Design and Analysis of Algorithms

UNIT-2

Part – I : Greedy Method

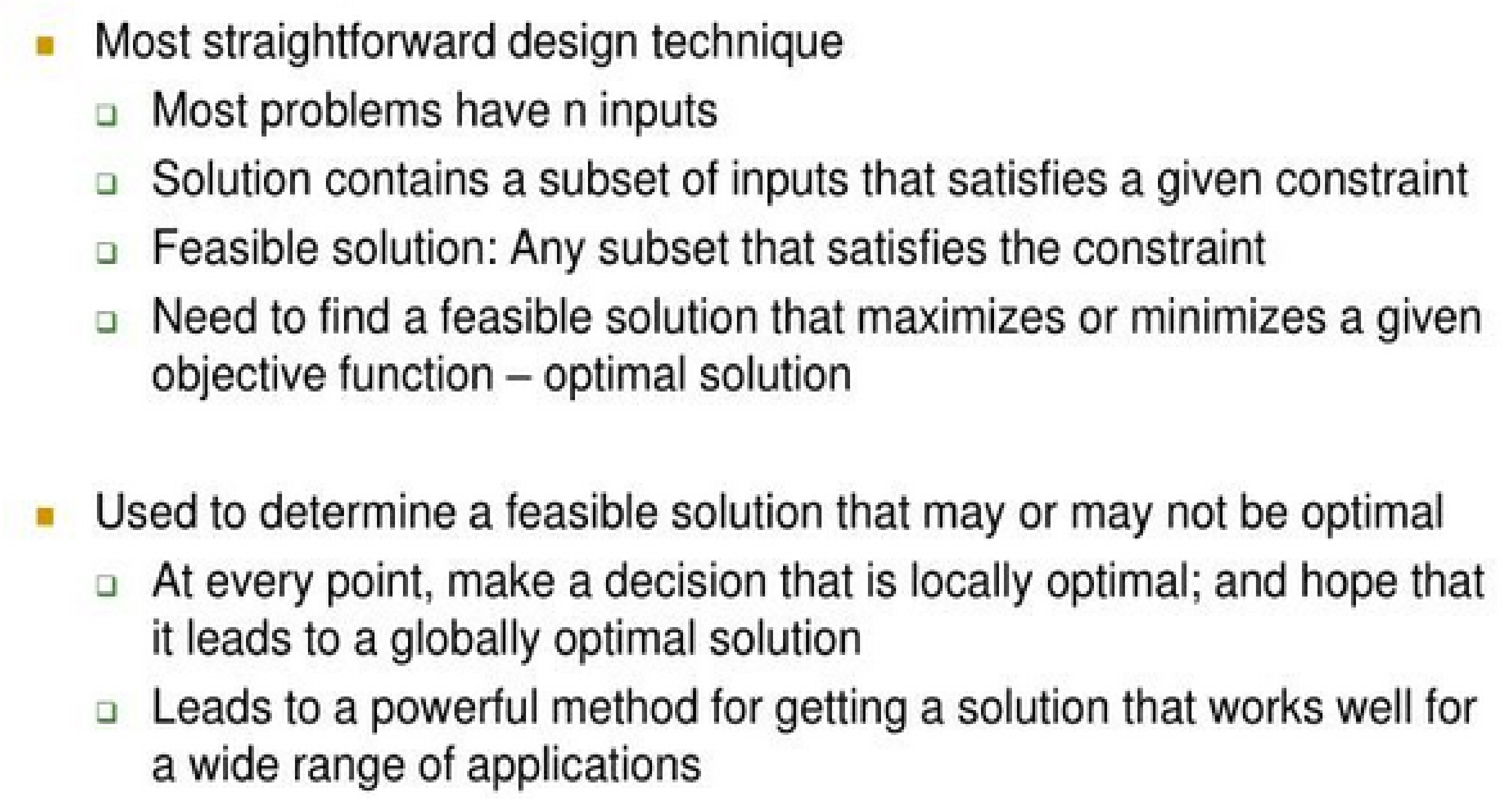
By

Y. Indira Priyadarshini
Assistant Professor
Department of Computer Science and Engineering

Contents:

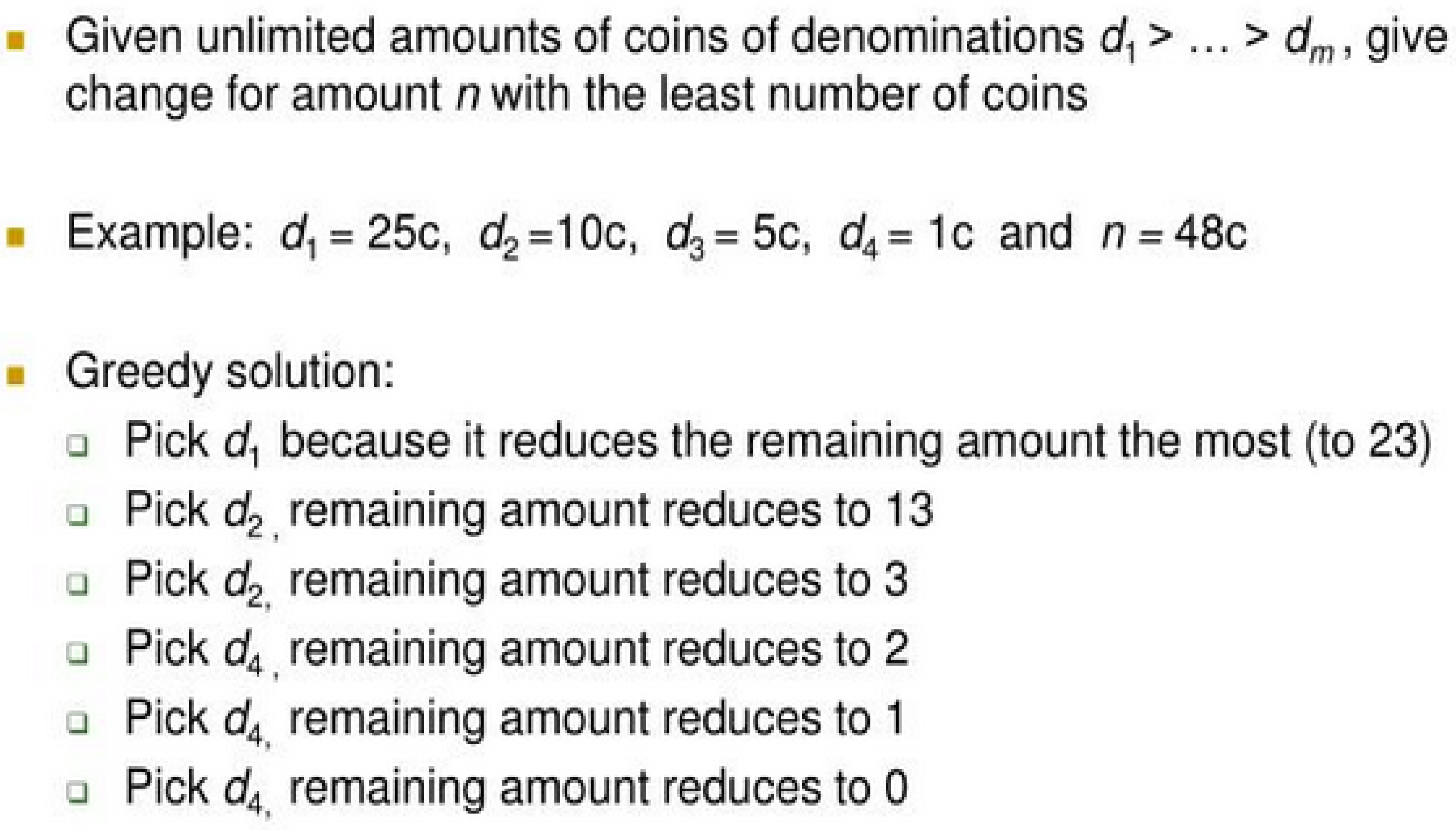
1. General method.
2. Knapsack problem.
3. Job Scheduling with Deadlines.
4. Minimum cost Spanning Trees.
5. Optimal storage on tapes.
6. Single-source shortest paths.

1.General method:



- Most straightforward design technique
  - Most problems have n inputs
  - Solution contains a subset of inputs that satisfies a given constraint
  - Feasible solution: Any subset that satisfies the constraint
  - Need to find a feasible solution that maximizes or minimizes a given objective function – optimal solution
- Used to determine a feasible solution that may or may not be optimal
  - At every point, make a decision that is locally optimal; and hope that it leads to a globally optimal solution
  - Leads to a powerful method for getting a solution that works well for a wide range of applications

Change –Making Problem:

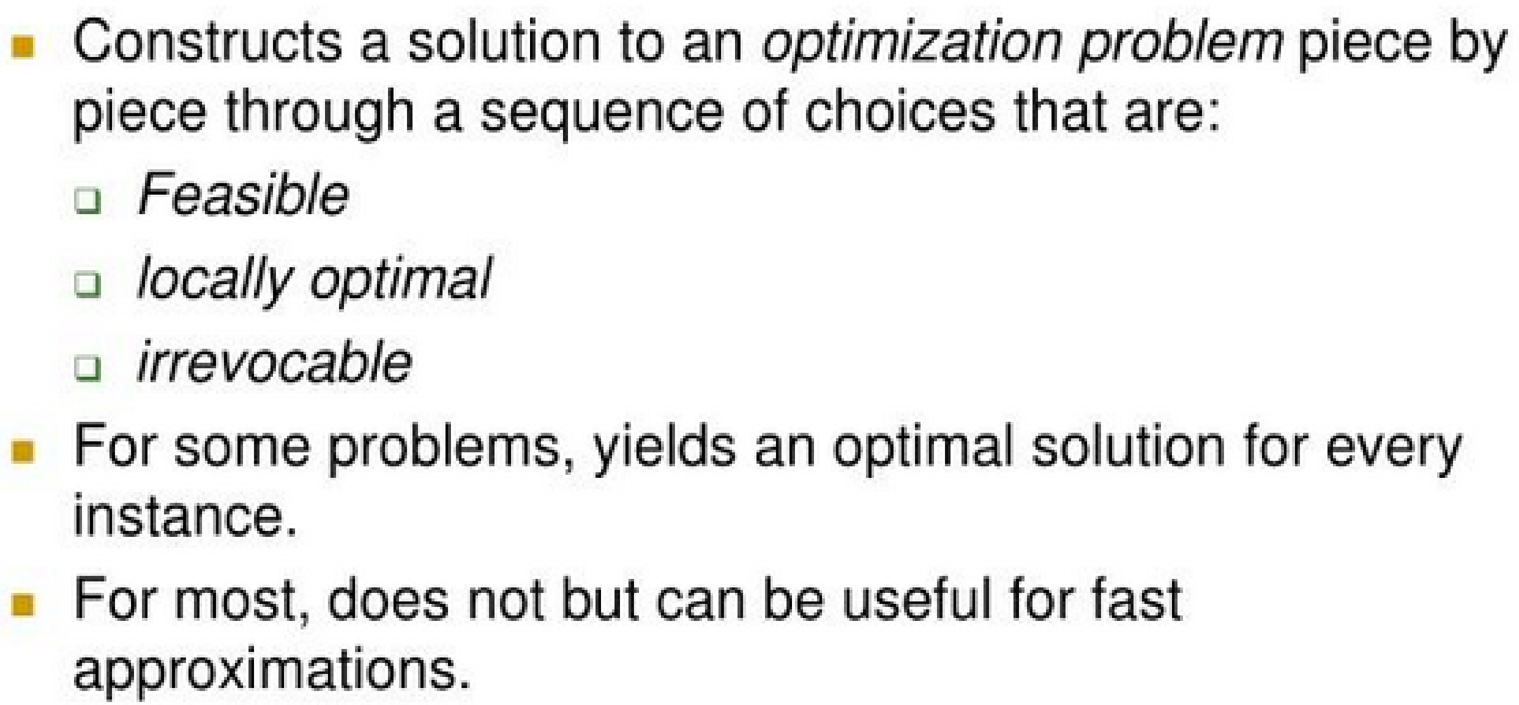


- Given unlimited amounts of coins of denominations $d_1 > \ldots > d_m$, give change for amount n with the least number of coins

- Example: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$

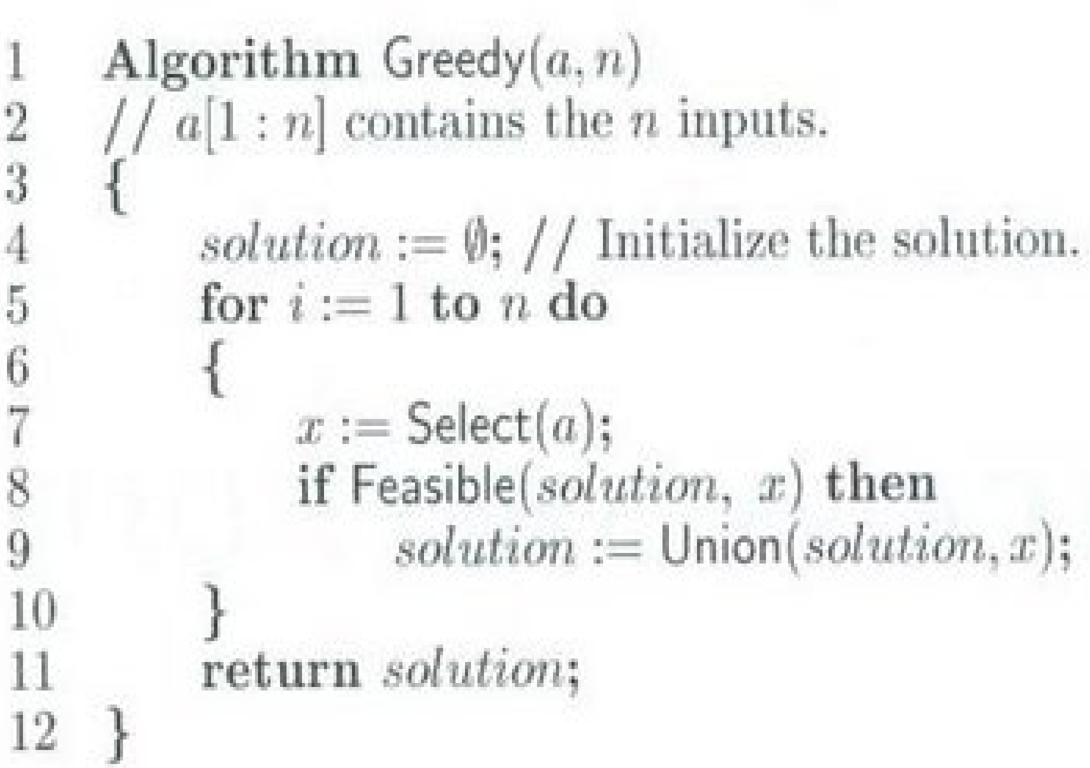
- Greedy solution:
  - Pick d_1 because it reduces the remaining amount the most (to 23)
  - Pick $d_{2\,,}$ remaining amount reduces to 13
  - Pick $d_{2,}$ remaining amount reduces to 3
  - Pick $d_{4\,,}$ remaining amount reduces to 2
  - Pick $d_{4,}$ remaining amount reduces to 1
  - Pick $d_{4,}$ remaining amount reduces to 0

Greedy Technique:



- Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:
  - *Feasible*
  - *locally optimal*
  - *irrevocable*
- For some problems, yields an optimal solution for every instance.
- For most, does not but can be useful for fast approximations.

The General Method:



```
Algorithm Greedy($a, n$)
// $a[1:n]$ contains the $n$ inputs.
{
    $solution := \emptyset$; // Initialize the solution.
    for $i := 1$ to $n$ do
    {
        $x :=$ Select($a$);
        if Feasible($solution, x$) then
            $solution :=$ Union($solution, x$);
    }
    return $solution$;
}
```

2. Knapsack problem:

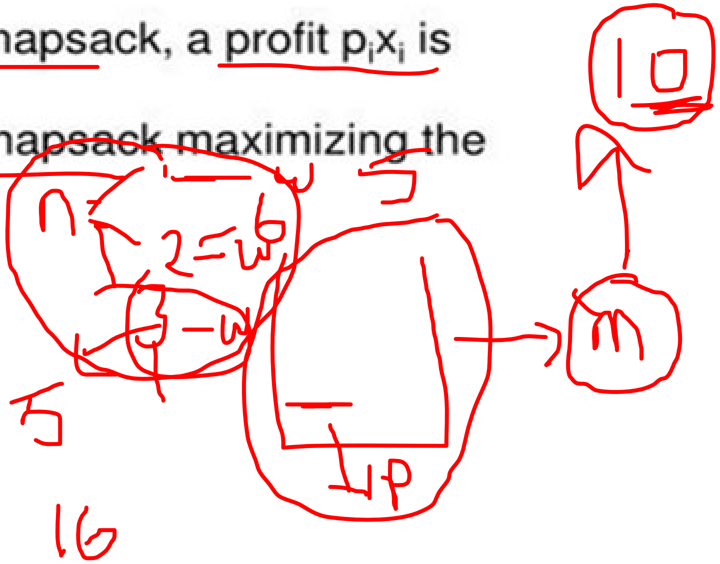
→ Greedy

- Problem definition
  - Given n objects and a knapsack where object i has a weight w_i and the knapsack has a capacity m
  - If a fraction x_i of object i placed into knapsack, a profit $p_i x_i$ is earned
  - The objective is to obtain a filling of knapsack maximizing the total profit

$$maximize \sum_{1 \le i \le n} p_i x_i \qquad (4.1)$$

$$subject\ to \sum_{1 \le i \le n} w_i x_i \le m \qquad (4.2)$$

$$and\ 0 \le x_i \le 1, \quad 1 \le i \le n \qquad (4.3)$$

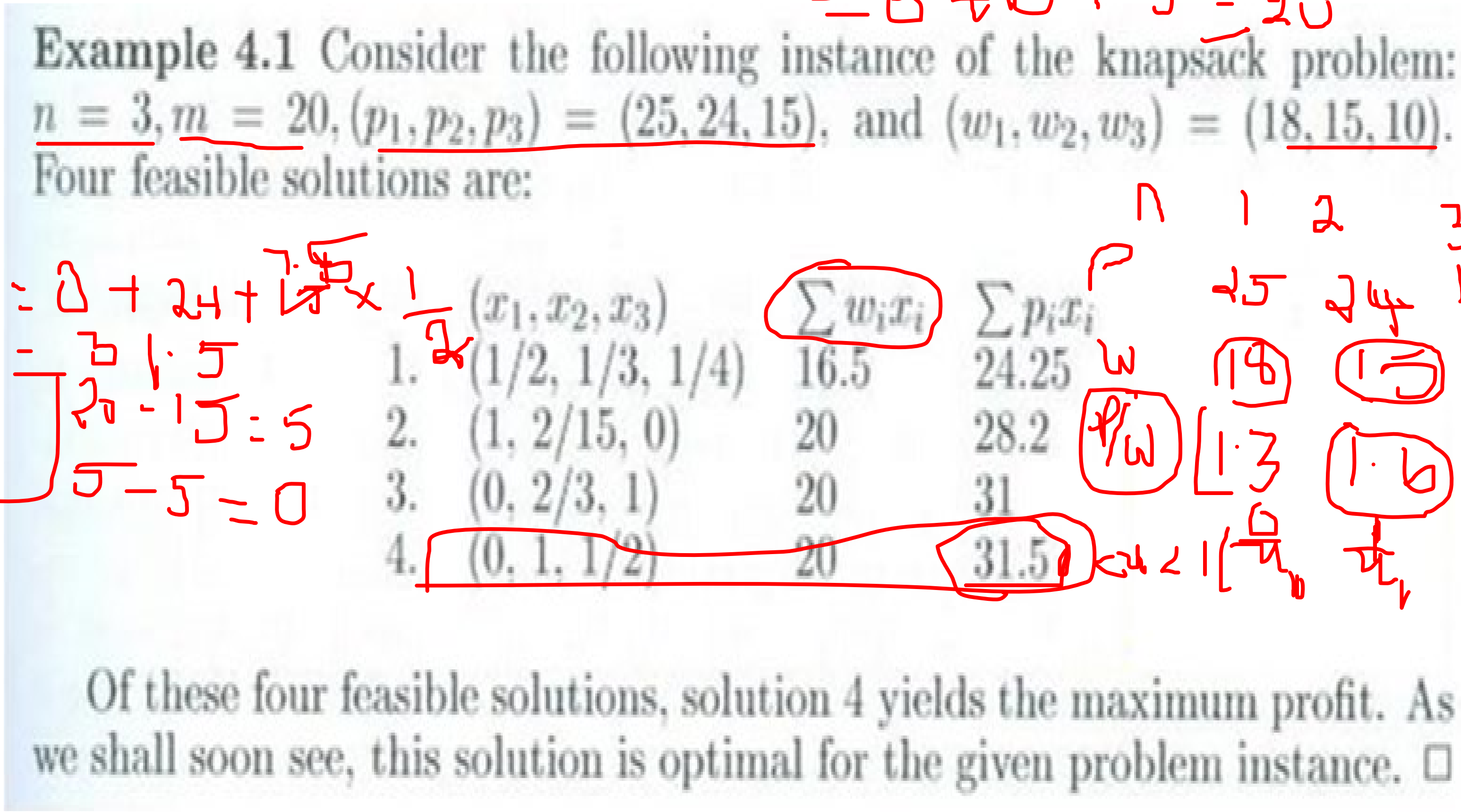


- A *feasible solution* is any set satisfying (4.2) and (4.3)
- An *optimal solution* is a feasible solution for which (4.1) is maximized

Example:

$\sum w_i x_i = 18\times 0 + 15\times 1 + 10\times 1/2$

$= 0 + 15 + 5 = 20$



Example 4.1 Consider the following instance of the knapsack problem: $n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

$\sum p_i x_i = 0 + 24 + 15\times\frac{1}{2}$ (7.5)

$= 31.5$

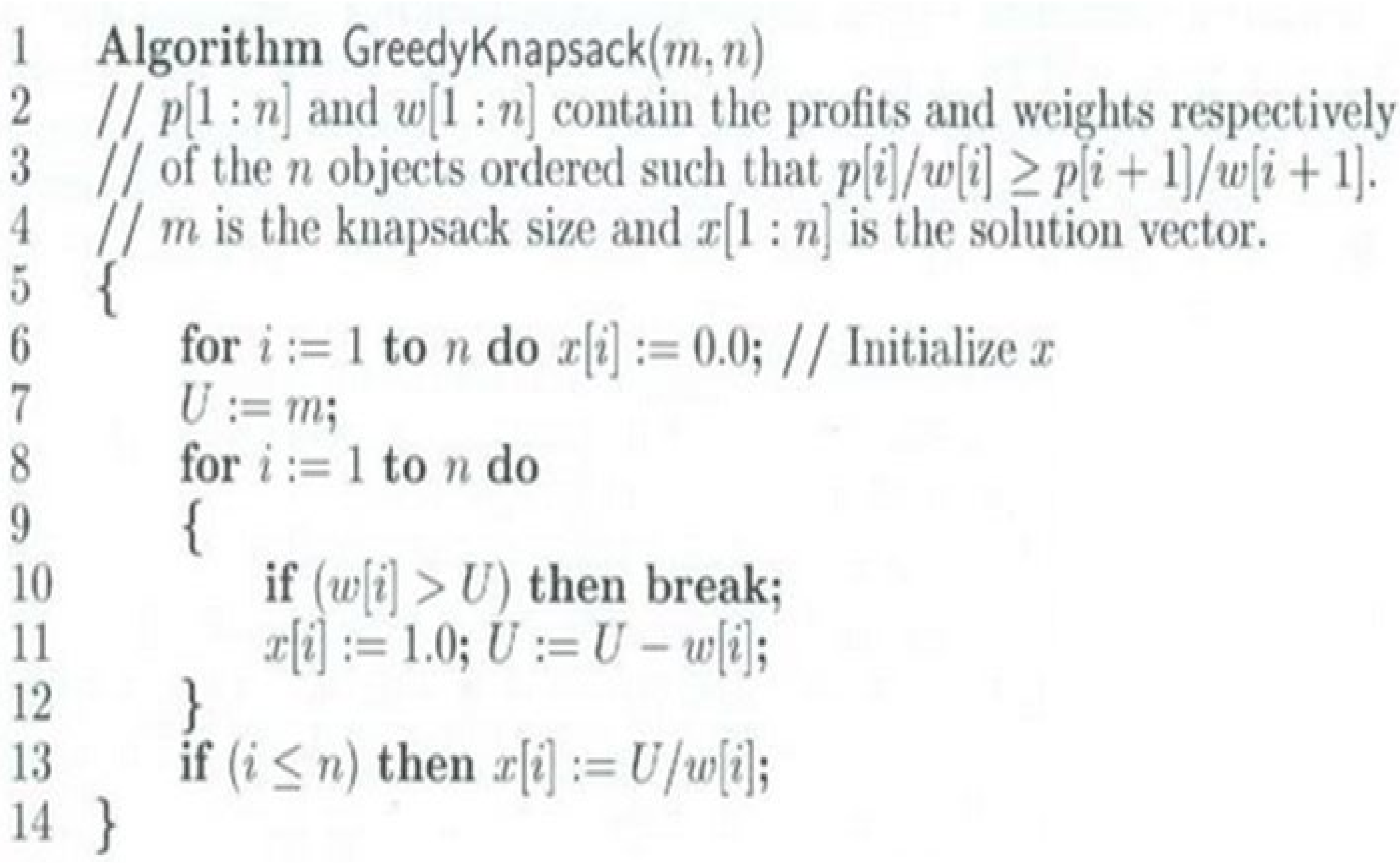
$20 - 15 = 5$

$5 - 5 = 0$

20

n	1	2	3
P	25	24	15
W	18	15	10
P/W	1.3	1.6	1.5

$(x_1 = 0,\ x_2 = 1,\ x_3 = \frac{1}{2})$



```
Algorithm GreedyKnapsack($m, n$)
// $p[1:n]$ and $w[1:n]$ contain the profits and weights respectively
// of the $n$ objects ordered such that $p[i]/w[i] \geq p[i+1]/w[i+1]$.
// $m$ is the knapsack size and $x[1:n]$ is the solution vector.
{
    for $i := 1$ to $n$ do $x[i] := 0.0$; // Initialize $x$
    $U := m$;
    for $i := 1$ to $n$ do
    {
        if ($w[i] > U$) then break;
        $x[i] := 1.0$; $U := U - w[i]$;
    }
    if ($i \leq n$) then $x[i] := U/w[i]$;
}
```

Example:

- Find an optimal solution to the knapsack instance

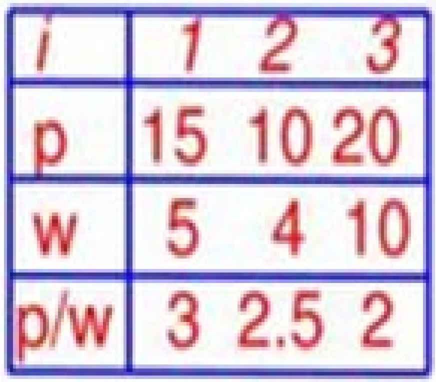
  n=7

  m=15

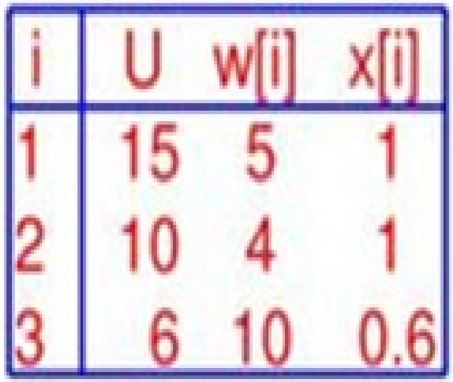
  (p1, p2, …, p7) = (10, 5, 15, 7, 6, 18, 3)

  (w1, w2, …, w7) = (2, 3, 5, 7, 1, 4, 1)

$m = 15$



i	1	2	3
p	15	10	20
w	5	4	10
p/w	3	2.5	2



i	U	w[i]	x[i]
1	15	5	1
2	10	4	1
3	6	10	0.6

$P = 15 + 10 + 20 \times 0.6 = 37$

- Time complexity
  - Sorting: O(n log n) using fast sorting algorithm like merge sort
  - GreedyKnapsack: O(n)
  - So, total time is O(n log n)

- If $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$, then GreedyKnapsack generates an optimal solution to the given instance of the knapsack problem.

3. Job Scheduling with Deadlines:

n = 3 {J_1, J_2, J_3}

P 15 10 5

D 2 2 1

1 unit / 1hr

J_1

- we are given a set of 'n' jobs.
- Associated with each Job i, deadline $d_i > 0$ and profit $P_i > 0$.
- For any job 'i' the profit pi is earned iff the job is completed by its deadline.
- Only one machine is available for processing jobs.
- A feasible solution is a subset J of jobs such that each job in this subset can be completed by its deadline and the total profit is the sum of the jobs profit in J.
- An optimal solution is the feasible solution with maximum profit.

{J_2, 1} {J_3, 1} {J_1, 2} {J_1}

10 + 15

= 25

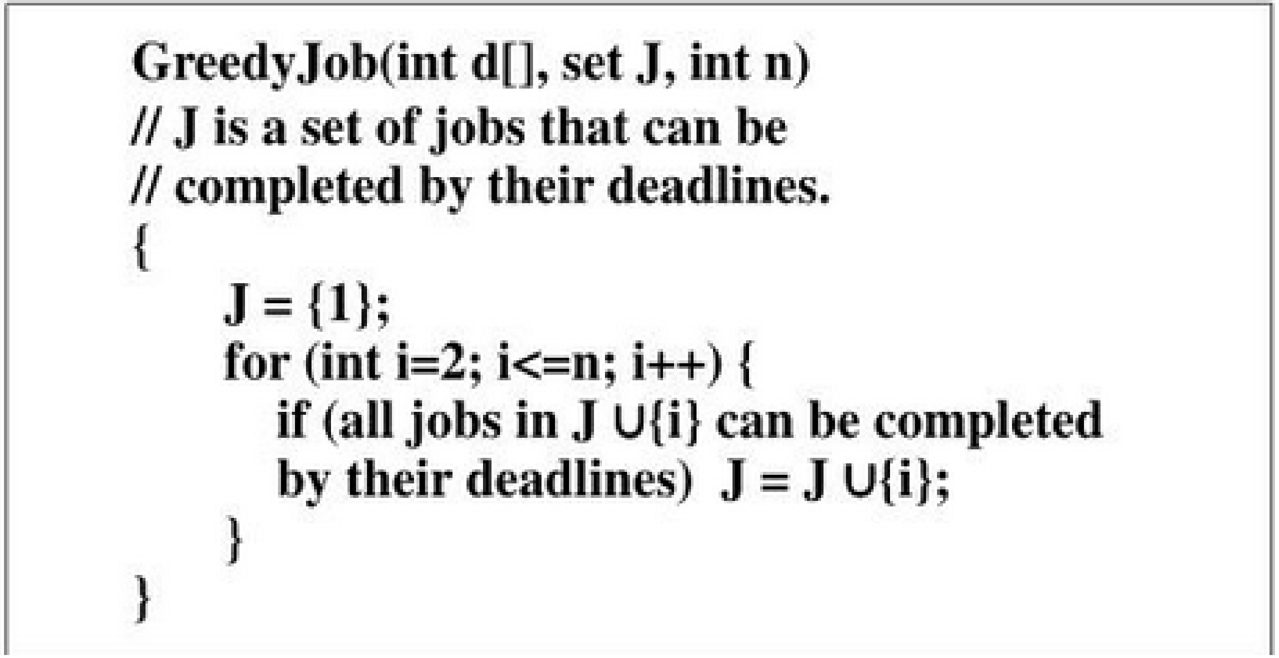
- Example
  - n=4, $(p_1,p_2,p_3,p_4)=(100,10,15,\ 27)$,
    $(d_1,d_2,d_3,d_4)=(\ 2,\ \ 1,\ \ 2,\ 1)$

	Feasible Solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

- Greedy strategy using *total profit* as optimization function
  - Applying to Example 4.2
    - Begin with J=ϕ
    - Job 1 considered, and added to J → J={1}
    - Job 4 considered, and added to J → J={1,4}
    - Job 3 considered, but discarded because not feasible → J={1,4}
    - Job 2 considered, but discarded because not feasible → J={1,4}
    - Final solution is J={1,4} with total profit 127
  - It is optimal

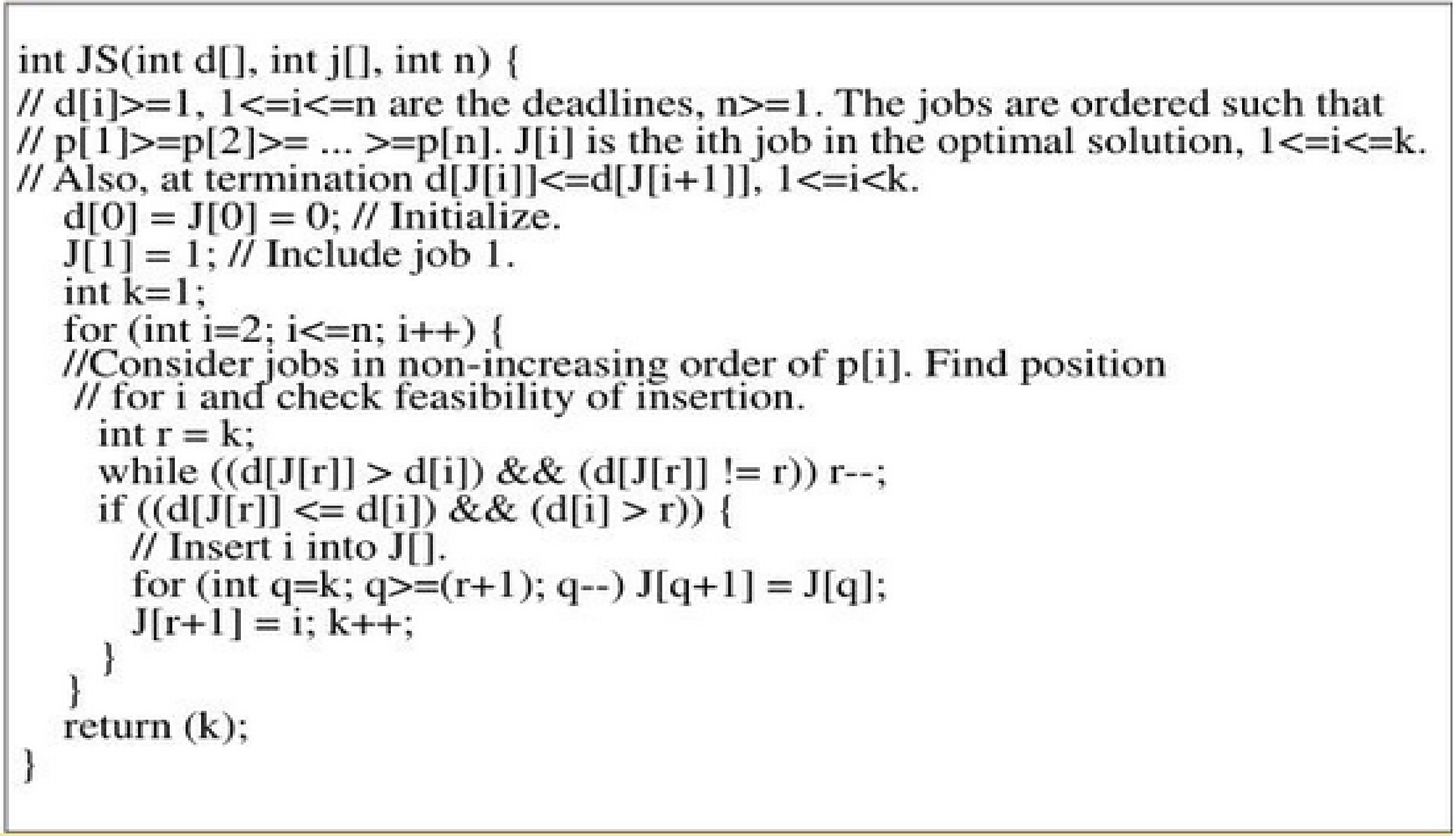
- How to determine the feasibility of J ?
  - Trying out all the permutations
    - Computational explosion since there are n! permutations
  - Possible by checking only one permutation
    - By Theorem 4.3
- Theorem 4.3 Let *J* be a set of *k* jobs and $\sigma = i_1, i_2, \ldots, i_k$ a permutation of jobs in *J* such that $d_{i1} \le d_{i2} \le \ldots \le d_{ik}$. Then *J* is a feasible solution iff the jobs in *J* can be processed in the order without violating any deadline.

- Theorem 4.4 The greedy method described above always obtains an optimal solution to the job sequencing problem.
- High level description of job sequencing algorithm
  - **Assuming the jobs are ordered such that p[1]≥p[2]≥…≥p[n]**



```
GreedyJob(int d[], set J, int n)
// J is a set of jobs that can be
// completed by their deadlines.
{
      J = {1};
      for (int i=2; i<=n; i++) {
         if (all jobs in J ∪{i} can be completed
         by their deadlines)  J = J ∪{i};
      }
}
```

- How to implement ?
  - How to represent *J* to avoid sorting the jobs in *J* each time ?
    - 1-D array J[1:k] such that J[r], $1 \le r \le k$, are the jobs in *J* and $d[J[1]] \le d[J[2]] \le \ldots . \le d[J[k]]$
    - To test whether $J \cup \{i\}$ is feasible, just insert i into *J* preserving the deadline ordering and then verify that $d[J[r]] \le r$, $1 \le r \le k+1$



```
int JS(int d[], int j[], int n) {
// d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs are ordered such that
// p[1]>=p[2]>= ... >=p[n]. J[i] is the ith job in the optimal solution, 1<=i<=k.
// Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
    d[0] = J[0] = 0; // Initialize.
    J[1] = 1; // Include job 1.
    int k=1;
    for (int i=2; i<=n; i++) {
    //Consider jobs in non-increasing order of p[i]. Find position
     // for i and check feasibility of insertion.
       int r = k;
       while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
       if ((d[J[r]] <= d[i]) && (d[i] > r)) {
          // Insert i into J[].
          for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
          J[r+1] = i; k++;
       }
    }
    return (k);
}
```

- Computing time of JS can be reduced from $O(n^2)$ to $O(n)$ by using disjoint set union and find algorithm

- Let J be a feasible subset of jobs, processing time of each job can be determined by the rule
  - If job i is not assigned a processing time, then assign to the slot $[\alpha-1, \alpha]$ where α is the largest r such that $1 \leq r \leq d_i$ and slot $[\alpha-1, \alpha]$ is free.
  - If for a new job there is no free α , then it can not be included in J.

n = 5 J_1 J_2 J_3 J_4 J_5

P   20   15   10   5   1

d   2   2   1   3   3

0 – 1

Let n=5,

D 3 2 1 4 2 J 1 3

1 15 10 20 11 5 12 = 40

$(p_1,\ldots, p_5)$= (20,15,10, 5, 1) and

$(d_1,\ldots, d_5)$= (2,   2,   1,   3, 3)

J	Assigned slots	Job Considered	Action	Profit
Φ	none	1	Assign to [1, 2]	0
{1}	[1, 2]	2	Assign to [0, 1]	20 (J_1)
{1, 2}	[0, 1], [1, 2]	3	Can't fit; Reject	35 (J_2+J_1)
{1, 2}	[0, 1], [1, 2]	4	Assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2],[2, 3]	5	Reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40

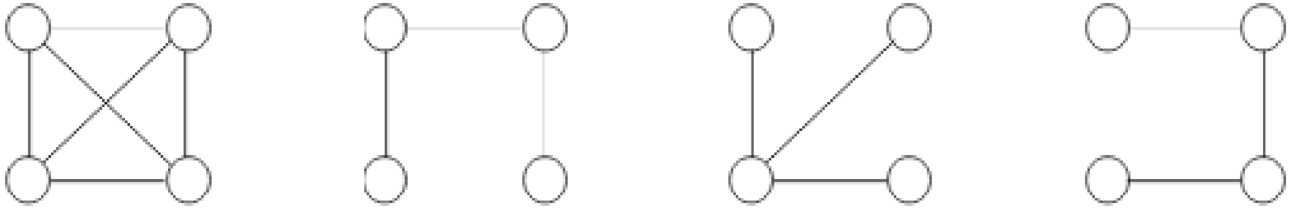
- N=7

jobs	J1	J2	J3	J4	J5	J6	J7
profit	35	30	25	2	15	5	4
deadlines	3	4	4	2	3	1	2

4. Minimum cost Spanning Trees :

- Definition 4.1 Let*G=(V, E*) be at undirected connected graph. A subgraph*t=(V, E')* of *G* is a *spanning tree*of *G* iff t is a tree.

- Example 4.5

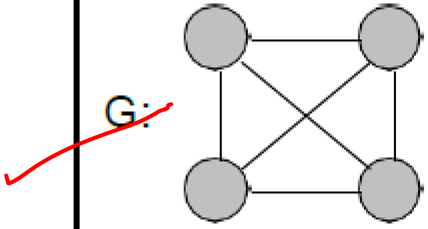


Spanning trees

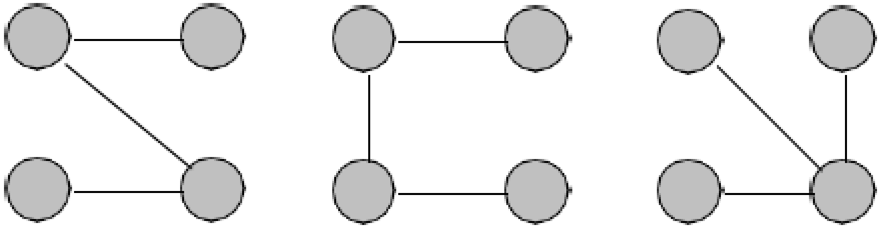
- Applications
  - Obtaining an independent set of circuit equations for an electric network
  - etc

- A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

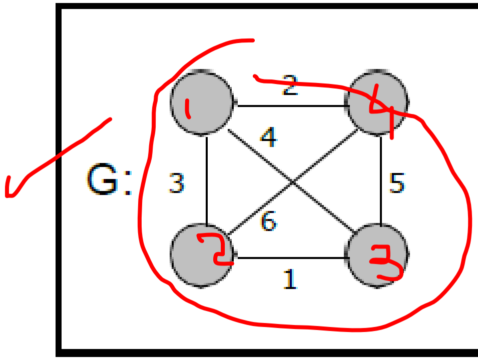
- Weight of a spanning tree w (T) is the sum of weights of all edges in T. The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.



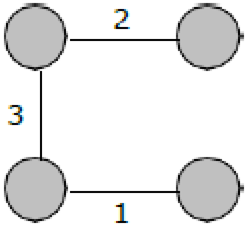
A graph G:



Three (of many possible) spanning trees from graph G:



A weighted graph G:



The minimal spanning tree from weighted graph G:

To find a Minimum Spanning Tree:

a)Kruskal's algorithm uses edges, in determining the MST.

b)Prim‟s algorithm uses vertex connections in determining the MST.

a)Kruskal’s Algorithm : → F

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows:

- Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added.
- Sometimes two or more edges may have the same cost.
- The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

```
Algorithm Kruskal (E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
        Construct a heap out of the edge costs using heapify;
        for  i := 1 to n do parent [i] := -1;
                                        // Each vertex is in a different set.
        i := 0; mincost := 0.0;
        while ((i < n -1) and (heap not empty)) do
        {
                Delete a minimum cost edge (u, v) from the heap and
                re-heapify using Adjust;
                j := Find (u); k := Find (v);
                if  (j ≠ k) then
                {
                        i := i + 1;
                        t [i, 1] := u; t [i, 2] := v;
                        mincost :=mincost + cost [u, v];
                        Union (j, k);
                }
        }
        if (i ≠ n-1) then write ("no spanning tree");
        else return mincost;
}
```

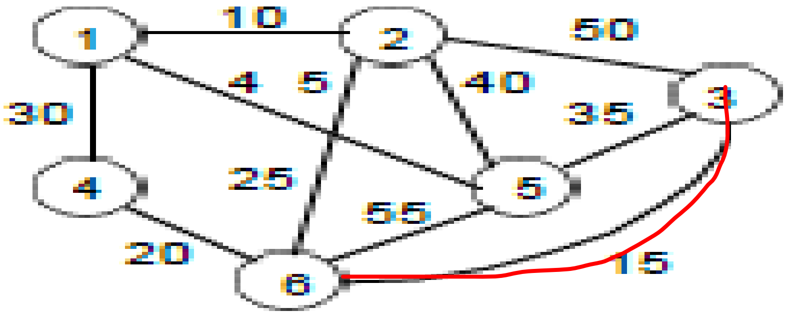
Running time:

- The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n + e).

- We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

- Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity

Example 1:



Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

b) Prim’s Algorithm :

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an Vertex with the least weight in a MST).

Prim’s algorithm works as follows:

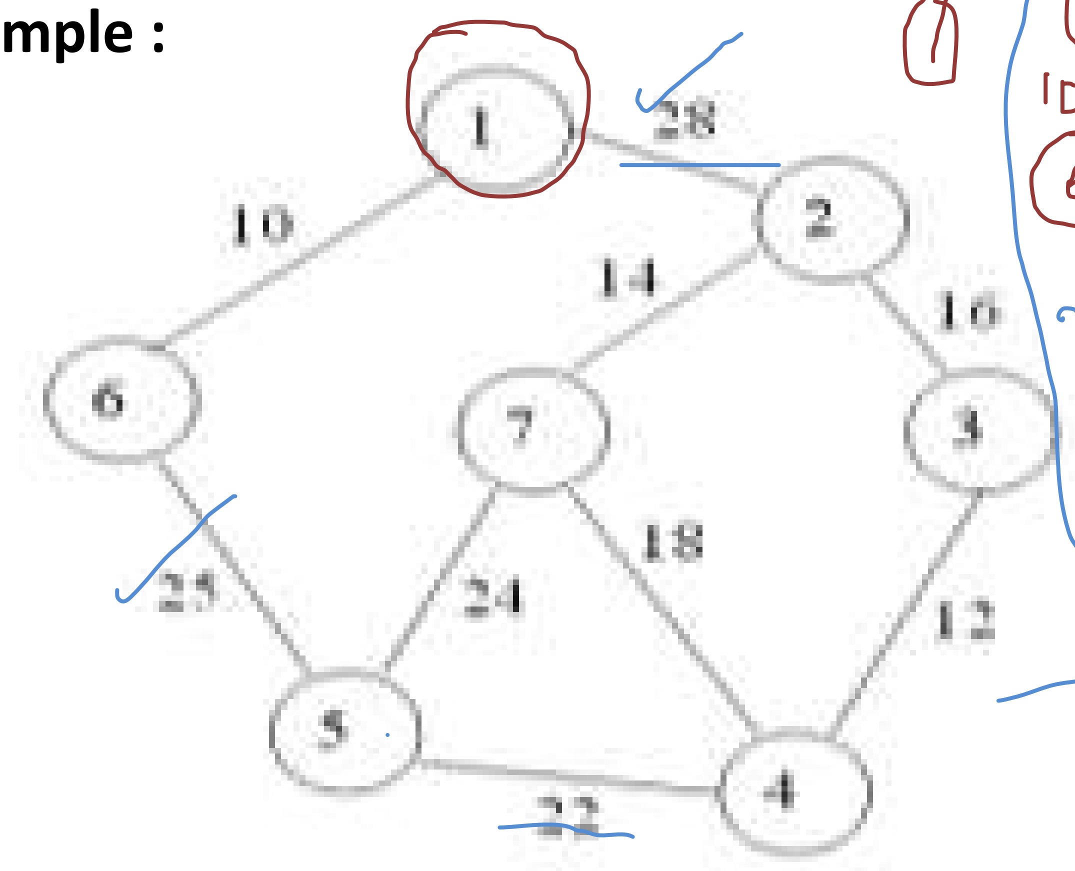
- In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.
- This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

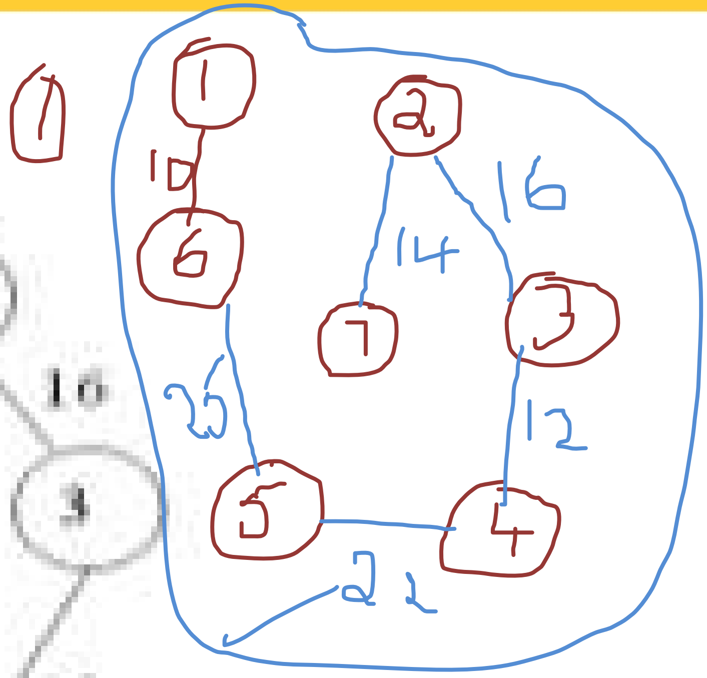
Algorithm Algorithm Prim

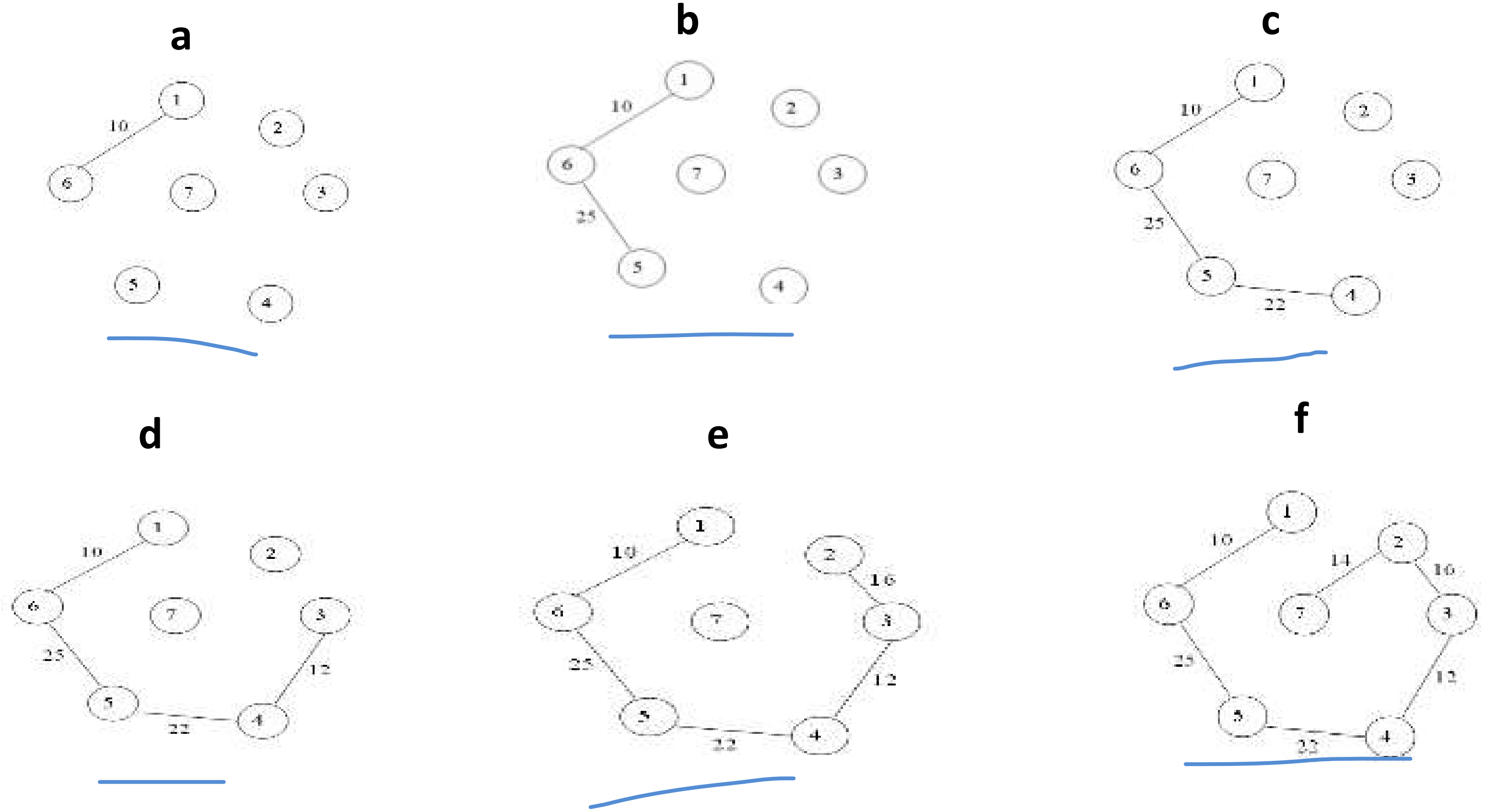
```
(E, cost, n, t)
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j] is
// either a positive real number or  if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
        Let (k, l) be an edge of minimum cost in E;
        mincost := cost [k, l];
        t [1, 1] := k; t [1, 2] := l;
        for   i :=1 to n do                             // Initialize near
                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                          // Find n - 2 additional edges for t.
        {
                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];
                mincost := mincost + cost [j, near [j]];
                near [j] := 0
                for   k:= 1 to n  do                            // Update near[].
                        if ((near [k]  0) and (cost [k, near [k]] > cost [k, j]))
                                then near [k] := j;
        }
        return mincost;
}
```

- The prims algorithm will start with a tree that includes only a minimum cost edge of G.
- Then, edges are added to the tree one by one. the next edge (i,j) to be added in such that I is a vertex included in the tree, j is a vertex not yet included, and cost of (i,j), cost[i,j] is minimum among all the edges.

Example :





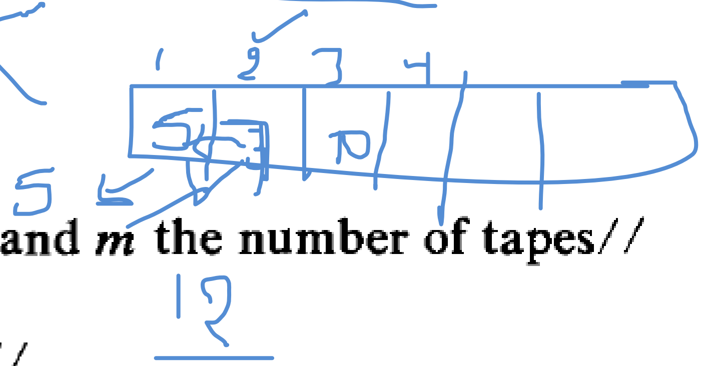

a
1
2
3
4
5
6
7
10
b
1
2
3
4
5
6
7
10
25
c
1
2
3
4
5
6
7
10
25
22
d
1
2
3
4
5
6
7
10
25
22
12
e
1
2
3
4
5
6
7
10
25
22
12
16
f
1
2
3
4
5
6
7
10
25
22
12
16
14

Running time:

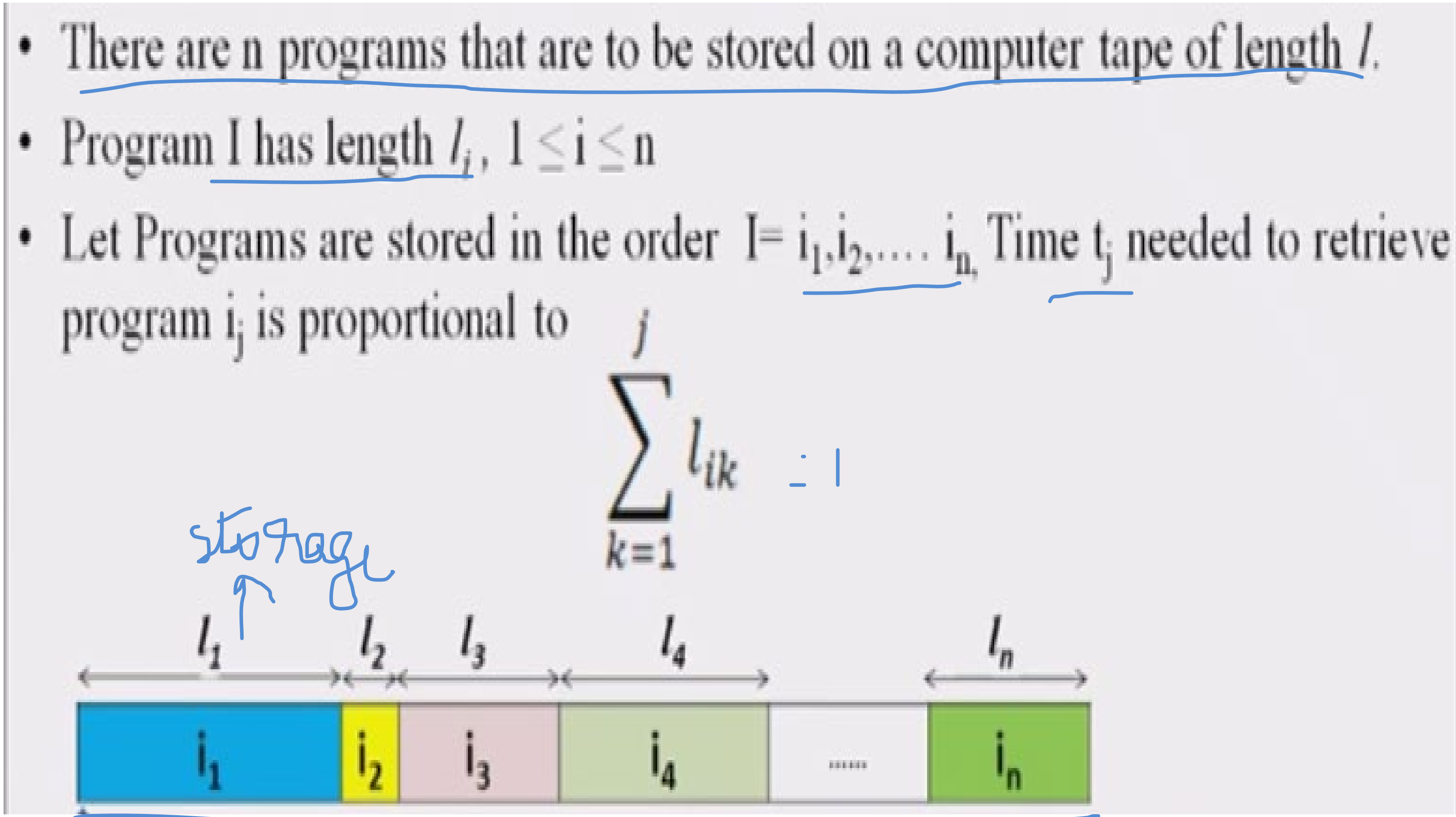
•We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get O (n2) time when we implement dist with array, O (n + E log n) when we implement it with a heap.

5.Optimal storage on tapes:

- Optimal Storage on Tapes is one of the application of the Greedy Method.
- The objective is to find the Optimal retrieval time for accessing programs that are stored on tape.



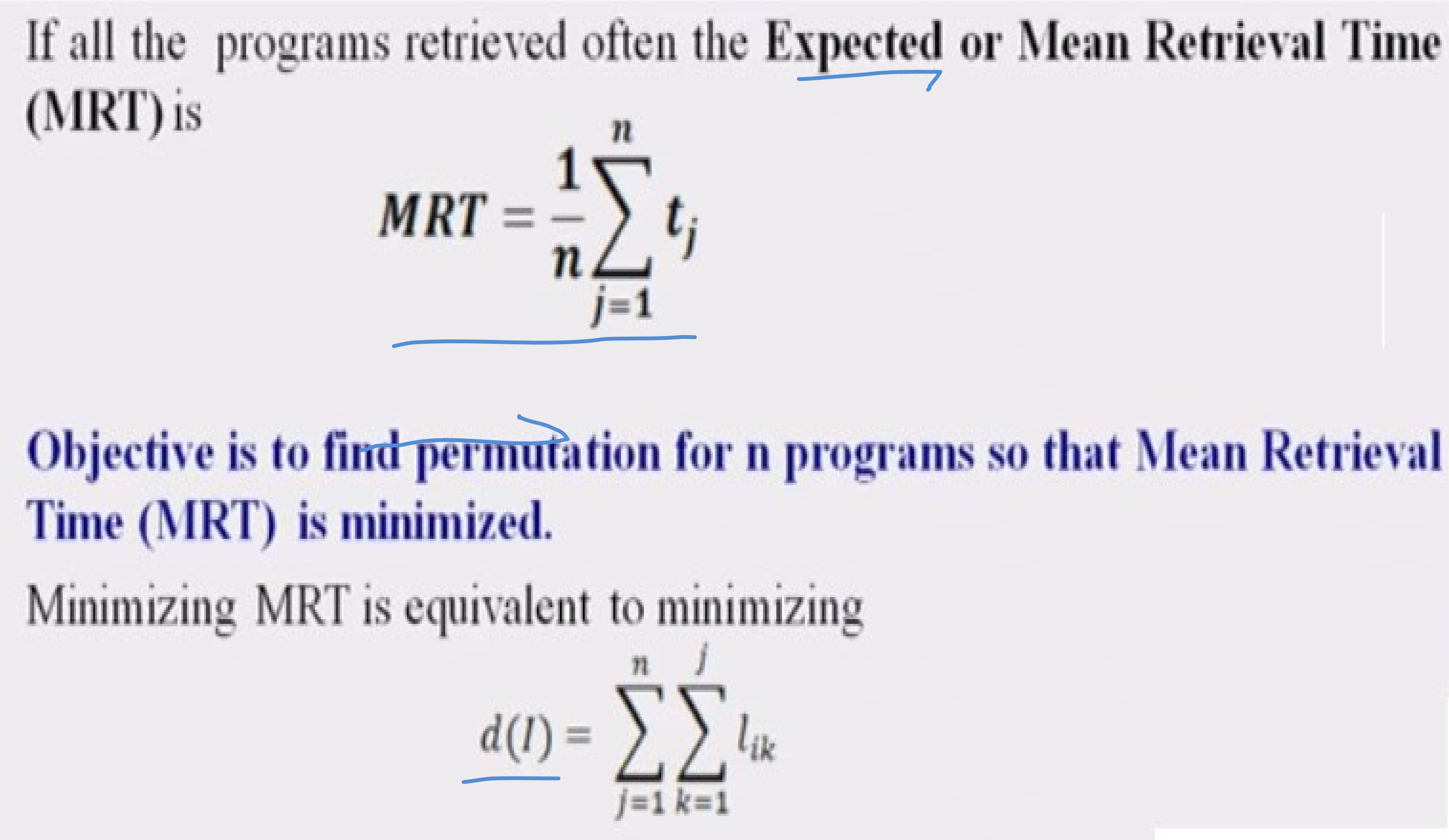
```
procedure STORE(n, m)
   //n is the number of programs and m the number of tapes//
   integer m, n, j
   j ← 0   //next tape to store on//
   for i ← 1 to n do
      print ('append program', i, 'to permutation for tape', j)
      j ← (j + 1) mod m
   repeat
end STORE
```



- There are n programs that are to be stored on a computer tape of length l.
- Program I has length l_i , $1 \leq i \leq n$
- Let Programs are stored in the order $I= i_1,i_2,\ldots\ i_n,$ Time t_j needed to retrieve program i_j is proportional to

$$\sum_{k=1}^{j} l_{ik} \quad : 1$$

storage



If all the programs retrieved often the Expected or Mean Retrieval Time (MRT) is

$$MRT = \frac{1}{n}\sum_{j=1}^{n} t_j$$

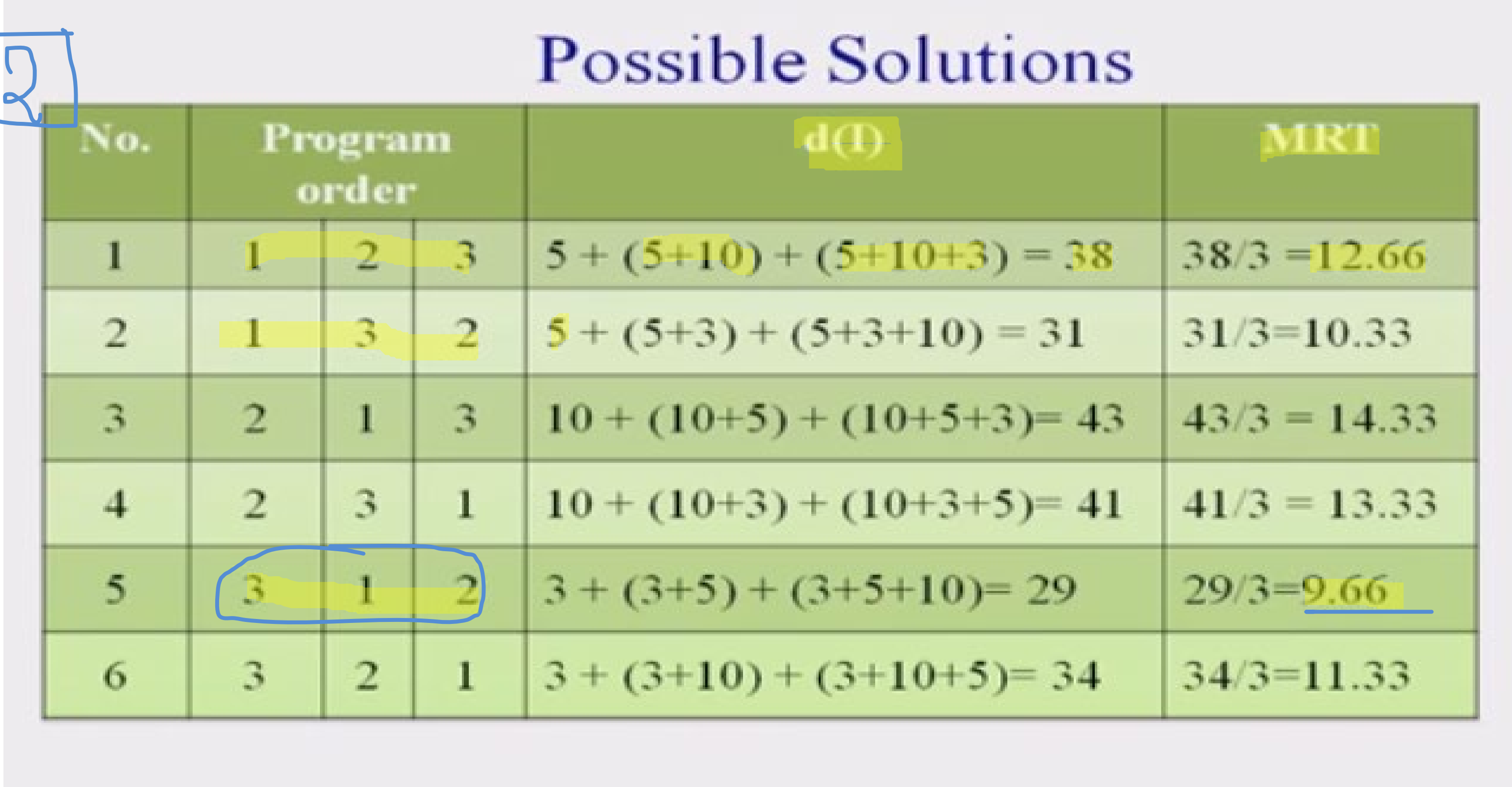
Objective is to find permutation for n programs so that Mean Retrieval Time (MRT) is minimized.

Minimizing MRT is equivalent to minimizing

$$d(I) = \sum_{j=1}^{n}\sum_{k=1}^{j} l_{ik}$$

Example 1 Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective D values are:

3 | 1 | 2



Possible Solutions

No.	Program order			d(I)	MRT
1	1	2	3	5 + (5+10) + (5+10+3) = 38	38/3 =12.66
2	1	3	2	5 + (5+3) + (5+3+10) = 31	31/3=10.33
3	2	1	3	10 + (10+5) + (10+5+3)= 43	43/3 = 14.33
4	2	3	1	10 + (10+3) + (10+3+5)= 41	41/3 = 13.33
5	3	1	2	3 + (3+5) + (3+5+10)= 29	29/3=9.66
6	3	2	1	3 + (3+10) + (3+10+5)= 34	34/3=11.33

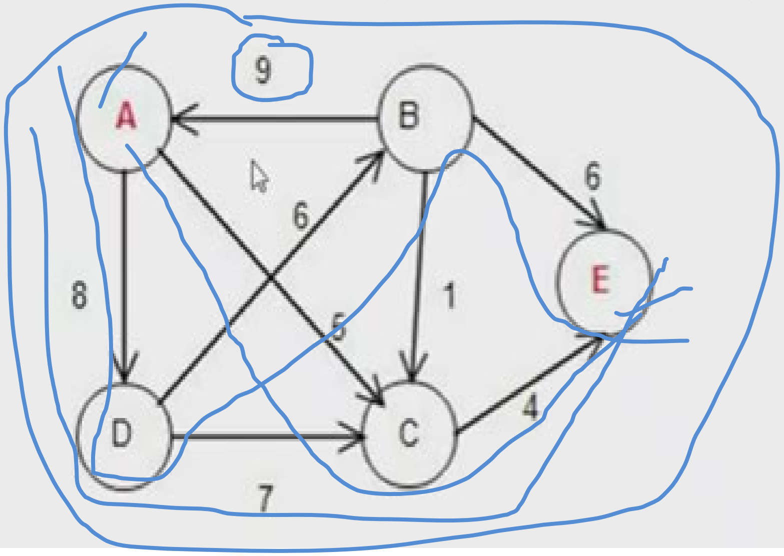
- The greedy method simply requires us to store the programs in non-decreasing order of their lengths.
- This ordering (sorting) can be carried out in O(n log n) time using an efficient sorting algorithm.

6. Single-source shortest paths:

Given a weighted, directed graph *G* = *(V, E)*, such that every edge in *E* has a positive weight.

To find the shortest path starting from source vertex to the other vertices in graph.

A path is a sum of subsequent connected edges in a given direction.



Source – A
Destination – E

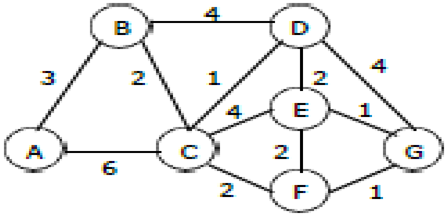
Available paths –
A – D – C – E = 19
A – C – E = 9
A – D – B – E = 20

Optimal route = **A – C – E**

- **Dijkstra's algorithm - is a solution to the single-source** shortest path problem in graph theory.
- Works on both directed and undirected graphs. However, all edges must have nonnegative weights.
- **Approach**: Greedy
- **Input**: Weighted graph G={E,V} and source vertex *v∈V,* such that all edge weights are nonnegative
- **Output:** Lengths of shortest paths (or the shortest paths themselves) from a given source vertex *v∈V to* all other vertices

Example 1:

Use Dijkstras algorithm to find the shortest path from A to each of the other six vertices in the graph:



Solution:

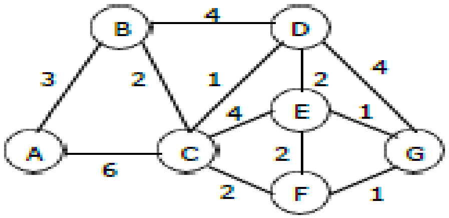
The cost adjacency matrix is
$$\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$$

The problem is solved by considering the following information:

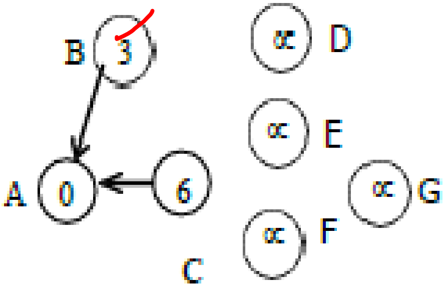
- Status[v] will be either '0', meaning that the shortest path from v to v_0 has definitely been found; or '1', meaning that it hasn't.
- Dist[v] will be a number, representing the length of the shortest path from v to v_0 found so far.
- Next[v] will be the first vertex on the way to v_0 along the shortest path found so far from v to v_0

The progress of Dijkstra's algorithm on the graph shown above is as follows:

$$\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$$

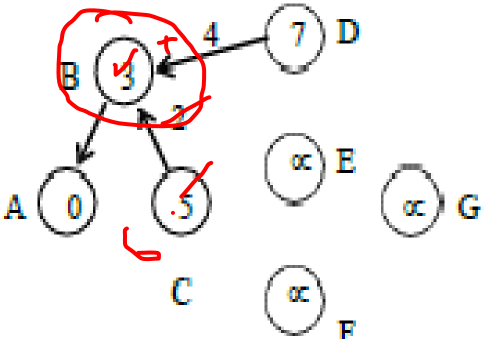


Step 1:



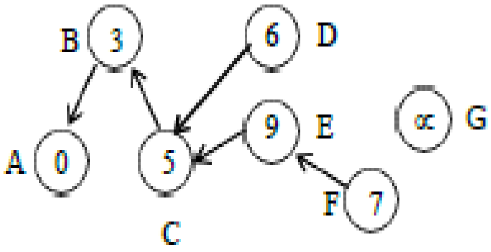
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step 2:



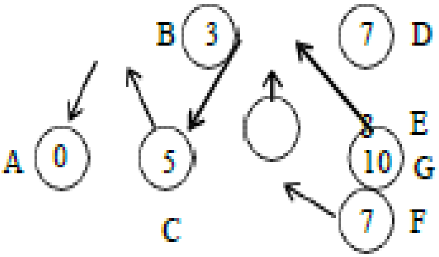
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7	∝	∝	∝
Next	*	A	B	B	A	A	A

Step 3:



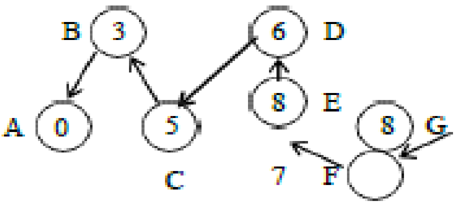
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	∝
Next	*	A	B	C	C	C	A

Step 4:



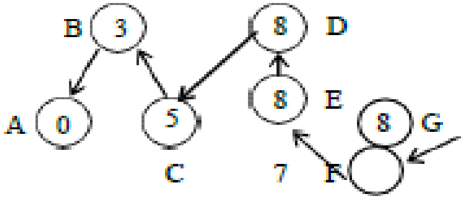
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step 5:



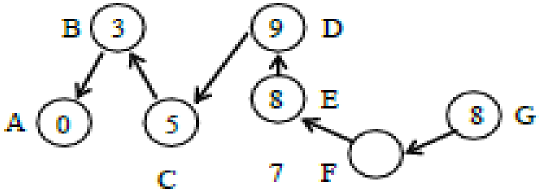
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Design and Analysis of Algorithms

UNIT-2

Part – II : Dynamic Programming

By

Y. Indira Priyadarshini
Assistant Professor
Department of Computer Science and Engineering

RCEW, Pasupula (V), Nandikotkur Road,
Near Venkayapalli, KURNOOL

Contents:

1. General Method.
2. Multistage graphs.
3. All-pairs shortest paths.
4. Optimal binary search trees.
5. 0/1 knapsack.
6. The travelling salesperson problem.

1.General method:

- Dynamic programming is a technique for solving problems with overlapping sub-problems.

- Typically, these sub-problems arise from a recurrence relating a solution to a given problem with solutions to its smaller sub-problems of the same type.

- Rather than solving overlapping sub-problems again and again, dynamic programming suggests solving each of the smaller sub-problems only once and recording the results in a table from which we can then obtain a solution to the original problem.

•Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub instances.

Main idea:

•set up a recurrence relating a solution to a larger instance to solutions of some smaller instances.

•solve smaller instances once.

•record solutions in a table.

•extract solution to the initial instance from that table.

Example: Fibonacci Numbers

- Recall definition of Fibonacci numbers:

$F(n) = F(n\text{-}1) + F(n\text{-}2)$

$F(0) = 0$

$F(1) = 1$

- Computing the n^{th} Fibonacci number recursively (top-down):

$F(n)$

$F(n\text{-}1) \quad + \quad F(n\text{-}2)$

$F(n\text{-}2) \quad + \quad F(n\text{-}3) \qquad F(n\text{-}3) \quad + \quad F(n\text{-}4)$

...

- Dynamic programming usually takes one of two approaches:

Bottom-up approach: All sub problems that might be needed are solved in advance and then used to build up solutions to larger problems. This approach is slightly better in stack space and number of function calls, but it is sometimes not intuitive to figure out all the sub problems needed for solving the given problem.

Top-down approach: The problem is broken into sub problems, and these sub problems are solved and the solutions remembered, in case they need to be solved again. This is recursion and Memory Function combined together.

Bottom Up

- In the bottom-up approach we calculate the smaller values of Fibo first, then build larger values from them. This method also uses linear (O(n)) time since it contains a loop that repeats n − 1 times.

```
Algorithm Fibo(n)
        a = 0, b = 1
        repeat n − 1 times
                c = a + b
                 a = b
                 b  = c
        return b
```

- In both these examples, we only calculate fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of putting t every time either of them is evaluated.

Top-Down

- suppose we have a simple map object, m, which maps each value of Fibo that has already been calculated to its result, and we modify our function to use it and update it. The resulting function requires only O(n) time instead of exponential time:

```
m [0] = 0
m [1] = 1

Algorithm Fibo(n)
    if map m does not contain key n
        m[n] := Fibo(n − 1) + Fibo(n − 2)
    return m[n]
```

- This technique of saving values that have already been calculated is called Memory Function; this is the top-down approach, since we first break the problem into subproblems and then calculate and store values.

2. Multistage graphs:

- A multistage graph G = (V,E) is a directed graph in which the vertices are portioned into K > = 2 disjoint sets Vi, 1 <= i<= k.
- In addition, if < u,v > is an edge in E, then u < = Vi and V □ Vi+1 for some i, 1<= i < k.
- If there will be only one vertex, then the sets Vi and Vk are such that [Vi]=[Vk] = 1.
- The cost of a path from source (s) to destination (t) is the sum of the costs of the edger on the path.

- The MULTISTAGE GRAPH problem is to find a minimum cost path from ‘s’ to ‘t’.
- Each set Vi defines a stage in the graph. Every path from ‘s’ to ‘t’ starts in stage-1, goes to stage-2 then to stage-3, then to stage-4, and so on, and terminates in stage-k.
- This MULISTAGE GRAPH problem can be solved in 2 ways.
- a) Forward Method.
- b) Backward Method

Forward Method:

1. Assume that there are ‘k’ stages in a graph.
2. In this FORWARD approach, we will find out the cost of each and every node starling from the ‘k’ th stage to the 1st stage.
3. We will find out the path (i.e.) minimum cost path from source to the destination (ie) [Stage-1 to Stage-k].

Backward Method

1. If there one ‘K’ stages in a graph using back ward approach. we will find out the cost of each & every vertex starting from 1st stage to the kth stage.
2. We will find out the minimum cost path from destination to source (ie)[from stage k to stage 1]

FORWARD APPROACH:

We use the following equation to find the minimum cost path from s to t:

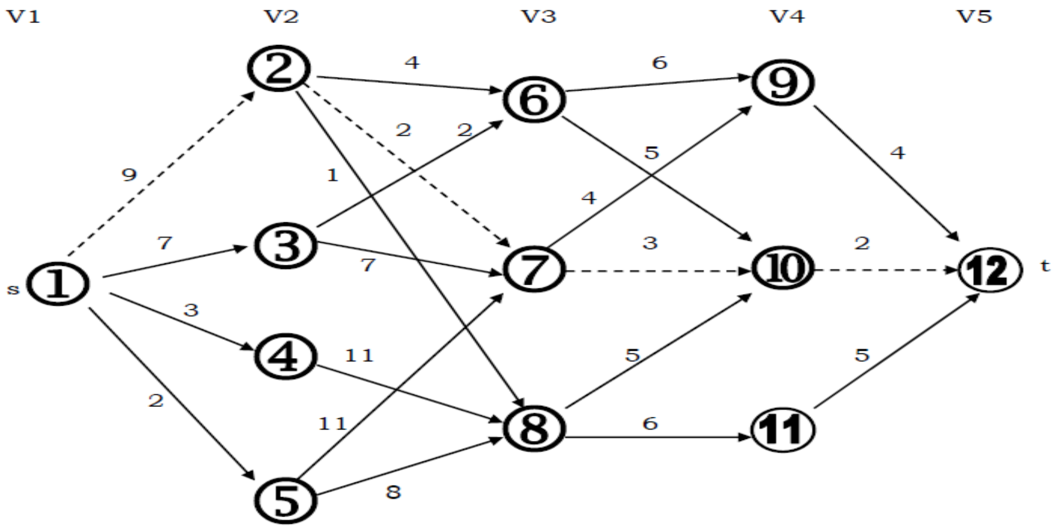
$$\text{cost}\ (i, j) = \min_{l\ \varepsilon\ V_{i+1}} \{c\ (j, l) + \text{cost}\ (i + 1, l)\}$$

BACKWARD APPROACH:

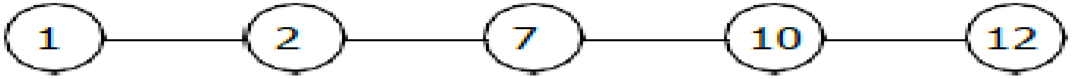
We use the following equation to find the minimum cost path from t to s:

$$\text{Bcost}\ (i, J) = \min_{\substack{l\ \varepsilon\ V_{i-1} \\ <l,\ j>\ \varepsilon\ E}} \{\text{Bcost}\ (i - 1, l) + c\ (l, J)\}$$

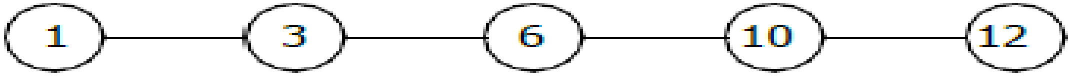
Forward Approach:



The path is



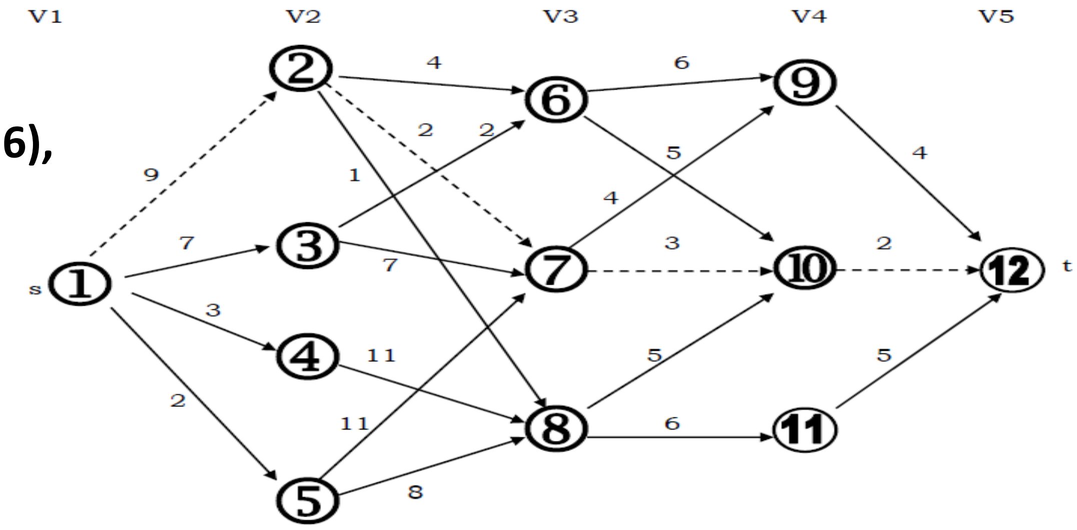
or



Backward Approach:

C(3,6)=
MIN{C(2,2)+C(2,6),
C(2,3)+C(3,6)}

MIN{9+4,7+2}
=MIN{13,9}
=9



V	1	2	3	4	5	6	7	8	9	10	11	12
C	0	9	7	3	2	9						
D	1	1	1	1	1	3						

3. All-pairs shortest paths :

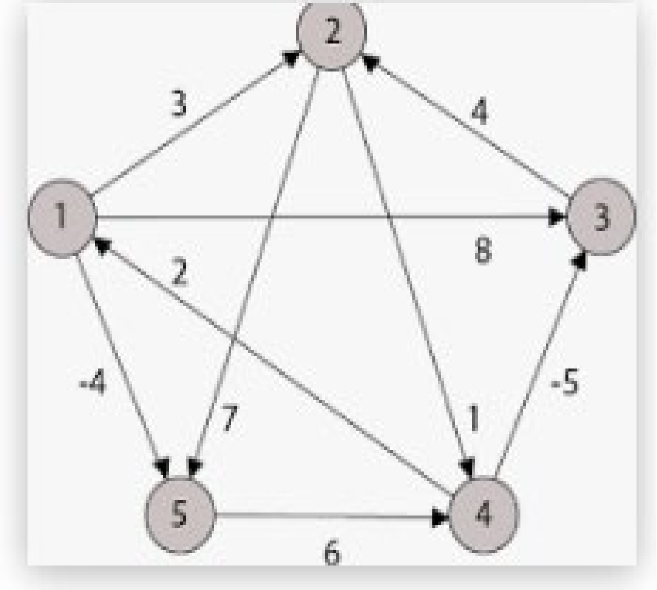
- Let G=<N,A> be a directed graph ’N’ is a set of nodes and ‘A’ is the set of edges.

- Each edge has an associated non-negative length.

- We want to calculate the length of the shortest path between each pair of nodes.

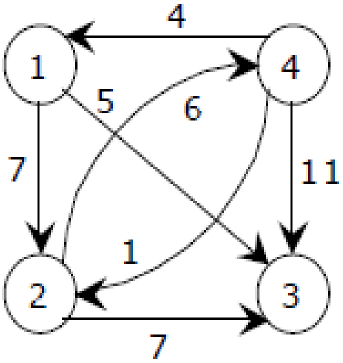
- Suppose the nodes of G are numbered from 1 to n, so N={1,2,...N},and suppose G matrix L gives the length of each edge, with L(i,j)=0 for i=1,2...n,L(i,j)>=for all i & j, and L(i,j)=infinity, if the edge (i,j) does not exist.

- The principle of optimality applies: if k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.


2
3
4
1
3
8
2
-4
7
1
-5
5
4
6

Example 1:

Use warshall's algorithm to calculate the reachability matrix for the graph:



We begin with the adjacency matrix of the graph 'A_0'

$$A_0 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

The update rule for computing A_i from A_{i-1} in warshall's algorithm is:

$$A_i\,[x, y] = A_{i-1}\,[x, y] \vee (A_{i-1}\,[x, i] \wedge A_{i-1}\,[i, y]) \qquad ----$$

The first step is to compute 'A_1' matrix. To do so we will use the updating rule – (1).

Before doing so, we notice that only one entry in A_0 must remain one in A_1, since in Boolean algebra 1 + (anything) = 1. Since these are only nine zero entries in A_0, there are only nine entries in A_0 that need to be updated.

$A_1[1, 1] = A_0[1, 1] \vee (A_0[1, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$

$A_1[1, 4] = A_0[1, 4] \vee (A_0[1, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0$

$A_1[2, 1] = A_0[2, 1] \vee (A_0[2, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$

$A_1[2, 2] = A_0[2, 2] \vee (A_0[2, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0$

$A_1[3, 1] = A_0[3, 1] \vee (A_0[3, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$

$A_1[3, 2] = A_0[3, 2] \vee (A_0[3, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0$

$A_1[3, 3] = A_0[3, 3] \vee (A_0[3, 1] \wedge A_0[1, 3]) = 0 \vee (0 \wedge 1) = 0$

$A_1[3, 4] = A_0[3, 4] \vee (A_0[3, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0$

$A_1[4, 4] = A_0[4, 4] \vee (A_0[4, 1] \wedge A_0[1, 4]) = 0 \vee (1 \wedge 0) = 0$

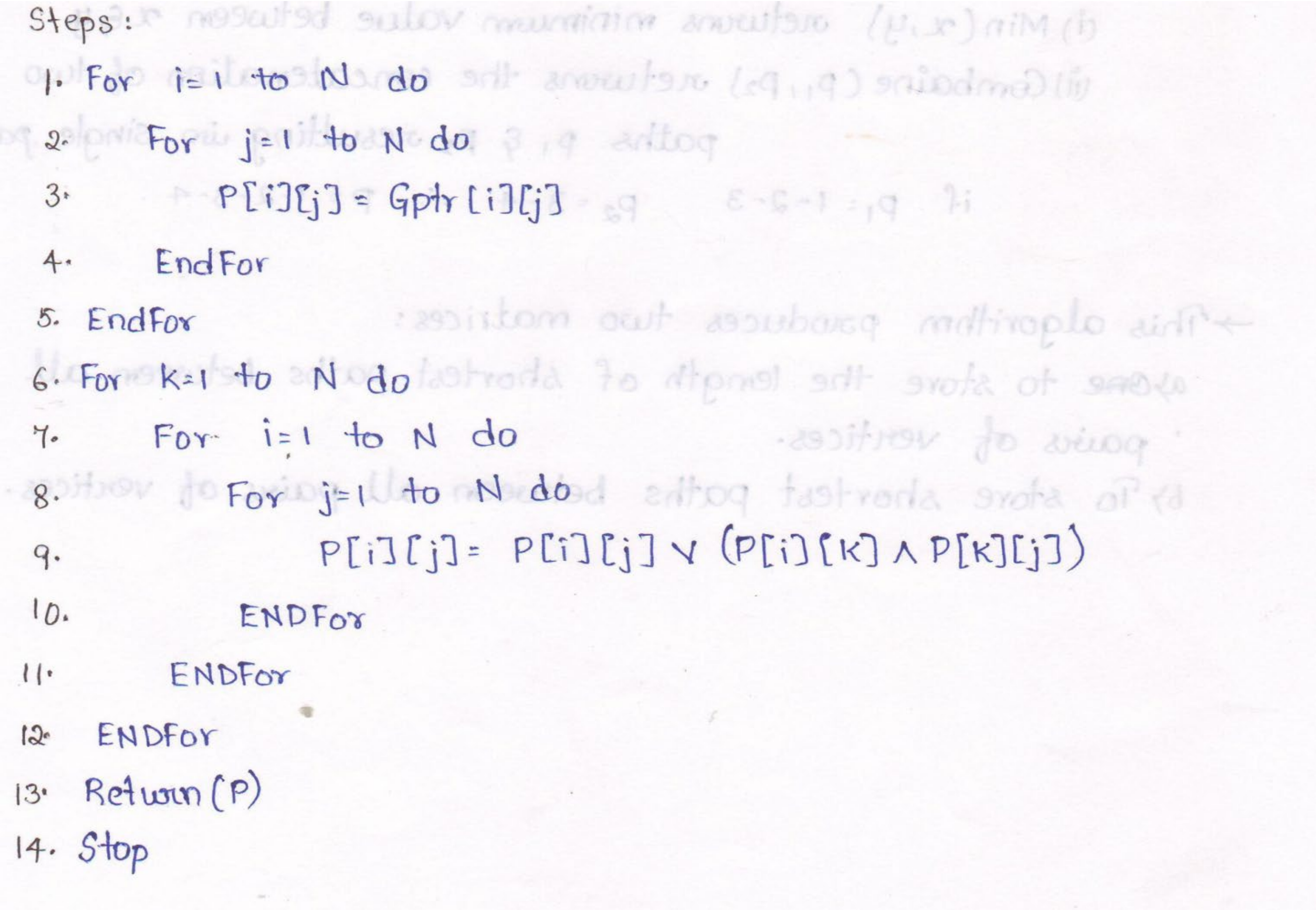
$$A_1 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

$$A_2 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \qquad A_3 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$A_4 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Note that according to the algorithm vertex 3 is not reachable from itself 1. This is because as can be seen in the graph, there is no path from vertex 3 back to itself.

Algorithm:



Steps:

```
For i=1 to N do
    For j=1 to N do
        P[i][j] = Gptr[i][j]
    EndFor
EndFor
For k=1 to N do
    For i=1 to N do
        For j=1 to N do
            P[i][j] = P[i][j] ∨ (P[i][k] ∧ P[k][j])
        ENDFor
    ENDFor
ENDFor
Return (P)
Stop
```

Floyd’s Algorithm:

- Weights of edges are taken into account for getting the length of shortest path between any pair of vertices.
- Here we use two functions,

1.min(x, y) returns minimum between two values x, y.

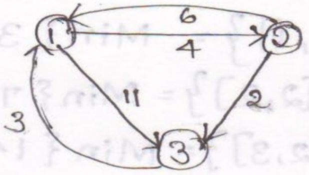
2.combine(p1,p2), returns the concatenation of two paths p1 and p2 resulting in single path.

- **Ex:** if p1=1-2-3, p2=3-4 then p=1-2-3-4
- The algorithm produces two matrices,

1.To store length of shortest path between all pairs of vertices.

2. To store shortest path between all pairs of vertices.

Example:



$$A_0 = \begin{array}{cccc} & 1 & 2 & 3 \\ 1 & \infty & 4 & 11 \\ 2 & 6 & \infty & 2 \\ 3 & 3 & \infty & \infty \end{array}$$

Path

–	1	2	3
1	–	[1-2]	[1-3]
2	[2-1]	–	[2-3]
3	[3-1]	–	–

Computing A_1 matrix: $K=1$

$A_1[1,1] = \text{Min}\{A_0[1,1], A_0[1,1]+A_0[1,1]\} = \text{Min}\{\infty, (\infty+\infty)\} = \infty$

$A_1[1,2] = \text{Min}\{A_0[1,2], A_0[1,1]+A_0[1,2]\} = \text{Min}\{4, \overset{(\infty+4)}{\infty}\} = 4$

$A_1[1,3] = \text{Min}\{A_0[1,3], A_0[1,1]+A_0[1,3]\} = \text{Min}\{11, (\infty+11)\} = 11$

$A_1[2,1] = \text{min}\{A_0[2,1], A_0[2,1]+A_0[1,1]\} = \text{Min}\{6, (6+\infty)\} = 6$

$A_1[2,2] = \text{min}\{A_0[2,2], A_0[2,1]+A_0[1,2]\} = \text{Min}\{\infty, 6+4\} = 10$

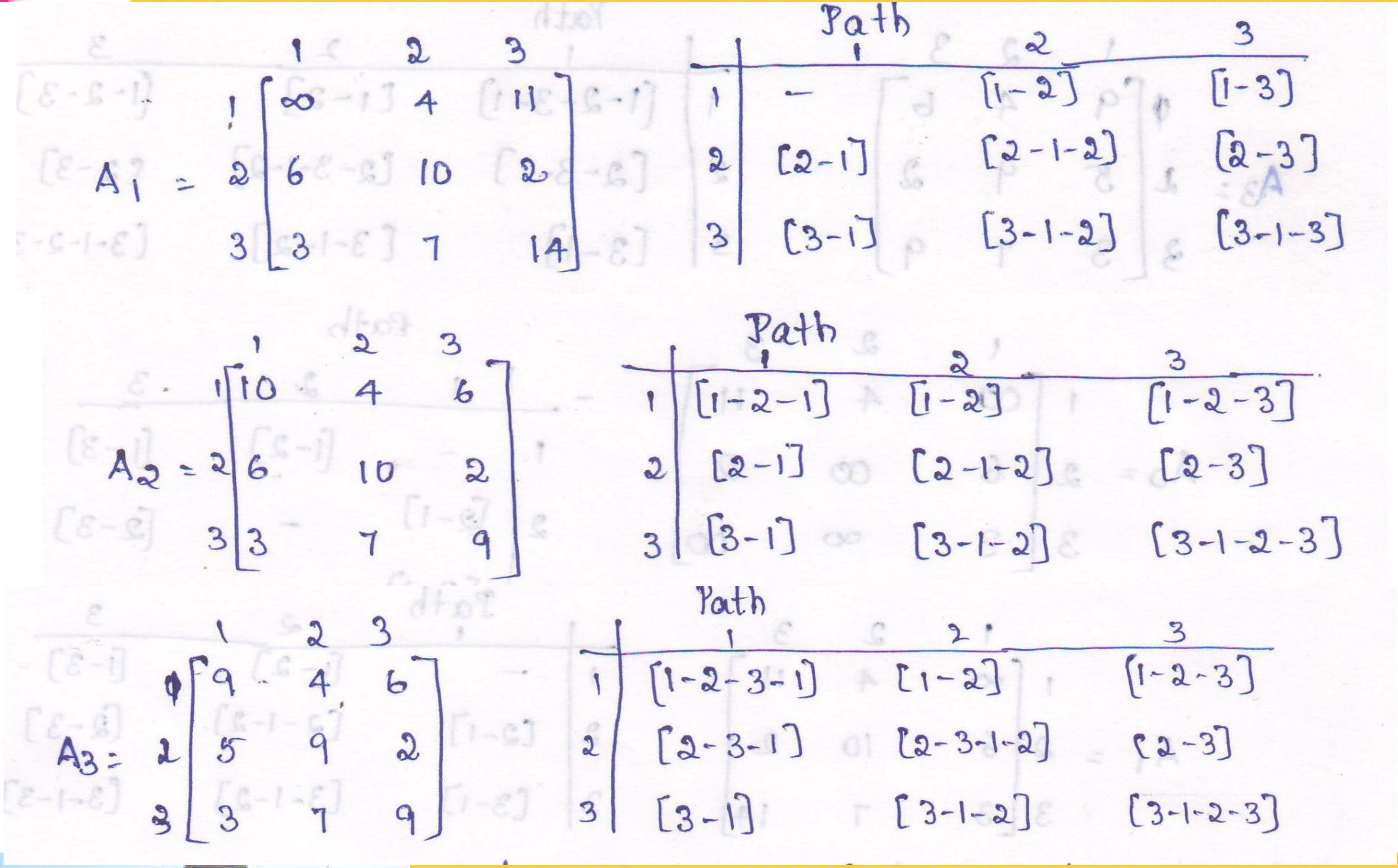
$A_1[2,3] = \text{min}\{A_0[2,3], A_0[2,1]+A_0[1,3]\} = \text{Min}\{2, (6+11)\} = 2$

$A_1[3,1] = \text{min}\{A_0[3,1], A_0[3,1]+A_0[1,1]\} = \text{Min}\{3, (3+\infty)\} = 3$

$A_1[3,2] = \text{Min}\{A_0[3,2], A_0[3,1]+A_0[1,2]\} = \text{Min}\{\infty, (3+4)\} = 7$

$A_1[3,3] = \text{Min}\{A_0[3,3], A_0[3,1]+A_0[1,3]\} = \text{Min}\{\infty, (3+11)\} = 14$

Path



$$A_1 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & \infty & 4 & 11 \\ 2 & 6 & 10 & 2 \\ 3 & 3 & 7 & 14 \end{array}$$

Path

	1	2	3
1	—	[1-2]	[1-3]
2	[2-1]	[2-1-2]	[2-3]
3	[3-1]	[3-1-2]	[3-1-3]

$$A_2 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 10 & 4 & 6 \\ 2 & 6 & 10 & 2 \\ 3 & 3 & 7 & 9 \end{array}$$

Path

	1	2	3
1	[1-2-1]	[1-2]	[1-2-3]
2	[2-1]	[2-1-2]	[2-3]
3	[3-1]	[3-1-2]	[3-1-2-3]

$$A_3 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 9 & 4 & 6 \\ 2 & 5 & 9 & 2 \\ 3 & 3 & 7 & 9 \end{array}$$

Path

	1	2	3
1	[1-2-3-1]	[1-2]	[1-2-3]
2	[2-3-1]	[2-3-1-2]	[2-3]
3	[3-1]	[3-1-2]	[3-1-2-3]

4.Optimal binary search trees:

- OBST is one special kind of advanced tree.
- It focus on how to reduce the cost of the search of the BST.
- It may not have the lowest height !
- It needs 3 tables to record probabilities, cost, and root.
- The case of search are two situations, one is success, and the other, without saying, is failure.

- The expected cost contribution for the internal node for 'ai' is:

*P (i) * level (ai) .*

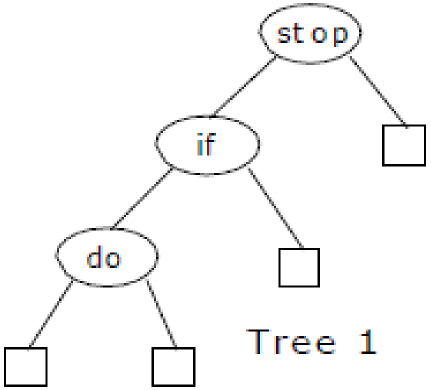
- Unsuccessful search terminate with I = 0 (i.e at an external node). Hence the cost contribution for this node is:

Q (i) * level ((Ei) - 1)

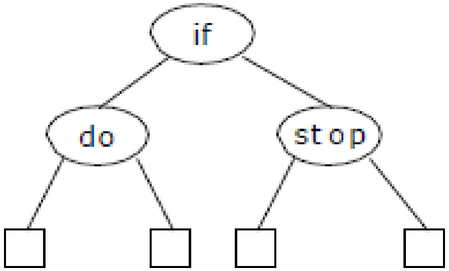
- The expected cost of binary search tree is:

$$\sum_{i=1}^{n} P(i) * level\ (a_i) + \sum_{i=0}^{n} Q\ (i) * level\ ((E_i) - 1)$$

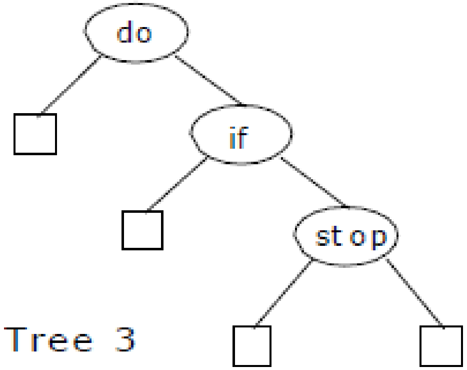
Example 1: The possible binary search trees for the identifier set (a_1, a_2, a_3) = (do, if, stop) are as follows. Given the equal probabilities p (i) = Q (i) = 1/7 for all i, we have:



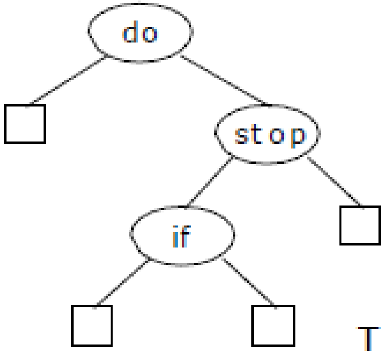
Tree 1



Tree 2



Tree 3



Tree 4

$$\text{Cost (tree \# 1)} = \left(\frac{1}{7} x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3\right) + \left(\frac{1}{7} x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3 + \frac{1}{7} x\ 3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

$$\text{Cost (tree \# 2)} = \left(\frac{1}{7} x1 + \frac{1}{7}\ x2 + \frac{1}{7} x2\right) + \left(\frac{1}{7} x2 + \frac{1}{7} x\ 2 + \frac{1}{7}\ x\ 2 + \frac{1}{7} x2\right)$$

$$= \frac{1+2+2}{7} + \frac{2+2+2+2}{7} = \frac{5+8}{7} = \frac{13}{7}$$

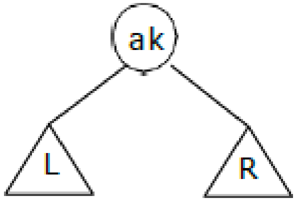
$$\text{Cost (tree \# 3)} = \left(\frac{1}{7}\ x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3\right) + \left(\frac{1}{7} x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3 + \frac{1}{7} x\ 3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

$$\text{Cost (tree \# 4)} = \left(\frac{1}{7}\ x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3\right) + \left(\frac{1}{7} x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3 + \frac{1}{7} x\ 3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

- The structure of an optimal binary search tree is:



$$\text{Cost (L)} = \sum_{i=1}^{K} P(i)^* \; level \;\; (a_i) + \sum_{i=0}^{K} Q(i)^* \left(level\, (E_i) - 1 \right)$$

$$\text{Cost (R)} = \sum_{i=K}^{n} P(i)^* \; level \;\; (a_i) + \sum_{i=K}^{n} Q(i)^* \left(level\, (E_i) - 1 \right)$$

The C (i, J) can be computed as:

$$C\ (i, J) = \min_{i<k\leq J} \{C\ (i, k\text{-}1) + C\ (k, J) + P\ (K) + w\ (i, K\text{-}1) + w\ (K, J)\}$$

$$= \min_{i<k\leq J} \{C\ (i, K\text{-}1) + C\ (K, J)\} + w\ \ (i, J) \qquad \text{--} \qquad (1)$$

$$\text{Where } W\ (i, J) = P\ (J) + Q\ (J) + w\ \ (i, J\text{-}1) \qquad \text{--} \qquad (2)$$

Initially C (i, i) = 0 and w (i, i) = Q (i) for $0 \leq i \leq n$.

Equation (1) may be solved for C (0, n) by first computing all C (i, J) such that J - i = 1
Next, we can compute all C (i, J) such that J - i = 2, Then all C (i, J) with J - i = 3 and so on.

C (i, J) is the cost of the optimal binary search tree 'T_{ij}' during computation we record the root R (i, J) of each tree 'T_{ij}'. Then an optimal binary search tree may be constructed from these R (i, J). R (i, J) is the value of 'K' that minimizes equation (1).

We solve the problem by knowing W (i, i+1), C (i, i+1) and R (i, i+1), $0 \leq i < 4$; Knowing W (i, i+2), C (i, i+2) and R (i, i+2), $0 \leq i < 3$ and repeating until W (0, n), C (0, n) and R (0, n) are obtained.

The results are tabulated to recover the actual tree.

Example 1:

Let n = 4, and (a1, a2, a3, a4) = (do, if, need, while) Let P (1: 4) = (3, 3, 1, 1) and Q (0: 4) = (2, 3, 1, 1, 1)   1 2 3 4

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{1,2} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	W02=12 C02=19 R02=1	W13=9 C13=12 R13=2	W24=5 C24=8 R24=3		
3	W03=14 C03=25 R03=2	W14=11 C14=19 R14=2			
4	W04=16 C04=32 R04=2				

$j - i = 0$

$0 - 0 = 0 \rightarrow c(0,0) = 0$

$1 - 1 = 0 \rightarrow c(1,1) = 0$

$2 - 2 = 0 \rightarrow c(2,2) = 0$

$3 - 3 = 0 \rightarrow c(3,3) = 0$

$4 - 4 = 0 \rightarrow c(4,4) = 0$

$w(0,0) = Q(0) =$

$w(1,1) = Q(1) =$

$w(2,2) = Q(2) =$

$w(3,3) = Q(3) =$

$w(4,4) = Q(4) =$

$r_{00} = k = 0$

$r_{11} = k = 0$

$r_{22} = k = 0$

$r_{33} = k = 0$

$r_{44} = k = 0$

$$c(i,j) = \min_{\substack{i \le k \le j \\ 0 < 1 \le 1 \\ k=1}} \left(\underline{c[i,k-1) + c(k,j)} + w[i,j) \right)$$

$\hookrightarrow = w[i,j-1) + P(j) + Q(j)$

$j - i = 1$

$1 - 0 = 1 \rightarrow c(0,1) = \min[c(0,1-1(0) + c(1,1)) + [\underline{w(0,0) + P(1) + Q(1)}]$

$2 - 1 = 1 \rightarrow c(1,2)$

$3 - 2 = 1 \rightarrow c(2,3)$

$4 - 3 = 1 \rightarrow c(3,4)$

$= \min(0 + 0) + w[2 + 3 + 3)$

$= 0 + 8 = \underline{8}$

$c[3,4) = \min(c[3,3) + c(4,4)] + w(3,4]$

$3 < 1 \le 4$, $k = 4$ $= 0 + 0 + 3 = \underline{3}$

$c(1,2) = \min[c(1,1) + c(2,2)) + w(1,2]$

$k = 2 = 0 + 0 + 7 = 7$

$c(2,3) = \min(c(2,2) + c(3,3)) + w(2,3)$

$k = 3 = 0 + 0 + 3 = 3$

$w[1,2) = w[1,1] + P[2) + Q(2) = 7$

$w(2,3) = w(2,2) + P(3) + Q(3) = 3$

$w(3,4) = w[3,3) + P(4) + Q(4) = 3$

$w(0,1) = \underline{8}$

$j - i = 2$

$2 - 0 = 2$

$3 - 1 = 2$

$4 - 2 = 2$

$c(0,2) = 0 \le \overset{1,2}{k} \le 2$

$c(1,3) = 1 \le \overset{2,3}{k} \le 3$

$c(2,4) = 2 < \overset{3,4}{k} \le 4$

$w(0,2) = w(0,1) + P(2) + Q(2)$

$= 8 + 3 + 1 = 12$

$w(1,3) = w(1,2) + P(3) + Q(3)$

$= 7 + 1 + 1 = 9$

$w(2,4) = w(2,3) + P(4) + Q(4)$

$= 3 + 1 + 1 = 5$

$$c(0,2) = \min\{\underbrace{c(0,0) + c(1,2)}_{k=1},\ \underbrace{c(0,1) + c(2,2)}_{k=2}\} + w(0,2)$$

$$= \min(\underline{0 + 7},\ 8 + 0) + 12$$

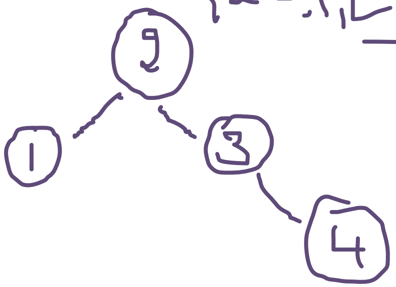
$$= 7 + 12 = 19$$

$r_{0,2} = k = 1$

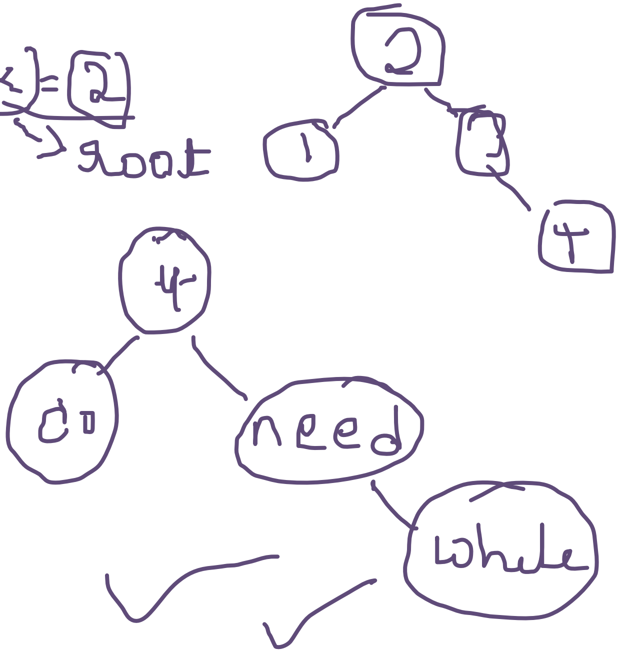
$c(0,4) = \underline{32} \rightarrow \boxed{r} = \boxed{2}$ → root

□ 1 4

1,2 - 1, 2, 3, 4



→



5.0/1 knapsack:

- This problem is similar to ordinary knapsack problem but we may not take a fraction of an object.
- We are given ' N ' object with weight W_i and profits P_i where I varies from 1 to N and also a knapsack with capacity ' M '.
- The problem is, we have to fill the bag with the help of ' N ' objects and the resulting profit has to be maximum.
- Formally, the problem can be started as, maximize $\sum_{i=1}^{n} X_i P_i$

  subject to $\sum_{i=1}^{n} X_i W_i \text{ L } M$
- Where X_i are constraints on the solution $X_i \in \{0,1\}$. (u) X_i is required to be 0 or 1. if the object is selected then the unit in 1. if the object is rejected than the unit is 0. That is why it is called as 0/1, knapsack problem.
- To solve the problem by dynamic programming we up a table T[1…N, 0…M] (ic) the size is N. where 'N' is the no. of objects and column starts with 'O' to capacity (ic) 'M'.
- In the table T[i,j] will be the maximum valve of the objects i varies from 1 to n and j varies from O to M.

Design and Analysis of Algorithms

Greedy

fractional

p_i / w_i = (1.36)

Dynamic

x ← 1

0

( )

↓

(maxim)

1   0

n - objects

m - capacity   m = 20

P –

W =

n = 3   1   2   3

P – 10   15   20

W – 18   12   15

p_i / w_i – (1.6)   (1.8)   (1.2)

fw

1

0

x

0 – 1

0 –

1

x_1 x_2 x_3

30 = 1   0.5   1

RCEW, Pasupula (V), Nandikotkur Road,
Near Venkayapalli, KURNOOL

n=4

m=8

w

			0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

8-6=2

2-2=0

8-6

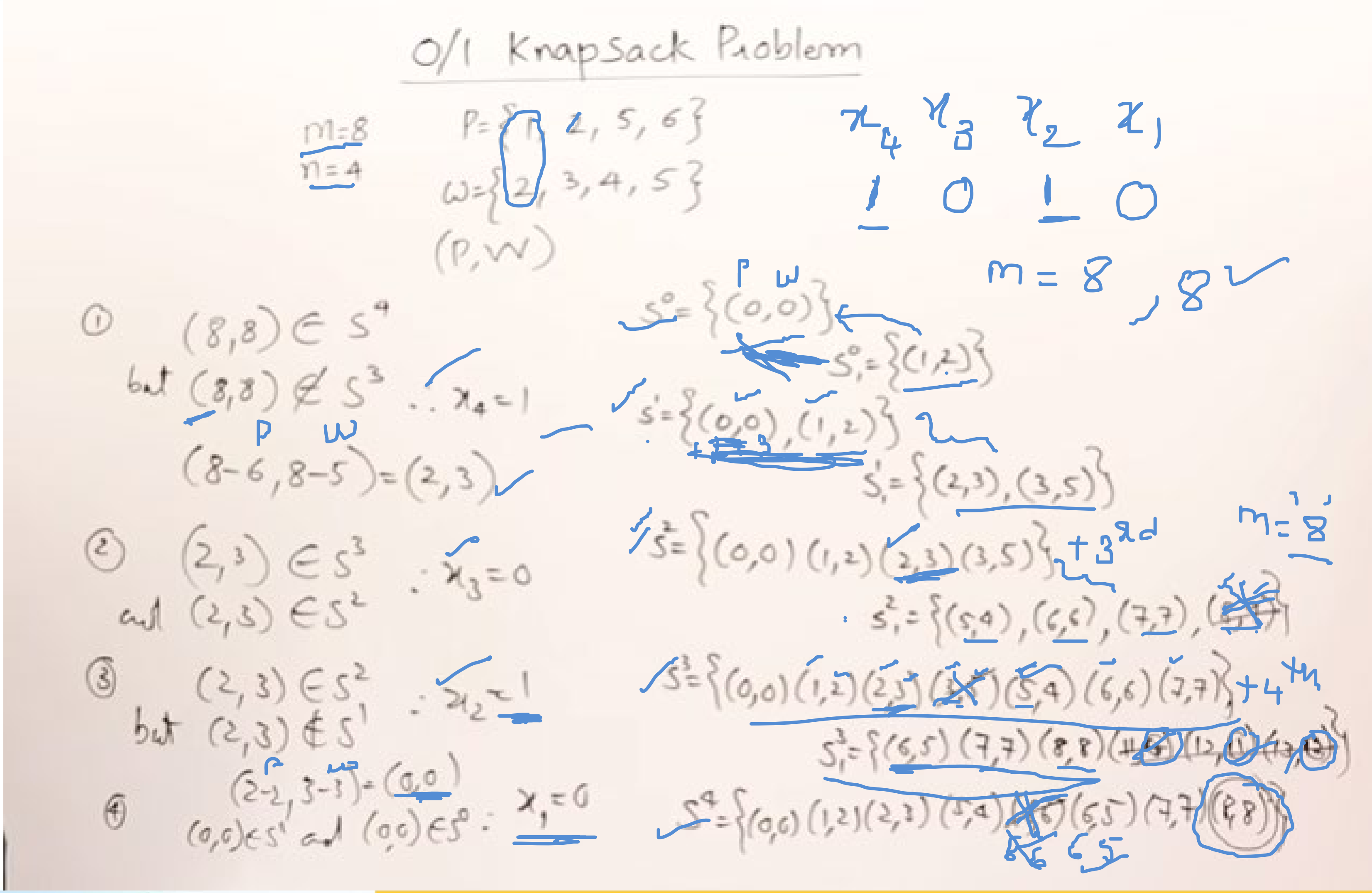
x_1 x_2 x_3 x_4

3 5

8 8

X1	X2	X3	x4
0	1	0	1

Design and Analysis of Algorithms



0/1 KnapSack Problem

$m=8$
$n=4$

$P=\{1, 2, 5, 6\}$
$w=\{2, 3, 4, 5\}$
(P, W)

$x_4 \quad x_3 \quad x_2 \quad x_1$
$1 \quad 0 \quad 1 \quad 0$

$m = 8, \; 8$ ✓

① $(8,8) \in S^4$
but $(8,8) \notin S^3 \quad \therefore x_4 = 1$

$(\overset{P}{8-6}, \overset{W}{8-5}) = (2,3)$

② $(2,3) \in S^3 \quad \therefore x_3 = 0$
and $(2,3) \in S^2$

③ $(2,3) \in S^2 \quad \therefore x_2 = 1$
but $(2,3) \notin S^1$
$(2-2, 3-3) = (0,0)$

④ $(0,0) \in S^1$ and $(0,0) \in S^0 \therefore x_1 = 0$

$S^0 = \{(0,0)\}$

$S^0_1 = \{(1,2)\}$

$S^1 = \{(0,0), (1,2)\}$ $+3$

$S^1_1 = \{(2,3), (3,5)\}$

$S^2 = \{(0,0)(1,2)(2,3)(3,5)\}$ $+3^{rd}$

$m=8$

$S^2_1 = \{(5,4), (6,6), (7,7), \cancel{(8,9)}\}$

$S^3 = \{(0,0)(1,2)(2,3)\cancel{(3,5)}(5,4)(6,6)(7,7)\}$ $+4^{th}$

$S^3_1 = \{(6,5)(7,7)(8,8)\cancel{(11,9)}(12,11)(13,12)\}$

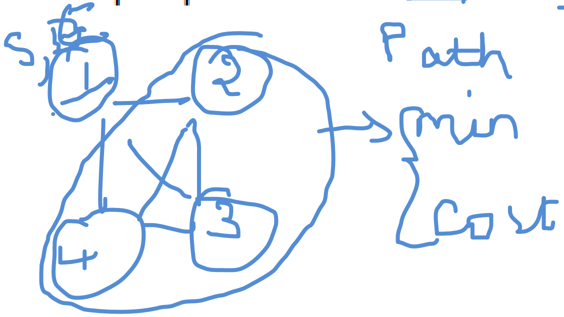
$S^4 = \{(0,0)(1,2)(2,3)(5,4)\cancel{(6,6)}(6,5)(7,7)(8,8)\}$

$6 \; 6 \quad 6 \; 5$

RCEW, Pasupula (V), Nandikotkur Road,
Near Venkayapalli, KURNOOL

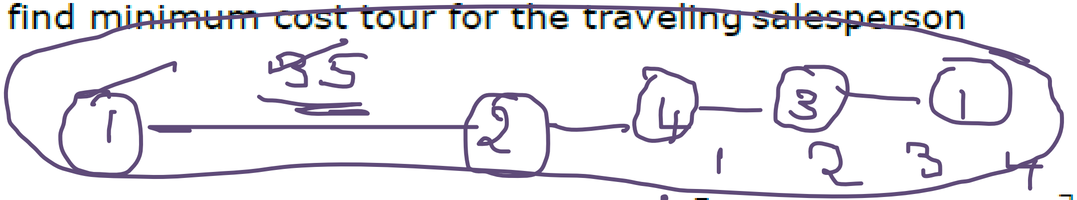
6.The travelling salesperson problem:

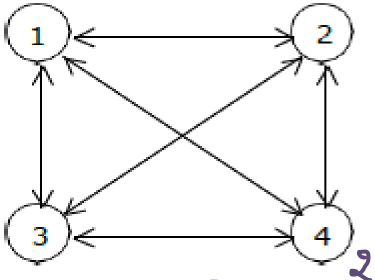
Let G = (V, E) be a directed graph with edge costs C_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \alpha$ if $< i, j> \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex 1.



Example 1:

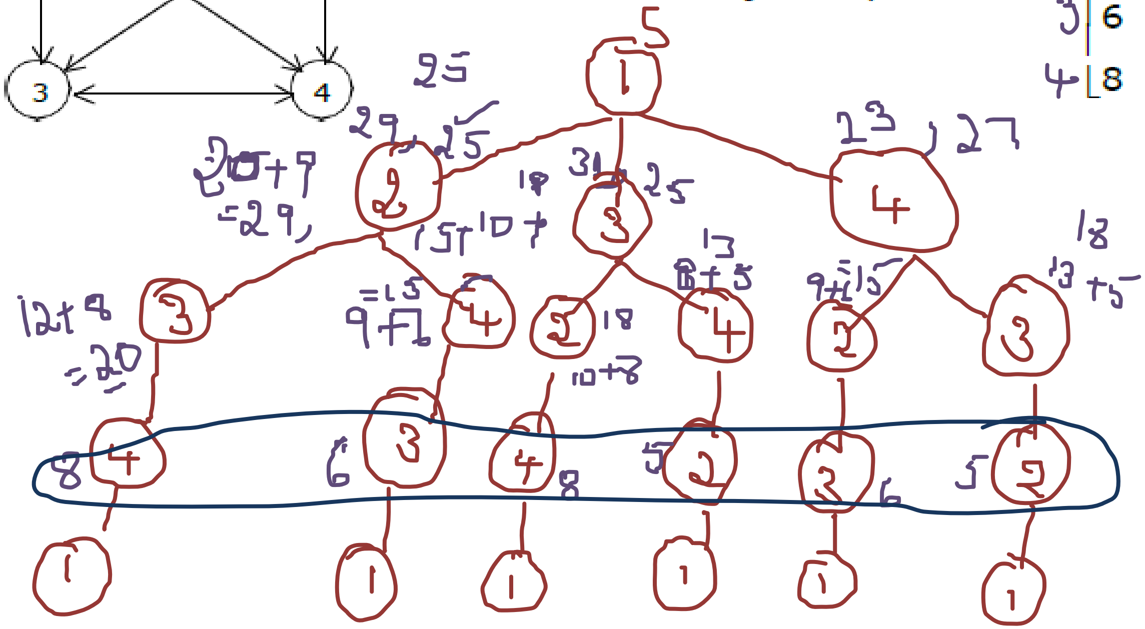
For the following graph find minimum cost tour for the traveling salesperson problem:

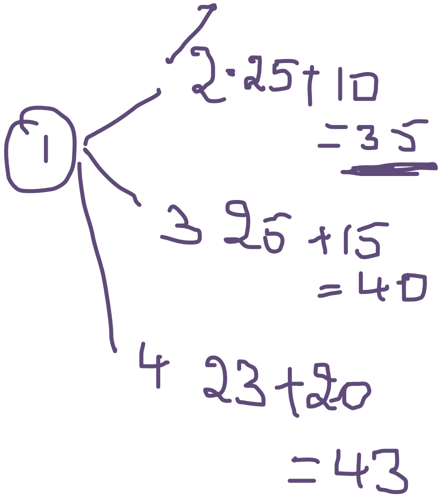




The cost adjacency matrix =

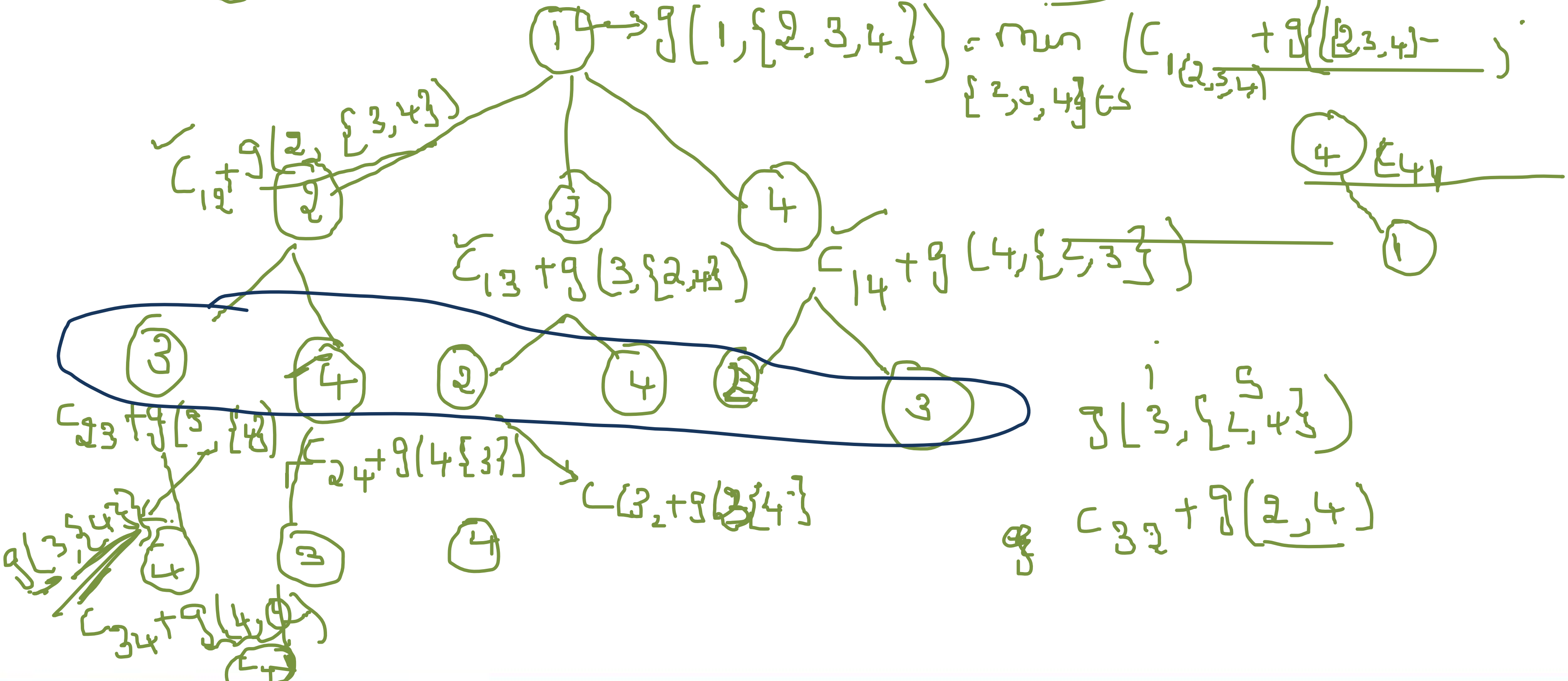
$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 10 & 15 & 20 \\ 2 & 5 & 0 & 9 & 10 \\ 3 & 6 & 13 & 0 & 12 \\ 4 & 8 & 8 & 9 & 0 \end{array}$$





$$g(i, S) = \min_{k \in S} \{ C_{ik} + g(k, \underbrace{S - \{k\}}_{\{3,4\}}) \}$$

$\{2,3,4\} - \{2\}$



Design and Analysis of Algorithms

$g(2, \overset{\boxed{1}}{\phi}) = 5$

$g(3, \overset{\boxed{1}}{\phi}) = 6$

$g(4, \overset{\boxed{1}}{\phi}) = 8$

$g(2, \{3\}) = c_{23} + g(3, \phi) = 9 + 6 = 15$

$g(2, \{4\}) = c_{24} + g(4, \phi) = 10 + 8 = 18$

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

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

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

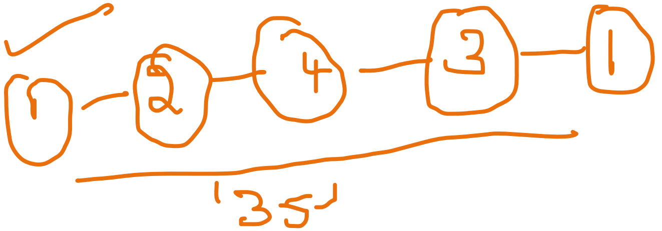
$g(4, \{3\}) = 15$

① $\min\left(\underset{9\ +\ 20}{c_{23}+g(3,\{4\})},\ \underset{10\ +\ 15}{c_{24}+g(4,\{3\})}\right)$

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

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

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



$$g(1,\{2,3,4\}) = \underset{\{2,3,4\}\in S}{\min}\left\{\underset{10+25}{c_{12}+g(2,\{3,4\})},\ \underset{15+25}{c_{13}+g(3,\{2,4\})},\ \underset{20+23}{c_{14}+g(4,\{2,3\})}\right.$$

$= 35$