$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$C = A \times B = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

$P = 5 \times 13 = 65$ $\quad Q = 7 \times 7 = 49$ $\quad R = 1 \times -2 = -2$ $\quad S = 4 \times 2 = 8$

$T = 3 \times 8 = 24$ $\quad U = 2 \times 12 = 24$ $\quad V = 2 \times 15 = 30$

$C_{11} = P + S - T + V = 65 + 6 - 24 + 30 =$

$C_{12} = R + T = -2 + 24 = 22$ ✓

$C_{21} = Q + S = 49 + 6 = 55$

$C_{22} = P + R - Q + U = 65 + (-2) - 49 + 24$

23-06-'18

GREEDY METHOD

If a problem 'P' takes 'n' number of inputs we select a some subset of inputs from 'P' that follow problem constraints. This subset is the input set satisfying some constraints is called feasible solution. Optimal solution is feasible solution where inputs are maximas or minimas.

★★ → Abstract Method of Greedy Approach

```
Algorithm Greedy (a, n)
{
    sol := ϕ    // initially, solution vector is empty
    for i := 1 to n do
    {
        x := select(a);  // selection procedure to select next input
        if (feasible(sol, x)) then
            sol := union(sol, x);
    }
    return sol;
}
```

Fractional Knapsack Problem

Knapsack is a bag. Imagine a salesman. He has to fill the knap. Each item is associated with weight and profit. He has to fill the knapsack such that he earns maximum profit. The maximum capacity of the knapsack is 'M'.

Items : i_1 i_2 i_3 ..... i_n

Weight : w_1 w_2 w_3 ..... w_n

Price : P_1 P_2 P_3 ..... P_n

x_1	x_2	x_3	Profit
0	$10/15$	1	$0 + \frac{10}{15} \times 24 + 1 \times 10 = 16 + 10 = 26$

Optimal solution

x_1	x_2	x_3	Profit
0	1	$1/2$	$0 + 24 + \frac{1}{2} \times 15 = 24 + 7.5 = 31.5$

To find the solution (optimal solution) for the above problem, we find the profit rate (P_i/w_i) of each product and then arrange all items in descending order of profit rates.

$\frac{P_1}{w_1} = \frac{25}{18} = 1.388$ $\quad \frac{P_2}{w_2} = \frac{24}{15} = 1.6$ $\quad \frac{P_3}{w_3} = \frac{15}{10} = 1.5$

$i_2 > i_3 > i_1$

1 $\quad$ $1/2$ $\quad$ 0

15 $\quad$ $(20 - 15 = 5 = 5/10 = 1/2)$

→ Algorithm Greedy knapsack (M, n)

{

for i := 1 to n do

x[i] := 0.0;

U := M;

for i := 1 to n do

{

Note: Must give the inputs in order.

$\frac{P_1}{w_1} > \frac{P_2}{w_2} > \frac{P_3}{w_3} > \ldots > \frac{P_n}{w_n}$

if (w[i] > U) then Break;
x[i] := 1.0; U := U - w[i];
}
if (i ≤ n) then x[i] := U / w[i];
}

Eg

Item	Weight	Profit
1	5	6
2	7	4
3	2	6
4	3	9
5	4	5

M = 15

$\frac{P_1}{w_1} = \frac{6}{5} = 1.2$ $\frac{P_2}{w_2} = \frac{4}{7} = 0.55$ $\frac{P_3}{w_3} = \frac{6}{2} = 3.0$

$\frac{P_4}{w_4} = \frac{9}{3} = 3.0$ $\frac{P_5}{w_5} = \frac{5}{4} = 1.25$

NOTE: If knapsack contains 2 items, $P_3=4$ $w_3=4$, $P_2=3$ $w_2=2$ we must select $P_2=3$ $w_2=2$ only

Solution vector for above problem (1, 1, 1, 3, 0, 0, 0, 0, 0, 0)

$i_4 > i_3 > i_5 > i_1 > i_2$

(max profit)

1 1 1 1 $\frac{1}{7}$

→ Consider 'n' no of items for knapsack problem 'y' is the solution vector.

$\sum_{i=1}^{n} y_i w_i = M$

y_1 y_2 y_3 y_4 ..... y_i .... y_j ..... y_m

↳ Assume that this is the solution vector

If you feel that it is not the optimal solution, then we exchange some portion from y_i to y_j (decrease in one and increase in the other). Substituting these values, the optimal solution is.

y_1 y_2 y_3 y_4 ..... $(y_i - \epsilon)$ ..... $(y_j + \epsilon')$ ..... y_m

Substitute these values in the original equation

$y_1w_1 + y_2w_2 + y_3w_3 + y_4w_4 + \cdots + (y_i - \epsilon)w_i + \cdots + (y_j + \epsilon')w_j$

$\rightarrow \cdots + y_n w_n = M$

$= \sum_{i=1}^{n} y_i w_i - w_i\epsilon + w_j\epsilon'$

If $w_j\epsilon' = w_i\epsilon$ ⇒ the given solution is optimal solution

$w_i\epsilon = w_j\epsilon' \Rightarrow \boxed{\frac{\epsilon}{\epsilon'} = \frac{w_j}{w_i}}$

To maximize the profit values.

$y_1P_1 + y_2P_2 + y_3P_3 + y_4P_4 + \cdots + (y_i - \epsilon)P_i + \cdots + (y_j + \epsilon')P_j$

$\rightarrow \cdots + y_n P_n$

Profit (new) = old profit $- P_i\epsilon + P_j\epsilon'$

To earn more profit (maximum profit), $P_j\epsilon' > P_i\epsilon \Rightarrow \boxed{\frac{P_j}{P_i} > \frac{\epsilon}{\epsilon'}}$

$\frac{P_j}{P_i} > \frac{\epsilon}{\epsilon'}$; $\frac{\epsilon}{\epsilon'} = \frac{w_j}{w_i} \Rightarrow \frac{P_j}{P_i} > \frac{w_j}{w_i} \Rightarrow \frac{P_j}{w_j} > \frac{P_i}{w_i}$

↳ To earn more profit $\frac{P_j}{w_j} > \frac{P_i}{w_i}$ ⇒ Input order of knapsack problem is decreasing order of profit rates

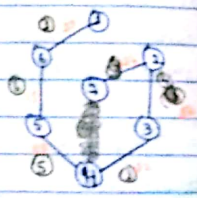
23-02-24

Minimum Spanning Tree (MST)

It comes under ordering problem. It contains undirected graph which contains a set of vertices & edges. Given graph G(V, E); construct G'(V, E'). We need to select all the edges which give (i, j) such that minimum cost. We need to select near(j) and take cost[j, near(j)] where cost is the cost matrix of the given graph.

Eg

By Kruskal's, MST

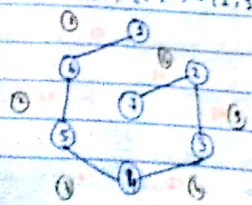


Minimum cost = 10 + 22 + 14 + 16 + 25 + 12

By Prim's, MST

(2,6) = 10 ⇒ (6,1) = 7 minimum weight

(2,6), [(6,5) < (6,1)], [(5,4) < (5,2) < (6,1)], [(4,3) < (4,2) < (1,2)], [(3,2) < (1,2)], [(2,1) < (2,3)] ⇒ All vertices covered



Minimum Cost

= 10 + 25 + 22 + 12 + 16 + 14

= 99

cost [1:n, 1:n] // cost matrix , E // set of edges

t [1:n-1, 1:2] // resulting MST (2-D array, (n-1) edges, n vertices)

t [(i,1) (i,2)] // current edge

We must define near(j)

If near(j) = 0

↳ j is already in the minimum spanning tree

If near(j) = i

↳ i is neighbour, cost of [near(j), i] = minimum cost

To construct cost matrix,

↳ If an edge exists between two vertices, the cost is the cost of the edge between the two vertices.

cost (i,j) = cost E(i,j)

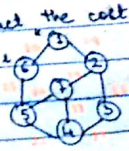
↳ The cost is 0 for a vertex that starts and ends at itself

cost (i,i) = 0

↳ The cost is infinite for two vertices that have no edge between them.

cost (i,j) = ∞

Problem: Construct the cost matrix for the given graph



	1	2	3	4	5	6	7
1	0	28	∞	∞	∞	10	∞
2	28	0	16	∞	∞	∞	14
3	∞	16	0	12	∞	∞	∞
4	∞	∞	12	0	22	∞	18
5	∞	∞	∞	22	0	25	24
6	10	∞	∞	∞	25	0	∞
7	∞	14	∞	18	24	∞	0

→ Prim's Algorithm to construct Minimum Spanning Tree

```
Algorithm Prims (G, E, V, cost) //Graph, set of edges, set of vertices, cost matrix
{
    let (k, l) be an edge of minimum cost in E
    min cost := cost (k, l);
    t[1, 1] := k;  // k = 1        cost (1,6) = least
    t[1, 2] := l;  // l = 6
    for i := 1 to n do //to find vertices which are nearer to k, l
        if (cost [i, l] < cost [i, k]) then
            near[i] := l;
        else
            near[i] := k;
    near[k] := near[l] := 0
    for i := 2 to n-1 do //to calculate minimum cost
    {
        let j be an index such that
        near[j] ≠ 0 and cost[j, near[j]] is minimum
        t[i, 1] := j;
        t[i, 2] := near[j];
        for k := 1 to n do
            if (near[k] ≠ 0 and (cost [k, near[k]] > cost [k, j]))
                then //to update near values
                    near[k] := j;
        }
        return min cost;
    }
```

→ near[i]		
	1	0
	2	1
	3	-
	4	-
	5	6
	6	0
	7	-

→ * Since they already exist as first vertex

→ *

Time Complexity (for Prim's)

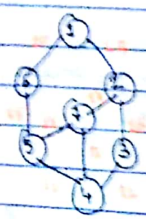
Worst Case = $O(n^2)$ // two for loops

* Kruskal's Algorithm :: Select minimum edge (many approach)
* Prim's Algorithm :: Select nearest neighbour

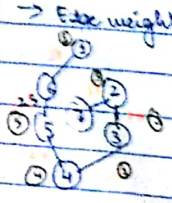
→ Kruskal's algorithm for MST

- Initially, each vertex is one set.
- Find minimum edge. Each vertex is one set.
- Then Construct minimum heap.
- Delete minimum element edge cost.
- Find the vertices associated with minimum edge (j, k)
- j = k → two vertices belong to same set, Cycle is formed
- j ≠ k → Assign that set of vertices to the minimum spanning tree.

Eg:



→ Edge weight in ascending order : 10, 12, 14, 16, 22, 24, 28



∴ Minimum cost = 10 + 12 + 14 + 16 + 22 + 25

= 99

Inputs: E → set of edges, cost → cost matrix, n → no. of vertices
t → minimum spanning tree t[1:n-1, 1:2]

t(1,1) = 1 t(1,2) = 6 ← 1 to n-1 edges 1, 2 are vertices

→ To store each edge if taking log E time, if E no. of edges, it takes O(log E) time

```
Algorithm Kruskal (E, cost, n, t)
{
  Construct a heap out of the edge cost using heapify
  for i := 1 to n do parent[i] := -1;
      i := 0; mincost := 0;
  while (i < n-1) and heap not empty do
  {
    Delete minimum cost edge (u, v) from heap
    and reheapify using adjust

    j := find(u); k := find(v);
    if (j ≠ k) then
    {
       i := i + 1;
       t[i, 1] = u;
       t[i, 2] = v;
    }
    Min Cost = mincost + cost[u, v];
    Union(j, k)
  }
}
  if (i ≠ n-1) then write (no spanning tree)
  else
    return min cost
}
```

Optimal Storage on Tape

→ We have 'n' no. of programs each of length l_i.
→ We have a computer tape of length 'L'.
→ We must store the programs on this tape so that minimum time required in retrieving the program from the tape
→ We must retrieve the program from the tape
→ This is related to ordering problem, ∧ formula

→ $\sum_{1 \le j \le k} l_{i_j}$

$i = 1$ to n

$i_1\ i_2\ i_3\ i_4 \ldots\ i_j \ldots\ i_n$

↳ Each take equal time

$\frac{1}{n} \sum_{1 \le k \le n} l_{i_k}$

→ Minimum Retrieval Time

$d(I, j) = \sum_{1 \le i \le n} \sum_{i \le k \le j} l_{i_k}$ for $i \le j$

→ To retrieve i_1 = l_1, $i_2 = (l_1 + l_2)$, $i_3 = (l_1 + l_2 + l_3)$

→ Eg: $n = 3$

$(l_1, l_2, l_3) = (5, 10, 3)$

Total number of orders = $3! = 6$

Orderings:

1 2 3 = $5 + (5+10) + (5+10+3) = 38$
1 3 2 = $5 + (5+3) + (5+3+10) = 31$
2 1 3 = $10 + (10+5) + (10+5+3) = 43$
2 3 1 = $10 + (10+3) + (10+3+5) = 41$
3 1 2 = $3 + (3+5) + (3+5+10) = 29$
3 2 1 = $3 + (3+10) + (3+10+5) = 31$

Optimal order = Order with minimum cost = 3 1 2

$(l_1, l_2, l_3) = (5, 10, 3)$
1 2 3

→ To find optimal solution cost, we arrange all the lengths in ascending order.

→ If we have 'n' no. of programs and 'm' no. of tapes

→ Eg: $n = 13$, $m = 3$

13 programs, 3 tapes (T_0, T_1, T_2)

12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10, 6

• Arrange all programs in ascending order

3, 4, 5, 5, 6, 7, 8, 10, 11, 12, 18, 26, 32
1 2 3 4 5 6 7 8 9 10 11 12 13

• Assign one by one

T_0 – 3 5 8 12 32

T_1 – 4 6 10 18 36

T_2 – 5 7 11 26

• Find minimum retrieval time for tapes

$T_0 = (3) + (3+5) + (3+5+8) + (3+5+8+12) + (3+5+8+12+32)$
$= 115$

$T_1 = (4) + (4+6) + (4+6+10) + (4+6+10+18)$
$= 72$

$T_2 = (5) + (5+7) + (5+7+11) + (5+7+11+26)$
$= 89$

∴ Optimal order = (2, 5, 8, 11)

Storage on Tapes

Optimal Merge pattern (continuation)

→ We have n! orders for 'n' no of programs.

→ Optimal storage on tapes. Input order = $l_i \leq l_{i+1}$

→ Algorithm for optimal storage on tapes where no of programs = n, no of tapes = m, input order = $l_i \leq l_{i+1}$

```
Algorithm OST(m, n)
// li ≤ li+1 : input order
{
    j := 0; // first tape
    for i := 0 to n
    {
        write ("append program i to tape j");
        j := (j + 1) mod m;
    }
}
```

Output of tapes programs must be in ascending order

→ Programs: $i_1, i_2, i_3, \ldots \ldots, i_n$

→ in ascending order

To access i_2 time = $l_1 + l_2$; access i_3 time = $l_1 + l_2 + l_3$; access i_n time = $l_1 + l_2 + l_3 + \ldots + l_n$

Mean retrieval time = MRT

$= \frac{1}{n}(l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots \ldots \ldots) \rightarrow \frac{1}{n}(l_1 + l_2 + l_3 + \ldots + l_n)$

Optimal Merge Pattern

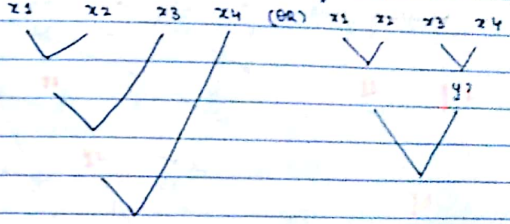
→ Merging after merge sort

2 sorted lists: [1|3|5] [2|4|6]

$i \rightarrow j$ $i \rightarrow j$

[1|2|3|4|5|6]

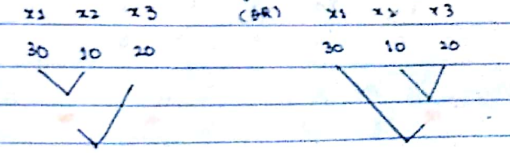
→ Problem Definition: To merge 'n' number of sorted lists with minimum number of comparisons



↳ 2 orders for merging

→ If we have 'n' no of ordered lists, we have 'n!' no of orders to merge. We need to find best order

→ Eg



No of comparisons = 60 + 100 = 160

No of comparisons = 40 + 60 = 100

↳ Best

→ # Fuest, start merging from least number of elements in sorted list.

→ # x1, x2, x3 : sorted lists with 20, 10, 20 elements in given example.

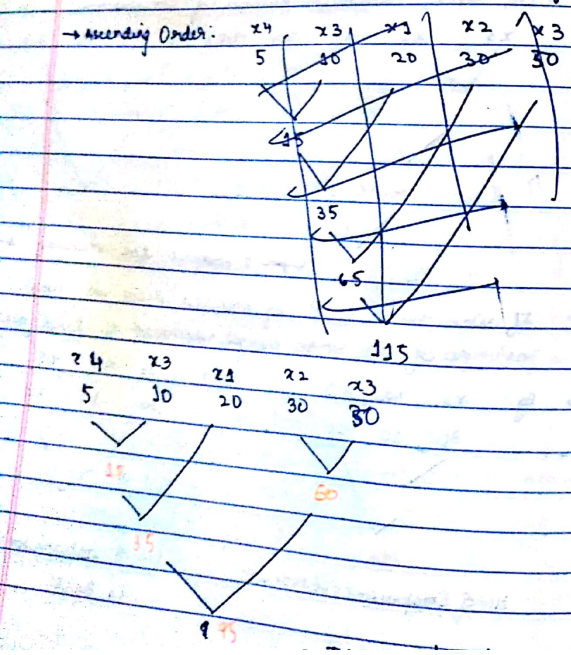
→ Input order: Arrange all the number of elements in ascending order and then merge one by one.

→ Eg:

x1	x2	x3	x4	x5	→ sorted lists
20	30	10	5	30	→ no. of elements

→ Ascending Order: x4 x3 x1 x2 x3

5 10 20 30 30



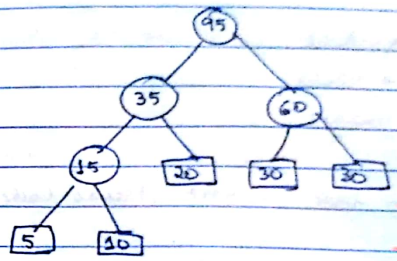
x4 x3 x1 x2 x3

5 10 20 30 30

Tree: leaf nodes - (squares), root nodes - (circles)

files

no. of comparisons



$\sum_{i=1}^{n} d_i q_i$ = No. of comparisons (minimum)

d_i = distance from root node to leaf node

q_i = no. of elements in each file

$\therefore \sum_{i=1}^{n} d_i q_i = (5\times3) + (10\times3) + (20\times2) + (30\times2) + (30\times2)$

$= 15 + 30 + 40 + 60 + 60 = 205$

→ Algorithm:

Initially, each node is one tree, each node has weight value

Define two functions

i) least (i) → returns minimum value

ii) ~~Combine~~ insert (i, t) → to insert new node into binary tree

~~Combine~~ # Initially, find two least files.

Combine these two least files to ~~combo~~ form a binary tree. => i.e., Combine the weight value

Huffman's code is application of Greedy
Takes 7 bits to transform each bit from 1 place to
other place. Huffman encoding max frequency - min bits
min frequency - max bits - lossless encoding - there is no

```
treenode = record
Record = node
{
treenode *lchild;
treenode *rchild;
integer weight;
}
Algorithm Tree(n) //Pseudo code for 2 way merge
{
    for i:=1 to n-1 do
    {
        pt:= new Treenode;
        pt→lchild = least(list); //returns 1st minimum
        pt→rchild = least(list); //returns 2nd minimum
        pt→weight = ((pt→lchild)→weight)+((pt→rchild)→
        Insert(list, pt);
    }
    return Least(list);
}
```

→ Initially, each node is one tree

→

Huffman Encoding

→ We have approximately some 100 characters on the keyboard.

→ We require $\lceil \log_2 100 \rceil = \lceil \log_2 100 \rceil = \text{ceil}(\log_2 100) = 7$

→ To transfer one file from source to destination each character takes 7 no. of bits

→ But in alphabets, characters have frequency

→ We reduce the bit loss using this method.

→ Highest frequency character takes minimum number of bits
Least frequency character takes maximum number of bits

→ It is an application of Optimal Storage Merge Pattern

→ Eg. To represent 7 characters we need three bits

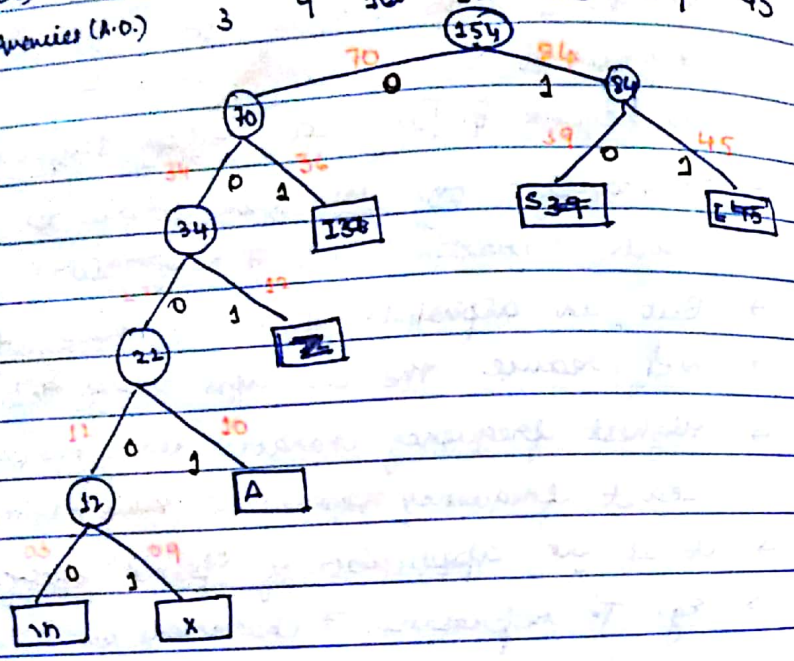
Letter	Code	Frequency (in file)	No. of bits with encoding
A	000	10	30 (10×3)
E	001	45	135
I	010	36	108
X	011	09	27
Z	100	12	36
S	101	39	117
V	110	03	+ 9
			462

↳ Total no. of bits required without Huffman's encoding

→ We can reduce the number of bits using Huffman's encod

→ Construct tree using optimal merge pattern

Letters	In	X	A	Z	I	S	E
Frequencies (A.O.)	3	9	10	12	36	39	45



	(no. of bits × frequency)
In = 00000	— 5 × 3 = 15
X = 00001	— 5 × 9 = 45
A = 0001	— 4 × 10 = 40
Z = 001	— 3 × 12 = 36
I = 01	— 2 × 36 = 72
S = 10	— 2 × 39 = 78
E = 11	— 2 × 45 = 90
	376

↳ This bit number is less than the previous one.

29-01-18

Single Source Shortest Paths (Dijkstra's Algorithm)

→ Problem Statement: Graph (directed) contains set of vertices and set of directed edges. Find out shortest path from given source vertex to remaining all vertices.

→ G (V, E)

Assume edge cost is +ve value, -ve values not allowed

→ Find shortest path from given source vertex to 1st nearest vertex, 2nd nearest vertex, ... till all nodes are covered.

→ This is a multi stage problem.

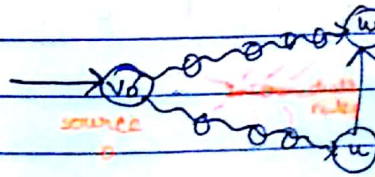
→ We are adding edges one by one.

→ We use two data structures:

dist[v] → shortest path from source vertex V_0 to node v

S → set of vertices already for which we generated shortest path

→

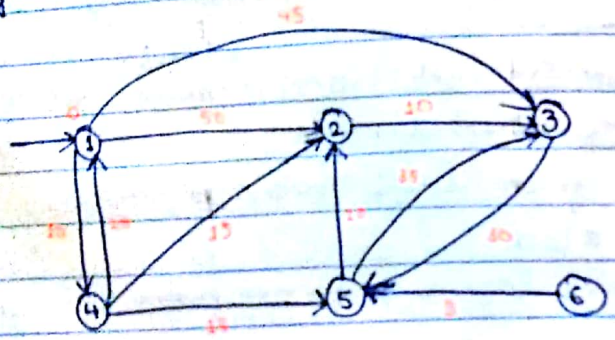


dist[w] > dist[u] + cost [u, w]

dist[w] = dist[u] + cost[u, w];

→ Add 1st shortest path edge, 2nd shortest path edge and so on to set S and proceed till you reach (n - 2) edges in set S.

→ Eg. G(V, E)



Cost Matrix:

	1	2	3	4	5	6
1	0	50	45	10	∞	∞
2	0	∞	10	∞	∞	∞
3	∞	∞	0	∞	30	∞
4	20	15	∞	0	25	∞
5	∞	20	35	∞	0	∞
6	∞	∞	∞	∞	3	0

P.T.O →

Iteration	S	Vertex Selected	Distance (dist[v]) 1	2	3	4	5	6
→ Initial	–		0	50	45	10	∞	∞
→ 1	{1}	(4)	0	50	45	(10)	∞	∞
→ 2 (d[4] = 10)	{1,4}							
select vertex adjacent to u, i.e. 4: 2 = w; d[w] > d[u] + cost[u,w]		(5)	0	25	45	10	∞	∞
=> 50 > 10 + 15 => 50 > 25 ✓								
5 = w; d[w] > d[u] + cost[u,w] => ∞ > 10 + 15 => ∞ > 25 ✓			0	25	45	10	25	∞
→ Select next min vertex, either 2 or 5								
3	{1,4,5}	(2)	0	25	45	10	25	∞
2 => 50 > (10 + 35 + 20) ✗			0	25	45	10	25	∞
3 => 45 > (10 + 15 + 35) ✗								
→ 4	{1,4,5,2}	(3)	0	25	45	10	25	∞
3 => 45 > ~~30~~ ✗ (10 + 15 + 20 + 10) ✗								
→ 5	{1,4,5,2,3}		0	25	45	10	25	∞
5 => 30 > ~~(10 + 15 ✓~~ (10 + 15 + 20 + 10) ✗								

Algorithm ShortestPath (v, cost, dist, n) // O(n²)

```
{
    for i := 1 to n do          // O(n)
    {
        S[i] := false;
        dist[i] = cost[v, i];
    }
```

S[v] := true;
dist[v] := 0.0;
for num := 2 to n-1 do //O(n-1)
{
choose u among those vertices not in S
such that dist[u] is minimum
S[u] := true;
for (each w adjacent to u with S[w] := false)
{
if (dist[w] > dist[u] + cost[u,w]) then
dist[w] := dist[u] + cost[u,w];
}
}
}

- Logn times for choosing using red black tree
- E to perform operation

$T_{Total} = (E \log n)$

→

g.01.19

→ Job Sequencing with Deadlines

→ Problem statement : 'n' no of jobs $j_1, j_2, j_3, \ldots, j_n$
Each job has profit $p_1, p_2, p_3, \ldots, p_n$. To complete each job, each job has deadline $d_1, d_2, d_3, \ldots, d_n$
we have a single machine. It takes 1 unit of time to complete each job on the machine, within its deadline. We have to maximize the profit. If we have 'n' no of jobs, we have to find a subset of jobs (set of 'k' jobs) to maximize profit $\sum_{i=1}^{k} p_i$, $k \leq n$

→ We use Greedy Approach to provide the solution

→ Example:

n = 4

job	1	2	3	4
profit	100	10	15	27
deadline	2	1	2	1

→ job (1,2) ; profit = 100 + 10 = 110 sequence of jobs : 2,1
→ 1,3 -> 100 + 15 = 115 => 1,3 or 3,1
→ 1,4 => 100 + 27 = 127 => 4,1 (deadline of 4,3 deadline of 1
→ 1 => 100
→ 2 => 10
→ 3 => 15
→ 4 => 27

0 - 1 - 2 => Machine Cycles (0 to 1, 1 to 2)

∴ Feasible solution = (4, 1) = Maximum Profit (Jobs)

Scanned by CamScanner

→ Order of input: Consider the jobs that have maximum profit. Arrange all the jobs in the decreasing order of their profit. $P_1 \geq P_2 \geq P_3 \geq \cdots \geq P_n$

Constraint is to complete jobs by deadline to get maximum profit. Arrange all the jobs in the increasing order of their deadlines. $d_3 \leq d_2 \leq d_3 \leq \cdots \leq d_n$

This gives feasible solution (sequence $d_1 \leq d_2 \leq d_3 \leq \cdots \leq d_n$

→ Theorem to prove that $d_1 \leq d_2 \leq d_3 \leq \cdots \leq d_n$ is feasible sol

n → no. of jobs

Assume: $d_1 \leq d_2 \leq d_3 \leq \cdots \leq d_k$ → is a feasible solution

Assume: $\sigma = i_1, i_2, i_3, \cdots, i_k$ → is another feasible solution

Assume: $\sigma' = r_1, r_2, r_3, \cdots, r_k$ → is another feasible solution

⇓

Compare whether σ, σ' are equal or not; i.e.,

$i_1 = r_1, i_2 = r_2, \cdots, i_n = r_n$

If least sequence $i_a \neq r_a$ (smallest index) for least a, if we find a sequence later where $i_a = r_p$, swap r_a with r_b and apply successive swap operations until $\sigma = \sigma'$. This sequence (new) is assumed to be σ''. This sequence is feasible solution.

=) This sequence always produces feasible solution

→ Assume: $j_1, j_2, j_3, \cdots, j_k$ produces feasible solution. If we want to add new job j to this sequence, (insert) at $d_i \geq$ insert

we have to verify whether the solution produced is feasible or not. For that, we have to find out the location where we have to insert new job j_i such that $d_{i-1} \leq d_i$. After inserting j_i, we have to shift remaining jobs after it one position to their right, and verify again if the solution (remaining jobs) meet their deadlines or not, i.e. if the solution is feasible or not.

→ Algorithm JS (d, J, n) → 1st approach

//General criteria: $d_i \leq d_{i+1}$

```
{
    d[0] := J[0] := 0;   // dummy job
    J[1] := {1};  // ∵ P1 ≥ P2 ≥ P3 ≥ ... ≥ Pn
    k := 1,  // feasible solution is provided upto 'k'
    for i := 2 to n do //to place remaining jobs in order
    {
        r := k;
        while (d[J[r]] > d[i] and d[J[r]] != r) //to find correct position of new job
            r := r - 1;
        if (d[J[r]] <= d[i] and d[i] > r) then // if di-1 ≤ di
        {
            for z = k to r+1 step-1 do
            {
                J[q+1] := J[z];
                J[r+1] := i;
                k := k+1;
            }
        }
    return j;
```

→ Approach 2:

$d_1 \leq d_2 \leq d_3 \leq \cdots\cdots \leq d_n$

↳ This sequence produces both feasible and optimal sol

↳ We have to prove whether this sequence produces optimal solution or not

'I' is a sequence of jobs produced by Greedy Approach

Greedy Approach: $P_1 \geq P_2 \geq P_3 \geq \cdots\cdots \geq P_n$

'J' is a sequence of jobs which is produced by Optimal Sol

We have to prove that these two sequences are equal or

$I \nsubseteq J$ → Two sequences are incomparable, ruled out

$I \subset J$ → Ruled Out (⊂ → subset)

$J \subset I$ → Ruled Out

Two jobs 'a' and 'b' with profits P_a and P_b respectively

If $P_a > P_b$, job 'a' comes before job 'b' in greedy sol

If $P_b > P_a$, job 'b' comes before job 'a' in greedy sol

I ——————

J ——————

↳ verify in backward approach (from the back).

If some job 'x' $\overset{\text{in } I}{}$ is not equal to the job in 'J' at the that position. find the position where the job is equal and swap the locations in J. Similarly, if some job in J is not equal to the job in 'I' at that position find position where the job is equal and swap the locations after series of swaps, we finally get I = J.

This is feasible as well as optimal solution.

→ Example:

Job	J1	J2	J3	J4	J5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

If any two jobs have same profits, take the input order of the jobs. First process Job i (J3), then Job k (J5). such that $i < k$.

=> dmax = 3

n = 5

Slots 0-1-2-3 (0 to 1, 1 to 2, 2 to 3)

Initially:

Slot 1	Slot 2	Slot 3
Empty	Empty	Empty

no. of slots = max deadline

Find: k = min (d[i], dmax)

Arrange all jobs in decreasing order of profits

J2	J1	J4	J3	J5	→ Job
1	2	2	3	1	→ Deadline
100	60	40	20	20	→ Profit

i → 1, 2, 3, 4, 5

* i = 1, d[i] = 1 dmax = 3

↳ k = min (d[i], dmax) = min (1, 3) = 1 k ≥ 1 ✓

Verify slot k is empty, assign job if empty.

Slot 1	Slot 2	Slot 3
J2	Empty	Empty

* i = 2 , d[i] = 2 , dmax = 3

k = min(d[i], dmax) = min(2, 3) = 2

k ≥ 1 :) 2 ≥ 1 ✓

Is slot k empty? Fill it with job i if yes

Else, reduce k by 1 (k = k - 1)

Slot 1	Slot 2	Slot 3
J2	J1	Empty

* i = 3, d[i] = 2, dmax = 3

k = min(d[i], dmax) = min(2, 3) = 2

k ≥ 1 :) 2 ≥ 1 ✓

Slot k is empty → x

k = k - 1 :) k = 1 :) 1 ≥ 1

Slot k is empty → x

k = k - 1 :) k = 0 :) 0 ≥ 1 x

* i = 4, d[i] = 3, dmax = 3

k = min(d[i], dmax) = min(3, 3) = 3

k ≥ 1 :) 3 ≥ 1

Is slot k empty? Fill it with job i if yes

Slot 1	Slot 2	Slot 3
J2	J1	J4

Hence, we have filled the table

Maximum Profit = P2 + P1 + P4 = 100 + 80 + 20 = 180

→ Example:

n = 5

Pi = (20, 15, 10, 5, 1)

di = (2, 2, 1, 3, 3)

* i = 1, d[i] = 2, dmax = 3

k = min(d[i], dmax) = 2

k ≥ 1 :) 2 ≥ 1 ✓

Slot 2 is empty ✓

[0,1] [1,2] [2,3]

Slot 1	Slot 2	Slot 3
-	J1	-

* i = 2, d[i] = 2, dmax = 3

k = min(d[i], dmax) = 2

k ≥ 1 :) 2 ≥ 1 ✓

Slot 2 is empty x

k = k - 1 : 1 :) k ≥ 1 :) 1 ≥ 1 ✓

Slot 1 is empty

[0,1] [1,2] [2,3]

Slot 1	Slot 2	Slot 3
J2	J1	-

* i = 3, d[i] = 1, dmax = 3

k = min(d[i], dmax) = 1

k ≥ 1 :) 1 ≥ 1 ✓

Slot 1 is not empty x

k = k - 1 : 0 :) k ≥ 1 :) 1 ≥ 1 x

[0,1] [1,2] [2,3]

Slot 1	Slot 2	Slot 3
J2	J1	-

* i = 4, d[i] = 3, dmax = 3

k = min(d[i], d[max]) = 3

k ≥ 1 :) 3 ≥ 1 ✓

Slot 3 is empty ✓

[0,1] [1,2] [2,3]

Slot 1	Slot 2	Slot 3
J2	J1	J4

:) Maximum Profit = P2 + P1 + P4 = ~~20 + 2 = 7~~

= 15 + 20 + 5 = 40