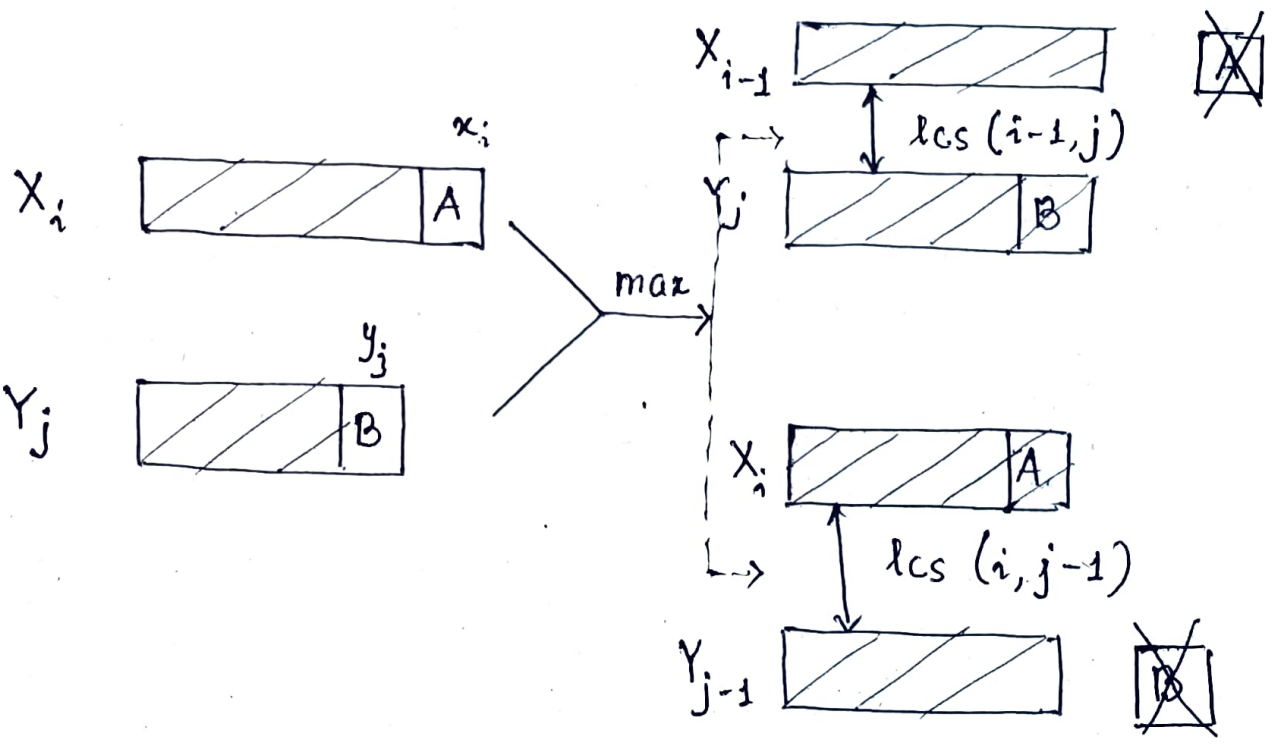
both options –

1. (x_i is not in the LCS): $x_i \neq z_k$

We can infer that the LCS of X_i & Y_j is the LCS of X_{i-1} & Y_j, which is given by lcs $(i-1, j)$.

2. (y_j is not in the LCS): $y_i \neq z_k$

LCS of X_i & Y_j is the LCS of X_i & Y_{j-1}, which is given by lcs $(i, j-1)$.



So, if $x_i \neq y_j$,

$$\text{lcs}(i,j) = \max\left(\text{lcs}(i-1,j),\ \text{lcs}(i,j-1)\right)$$

Combining all, we have recursive formulation –

$$\text{lcs}(i,j) = \begin{cases} 0 & i=0 \text{ or } j=0 \\ \text{lcs}(i-1,j-1)+1 & i,j>0 \text{ \& } x_i = y_j \\ \max\left(\begin{matrix}\text{lcs}(i-1,j),\\ \text{lcs}(i,j-1)\end{matrix}\right) & i,j>0 \text{ \& } x_i \neq y_j \end{cases}$$

Direct recursive implementation –

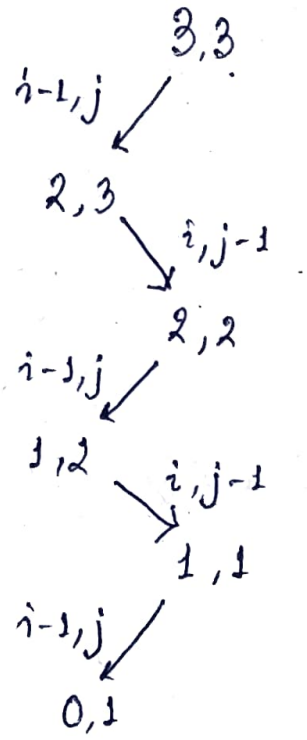
$i = 3 \quad j = 3.$

best case when every character matches.

3,3
↓
2,2
↓
1,1
↓
0,0.

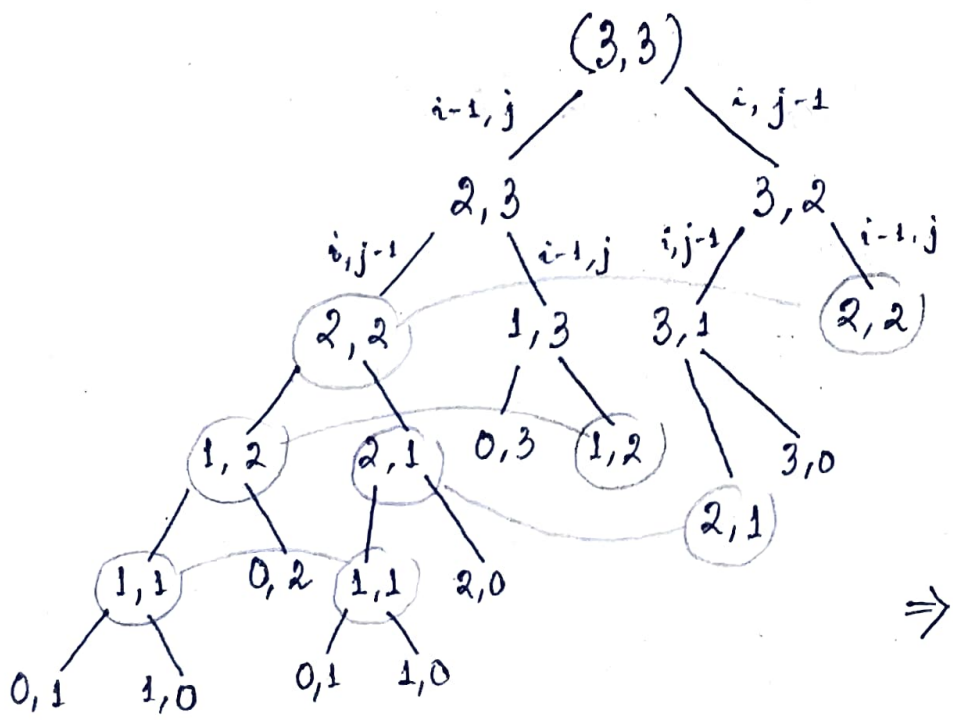
Total 4 function calls.

Worst case when characters don't match and the maximum lcs for $(i-1, j)$ & $(i, j-1)$ in an alternative fashion.



Total. 6 function calls.

Recursion tree when characters don't match.



We see overlapping subproobs

⇒ Apply DP.

→ Dynamic programming approach.

---

eg. 1. Instead of recomputing we store in a table.

$X = \langle A, A, B \rangle$ $\quad Y = \langle A, C, A \rangle$

DP table

		0	1	2	3
		Y	A	C	A
0	X	0	0	0	0
1	A	0	1+0 = 1	max(1,0) = 1	1+0 = 1
2	A	0	1+0 = 1	max(1,1) = 1	1+1 = 2
3	B	0	max(1,0) = 1	max(1,1) = 1	max(1,2) = (2)

LCS table

LCS length

(1,1) ≡ (A,A) matches so $1 + lcs(i-1, j-1)$

(1,2) ≡ (A, C) doesn't match so

$\max(lcs(i-1, j), lcs(i, j-1))$

We can go rowwise, or columnwise to calculate the lcs values.

Finding the LCS —

		Y	A	C	A
0	X	0	0	0	0
1	A	0	(1)	1	(1)
2	A	0	(1)	1	(2)
3	B	0	1	1	(2)

○ circle if match

$\langle A\ A \rangle$ LCS

Start from here & put the character that matches at the end of LCS

- When case like max (2, 2) arises, we take both path one by one to find different LCSs.

- Finding LCS from table :

if 0 (or match) go ↖

if not a 0 go ← ↑ & find max and that way

eg.2. $X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$.

		0	1	2	3	4	5	6
		Y	B	D	C	A	B	A
0	X	0	0	0	0	0	0	0
1	A	0	0	0	0	(1)	1	(1)
2	B	0	(1)	1	1	1	(2)	2
3	C	0	1	1	(2)	2	2	2
5	B	0	(1)	1	2	2	(3)	3
6	D	0	1	(2)	2	2	3	3
7	A	0	1	2	2	(3)	3	(4)
8	B	0	(1)	2	2	3	(4)	4.

○ circle if match occurs

LCS length 4.

LCS

$\langle BCBA \rangle$

$\langle BDAB \rangle$

$\langle BCAB \rangle$.

※ Memoized implementation.

```
memoized_lcs (i, j) {
    if (lcs[i,j] has not yet been computed) {
        if (i=0 || j=0)
            lcs[i,j] = 0
        else if (x[i] == y[j])
            lcs[i,j] = 1 + memoized_lcs (i-1, j-1)
        else
            lcs[i,j] = max (memoized_lcs (i-1, j),
                            memoized_lcs (i, j-1))
    }
    return lcs[i,j] ;
}
```

There are $m+1$ possible values for i, $n+1$ possible values for j. Each time we call memoized_lcs(i,j) if it has already been computed then it returns in $O(1)$. Each call to memoized_lcs (i,j) generates a constant no. of additional calls. Therefore, the time needed to compute the initial value of any entry is $O(1)$ & all subsequent calls with the same arguments is $O(1)$. Total running time is equal to the no. of entries computed that is

$$O((m+1)(n+1)) = O(mn).$$

SC = $O(mn)$

* Bottom-up implementation.

Create the lcs table in a bottom-up manner. By the recursive calls, in order to compute lcs $[i,j]$, we need to have already computed lcs $[i-1,j-1]$, lcs $[i-1,j]$ & lcs $[i,j-1]$.

```
bottom_up_lcs (x,y) { m = x.len ; n = y.len ;
    lcs = new array [0..m, 0..n] ;
    for (i = 0 to m)   lcs [i,0] = 0 ;
    for (j = 1 to n)   lcs [0,j] = 0 ;
```

```
for (i=1 to m) {
    for (j=1 to n) {
        if ( X[i] == Y[j] )
            lcs[i,j] = lcs[i-1,j-1] + 1 ;
        else
            lcs[i,j] = max ( lcs[i-1,j], lcs[i,j-1] ) ;
    }
}
return lcs[m,n] ;
}
```

✓ Running time $\Theta(mn)$

since each entry take $\Theta(1)$

Space complexity $O(mn)$.

Q. G'14. $A = \langle q\ p\ q\ r\ r \rangle$, $B = \langle p\ q\ p\ r\ q\ r\ p \rangle$.

x be the length of LCS & y be the no. of such LCSs. Then $x + 10y = 34$.

→ Instead of making DP table, use brute force as length of strings is small. We first check if there exists a subseq. of length 5 since $\min(|A|, |B|) = 5$.

len 5 ⇒ no subseq.

len 4 ⇒ $q\ p\ q\ r$, $q\ p\ r\ r$, $q\ q\ r\ r$, $p\ q\ r\ r$ from A

3 LCSs

$x = 4$, $y = 3$

$q\ p\ q\ r$ all
$q\ p\ r\ r$ possible
$q\ q\ r\ r$ subseq
$p\ q\ r\ r$ of len 4

take the least length string & check for its subsequences

Q. G'09 A subsequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given 2 sequences X[m], Y[n] of lengths m,n, with indices of X & Y starting from 0.

$lcs(i,j) = 0 \quad , \quad i=0 \,||\, j=0$

expr1 , $i,j>0$ & $X[i-1] = Y[j-1]$

expr2 , $i,j>0$ & $X[i-1] \neq Y[j-1]$

Using DP, array L[M,N] where $M = m+1$, $N = n+1$ such that $L[i,j] = l(i,j)$.

Which is true?

a) All elems of L should be initialized to 0 for the values of $l(i,j)$ to be properly computed.

✓ b) Values of $l(i,j)$ may be computed in a row major or column major order of L[M,N].

c) Values of $l(i,j)$ can't be computed in either row or column major order of L[M,N].

X d) L[p,q] needs to be computed before L[r,s] if either $p<r$ or $q<s$.

(d) to be true $p<r$ & $q<s$ should be the condition.

Read questions carefully!

* Shortest Path in Multistage Graph.

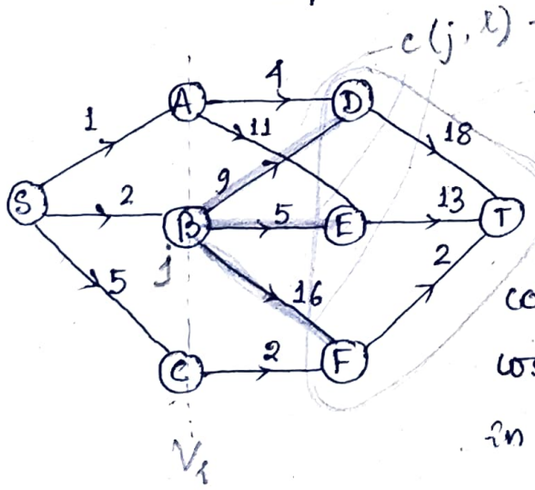
{ In optimisation problems, greedy methods may fail to give the optimal answer. Greedy is fast compared to DP.

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if $\langle u, v \rangle$ is an edge in E then $u \in V_i$ & $v \in V_{i+1}$ for some i, $1 \leq i < k$. Sets V_1 & V_k are such that $|V_1| = |V_k| = 1$.

Let s & t be the vertex in V_1 & V_k (s - source, t - sink). Let $c(i,j)$ be the cost of edge $\langle i,j \rangle$. The multi stage graph problem is to find a minimum cost path from s to t.

Each set V_i defines a stage in the graph. Because of the constraints on E, every path from s to t starts in stage 1, goes to stage 2, then to stage 3 and eventually to stage k.

✓ → Optimal substructure : Every s to t path is a result of a sequence of $k-2$ decisions. The ith decision involves determining which vertex in V_{i+1}, $1 \leq i \leq k+2$, is to be on the shortest path. cost (i,j) be the cost of the path which is minimum cost and exists between vertex j in V_i to vertex t.

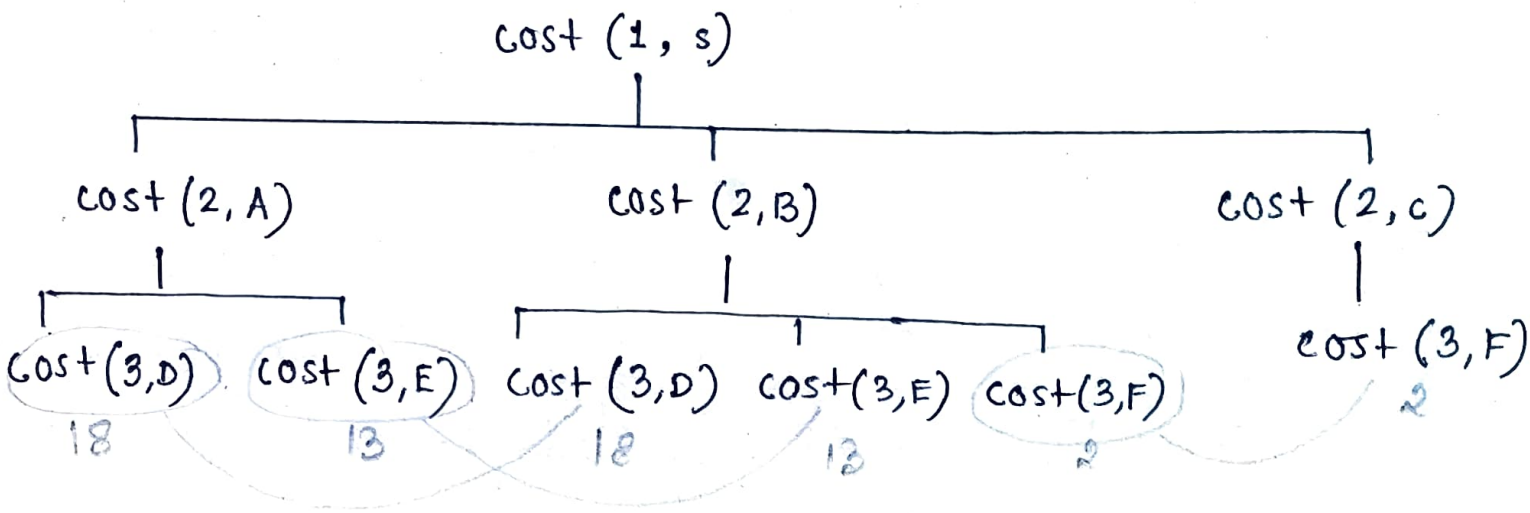


We obtain

✓ $$\text{cost}(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{ c(j,l) + \text{cost}(i+1, l) \}$$

→ Overlapping subproblems :

For previous example,

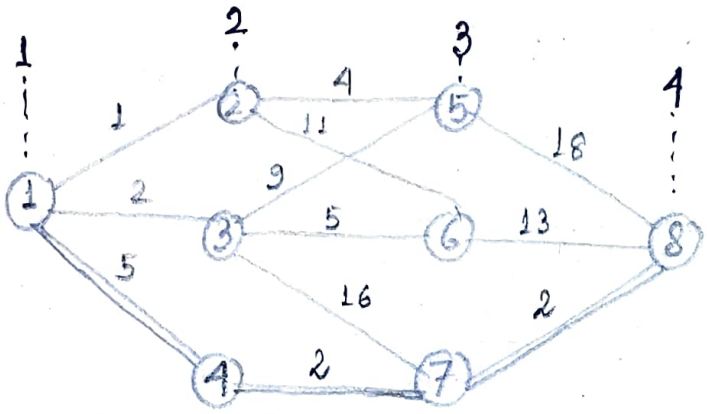


Base case : $cost(k-1, i) = c(i, t)$ if $\langle i, t \rangle \in E$

$cost(k-1, i) = \infty$ if $\langle i, t \rangle \notin E$

Simple recursive solution will have exponential running time.

✓ → Here, # unique subproblems = # nodes.



$k = 4$.

= shortest path

→ Bottom-up DP approach.

We need to find the costs starting from k-2 level (as from k-1 level to t it is base case and known).

$$cost(k-2, j) = cost(2, 2) = \min \{ c(j, l) + cost(i+1, l) \}$$
$$l \in V_{i+1}$$
$$\langle j, l \rangle \in E.$$

$$= \min \{ c(2,5) + cost(3,5), \; c(2,6) + cost(3,6) \}$$

$$= \min \{ 4 + 18, \; 11 + 13 \}$$

$$= 22$$

$$cost(2,3) = \min\{\, c(3,5) + cost(3,5),\ c(3,6) + cost(3,6),\ c(3,7) + cost(3,7)\,\}$$

$$= \min\{\, 9+18,\ 5+13,\ 16+2\,\} = 18$$

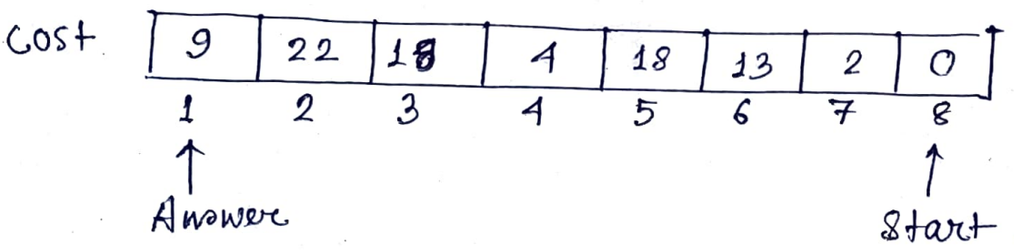
$$cost(2,4) = \min\{\, c(4,7) + cost(3,7)\,\}$$

$$= 2 + 2 = 4.$$

$$cost(1,1) = \min\{\, c(1,2) + cost(2,2),\ c(1,3) + cost(2,3),\ c(1,4) + cost(2,4)\,\}$$

$$= \min\{\, 1+22,\ 2+18,\ 5+4\,\} = 9. \text{ (Answer).}$$

We can use array to save the cost values.



✓ Time complexity - $O(E) = O(V^2)$

(Whereas if we had used Dijkstra's, $O(E \lg V)$.)

Dynamic Programming (Contd)

* Binary Knapsack Problem (0-1 KP)

Given a set of n items & a knapsack, with

p_j = profit of item j

w_j = weight of item j.

c = capacity of the knapsack;

Select a subset of the items so as to

$$\text{maximise } z = \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to } \sum_{j=1}^{n} w_j x_j \leq c,$$

$$x_j = 0 \text{ or } 1, \quad j \in N = \{1, \ldots, n\}$$

Where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

Assume, p_j, w_j, c are +ve integers.

$$\sum_{j=1}^{n} w_j > c$$

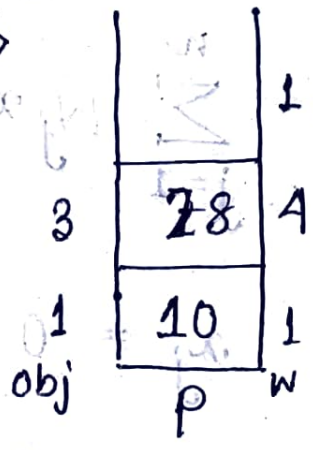
$$w_j \leq c \quad \text{for } j \in N$$

→ Simple greedy method (greedy upon profit/weight) fails to provide the optimal solution.

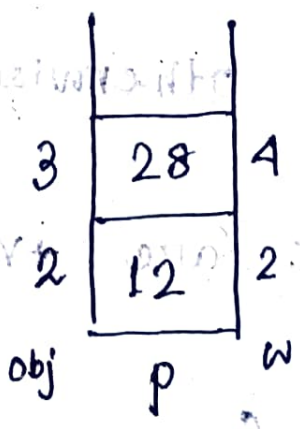
eg. capacity, c = 6

Objects	1	2	3
Weight, w	1	2	4
profit, p	10	12	28
p/w	10	6	7.

Greedy upon p/w ⇒



Better solution ⇒

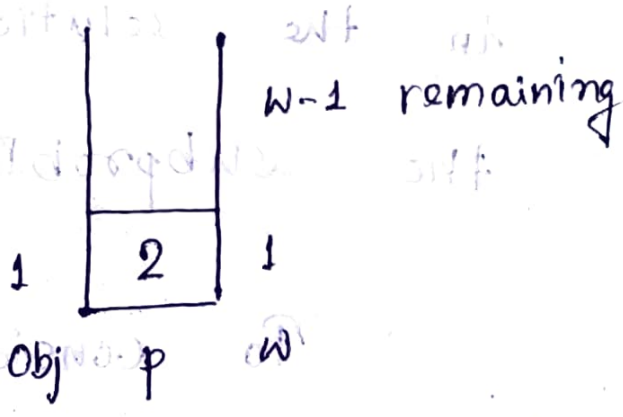


→ Why greedy method will fail?

Say, capacity of knapsack is w.

Object	1	2
P	2	w
W	1	w
P/W	2	1.

Acc. to greedy method,



Whatever be the value of w, greedy method will yield optimal profit to be 2. But, for any value of $w > 2$, if we put obj 2 in the knapsack it will give better yield.

Hence, greedy method surely fails in this scenario.

→ Optimal substructure

Let i be the highest numbered item in an optimal solution S for remaining weight W & items $1, \ldots, n$. Then $S' = S - \{i\}$ must be an optimal solution for $W - w_i$ weight & items $1, \ldots, i-1$ & the value of the solution S is p_i (when included in the solution) plus the value of the subproblem solution S'.

To consider all subsets of items, there can be 2 cases for every item : (I) item included in the optimal subset (II) not included.

to decide object's inclusion

Now, following cases arise —

When included —

i) profit of the object is added with optimal profit for remaining objects. $i > 0, w \geq w_i$

When not included —

ii) optimal profit for the next $i-1^{th}$ object yields better solution. Subproblem solution is better.

iii) Weight of the object w_i is greater than the remaining capacity w.

$$w_i > w.$$

Define $c[i, w]$ to be the value of the solution for item's $1, \ldots, i$ & maximum weight w. Then,

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } w_i > w \quad \text{(iii)} \\ \max \{ p_i + c[i-1, w-w_i], \; c[i-1, w] \} & \text{if } i > 0 \text{ \& } w \geq w_i \quad \text{(i)} \end{cases}$$

(ii)

Last case says value of a soln for i items either includes item i in which case it is p_i plus a subproblem soln for $i-1$ items & the weight excluding w_i - or - doesn't include item i, in which case it is a subproblem soln for $i-1$ items and same weight.

We need to find $c[n, C]$ for n objects & capacity C.

→ Subproblems

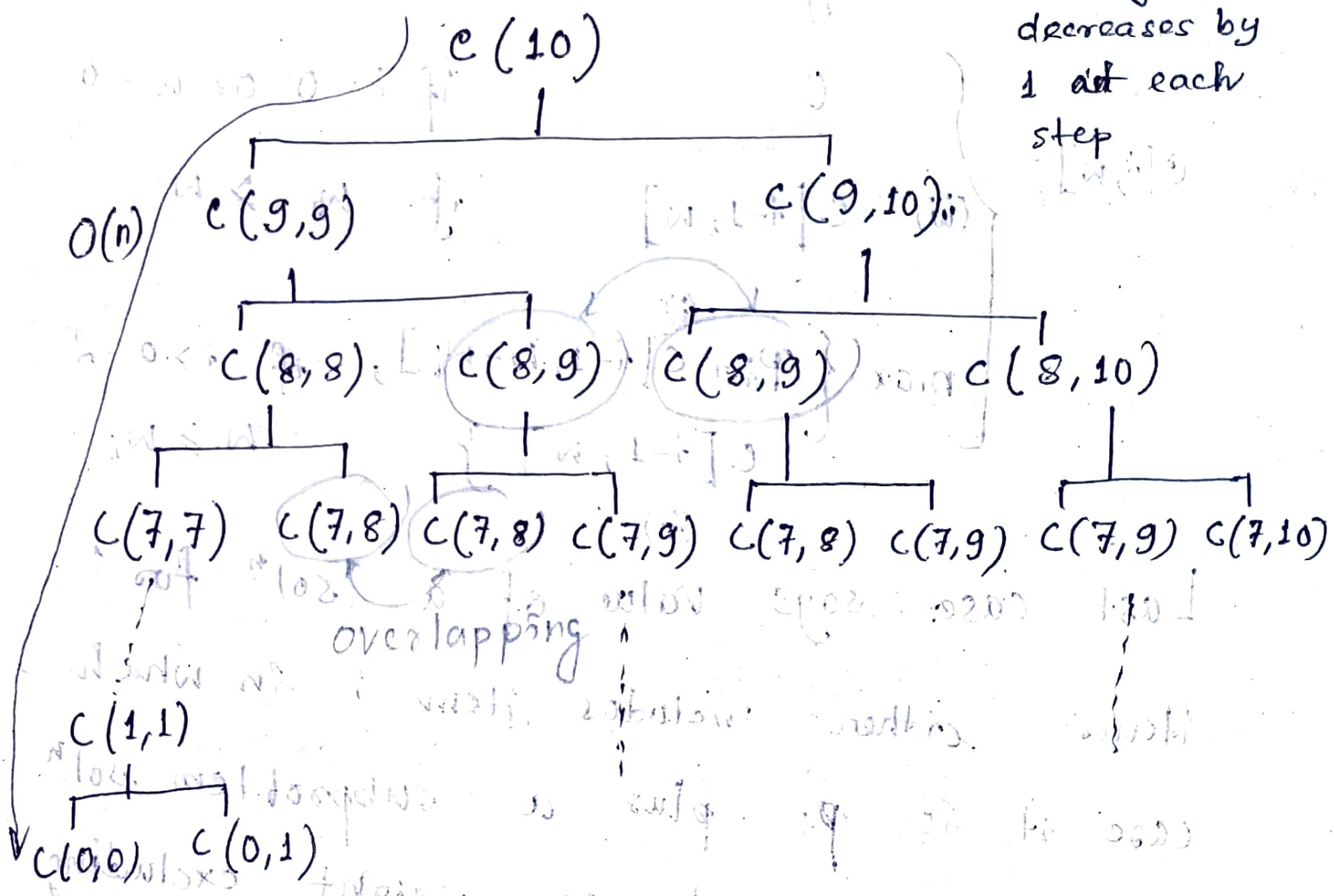
eg. $n = 10$ objects

capacity $c = 10$

assume weight of each object = 1 (worst case)

Function call recursion tree.

weight decreases by 1 at each step



Depth $O(n)$ $SC = O(n)$

nodes $O(2^n)$.

So, recursive implementation will take $O(2^n)$.

DP is better approach as we have overlapping subproblems in the tree.

No. of unique subproblems we have

$= 10 \times 10.$

(# of objects) x (capacity)

$n \times c.$

So, we can use a table of size $n \times c$ to memoize the results of the unique subproblems & reuse them.

→ Example of DP solution.

$c = 6$

	1	2	3
W	1	2	4
P	10	12	28

n \ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28	38	40

→ W = 3 so we can include both 1 & 2

Ans. C(3,6)

$C[1,1] = \max\{10 + C[0,0],\ C[0,1]\}$

$= \max\{10 + 0,\ 0\} = 10$

$C[1,2] = \max\{10 + C[0,1],\ C[0,2]\} = 10$

$C[2,1] = C[1,1]$ [as $(W_2 = 2) > (W = 1)$]

Once we reach the capacity where all the objects can be included & if we are examining only those objects then profit is going to be sum of all objects.

✓ Time complexity $O(nc)$.

SC $O(nc)$

If $c = O(2^n)$ then brute-force is better.

| Brute-force $O(2^n)$
| DP $O(n \cdot c) = O(n \cdot 2$

→ Bottom-up pseudocode

Dynamic_binary-KS (p, c, n, W) (c: capacity)

```
for j = 0 to c                    | w = 0 to c
    T[0, j] = 0                   |   c[0, w] ← 0

for i = 1 to n
{
    T[i, 0] = 0
    for k = 1 to c                | for w = 1 to W
    {
        if (w_i ≤ k)
        {
            if (p_i + T[i-1, c-w_i] > T[i-1, c])
                T[i, c] = p_i + T[i-1, c-w_i]
            else
                T[i, c] = T[i-1, c]
        }
        else
            T[i, c] = T[i-1, w]
    }
}
```

TC - $O(nc)$

SC - $O(n)$

* Subset Sum Problem

Given a set of non-negative integers and a value 'sum', determine if there is a subset of the given set with sum equal to given 'sum'.

Analysis same as 0/1 KS problem.

Let isSS (int set[], int n, int sum) be the function to find whether there is a subset of set[] with sum equal to sum (n is #elems in set).

The isss problem can be divided into 2 subproblems:

a) Include last elem,
   recur for n-1 elems,
   sum = sum - set[n-1]

b) Exclude last elem,
   recur for n-1 elems
   sum = sum

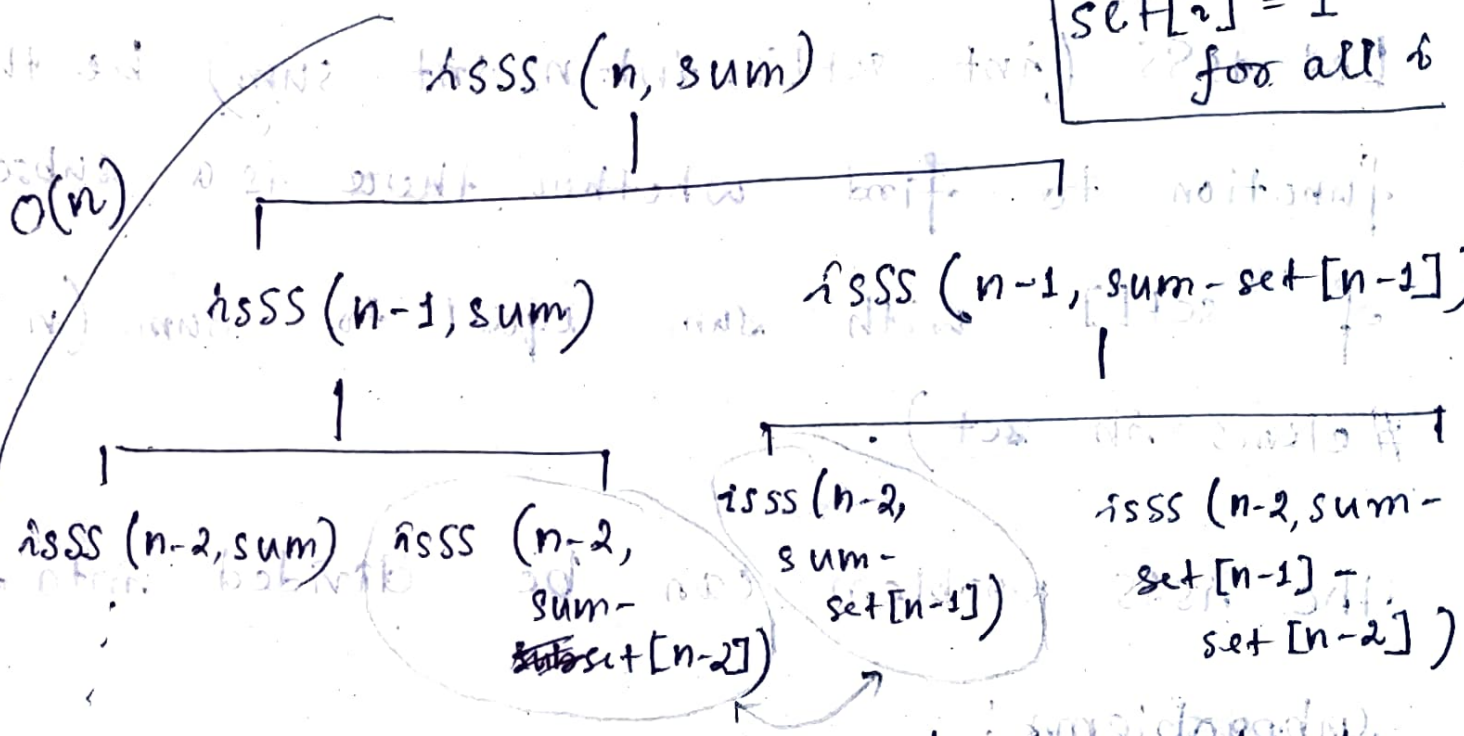
} ✓ Optimal substructure

Recursive formula for isSS()

Base cases

$$\begin{cases} isSS(set, n, sum) = false, & \text{if } sum > 0 \text{ and } n == 0 \\ isSS(set, n, sum) = true, & \text{if } sum == 0 \end{cases}$$

assume set[i] = 1 for all i



overlapping subprobs

Recursive step:

$$isSS(set, n, sum) = isSS(set, n-1, sum) \;||\; isSS(set, n-1, sum - set[n-1])$$

Recursive method will have $O(2^n)$ running time complexity.

We use DP.

→ Example of DP solution.

{6, 3, 2, 1}. W = 5. using recursive formulation

(1 2 3 4)

n \ Sum	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T	F	T	T	F	T
4	T	T	T	T	T	T Ans.

possible
3+2=5

isSS(2,3) = isSS(1,0) || isSS(1,3)

TC - O(nW) ← Sum. (pseudo-polynomial)

SC - O(n)

if W is very large, use brute-force O(2^n)

→ 0/1 KS & Subset-sum are NP complete problem. No polynomial time algorithm is there for them.
(pseudo-polynomial time algo)

W = O(2^n)

⇒ T = O(n · 2^n)
using DP

But, bruteforce O(2^n)

* The Traveling Salesman Problem.

A salesman is required to visit once & only once each of n different cities starting from a base city & returning to this city. What path minimises the total distance travelled by the salesman? (Min cost Hamiltonian cycle in a complete graph)

→ Optimal substructure.

This is a multistage decision problem. Since the tour is a round trip, fix origin at some city, say 0. Suppose that at a certain stage of an optimal tour starting at 0 one has reached a city i & there remain k cities $j_1, j_2, \ldots, j_k$ to be visited before returning to 0. Then it is clear that, the tour being optimal, the path from i through $j_1, \ldots, j_k$

to be visited ~~before~~ in some order & then to 0 must be of minimum length; for, if not the entire tour could not be optimal, since its total length could be reduced by choosing a shorter path from 1 through $j_1, j_2, \ldots, j_k$.

→ Recursive formulation.

C_{ij} - edge cost of $\langle i,j \rangle$ edge.

$C_{ij} > 0 \quad \forall\, i,j$

$C_{ij} = \infty$ if $\langle i,j \rangle \notin E$.

$G = (V, E)$

Let $g(i,S)$ be the length of a shortest path starting at vertex i, going through all vertices in S, & terminating at vertex 1 (origin - tour starts & ends at 1). Every tour consists of an edge $\langle 1,k \rangle$ for some $k \in V - \{1\}$ & a path from k to vertex 1. Path from

K to 1 goes through each vertex in $V - \{1, k\}$ exactly once.

Function $g(1, V-\{1\})$ is the length of an optimal salesman tour.

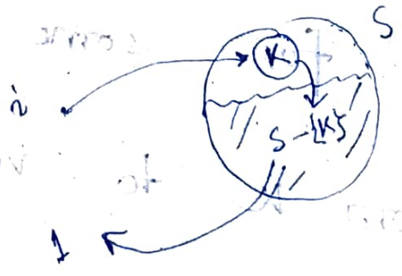
$$g(1, V-\{1\}) = \min_{2 \le k \le n} \{ c_{1k} + g(k, V-\{1,k\}) \}$$

Generalizing (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, S-\{j\}) \} \quad S \neq \phi$$

$$g(i, \phi) = c_{i1}, \quad 1 \le i < n. \quad S = \phi.$$

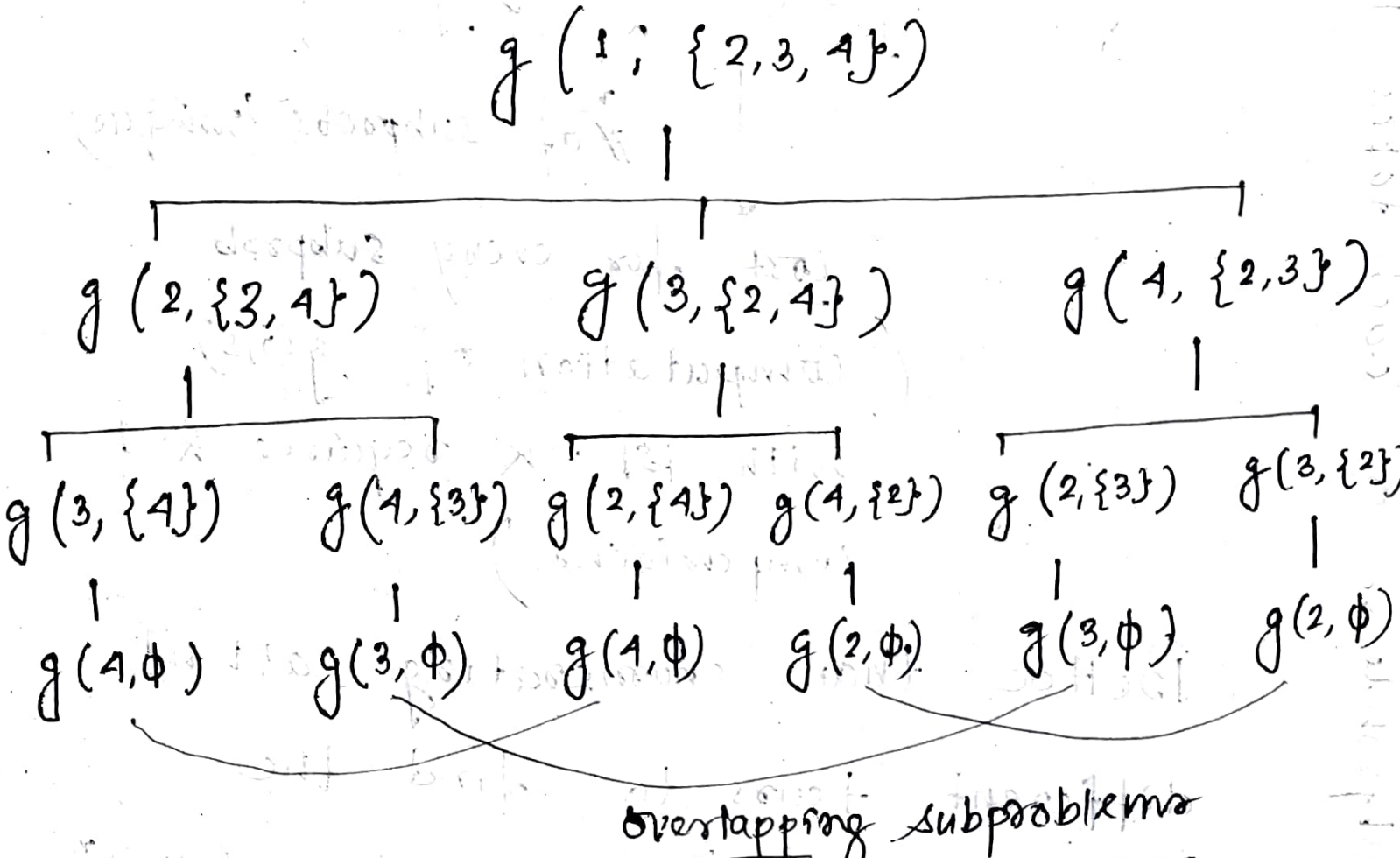
We can obtain $g(i, S)$ for all S of size 1. Then we can obtain $g(i, S)$ for S with $|S| = 2$ & so on.



Recursion tree example.

$V = \{1, 2, 3, 4\}$. 1 - origin

Recursion calls



overlapping subproblems

Number of unique subproblems. (recursion calls)

Let N be the number of $g(i, S)$'s that have to be computed before computing $g(1, V-\{1\})$. For each value of $|S|$ there are $n-1$ choices for i. The number of distinct sets S of size k not including 1 & i is $\binom{n-2}{k}$. Hence,

$$N = \sum_{k=0}^{n-2} \underbrace{(n-1)}_{\text{for } i} \underbrace{\binom{n-2}{k}}_{\text{for } S} = (n-1)\,2^{n-2}$$

⇒ At most $2^n \cdot n$ subproblems & each takes linear time.
TC $\in O(n^2 2^n)$

→ Using bottom-up DP approach, time complexity will be

$$\boxed{\Theta(n \cdot n2^n)}$$

n : # of subprobs (unique)

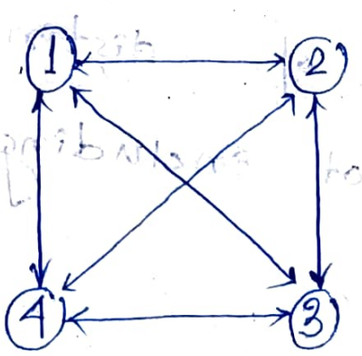
$n2^n$: cost for every subprob.
(computation of $g(i,S)$ with $|S| = k$ requires $k-1$ comparisons)

Better than enumerating all $n!$ different tours to find the optimal path. $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

→ Drawback is space needed —

$$\boxed{O(n2^n)}$$

Example.



cost matrix

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$g(2, \phi) = C_{21} = 5$

$g(3, \phi) = C_{31} = 6$

$g(4, \phi) = C_{41} = 8$

$|S| = 1$

$g(2, \{3\}) = C_{23} + g(3, \phi) = 15$

$g(3, \{2\}) = 18$ $\quad$ $g(3, \{4\}) = 20$

$g(4, \{2\}) = 13$ $\quad$ $g(4, \{3\}) = 15$

$g(2, \{4\}) = 18$

$|S| = 2$

$$g(2, \{3,4\}) = \min \begin{Bmatrix} C_{23} + g(3, \{4\}), \\ C_{24} + g(4, \{3\}) \end{Bmatrix} = 25$$

$g(3, \{2,4\}) = 25$ $\quad$ $g(4, \{2,3\}) = 23$

Now, $g(1, \{2,3,4\}) =$

$$\min \begin{Bmatrix} C_{12} + g(2, \{3,4\}), C_{13} + g(3, \{2,4\}), \\ C_{14} + g(4, \{2,3\}) \end{Bmatrix}$$

$= 35.$ (Ans - optimal tour cost)

A tour of this cost can be constructed if we retain with each $g(i,S)$ the value of j the minimises the g fun^n.

toprint (DAA)

Held-Karp Algorithm.

#Called

DP pseudocode.

$$C(S,j) = \min_{i \in S,\, i \neq j} \{ C(S-\{j\}, i) + d_{ij} \}$$

TRAVELING_SALESMAN_DP

$C(\{1\}, 1) = 0$

for $s = 2$ to n

  for all subsets $S \subseteq \{1, 2, \ldots, n\}$ of size s & containing 1:

    $C(S,1) = \infty$

    for all $j \in S$, $j \neq 1$:

      $c(S,j) = \min \{ c(S-\{j\}, i) + d_{ij} : i \in S,\, i \neq j \}$

  return $\min_j C(\{1, \ldots, n\}, j) + d_{j1}$

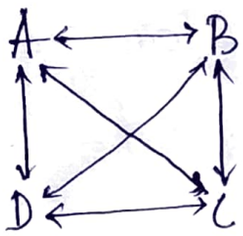
SC $O(2^n/\sqrt{n})$ when storing only subsets of size s & $s-1$.

TC $O(n^2 2^n)$

SC $O(n 2^n)$ too bad.

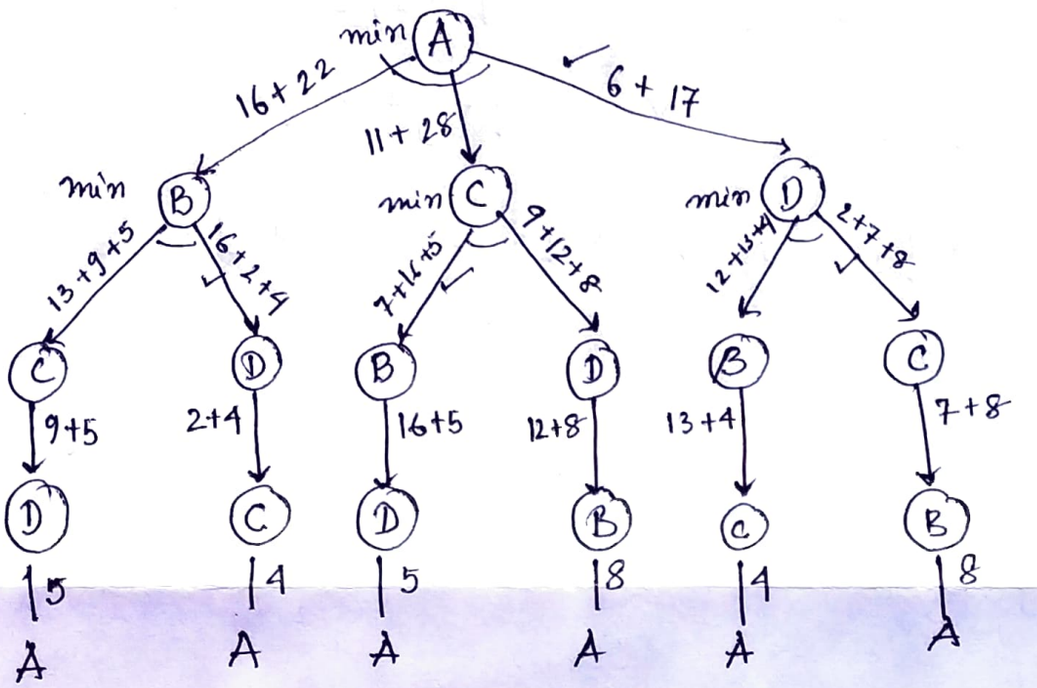
- We are not using any table here like previous DP implementations. Using bottom-up approach we are using previously calculated values.

* TSP



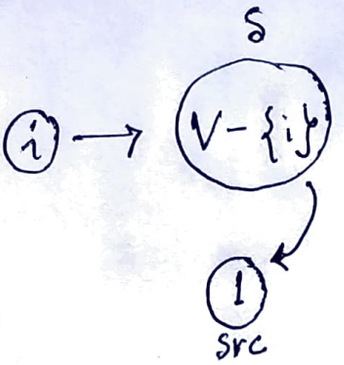
	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0

src - A



Min hamiltonian cycle

A - D - C - B - A



$g(i,s)$ be the cost for the path

cost of edge

$g(i,s) = c_{i1}$, when $S = \phi$

$= \min_{k \in S} \{ c_{ik} + g(k, V - \{k\}) \}$

We need to find $g(1, \{2,3,4\})$.

Q Subset Sum Problem: $X[i,j]$, $1 \le i \le n$, $0 \le j \le W$

true, iff subset of $\{a_1 \ldots a_i\}$ whose elems sum to j.

Which is valid for $2 \le i \le n$ & $a_i \le j \le W$?

a) $X[i,j] = X[i-1, j] \vee X[i, j-a_i]$.

b) $X[i,j] = X[i-1, j] \vee X[i-1, j-a_i]$

c) $X[i,j] = X[i-1, j] \vee X[i, j+a_i]$

d) $X[i,j] = X[i+1, j] \vee X[i-1, j-a_i]$

* All-pairs shortest paths

We can solve an all-pairs shortest-paths problem by running a single source shortest-paths algorithm $|V|$ times, once for each vertex as the source.

(1). If all edge weights are non negative we can use Dijkstra's algorithm.

Implementation	Running time
Linear array impl^n of min-priority queue	$O(V^3 + VE) = O(V^3)$
Binary min heap impl^n of min-prio queue	$O(VE \lg V)$ ✓
Min-prio queue with a Fibonacci heap	$O(V^2 \lg V + VE)$.

(2) If the graph has -ve weight edges, we must run Bellman-Ford algo once for each vertex. Running time will be $O(V^2E)$ ✓ which on a dense graph is $O(V^4)$.

-ve w edges may be present

Floyd - Warshall algorithm

Solves the problem in $O(V^3)$ time.

- Considers intermediate ~~values~~ vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1, v_l.

- Consider a shortest path $i \overset{p}{\rightsquigarrow} j$ with all intermediate vertices in $\{1, 2, \ldots, k\}$.

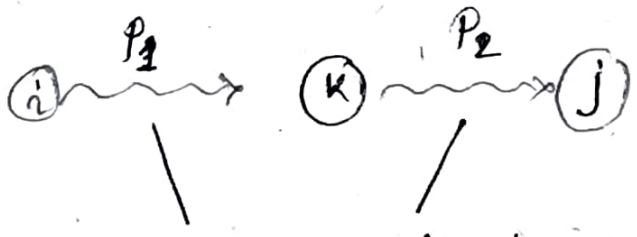
Algo exploits a relationship between path p & shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Relationship depends on whether or not k is an intermediate vertex of path p.

1. If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, 2, \ldots, k-1\}$.

Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the

set $\{1, 2, \ldots, k\}$.

2. If k is an intermediate vertex,



all intermediate vertices in $\{1, 2, \ldots, k-1\}$

as k is not an intermediate vertex of paths p_1, p_2

Recursive formulation.

$d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from i to j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge. So, $d_{ij}^{(0)} = w_{ij}$.

Now, recursively

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) & \text{if } k \geq 1 \end{cases}$$

Because for any path, all intermediate vertices are in the set $\{1, \ldots, n\}$, the matrix $(d_{ij}^{(n)})$ gives the final answer.

$$d_{ij}^{(n)} = \delta(i,j) \quad \forall\, i,j \in V$$

{ shortest distance

- Bottom-up DP approach

PLOYD_WARSHALL (W) ~ weight matrix

n = W. rows

$D^{(0)} = W$

for k = 1 to n

Let $D^{(k)} = (d_{ij}^{(k)})$ be a new n x n matrix

for i = 1 to n

for j = 1 to n

O(1) ——— $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

return $D^{(n)}$.

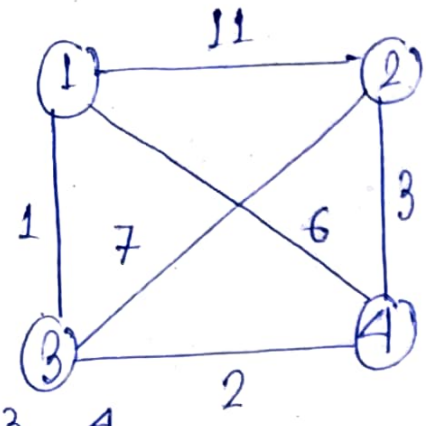
✓ TC — $\Theta(n^3)$

SC — $O(n^3)$ here ↑

can be reduced to $O(n^2)$ by using 2

matrices of $O(n^2)$ & reusing them to store future matrix. As any time, we just use the previously computed matrix to find current matrix.

Example



RBR

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$$D^1 = \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix}$$

$$\begin{cases} \dfrac{1-2 \quad (11)}{(1-1)+(1-2)} \quad \begin{cases} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} \\ \downarrow \qquad \downarrow \\ 0 \qquad 11 \\ \qquad (11) \end{cases}$$

$$D_2 = \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix}$$

$$\begin{cases} \dfrac{2-3 \quad (7)}{(2-2)+(2-3)} \\ 0 + 7 \\ = (7) \end{cases}$$

$$D_3 = \begin{bmatrix} 0 & 8 & 1 & (3) \\ 8 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix}$$

$$\begin{cases} \dfrac{1-4 \quad (6)}{(1-3)+(3-4)} \\ 1 + 2 \\ = (3) \ \text{min} \end{cases}$$

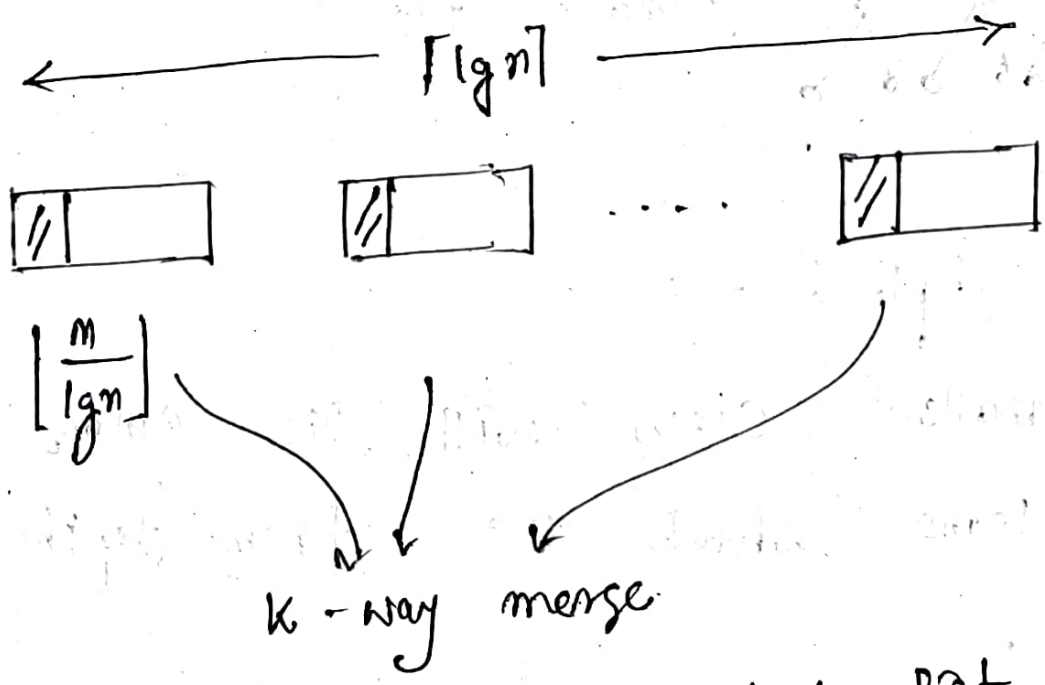
$$D_4 = \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & (5) & 3 \\ 1 & 5 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix}$$

$$\begin{cases} \dfrac{2-3 \quad (7)}{(2-1)+(1-3)} \\ 3 + 2 \\ = (5) \ \text{min} \end{cases}$$

Ans →

Heaps.

G'5 There are $\lceil \lg n \rceil$ sorted lists of $\lceil n/\lg n \rceil$ elems each. TC of producing a sorted list of all these elems (using a heap) is —



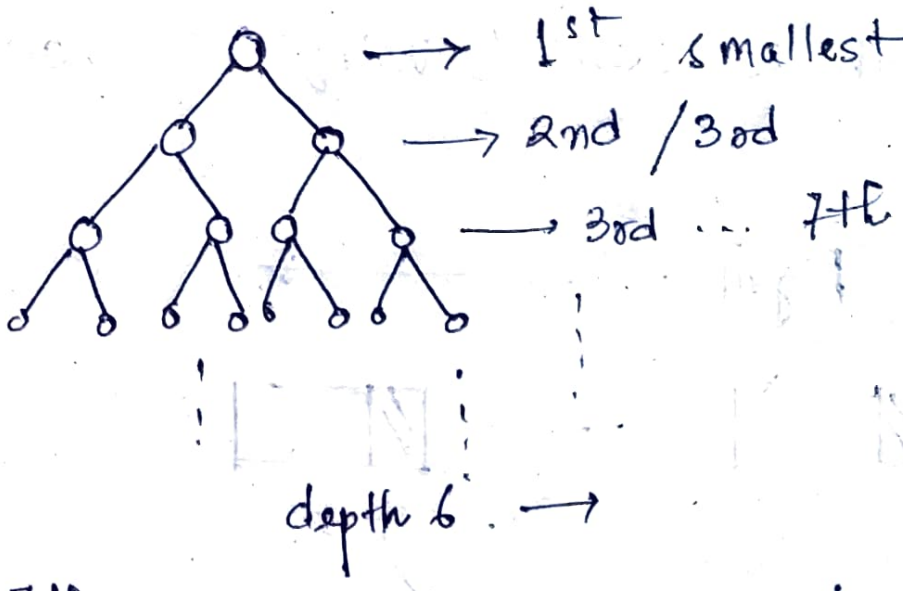
For each elem in a sorted list we have to do extract_min from the heap containing the elem & also insert_new into the new sorted list. Both, takes $O(\lg k)$ where k is # elems in heap, (take each elem from ⌈lg n⌉ sorted lists & build a heap; then extract_min)

$k = \lg n$.

~~And there are n~~ We need to do this for n total elements.

$$TC = O(n \lg k)$$
$$= O(n \lg \lg n).$$

Q. In a heap with n elems with the smallest at the root, the 7^{th} smallest elem can be found in time —



7th smallest elem will lie among the elems which are from depth 0 to 6.

depth	# elems
0	1
1	2
2	4
3	⋮
4	⋮
5	⋮
6	64
	$+(2^7-1)$

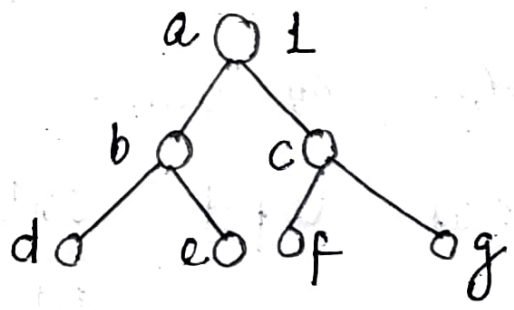
$= 127$ elems in total

So, 7th smallest will be among these 127 elems.

So, TC $= \Theta(1)$ as finding the elem is indep of n.

Q. # possible minheaps containing each value from $\{1, 2, 3, 4, 5, 6, 7\}$ exactly once is:

* a - 1



2 @ b or c

↓

→ If 2 @ b then @ d & e there are (5) & (4) possibilities. Now, 3 numbers left.

⇓

min will be at c

For f & g we have (2) & (1) possibilities.

So, $2 \times (5 \times 4) = 40$

→ If 2 @ c same way 40

So, ans = 80.

✓ Aliter: ① Smallest at root 'a'.

② For left subtree 

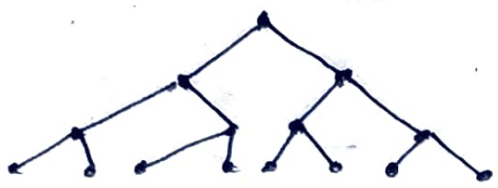
, pick any 3 elems $({}^6C_3)$. Among those 3, the min at b with 100% probability. Remaining 2 elems in 2! ways.

$= {}^6C_3 \times 2!$

③ for right subtree

, similarly ${}^3C_3 \times 2!$ ways. ⇒ Total = $\binom{6}{3} \times 2! \times \binom{3}{3} \times 2! = 80$

eg Min-heaps possible for 15 distinct elems.



1. Min @ root.
2. For left subtree, choose 7 elems among 14 elems. $(^{14}C_7)$.
Now, the 7 elems can be structured in $\binom{6}{3} \times 2! \times \binom{3}{3} \times 2!$ ways as before.

$$\Rightarrow \underbrace{\binom{14}{7} \times \left\{\binom{6}{3} \times 2! \times \binom{3}{3} \times 2!\right\}}_{X}$$

3. Remaining 7 elems, similarly

$$\underbrace{\binom{7}{7} \times \left\{\binom{6}{3} \times 2! \times \binom{3}{3} \times 2!\right\}}_{Y}$$

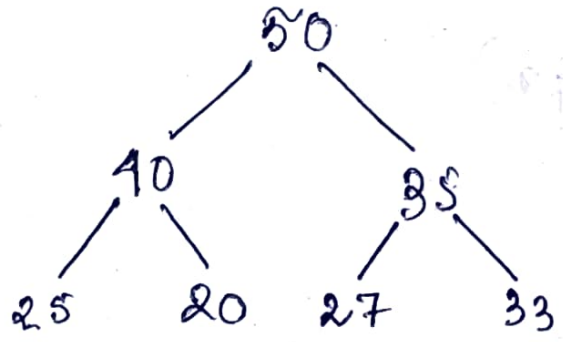
Total = X × Y

Q. An operation delete(i) for a binary heap is to be designed to delete the item in the ith node. Assume that the heap is implemented in an array & i refers to ith index. If heap has depth d, what is the TC to refix the heap efficiently after removal of elem?

a) $O(1)$ b) $O(d)$ but not $O(1)$

c) $O(2^d)$ but not $O(d)$

d) $O(d \cdot 2^d)$ but not $O(2^d)$

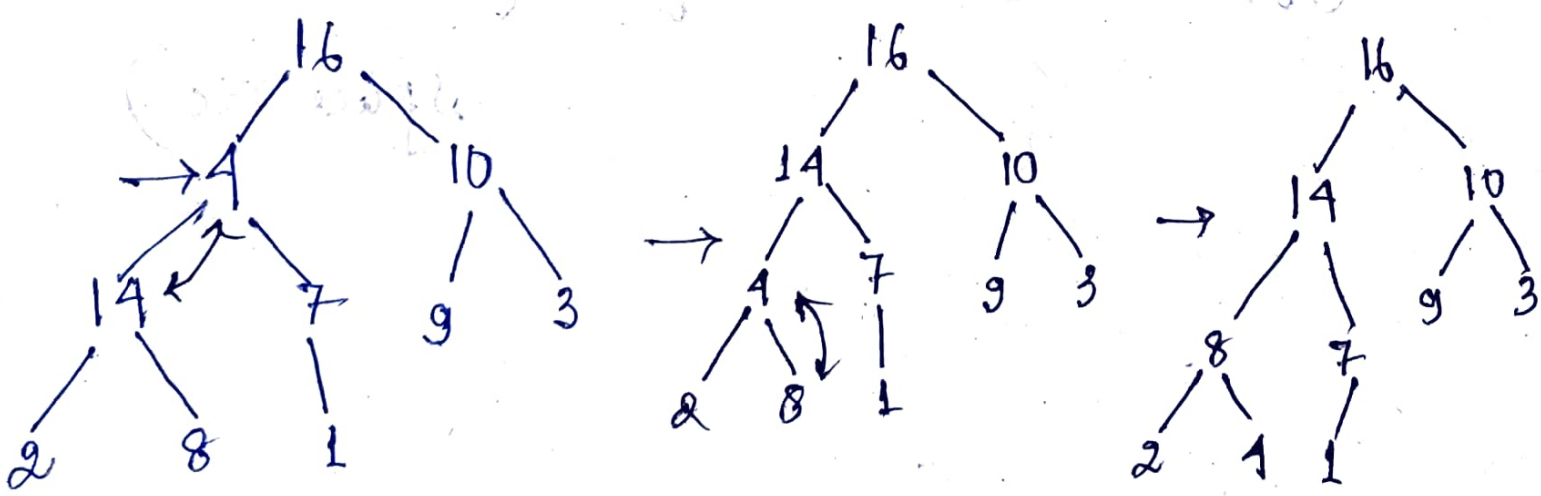


In worst case root (50) is deleted.
Heapify take $O(d)$
$d = \lg n$.

- delete(i) :

```
if A[i] < A[A.heap_size]
    Heap_increase_key (A, i, A[A.heap_size])
    A.heap_size -= 1
else
    A[i] = A[A.heap_size]
    A.heap_size -= 1
    max_heapify (A, i)
```

Q. Heapify. (max)



Hashing (Refer CLRS)

* Search times :

Unsorted array	O(n)
Sorted array	O(lg n)
Linked list	O(n)
Binary tree	O(n)
BST	O(n)
Balanced BST	O(lg n)
Priority Queue/ Min, max heap	O(n)

* Direct address table :

Array [1..n]

Insert element i in the index i of array.

Obvious disadvantages (size, unused space etc.)

- Dictionary operations (insert, search, delete)

Hashing

Using hashing, under reasonable assumptions the expected time to search for an element in a hash table is O(1).

Basic dictionary operations take O(1) time on the average.

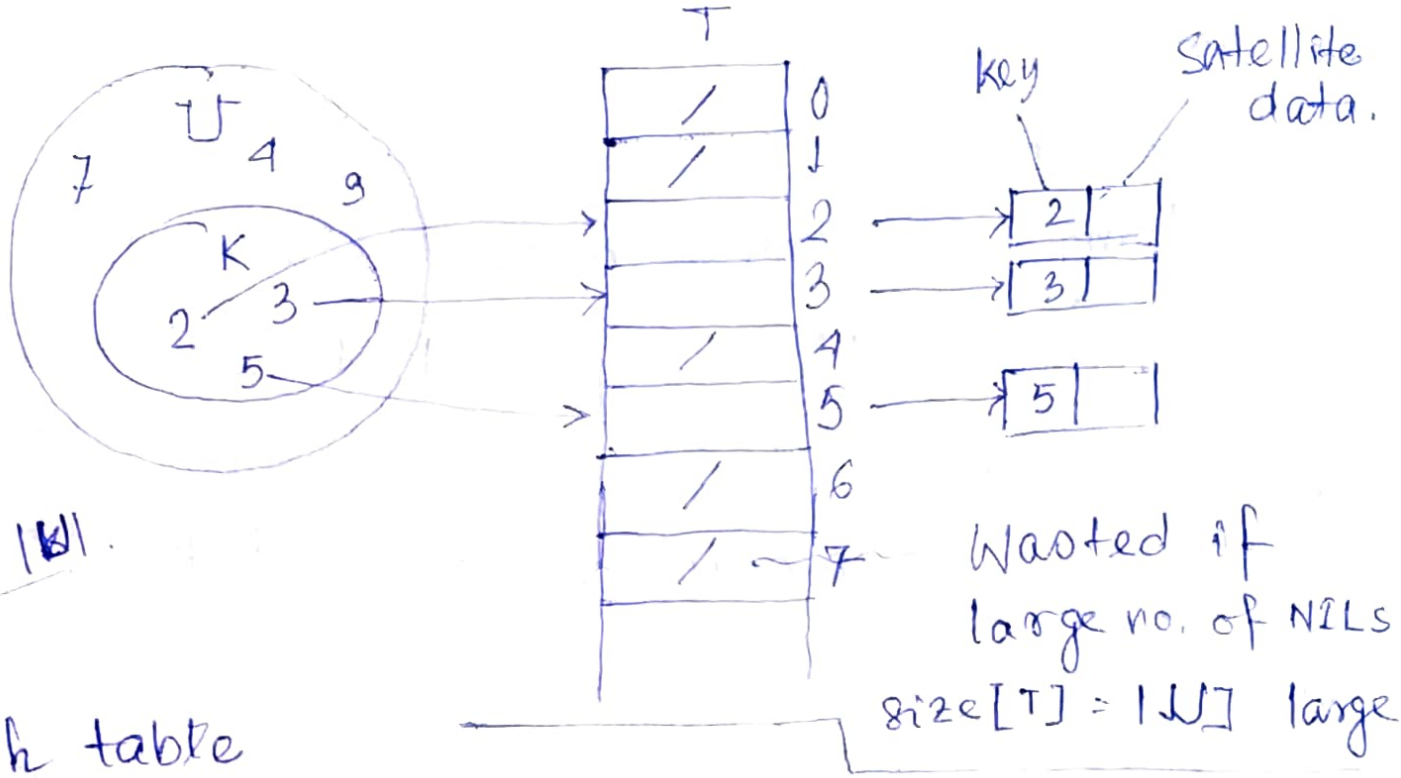
Perfect hashing can support 'searches' in O(1) worst case time, when the set of keys being stored is static.

- Direct Addressing: Works well when universe of keys is reasonably small. To represent dynamic set, we use an array (direct address table) denoted by T[0...m-1], in which each position or slot corresponds to a key in the universe U.

Elem with key k is stored in slot k.

Search	Insert	Delete
return T[k]	T[key[x]] ← x	T[key[x]] ← NIL

O(1) time required.



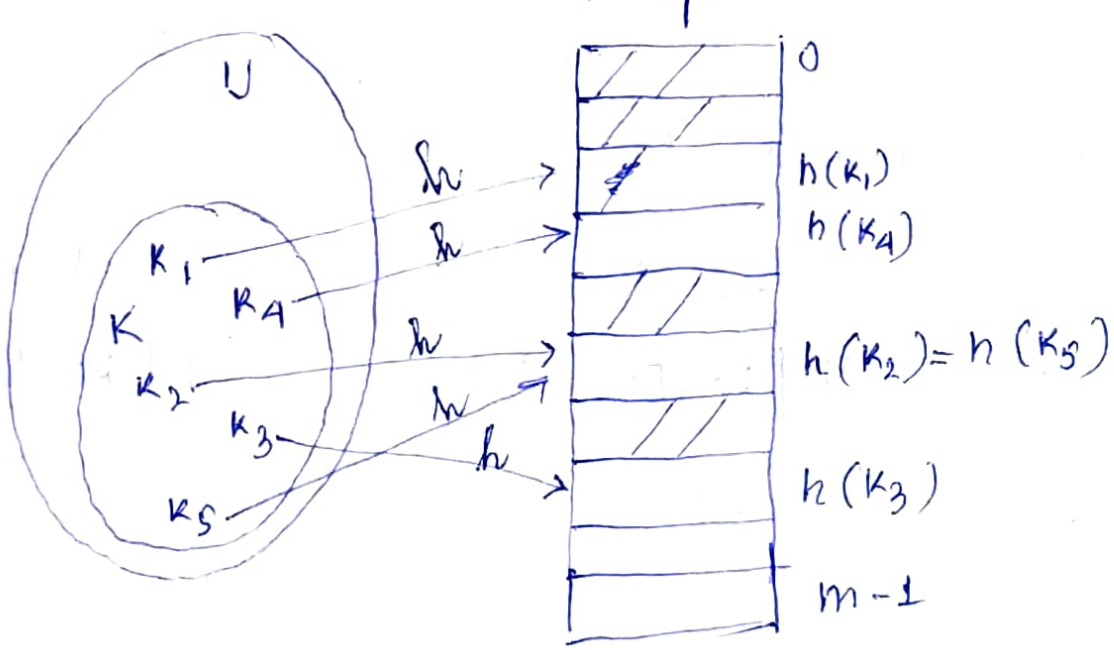
size[T] = |K|.

Wasted if large no. of NILs

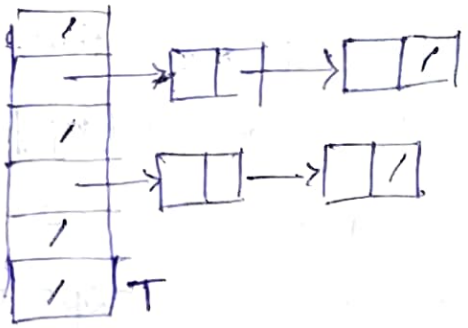
size[T] = |U| large

- **Hash table**

When set of keys stored in dictionary is much smaller than the universe U of all keys a hash table requires much less storage than a direct addr. table. In hashing elem. with key k is stored in slot h(k). h is hash function. $h : U \rightarrow \{0, 1, \ldots, m-1\}$. h(k) is the hash value of key k.



collision: 2 keys hash to same slot.



Collision resolution by chaining

If hash to same slot, put in LL.

insert (T, x) worst case O(1)

insert x at the head of list T[h(key[x])]

search (T, k) worst Θ(n)

search for an elem with key k in list T[h(k)]

delete (T, x) O(1) when doubly LL

delete x from the list T[h(key[x])]

Analysis by chaining (suc. & unsuc. searching)

Given hash table T with m slots that stores n elems. Load factor α for T as n/m. (avg. # of elems. stored in chain). (#items/# slots)

- Worst case - all n keys hash to same slot. (creating list of length n)
  Θ(n) + time to compute hash fn
- Avg - how well the hash fn h distributes the set of keys to be stored among m slots
  - Simple uniform hashing: Any given elem is equally likely to hash into any of the m slots, indep. of where any other elem has hashed to.

for $j = 0, 1, \ldots, m-1$, we denote the length of list $T[j]$ by n_j, so that $n = n_0 + n_1 + \cdots + n_{m-1}$. & avg. value of n_j is $E[n_j] = \alpha = n/m$. Assume $h(k)$ can be computed in $O(1)$. So, time reqd. to search for an elem with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$. (No. of elems in the list $T[h(k)]$ that are checked to see if their keys are equal to k.)

✓ • In hash table (chaining resolution), an unsuccessful search (no elem in the table has key k) takes expected time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

✓ • In hash table (chaining), successful search takes time $\Theta(1+\alpha)$ on the avg, under assumption of simple uniform hashing.

Hash table (Simple uniform) - 1

Collision resolution by chaining - Suc & Unsuc. search

- Proof (unsuccessful search $\Theta(1+\alpha)$) ✓

Any key k not already stored in the table is equally likely to hash to any of the m slots.

Expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length of $E[n_{h(k)}] = \alpha$. Thus the expected # of elems examined in an unsuccessful search is α. Total time required (computing $h(k) \to 1$) is $\Theta(1+\alpha)$.

- Proof (successful search $\Theta(1+\alpha)$) ✓

We assume that the elem being searched for is equally likely to be any of the n elems stored in the table. # of elems examined during a successful search for an elem x is one more than the # of elems that appear before x in x's list. (Prob. that a list is searched is proportional to the # of elems it contains.). Because new elems are placed at the front of the list, elems before x in the list were all inserted after x was inserted. Let x_i denote the ith elem inserted into the table, for $i = 1, 2, \ldots, n$ & $k_i = x_i.key$. For keys k_i & k_j we define the indicator RV $X_{ij} = I\{h(k_i) = h(k_j)\}$.

Under the assumption of simple uniform hashing, we have

$$Pr\{h(k_i) = h(k_j)\} = 1/m.$$

$$So, \quad E[X_{ij}] = 1/m.$$

Thus, expected # of elems examined in a suc. search =

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} E[X_{ij}]\right).$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right)$$

$$= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1 + \frac{1}{nm}\left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right)$$

$$= 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) = 1 + \frac{n-1}{2m}$$

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

Total time reqd. for a suc. search (+ computing $h(x)$) is

$$\Theta\left(1 + 1 + \alpha/2 - \alpha/2n\right) = \Theta(1+\alpha)$$

Interpretation: If $n = O(m)$, then $\alpha = n/m = O(1)$, which means that searching takes constant time on average.

Since, insertion takes $O(1)$ worst-, deletion also $O(1)$ worst case time when the lists are doubly LL., all dictionary operations take $O(1)$ time on avg.

* Hash Functions

Good hash f^n.

Ideally the hash f^n satisfies the assumption of simple uniform hashing. In practice, it's not possible to satisfy this assumption, as we don't know in advance the prob. dist. that keys are drawn from & the keys may not be drawn independently.

Often use heuristics based on the domain of the keys to create a hash f^n that performs well.

→ The Division Method

$h(k) = k \bmod m$.

$m = 20$, $k = 91 \Rightarrow h(k) = 11$.

good choice of m: A prime not too close to an exact power of 2

adv. fast, since requires just one division op^n

disadv. have to avoid certain values of m. ($m = 2^p$ for integer p, $h(k)$ is least sign. p bits of k

$k = 1100$ $m = 1000 = 2^3$

→ The Multiplication Method.

1. Choose const. A in the range $0 < A < 1$
2. Mult. key k by A.
3. Extract fractional part of kA.
4. Mult. the frac. part by m.
5. Take floor of the result.

$$h(k) = \lfloor m (kA \bmod 1) \rfloor \text{, where}$$

$$kA \bmod 1 = kA - \lfloor kA \rfloor = \text{frac. part of } kA.$$

Adv. Value of m is not critical
Disadv. Slower

Choosing A — $A \approx (\sqrt{5} - 1)/2$ (Knuth)

→ Universal Hashing

Different hash f^n's each time.

Consider a finite collection H of hash functions that map a universe U of keys into the range $\{0, 1, .., m-1\}$. H is univ if for each pair of keys $k, l \in U$, where $k \neq l$, # of hash f^n's $h \in H$ for which $h(k) = h(l)$ is $\leq |H|/m$.

So, H is universal if ~~for each~~ with a hash f^n h chosen randomly from H,

the probability of a collision between 2 different keys is no more than the chance $1/m$ of a collision if ~~$h(k) \neq h(l)$~~ 2 slots were chosen randomly & independently.

Using chaining & universal hashing on key k: if k is not in the table, the expected length $E[n_{h(k)}]$ of the list that k hashes to is $\leq \alpha$.

If k is in the table, the expected length $E[n_{h(k)}]$ of the list that holds k is $\leq 1+\alpha$.

So, using chaining & universal hashing, the expected time for each Search op^n is $O(1)$.

- **Open-Addressing** An alternative to chaining for handling collisions.

Idea:
- Store all keys in hash table itself.
- Each slot contains either a key/NIL
- To search for key k:
  - compute $h(k)$ & examine slot $h(k)$. Examining a slot is known as ~~probing~~ **probe**.
  - If slot $h(k)$ contains key k, search is suc. If NIL, unsuc.

- 3rd possibility: slot $h(k)$ contains a key that is not k. We compute the index of some other slot, based on k & on which probe we're on.

- keep probing until we find key k (succ) or we find a slot holding NIL (Unsu

- $h : U \times \underbrace{\{0, 1, \cdots, m-1\}}_{\text{probe no.}} \rightarrow \{0, 1, \cdots, m-1\}$ slot no.

To insert, act as if searching, → insert at first NIL

Formula in open addressing:

$h(k,i)$ Initially $i = 0$.

Now, $h(k, 0)$ will generate a number among $\{0, 1, \cdots, m$

$h(k,0) = 2$ (say)

If 2nd loc^n is NIL we can insert. Otherwise collision & then we find $h(k, 1) = 7$ (say)

Now repeat.

Computing probe sequences

a) Linear Probing

$h_L'(k,i) = \{h(k) + i\} \bmod m$

$h_L = [h(k) + i] \,\%\, 13$ say

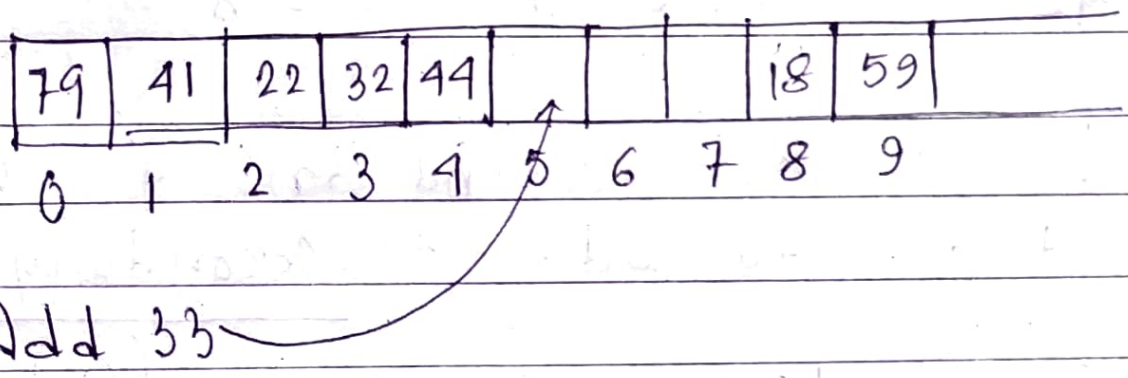
Auxiliary $h(k) = k \,\%\, 13$

If collision occurs, colliding elem. will be forced to be placed at the next circularly available slot.

eg: $h(k) = k \bmod 10$. ($h_L(k) = (h(k) + i) \% 10$

Keys (18, 41, 22, 32, 44, 59, 79)

hash (8, 1, 2, 2, 4, 9, 9) $h(k)$



Linear probing suffers with primary clustering : long runs of occupied sequences build up. And long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with Pr $(i+1)/m$.
Avg. search & insertion time increase.

b) Quadratic Probing

$$h_0(i) = (h(k) + c_1 i + c_2 i^2) \% m$$

$c_1, c_2 \neq 0$

$h(k) = k \% m$

$$(hash(key) + i^2) \bmod m$$

eg. $c_1 = 0, \quad c_2 = 1$

$h(x) = x \bmod 10.$

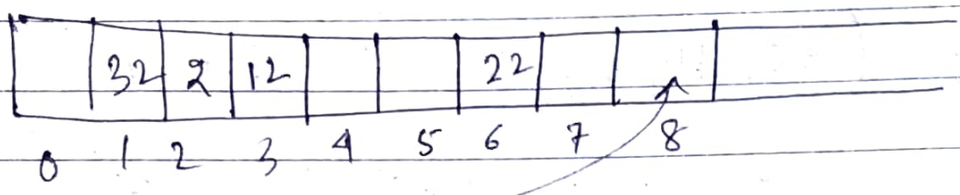
Keys (2, 12, 22, 32)

hash (2, 2, 2, 2)

$2 + 1^2 = 3$

$2 + 2^2 = 6$

$2 + 3^2 = 11 \% 10 = 1$



Add 42. $2 + 4^2 = 18 \% 10 = 8$

- May be jumping around in the array while trying to add, → secondary clustering
- 2 distinct keys have same $h(k)$ value then they have same probe sequence.

c) Double Hashing :

2 aux. hash fn's h_1 & h_2.

h_1 gives initial probe

h_2 gives remaining probes

$$h(k, i) = [h_1(k) + i\, h_2(k)] \bmod m$$

Must have $h_2(k)$ be relatively prime to m (no factors in common other than 1) in order to guarantee that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$

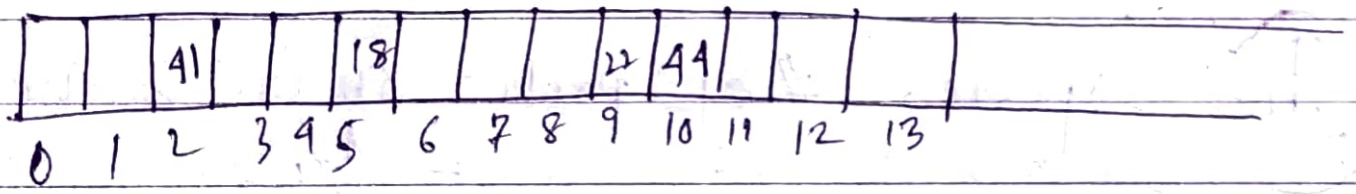
hash1(k) + j hash2(k).

j=0 init.

eg. hash1(k) = k mod 13
hash2(k) = 7 - k mod 7

Keys (18, 41, 22, 44)

K	hash1	hash2
18	5	3
41	2	1
22	9	6
44	5	5



5+1*5 j=0 ⟶ until index NIL

- Linear probing has best cache performance but suffers from primary clustering. It's easy to compute.
- Quad probing lies between the 2 in terms of cache performance & clustering.
- Double hashing has poor cache performance but no clustering. Double hashing requires more computation.

Write Search, Insert, Delete in Open Addressing (CLRS 3rd ed. 155p)

Analysis of open addressing

In terms of $\alpha = n/m$. We assume table never completely fills, so $0 \le \alpha < 1$ ($0 \le n < m$)

Assume uniform hashing.
No deletion.
In a suc. search, each key is equally likely to be searched for.

Th. Expected # of probes in an unsuc. search is at most $1/1-\alpha$.

Proof: Since the search is unsuc every probe is to an occupied slot except for last probe, which is to an empty slot.

RV X = # of probes made in an unsuccessful search

Define events A_i, for $i = 1, 2, \ldots$ to be the event that there is an ith probe & that it's an occupied slot.
$X \ge i$ if & only if probes $1, 2, \ldots, i-1$ are made & are to occupied slots $\Rightarrow$

$$Pr\{X \ge i\} = Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$$

$$Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} =$$

$$Pr\{A_1\} Pr\{A_2 / A_1\} \cdot Pr\{A_3 | A_1 \cap A_2\} \cdots$$
$$\cdots Pr\{A_{i-1} | A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}$$

Now, claim is

$$Pr\{A_j | A_1 \cap A_2 \cap \cdots \cap A_{j-1}\} = \frac{n-j+1}{m-j+1}$$

Boundary case: $j=1 \Rightarrow Pr\{A_1\} = n/m$

For $j=1$, there are n stored keys & m slots so the prob. that the 1st probe is to an occupied slot is n/m.

Given $j-1$ probes were made, there are $m-j+1$ slots remaining & $n-j+1$ of which are occupied. Thus prob. that jth probe is to an occupied slot is

$$\frac{n-j+1}{m-j+1}.$$

So, $Pr\{X \geq i\} = \underbrace{\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}}_{i-1 \text{ factors}}$

$n < m \Rightarrow \frac{n-j}{m-j} \leq n/m$ for $j \geq 0$ that implies $Pr\{X \geq i\} \leq \left(\frac{n}{m}\right)^{i-1}$

$$= \alpha^{i-1}$$

Now, $E[X] = \sum_{i=1}^{\infty} P_r\{X \geq i\}$

$\leq \sum_{i=1}^{\infty} \alpha^{i-1}$

$= \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$. (Ans)

Interpretation

If α is const. an unsucc. search takes $O(1)$ time.

If $\alpha = 0.5$, unsuc. search takes an avg. of 2 probes.

Cor: → Expected # of probes to insert is at most $1/1-\alpha$.

↓ Insertion uses same probe sequence as an unsuc. search (as no delete).

Th. Expected # of probes in suc. search is at most

$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

proof A suc. search for key k follows the same probe seq. as when k was inserted.

Is the expected # of probes to insert is at most $1/1-\alpha$, if k was $(i+1)$th key inserted, then α equaled i/m at the time. Expec. no. of probes made in a search for k is at most

$$\frac{1}{1 - i/m} = \frac{m}{m-i}$$

That was assuming that k was the $(i+1)$th key inserted. We need to avg over all n keys.

$$\frac{1}{n}\sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1} \frac{1}{m-i}$$

$$= \frac{1}{\alpha}\left(H_m - H_{m-n}\right)$$

Where $H_i = \sum_{j=1}^{i} 1/j$ is the ith harmonic no.

$$\frac{1}{\alpha}\left(H_m - H_{m-n}\right) = \frac{1}{\alpha}\sum_{K=m-n+1}^{m} 1/K$$

$$\leq \frac{1}{\alpha}\int_{m-n}^{m} \frac{1}{x}\,dx$$

$$= \frac{1}{\alpha}\ln\frac{m}{m-n}$$

$$= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

HASH_SEARCH (T, k)

```
i ← 0
repeat j ← h(k, i)
    if T[j] = k
        then return j
    i ← j + 1
until T[j] = NIL or i = m
return NIL
```

HASH_INSERT (T, k)

```
i ← 0
repeat  j ← h(k, i)
        if T[j] = NIL
        then T[j] ← k
             return j
        else i ← i + 1
until i = m
error " hash table overflow "
```

(Don't put NIL directly)

Deletion

Use a special value

*

'Deleted' instead of NIL when marking a slot as empty during deletion.

Search should treat 'deleted' as though the slot holds a key that does not match the one being searched for.

Insertion should treat 'deleted' as though the slot were empty, so that it can be reused.

(Disadv. Search time no longer dependent on α).

G'96 An advantage of chained hashed table over open addressing is -

a) worst case time complexity of search is less.

b) Space used is less.

✓c) Deletion is easier.

d) None.

G'14 $h(k) = k \bmod 9$, hash table has 9 slots, chaining used,

keys : 5, 28, 19, 15, 20, 33, 12, 17 & 10. Then max, min, avg chain length in hash table?

0

1 → 28 → 19 → 10

2 → 20

3 → 12

4

5 → 5

6 → 15 → 33

7

8 → 17

Max 3

Min 0

avg $\frac{9}{9} = 1$

G'14 Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the prob. that the first 3 slots are unfilled after the first 3 insertions?

$\frac{97}{100} \cdot \frac{97}{100} \cdot \frac{97}{100}$ (as chaining is used)

Q'97 Hash table with n buckets, chaining used. Hash f^n is s.t. prob. that a key value is hashed to a particular bucket is $1/n$. Initially empty. K distinct values inserted.

a) Prob. that bucket 1 is empty after K insertions.

b) Prob. that no collision has occurred in any of K insertions.

c) Prob. that 1st collision occurs at Kth insertion.

a) $1 - \frac{1}{n} = \frac{n-1}{n}$

$\left(\frac{n-1}{n}\right)\left(\frac{n-1}{n}\right)\cdots\left(\frac{n-1}{n}\right) = \left(\frac{n-1}{n}\right)^K$

K times

b) $\left(\frac{n}{n}\right)\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-K+1}{n}\right)$

K terms

c) $\left(\frac{n}{n}\right)\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-K+2}{n}\right)\left(\frac{K-1}{n}\right)$

$(K-1)$th Kth insertion

For simple uniform hashing, expected value of time taken for unsucc. as well as suc. search $\theta(1+\alpha)$.
When $n = O(m)$, search is $O(1)$

- **Chaining**

Searching $O(n)$ Inserting $O(1)$.

Deleting $O(n)$ [When list is doubly linked, $O(1)$]

- Load factor (α) =

$$\frac{\#\text{elems present in hash table}}{\text{Total size of table}}$$

$$= n/m$$

if $n = k, m$, where k is constant

$$\alpha = k$$

When α is constant, TC for insert, search, delete - $\theta(1)$.

***Primary clustering** (Open Addressing)

Clusters of elems in the list or table. (Prominent in linear probing)

Secondary clustering (Open Addressing)

If 2 elems map to same bucket, their probe sequences become identical.

Prob. of going to a bucket # i is $\frac{\#\text{consecutive elems present before it} + 1}{m}$

- **Open addressing**

→ **Linear Probing**

$$h'(k, i) = (h(k) + i) \bmod m$$

Primary ↓ Secondary clustering

$k \rightarrow$ key

$i \rightarrow$ collision #

$m \rightarrow$ # buckets

m probe sequences

→ **Quadrating probing**

$$h'(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$$

No primary clustering ✓

Secondary clustering ✓

m probe sequences.

Choose c_1, c_2 s.t. in m probes all buckets are examined.

→ **Double hashing**

$$h'(k) = (h_1(k) + i\, h_2(k)) \bmod m$$

No p or s clustering

m m

m^2 probe sequences

Make sure $h_2(k)$ & m are relatively prime.

making h_2 & m relatively prime

$h_2(k)$	m
odd #	2^k
$< m$	prime #

→ Linear probing has the best cache performance but suffers from clustering.

Quad. probing lies b/w lin. & dou. has. in terms of cache performance, & clustering.

Double hashing has poor cache performance & no clustering problem.

*

Separate chaining	Open addressing
i) # keys to be stored can exceed size of hash table.	i) # keys never exceed size of hash table.
ii) Deletion easier.	ii) Deletion difficult.
iii) Extra space reqd. for ptrs to store keys outside hash table.	iii) No extra space reqd.
iv) Cache performance poor.	iv) Cache performance better.
v) Some buckets are never used that leads to wastage of space.	v) Buckets may be used even if no key maps to those particular buckets.

→ Use chaining when all 3 operations have to be done.

Use open addressing when only search & insert have to be done.

Q'07 Table size = 20. After how many IT keys after getting hashed, will the prob. that any key hashed collides with an existing one exceeds 0.5.

→ i elems already inserted

$$\frac{i}{m} > 0.5$$

$$\Rightarrow i > 10$$

Ans = 11

Q'10 How many insertion sequences possible for this table?

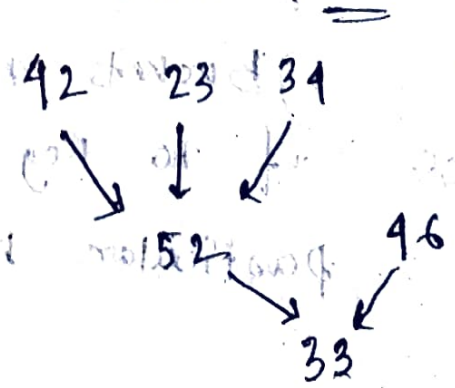
46	34	42	23	52	33

example seq.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

46, 34, 42, 23 inserted at first probe.

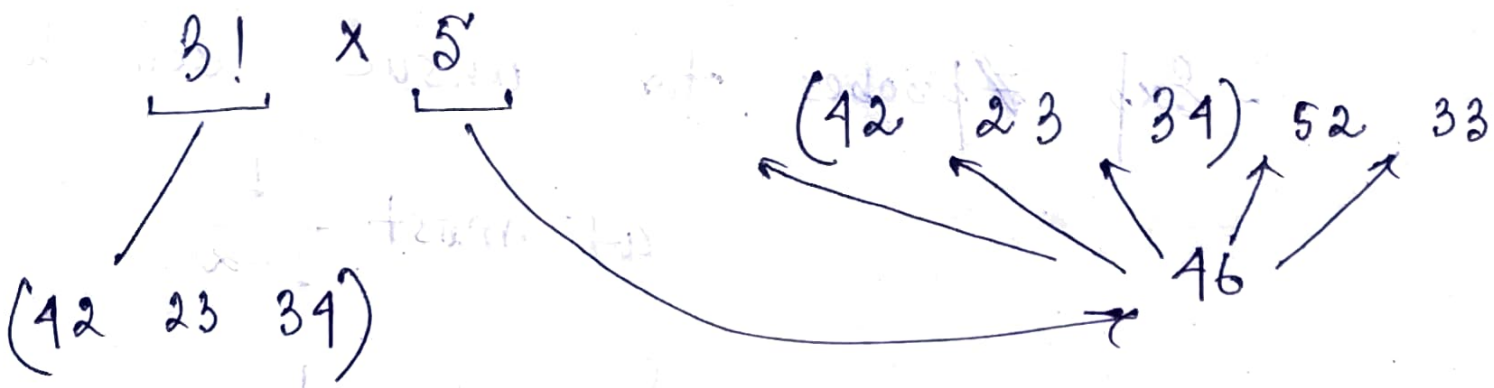
Ans = 30 (90) (rbr)



great qsn!

- In a valid insertion sequence elem 42, 23, 34 must appear before 52, 33.

  46 must appear before 33.



- Alter:

i) $\underbrace{(42\ 23\ 34\ 46)}_{4!}\ 52\ 33$

ii) $\underbrace{(42\ 23\ 34)}_{3!}\ 52\ 46\ 33$

$\Rightarrow 4! + 3! = 30.$

* Chaining (Under assumption of simple uniform hashing)

Expected time for suc, unsuc. search is $\theta(1+\alpha)$.

Insert $O(1)$

Delete $O(1)$ when used doubly linked list

If $\alpha = O(1)$, search also $O(1)$.

* Open addressing

$\alpha = n/m \quad n \leq m$

(assume uniform hashing, no deletion)

- Exp. #probes for unsuc. search = at most $\frac{1}{1-\alpha}$
- Exp. #probes for insert = at most $\frac{1}{1-\alpha}$
- Exp. #probes in suc. search = $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
- Search, insert, delete takes at most $\frac{1}{1-\alpha}$ time.

→ In linear probing, for $0.01 < \alpha < 0.99$, clusters are of size $\Theta(\lg n)$.

- OA - better cache performance

Chaining - Less sensitive to hash fn's & α.

Complexity Theory (iCake)

⇒ P : Set of polynomial time algorithms.

✓ $O(\lg n)$ $O(n)$ $O(n^{127})$

✗ $O(n^n)$ $O(2^n)$ $O(n!)$

⇒ NP : Set of problems whose answers can be verified in polynomial time.

(Non-deterministic polynomial)

eg. factoring a large no, if the factors are given we can verify the correctness of those factors in polynomial time by multiplying them to get the desired number ($O(n^2)$ multiplication).

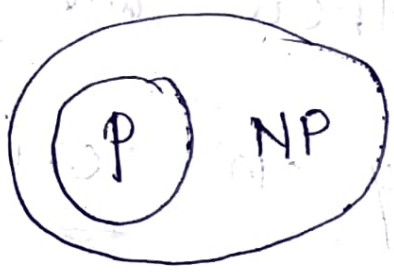
↓ of a no.

mult each digit with each of other no.

- Same for k-graph coloring.
  Verifiable in polynomial time.

※ P, NP Relation

If we can solve a problem in polynomial time, then we can verify a sol^n to the problem by solving it (in polynomial time) & comparing our answer with the provided one (in constant time). ⇒ So, any problem in P is also in NP.



⇒ Question now is:

Are there problems that are in NP but not in P?

Is P = NP?

We DON'T know!

There are lots of problems that we know are in NP where we don't know any polynomial time algorithms for them. But, we don't know if that's because there aren't any polynomial time algos or if that's because we just haven't found them yet.

Is P = NP ?

⇒ If there are problems that are easy to verify, hard to solve.

* The Satisfyability Problem (SAT).

This problem asks if there's an assignment of values to the constituent booleans, for which a whole boolean expression evaluates to be true.

↝ $A \wedge \bar{B}$ satisfiable

A true, B false

↝ $A \wedge \bar{A}$ unsatisfiable.

↝ $(\bar{x} \wedge y) \vee (x \wedge \bar{z})$ satisfiable.

$x = z = \text{false}$

$y = \text{true}$.

We dont know of a polynomial time algorithm to solve SAT expressions. So, SAT may or may not be in P. However, we can verify in polynomial time; when we are given the individual value of the booleans. So, SAT is in NP.

- Any problem of NP could be transformed into an instance of ~~a~~ SAT. (Cook-Levin theorem)

⇒ If SAT can be solved in polynomial time, every other NP problem can be solved in polynomial time.

So, P = NP in that case.

⇒ Any problem in NP, if solvable in polynomial time, then P = NP.

Using this approach, researchers have found a lot of NP problems where if any of them can be solved in polynomial time, then all of them can be solved in polynomial time & P = NP. This group of problems

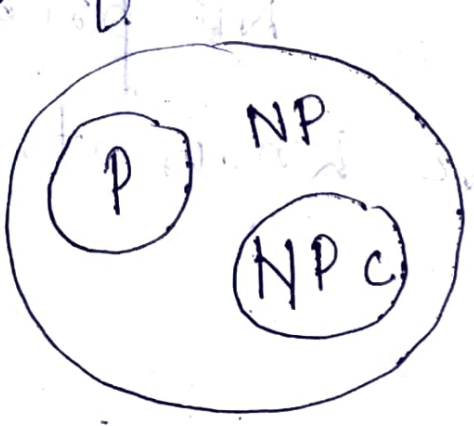
is called NP-complete.

※ NP Complete

A problem is NP complete if it has 2 properties:

i) It's in NP (Sol^n's to it can be verified in poly. time)

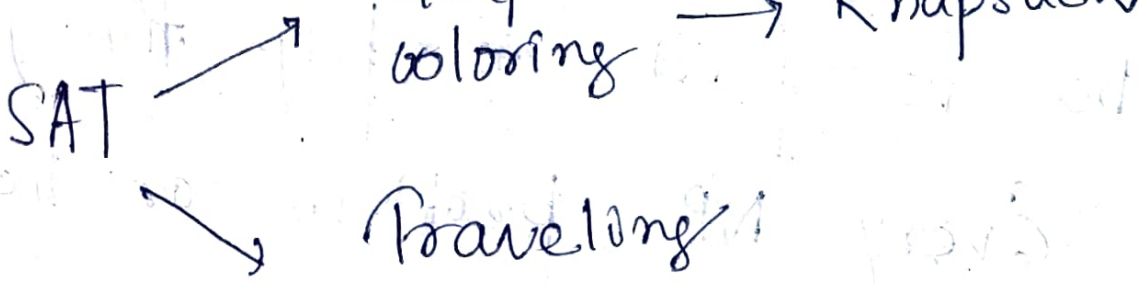
ii) Every NP problem can be transformed into it.



(We can check (ii) by transforming the known NP-complete problem (like SAT, graph-coloring) into the new problem.)

III At a high level, we're building out a chain of transf^n's,

showing how we can take one problem & reshape it as a different one. SAT is at the root of our chain & other problems follow.



If we can add an NP problem to the chain, we know its NP-complete.

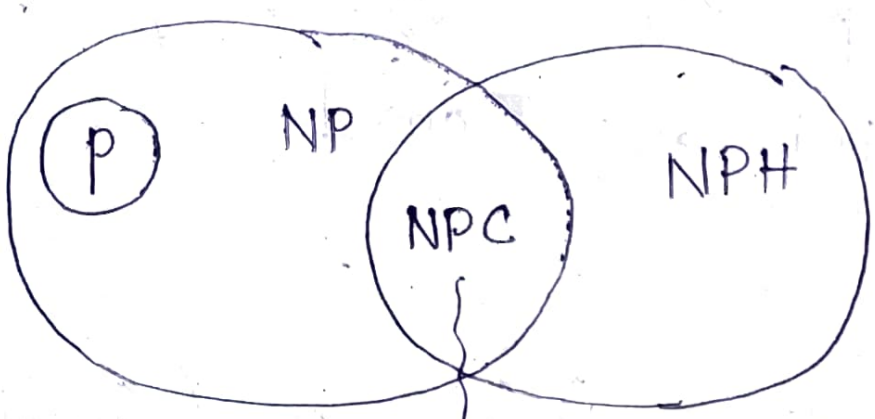
NP-hard.

A problem is NP-hard if it has the 2nd property of NPC problem: every NP problem can be transformed into it.

Since every NPC problem has this property, every NPC problem

is NP-hard.

⇒ Some, NPH problems are not in NP, as NPH probs may not be verified in poly. time..



In NP + (each NP problem can be transformed into it)

eg. Halting problem is NP-hard. SAT can be transformed into an instance of it. But, we can't quickly verify an answer to the halting problem. So, it is not in NP. (How to double check if somet said that a prog. terminated ?)

Complexity Theory

P Problems

As the name says these problems can be solved in polynomial time, i.e.; $O(n)$, $O(n^2)$ or $O(n^k)$, where k is a constant.

NP Problems

Some think NP as Non-Polynomial. But actually it is Non-deterministic Polynomial time (https://en.wikipedia.org/wiki/NP_(complexity)). i.e.; "yes/no" instances of these problems can be solved in polynomial time by a non-deterministic Turing machine and hence can take up to exponential time (some problems can be solved in sub-exponential but super polynomial time) by a deterministic Turing machine. In other words these problems can be verified (if a solution is given, say if it is correct or wrong) in polynomial time by a deterministic Turing machine (or equivalently our computer). Examples include all P problems. One example of a problem not in P but in NP is Integer Factorization (https://en.wikipedia.org/wiki/Integer_factorization_problem).

NP Complete Problems(NPC)

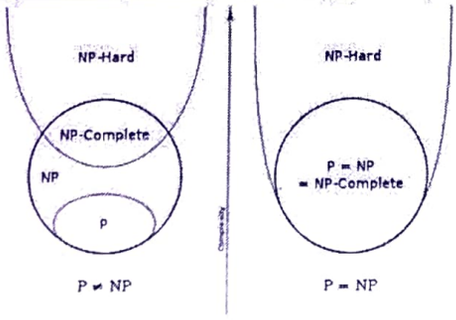
Over the years many problems in NP have been proved to be in P (like Primality Testing (https://en.wikipedia.org/wiki/Primality_test)). Still, there are many problems in NP not proved to be in P. i.e.; the question still remains whether $P = NP$ (i.e.; whether all NP problems are actually P problems).

NP Complete Problems helps in solving the above question. They are a subset of NP problems with the property that all other NP problems can be reduced to any of them in polynomial time. So, they are the hardest problems in NP, in terms of running time. If it can be showed that any NPC Problem is in P, then all problems in NP will be in P (because of NPC definition), and hence $P = NP = NPC$.

All NPC problems are in NP (again, due to NPC definition). Examples of NPC problems (https://en.wikipedia.org/wiki/List_of_NP-complete_problems)

NP Hard Problems (NPH)

These problems need not have any bound on their running time. If any NPC Problem is polynomial time reducible to a problem X, that problem X belongs to NP Hard class. Hence, all NP Complete problems are also NPH. In other words if a NPH problem is non-deterministic polynomial time solvable, it is a NPC problem. Example of a NP problem that is not NPC is Halting Problem (https://en.wikipedia.org/wiki/Halting_problem).



From the diagram, its clear that NPC problems are the hardest problems in NP while being the simplest ones in NPH. i.e.; $NP \cap NPH = NPC$

Note

Given a general problem, we can say its in NPC, if and only if we can reduce it to some NP problem (which shows it is in NP) and also some NPC problem can be reduced to it (which shows all NP problems can be reduced to this problem).

Also, if a NPH problem is in NP, then it is NPC

P, NP, NPC, NPH (http://gatecse.in/wiki/NP,_NP_Complete,_NP_Hard)

- P is a subset of NP, but whether $P = NP$ is still not known
- NPC is a subset of NP (only if $P \neq NP$, NPC is a proper subset of NP)
- NPC is a proper subset of NPH

Reduction

Reducing a problem A to problem B means converting an instance of problem A to an instance of problem B. Then, if we know the solution of problem B, we can solve problem A by using it.

Consider the following example:

You want to go from Bangalore to Delhi and you can get a ticket from Mumbai to Delhi. So, you ask your friend for a lift from Bangalore to Mumbai. So, the problem of going from Bangalore to Delhi got reduced to a problem of going from Mumbai to Delhi.

Some Inferences:

Consider problems A, B and C.

Assume all reductions are done in polynomial time.

Two Important Points:

1. If A is reduced to B and $B \in$ class X, then A cannot be harder than X, because we can always do a reduction from A to B and solve B instead of directly solving A. This reduction thus gives an upper bound for the complexity of A.
2. If A is reduced to B and $A \in$ class X, then B cannot be easier than X. The argument is exactly same as for the one above. This reduction is used to show if a problem belongs to NPH – just reduce some known NPH problem to the given problem. This reduction thus gives a lower bound for the complexity of B.

Somethings to take care of:

- If A is reduced to B and $B \in NPC$, then $A \in NP$
  - Here, we cannot say A is NPC. All we can say is A cannot be harder than NPC and hence NP (all NPC problems are in NP). To belong to NPC, all NP problems must be reducible to A, which we cannot guarantee from the given statement.
- If A is reduced to B, C is reduced to A , $B \in NP$ and $C \in NPC$, then $A \in NPC$
  - Here, the first reduction says that A is in NP. The second reduction says that all NP problems can be reduced to A (because C is in NPC, by definition of NPC, all problems in NP are reducible to C and now C is reduced to A). Hence, the two necessary and sufficient conditions for NPC are satisfied and A is in NPC
- If A is reduced to B and $B \in NPH$, then $A \in$?
  - Here we can't say anything about A. It can be as hard as NPH, or as simple as P.

P, NP, NP-hard, NP-complete

(gatecse.in article)

* P Class problem.

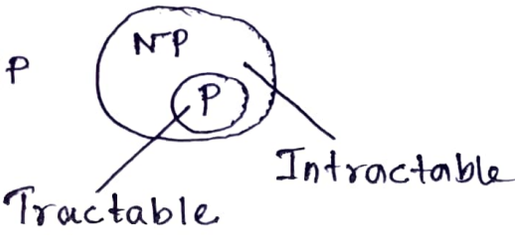
Problem that can be solved in polynomial time (n^k). (Also verifiable in polynomial time)

eg. Sorting, Searching algo.

* NP Class Problem (Non-deterministic Polynomial)

Can't be solved in polynomial time but is verified in polynomial time.

eg. Sudoku, Prime factor, Traveling salesman, Scheduling.



→ If P ≠ NP, some problems are not solvable at polynomial time.

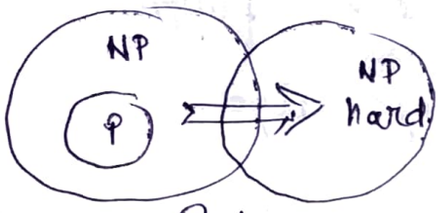
* Reduction

Let A & B are 2 problems. Then A reduces to B iff there's a way to solve A by deterministic algo that solves B in polynomial time.

A reducible to B $A \propto B$

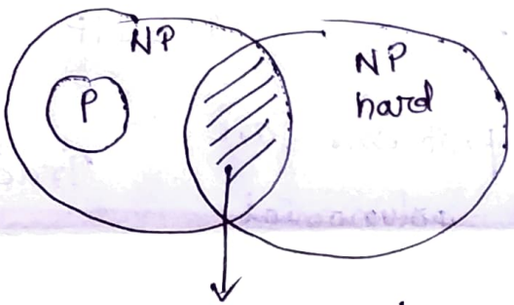
1. If $A \propto B$ & B in P class, the A is also in P class.
2. A is not in P ⇒ B not in P class.

* NP Hard If every problem in NP can be polynomially reduced to this class NP-hard.
(Optimization problem)



Reduce
in polynomial time

* NP - complete If NP ∩ NP-hard.
(decision problems)



→ If A is a decision problem, B an optimⁿ problem, then it's possible to A ∝ B.

Knapsack decision problem ∝ 0/1 opⁿ problem.

* Problem

- Solvable (Potential algorithm exists to find the solution or there's a proof that the problem can't be solved.)
- Unsolvable (No algo exists to solve the problem, also we don't know whether it is solvable or not)

- Solvable
  - Decidable (Algo & procedure)
  - Undecidable (Procedure)

* Decision problem. Yes/No answer.

Input → ○ → Yes/No

Decision procedure : Algo to solve decision problems.