

PDFZilla – Unregistered

PDFZilla - Unregistered

PDFZilla - Unregistered

Module - 4

Dynamic programming: It was designed for optimizing multistage decision problems. Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

- Dynamic programming is mainly an optimization over plain recursion.
- Dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choice, each subsequence must also be optimal.
- Dynamic programming is used where we have problems, which can be divided into similar subproblems, so that their results can be re-used.

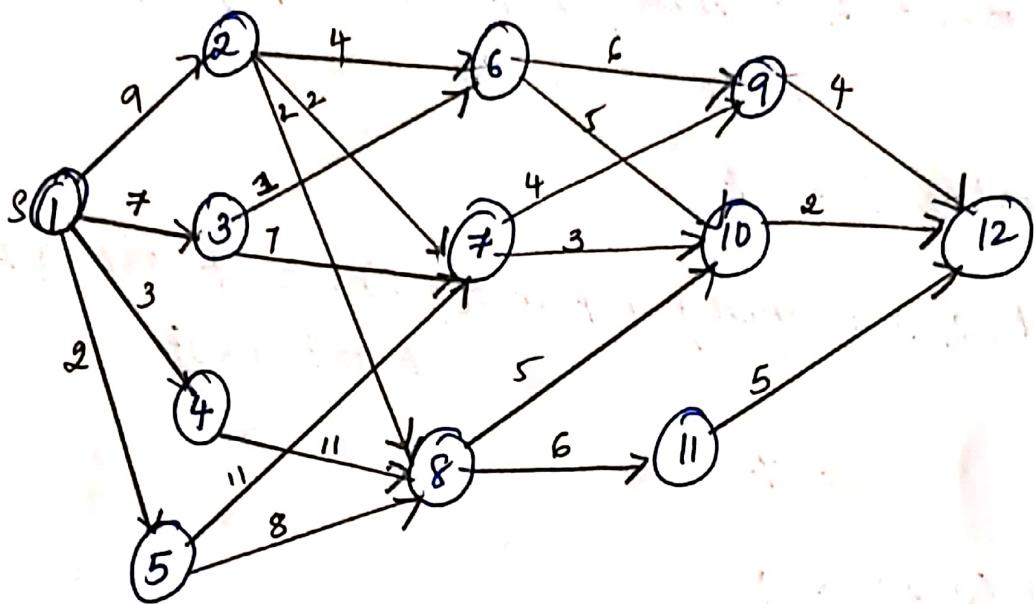
Ex: - optimal merge pattern

An optimal merge pattern tells which pair of files should be merged at each step. As a decision sequence, problem is to decide which pair of files should be merged first, which pair second and so on.

An optimal sequence of decisions is a least-cost sequence.

Multistage Graph

- It is a directed unweighted graph, $G = (V, E)$, vertices are divided into stages such that, the edges connects vertex from one stage to another. First stage and last stage has single vertex only.
- It is mainly applied in resource allocation.
- The vertex s is the source, and t is the sink.
- $c(i, j)$ - cost of edge (i, j) .
- The cost of the path from s to t is the sum of the costs of the edges on the path.
- Multistage graph problem is to find a minimum-cost path from s to t .
- Dynamic programming solution can be applied to find a optimal solution i.e principle of optimality



V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d.	2/3	7	6	8	8	10	10	10	12	12	12	12

consider last - 1 stage - IV stage

stage vertex

$$\text{cost}(4, 9) = 4$$

$$\text{cost}(4, 10) = 2$$

$$\text{cost}(4, 11) = 5$$

III stage :

$$\text{cost}(3, 6) = \min \{ \text{cost}(6, 9) + \text{cost}(4, 9), \\ \text{cost}(6, 10) + \text{cost}(4, 10) \}$$

$$= \min \{ 6 + 4, 5 + 2 \}$$

$$= 7$$

$$\text{cost}(3, 7) = \min \{ 4 + \text{cost}(4, 9), 3 + \text{cost}(4, 10) \}$$
$$= 5$$

$$\text{cost}(3, 8) = 7$$

$$\text{cost}(2, 2) = \min \{ 4 + \text{cost}(3, 6), 2 + \text{cost}(3, 7), 1 + \text{cost}(3, 8) \}$$

$$= 7$$

$$\text{cost}(9, 3) = 9$$

$$\text{cost}(2, 4) = 18$$

$$\text{cost}(2, 5) = 15$$

$$\text{cost}(1, 1) = \min \{ 9 + \text{cost}(2, 2), 7 + \text{cost}(2, 3), 3 + \text{cost}(2, 4), 2 + \text{cost}(2, 5) \}$$

$$= 16$$

stage vertex

$$\text{Cost}(i', j') = \min \{ c(j', l) + \text{cost}(i+1, l) \}$$

let v_{i+1}

$(j', l) \in E$

Algorithm FGraph (G, k, n, p)

// the input is a k -stage graph $G = (V, E)$ with n vertices.
// indexed in order of stages. E is the set of edges and $c[i, j]$ is the cost. $P[1:k]$ is the minimum-cost path.

{

$\text{cost}[n] = 0;$

for $j = n-1$ to 1 step -1 do

{

let r_j be a vertex such that (j, r_j) is an edge of G_j and $c[j, r_j] + \text{cost}[r_j]$ is minimum;

$\text{cost}[j] = \text{cost}[j, r_j] + \text{cost}[r_j];$

$d[j] = r_j;$

{

$p[1] = 1;$

$p[k] = n;$

for $j = 2$ to $k-1$ do

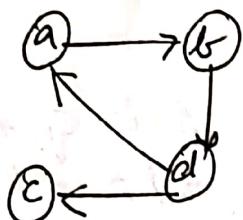
$p[j] = d[p[j-1]]$

{

Transitive closure:

- The transitive closure of a directed graph with 'n' vertices can be defined as n by n boolean matrix $T = \{t_{ij}\}$, in which the element in the i^{th} row ($1 \leq i \leq n$) & j^{th} column ($1 \leq j \leq n$) is 1 if there exists a non-trivial directed path

Example:



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Adjacency Matrix

Transitive closure

- Transitive closure can be generated by DFS or BFS.

Warshall's Algorithm:

Warshall's algorithm constructs transitive closure of a given digraph with n vertices through a series of n by n boolean matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^k, \dots, R^n$$

- where the element $r_{ij}^{(k)}$ in the i^{th} row and j^{th} column of matrix $R^{(k)}$

is equal to 1, iff there exists a directed path from i^{th} vertex to the j^{th} vertex with each intermediate vertex if any numbered not higher than k .

$R^{(0)}$ → Adjacency matrix itself

Recurrence Relation for Warshall's Algorithm:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } (r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)})$$

i.e. (1) If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in R^k .

- (2) If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in R^k iff the element in its row 'i' and column 'k' & the element in its column 'j' & row 'k' are both 1's in $R^{(k-1)}$.

$$R^{(k-1)} = R \begin{bmatrix} & i & k \\ i & \begin{array}{|c|c|} \hline 1 & & \\ \hline & 1 & \\ \hline \end{array} & \begin{array}{|c|c|} \hline & 1 \\ \hline \end{array} \\ \begin{array}{|c|} \hline 0 \rightarrow \\ \hline \end{array} & \begin{array}{|c|c|} \hline & 1 \\ \hline \end{array} & \begin{array}{|c|c|} \hline & 1 \\ \hline \end{array} \end{bmatrix} \Rightarrow R^k = R \begin{bmatrix} & i & k \\ i & \begin{array}{|c|c|} \hline 1 & & \\ \hline & 1 & \\ \hline \end{array} & \begin{array}{|c|c|} \hline & 1 \\ \hline \end{array} \\ \begin{array}{|c|} \hline 1 \\ \hline \end{array} & \begin{array}{|c|c|} \hline 1 & \\ \hline & 1 \\ \hline \end{array} & \begin{array}{|c|c|} \hline 1 & \\ \hline & 1 \\ \hline \end{array} \end{bmatrix}$$

Algorithm:

Algorithm Warshall ($A[1 \dots n][1 \dots n]$)

// Computes - transitive closure

// Input - Adjacency matrix A of a digraph

// with n vertices

// Output: Transitive closure

$$\{ R^{(0)} = A$$

```

for k=1 to n do
    for i=1 to n do
        for j=1 to n do

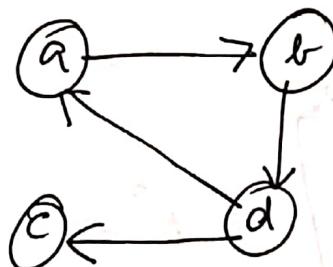
```

$$R^k[i, j] = R^{k-1}[i, j] \text{ or } (R^{k-1}[i, k] \text{ and } R^{k-1}[k, j])$$

return R^k

}

problem : obtain the transitive closure for the following digraph using warshall's algorithm:



Solution:

Adjacency matrix

$$A = R^{(0)} = \begin{bmatrix} a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{bmatrix}$$

$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

highlight 1st row & 1st column
- observe 1st in column & row
 $(d, a) = 1$ & $(a, b) = 1$, then
 $(d, b) = 1$

$$R^{(2)} = a \begin{bmatrix} a & b & c & d \\ 0 & 1 & 0 & 0 \\ b & 0 & 0 & 1 \\ c & 0 & 0 & 0 \\ d & 1 & 1 & 0 \end{bmatrix}$$

Highlight 2nd row & 2nd column:

$$(a, b) = 1, (b, d) = 1 \\ \therefore (a, d) = 1$$

$$(d, b) = 1, (b, d) = 1 \\ \therefore (d, d) = 1$$

$$R^{(2)} = a \begin{bmatrix} a & b & c & d \\ 0 & 1 & 0 & 1 \\ b & 0 & 0 & 1 \\ c & 0 & 0 & 0 \\ d & 1 & 1 & 1 \end{bmatrix}$$

$d, c = 1$, so no new paths.

$$R^{(3)} = a \begin{bmatrix} a & b & c & d \\ 0 & 1 & 0 & 1 \\ b & 0 & 0 & 1 \\ c & 0 & 0 & 0 \\ d & 1 & 1 & 1 \end{bmatrix}$$

$$(a, d) = 1 \quad \begin{cases} (d, a) = 1 & \therefore (a, a) = 1 \\ (d, b) = 1 & (a, b) = 1 \\ (d, c) = 1 & (a, c) = 1 \\ (d, d) = 1 & (a, d) = 1 \end{cases}$$

$$(b, d) = 1 \quad \begin{cases} (d, a) = 1 & \therefore (b, a) = 1 \\ (d, b) = 1 & (b, b) = 1 \\ (d, c) = 1 & (b, c) = 1 \\ (d, d) = 1 & (b, d) = 1 \end{cases}$$

$$R^{(4)} = a \begin{bmatrix} a & b & c & d \\ 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 \\ c & 0 & 0 & 0 \\ d & 1 & 1 & 1 \end{bmatrix}$$

Transitive closure:

Floyd's Algorithm:

It computes the distance matrix D of a weighted graph with n vertices through a series of n by n matrices.

$$D^{(0)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}$$

where the element $d_{ij}^{(k)}$ in the i^{th} row and j^{th} column of matrix $D^{(k)}$ ($k=0, 1, 2, \dots, n$) is equal to the length of the shortest path among all paths from the i^{th} vertex to j^{th} vertex, with each intermediate vertex, if any, numbered not higher than k .

→ $D^{(0)}$ is the cost adjacency matrix

Recurrence Relation for Floyd's Algorithm:

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

$$\boxed{d_{ij}^{(0)} = a_{ij}}$$

- the element in the i^{th} row & j^{th} column of the current distance matrix $D^{(k-1)}$ is replaced by the sum of the elements in the same row i & the k^{th} column & in the same column j & the k^{th} column iff the latter sum is smaller than its current value.

Algorithm

ALGORITHM FLOYD ($A[1 \dots n][1 \dots n]$)

//Finds the shortest path b/w all pair of vertices.

// Input - The cost adjacency matrix A

// Output - the distance matrix D

{

$D = A$

for $k=1$ to n do

 for $i=1$ to n do

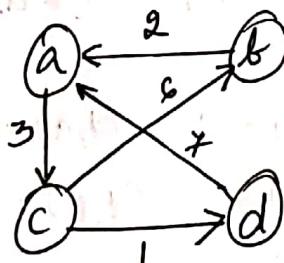
 for $j=1$ to n do

$$D[i, j] = \min \{D[i, j], D[i, k] + D[k, j]\}$$

return D

}

Problem: Apply Floyd's Algorithm to the following graph & find the shortest paths b/w all pairs of vertices.



Adjacency matrix $A = D^{(0)} =$

$$\begin{array}{l} \text{a b c d} \\ \hline \text{a} & 0 & \infty & 3 & \infty \\ \text{b} & 2 & 0 & \infty & \infty \\ \text{c} & \infty & 5 & 0 & 1 \\ \text{d} & 6 & \infty & \infty & 0 \end{array}$$

highlight 1st row & 1st column & observe non zero, non infinity values.

$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & \neq & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

$(b, a) = 2$ $(a, c) = 3$
 $\therefore (b, c) = \min(2, 3+2) = 5$
 $(d, a) = 6$ $(a, c) = 3$
 $\therefore (d, c) = \min(6, 6+3) = 9$

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \infty & \neq & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

$(c, b) = 7$ $(b, a) = 2$
 $\therefore (c, a) = \min(7, 2+2) = 9$
 $(b, c) = 5$ $(c, b) = 7$ $\therefore (b, b) \rightarrow \text{no change}$

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

$(a, c) = 3$ $(c, a) = 9$ $\therefore (a, a) - \text{no change}$
 $(a, c) = 3$ $(c, b) = 7$ $\therefore (a, b) = \min(10, 7+3) = 10$
 $(a, c) = 3$ $(c, d) = 1$ $\therefore (a, d) = \min(10, 3+1) = 4$

$(b, c) = 5$ $\begin{cases} (c, a) = 9 & \therefore (b, a) - \text{no change} \\ (c, b) = 7 & (b, b) - \text{no change} \\ (c, d) = 1 & (b, d) = \min(10, 5+1) = 6 \end{cases}$

$(d, c) = 9$ $\begin{cases} (c, a) = 9 & (d, a) = \text{no change} \\ (c, b) = 7 & (d, b) = \min(10, 9+7) = 16 \\ (c, d) = 1 & (d, d) = \text{no change} \end{cases}$

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 9 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

$$(a, d) = 4 \left\{ \begin{array}{l} (d, a) = 6 \quad \therefore (a, a) - \text{No change} \\ (d, b) = 16 \quad \therefore (a, b) - \text{No change} \\ (d, c) = 9 \quad \therefore (a, c) - \text{No change} \\ (d, d) = 0 \quad \therefore (a, d) - \text{No change} \end{array} \right.$$

$$(b, d) = 6 \left\{ \begin{array}{l} (d, a) = 6 \quad \therefore (b, a) = \text{No change} \\ (d, b) = 16 \quad \therefore (b, b) = \text{No change} \\ (d, c) = 9 \quad \therefore (b, c) = \text{No change} \\ (d, d) = 0 \quad \therefore (b, d) = \text{No change} \end{array} \right.$$

$$(c, d) = 1 \left\{ \begin{array}{l} (d, a) = 6 \quad \therefore (c, a) = \min(9, 1+6) = 7 \\ (d, b) = 16 \quad \therefore (c, b) = \text{No change} \\ (d, c) = 9 \quad \therefore (c, c) = \text{No change} \\ (d, d) = 0 \quad \therefore (c, d) = \text{No change} \end{array} \right.$$

$$D^4 = \underbrace{\begin{matrix} & a & b & c & d \\ a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{matrix}}_{\text{Distance Matrix}}$$

problem

① Solve the following to find transitive closure.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

② solve the all pairs shortest paths problem for the digraph given a weight matrix

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

Optimal Binary Search Tree:

- It is a binary search tree which provides the smallest possible search time for a given sequence of access probabilities.

Example:

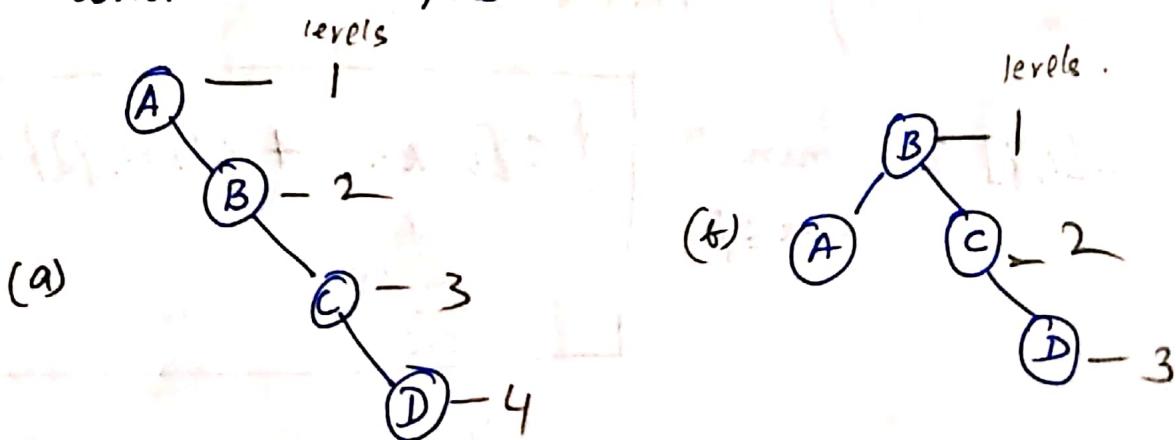
Let there be four keys A, B, C, D with probabilities 0.1, 0.2, 0.4 and 0.3 respectively. The number of binary search tree that can be formed

$$= \frac{2^n C_n}{n+1}$$

so 4 Keys

$$= \frac{2^4 C_4}{5} = \frac{8C_4}{5} = \frac{8! / 4!(8-4)!}{5} = 14$$

consider 2 possible BST with A, B, C, D



- Average number of comparisons in a successful search in tree (a)

is $0.1*1 + 0.2*2 + 0.4*3 + 0.3*4$
 $= 2.9$

tree (v) \rightarrow

$$0.1 \times 2 + 0.2 \times 1 + 0.4 \times 2 + 0.3 \times 3 = 2.1$$

using exhaustive search approach is unrealistic:
as finding solution for all different
binary tree will grow to $\frac{4^n}{n^{1.5}} \rightarrow \infty$

Dynamic programming Approach:

- set $a_1, \dots, a_n \rightarrow$ distinct keys ordered from the smallest to largest.
- $p_1, \dots, p_n \rightarrow$ probability of searching for them.
- let $c[i, j] -$ smallest average number of comparisons made in successful search in a binary search tree T_i^j made up of keys a_i, \dots, a_j , where i, j are some integer indices, $1 \leq i \leq j \leq n$.

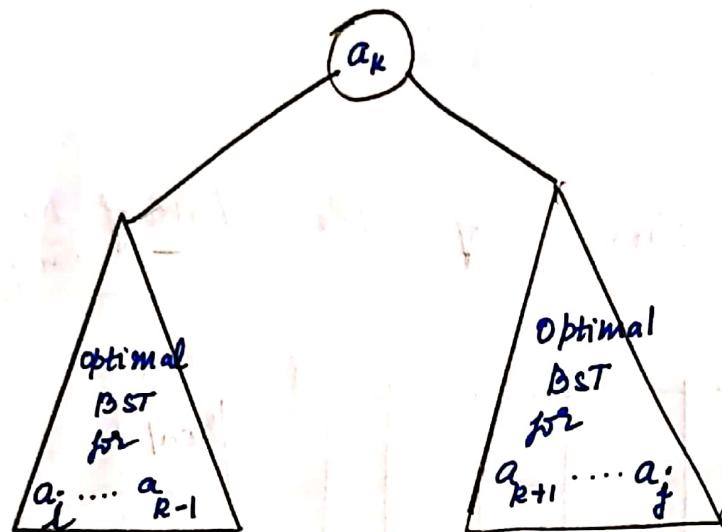
$$c[i, j] = \min_{i \leq k \leq j}$$

$$\boxed{\left\{ c[i, k-1] + c[k+1, j] \right\} + \sum_{s=i}^j p_s}$$

$$\nexists 1 \leq i \leq j \leq n$$

Note:

- $C[i, i-1] = 0 \quad \forall 1 \leq i \leq n+1$
- number of compositions in empty tree
- $C[i, i] = p_i \quad \forall 1 \leq i \leq n$
- one node binary search tree with a_i



Binary Search Tree (BST) with root a_k
and two optimal binary search subtrees

T_i^{k-1} and T_{k+1}^j

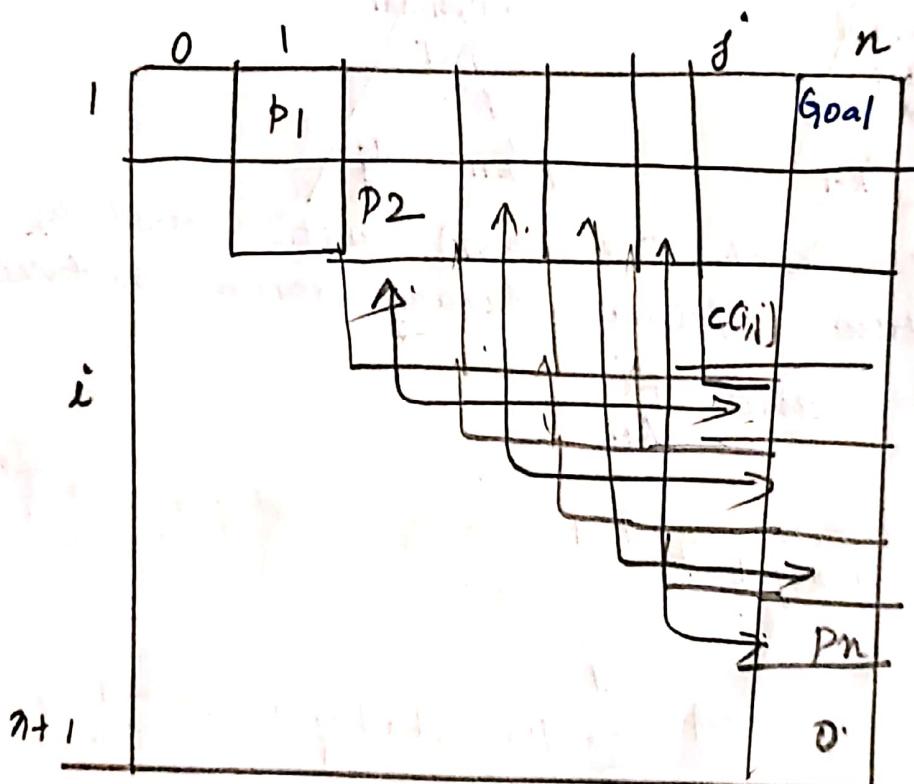
$$C[i, j] = \min_{i \leq k \leq j} \left\{ p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) \right. \\ \left. + \sum_{s=i}^j p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^j + 1) \right\}$$

= :

$$= \boxed{\min_{i \leq k \leq j} \left\{ C[i, k-1] + C[k+1, j] \right\} + \sum_{s=i}^j p_s}$$

a	b	c	d
a	ab	abc	abcd
b	x	bc	bcd
c	x	x	cd
d	x	x	x

General structure of the table of Dynamic programming



Ex: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.3	0.3

s.t.:

Initial table ~~book~~:

Main Table

	0	1	2	3	4
0	0.1				
1		0.2			
2			0.4		
3				0.3	
4					0
5					

Root Table.

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Compute $c[1,2]$ for $k=1, 2$

$$c[i,j] = \min_{i \leq k \leq j} \{ c[i, k-1] + c[k+1, j] \} + \sum_{s=i}^j p_s$$

$$c[1,2] = \min_{k=1, 2} \{ c[1,0] + c[2,2] \} + 0.3$$

$$c[1,2] = \min_{k=1, 2} \{ c[1,1] + c[3,2] \} + 0.3$$

$$= \min \{ 0 + 0.2 + 0.3 = 0.5 \quad = 0.4 \}$$

$$k=2 : 0.1 + 0 + 0.3 = 0.3$$

$$C[1,2] = 0.4 \quad \text{with } k=2$$

- update the table with $C[1,2]$ with
 $0.4 \& R[1,2] = 2$

Now compute $C[2,3]$ for $k=2, 3$

$$C[2,3] = \min \quad k=2 \rightarrow C[2,1] + C[3,3] + 0.6$$

$$k=3 \rightarrow C[2,2] + C[4,3] + 0.6$$

$$= \min \quad k=2 \rightarrow 0 + 0.4 + 0.6 = 1.0$$

$$k=3 \rightarrow 0.2 + 0 + 0.6 = 0.8$$

$$C[2,3] = 0.8 \quad \text{with } k=3$$

- update the table with

$$C[2,3] \text{ with } 0.8 \& R[2,3] = 3$$

Now compute $3,4$ for $k=3 \& 4$

$$k=3 \rightarrow C[3,2] + C[4,4] + 0.7$$

$$C[3,4] = \min$$

$$k=4 \rightarrow C[3,3] + C[5,4] + 0.7.$$

$$= \min \quad k=3 \rightarrow 0 + 0.3 + 0.7 = 1.0$$

$$k=4 \rightarrow 0.4 + 0 + 0.7 = 1.1$$

- Now compute $c[2,4]$ for $k=2,3,4$

$$c[2,4] = \min \begin{cases} k=2 \rightarrow c[2,1] + c[3,4] + 0.9 \\ k=3 \rightarrow c[2,2] + c[4,4] + 0.9 \\ k=4 \rightarrow c[2,3] + c[5,4] + 0.9 \end{cases}$$

$$= \min \begin{cases} k=2 \rightarrow 0 + 1.0 + 0.9 = 1.9 \\ k=3 \rightarrow 0.2 + 0.3 + 0.9 = 1.4 \\ k=4 \rightarrow 0.8 + 0 + 0.9 = 1.7 \end{cases}$$

$$c[2,4] = 1.4 \quad \text{with } k=3$$

update the table with $c[2,4] = 1.4$ &
 $R[2,4] = 3$

- Now compute $c[1,4]$ for $k=1,2,3,4$

$$c[1,4] = \min \begin{cases} k=1 \rightarrow c[1,0] + c[2,4] + 1.0 \\ k=2 \rightarrow c[1,1] + c[3,4] + 1.0 \\ k=3 \rightarrow c[1,2] + c[4,4] + 1.0 \\ k=4 \rightarrow c[1,3] + c[5,4] + 1.0 \end{cases}$$

$$c[1,4] = \min \begin{cases} k=1 \rightarrow 0 + 1.4 + 1 = 2.4 \\ k=2 \rightarrow 0.1 + 1.0 + 1.0 = 2.1 \\ k=3 \rightarrow 0.4 + 0.3 + 1.0 = 1.7 \\ k=4 \rightarrow 1.1 + 0 + 1.0 = 2.1 \end{cases}$$

$$\boxed{\therefore c[3,4] = 1.0 \text{ with } k=3}$$

update the table with $c[3,4] = 1.0$ &
 $R[3,4] = 3$

Now compute $c[1,2]$ for $k=1, 2$ and 3.

$$c[1,3] = \min \begin{cases} k=1 \rightarrow c[1,0] + c[2,3] + 0.7 \\ k=2 \rightarrow c[1,1] + c[3,3] + 0.7 \\ k=3 \rightarrow c[1,2] + c[4,3] + 0.7 \end{cases}$$

$$= \min \begin{cases} k=1 \rightarrow 0 + 0.8 + 0.7 = 1.5 \\ k=2 \rightarrow 0.1 + 0.4 + 0.7 = 1.2 \\ k=3 \rightarrow 0.4 + 0 + 0.7 = 1.1 \end{cases}$$

$$\boxed{\therefore c[1,3] = 1.1 \text{ with } k=3}$$

update the table with $c[1,3] = 1.1$
 and $R[1,3] = 3$.

$$C[1,4] = 1.7 \quad \text{with} \quad k=3$$

- update the table with $C[1,4] = 1.7$ and
 $R[1,4] = 3$

Final Table

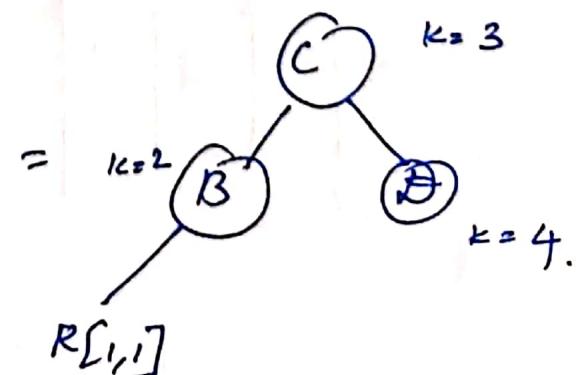
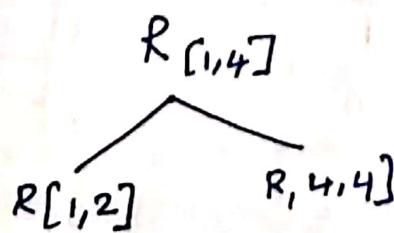
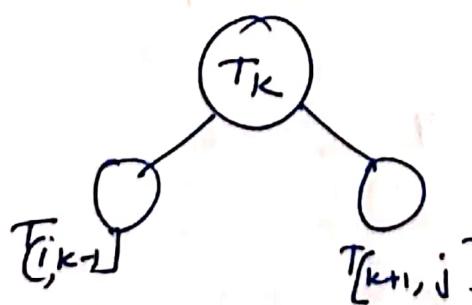
Main Table

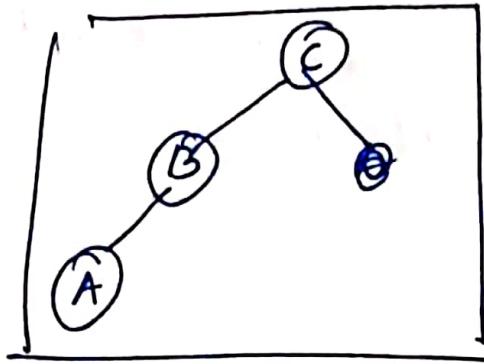
	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2	0	0.2	0.8	1.4	
3	0	0.4	1.0		
4	0	0.3			
5			0		

Root Table

	1	2	3	4
1	1	2	3	3
2		2	3	3
3			3	3
4				4

- To build a tree $R[i][n]$, i.e $R[1][4]$ is
 $\text{Root} \rightarrow 3$ i.e C



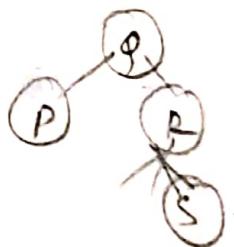


The optimal
Binary search Tree

Ex: construct optimal Binary Search tree for the following data.

key	P	Q	R	S
probability	$\frac{1}{20}$	$\frac{1}{5}$	$\frac{1}{10}$	$\frac{1}{20}$
	1	4	2	1

} Take LCM



sol

	0	1	2	3	4
1	0	1	6	10	13
2	0	4	8	12	
3		0	2	4	
4			0	1	
5					0

	1	2	3	4
1	1	2	2	2
2		2	2	2
3			3	3
4				4.

- If the keys are in nondecreasing order, then running time of the algorithm to $\Theta(n^2)$.

Algorithm : optimalBST ($P[1 \dots n]$)

// Finds an optimal binary tree by dynamic programming

// Input : An array $P[1 \dots n]$ of search probabilities for a sorted list of n keys.

// Output: Average number of comparisons in successful searches in the optimal BST and table R of subtrees root in the optimal BST.

```
for i ← 1 to n do
    c[i, i-1] ← 0;
    c[i, i] ← p[i];
    R[i, i] ← i;
    c[n+1, n] ← 0;

for d ← 1 to n-1 do
    for i ← 1 to n-d do
        j = i+d;
        minval ← ∞;
        for k ← i to j do
            if c[i, k-1] + c[k+1, j] < minval
                minval ← c[i, k-1] + c[k+1, j];
                kmin ← k;
        R[i, j] = kmin;
        sum ← p[i];
```

```
sum ← p[i];
for s ← i+1 to j do
    sum ← sum + p[s];
c[i, j] ← minval + sum
return c[i, n], R
```

The knapsack problem & memory functions

Problem statement:

Given n objects of known weights (w_1, w_2, \dots, w_n) and profits (v_1, v_2, \dots, v_n) & the knapsack capacity $-W$,

the task is to find most valuable subset of items that fit into the knapsack.

- All the weights and knapsack capacity are tre integers, the item values do not have to integers.

Dynamic program Approach:

- Let's consider an instance of the problem with first i items, $1 \leq i \leq n$ with the weights w_1, \dots, w_i , values (profits) v_1, \dots, v_i & knapsack capacity j , $1 \leq j \leq W$.
- Let $v[i, j]$ be the value of the optimal solution to the instance i.e value of the most valuable subset of the first i items that fit onto the knapsack of capacity j .
- This subset can be divided into 2 categories:
 - (i) those that do not include i^{th} item;
 - (ii) Those that include i^{th} item.

Recurrence Relation:

- (i) Among the subsets that do not include i^{th} item, the value of an optimal subset is $V[i-1, j]$.
- (ii) Among the subsets that include i^{th} item hence $(j-w_i \geq 0)$, an optimal subset is made up of this item & an optimal subset of the first $(i-1)$ items that fit into the knapsack of capacity $j-w_i$. The value of such an optimal subset is $v_i + V[i-1, j-w_i]$.

∴ The recurrence relation is given by:

$$V[i, j] = \begin{cases} V[i-1, j] \\ \max \{ V[i-1, j], v_i + V[i-1, j-w_i] \} \end{cases} \quad \text{if } j-w_i \geq 0$$

with initial conditions:

$$V[0, j] = 0 \quad \text{for } j \geq 0 \text{ &}$$

$$V[i, 0] = 0 \quad \text{for } i \geq 0$$

problem: find an optimal solution for the following instance of knapsack problem using dynamic programming:

item	weight	value	
1	2	12	
2	1	10	
3	3	20	
4	2	15	capacity, $w=5$

Sol since $n=4$ & $w=5$, the solution matrix will have $n+1$ rows & $w+1$ columns as shown below:

	Capacity j					
i	0	1	2	3	4	5
0						
1						
2						
3						
4						

- we know from the recurrence relation that $V[i, j] = 0$ for $i=j=0$.
[$i=0$ indicates no objects & $j=0$ indicates knapsack capacity is not filled].

- From the recurrence relation

$$\text{we know } v[i, j] = 0$$

so first row and first column = 0

- consider $i=1, w_i=2, v_i=12$ & compute the values for $j=1, 2, 3, 4, 5$.

$$\text{ie } v[i, j] = v[i-1, j] \quad \text{if } (j-w_i \leq 0 \text{ & } j \leq w_i)$$

$$v[i, j] = \max [v[i-1, j], v_i + v[i-1, j-w_i]]$$

$$\text{if } (j-w_i > 0 \text{ & } j \geq w_i)$$

$$v[1, 1] = v[0, 1] = 0$$

$$v[1, 2] = \max [v[0, 2], 12 + v[0, 0]] = \max (0, 12) = 12$$

$$v[1, 3] = \max [v[0, 3], 12 + v[0, 1]] = \max (0, 12) = 12$$

$$v[1, 4] = \max [v[0, 4], 12 + v[0, 2]] = \max (0, 12) = 12$$

$$v[1, 5] = \max [v[0, 5], 12 + v[0, 3]] = \max (0, 12) = 12$$

→ consider $i=2, w_i=1, v_i=10$ & compute the values for j :

$$v[2, 1] = \max [v[1, 1], 10 + v[1, 10]] = \max (0, 10) = 10$$

$$v[2, 2] = \max [v[1, 2], 10 + v[1, 1]] = \max (12, 10) = 12$$

$$v[2, 3] = \max [v[1, 3], 10 + v[1, 2]] = \max (12, 22) = 22$$

$$v[2, 4] = \max [v[1, 4], 10 + v[1, 3]] = \max (12, 22) = 22$$

$$v[2, 5] = \max [v[1, 5], 10 + v[1, 4]] = \max (12, 22) = 22$$

- consider $i=3$, $w_i = 3$, $v_i = 20$ & compute the values:

$$v[3,1] = v[2,1] = 10$$

$$v[3,2] = v[2,2] = 12$$

$$v[3,3] = \max[v[2,3], 20 + v[2,0]] = \max(22, 20) \\ = 22$$

$$v[3,4] = \max[v[2,4], 20 + v[2,1]] = \max(22, 30) = 30$$

$$v[3,5] = \max[v[2,5], 20 + v[2,2]] = \max(22, 32) = 32$$

- consider $i=4$, $w_i = 2$, $v_i = 15$

$$v[4,1] = v[3,1] = 10$$

$$v[4,2] = \max[v[3,2], 15 + v[3,0]] = \max(12, 15) = 15$$

$$v[4,3] = \max[v[3,3], 15 + v[3,1]] = \max(22, 15) = 22$$

$$v[4,4] = \max[v[3,4], 15 + v[3,2]] = \max(30, 27) = 30$$

$$v[4,5] = \max[v[3,5], 15 + v[3,3]] = \max(32, 37) = 37$$

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Solve the following instance of knapsack problem
using dynamic programming:

item	weight	value
1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

$$W = 6$$

Travelling Salesman problem (TSP)

Problem statement:

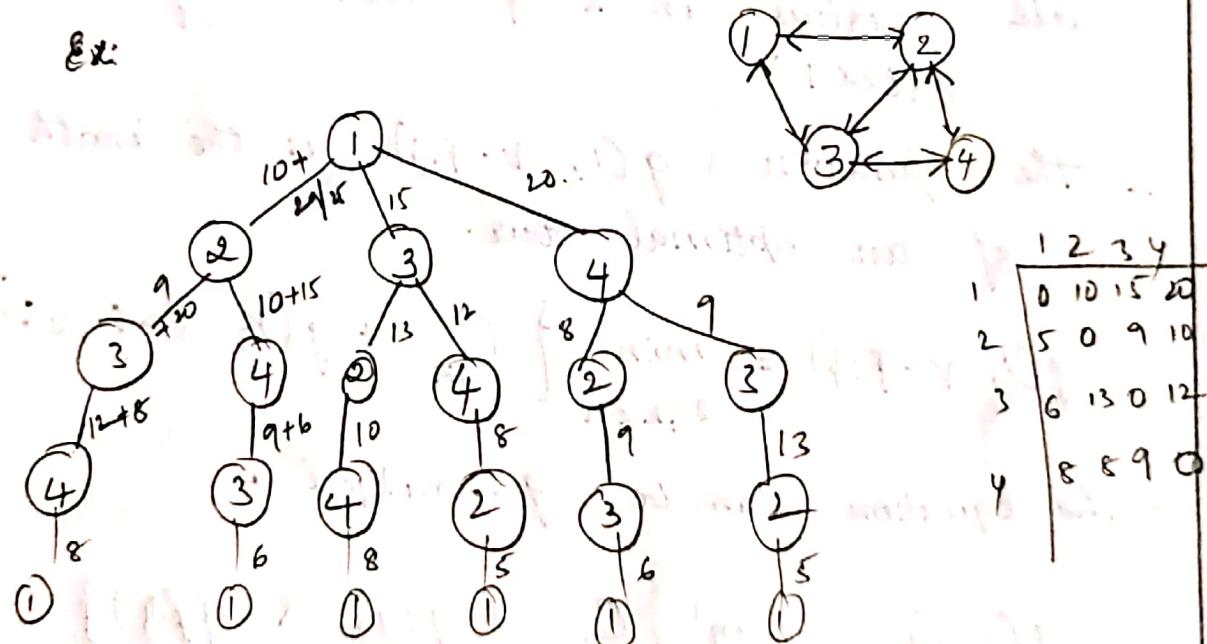
Given \rightarrow n cities & distances between them

- Source (starting point).

The task is: the salesperson starts his tour from source and visits all $(n-1)$ cities exactly once and returns back to the source. Objective is - to finish this tour at minimum cost.

Dynamic programming Approach:

Ex:



$$g(1, \{2, 3, 4\}) = \min \left(C_{1k} + g(k, \{2, 3, 4\} - \{k\}) \right)$$

from 1 to 2, 3, 4.

Remaining.

subtract
which is
considered

- TSP can be modelled as a graph $G = (V, E)$, where vertices represent cities, weights linked to edges represent distance/cost involved between adjacent vertices]
 - Consider source vertex as 1
 - Tour is a simple path that starts and ends at vertex 1
 - Every tour consists of an edge $(1, k)$ for some $k \in V - \{1\}$ & a path from vertex k to vertex 1
 - Let $g(i, s)$ be length of shortest path starting at vertex i , going through all vertices in s & terminating at vertex 1
 - The function $g(1, V - \{1\})$ is the length of an optimal tour.
- $$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \left\{ C_{1k} + g(k, V - \{1, k\}) \right\}$$
- The equation can be generalized to:

$$g(i, s) = \min_{j \in s} \left\{ C_{ij} + g(j, V - \{i, j\}) \right\}$$

for $i \notin s$

The base case is given by:

$$g(i, \emptyset) = C_{ii}, \quad 1 \leq i \leq n$$

In summary: equations used to solve TSP are:

$$(i) g(i, \emptyset) = C_{ii}, \quad 1 \leq i \leq n, \quad i \neq 1$$

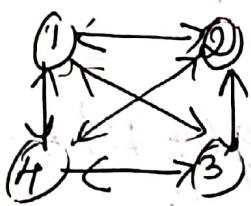
$$(ii) g(i, s) = \min_{j \in s} \{ C_{ij} + g(j, s - \{j\}) \}$$

for $i \neq 1, i \notin s, 1 \notin s$

& compute this for $|s| < n-1$

$$(iii) g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{ C_{1k} + g(k, V - \{1, k\}) \}$$

Problem → solve the following graph for TSP



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

Sol → use base case:

$$g(i, \emptyset) = C_{ii} \quad \text{for } 1 \leq i \leq n, i \neq 1 \text{ as } s = \emptyset \Rightarrow |s| = 0$$

$$g(2, \emptyset) = C_{22} = 5$$

$$g(3, \emptyset) = C_{33} = 6$$

$$g(4, \emptyset) = C_{44} = 8$$

use

$$g(i, s) = \min_{j \in s} \{ C_{ij} + g(j, s - f_j) \}$$

for $i \neq 1$, $i \in s$, $|s| = 1$

for $i=2$

$$g(2, \{3\}) = \min_{j \in \{3\}} \{ C_{23} + g(3, \emptyset) \} = 9 + 6 = 15$$

$$g(2, \{4\}) = \min_{j \in \{4\}} \{ C_{24} + g(4, \emptyset) \} = 10 + 8 = 18$$

for $i=3$

$$g(3, \{2\}) = \min_{j \in \{2\}} \{ C_{32} + g(2, \emptyset) \} = 13 + 5 = 18$$

$$g(3, \{4\}) = \min_{j \in \{4\}} \{ C_{34} + g(4, \emptyset) \} = 12 + 8 = 20$$

for $i=4$

$$g(4, \{2\}) = \min_{j \in \{2\}} \{ C_{42} + g(2, \emptyset) \} = 8 + 5 = 13$$

$$g(4, \{3\}) = \min_{j \in \{3\}} \{ C_{43} + g(3, \emptyset) \} = 9 + 6 = 15$$

$|s|=2$

for $i=2$

$$g(2, \{3, 4\}) = \min_{j \in \{3, 4\}} \{ C_{23} + g(3, 4), C_{24} + g(4, 3) \}$$

$$= \min (9+20, 10+15) = \min (29, 25)$$

= 25

for $i=3$

$$g(3, \{2, 4\}) = \min_{j \in \{2, 4\}} \{C_{3j} + g(2, 4), C_{34} + g(4, 2)\}$$
$$= \min(13+18, 12+13) = \min(31, 25) = \underline{25}$$

for $i=4$

$$g(4, \{2, 3\}) = \min_{j \in \{2, 3\}} \{C_{4j} + g(2, 3), C_{43} + g(3, 2)\}$$
$$= \min(8+15, 9+18)$$
$$= \min(23, 27) = 23$$

since $|S| < n-1$, i.e. $|S| < 3$ is met, the iteration stops here:

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

$$g(1, \{1, 2, 3, 4\} - \{1\}) = \min_{k=2, 3, 4} \{C_{1k} + g(k, \{1, 2, 3, 4\} - \{1, k\})\}$$

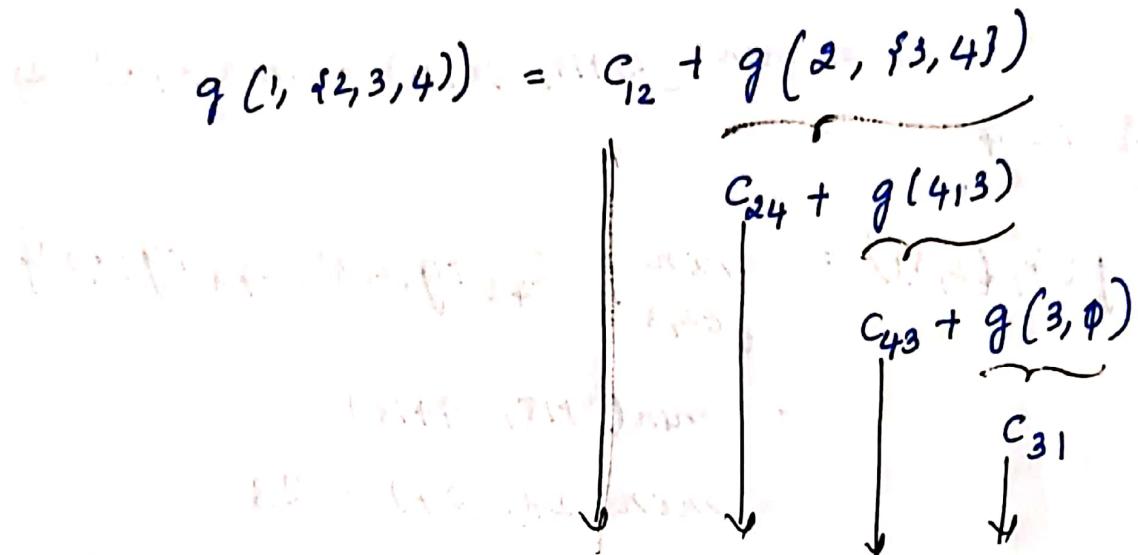
$$g(1, \{2, 3, 4\}) = \min \{C_2 + g(2, \{3, 4\}), C_3 + g(3, \{2, 4\}), C_4 + g(4, \{2, 3\})\}$$

$$= \min(10+25, 15+25, 20+23)$$

$$= \min(35, 40, 43)$$

$$= \underline{35}.$$

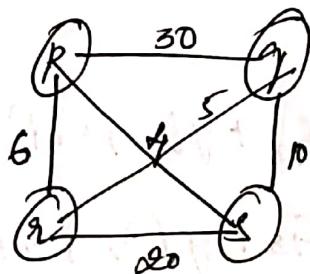
$$\therefore g(1, f_{2,3,4}) = 35 \quad - \text{optimal solution.}$$



Path: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

\therefore optimal tour is $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$ with optimal cost of 35.

point using dynamic programming solve the following instance of TSP.

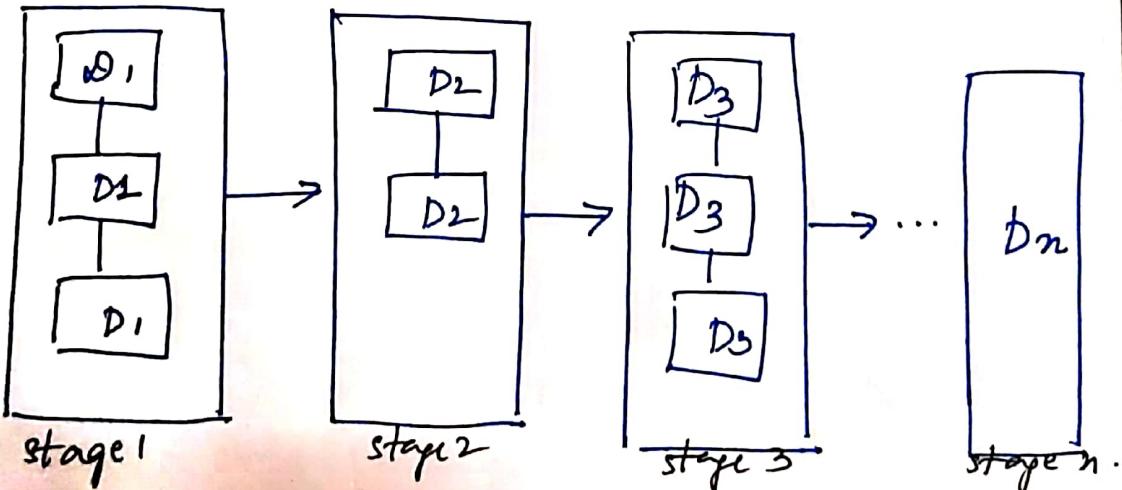


Bellman-Ford Algorithm:

- It is single source shortest path Algo.
- Dijkstra's algorithm works correct on graphs, which have edges weight as positive numbers. If the edge weight are negative, Dijkstra's Algo may not give the correct shortest path.
- Bellman-Ford Algo is used to obtain the shortest paths from a given vertex (source) to all other vertices in a graph with general (+ve or -ve) weights.

Reliability Design:

- set up a system which consists of set of devices. Each device is associated with cost and reliability.
- the probability that device 'i' will work properly is called Reliability of that device.
- Let r_i be the reliability of device D_i , then reliability of entire s/m is $\prod r_i$.
- Even if reliability of individual device is very good but reliability of entire system may not be good.
- To obtain good performance from entire system, individual device can be duplicated.
- different devices are connected in series & duplicated devices are connected in parallel



- let m_i be the copies of Device D_i , then $(1-r_i)^{m_i}$ be the probability that all m_i have a malfunction. Hence stage reliability is $1 - (1-r_i)^{m_i}$

Module - 5

Backtracking:

- Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally.
- Principal idea is to construct solutions of one component at a time & evaluate such partially constructed candidate as follows:
 - Constraints are verified for partially constructed solution, if not violated it is continued.
 - if violated, backtracks to replace the last component of the partially constructed solution with its next option.

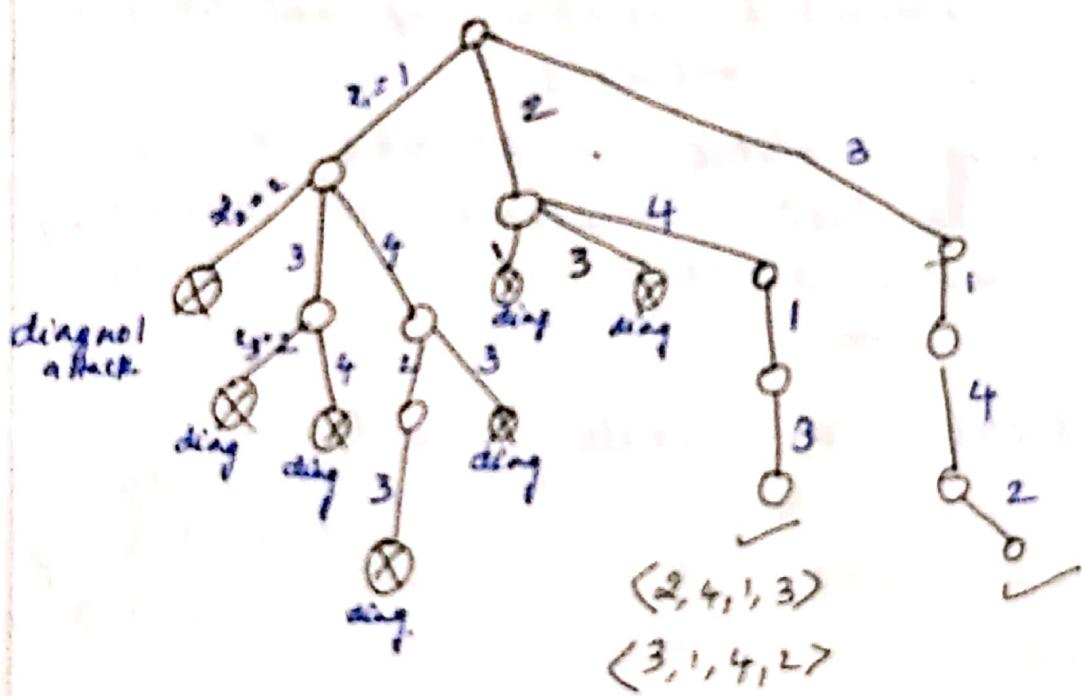
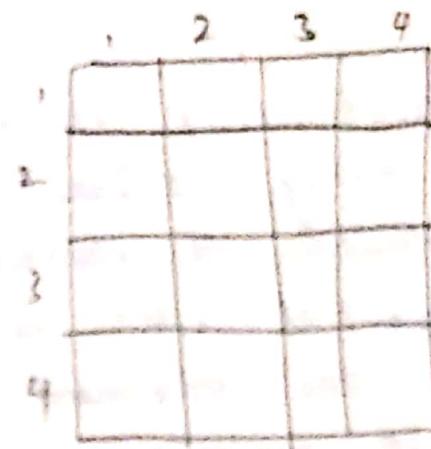
Implementation - state space tree.

- A state space tree consists of nodes representing state, & arc representing the legal moves from one state to another.
- Backtracking is mainly applied to combinatorial problems.

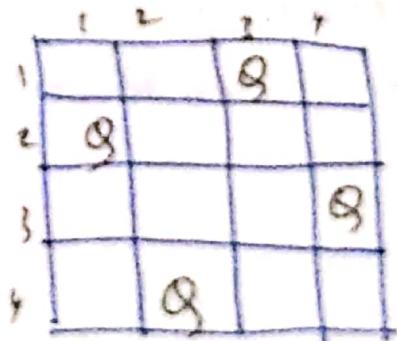
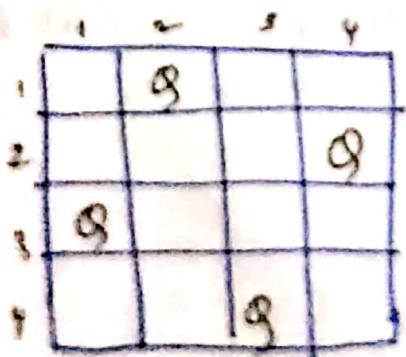
n -queens Problem:

Q_1, Q_2, Q_3, Q_4

→ The problem is to place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.



2 Solutions :



Sum of Subsets:

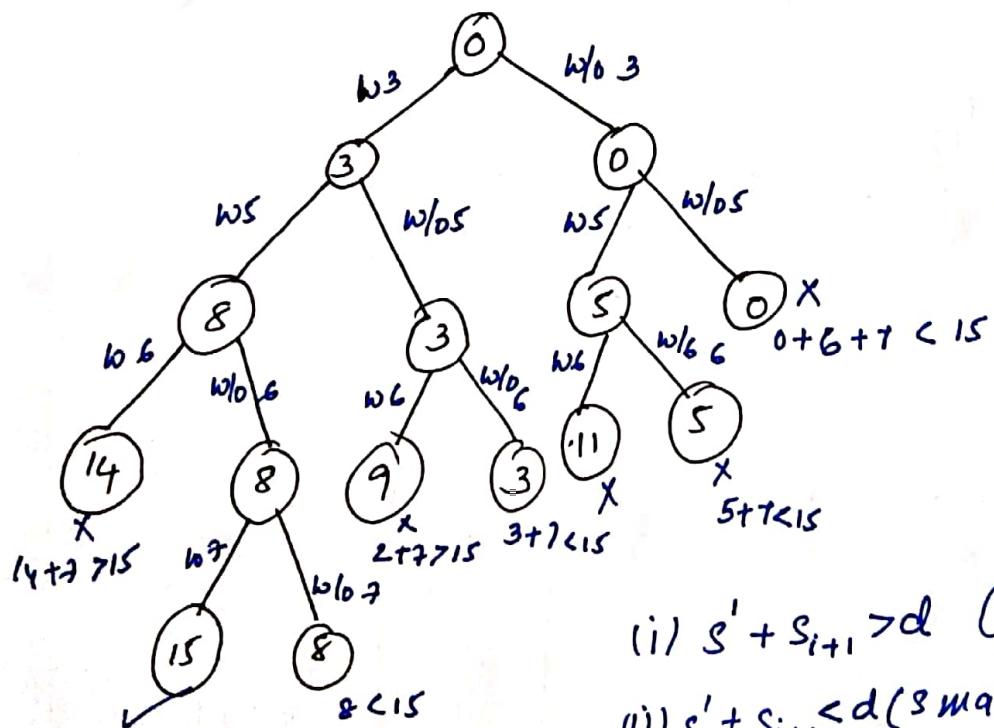
- Given a set of $S = \{s_1, \dots, s_n\}$ of n positive integers, find a subset whose sum is equal to a given positive integer d "

Ex: Let $S = \{1, 2, 5, 6, 8\}$ and $d = 9$, then the subsets summing to d are:

$\{1, 2, 6\}$ and $\{1, 8\}$

problem: construct the state space tree to solve the following instance of subset-sum problem: $S = \{3, 5, 6, 7\}$ and $d = 15$.

Sol \Rightarrow The root of the tree represents starting points. Its left & right children represent the inclusion & exclusion of elements respectively:



- (i) $s'_i + s_{i+1} > d$ (sum's too large)
- (ii) $s'_i + s_{i+1} < d$ (sum's small)

* Apply backtracking to solve the following instance of subset-sum problem.

(i) $S = \{1, 3, 4, 5\}$ and $d = 11$

(ii) $S = \{11, 13, 24, 7\}$ and $d = 31$

and draw the state-space tree.

Find all possible solutions:

* Let $w = \{5, 7, 10, 12, 15, 18, 20\}$ & $m = 35$.
Find all subsets of w that sum to m
Draw the state-space tree.

Graph coloring:

Let $G = (V, E)$ be a graph & "m" a positive integer - no of colors.

problem \rightarrow Find whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color and to use only "m" colors.

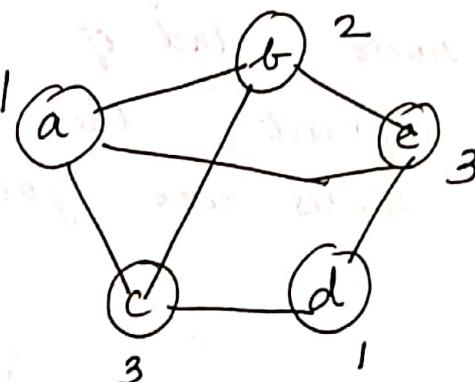
This is called m-color decision problem.

- m color-optimization problem -

smallest integer "m" that can be used to solve m-color decision problem.

This "integer m" is called chromatic number of graph.

- Consider the following graph which can be colored with 3 colors - 1, 2, 3



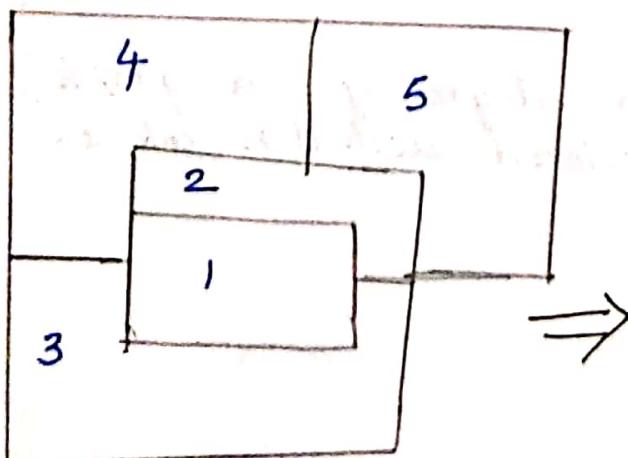
Note: If d is degree of a graph, then it can be colored with $d+1$ colors.

4 color problem for planar graphs:

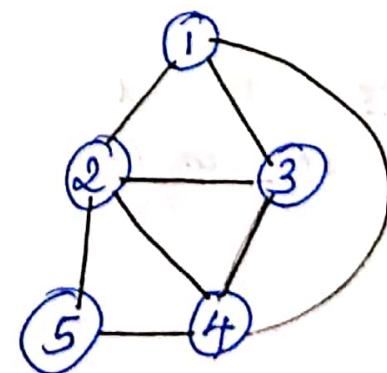
- A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.
- The famous special case of m -colorability decision problem is 4-color problem for planar graph:

"Given any map, can the regions be colored in such a way that no two adjacent regions have the same color, yet only four colors are needed."
- A map can easily be transformed into a graph. Each region of map becomes a node and if two regions are adjacent, then the corresponding nodes are joined by an edge:

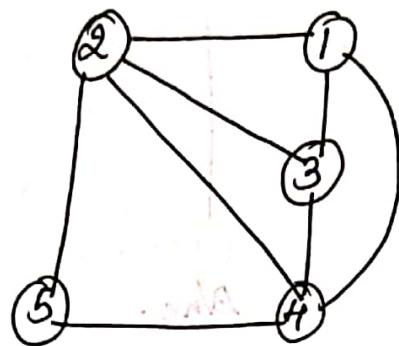
Ex: map



Graph

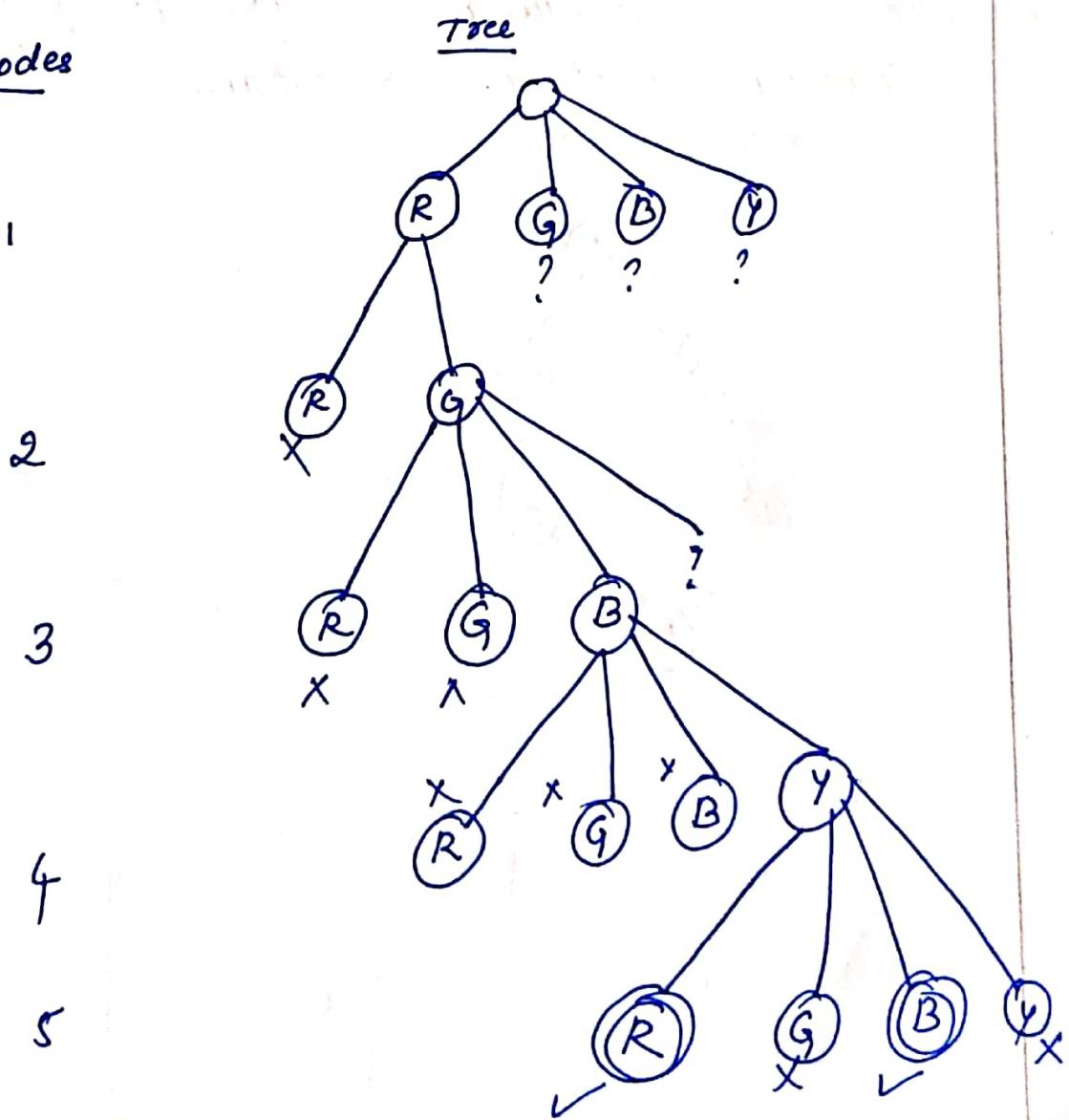


* Solve the following instance of graph coloring problem. Draw the state space tree, let $m=4$

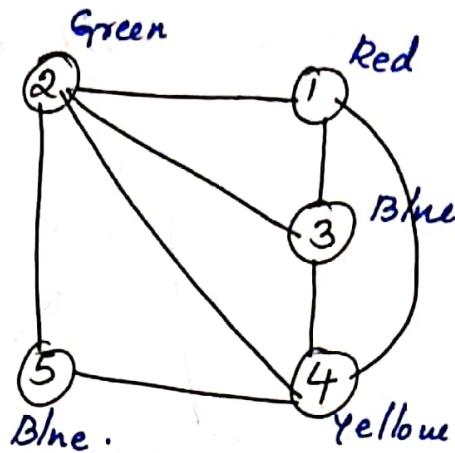
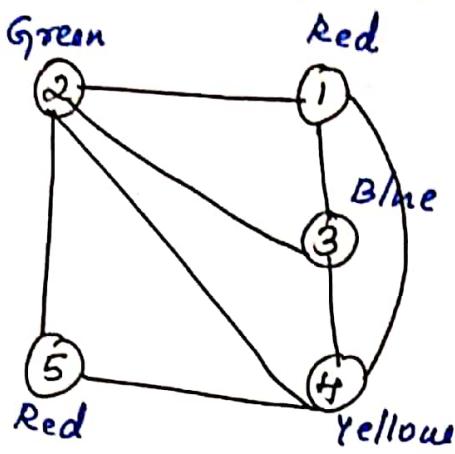


Solutions:

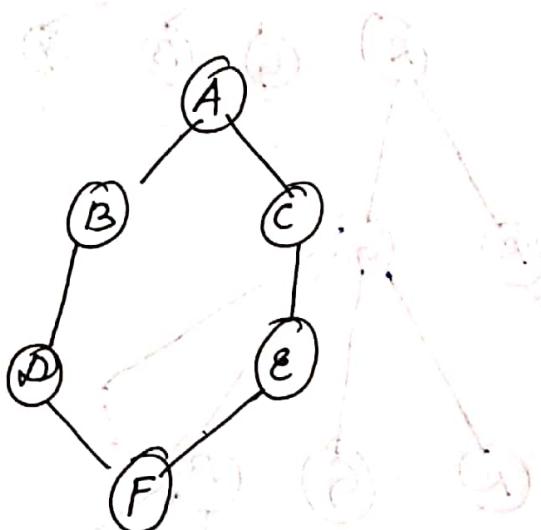
Nodes



Solutions are:

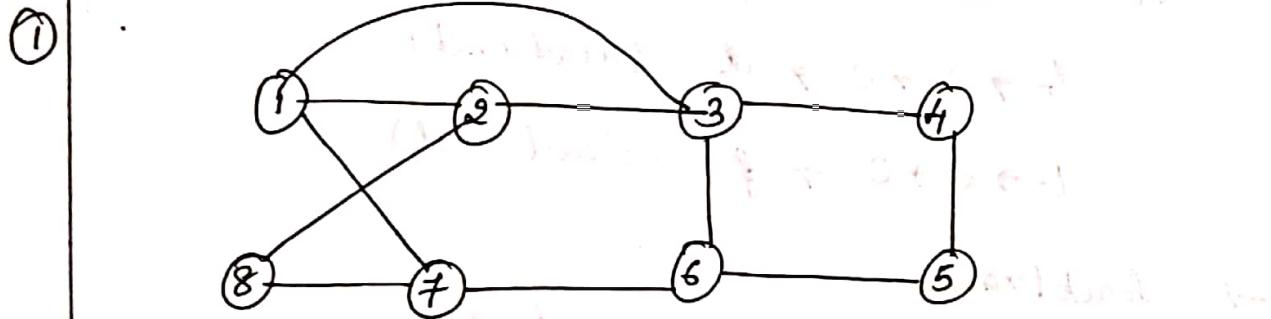


- * solve the following instance of graph coloring problem & obtain the state space tree:

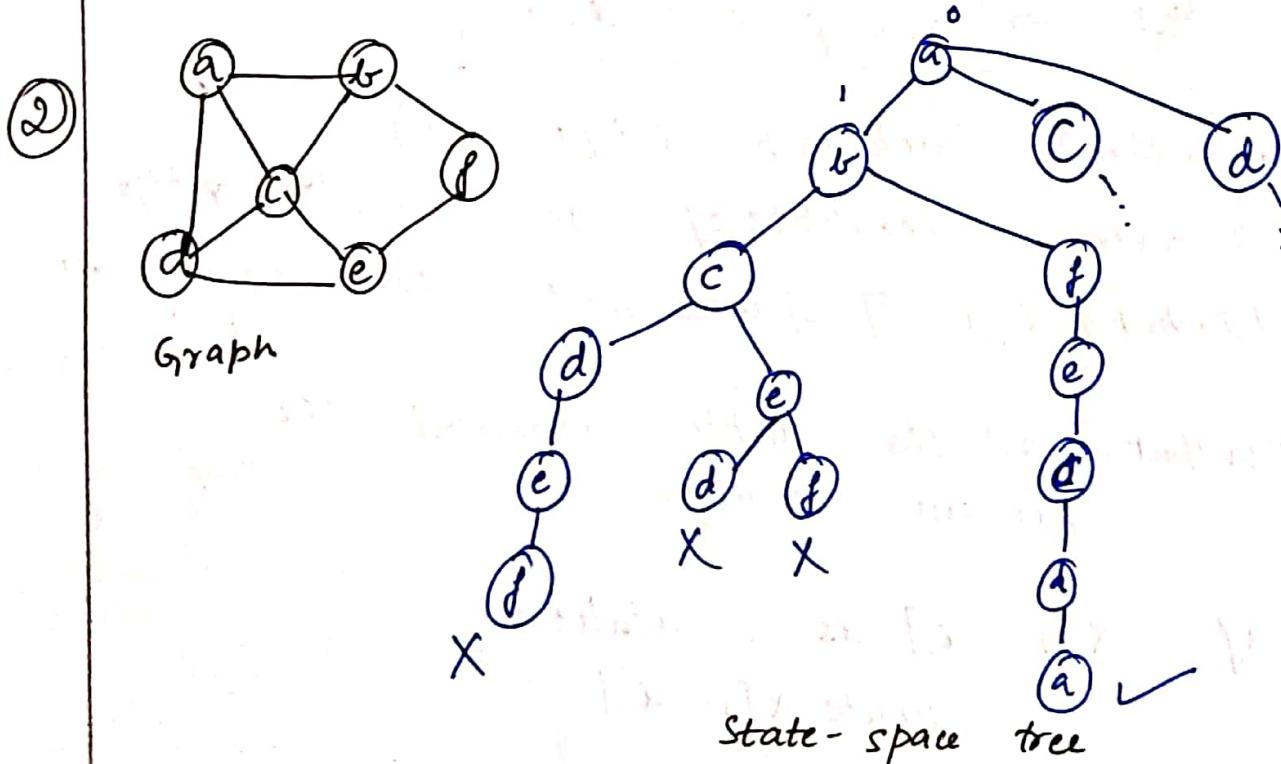


Hamilton cycles:

- Let $G = (V, E)$ be a connected graph with n vertices. A Hamilton cycle is a round trip along n edges of G that visits every vertex once and returns to its starting position.
- Hamilton cycle begins at some vertex $v_i \in G$ and the vertices are listed with edges (v_i, v_{i+1}) where v_i are distinct.
- Ex:



Here $1, 2, 8, 7, 6, 5, 4, 3, 1$ is the Hamilton cycle.



Steps:

- Alphabet 'a' vertex is selected as a root vertex.
- adjacent to a \rightarrow b, c, d
- first select b (DFS)
 - \rightarrow from b, the algorithm proceeds to c if
 - \rightarrow from b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \times dead end.
 - \rightarrow so backtrack from
 - $b \rightarrow c \rightarrow e \rightarrow d$ (dead end)
 - $b \rightarrow e \rightarrow c \rightarrow f$ (dead end)
- \rightarrow backtrack
- $b \rightarrow f \rightarrow e \rightarrow c \rightarrow d \rightarrow a$

solt \rightarrow a, b, f, e, c, d, a

Algorithm: Backtrack ($x[1 \dots i]$)

// Given a template of generic backtracking algo

// Input: $x[1 \dots i]$ specifies first i promising components
of soln

// Output: All the tuples represent the problem solution.

if $x[1 \dots i]$ is a solution
write $x[1 \dots i]$

else

discard
 for each element $x \in S_{i+1}$ consistent
 with $x[1 \dots i]$ & constraints do
 $x[i+1] \leftarrow x$
 Backtrack $\{x[1 \dots i+1]\}$

Hamiltonian cycle

Algorithm HAMILTONIAN(k)
 {
 do
 {
 next-vertex(k);
 if ($x[k] == 0$)
 return n ;
 if ($k == n$)
 print $\{x[1 \dots n]\}$;
 else Hamiltonian($k+1$);
 }
 while (true);
}

Algorithm Next vertex(k)
 {
 do
 {
 $x[k] = (x[k]+1) \bmod (n+1)$;
 if ($x[k] == 0$) return;

if ($G(x[k-1], x[k]) \neq 0$)

{

for $j \leftarrow 1$ to $k-1$ do

if ($x[j] == x[k]$) break;

if ($j == k$)

if ($k < n$ as ($k == n$) $\&$

$G(x[n], x[1]) \neq 0$

return;

}

} while (true);

}

Branch and Bound:

- It is an algorithmic approach to solve discrete & combinatorial optimization problem.
 - In this method, a space tree with all possible solutions is generated. Then partitioning or branching is done at each node of the tree.
 - Bound value is computed at each node.
 - Node's bound value is compared with best solution so far.
 - If the bound value of some node is not better than the best solution, then that corresponding node is not expanded further, such node is called non-promising node.
- Reasons for terminating search paths in state space tree of branch & bound are:
1. value of node's bound is not better than best solution.
 2. The node represents no feasible solutions because the constraints of problem are already violated.
 3. Subset of feasible has no further choices can be made.

Backtracking

1. Typically decision problems can be solved using backtracking.

2. It can consider bad choices also

3. State space tree is searched until the solution is obtained

4. Applications:

- N-Queens
- sum of subset
- Graph coloring etc

5. Solution is traced using Depth first search

6. In this method all possible solutions are tried. If \underline{s} is not satisfying the constraint then backtrack & try another \underline{s}

Branch & Bound

Typically optimization problems can be solved using Branch and Bound.

It proceeds on better solutions. So there can't be a bad solution.

State space tree needs to be searched completely.

- TSP
- 0/1 Knapsack etc

Breadth first search.

It uses bounding values i.e either upper bound / lower bound.

Assignment problem:

Given 'n' people and 'n' jobs, the problem is to assign 'n' jobs to 'n' people such that cost of assignment is as small as possible.

- First lower bound (LB) is computed.

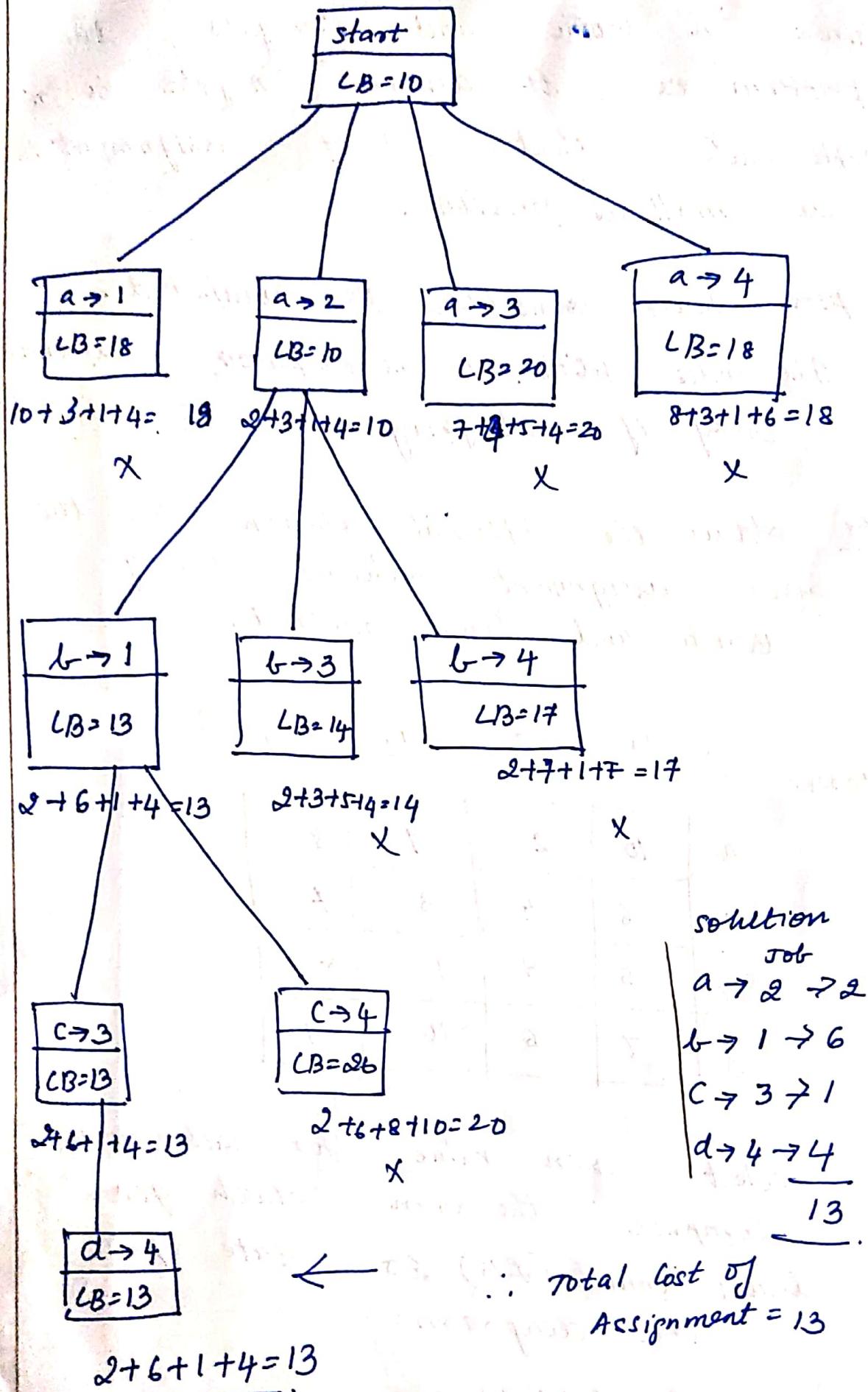
The nodes with LB is compared, & expanded only if satisfying

Ex. obtain the optimal solution for the given assignment problem using branch and bound method:

person	J_1	J_2	J_3	J_4
a	10	2	7	8
b	6	4	3	7
c	5	8	1	8
d	7	6	10	4

so that
select min value for each row &
compute the sum which gives
lower bound (LB) for state space
tree diagram:

$$2 + 3 + 1 + 4 = 10$$



solution
job
 $a \rightarrow 2 \rightarrow 2$
 $b \rightarrow 1 \rightarrow 6$
 $c \rightarrow 3 \rightarrow 1$
 $d \rightarrow 4 \rightarrow 4$

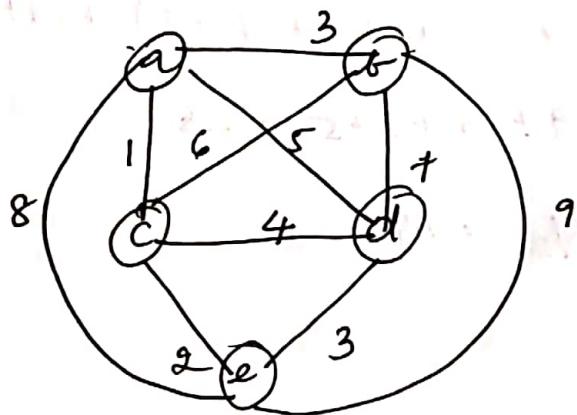
13

Travelling Salesperson - Branch & Bound:

Given 'n' cities and costs incurred in travelling from any city to other, the task is to find the shortest tour possible from a given source city.

problem:

Solve the following instance of TSP using branch & bound method:



compute LB

- for each city i , $1 \leq i \leq n$, find sum ' s_i ' of the distance from city i to the two nearest cities.

Compute sum's of these n numbers, divide the result by 2 i.e $LB = (s/2)$.

$$\therefore LB = [(1+3) + (3+6) + (1+2) + (3+4) + (2+3)]/2 = 28/2 = 14$$

Assumptions:

- (i) Consider only tours that starts at 'a'
- (ii) Generate only tours in which 'b' is listed before 'c'.

- Now consider the edges (g, b) (g, c) (g, d) (g, e) at level 1,

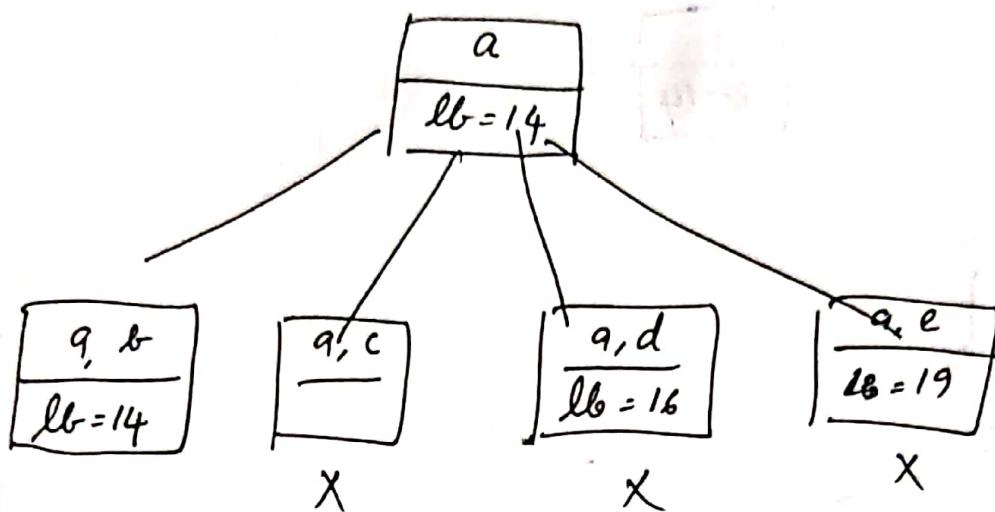
$$\begin{aligned}
 (g, b) &= (3+1) + (3+6) + (1+2) + (3+4) + (2+3) \\
 &= 4 + 9 + 3 + 7 + 5 = 28 \\
 &= \frac{28}{2} = 14
 \end{aligned}$$

$$(g, c) = 14.$$

$(g, c) \rightarrow$ Not considered as b is not before c

$$\begin{aligned}
 (g, d) &= (1+5) + (3+6) + (1+2) + (3+5) + (2+3) \\
 &= 6 + 9 + 3 + 8 + 5 = 31 \\
 &= \frac{31}{2} \approx 16
 \end{aligned}$$

$$\begin{aligned}
 (g, e) &= (1+8) + (3+6) + (1+2) + (4+3) + (2+8) \\
 &= 9 + 9 + 3 + 7 + 10 = 38 \\
 &= \frac{38}{2} = \underline{19}.
 \end{aligned}$$



→ Now consider (b, c), (b, d), (b, e)

level - 2

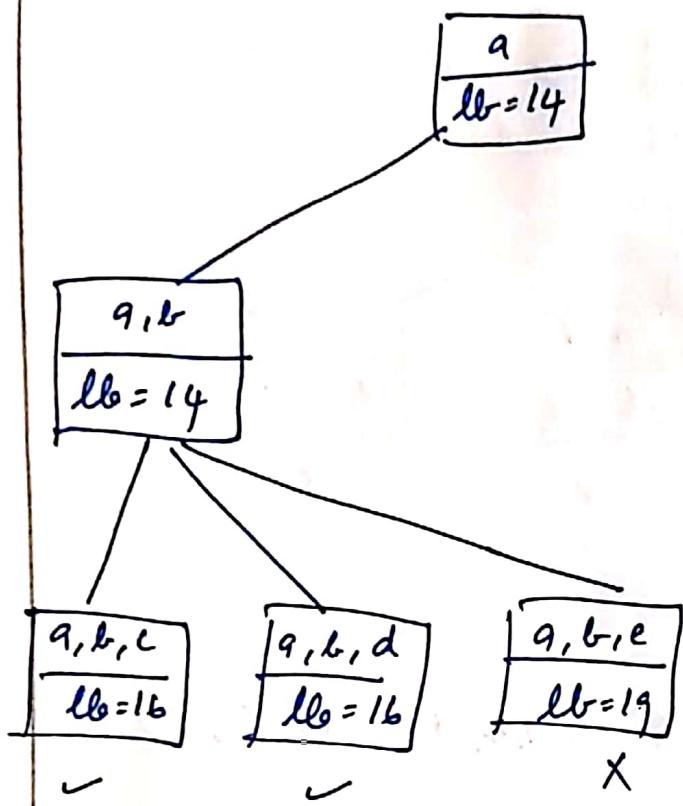
$$\begin{aligned}
 (b, c) &= (1+3) + (6+3) + (6+1) + (3+4) + (2+3) \\
 &= 4 + 9 + 7 + 7 + 5 \\
 &= 32/2 = 16
 \end{aligned}$$

$$(b, c) = \underline{16}$$

$$\begin{aligned}
 (b, d) &= (1+3) + (7+3) + (1+2) + (7+3) + (2+3) \\
 &= 4 + 10 + 3 + 10 + 5 \\
 &= 32/2 = 16
 \end{aligned}$$

$$\begin{aligned}
 (b, e) &= (1+3) + (9+3) + (1+2) + (3+4) + (9+2) \\
 &= 4 + 12 + 3 + 7 + 11 = 37 \\
 &= 37/2 \approx 19
 \end{aligned}$$

$$(b, e) = 19$$



Now consider

(i) $a, b, c, d, (e, a)$

(ii) $a, b, c, e (d, a)$

and

(i) $a, b, d, e, (c, a)$

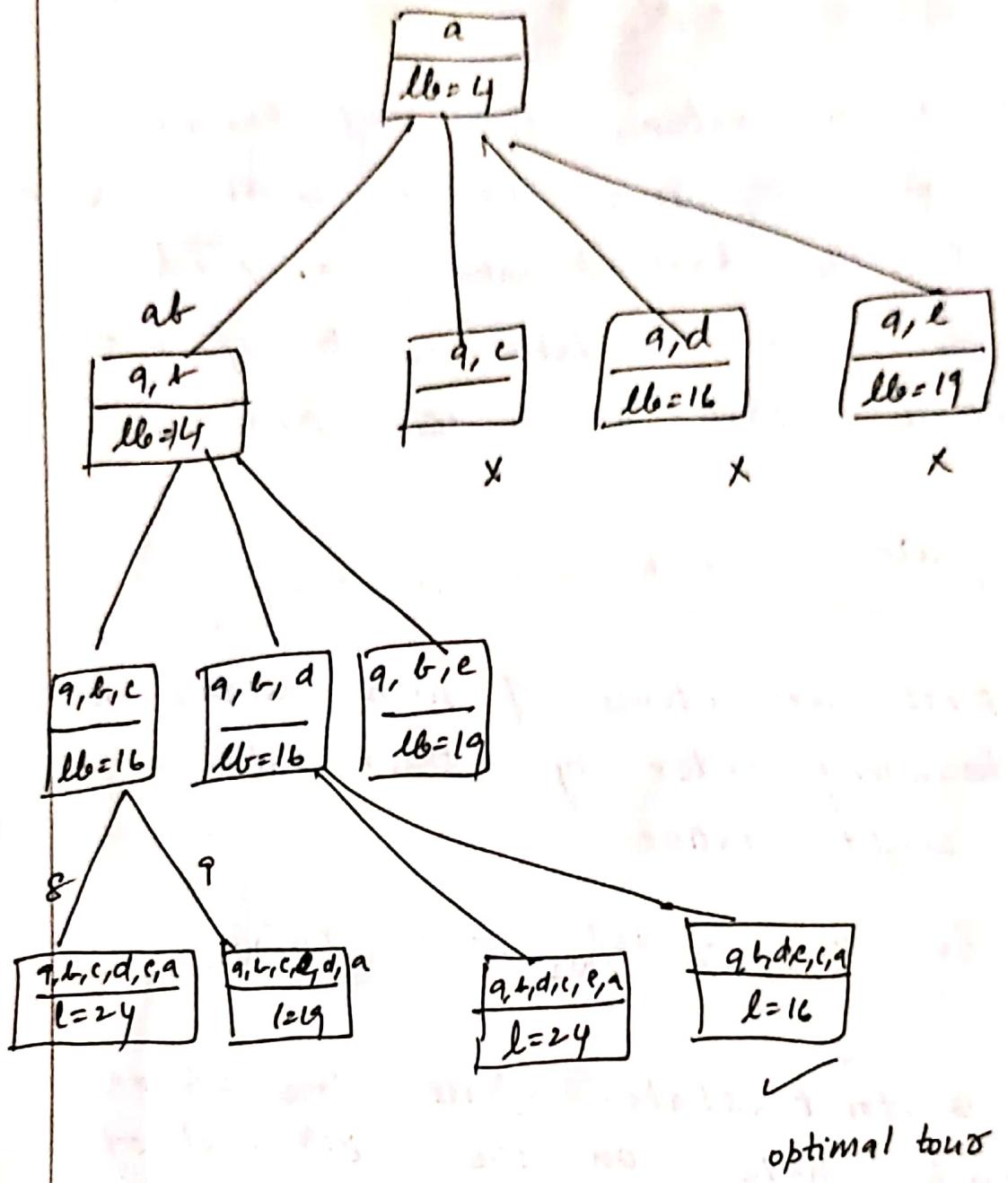
(ii) $a, b, d, e, (c, a)$

for a, b, c, d, e, a $\ell = 3+6+4+3+8 = 24$

a, b, c, c, d, a $\ell = 3+6+2+3+5 = 19$

a, b, d, c, e, a $\ell = 3+7+4+2+8 = 24$

a, b, d, e, c, a $\ell = 3+7+3+2+1 = 16$ ✓



optimal Tour = $a \rightarrow b \rightarrow d \rightarrow e \rightarrow c \rightarrow a$

optimal cost = 16

0/1 Knapsack problem:

Given n instances/items of known weights w_i and values v_i for $i=1, 2, \dots, n$ and the knapsack capacity W , find the most valuable subset of the items that fit in the knapsack.

$$UB = V + (W - w) \left(\frac{v_{i+1}}{w_{i+1}} \right)$$

- first sort items of given instance in descending order by their value to weight ratios:

$$\text{i.e } \frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$$

- construct a state-space tree where each node on the i^{th} level of this tree $0 \leq i \leq n$ represents all the subsets of n items.

- Obtain the optimal solution for the following instance of knapsack problem using branch & bound technique. Draw the state space tree:

item	weight	value	value/weight
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

$$W = 10$$

$$ub = v + (W - w) \left(\frac{v_{i+1}}{w_{i+1}} \right)$$

- Initial state

$$ub = 0 + (10 - 0) 10 = 100$$

Consider $i = 1$

with item = 1

$$v = 40, w = 4$$

$$ub = 40 + [10 - 4] \left[\frac{42}{7} \right]$$

$$= 40 + 6 * 6$$

$$ub = 76$$

w/o item = 1

$$v = 0, w = 0$$

$$ub = 0 + 10 \left[\frac{42}{7} \right]$$

$$= 0 + 60$$

$$= \underline{\underline{60}}$$

Greater, so include item 1

Consider $c=2$, with item 1

with item 2

$$V = 42 + 40 = 82$$

$$w = 7 + 4 = 11$$

$$W = 10$$

Not feasible

$$w > W$$

w/o item 2

$$V = 40, w = 4$$

$$ub = 40 + (10-4) \left(\frac{v_3}{w_3} \right)$$

$$= 40 + 6 * 5$$

$$= 40 + 30 = 70$$

$$\underline{ub = 70}$$

Consider $c=3$, w/o item 2

with item = 3

$$V = 25 + 40 = 65$$

$$w = 5 + 4 = 9$$

$$W = 10$$

$$ub = 65 + 1 * \left(\frac{v_4}{w_4} \right)$$

$$= 65 + 4 = 69$$

w/o item 3

$$V = 40$$

$$w = 4$$

$$ub = 40 + (10-4) \left(\frac{v_4}{w_4} \right)$$

$$= 40 + 6 * 4$$

$$= 40 + 24 = 64$$

consider $i = 4$

(i) with item 4

$$V = 12 + 65 = 77$$

$$W = 9 + 3 = 12$$

$$w > W$$

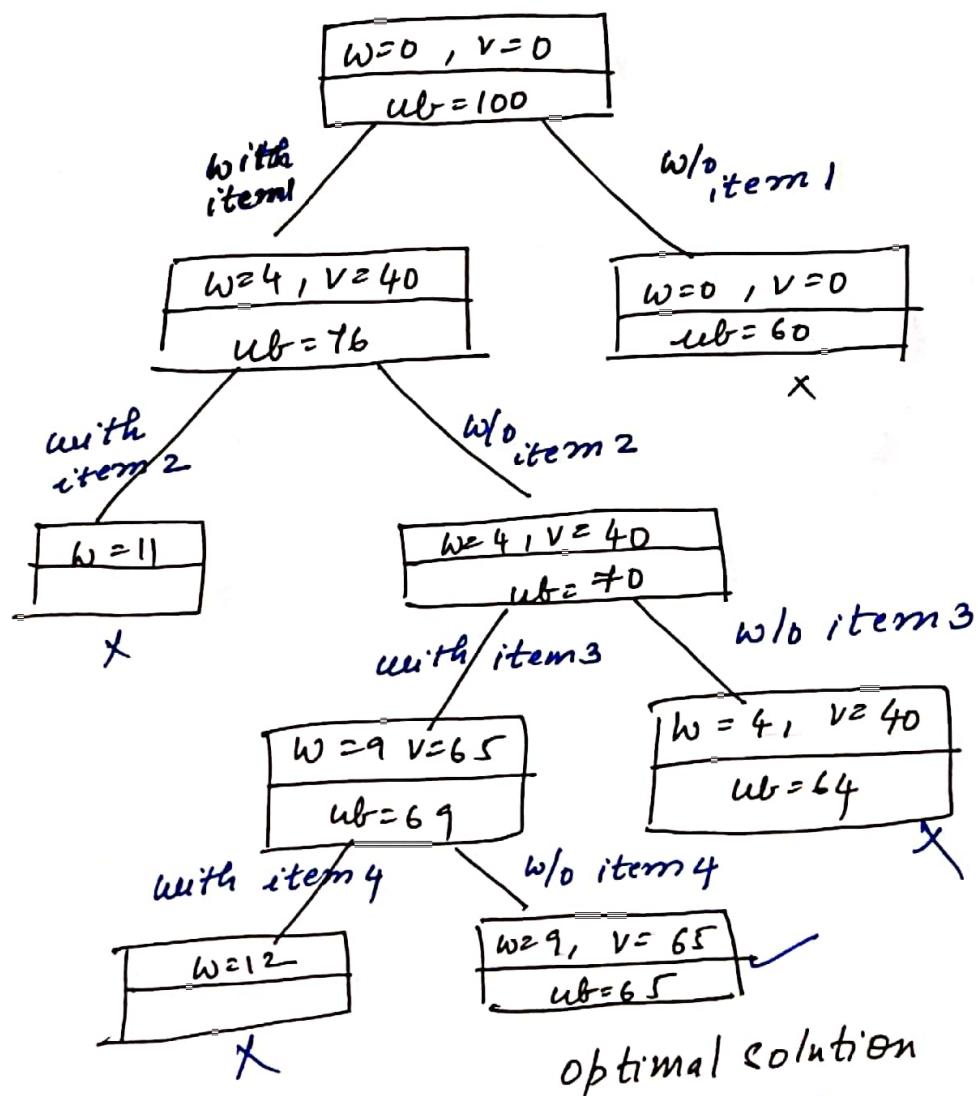
\therefore Not feasible

(ii) w/o item 4

$$V = 65, W = 4 + 5 = 9$$

$$w = 65$$

\therefore state space tree is



\therefore Solution vector = {1, 3}, optimal solution = 65

* with the help of state space tree, solve
 the following instance of knapsack problem
 by branch & bound algorithm

Item	weight	value
1	10	100
2	7	63
3	8	56
4	4	12
		$w = 16$

LC Branch and Bound solution:

Live Node: Is a node that has been generated but whose children have not yet been generated.

E-Node: Is a live node whose children are currently being explored

Dead-Node: Is a generated node, that is not to be expanded/explored any further

Solve the following instance of knapsack problem using LC Branch & Bound solution method. Draw the state space tree

Item	Profit	Weight
i	P_i	w_i
1	10	2
2	10	4
3	12	6
4	18	9

Max capacity = 15

Steps to be followed for LCBB soln are:-

1. Draw state space tree
2. Compute $\hat{c}(\cdot)$ and $u(\cdot)$ for each node

$\hat{c}(\alpha) \rightarrow$ Approximation cost used for computing the least cost $c(\alpha)$

$u(\alpha) \rightarrow$ denotes Upper Bound.

3. If $\hat{C}(x) > \text{upper Bound}$, Kill Node x
4. otherwise minimum cost $\hat{C}(x)$ becomes E-node. Generate Children for E-node.
5. Repeat step 3 & 4 until all the nodes get covered.
6. Minimum cost $\hat{C}(x)$ becomes the answer node. Trace the path in backward direction from x to root for solution subset.

SOL \rightarrow First calculate least cost & upper bound for root node. Root node ie node 1

$$u(1) = - \sum p_i = -(10+10+12) = -32$$

Item 4 is not select, as it exceed capacity.

$$\hat{C}(1) = u(1) - \left[\frac{m - \text{current total wt}}{\text{Actual weight of remaining object}} \right] \times \left[\begin{array}{l} \text{actual profit of} \\ \text{remaining object} \end{array} \right]$$

$$= -32 - \left[\frac{15 - (2+4+6)}{9} \right] + 18$$

$$= -32 - \frac{3}{9} \times 18 = \underline{\underline{-38}}$$

with item 1

$$u(2) = -\sum p_i = -(10+10+12)$$

$$v(2) = -32$$

$$c^*(2) = -38$$

w/o item 1

$$u(3) = -\sum p_i$$

$$v(3) = -(10+12) = -22$$

$$c^*(3) = -22 - \left\lceil \frac{15-10}{7} \right\rceil + 18$$

$$= -22 - \frac{5}{7} + 18$$

$$c^*(3) = \underline{\underline{-32}}$$

$$\underline{u(2) > u(3)}$$

∴ Branching takes place at node 2, i.e. with item 1 as it has least upper bound.

with item 2

$$v(4) = -\sum p_i$$

$$= -(10+10+12)$$

$$v(4) = -32$$

$$c^*(4) = -38$$

w/o item 2

$$v(5) = -\sum p_i$$

$$= -(10+12) = -22$$

$$c^*(5) = -22 - \left\lceil \frac{15-8}{7} \right\rceil + 18$$

$$= -22 - \frac{7}{7} + 18$$

$$= -22 - 14 = -36$$

$$\underline{v(4) > v(5)}$$

∴ Branching take place with item 2

node 6
with item 3

$$U(6) = - \sum P_i$$

$$U(6) = -32$$

$$C^*(6) = -32 - \frac{(15-12)}{9} + 18$$

$$= -32 - 3(2)$$

$$C^*(6) = -38$$

node 7
w/o item 3

$$U(7) = - \sum P_i$$

$$U(7) = -38$$

$$C^*(7) = -38 - \frac{(15-15)}{c} + 12$$

$$= -38 - 0$$

$$\underline{C^*(7) = -38}$$

$$U(6) < U(7)$$

So branching take place at $U(7)$, w/o item 3

with item 4

$$U(8) = - \sum P_i = -(10+10+18)$$

$$= -38$$

$$C^*(8) = -38 - \frac{(15-15)}{6} + 12$$

$$\underline{C^*(8) = -38}$$

$$U(9) = - \sum P_i$$

$$U(9) = -(30+10) = \underline{-20}$$

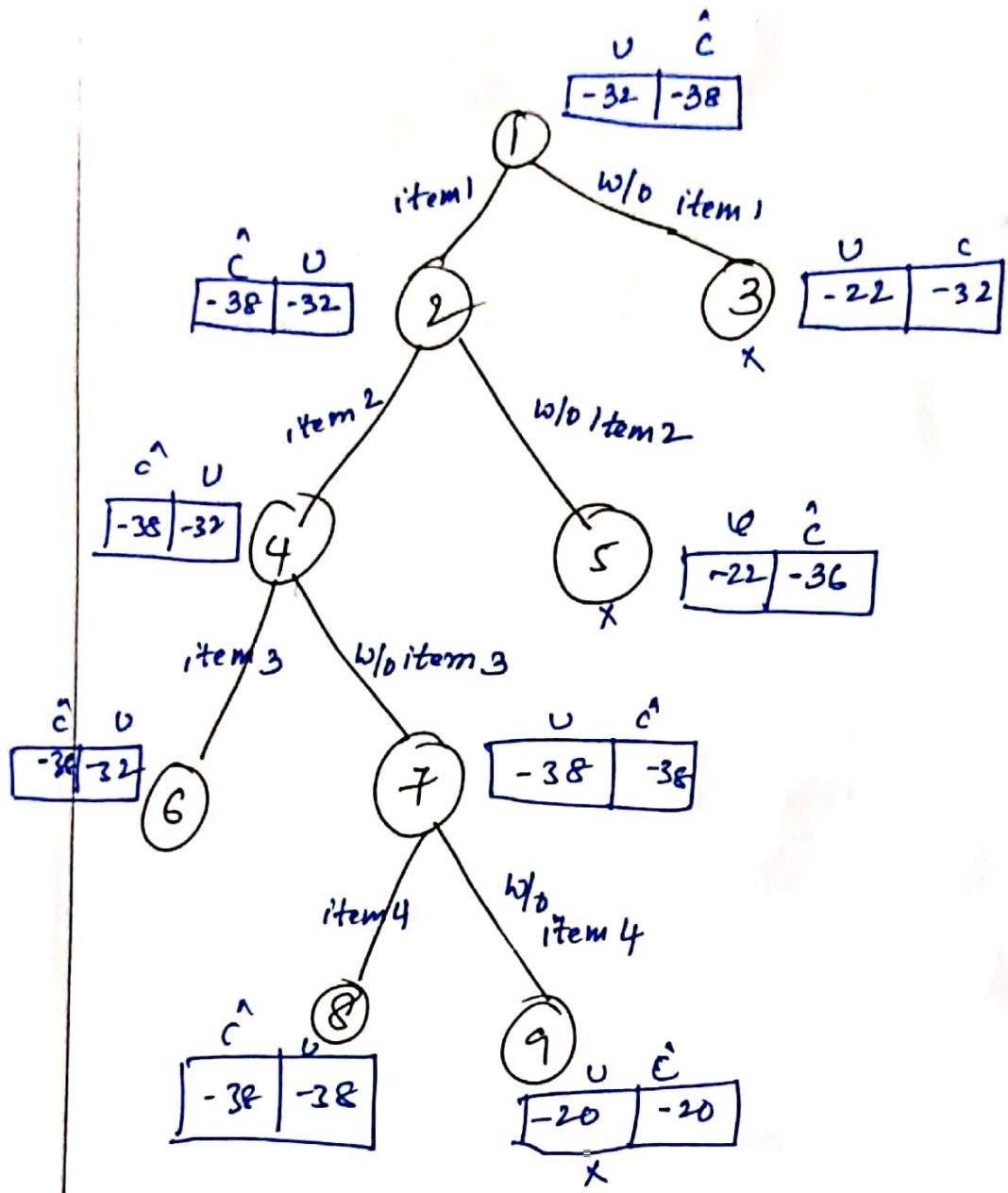
$$C^*(9) = -20 - \frac{(15-6)}{10} + 0$$

$$\underline{\underline{= -20}}$$

$$\underline{U(8) > U(9)}$$

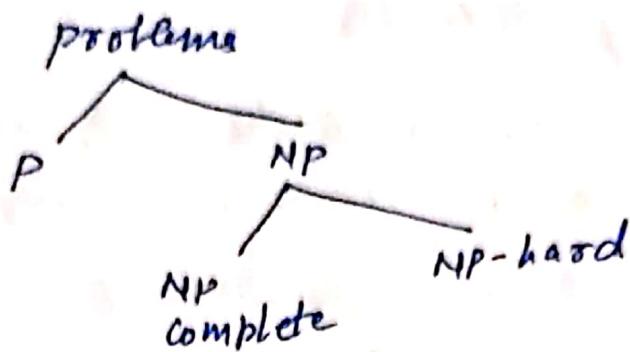
solution vector = {1, 1, 0, 1}

Max profit = 38



P, NP, NP complete & NP hard problems

- The problems can be classified as 2 groups:



P class: set of all decision problems solvable by deterministic algorithms in polynomial time:
eg: searching, sorting.

NP Class: set of all decision problems solvable by non deterministic algorithms in polynomial time.
eg: TSP, Graph coloring, knapsack problem etc

$$P \subseteq NP$$

NP-Complete:
A decision problem D is said to be NP-complete if

- (i) It belongs to class NP
- (ii) Every problem in NP is polynomially reducible to D.

NP-hard:

A problem L is NP hard iff satisfiability reduces to L . A problem is NP complete iff L is NP hard & $\underline{L \in NP}$.

Polynomially Reducible:

A decision problem D_1 is polynomially reducible to a decision problem D_2 if there exists a function t that transforms instances of D_1 to instances of D_2 such that:

(i) t maps all yes instances of D_1 to yes instances of D_2 & all no instances of D_1 to no instances of D_2

(ii) t is computable by a polynomial time algo.

If problem D_1 is polynomially reducible to some problem D_2 that can be solved in polynomial time, then problem D_1 can also be solved in polynomial time.



DYNAMIC PROGRAMMING

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

DYNAMIC PROGRAMMING

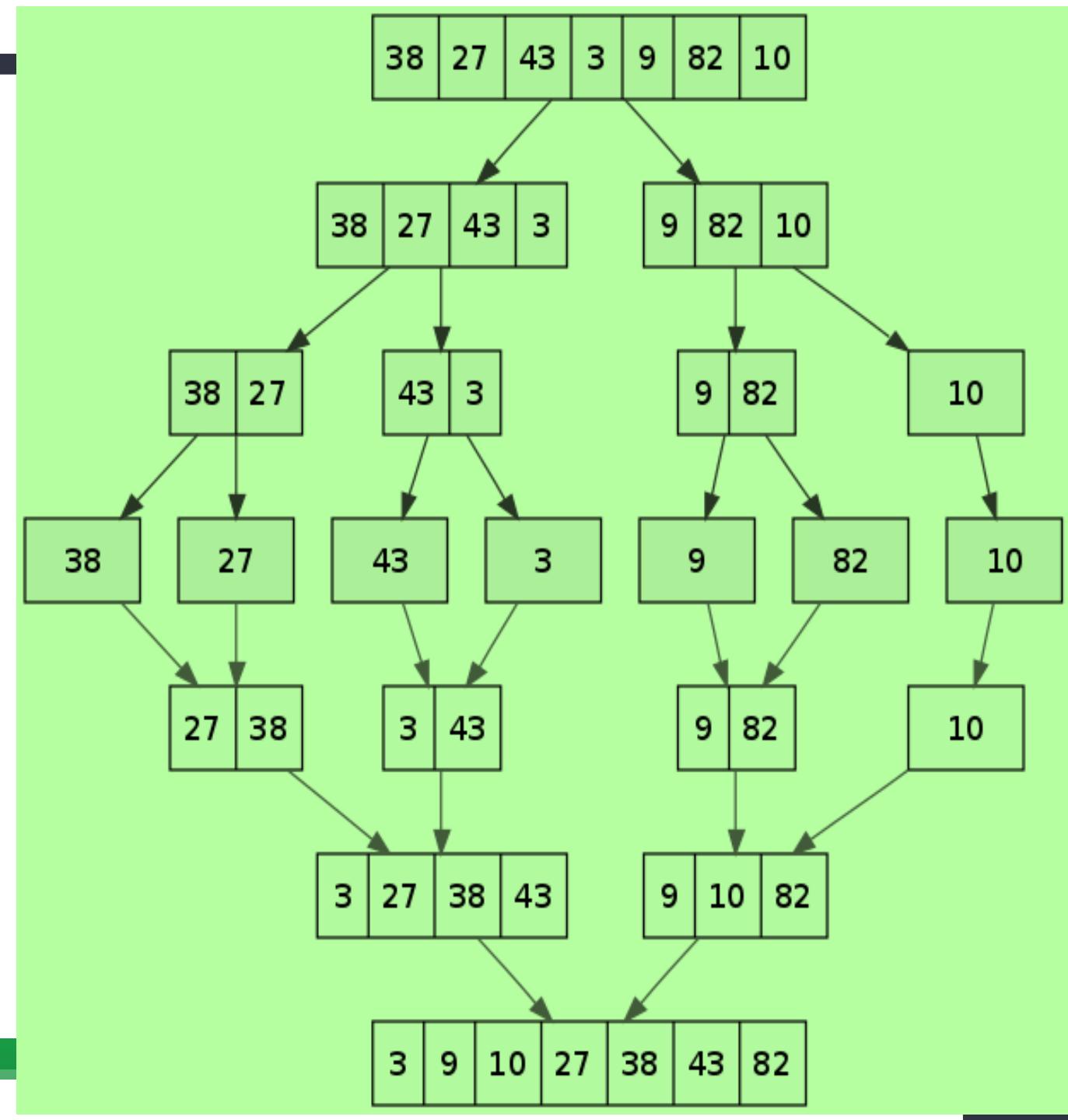
- It was invented by a prominent U.S. mathematician, Richard Bellman
- Thus, the word “programming” in the name of this technique stands for “planning” / “tabulation” and does not refer to computer programming.
- Dynamic programming is a technique for solving problems with overlapping subproblems
 - Typically, these subproblems arise from a recurrence relating a given problem's solution **to** solutions of its smaller subproblems
- Rather than solving overlapping subproblems again and again,
 - dynamic programming suggests solving each of the smaller subproblems only once , and
 - recording the results in a table from which a solution to the original problem can then be obtained.

DYNAMIC PROGRAMMING

- A widely used technique to solve optimization problems
- Optimal=maximization/minimization
- The main difference between divide and conquer and dynamic programming is that
 - the divide and conquer combines the solutions of the sub-problems to obtain the solution of the main problem
 - while dynamic programming uses the result of the sub-problems to find the optimum solution of the main problem.
 - A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table

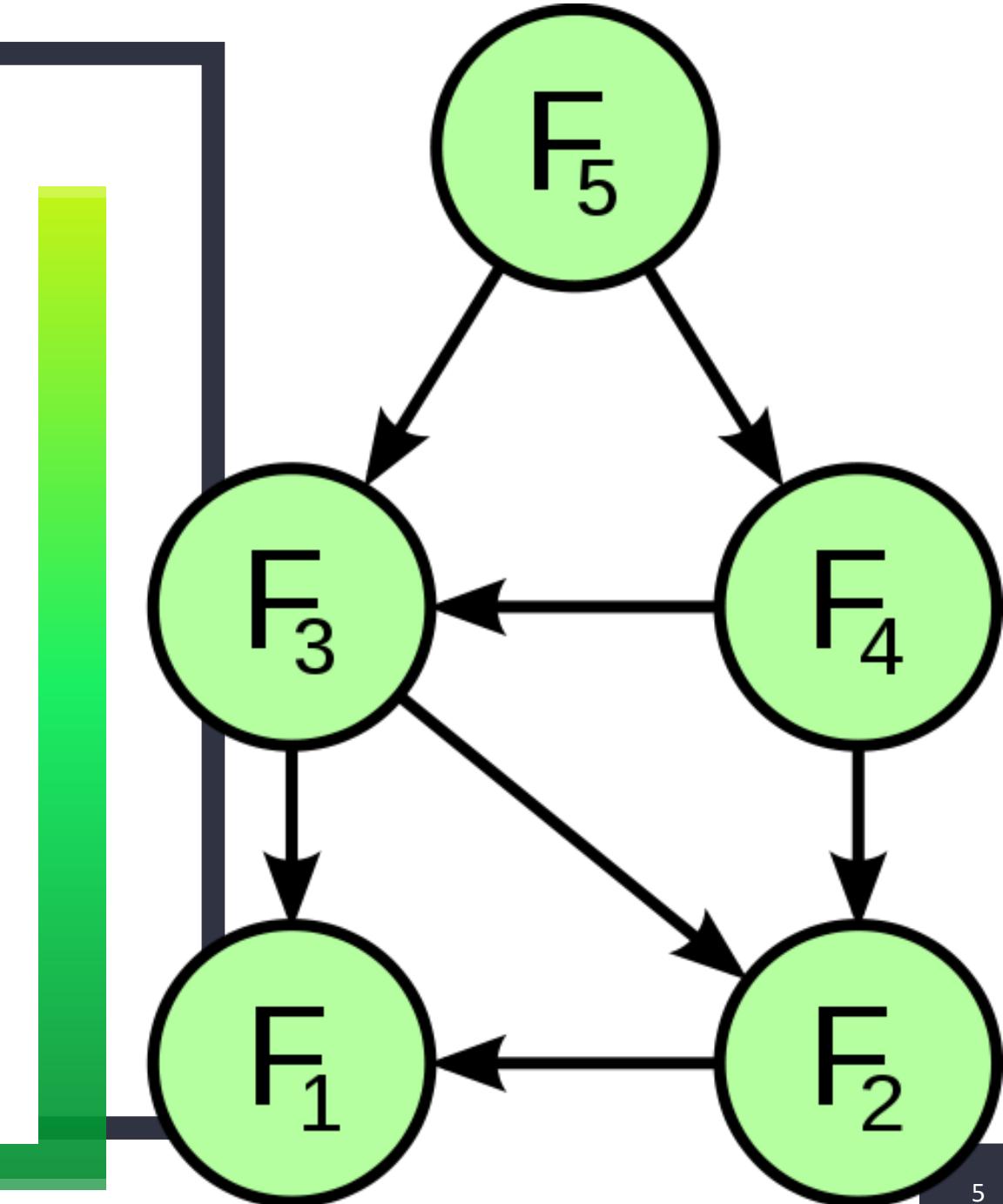
WHAT IS DIVIDE AND CONQUER ?

- Divide and conquer divides the main problem into small subproblems.
- The subproblems are divided again and again. At one point, there will be a stage where we cannot divide the subproblems further.
- Then, we can solve each subproblem independently.
- Finally, we can combine the solutions of all subproblems to get the solution to the main problem.



WHAT IS DYNAMIC PROGRAMMING

- Dynamic programming divides the main problem into smaller subproblems, but it does not solve the subproblems independently.
- It stores the results of the subproblems to use when solving similar subproblems. Storing the results of subproblems is called memorization.
- Before solving the current subproblem, it checks the results of the previous subproblems.
- Finally, it checks the results of all subproblems to find the best solution or the optimal solution. This method is effective as it does not compute the answers again and again. Usually, dynamic programming is used for optimization.



DIVIDE AND CONQUER

VERSUS

DYNAMIC PROGRAMMING

DIVIDE AND CONQUER

An algorithm that recursively breaks down a problem into two or more sub-problems of the same or related type until it becomes simple enough to be solved directly

Subproblems are independent of each other

Recursive

More time-consuming as it solves each subproblem independently

Less efficient

Used by merge sort, quicksort, and binary search

DYNAMIC PROGRAMMING

An algorithm that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property

Subproblems are interdependent

Non-recursive

Less time-consuming as it uses the answers of the previous subproblems

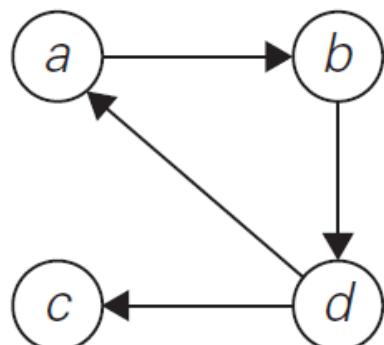
More efficient

Used by matrix chain multiplication, optimal binary search tree

DYNAMIC PROGRAMMING

Warshall's Algorithm

- An algorithm for computing the transitive closure of a directed graph
- The *transitive closure* of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the i^{th} row and the j^{th} column is **1** if there exists a nontrivial path (i.e., directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is **0**.



$$A = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \left[\begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{matrix} \right] \end{matrix}$$

$$T = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \left[\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \right] \end{matrix}$$

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Transitive closure of the digraph, would allow us to determine in constant time whether the j^{th} vertex is reachable from the i^{th} vertex.
- We can generate the transitive closure of a digraph with the help of depth first search or breadth-first search.?
- Doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.
- But this method traverses the same digraph several times.

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Transitive closure of the digraph, would allow us to determine in constant time whether the j^{th} vertex is reachable from the i^{th} vertex.
- We can generate the transitive closure of a digraph with the help of depth first search or breadth-first search.?
- Doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.
- But this method traverses the same digraph several times.

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

- Each of these matrices provides certain information about directed paths in the digraph.
- The element $r^{(k)}_{ij}$ in the i^{th} row and j^{th} column of matrix $R^{(k)}$ ($i, j = 1, 2, \dots, n, k = 0, 1, \dots, n$) is equal to 1 if and only if there exists a directed path of a positive length from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k

DYNAMIC PROGRAMMING

Warshall's Algorithm

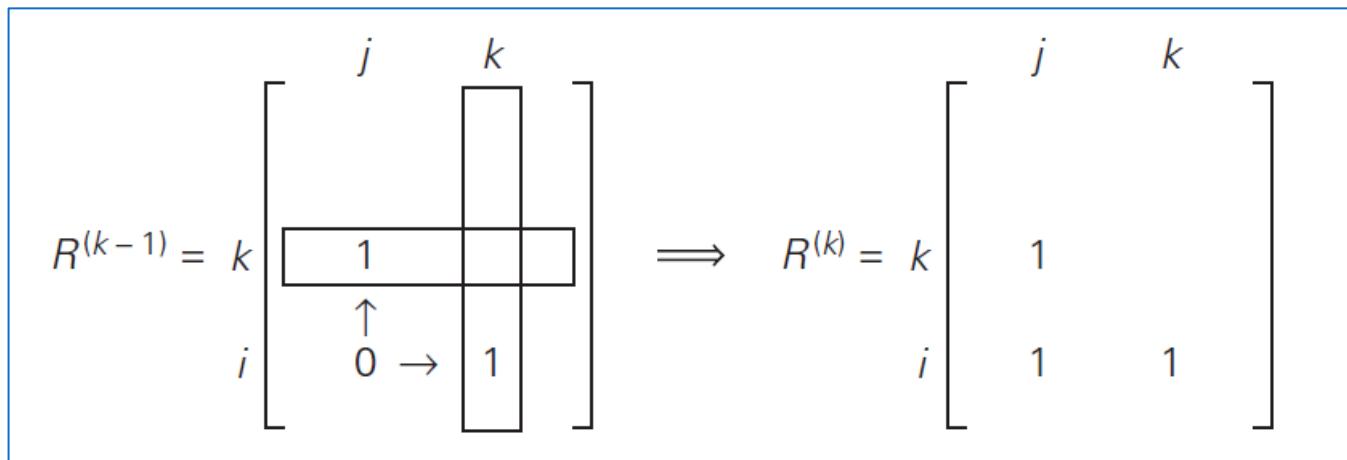
- Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph.
- $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; thus, with more freedom, so to speak, it may contain more 1's than $R^{(0)}$
- In general, each subsequent matrix in series has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more 1's.
- The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

DYNAMIC PROGRAMMING

Warshall's Algorithm

- The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series

$$r_{ij}^k = r_{ij}^{k-1} \text{ or } (r_{ik}^{k-1} \text{ and } r_{kj}^{k-1})$$



DYNAMIC PROGRAMMING

Warshall's Algorithm

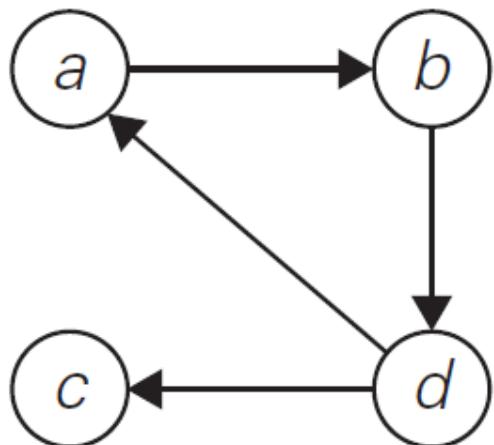
$$r_{ij}^k = r_{ij}^{k-1} \text{ or } (r_{ik}^{k-1} \text{ and } r_{kj}^{k-1})$$

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Obtain the transitive closure for the following digraph



$$R^{(0)} = \begin{bmatrix} & a & b & c & d \\ a & \boxed{0} & 1 & 0 & 0 \\ b & 0 & \boxed{0} & 0 & 1 \\ c & 0 & 0 & \boxed{0} & 0 \\ d & 1 & 0 & 1 & 0 \end{bmatrix}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

DYNAMIC PROGRAMMING

Warshall's Algorithm

$$R^{(0)} = \begin{bmatrix} & a & b & c & d \\ a & \boxed{0} & 1 & 0 & 0 \\ b & 0 & \boxed{0} & 0 & 1 \\ c & 0 & 0 & \boxed{0} & 0 \\ d & 1 & 0 & 1 & 0 \end{bmatrix}$$
$$R^{(1)} = \begin{bmatrix} & a & b & c & d \\ a & \boxed{0} & 1 & 0 & 0 \\ b & 0 & \boxed{0} & 0 & 1 \\ c & 0 & 0 & \boxed{0} & 0 \\ d & 1 & \boxed{1} & 1 & 0 \end{bmatrix}$$
$$R^{(2)} = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & \boxed{0} & 1 \\ b & 0 & 0 & \boxed{0} & 1 \\ c & 0 & 0 & \boxed{0} & 0 \\ d & 1 & 1 & \boxed{1} & 1 \end{bmatrix}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex a (note a new path from d to b); boxed row and column are used for getting $R^{(2)}$.

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., a and b (note two new paths); boxed row and column are used for getting $R^{(3)}$.

DYNAMIC PROGRAMMING

Warshall's Algorithm

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., a , b , and c (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., a , b , c , and d (note five new paths).

DYNAMIC PROGRAMMING

Warshall's Algorithm

ALGORITHM *Warshall($A[1..n, 1..n]$)*

//Implements Warshall's algorithm for computing the transitive closure

//Input: The adjacency matrix A of a digraph with n vertices

//Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$

for $k \leftarrow 1$ **to** n **do**

for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** n **do**

$R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** ($R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j]$)

return $R^{(n)}$

Time Complexity?

$\theta(n^3)$

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix:

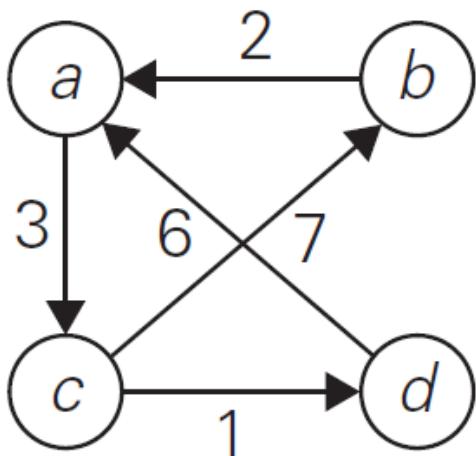
$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- Given a weighted connected graph (undirected or directed), the *all-pairs shortest paths problem* asks to find the the distances—i.e., the lengths of the shortest paths— from each vertex to all other vertices.
- Major applications**
 - communications,
 - transportation networks,
 - operations research
 - precomputing distances for motion planning in computer games. Etc.

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***:
 - the element d^{ij} in the i^{th} row and the j^{th} column of this matrix indicates the length of the shortest path from the i^{th} vertex to the j^{th} vertex.



$$W = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \left[\begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{matrix} \right] \end{matrix}$$

$$D = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \left[\begin{matrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{matrix} \right] \end{matrix}$$

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- The distance matrix can be generated by an algorithm that is very similar to **Warshall's algorithm**,
- It is **Floyd's algorithm** named after Robert W Floyd
- It is applicable to both undirected and directed weighted graphs provided they do not contain a cycle of a negative length
- Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

$$D^{(0)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}$$

- Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

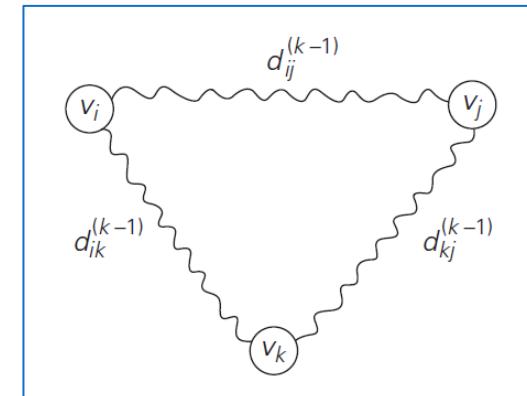
- The element $d^{(k)}_{ij}$ in the i^{th} row and the j^{th} column of matrix $D(k)$ ($i, j = 1, 2, \dots, n, k = 0, 1, \dots, n$) is equal to the length of the shortest path among all paths from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k .
- The series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence,
 - $D^{(0)}$ is simply the weight matrix of the graph.
- The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate.
- We can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$ in series

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- The recurrence relation for the Floyd's algorithm is given by

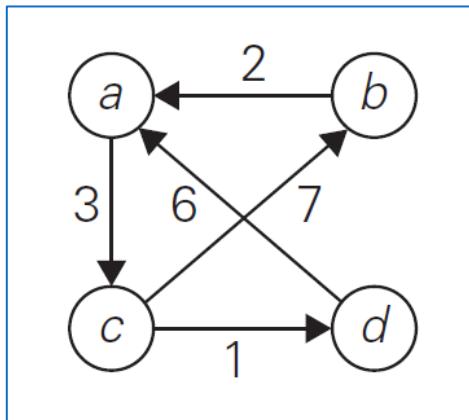
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}.$$

- the element in row i and column j of the current distance matrix $D^{(k-1)}$ is replaced by the sum of the elements in the same row i and the column k and in the same column j and the row k if and only if the latter sum is smaller than its current value.



FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- Obtain the shortest paths from each vertex to all other vertices in the graph shown



$$D^{(0)} = \begin{bmatrix} & a & b & c & d \\ a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{bmatrix}$$

Lengths of the shortest paths
with no intermediate vertices
($D^{(0)}$ is simply the weight matrix).

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

$$D^{(0)} = \begin{array}{c} \begin{array}{cccc} & a & b & c & d \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \left[\begin{array}{ccccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{array} \right] \end{array} \end{array}$$
$$D^{(1)} = \begin{array}{c} \begin{array}{cccc} & a & b & c & d \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \left[\begin{array}{ccccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \boxed{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{array} \right] \end{array} \end{array}$$
$$D^{(2)} = \begin{array}{c} \begin{array}{cccc} & a & b & c & d \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \left[\begin{array}{ccccc} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \boxed{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{array} \right] \end{array} \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just a (note two new shortest paths from b to c and from d to c).

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and b (note a new shortest path from c to a)

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

$$D^{(3)} = \begin{array}{c} \begin{array}{cccc} a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array} \end{array}$$

Lengths of the shortest paths
with intermediate vertices numbered
not higher than 3, i.e., a , b , and c
(note four new shortest paths from a to b ,
from a to d , from b to d , and from d to b).

$$D^{(4)} = \begin{array}{c} \begin{array}{cccc} a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array} \end{array}$$

Lengths of the shortest paths
with intermediate vertices numbered
not higher than 4, i.e., a , b , c , and d
(note a new shortest path from c to a).

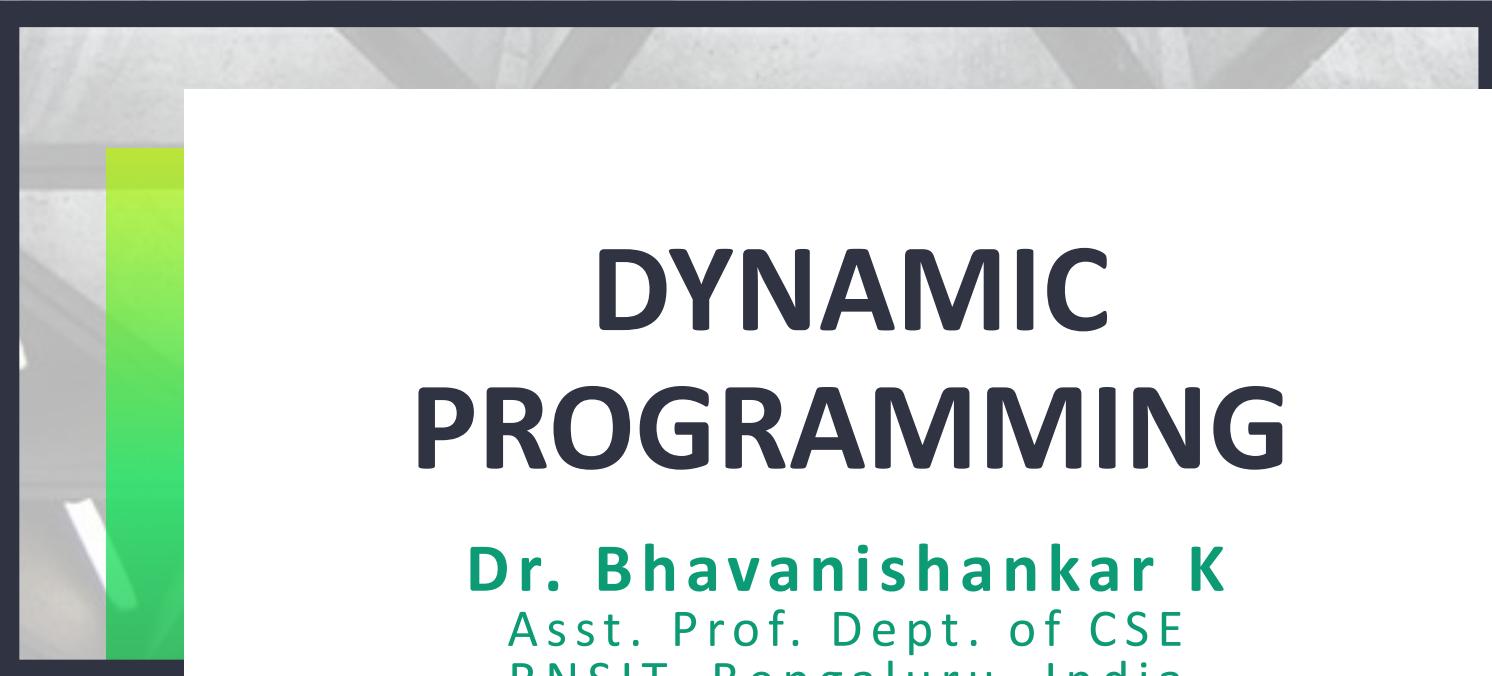
FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- Solve the all-pairs shortest-path problem for the digraph with the following weight matrix:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

```
ALGORITHM Floyd(W[1..n, 1..n])
    //Implements Floyd's algorithm for the all-pairs shortest-paths problem
    //Input: The weight matrix  $W$  of a graph with no negative-length cycle
    //Output: The distance matrix of the shortest paths' lengths
     $D \leftarrow W$  //is not necessary if  $W$  can be overwritten
    for  $k \leftarrow 1$  to  $n$  do
        for  $i \leftarrow 1$  to  $n$  do Time Complexity?  $\theta(n^3)$ 
            for  $j \leftarrow 1$  to  $n$  do
                 $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$ 
    return  $D$ 
```



DYNAMIC PROGRAMMING

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

0/1 KNAPSACK PROBLEM

Problem Statement :

Given n items of

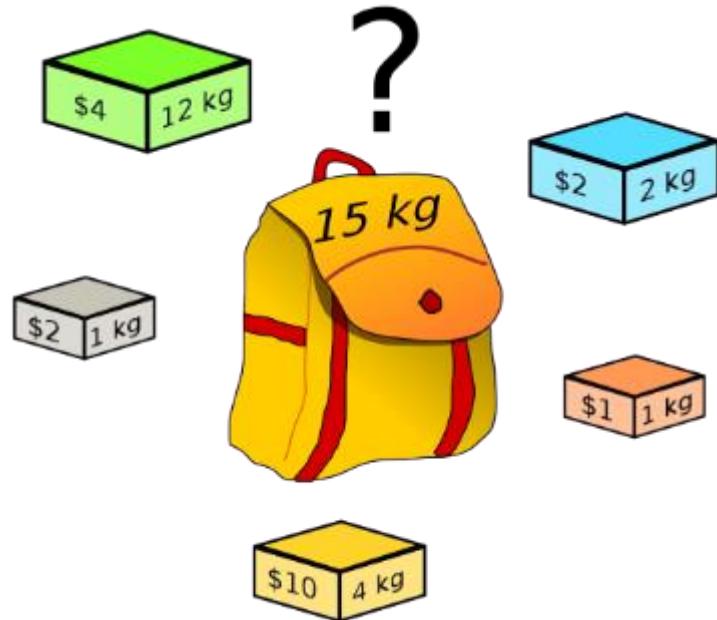
integer weights: $w_1 \ w_2 \dots w_n$
values: $v_1 \ v_2 \dots v_n$

and a knapsack of integer capacity W ,

**Find most valuable subset of the items
that fit into the knapsack**

Assumptions: Weights and capacity are positive integers,
values need not have to be integers

Items are indivisible, you
either select item or not !!!!!



0/1 KNAPSACK PROBLEM

- To design a dynamic programming algorithm, recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub instances is needed
- Consider such a smaller sub instance defined by first i items, $1 \leq i \leq n$, with weights w_1, \dots, w_i , values v_1, \dots, v_i , and knapsack capacity j , $1 \leq j \leq W$.
- Let $F(i, j)$ be the value of an optimal solution to this instance,
 - i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j .

0/1 KNAPSACK PROBLEM

- We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the i^{th} item and those that do.
- Among the subsets that **do not include** the i^{th} item, the value of an optimal subset is, by definition, $F(i - 1, j)$.
- Among the subsets that **do include** the i^{th} item (hence, $j - wi \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fits into the knapsack of capacity $j - wi$.
 - The value of such an optimal subset is $vi + F(i - 1, j - wi)$.

0/1 KNAPSACK PROBLEM

- Hence, the value of an optimal solution among all feasible subsets of the first i items is the maximum of these two values.
- If the i^{th} item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i - 1$ items.
- With these observations we can arrive at following recurrence

$$F[i,j] = \begin{cases} F[i-1,j] & \text{if } j - w_i < 0 \\ \max \{F[i-1,j], v_i + F[i-1,j - w_i]\} & \text{if } j - w_i \geq 0 \end{cases}$$

- Following are the initial conditions

$$F(0, j) = 0 \text{ for } j \geq 0 \text{ and } F(i, 0) = 0 \text{ for } i \geq 0.$$

0/1 KNAPSACK PROBLEM

- The goal is to find $F(n, W)$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W , and an optimal subset itself.

$F =$

	0	$j - w_i$	j	W
0	0	0	0	0
$i-1$	0	$F(i-1, j - w_i)$	$F(i-1, j)$	
w_i, v_i	0		$F(i, j)$	
n	0			goal

0/1 KNAPSACK PROBLEM

- Solve the following instance of knapsack problem to obtain the optimal solution using dynamic programming approach.

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

knapsack capacity = 5 kg,

0/1 KNAPSACK PROBLEM

Solution

- Draw a table say with $(n+1) = 4 + 1 = 5$ number of rows and $(w+1) = 5 + 1 = 6$ number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with 0. (initial condition)

F=

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

- Start filling this table using the recurrence relation

0/1 KNAPSACK PROBLEM

- $F(1, 1) = ?$
- $i = 1, j = 1, v_1 = 3, w_1 = 2$
- Find whether $j - wi < 0$? YES
- Then as per the recurrence $F(1, 1) = F(1 - 1, 1) = F(0, 1) = 0$
- $F(1, 2) = ?$
- $i = 1, j = 2, v_1 = 3, w_1 = 2$
- Find whether $j - wi < 0$? NO
- $F(1, 2) = \max\{F(0, 2), 3 + F(0, 0)\} = 3$

0/1 KNAPSACK PROBLEM

- $F(1, 3) = ?$
- $i = 1, j = 3, v_1 = 3, w_1 = 2$
- Find whether $j - wi < 0$? **NO**
- $F(1, 3) = \max\{F(0,3), 3 + F(0,1)\} = 3$
- $F(1, 4) = ?$
- $i = 1, j = 4, v_1 = 3, w_1 = 2$
- Find whether $j - wi < 0$? **NO**
- $F(1, 4) = \max\{F(0,4), 3 + F(0,2)\} = 3$

0/1 KNAPSACK PROBLEM

- $F(1, 5) = ?$
- $i = 1, j = 5, v_1 = 3, w_1 = 2$
- Find whether $j - wi < 0$? **NO**
- $F(1, 5) = \max\{F(0,5), 3 + F(0,3)\} = 3$
- $F(2, 1) = ?$
- $i = 2, j = 1, v_2 = 4, w_2 = 3$
- Find whether $j - wi < 0$? **YES**
- $F(2, 1) = F(1, 1) = 0$

0/1 KNAPSACK PROBLEM

- $F(2, 2,) = ?$
- $i = 2, j = 2, v_2 = 4, w_2 = 3$
- Find whether $j - wi < 0?$ YES
- $F(2, 2) = F(1, 2) = 3$
- $F(2, 3) = ?$
- $F(2, 3) = ?$ and so on till $F(4, 5)$

0/1 KNAPSACK PROBLEM

- The final table

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

0/1 KNAPSACK PROBLEM

- Solve the following instance of knapsack problem to obtain the optimal solution using dynamic programming approach.

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

knapsack capacity = 5 kg,

0/1 KNAPSACK PROBLEM

- Solution

		capacity j					
		0	1	2	3	4	5
i	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

Optimal solution



DYNAMIC PROGRAMMING

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

TRAVELING SALESMAN PROBLEM

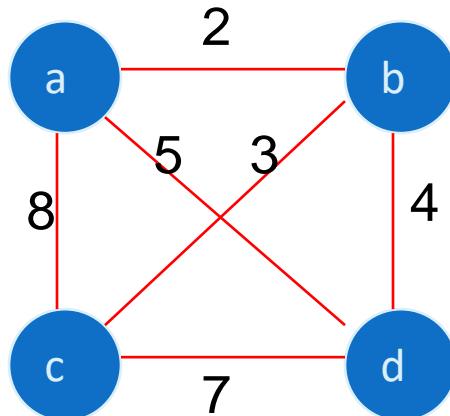
- 0/1 knapsack problem was an example for subset selection problem.
- Talking about permutation problems, they are much more harder to solve compared to subset selection problem
- as there are $n!$ different permutations of n objects where as there are only 2^n different subsets of n objects ($n! > 2^n$)
- Travelling salesman problem is a perfect example for permutation problem

TRAVELING SALESMAN PROBLEM

- Given n vertices the salesman wishes to make a ***tour***, visiting each city exactly once and finishing at the city he starts from
- Given a graph $G = (V, E)$ with n vertices and the **source** vertex, the salesman wishes to make a ***tour***, starting from the **source** vertex, visiting each vertex **exactly once** and **finishing** at the vertex he started from (**source**).
- Principle of Optimality ?
- Different techniques to solve this problem
 - Brute Force approach
 - Branch and bound
 - Dynamic programming

TRAVELING SALESMAN PROBLEM

- Brute Force approach



Let source be a

Tour

a → b → c → d → a

a → b → d → c → a

a → c → b → d → a

a → c → d → b → a

a → d → b → c → a

a → d → c → b → a

Cost

$2+3+7+5 = 17$ Best Tour

$2+4+7+8 = 21$

$8+3+4+5 = 20$

$8+7+4+2 = 21$

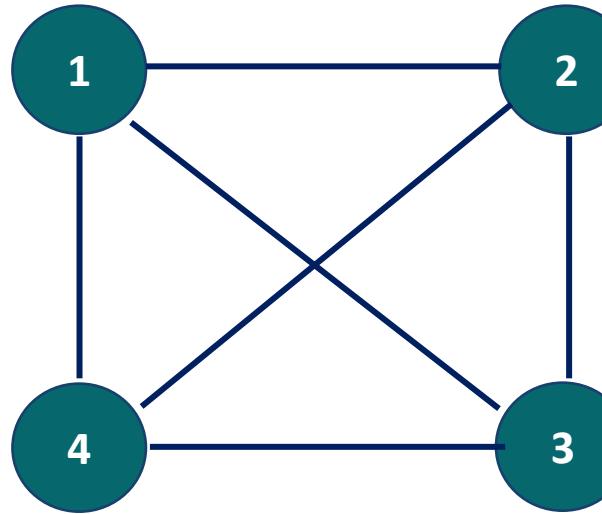
$5+4+3+8 = 20$

$5+7+3+2 = 17$ Best Tour

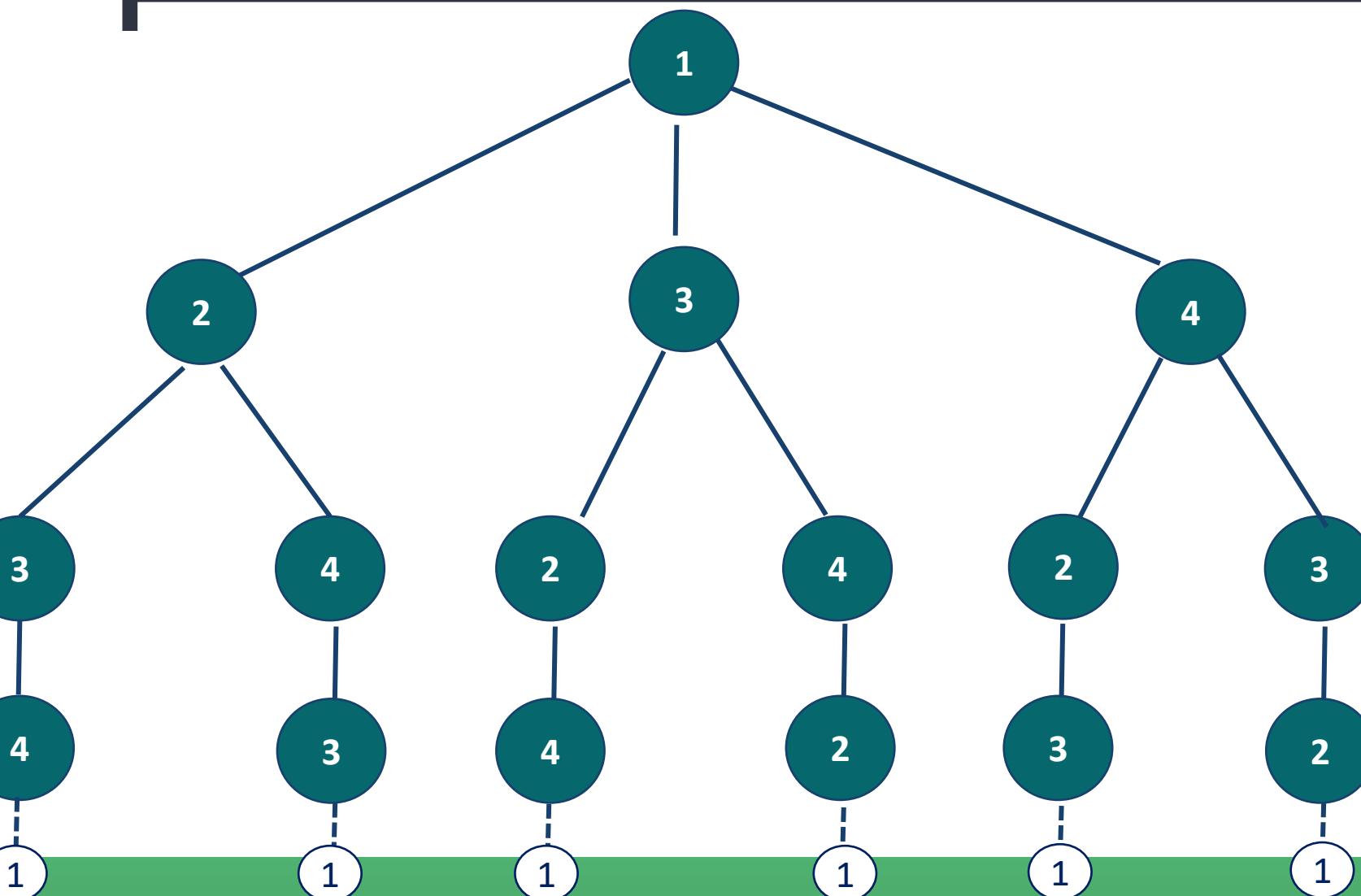
TRAVELING SALESMAN PROBLEM

•

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

| 0 | 20 | 10 | 12 |


TRAVELING SALESMAN PROBLEM



TRAVELING SALESMAN PROBLEM

- Let $\mathbf{G} = (V, E)$ 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 $c_{ij} > \infty$ if $(i, j) \notin E$.
- Let $|V|=n$ and assume $n>1$
- A tour of \mathbf{G} is a directed simple cycle that includes every vertex in V .
- The cost of the tour is the sum of the cost of the edges on the tour.
- The TSP is to find a tour of minimum cost.
- Applications
 - Postal van routing
 - Production environment

TRAVELING SALESMAN PROBLEM

The solution approach

- Consider a tour to be a simple path that starts and ends at vertex **1**
- Every tour consists of an edge **(1,k)** for some $k \in V - \{1\}$ and a path from vertex **k** to **1**
- The path from vertex **k** to vertex **1** goes through each vertex in $V - \{1, k\}$ exactly once
- It's is easy to see that if the tour is optimal, then the path from **k** to **1** must be the shortest **k** to **1** path going through all vertices in $V - \{1, k\}$

TRAVELING SALESMAN PROBLEM

- Let $g(i, S)$ be the length of a shortest path starting at vertex i , going through all vertices in S , and terminating at vertex 1
- The function $g(1, V - \{1\})$ is the length of an optimal salesman tour
- So

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{A}$$

generalizing the above equation we have

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

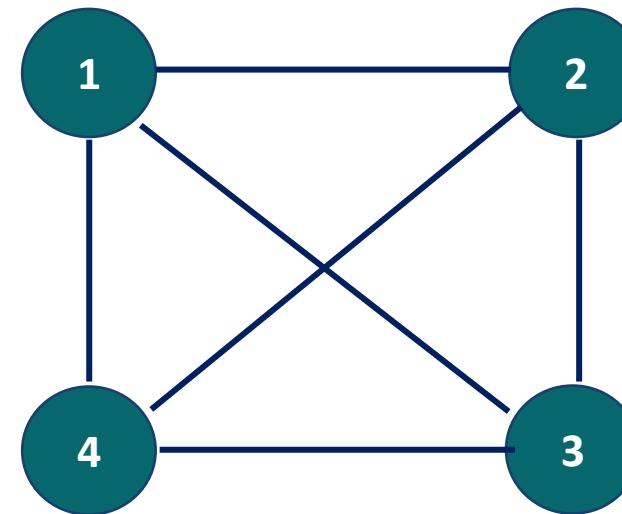
- $g(i, \phi) = c_{i1}, 1 \leq i \leq n.$
- Using equation B we can get $g(i, S)$ for all $|S| = 1, 2$ and so on
- When $|S| < n - 1$, the values of i and S for which $g(i, S)$ is needed are such that
 $i \neq 1, 1 \notin S \text{ and } i \notin S$

TRAVELING SALESMAN PROBLEM

- Solve the following instance of TSP using dynamic programming

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

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



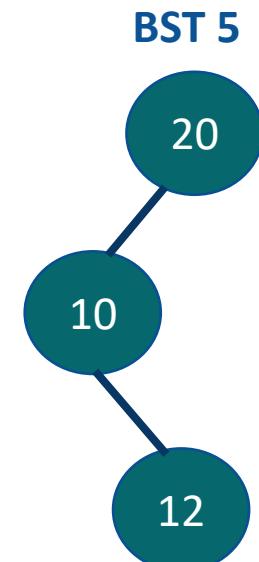
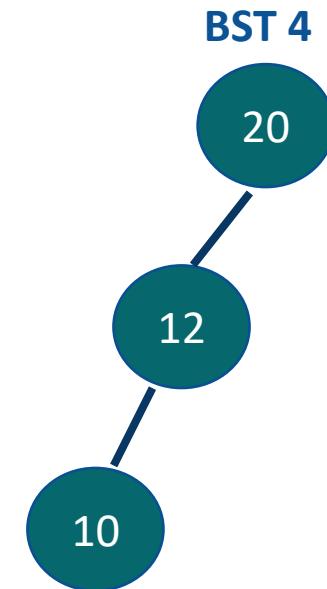
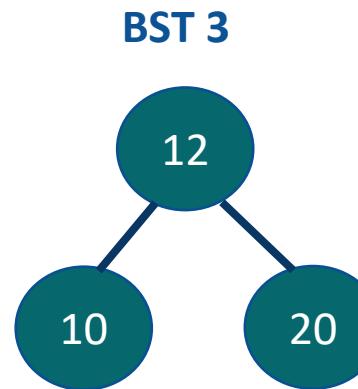
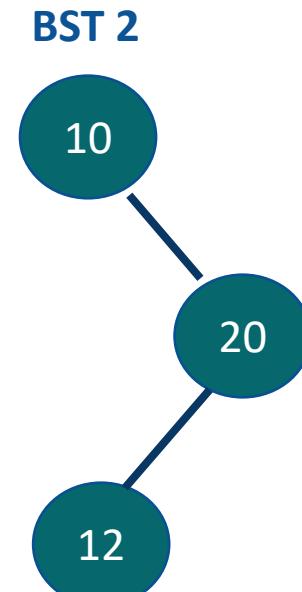
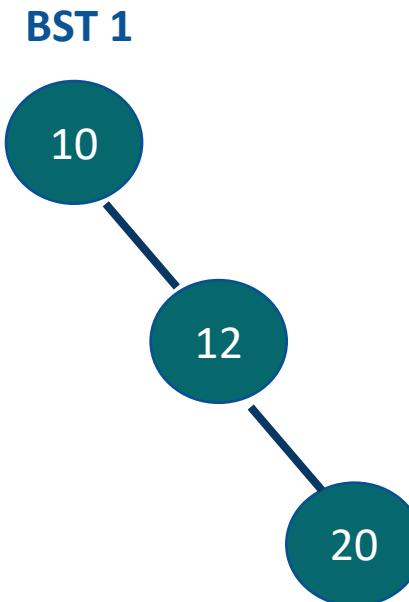
OPTIMAL BINARY SEARCH TREES

- A binary search tree is one of the most important data structures in computer science
- A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes.
 - The external nodes are null nodes.
 - The keys are ordered **lexicographically**, i.e. for each internal node all the keys in the **left sub-tree** are **less than** the keys in the node, and all the keys in the **right sub-tree** are **greater**.
- The major applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion
- If probabilities of searching for elements of a set are known
 - e.g., from accumulated data about past searches
 - it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible

OPTIMAL BINARY SEARCH TREES

Example

- Keys={10, 12, 20} Frequency=(34, 8, 50)
- Possible BST are

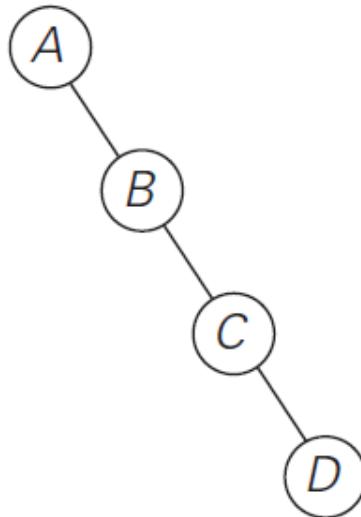


OPTIMAL BINARY SEARCH TREES

- Example

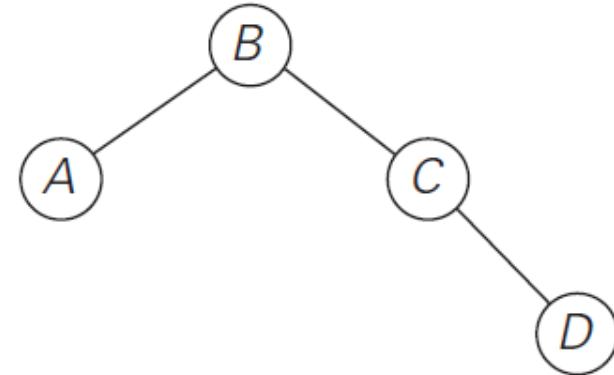
Keys	Probability
A	0.1
B	0.2
C	0.4
D	0.3

BST 1



Avg. no. of successful comparison
 $0.1 \cdot 1 + 0.2 \cdot 2 + 0.4 \cdot 3 + 0.3 \cdot 4 = 2.9$

BST 2



Avg. no. of successful comparison
 $0.1 \cdot 2 + 0.2 \cdot 1 + 0.4 \cdot 2 + 0.3 \cdot 3 = 2.1$

OPTIMAL BINARY SEARCH TREES

- Total number of binary search trees with n keys is equal to the n th *Catalan number*.

$$c(n) = \frac{1}{n+1} \binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1,$$

which grows to infinity as fast as $4^n/n^{1.5}$

$$C(n, r) = \binom{n}{r} = \frac{n!}{(r!(n-r)!)}$$

OPTIMAL BINARY SEARCH TREES

Alternate approach to find number of BST possible with n nodes

$$t(n) = \sum_{i=1}^n t(i-1) t(n-i).$$

- The base case is $t(0) = 1$ and $t(1) = 1$, i.e. there is one empty BST and there is one BST with one

$$t(2) = t(0)t(1) + t(1)t(0) = 2$$

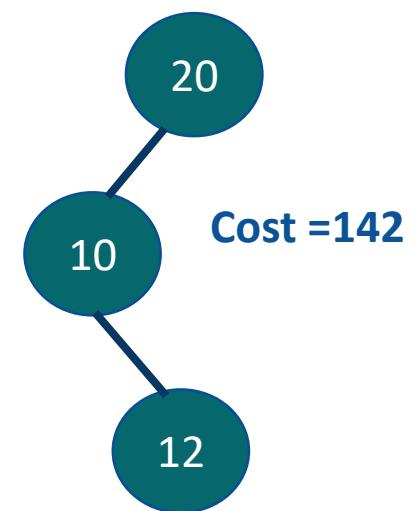
$$t(3) = t(0)t(2) + t(1)t(1) + t(2)t(0) = 2 + 1 + 2 = 5$$

$$t(4) = t(0)t(3) + t(1)t(2) + t(2)t(1) + t(3)t(0) = 5 + 2 + 2 + 5 = 14$$

OPTIMAL BINARY SEARCH TREES

Definitions

- An Optimal Binary Search Tree (**OBST**) is a Binary Search Tree (**BST**), which has minimal expected cost of locating each node
- An **OBST** is a **BST** for which the nodes are arranged on levels such that the tree cost is minimum
- Example
 Keys={10, 12, 20} Frequency=(34, 8, 50)
 For n = 0, 1, 2, 3, ... values of Catalan numbers are
 - 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862,
 - So are numbers of Binary Search Trees.



OPTIMAL BINARY SEARCH TREES

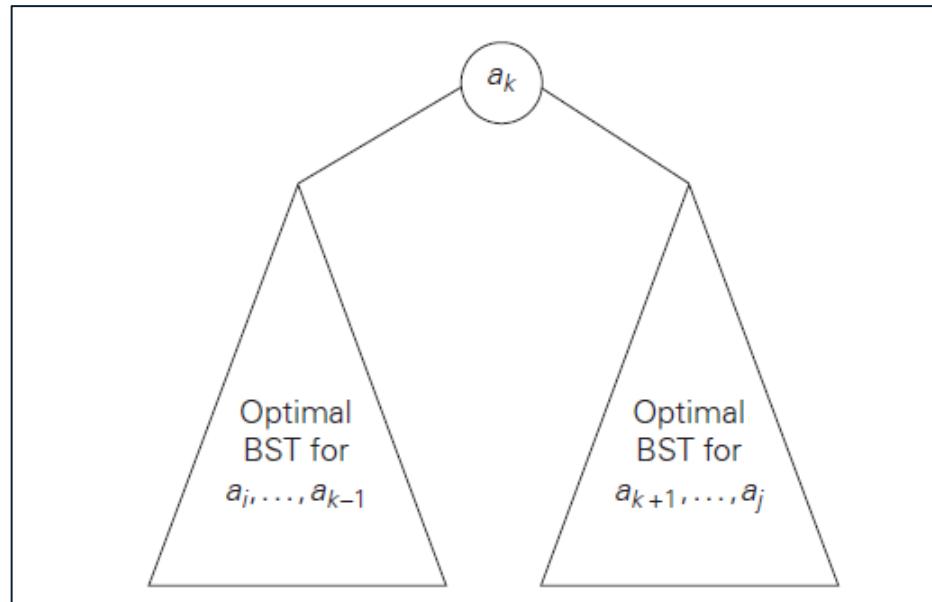
Dynamic Programming Approach

- let a_1, \dots, a_n be distinct keys ordered from the smallest to the largest
- let p_1, \dots, p_n be the probabilities of searching for them
- Let $C(i, j)$ be the smallest average number of comparisons made in a successful search in a binary search tree T_i^j made up of keys a_i, \dots, a_j where i, j are some integer indices, $1 \leq i \leq j \leq n$.
 - To derive a recurrence underlying a dynamic programming algorithm, we will consider all possible ways to choose a root a_k among the keys a_i, \dots, a_j .

OPTIMAL BINARY SEARCH TREES

Dynamic Programming Approach

- For such a binary search tree (shown below) the root contains key a_k , the left subtree T_i^{k-1} contains keys a_i, \dots, a_{k-1} optimally arranged, the right subtree T_{k+1}^j contains keys a_{k+1}, \dots, a_j also optimally arranged



OPTIMAL BINARY SEARCH TREES

- The recurrence for solving the problem of finding OBST is

$$C(i, j) = \min_{i \leq k \leq j} \{C(i, k - 1) + C(k + 1, j)\} + \sum_{s=i}^j p_s \quad \text{for } 1 \leq i \leq j \leq n.$$

- Assumptions

- $C(i, i - 1) = 0$ for $1 \leq i \leq n + 1$, (number of comparisons in the empty tree)
- $C(i, i) = p_i$ for $1 \leq i \leq n$, (*BST with one node a_i*)

OPTIMAL BINARY SEARCH TREES

- Table structure

	0	p_1							goal
		0	p_2						
i							$C[i, j]$		
$n + 1$							p_n		0

The diagram illustrates a dynamic programming table for constructing an optimal binary search tree. The table has rows labeled 1, i , and $n + 1$. The columns are unlabeled but correspond to nodes p_1, p_2, \dots, p_n . The cell $C[i, j]$ is highlighted in blue. Arrows point from the bottom right towards $C[i, j]$, indicating the recursive nature of the algorithm.

OPTIMAL BINARY SEARCH TREES

- Obtain the optimal binary search tree for the following given keys and their probabilities
-

Keys	Probability
A	0.1
B	0.2
C	0.4
D	0.3

OPTIMAL BINARY SEARCH TREES

- Solution

		main table				
		0	1	2	3	4
1	0	0.1				
2		0	0.2			
3			0	0.4		
4				0	0.3	
5					0	

$$C(i, i - 1) = 0 \text{ for } 1 \leq i \leq n + 1$$

$$C(i, i) = p_i \text{ for } 1 \leq i \leq n,$$

		root table				
		0	1	2	3	4
1	0	1				
2			2			
3				3		
4					4	
5						

$$R(i, i) = i \text{ for } 1 \leq i \leq n$$

OPTIMAL BINARY SEARCH TREES

- Solution is elaborated in the lecture notes

TRAVELING SALESMAN PROBLEM

- Jg

TRAVELING SALESMAN PROBLEM

DYNAMIC PROGRAMMING

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

THANK YOU

Design and Analysis of Algorithms

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

ESTD:2001

An Institute with a Difference

Backtracking – N Queens Problem

- The problem is to place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

- When $n=1$?

Q

- When $n=2$?

Q	
	?

- When $n=3$?

Q		
?	?	?

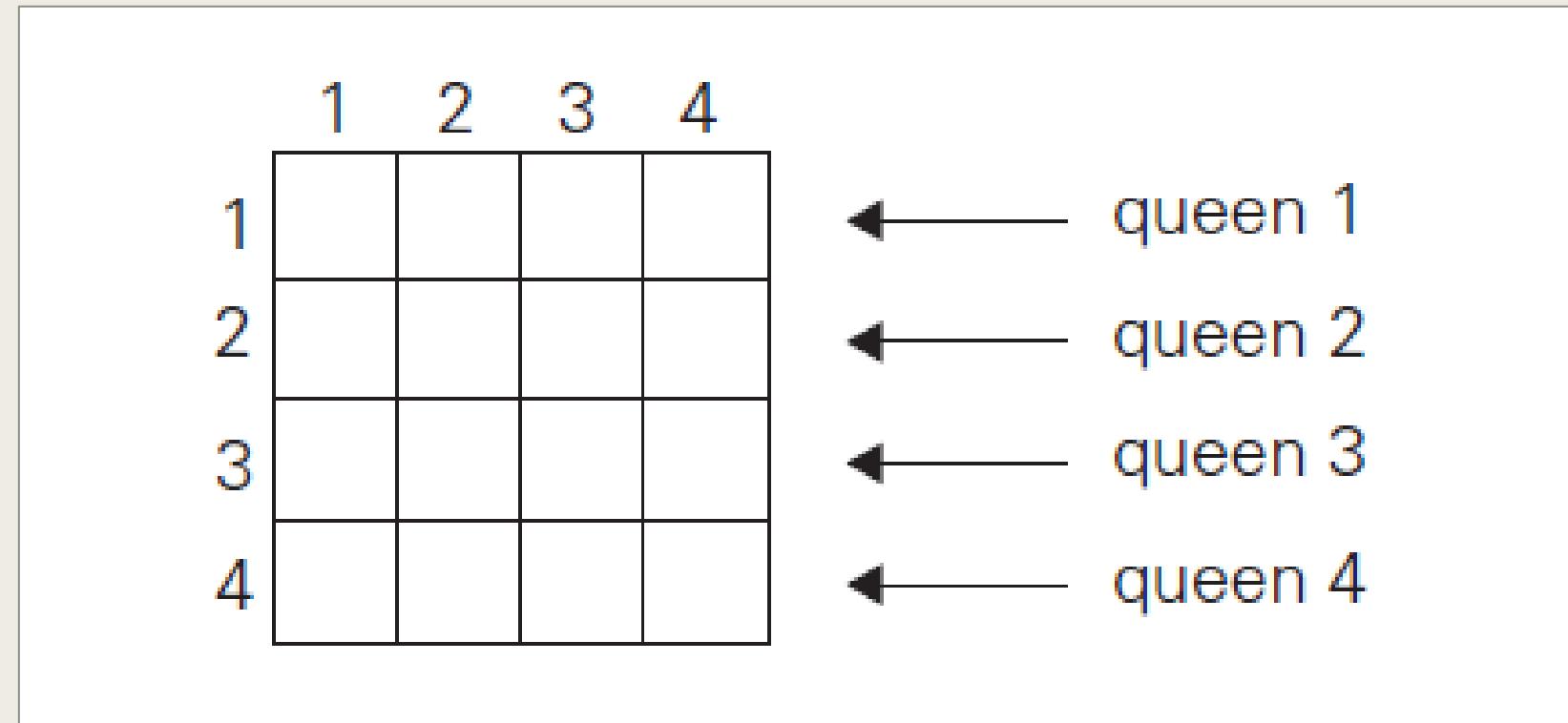
	Q	
?	?	?

		Q
?	?	?

- When $n=4$?

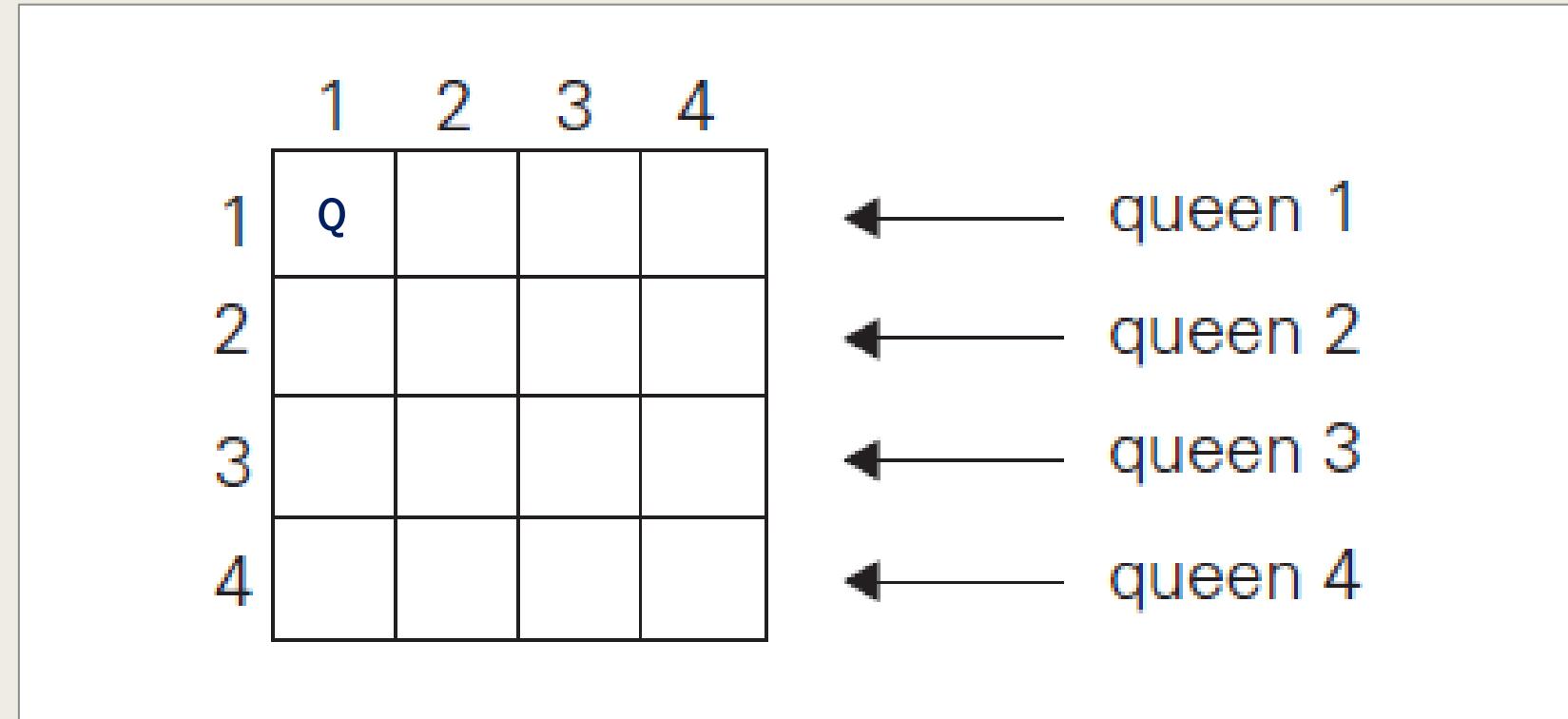
Backtracking – N Queens Problem

- When $n=4$?



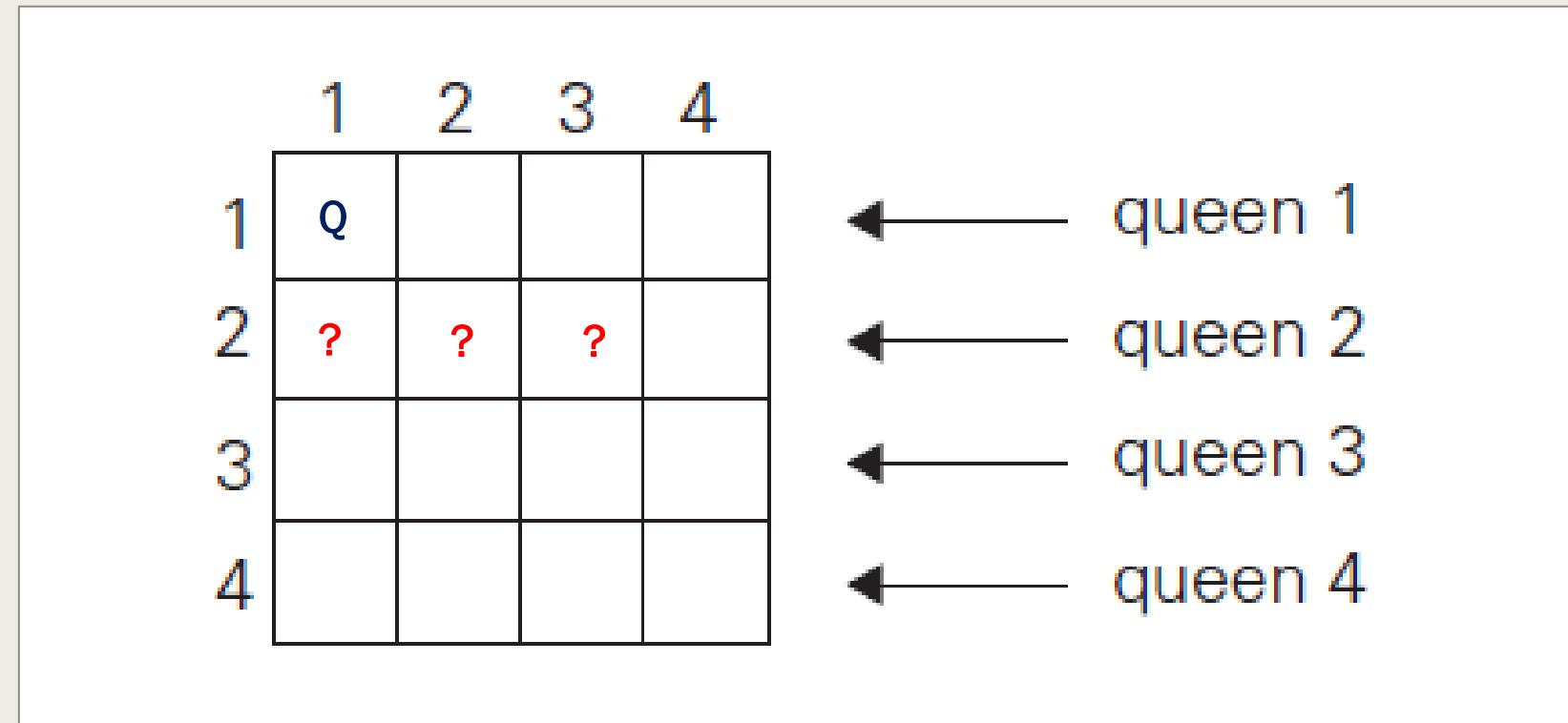
Backtracking – N Queens Problem

- When $n=4$?



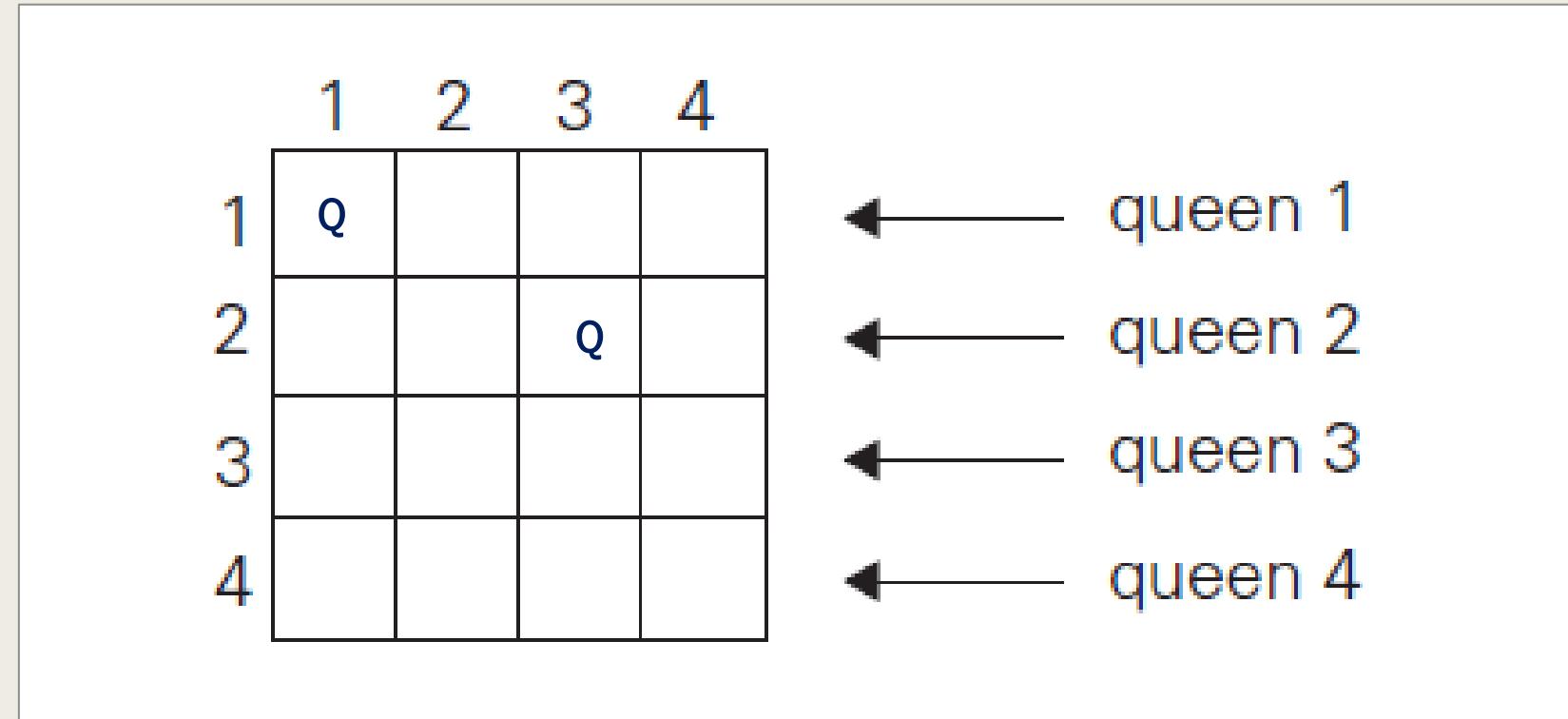
Backtracking – N Queens Problem

- When $n=4$?



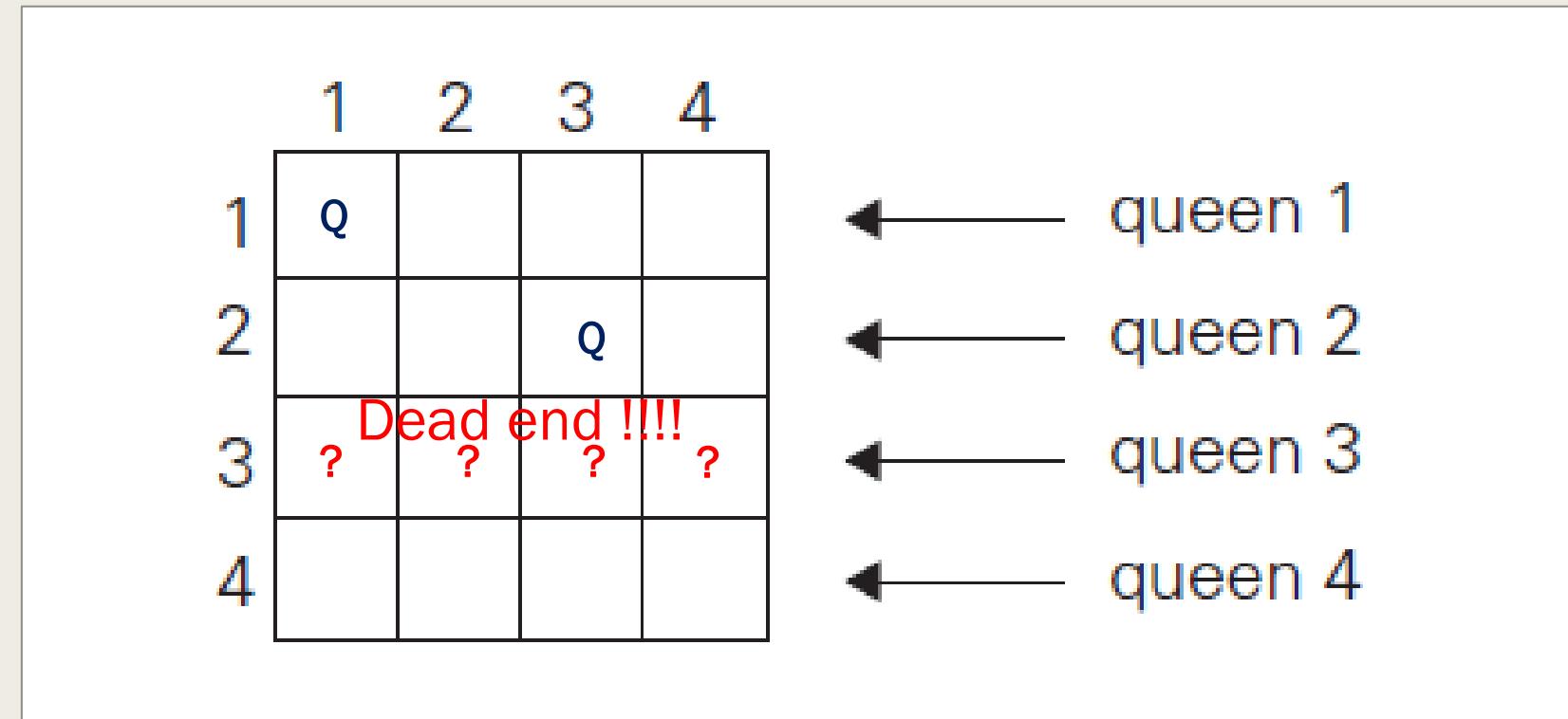
Backtracking – N Queens Problem

- When $n=4$?



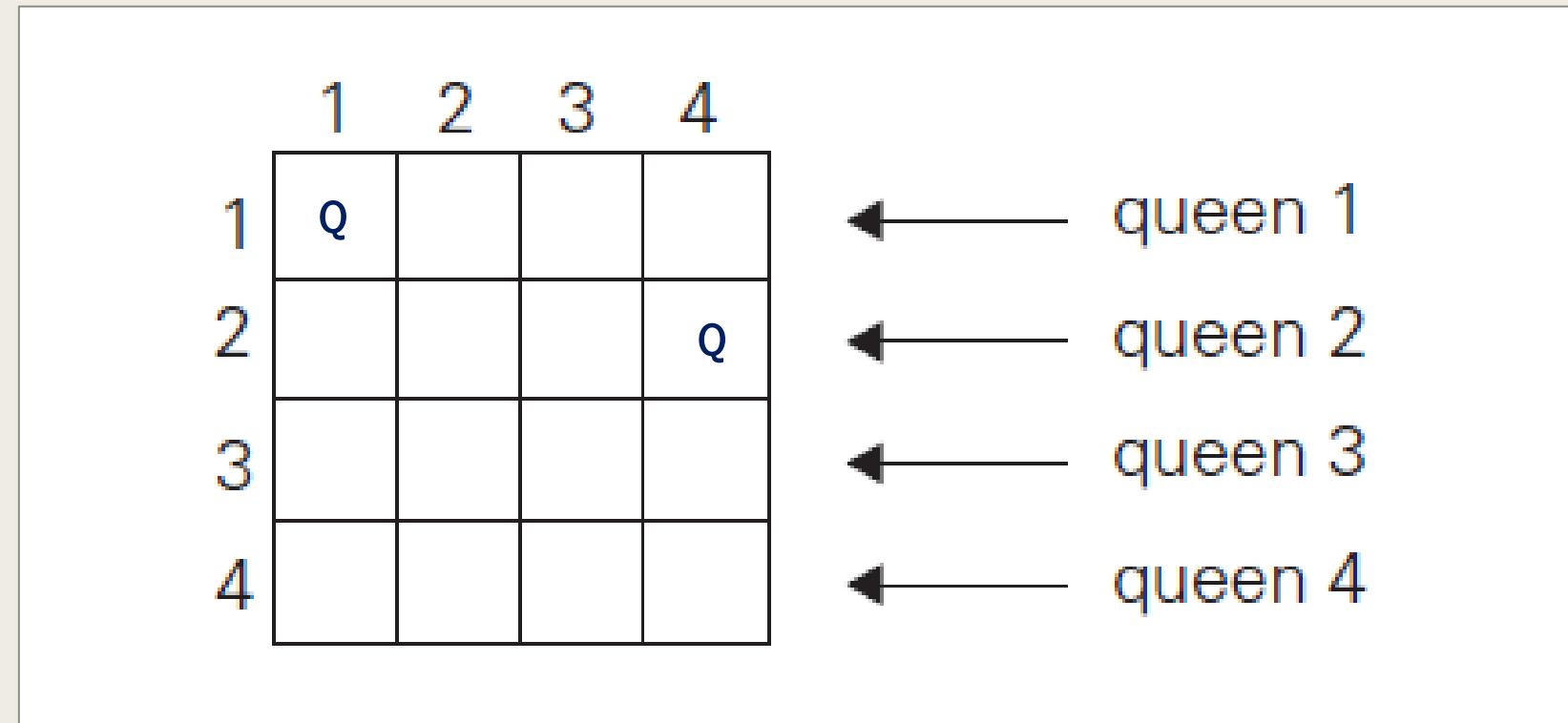
Backtracking – N Queens Problem

- When $n=4$?



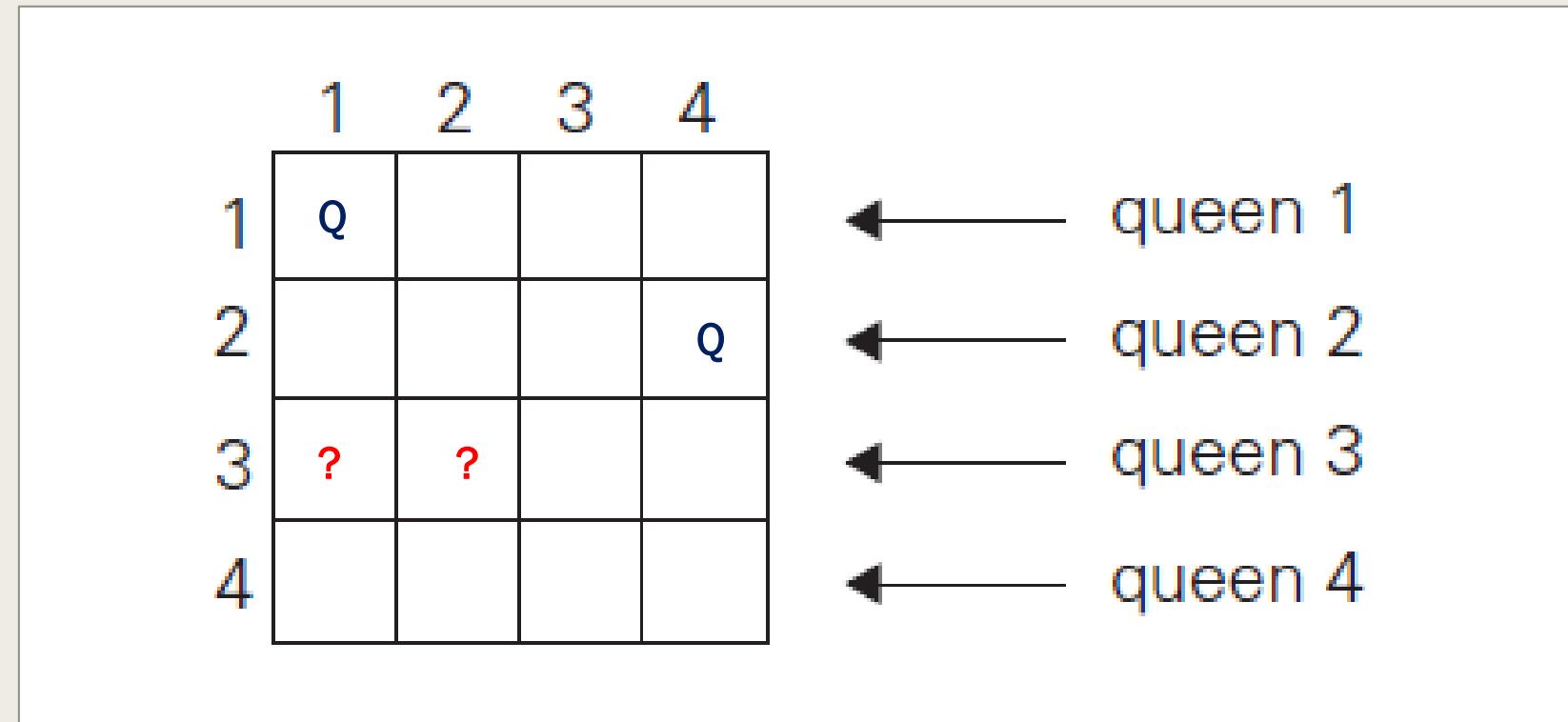
Backtracking – N Queens Problem

- When $n=4$?



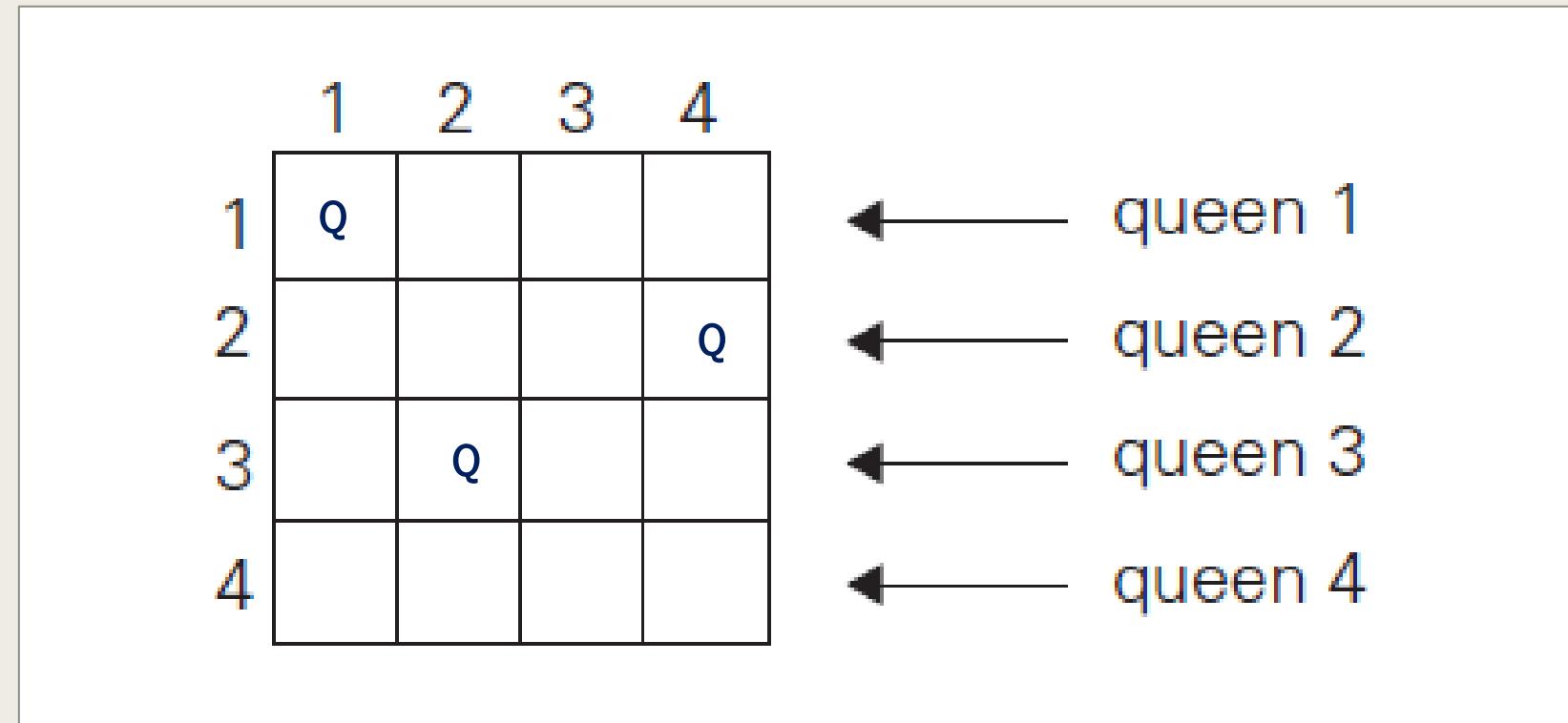
Backtracking – N Queens Problem

- When $n=4$?



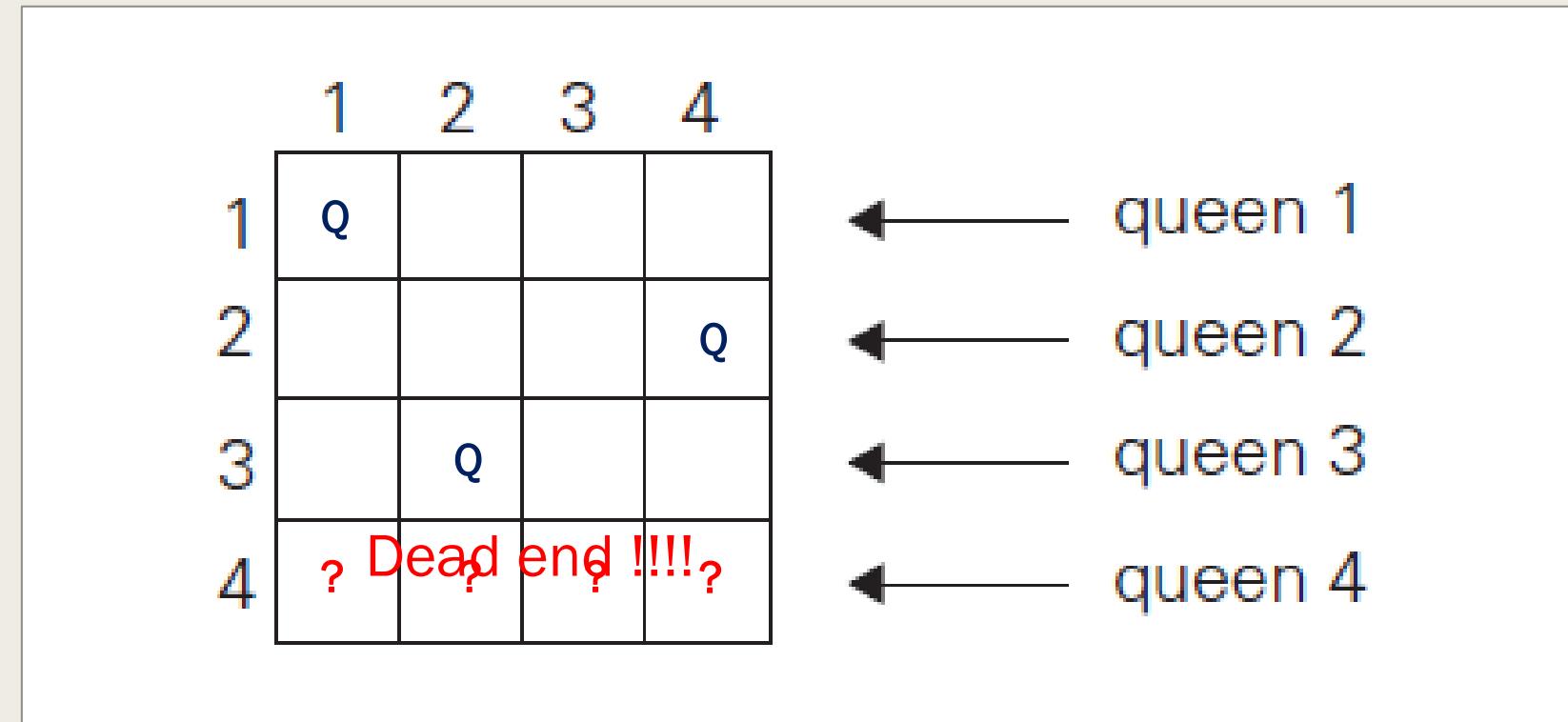
Backtracking – N Queens Problem

- When $n=4$?



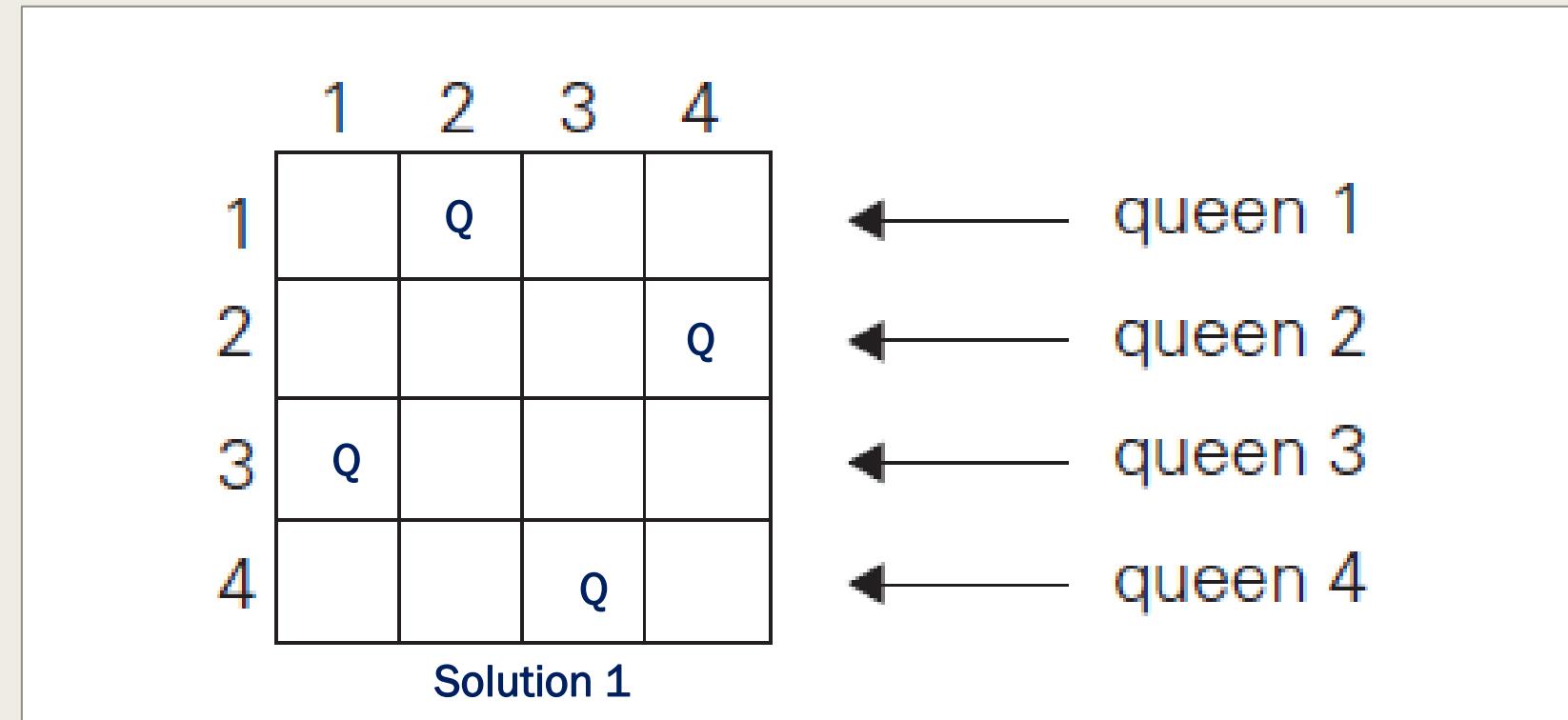
Backtracking – N Queens Problem

- When $n=4$?



Backtracking – N Queens Problem

- When $n=4$?



Backtracking – N Queens Problem

- When $n=4$? Alternate solution ??

	1	2	3	4
1			Q	
2	Q			
3				Q
4		Q		

Solution 1

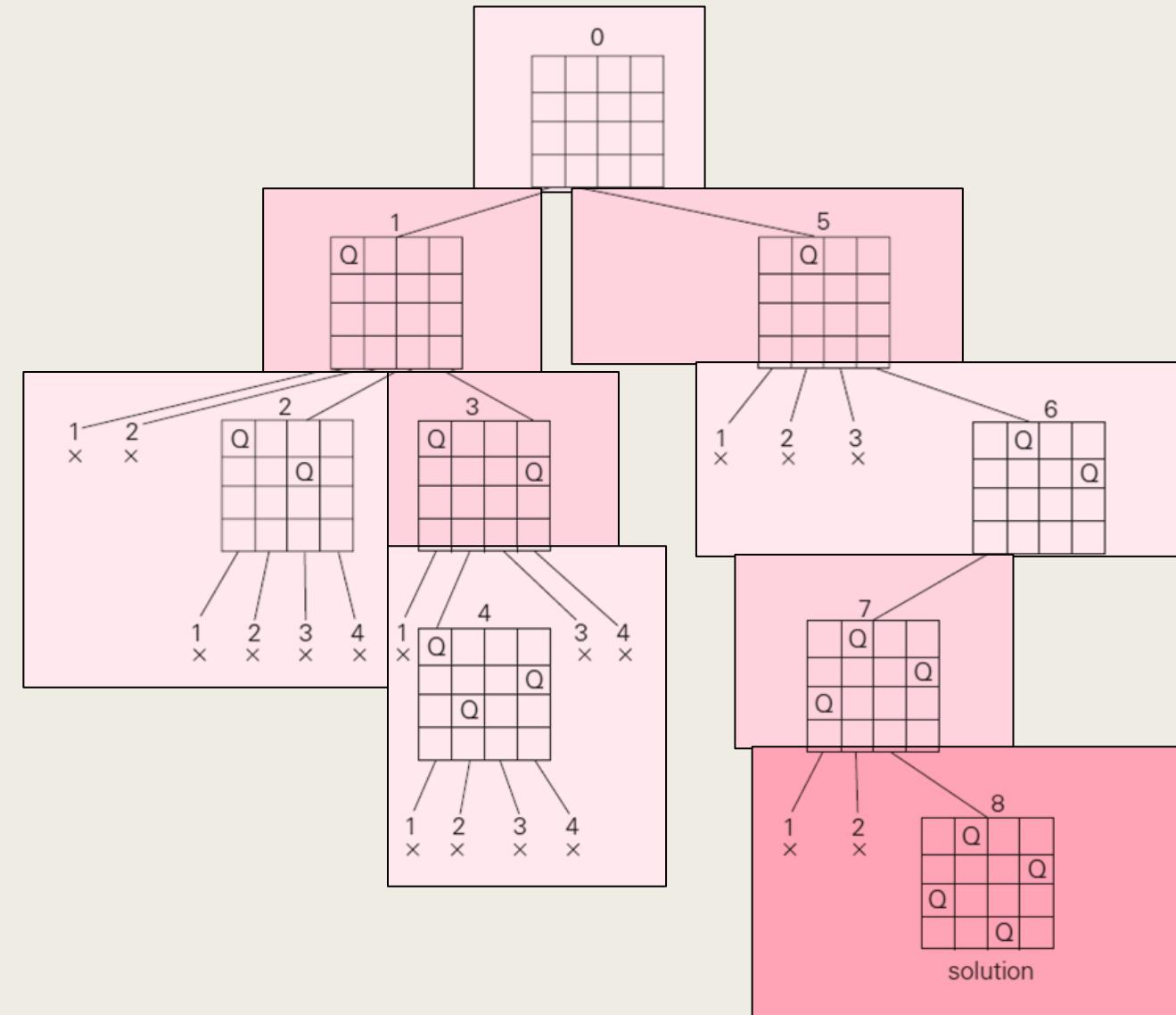
← queen 1

← queen 2

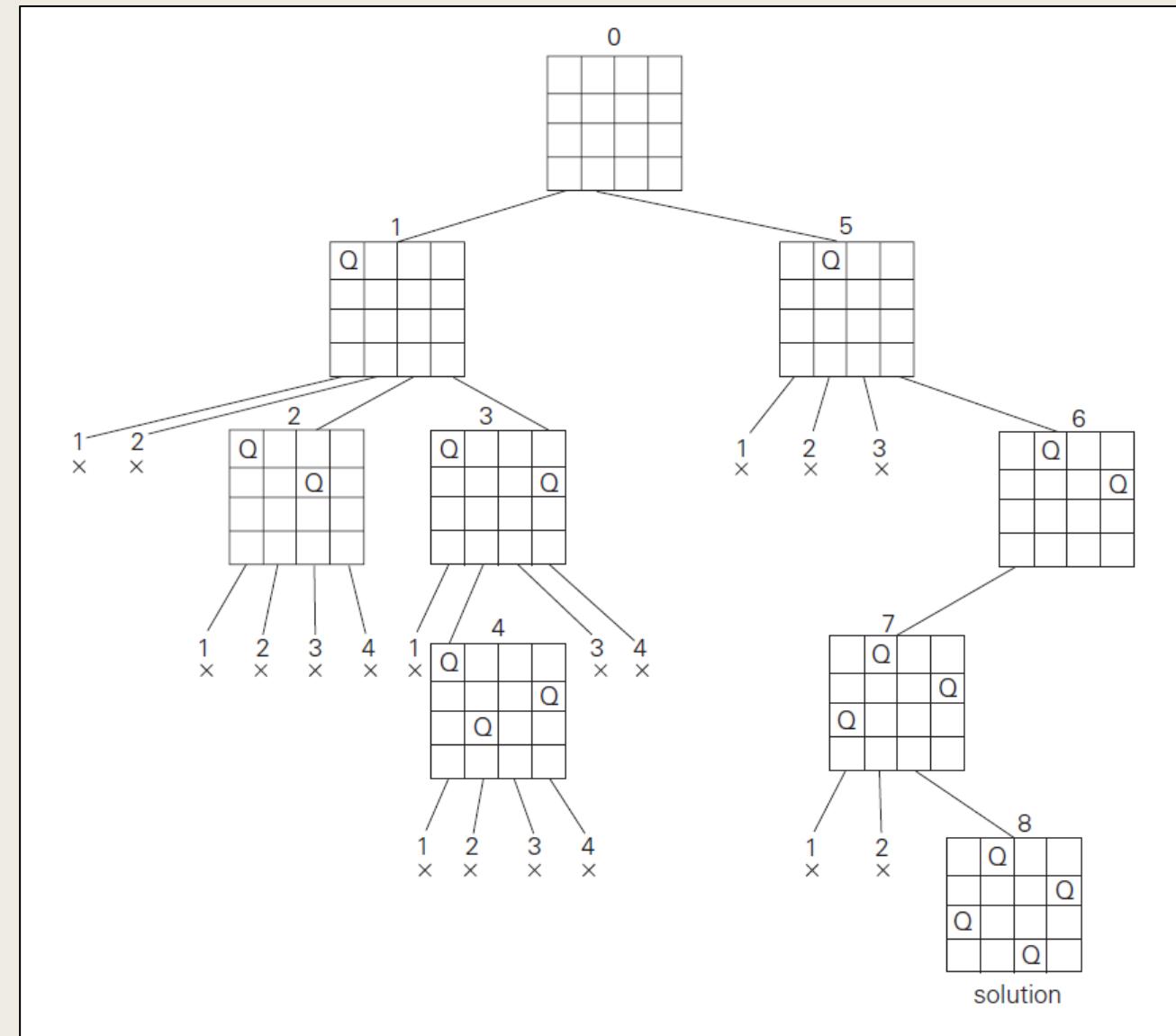
← queen 3

← queen 4

Backtracking – N Queens Problem



Backtracking – N Queens Problem



Design and Analysis of Algorithms



Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

ESTD:2001

An Institute with a Difference

Backtracking – Sum of subsets

- Given set $A = \{a_1, \dots, a_n\}$ of n positive integers, find **subset/s** whose sum is equal to a given positive integer d .
- **Example:**
 - $A = \{1, 2, 5, 6, 8\}$ and $d = 9$,
 - there are two solutions:
 - $\{1, 2, 6\}$
 - $\{1, 8\}$.
 - $B=\{11, 13, 24, 7\}$ and $d= 31$
 - **There are two solutions**
 - $\{11,13,7\}$
 - $\{24, 7\}$

Backtracking – Sum of subsets



Solution approach

- Sort the set's elements in increasing order.

$$a_1 < a_2 < \dots < a_n$$

- The **state space tree** can be constructed as **binary tree**
- The **root** of the tree represents the **starting point**, with no decisions about the given elements made as yet
- Its **left** and **right** children represent, respectively, **inclusion** and **exclusion** of a_1 in a set
- Similarly, going to the **left** from a node of the first level corresponds to **inclusion** of a_2 while going to the **right** corresponds to its **exclusion**, and so on

Backtracking – Sum of subsets

- Thus, a path from the root to a node on the i^{th} level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.
- We record the value of s , the sum of these numbers, in the node.
 - If s is equal to d , we have a **solution** to the problem.
- We can either **report** this result and **stop** or, if **all the solutions** need to be found, then, continue by **backtracking** to the node's parent.
- If s is not equal to d , we can terminate the node as **nonpromising** if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad (\text{sum } s \text{ is too large})$$

$$s + \sum_{j=1+1}^n a_j < d \quad (\text{sum } s \text{ is too small})$$

Backtracking – Sum of subsets



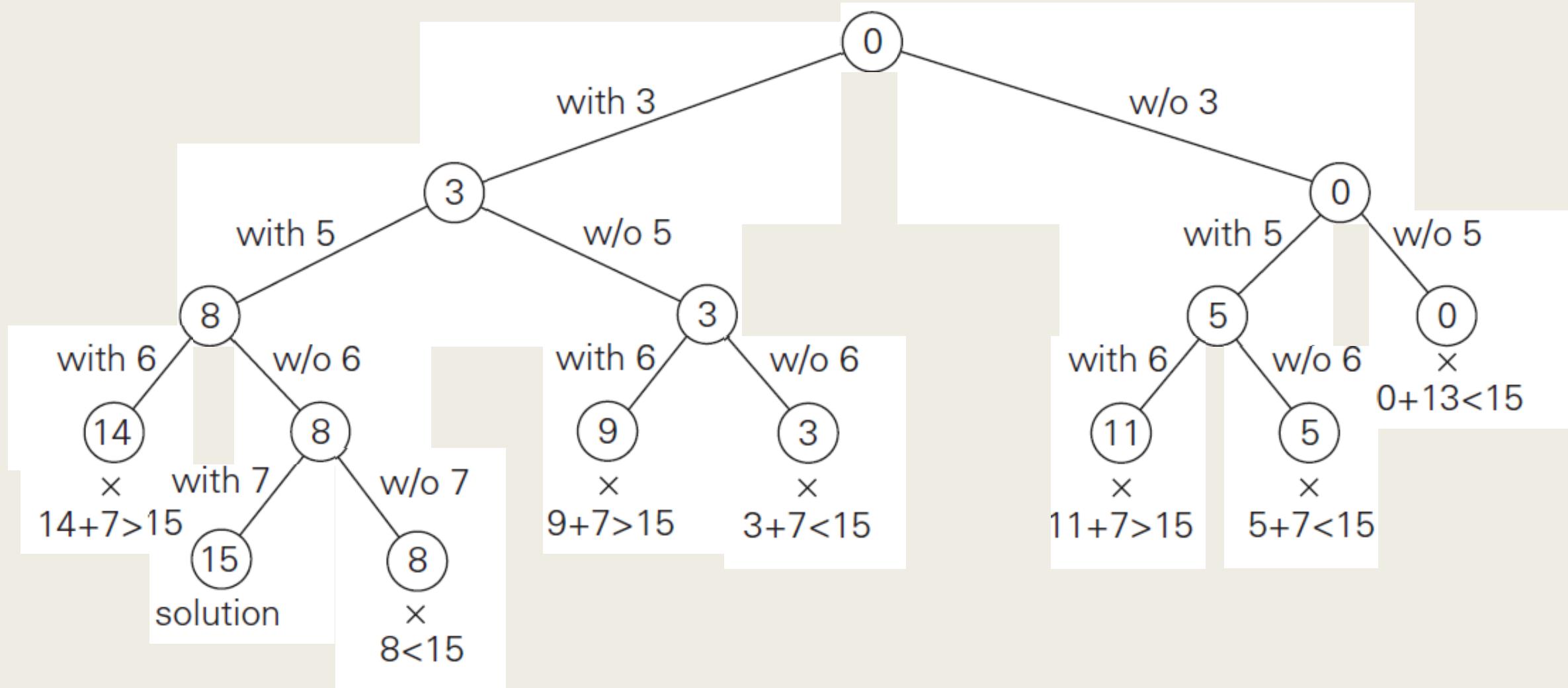
Apply backtracking to solve the following instance of the subset sum problem: $A = \{3, 5, 6, 7\}$ and $d = 15$.

– *Draw the complete state space tree*

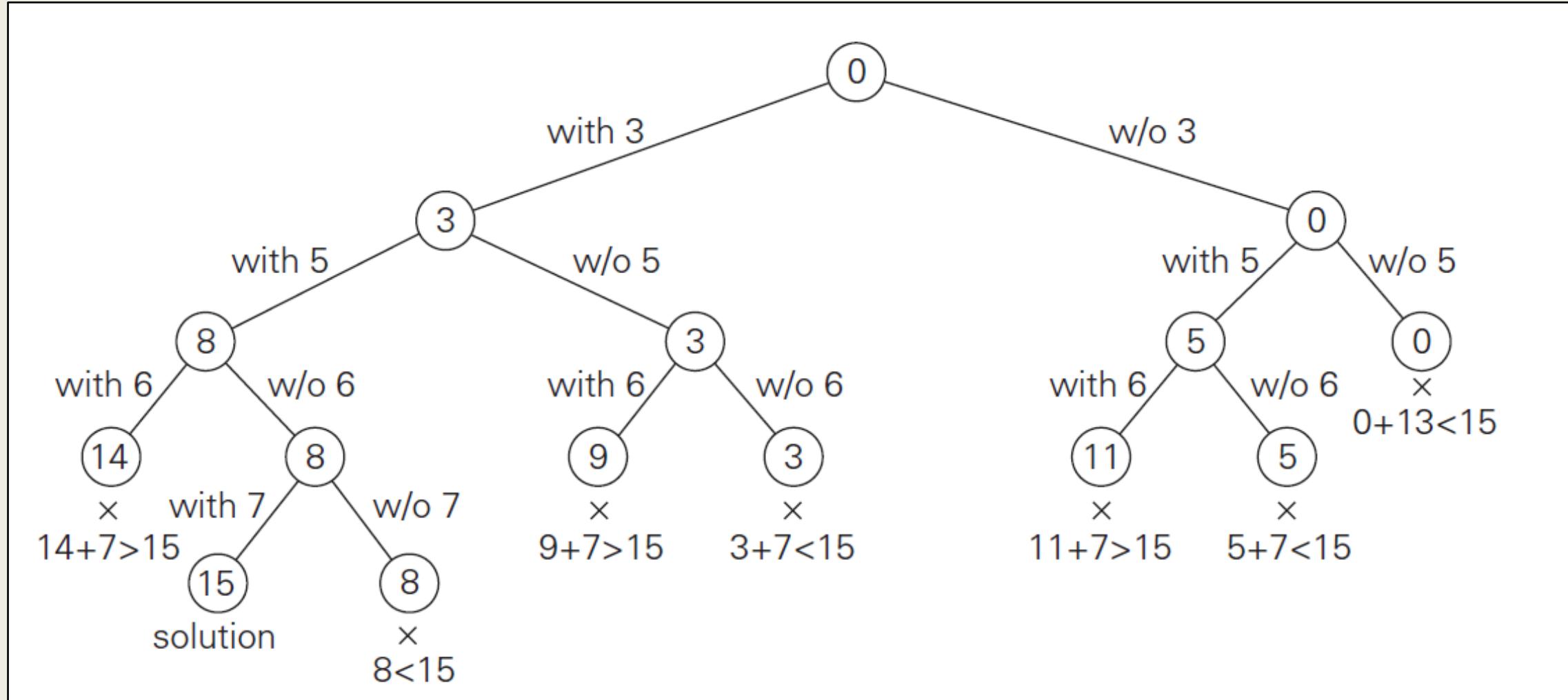
Solution

- Arrange the elements in **ascending** order
- Compute the sum **Ts** of all the elements
- If **Ts** is less than **d**
 - *then no solution*
- Start drawing the state space tree

Backtracking – Sum of subsets



Backtracking – Sum of subsets



Backtracking – Sum of subsets



- Apply backtracking to solve the following instance of the subset sum problem: $A = \{1, 3, 4, 5\}$ and $d = 11$.
- Draw the complete state space tree

Design and Analysis of Algorithms



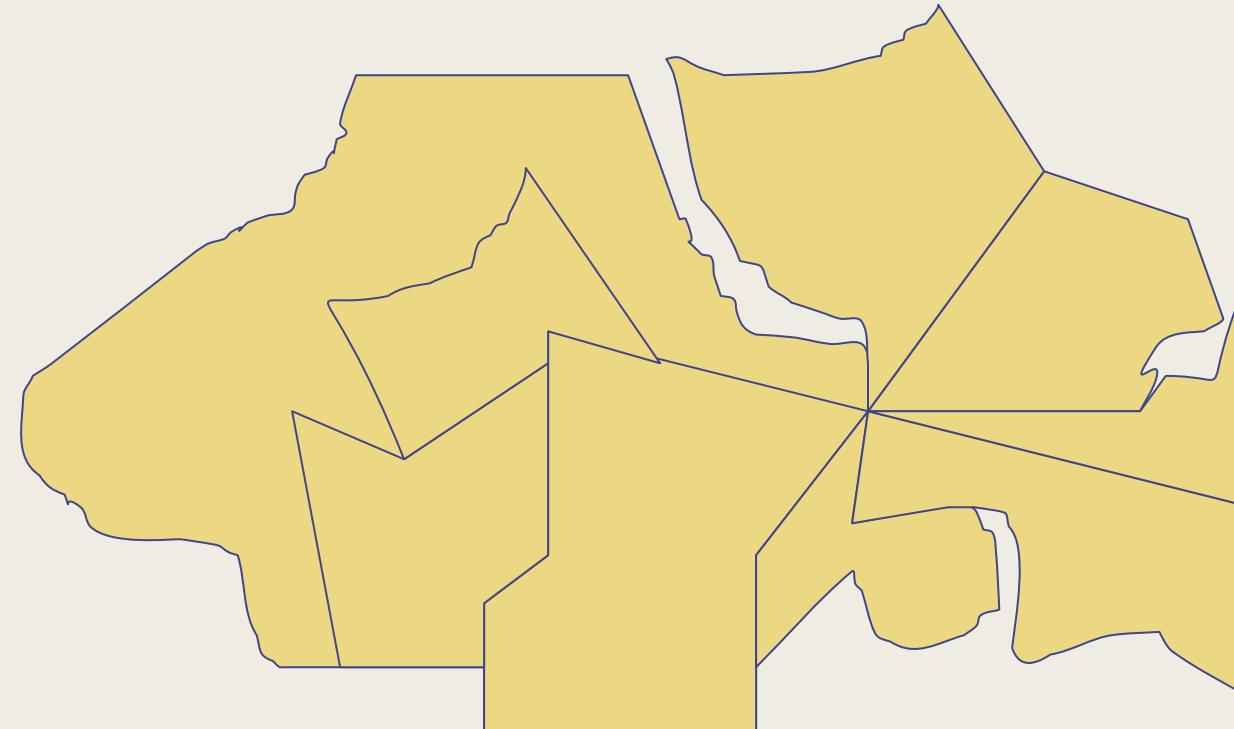
Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

ESTD:2001

An Institute with a Difference

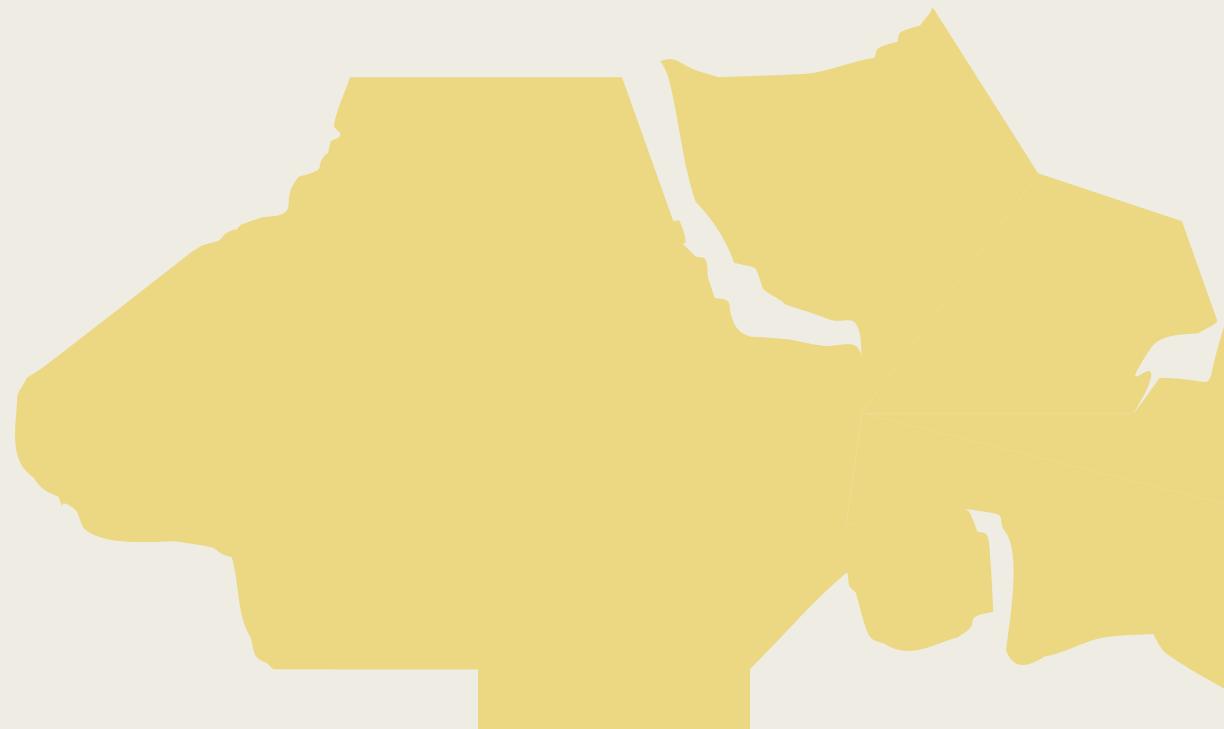
Backtracking – Graph Coloring

- Consider this fictional continent



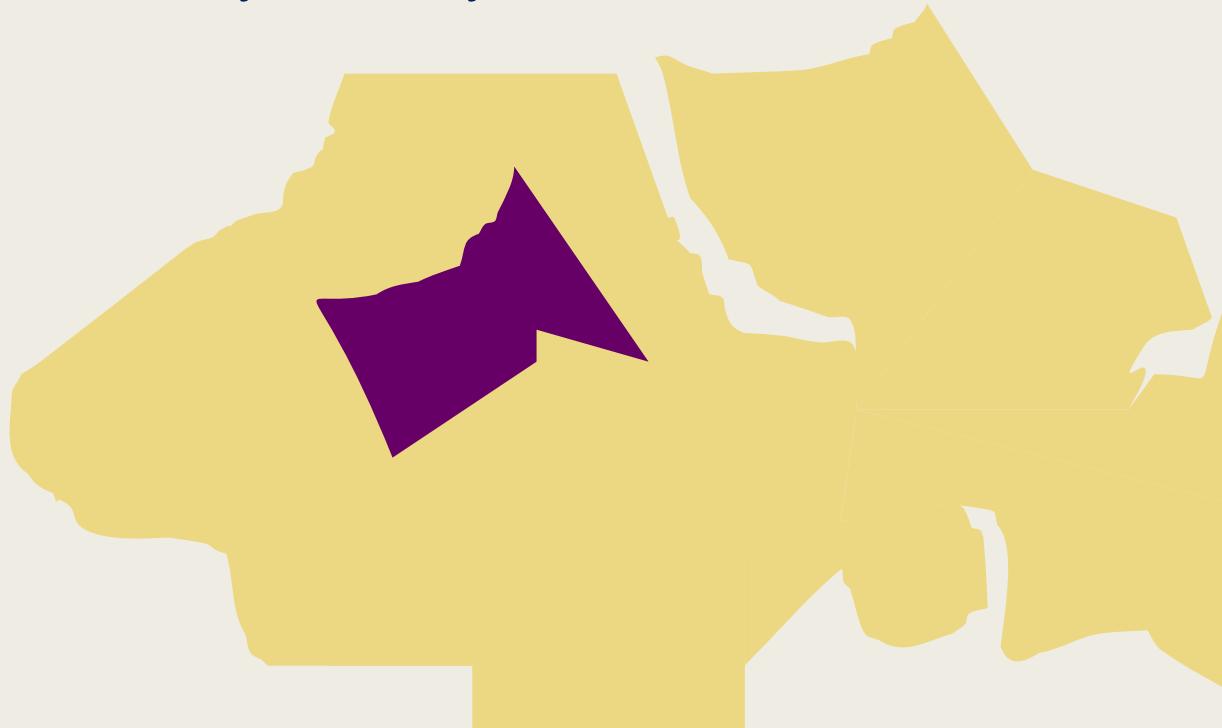
Backtracking – Graph Coloring

- Lets remove all borders but you may still want to see all the countries.
- 1 color is sufficient? **No !!!!** Its insufficient.



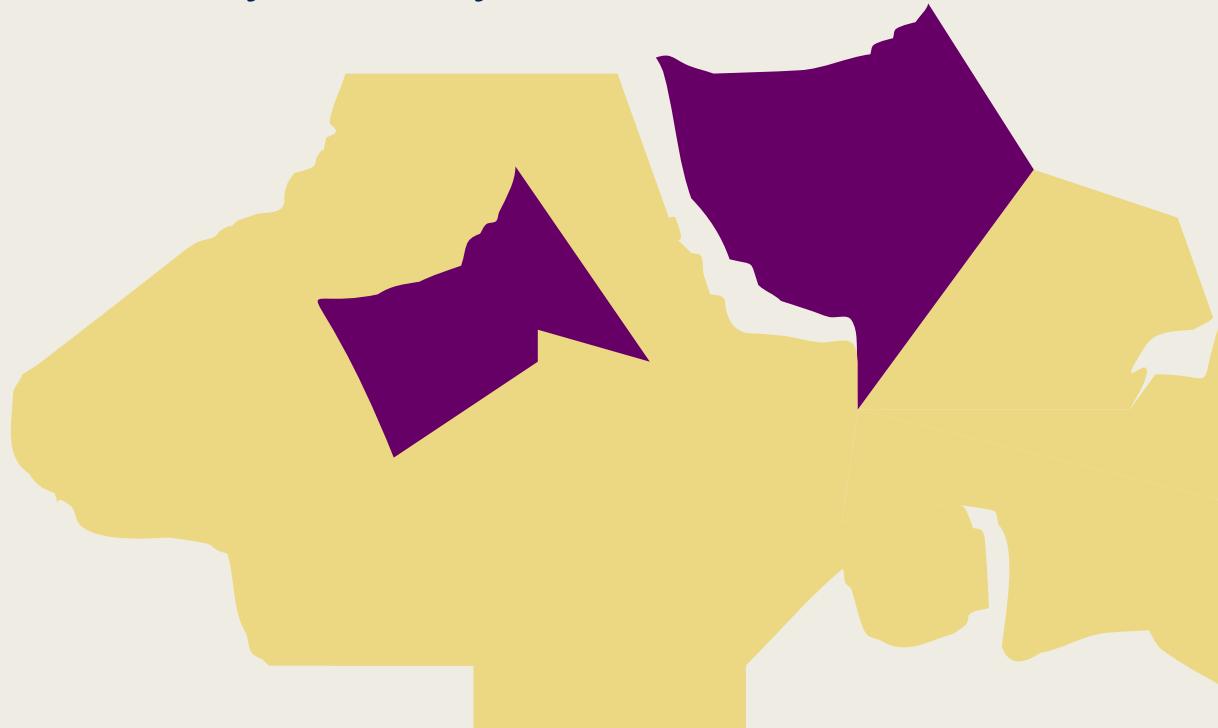
Backtracking – Graph Coloring

- So add another color.
- Now try to fill in every country with one of the two colors.



Backtracking – Graph Coloring

- So add another color.
- Now try to fill in every country with one of the two colors.



Backtracking – Graph Coloring

- So add another color.
- Now try to fill in every country with one of the two colors.



Backtracking – Graph Coloring

- So add another color.
- Now try to fill in every country with one of the two colors.



Backtracking – Graph Coloring

- **PROBLEM !!!!**
- Two adjacent countries forced to have same color. Border unseen.



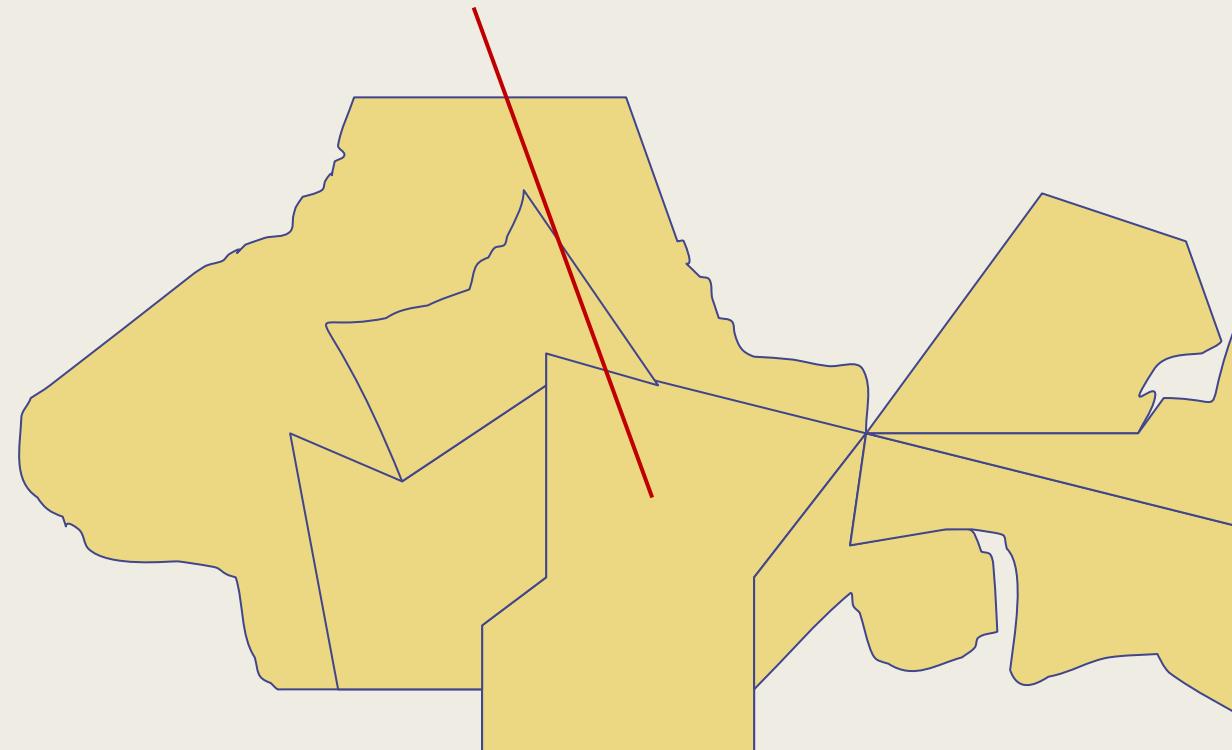
Backtracking – Graph Coloring

- So add another color



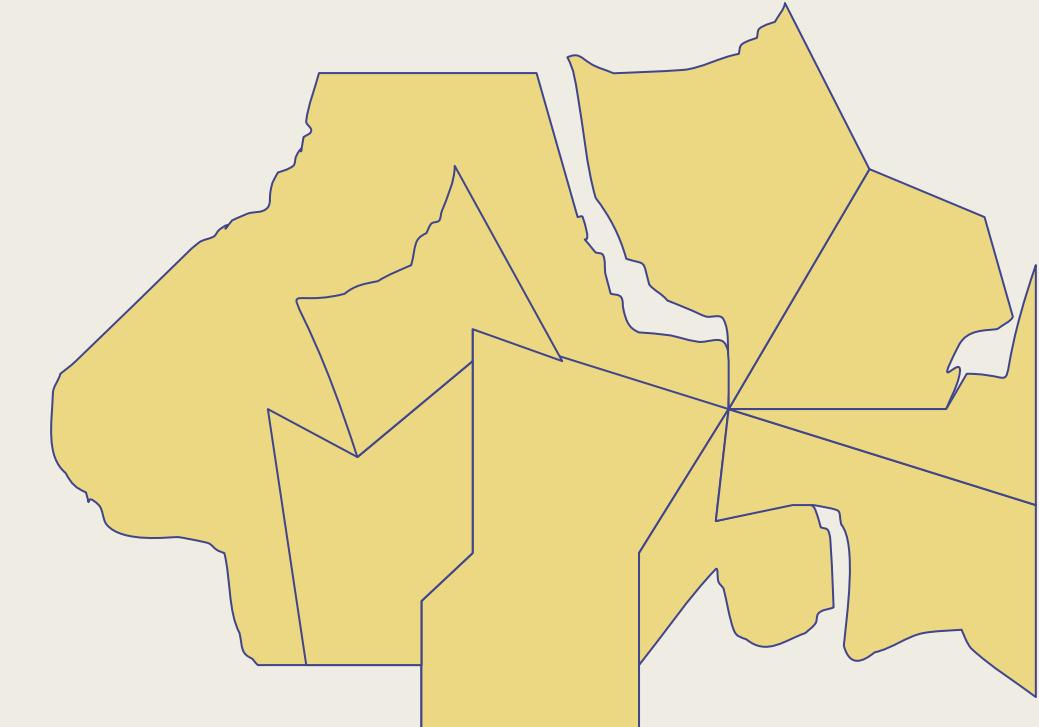
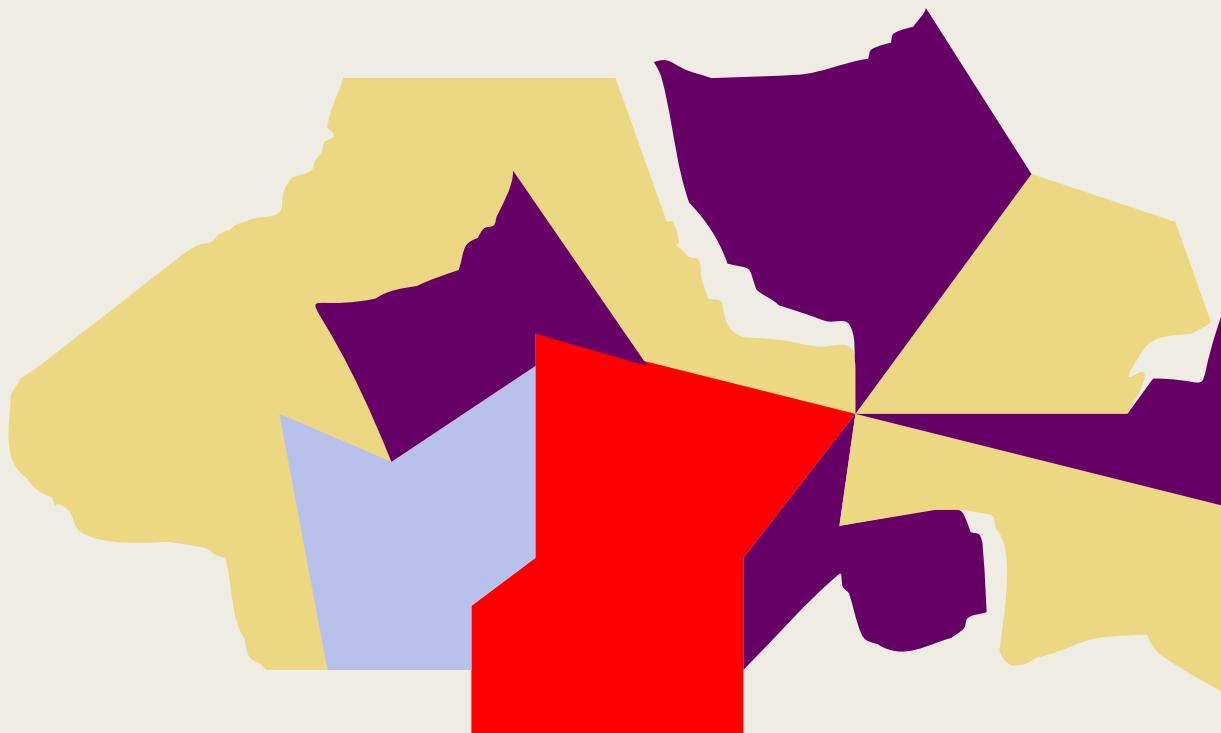
Backtracking – Graph Coloring

- Insufficient.
- Need 4 colors because of this country.



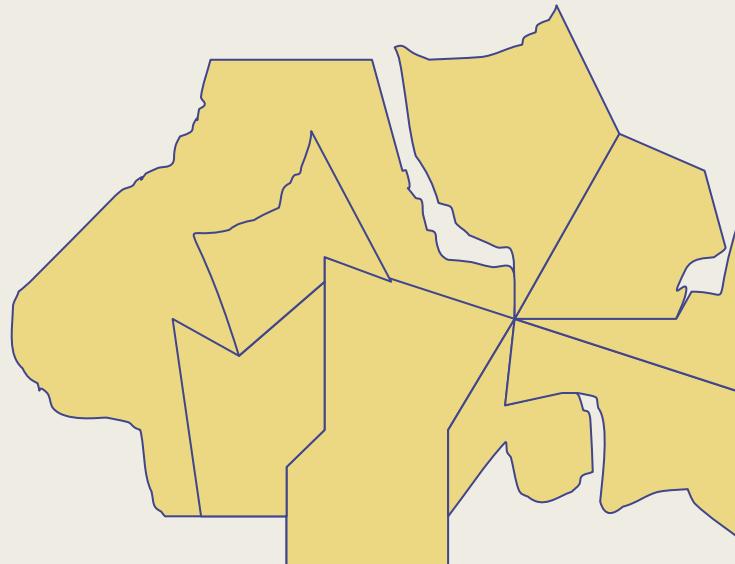
Backtracking – Graph Coloring

- Using 4 colors, the entire graph is colored and the borders are clearly seen

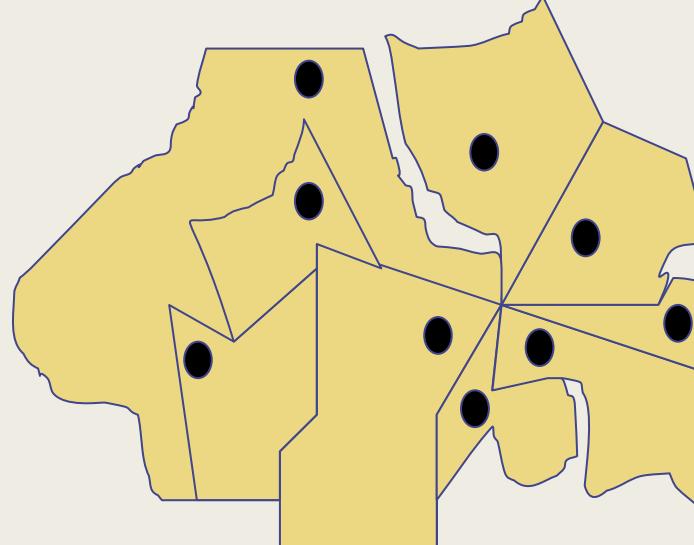


Backtracking – Graph Coloring

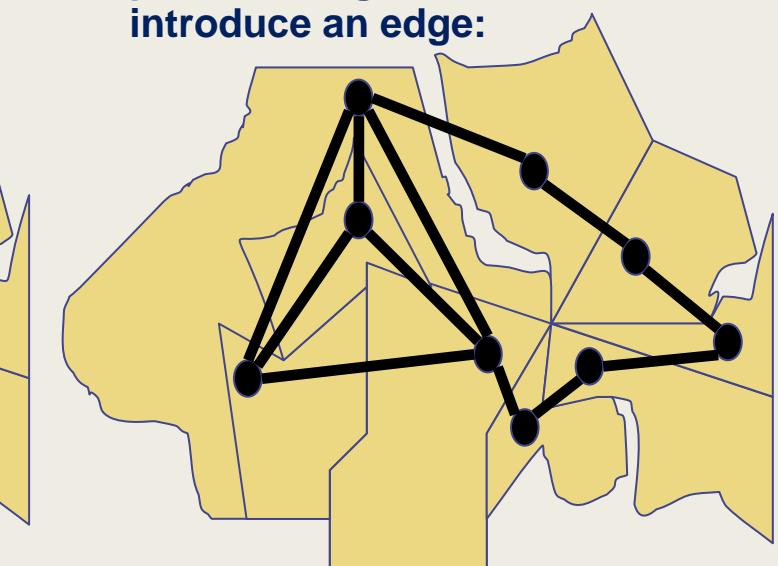
Region to be coloured



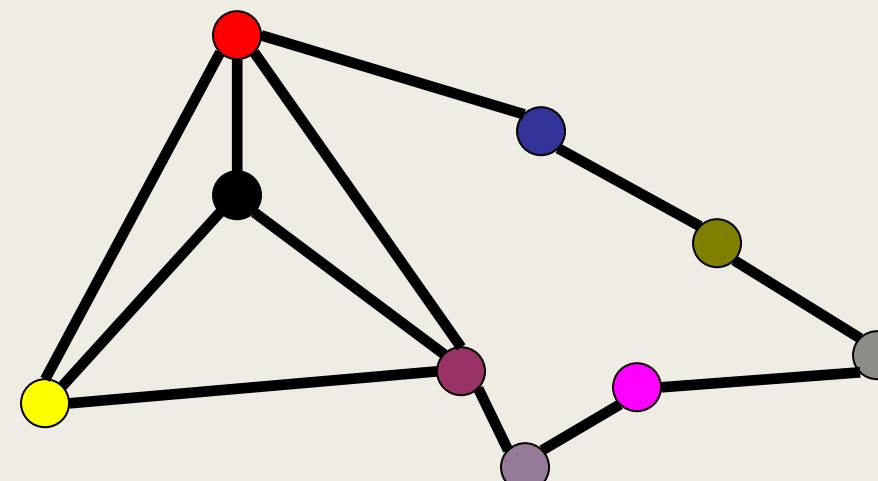
For each region introduce a vertex



For each pair of regions with a positive-length common border introduce an edge:

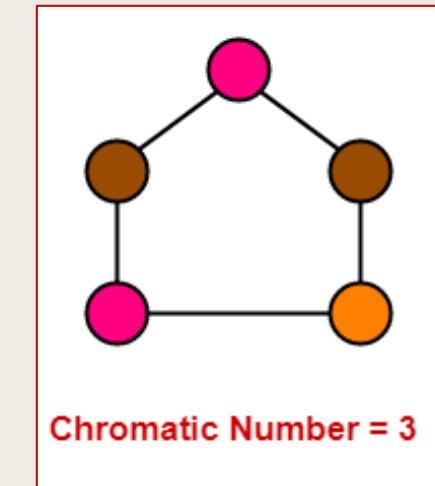
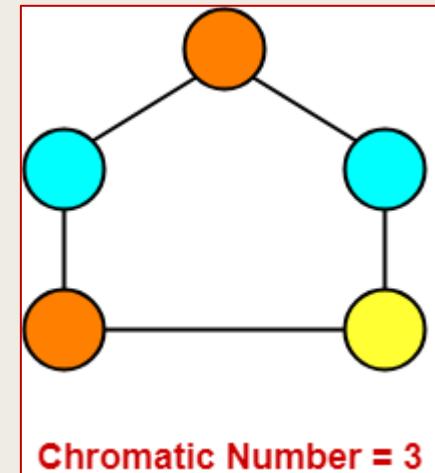


Coloring regions is equivalent to coloring vertices of dual graph.



Backtracking – Graph Coloring

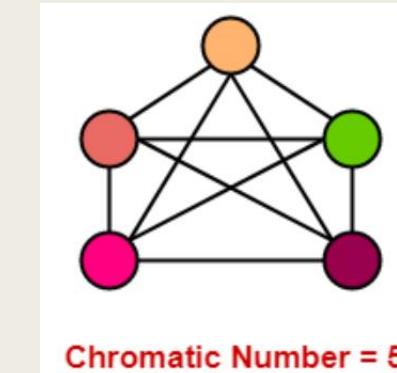
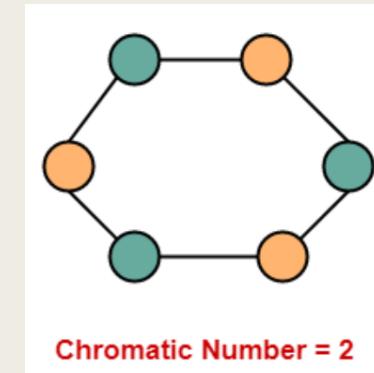
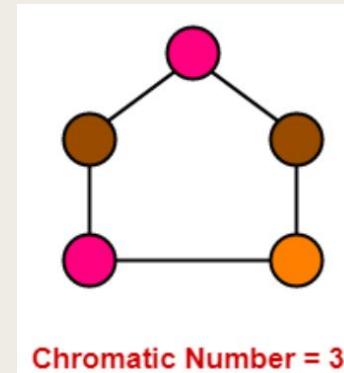
- Given an **undirected graph** and m colours, the task is to find whether it is possible to **colour the vertices** of the graph using m colours with a constraint that **no two adjacent vertices can have same colour**.
- Minimum number of colours required to colour the graph with the said constraint is called **chromatic number**



Backtracking – Graph Coloring

Some interesting facts about chromatic number

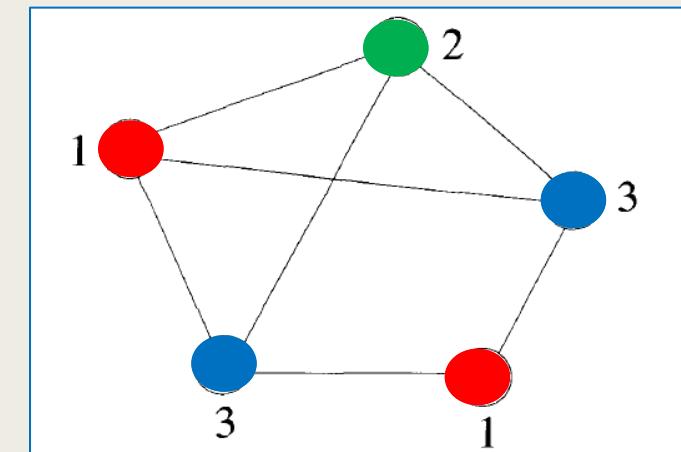
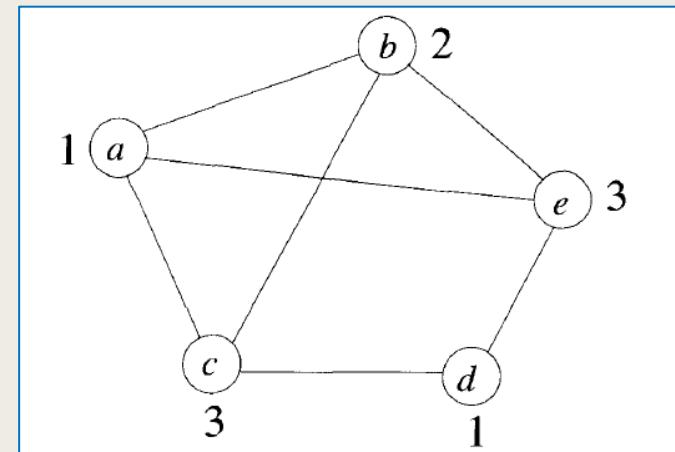
- In a cyclic graph
 - If number of vertices in cycle graph is even, then its chromatic number = 2.
 - If number of vertices in cycle graph is odd, then its chromatic number = 3.
- In a planar graph (a graph that can be drawn in a plane such that none of its edges cross each other.)
 - Chromatic number is ≤ 4
- In a complete graph (each vertex is connected with every other vertex)
 - Chromatic number = number of vertices in that complete graph



Backtracking – Graph Coloring

Problem statement

- Let \mathbf{G} be a graph and m be a given positive integer.
 - Discover whether the nodes of \mathbf{G} can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.
 - This is termed the **m -colorability** decision problem
 - The **m -colorability** optimization problem asks for the smallest integer m for which the graph \mathbf{G} can be colored
 - This integer m is referred to as the **chromatic number** of the graph



Chromatic no. =3

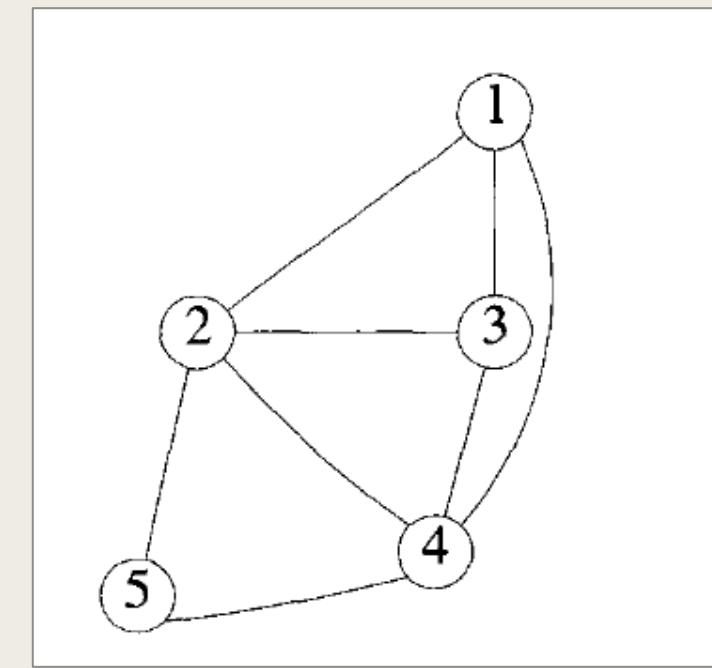
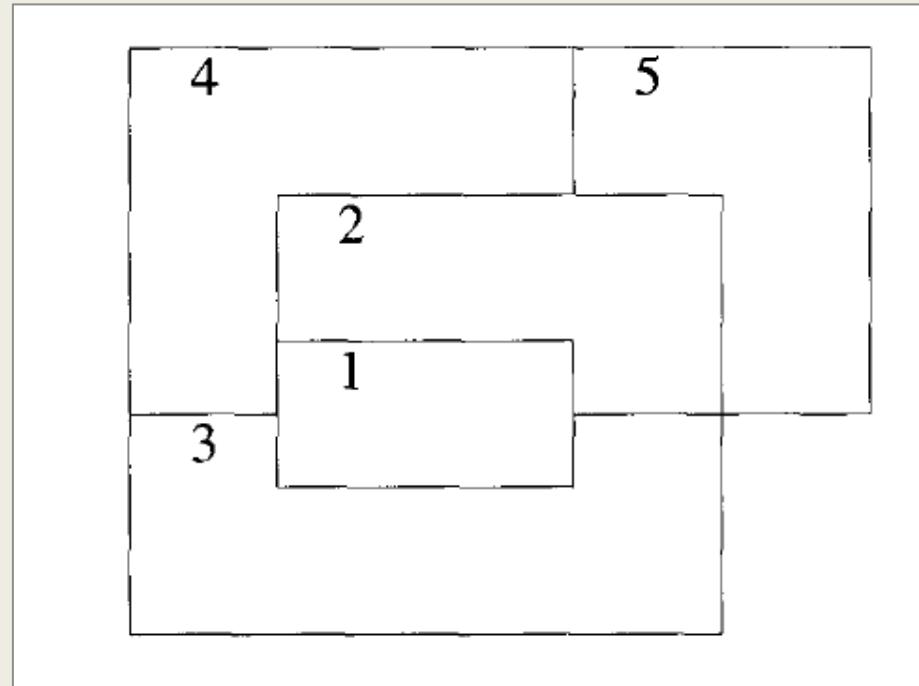
Backtracking – Graph Coloring

4 colour problem for planar graphs

- A graph is said to be planar iff it can be drawn in a plane in such away that no two edges cross each other
- A famous special case of the m colourability decision problem is the 4-color problem for planar graphs
- This problem asks the following question:
 - given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed
- This turns out to be a problem for which graphs are very useful, because a map can easily be transformed into a graph.
- Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge

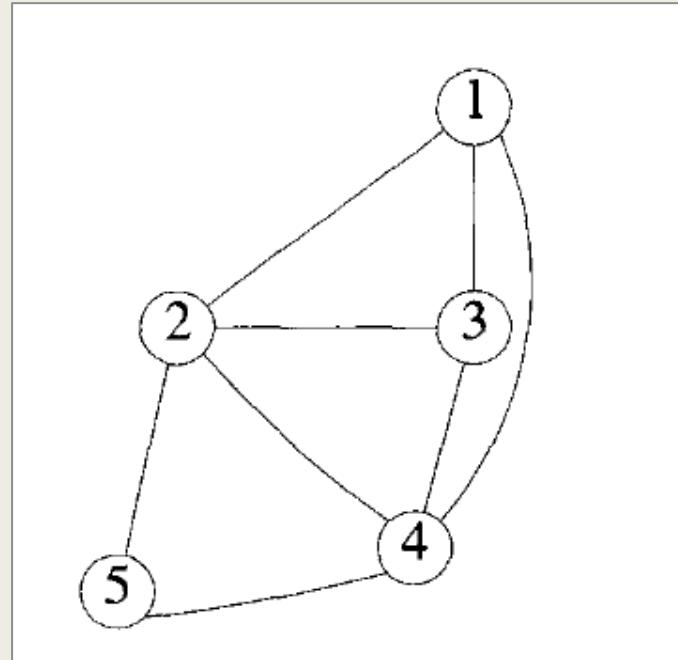
Backtracking – Graph Coloring

- Map and its corresponding graph representation

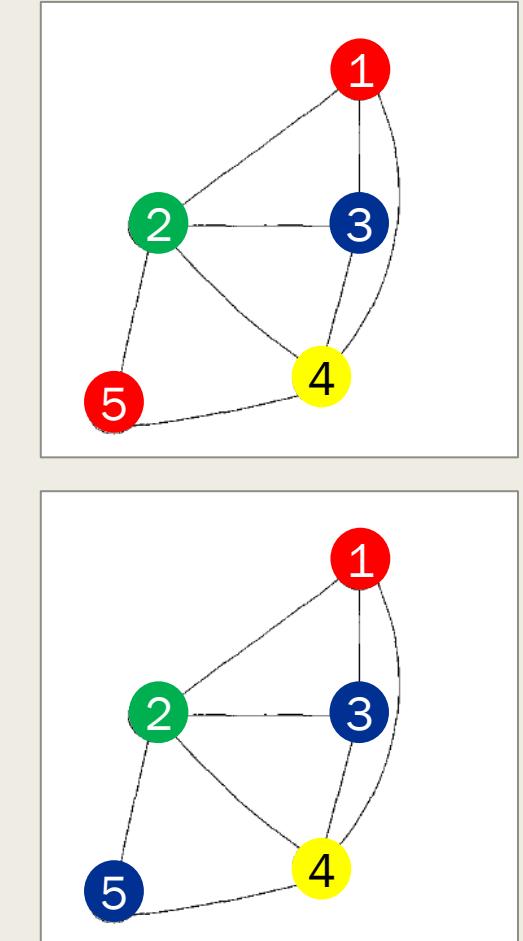
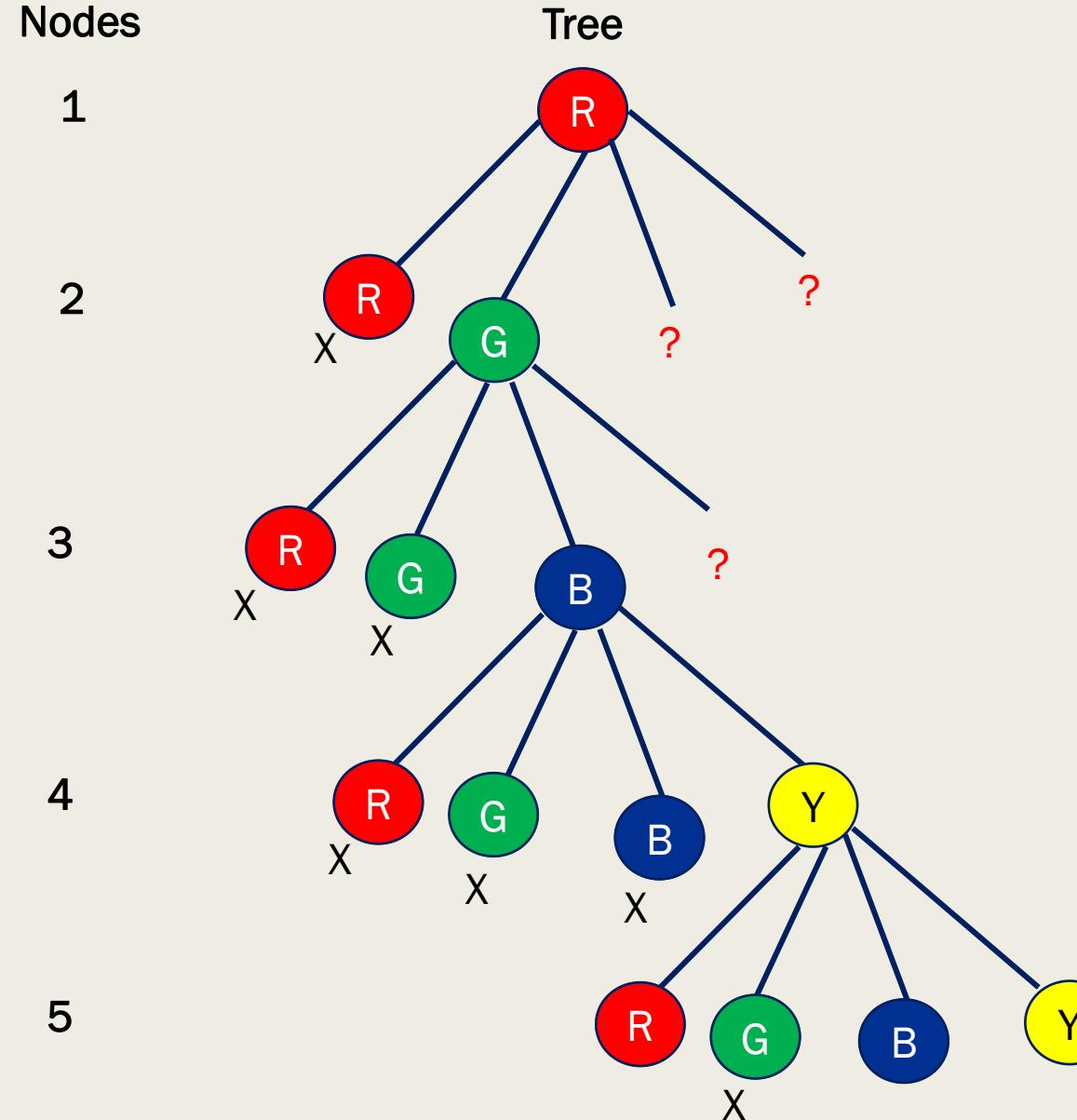
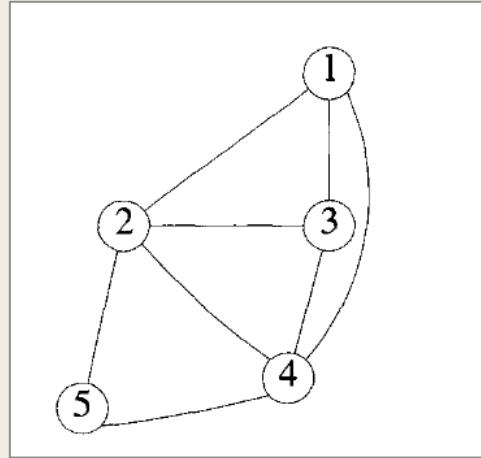


Backtracking – Graph Coloring

- Solve the following instance of graph colouring. Draw the state space tree
-



Backtracking – Graph Coloring



Design and Analysis of Algorithms

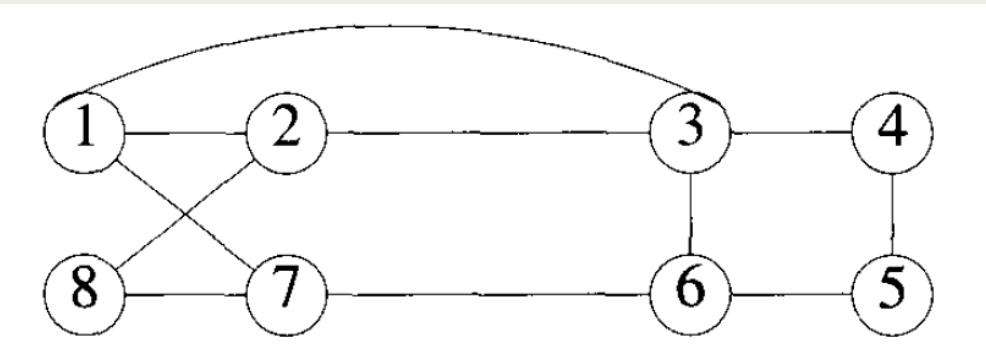
Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

ESTD:2001

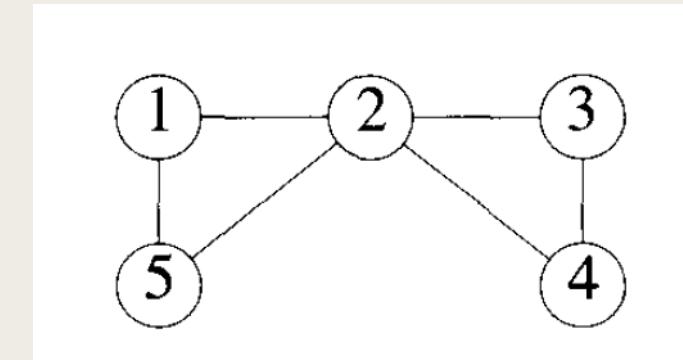
An Institute with a Difference

Backtracking – Hamiltonian Cycle

- Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be a connected graph with n vertices
- A Hamiltonian cycle (Sir William Hamilton) is a round-trip path along n edges of \mathbf{G} that visits every vertex once and returns to its starting position
- In other words if a Hamiltonian cycle begins at some vertex $v_i \in \mathbf{G}$ and the vertices of \mathbf{G} are visited in the order $v_1, v_2, v_3, \dots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in \mathbf{E} , $1 \leq i \leq n$ and v_i are distinct except for v_1 and v_{n+1} which are equal



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



No Hamiltonian cycle !!!!

Backtracking – Hamiltonian Cycle

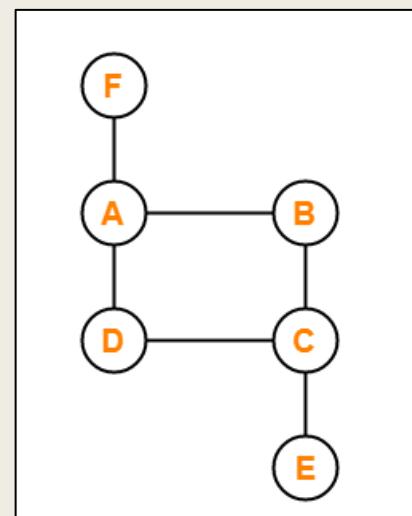
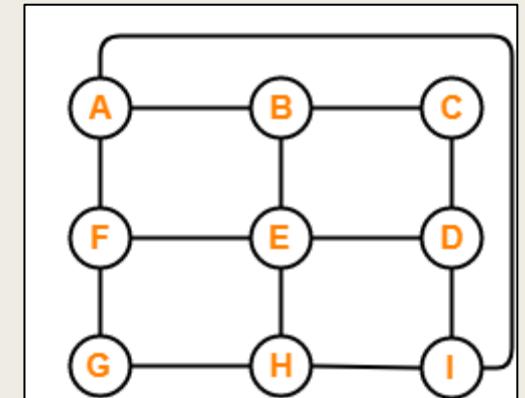
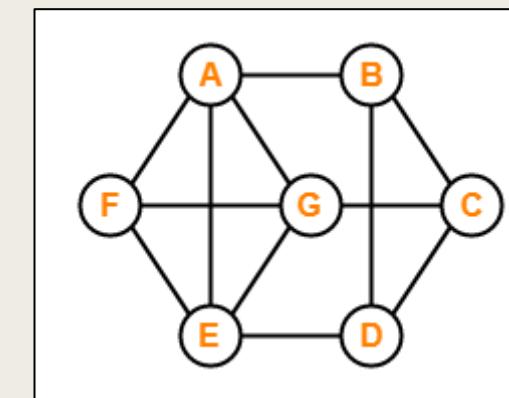
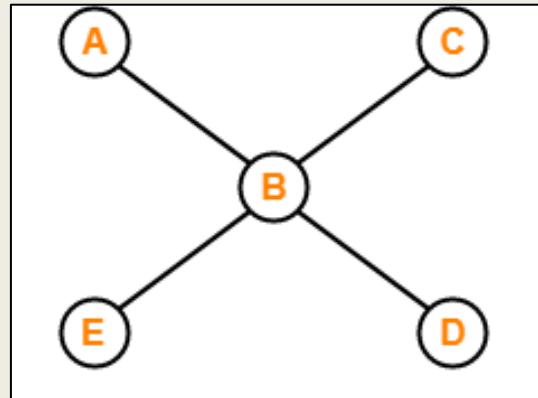
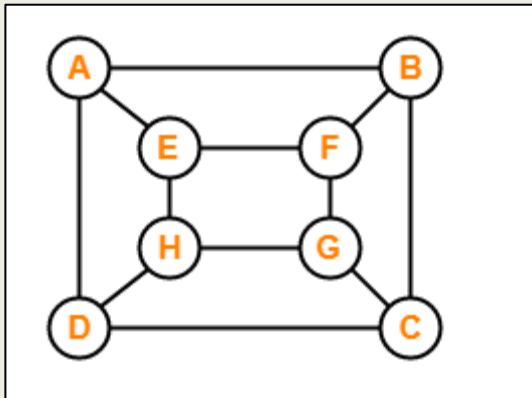


Solution Approach

- Let (x_1, x_2, \dots, x_n) be the solution vector so that x_i represents the i^{th} visited vertex in the proposed cycle
- We need to determine how to compute the set of possible vertices for x_k ,
if x_1, x_2, \dots, x_{k-1} have already been chosen
- If **$k=1$** then x_1 can be any of the **n** vertices
- To avoid printing the same cycles n times, we require that $x_1 = 1$
- If **$1 < k < n$** , then x_k can be any vertex **v** that is distinct from x_1, x_2, \dots, x_{k-1} and v is connected by an edge to x_{k-1}
- The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1

Backtracking – Hamiltonian Cycle

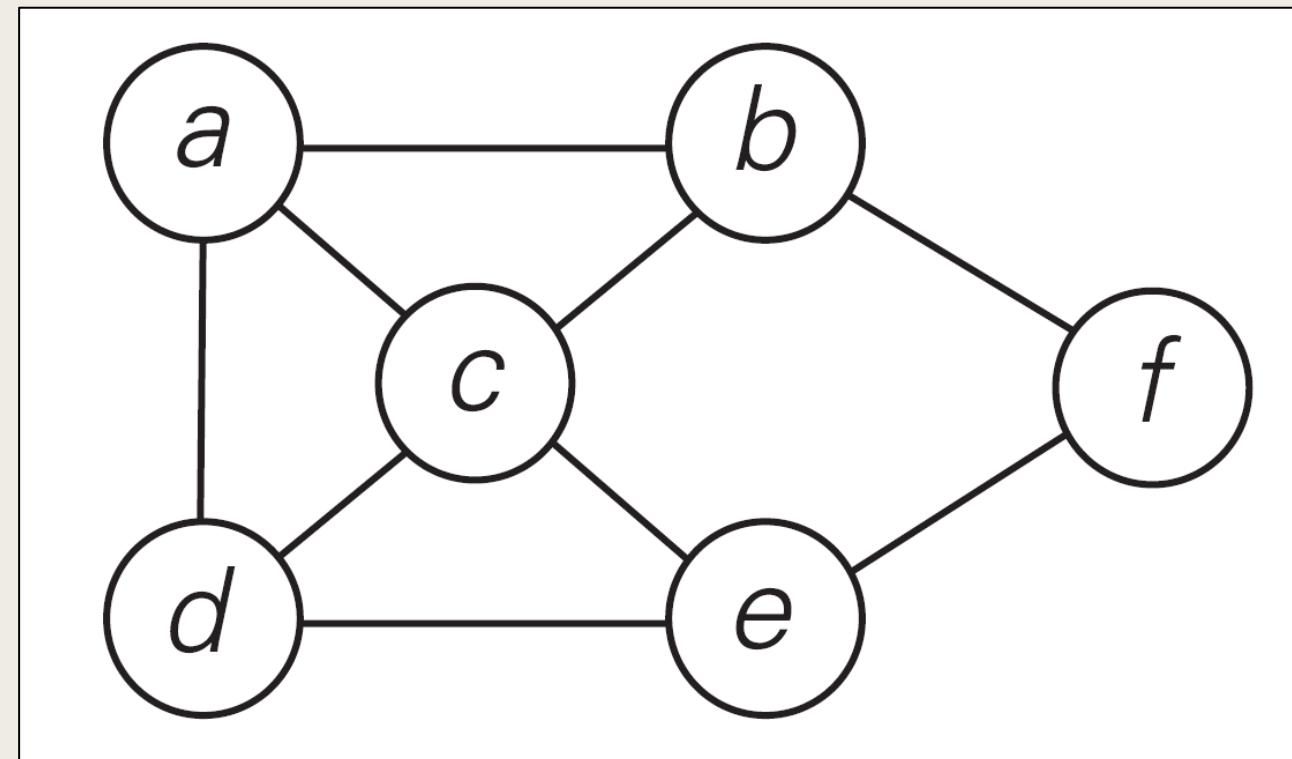
Which of the following are Hamiltonian graphs ?



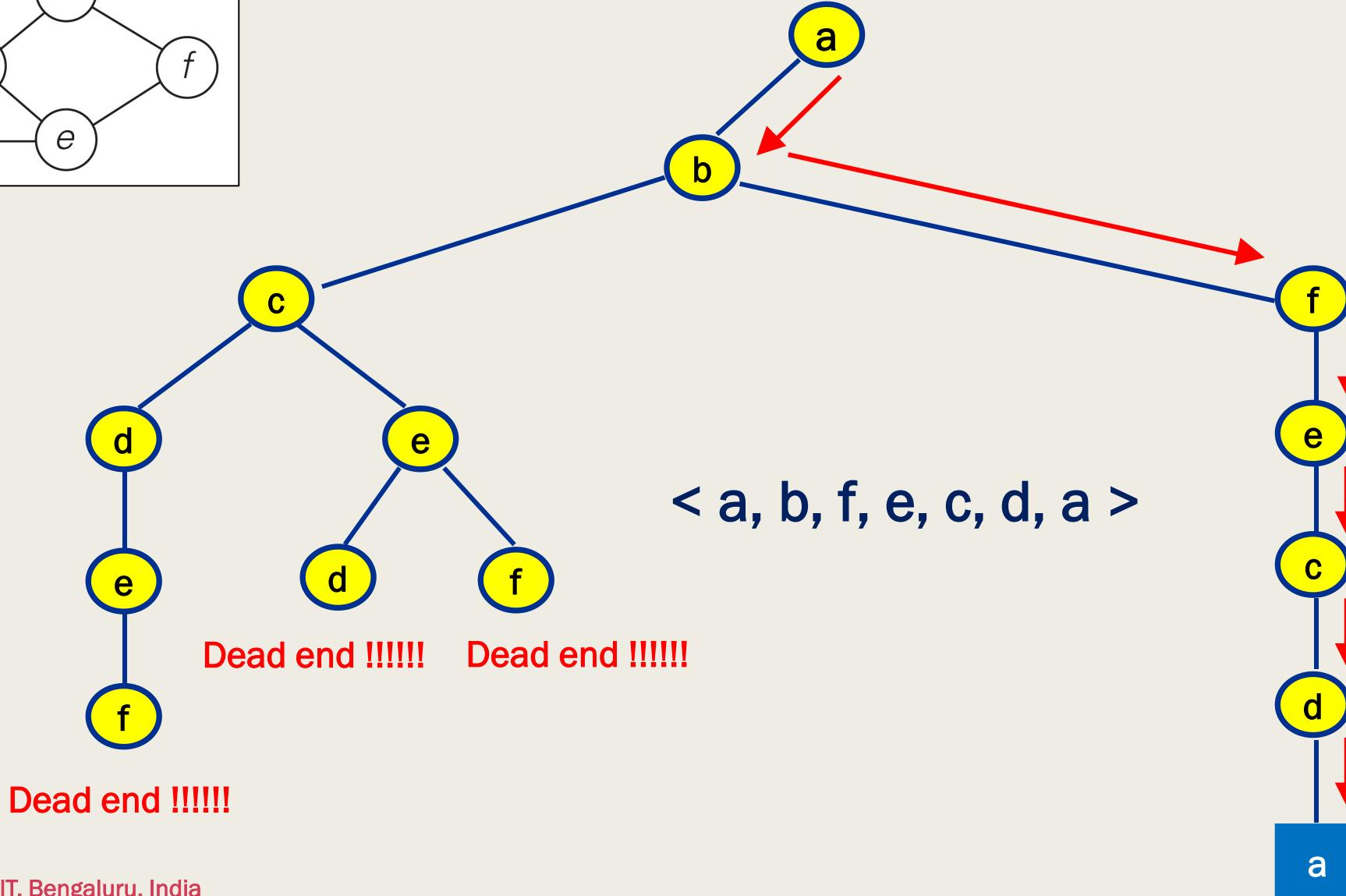
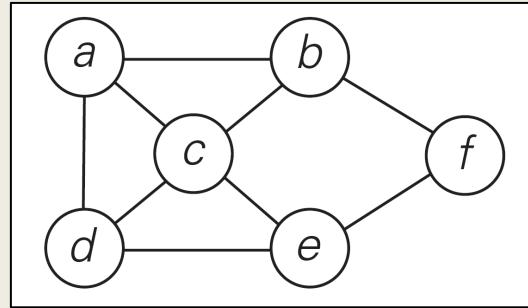
Backtracking – Hamiltonian Cycle

State Space Tree Construction

- Solve the following instance of Hamiltonian cycle and draw the state space tree



Backtracking – Hamiltonian Cycle





BACKTRACKING

Branch and Bound



- The idea of backtracking is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution
- To handle optimization problems, the same concept can be strengthened
- An optimization problem seeks to minimize or maximize some objective function (a tour length, the value of items selected, the cost of an assignment etc.) subjected to constraints
- Recall
 - Feasible solution
 - Optimal Solution

Branch and Bound

- Compared to backtracking, branch-and-bound requires two additional items:
 - a way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
 - the value of the best solution seen so far
- If this information is available, we can compare a node's bound value with the value of the best solution seen so far.
 - If the bound value is not better than the value of the best solution seen so far
 - i.e., not smaller for a minimization problem and not larger for a maximization problem
 - the node is nonpromising and can be terminated (some people say the branch is “pruned”).

Branch and Bound

- A search path is terminated at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:
- The value of the node's bound is not better than the value of the best solution seen so far.
- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—
 - in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Branch and Bound – Assignment Problem

Problem Statement

- Assign n people to n jobs so that the total cost of the assignment is as small as possible
- Assignment problem is specified by an $n \times n$ cost matrix C
 - Select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible

$$C = \begin{bmatrix} & \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \\ \text{person } a & 9 & 2 & 7 & 8 \\ \text{person } b & 6 & 4 & 3 & 7 \\ \text{person } c & 5 & 8 & 1 & 8 \\ \text{person } d & 7 & 6 & 9 & 4 \end{bmatrix}$$

Branch and Bound – Assignment Problem

- How to find a lower bound on the cost of an optimal selection without actually solving the problem
- It is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows
- For the instance here, this sum is $2 + 3 + 1 + 4 = 10$
- It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix);
- it is just a lower bound on the cost of any legitimate selection

Branch and Bound – Assignment Problem

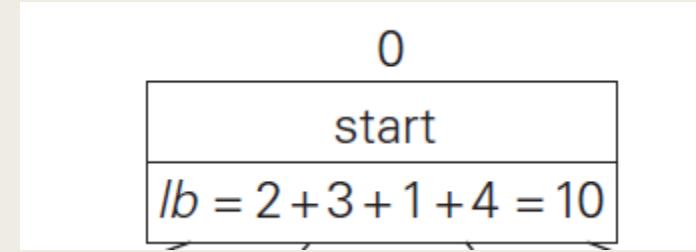
- Solve the following instance of assignment problem and obtain the state space tree

	job 1	job 2	job 3	job 4	
$C =$	9	2	7	8	person a
	6	4	3	7	person b
	5	8	1	8	person c
	7	6	9	4	person d

- First compute the lower bound by finding the sum of the smallest elements in each row

$$LB = 2+3+1+4=10$$

Branch and Bound – Assignment Problem



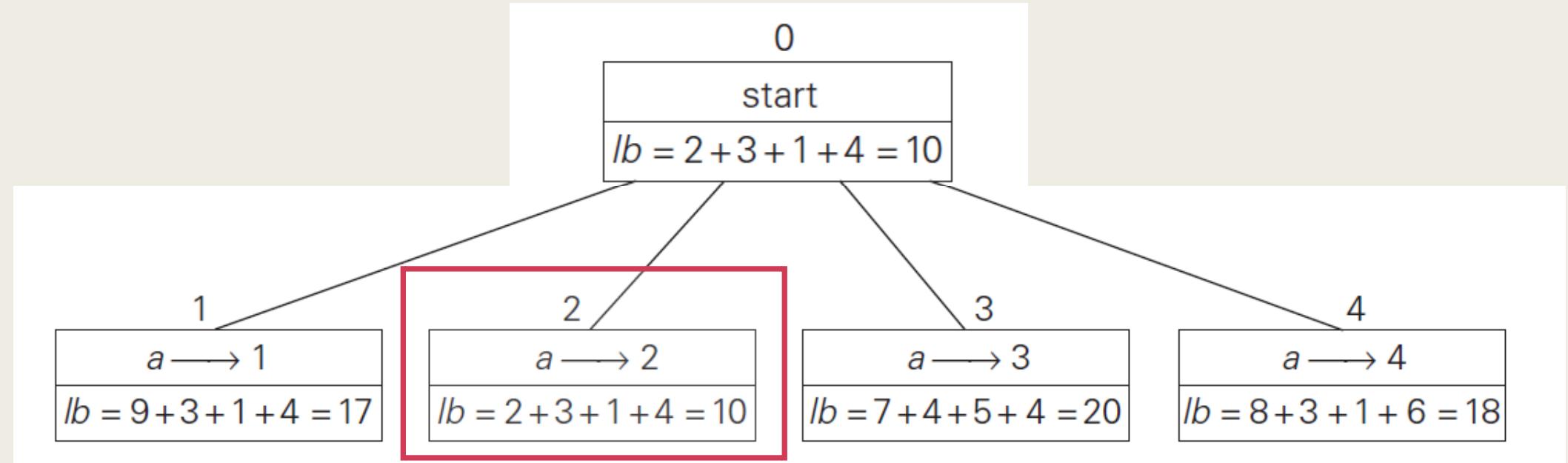
- Consider a job for person a
 - $a \rightarrow 1$ $lb = 9 + 3 + 1 + 4 = 17$
 - $a \rightarrow 2$ $lb = 2 + 3 + 1 + 4 = 10$
 - $a \rightarrow 3$ $lb = 7 + 4 + 5 + 4 = 20$
 - $a \rightarrow 4$ $lb = 8 + 3 + 1 + 6 = 18$
- Since $a \rightarrow 2$, $lb = 10$ is more promising node, we branch from there !!!!

	job 1	job 2	job 3	job 4	
$C =$	9	2	7	8	person <i>a</i>
	6	4	3	7	person <i>b</i>
	5	8	1	8	person <i>c</i>
	7	6	9	4	person <i>d</i>

Branch and Bound – Assignment Problem



ESTD:2001
An Institute with a Difference

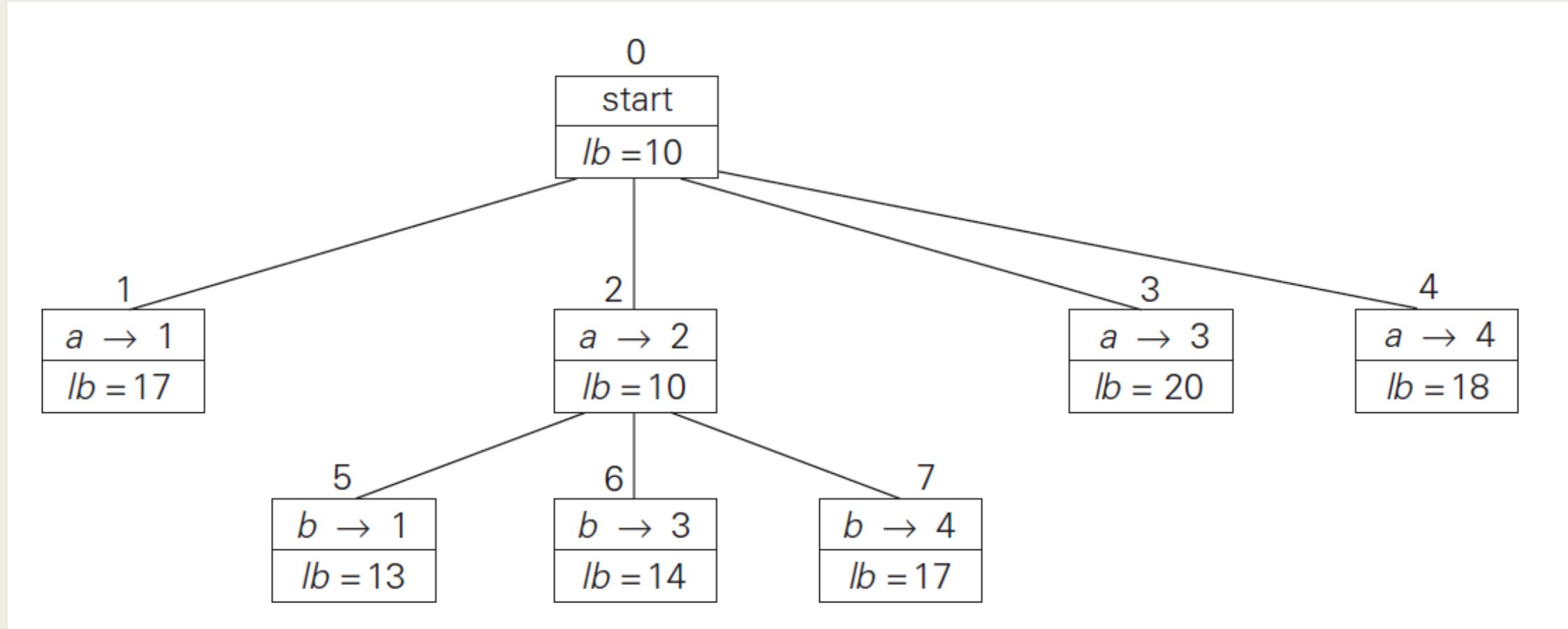


Branch and Bound – Assignment Problem

- Consider a job for person b
 - $b \rightarrow 1 \quad l_b = 2 + (6+1+4) = 13$
 - $b \rightarrow 3 \quad l_b = 2 + (3+5+4) = 14$
 - $b \rightarrow 4 \quad l_b = 2 + (7+1+7) = 17$
- Since $b \rightarrow 1$, $l_b = 13$ is more promising node, we branch from there !!!!

	job 1	job 2	job 3	job 4	
$C =$	9	2	7	8	person <i>a</i>
	6	4	3	7	person <i>b</i>
	5	8	1	8	person <i>c</i>
	7	6	9	4	person <i>d</i>

Branch and Bound – Assignment Problem



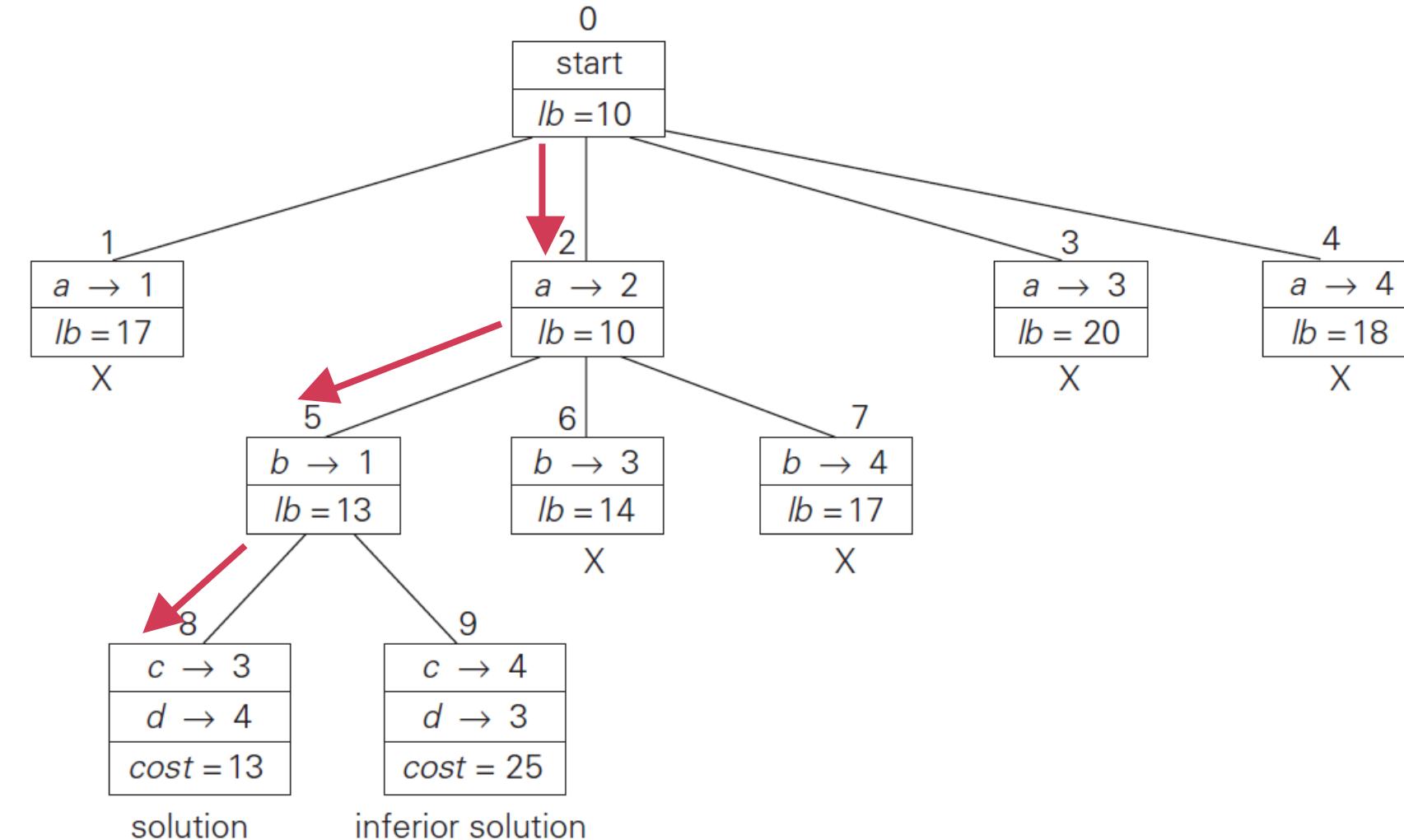
Branch and Bound – Assignment Problem

- Consider job for persons **c** and **d**
 - $c \rightarrow 3, d \rightarrow 4 \quad lb = 2+6+(1+4)=13$
 - $c \rightarrow 4, d \rightarrow 3 \quad lb = 2+6+(8+9)=25$
 - $c \rightarrow 3, d \rightarrow 4$ is more promising
- The final assignment of jobs is as follows

a->2 b->1 c->3 d->4

	job 1	job 2	job 3	job 4	
$C =$	9	2	7	8	person <i>a</i>
	6	4	3	7	person <i>b</i>
	5	8	1	8	person <i>c</i>
	7	6	9	4	person <i>d</i>

Branch and Bound – Assignment Problem



Design and Analysis of Algorithms

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

ESTD:2001

An Institute with a Difference

Branch and Bound – Knapsack Problem



■ Problem statement

- Given n items of known weights w_i and values v_i , $i = 1, 2, \dots, n$, and a knapsack of capacity W , find the most valuable subset of the items that fit in the knapsack.

Solution approach

- It is convenient to order the items of a given instance in descending order by their value-to-weight ratios
- Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n.$$

Branch and Bound – Knapsack Problem



Construction of State Space Tree

- Each node on the i^{th} level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items.
- This particular selection is uniquely determined by the path from the root to the node:
 - a branch going to the left indicates the inclusion of the next item, and
 - a branch going to the right indicates its exclusion
- Record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection.

Branch and Bound – Knapsack Problem

Computation of upper bound

- A simple way to compute the upper bound **ub** is to add to **v**, the total value of the items already selected, the product of the remaining capacity of the knapsack **W – w** and the best per unit payoff among the remaining items.

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$

item	weight	value	value / weight
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

Branch and Bound – Knapsack Problem

- Solve the following instance of knapsack problem using branch and bound method

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

Branch and Bound – Knapsack Problem

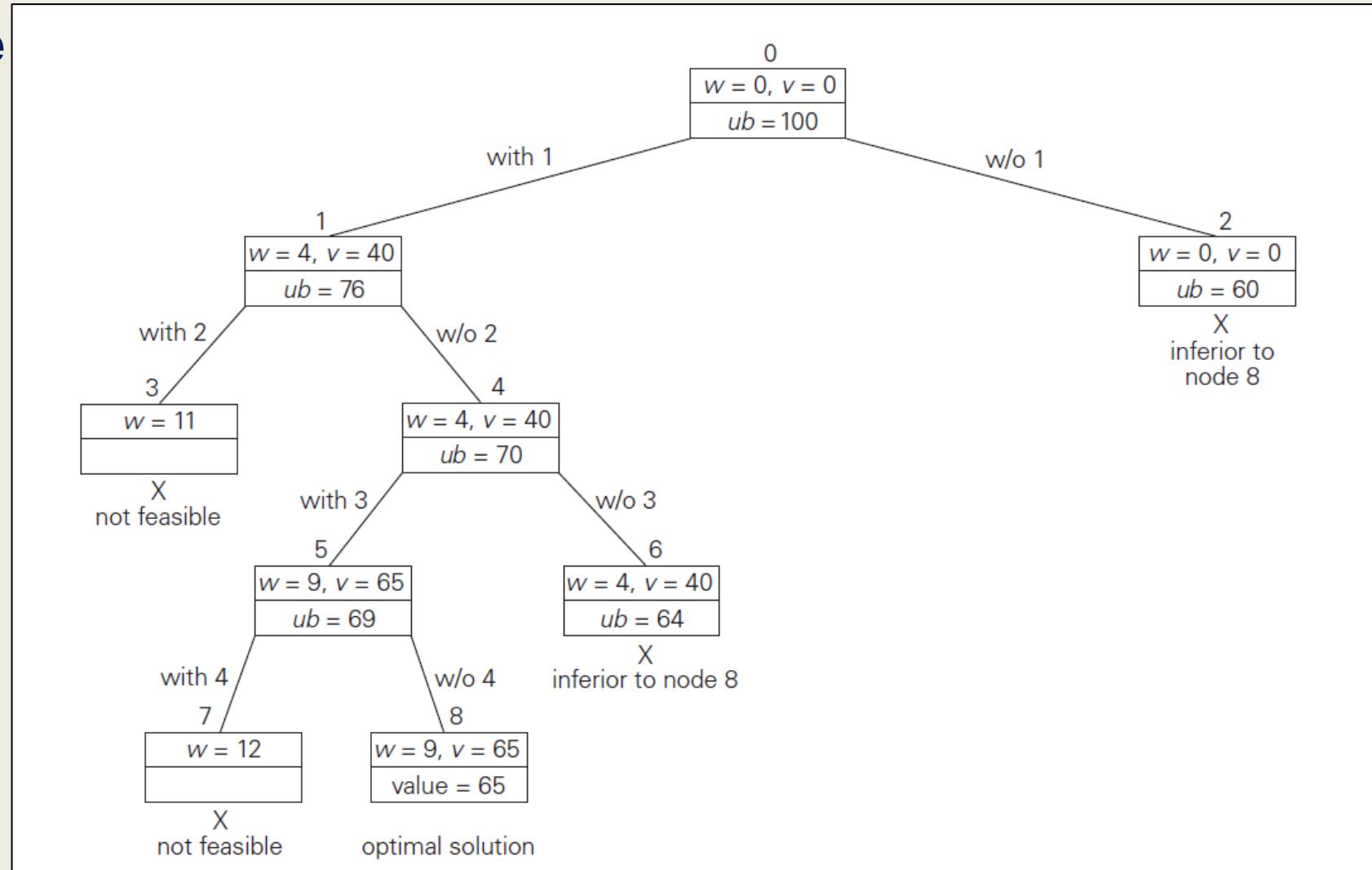


Solution

[Click for the solution](#)

Branch and Bound – Knapsack Problem

State space tree



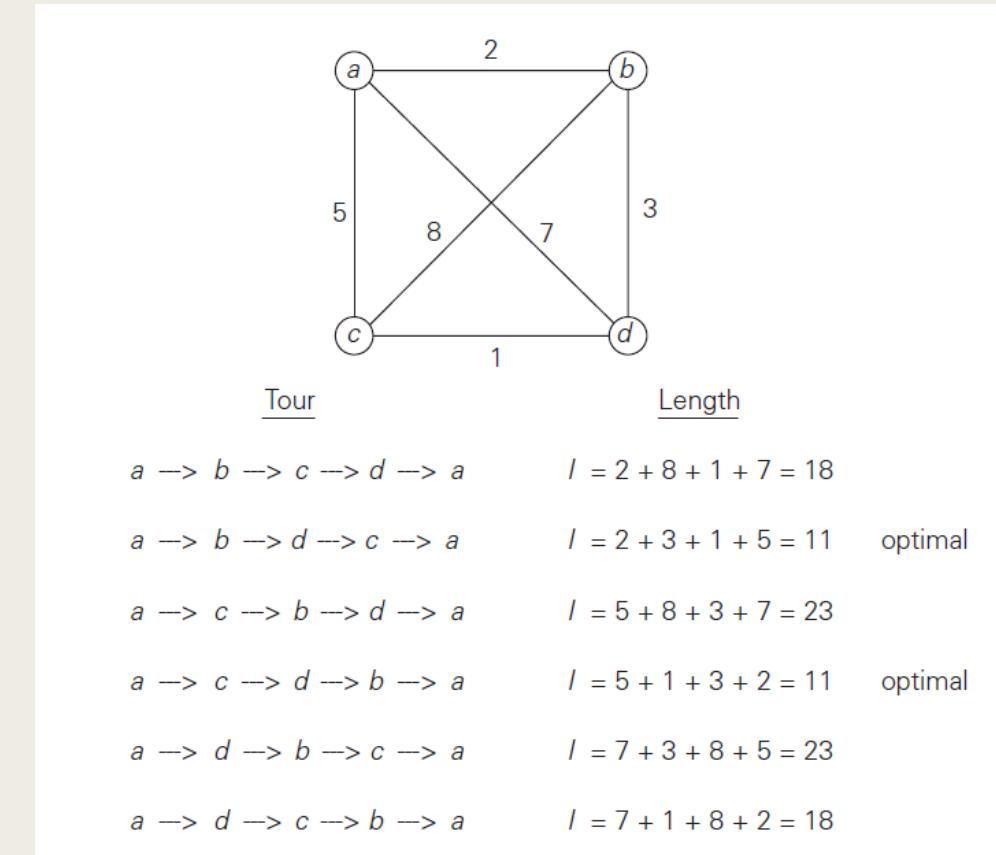
Branch and Bound – TSP

- The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started
- The tour is a sequence of vertices $v_{i0}, v_{i1}, \dots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct.
- All the tours starts and ends at one particular vertex (Hamiltonian circuit)
- Thus, we can get all the tours by generating all the permutations of $n - 1$ intermediate cities, compute the tour lengths, and find the shortest among them.

Branch and Bound – TSP

Observations

- Three pairs of tours that differ only by their direction.
- Hence, we could cut the number of vertex permutations by half.
- for example, choose any two intermediate vertices, say, **b** and **c**, and then consider only permutations in which **b** precedes **c**.
- The total number of permutations needed is still $1/2 (n - 1)!$
- Exhaustive search approach impractical!



Branch and Bound – TSP

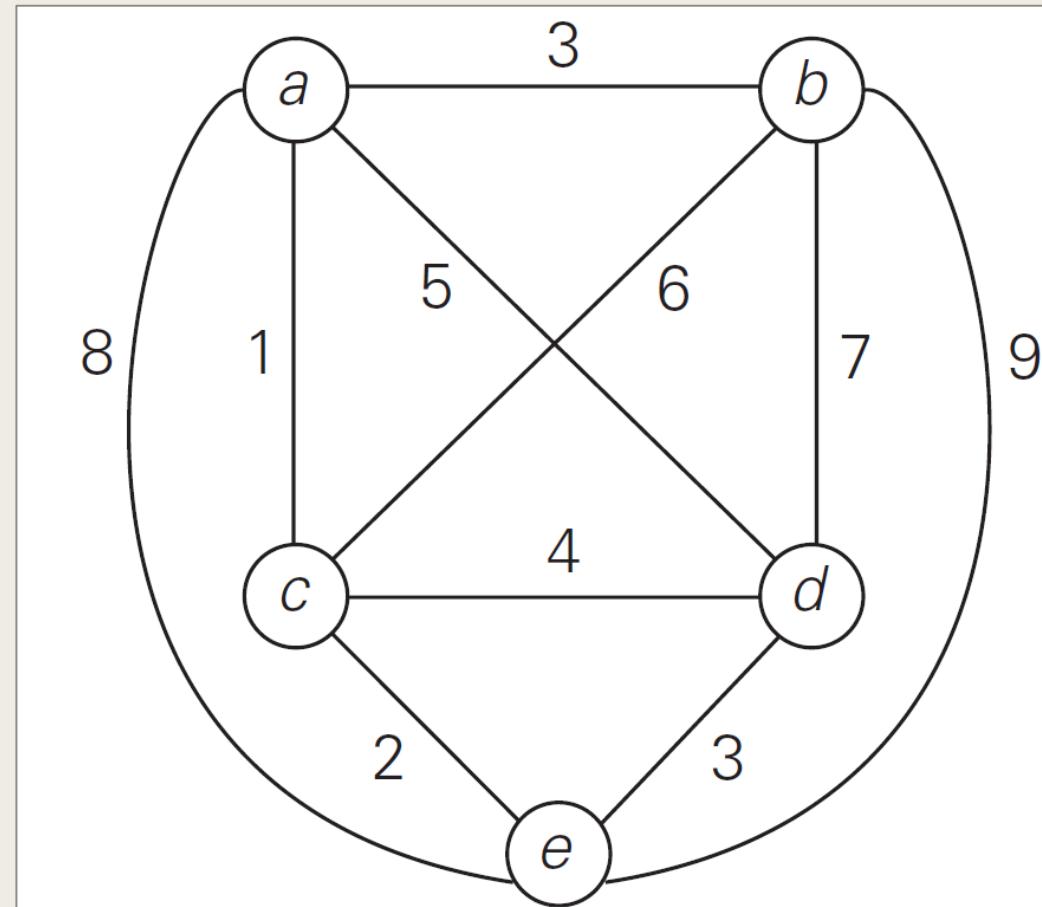
- **Computation of lower bound**
- For each city i , $1 \leq i \leq n$, find the sum s_i of the distances from city i to the two nearest cities;
- compute the sum s of these n numbers, divide the result by 2, and,
- if all the distances are integers, round up the result to the nearest integer:

$$\text{lb} = \lceil s/2 \rceil$$

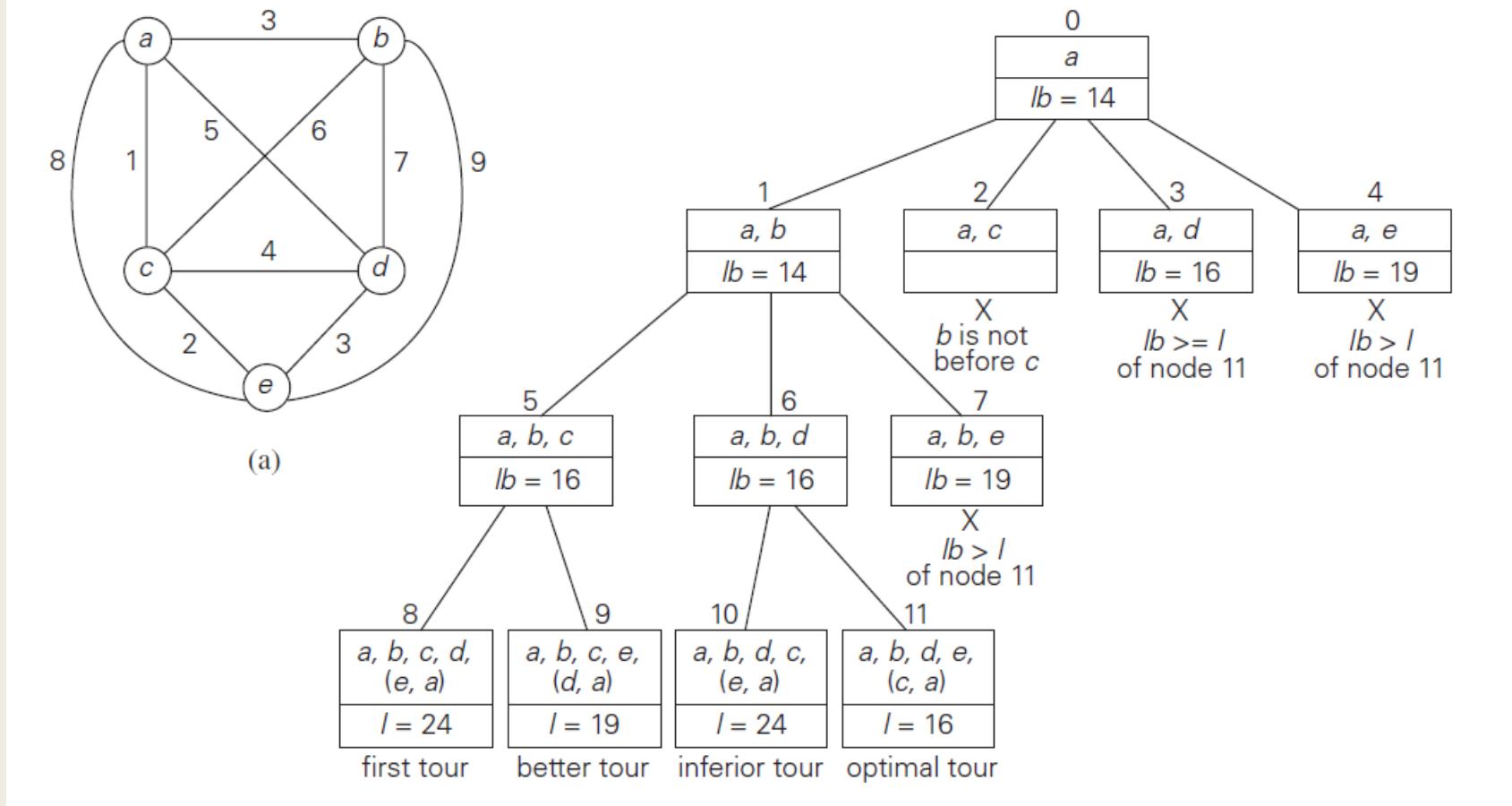
Branch and Bound – TSP

- Solve the following instance of TSP using branch and bound method

Solution



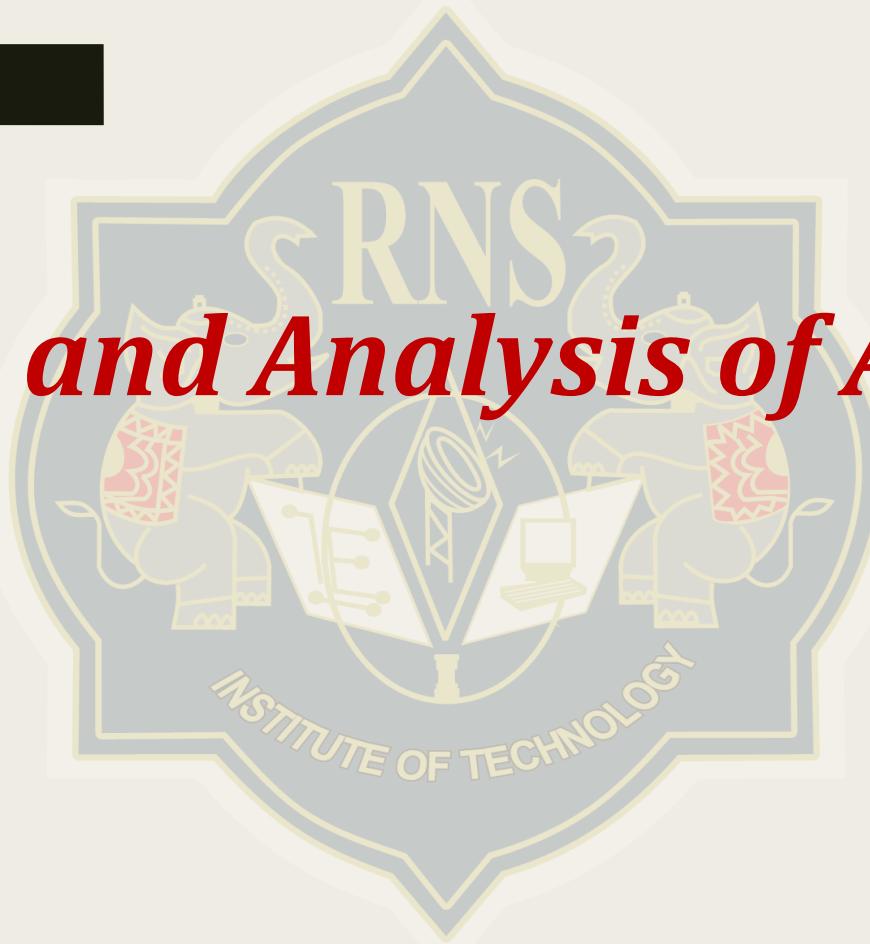
Branch and Bound – TSP





THANK YOU

Design and Analysis of Algorithms



ESTD:2001

An Institute with a Difference

Greedy Method

- It's a straight forward design technique that can be applied to a wide variety of problems
- Coin change problem
 - Let $A_n = \{a_1, a_2, a_3, \dots, a_n\}$ be a finite set of distinct coin types (for example, $a_1 = 5p$, $a_2 = 10p$, $a_3 = 20p$, $a_4 = 25p$, $a_5 = 50p$ and so on.)
 - Assume each a_i is an integer and $a_1 > a_2 > a_3 \dots a_n$.
 - Each type is available in unlimited quantity.
 - The coin-changing problem is to make up an exact amount C using a minimum total number of coins.
 - C is an integer >0 .



Greedy Method

■ Lets take an example

- X goes to a shop to buy an item worth 30p and hands over 1Rs.
- What is the change expected by X? 70p
- Probable set of coins given by shop keeper
 - A={10, 20, 20, 20}
 - B={10, 10, 10, 10 ,10 ,10, 10}
 - C={5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
 - D={25 ,25 ,20}
 - E={50, 20}
 - F= {50, 10, 10}

■ What was the constraint?

- The sum of the coins given by the shopkeeper =70p

■ The subset that satisfies this constraint is called **feasible solution**

■ The feasible solution that maximizes/minimizes the objective function is called an **optimal solution**

Greedy Method

■ Greedy algorithms work in stages

- At each stage, a decision is made regarding whether a particular input is in an optimal solution.
- This is done by considering the inputs in an order determined by some selection procedure.
- If the inclusion of the next input into the partially constructed optimal
- Solution will result in an infeasible solution, then this input is not added to the partial solution, otherwise, it is added

Greedy Method

Control Abstraction

```
Algorithm Greedy(a, n)
// a[1:n] contains the n inputs.
{
    solution:=0;// Initialize the solution.
    for i :=1 to n do
    {
        x :=Select(a);
        if Feasible(solution, x) then
            solution:=Union (solution, x);
    }
    return solution;
}
```

Greedy Method

Knapsack problem

- Given n objects, with each object i having a positive weight w_i and the profit p_i
- A knapsack / bag with a capacity m
- If a fraction x_i , $0 < x_i < 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned
- The objective is to find the set of objects which fills the knapsack and maximizes the total profit
- The total weight of the selected objects can be at most m

Greedy Method

Formal definition of Knapsack problem

$$\text{maximize} \quad \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{subject to} \quad \sum_{1 \leq i \leq n} w_i x_i \leq m$$

$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

$x_i = 1$ when item i is selected and let $x_i = 0$ when item i is not selected.

Greedy Method

Example

Consider the following instance of the knapsack problem

$n = 3, m= 20, (p_1,p_2,p_3)=(25,24,15)$ and $(w_1,w_2,w_3) = (18,15,10)$.

Find all the feasible solutions and hence an optimal solution

First feasible solution

- Consider the objects according to their profit (descending order)
- Object with largest profit is considered first and so on
- Object 1 (highest profit) with weight 18 can be put into the knapsack in its entirety . So $x_1=1$ and $\text{RemCap} = 2$
- Object 2 (second highest profit) with weight 15 is considered but cant be put into the bag in its entirety ($\text{RemCap} < w_2$)
- Hence a fraction $2/15$ is put into the knapsack, $x_2=2/15$, $\text{RemCap} = 0$

Greedy Method

-Remaining objects cant be put into the knapsack

$$\sum_{i=1 \text{ to } 3} w_i x_i = 18 \times 1 + 15 \times \frac{2}{15} + 0 = 20$$

$$\sum_{i=1 \text{ to } 3} p_i x_i = 25 \times 1 + 24 \times \frac{2}{15} + 0 = 28.2$$

The solution vector $(x_1, x_2, x_3) = (1, \frac{2}{15}, 0)$

Greedy Method

Second feasible solution

- Consider the objects according to their weights (increasing order)
- Object with smallest weight is considered first and so on
- Object 3 (smallest weight) with weight 10 can be put into the knapsack in its entirety .
So $x_3=1$ and $\text{RemCap} = 10$
- Object 2 (second smallest weight) with weight 15 is considered but cant be put into the bag in its entirety ($\text{RemCap} < w_2$)
- Hence a fraction $10/15$, i.e. $2/3$ is put into the knapsack, $x_2=2/3$, $\text{RemCap} = 0$
- Remaining objects cant be put into the knapsack

$$\sum_{i=1 \text{ to } 3} w_i x_i = 0 + 15 \times \frac{2}{3} + 10 \times 1 = 20$$

$$\sum_{i=1 \text{ to } 3} p_i x_i = 0 + 24 \times \frac{2}{3} + 15 \times 1 = 31$$

The solution vector $(x_1, x_2, x_3) = (0, \frac{2}{3}, 1)$

Greedy Method

Third feasible solution

- Consider the objects according to the ratio p_i/w_i (decreasing order)
- $p_1/w_1 = 1.38 \quad p_2/w_2 = 1.6 \quad p_3/w_3 = 1.5$
- Object with highest ratio is considered first and so on
- Object 2 (highest ratio) with weight 15 can be put into the knapsack in its entirety . So $x_2=1$ and $\text{RemCap} = 5$
- Object 3 (second highest ratio) with weight 10 is considered but cant be put into the bag in its entirety ($\text{RemCap} < w_3$)
- Hence a fraction 5/10, i.e. 1/2 is put into the knapsack, $x_3=1/2$, $\text{RemCap} = 0$
- Remaining objects cant be put into the knapsack

$$\sum_{i=1 \text{ to } 3} w_i x_i = 0 + 15 \times 1 + 10 \times \frac{1}{2} = 20$$

$$\sum_{i=1 \text{ to } 3} p_i x_i = 0 + 24 \times 1 + 15 \times \frac{1}{2} = 31.5$$

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

Greedy Method

- First feasible solution =28.2
- Second feasible solution =31
- Third feasible solution =31.5
- Since the objective function is to maximize the profit,
the optimal solution is
 - Profit =31.5
 - The solution vector $(x_1, x_2, x_3) = (0, 1, \frac{1}{2})$

Greedy Method

Homework

Consider the following instance of the knapsack problems.

- Find all the feasible solutions and hence an optimal solution
 $n = 3, m= 20, (p_1, p_2, p_3) = (30, 21, 18)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.
- Find all the feasible solutions and hence an optimal solution
 $N=7, m=15, p_i=(10,5,15,7,6,18,3)$ and $w_i=(2,3,5,7,1,4,1)$

Greedy Method

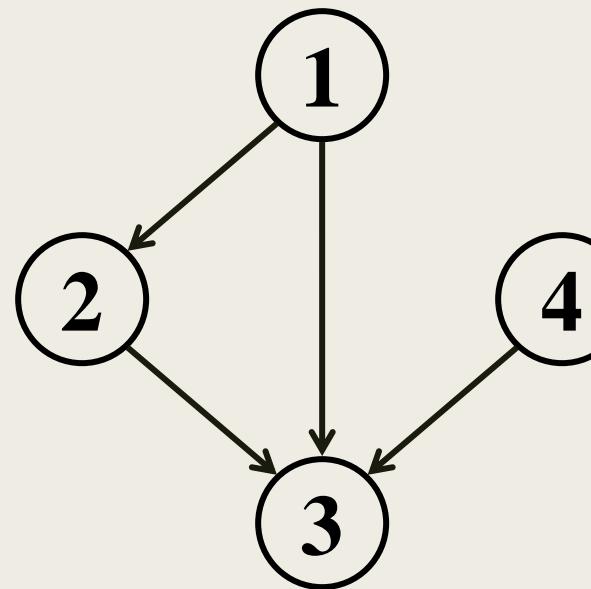
ALGORITHM Greedy_Knapsack(m,n)

```
{  
for( i = 1 to n ) do  x[ i ] = 0.0  
u=m  
for( i = 1 to n ) do  
{  
    if( w[ i ]>u ) then break  
    x[ i ] = 1.0  
    u=u-w[ i ]  
}  
if( i <= n ) then  
x[ i ] = u/w[ i ]  
}
```

Introduction to Graphs

■ A graph $G = (V, E)$

- $V = \text{set of vertices}$
- $E = \text{set of edges} = \text{subset of } V \times V$
- $\text{Thus } |E| = O(|V|^2)$



Vertices: {1, 2, 3, 4}

Edges: {(1, 2), (2, 3), (1, 3), (4, 3)}

Graph – Variations

■ Directed / undirected:

-In an undirected graph:

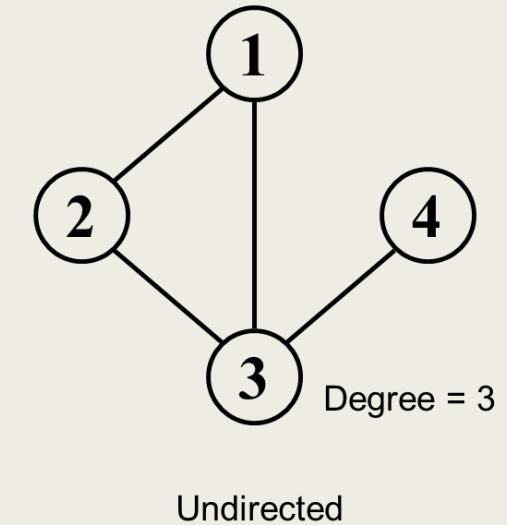
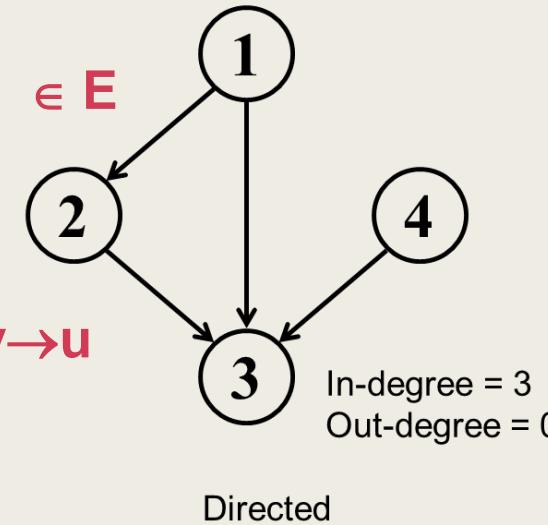
- Edge $(u,v) \in E$ implies edge $(v,u) \in E$
- Road networks between cities

-In a directed graph:

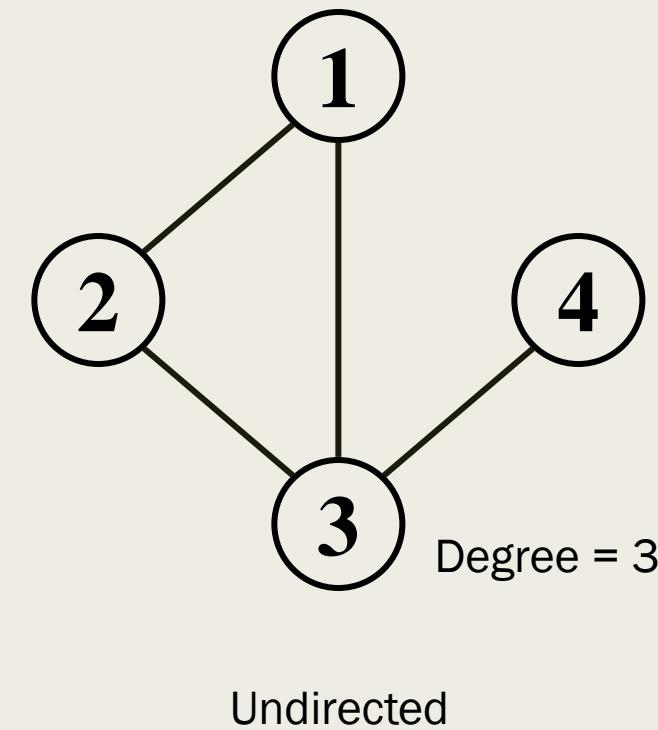
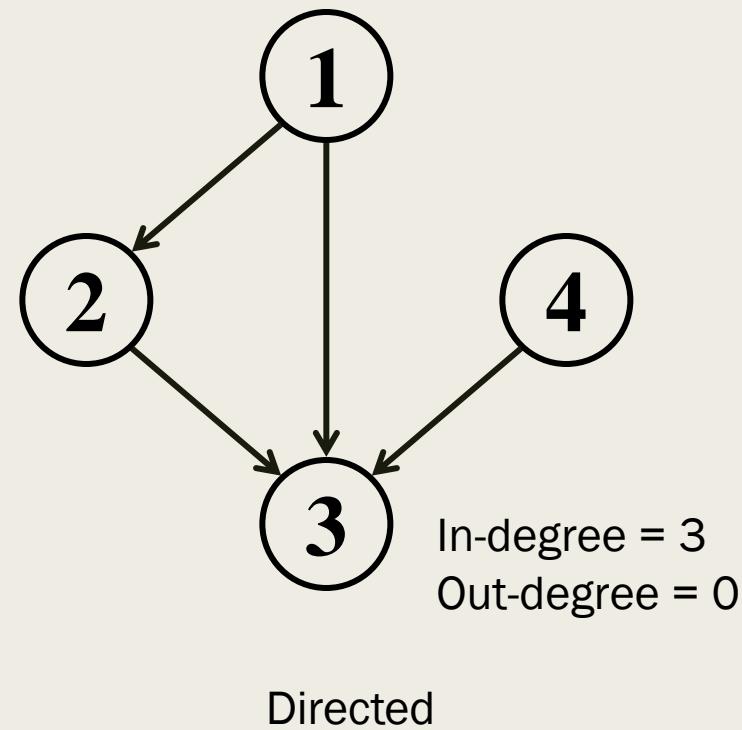
- Edge (u,v) : $u \rightarrow v$ does not imply $v \rightarrow u$
- Street networks in downtown

-Degree of vertex v:

- The number of edges adjacency to v
- For directed graph, there are in-degree and out-degree



Graph – Variations

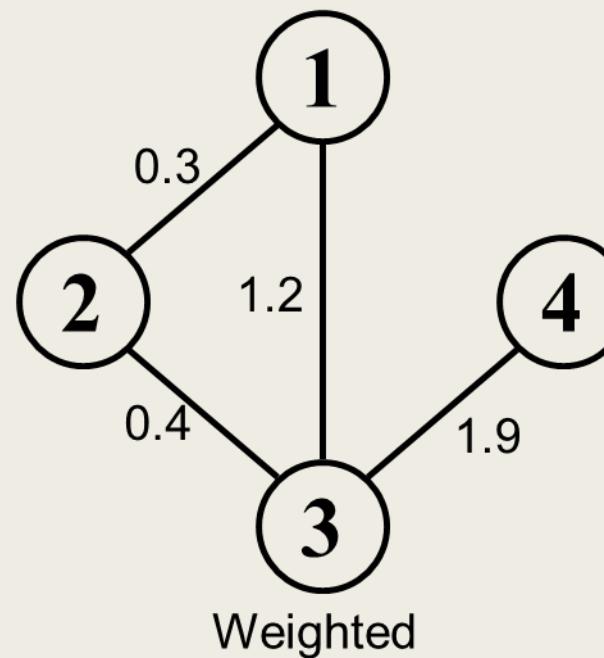
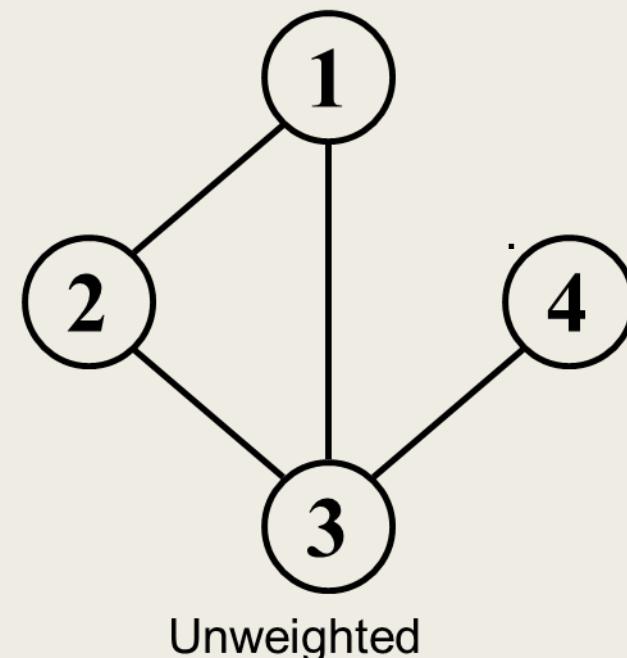


Graph – Variations

■ Weighted / unweighted:

- In a weighted graph, each edge or vertex has an associated weight (numerical value)

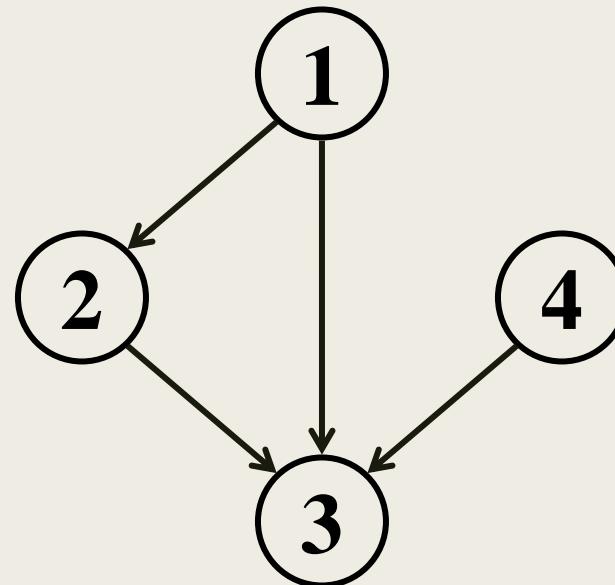
- E.g., a road map: edges might be weighted w/ distance



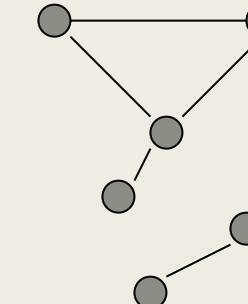
Graph – Variations

■ Connected / disconnected:

- A **connected graph** has a path from every vertex to every other
- A **directed graph is strongly connected if there is a directed path between any two vertices**



Connected but not
strongly connected



Graph – Variations

■ Dense / sparse:

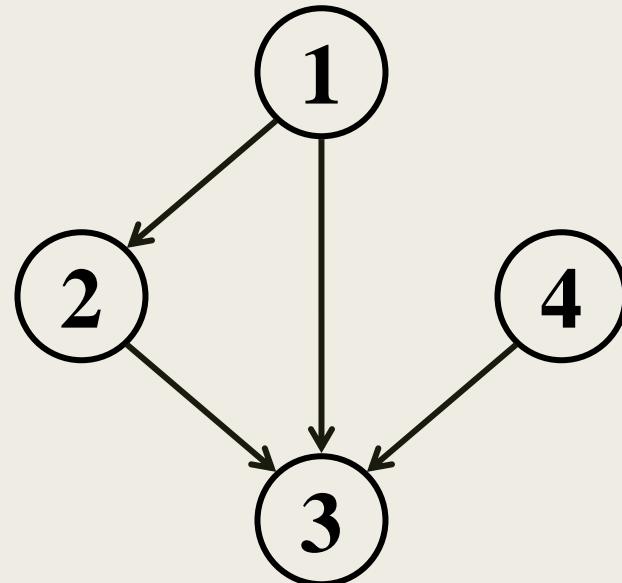
- *Graphs are sparse when the number of edges is linear to the number of vertices*
 - $|E| \in O(|V|)$
- *Graphs are dense when the number of edges is quadratic to the number of vertices*
 - $|E| \in O(|V|^2)$
- *Most graphs of interest are sparse*
- *If you know you are dealing with dense or sparse graphs, different data structures may make sense*

Representing Graphs

- Assume $V = \{1, 2, \dots, n\}$
- An adjacency matrix represents the graph as a $n \times n$ matrix A :
 - $A[i, j] = 1$ if edge $(i, j) \in E$
 - $A[i, j] = 0$ if edge $(i, j) \notin E$
- For weighted graph
 - $A[i, j] = w_{ij}$ if edge $(i, j) \in E$
 - $A[i, j] = 0$ if edge $(i, j) \notin E$
- For undirected graph
 - **Matrix is symmetric:** $A[i, j] = A[j, i]$

Graphs: Adjacency Matrix

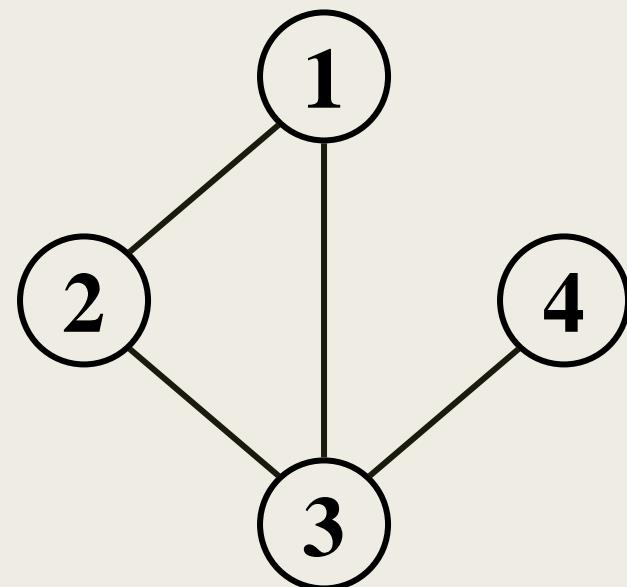
■ Example:



A	1	2	3	4
1				
2				
3				??
4				

Graphs: Adjacency Matrix

■ Example

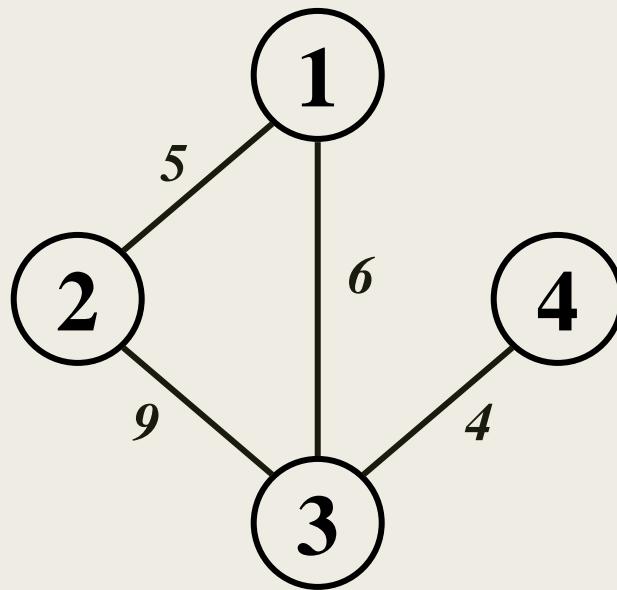


Undirected graph

A	1	2	3	4
1	0	1	1	999
2	1	0	1	0
3	1	1	0	1
4	999	0	1	0

Graphs: Adjacency Matrix

■ Example

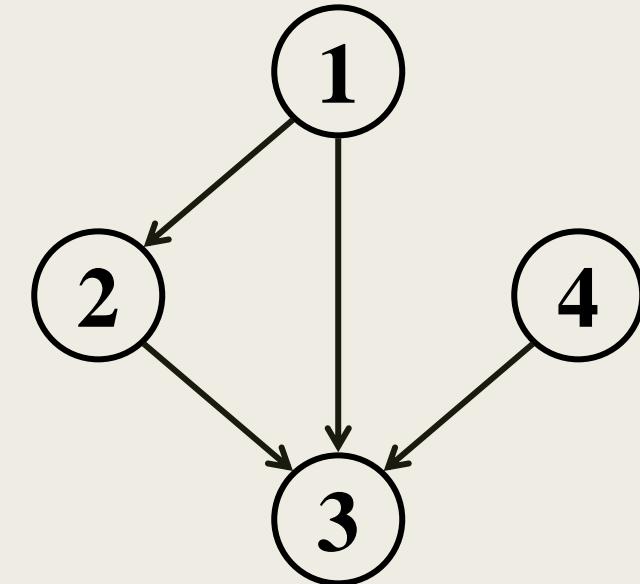


Weighted graph

A	1	2	3	4
1	0	5	6	999
2	5	0	9	0
3	6	9	0	4
4	99	0	4	0
	9			

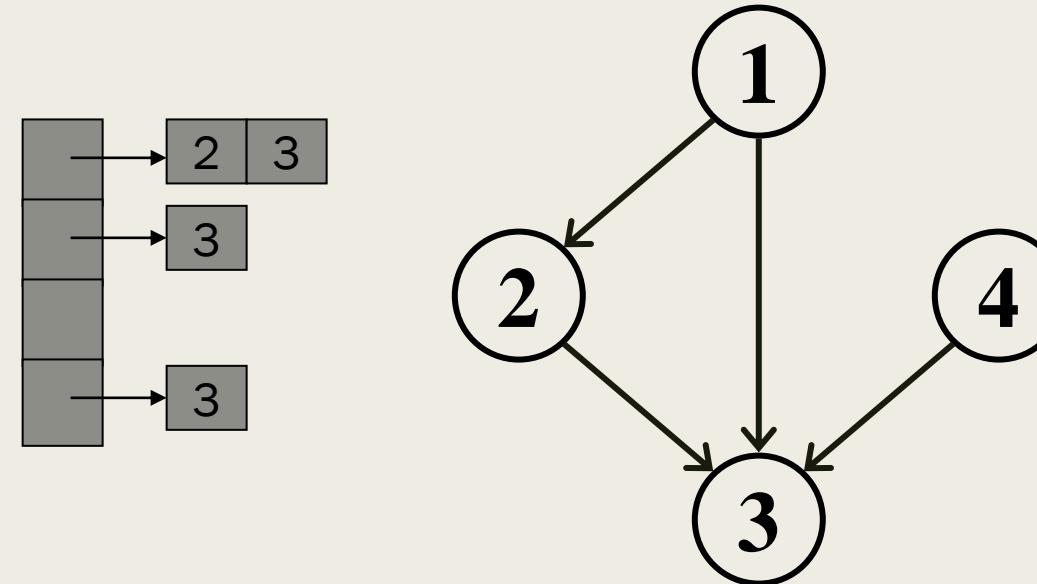
Graphs: Adjacency List

- **Adjacency list:** for each vertex $v \in V$, store a list of vertices adjacent to v
- **Example:**
 - $\text{Adj}[1] = \{2,3\}$
 - $\text{Adj}[2] = \{3\}$
 - $\text{Adj}[3] = \{\}$
 - $\text{Adj}[4] = \{3\}$
- **Variation:** can also keep a list of edges coming into vertex



Graphs: Adjacency List

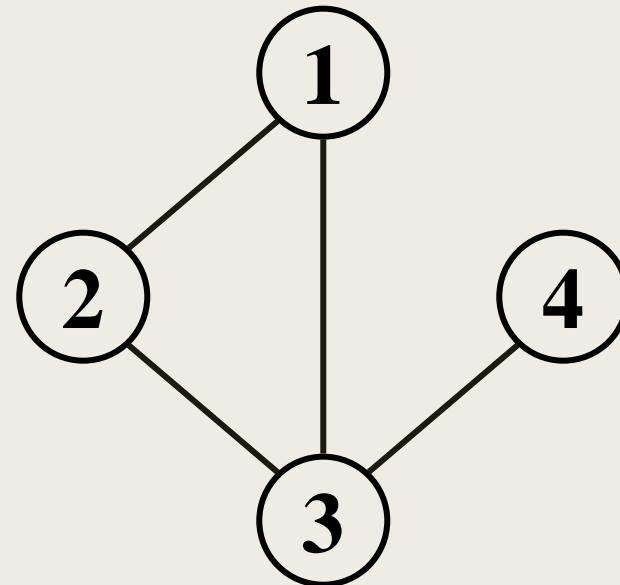
■ Adjacency list



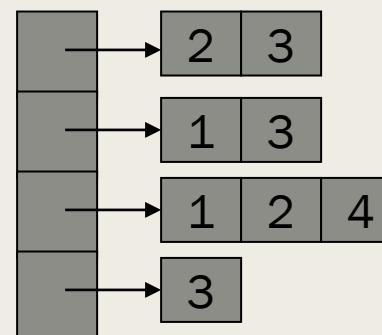
How much storage does the adjacency list require?
A: $O(V+E)$

Graphs: Adjacency List

■ Undirected graph

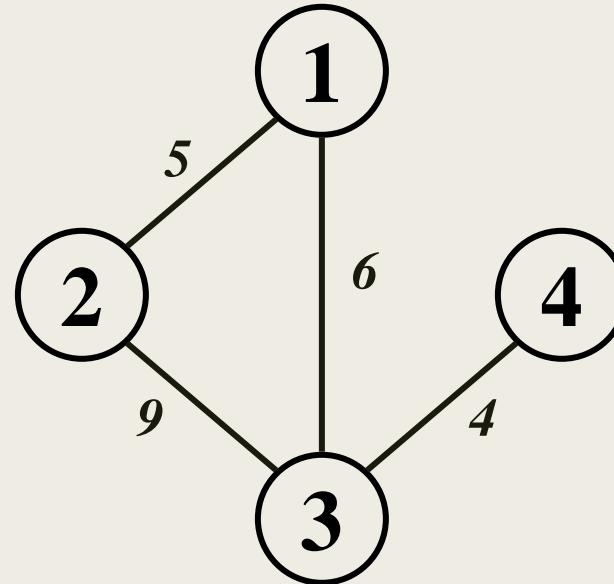


A	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

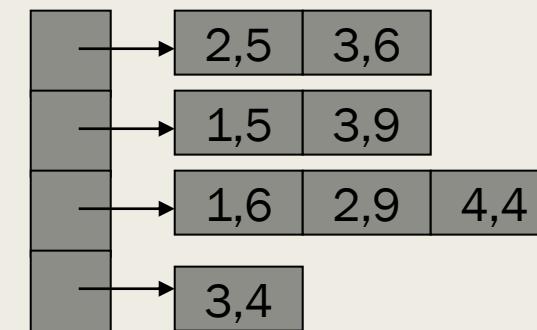


Graphs: Adjacency List

- Weighted graph



A	1	2	3	4
1	0	5	6	0
2	5	0	9	0
3	6	9	0	4
4	0	0	4	0



Trade off between two representations

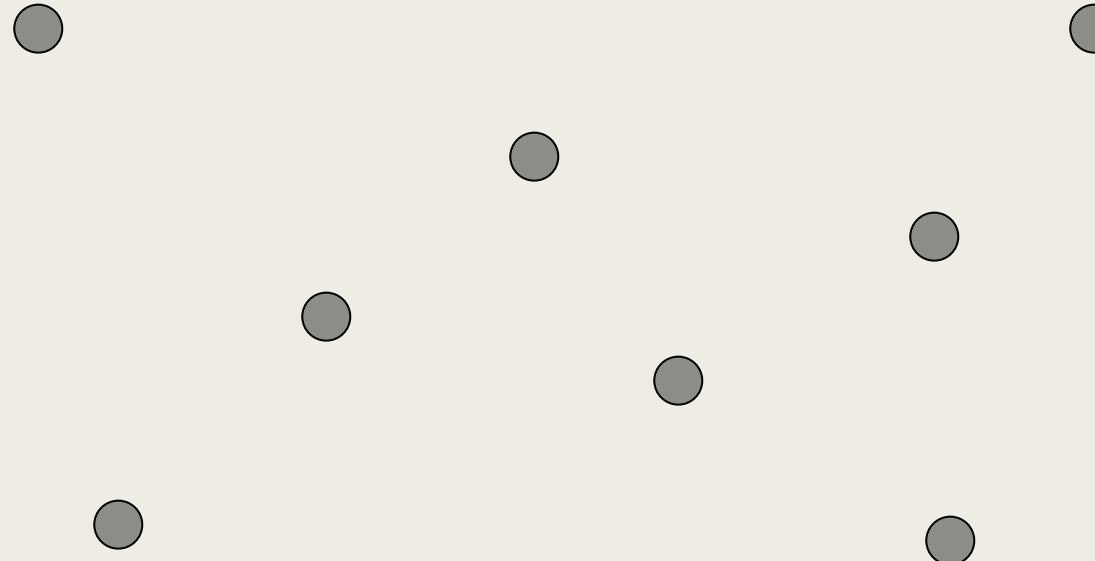
- $|V| = n, |E| = m$

	Adj Matrix	Adj List
test $(u, v) \in E$	$\Theta(1)$	$O(n)$
Degree(u)	$\Theta(n)$	$O(n)$
Memory	$\Theta(n^2)$	$\Theta(n+m)$
Edge insertion	$\Theta(1)$	$\Theta(1)$
Edge deletion	$\Theta(1)$	$O(n)$
Graph traversal	$\Theta(n^2)$	$\Theta(n+m)$

Both representations are very useful and have different properties, although adjacency lists are probably better for more problems

Problems

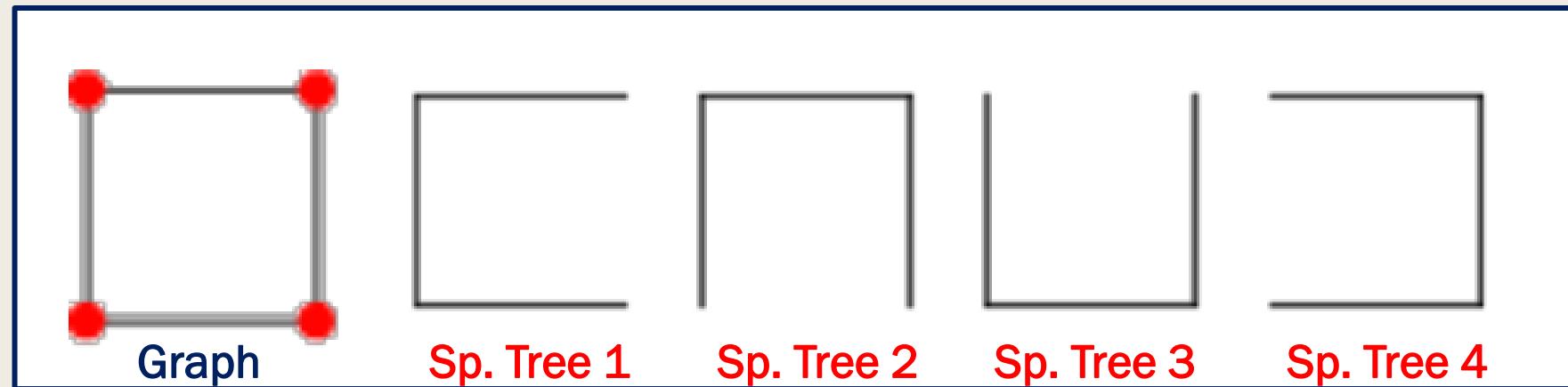
- Given a set of cities, how to construct minimum length of highways to connect the cities so that there are paths between every two cities?



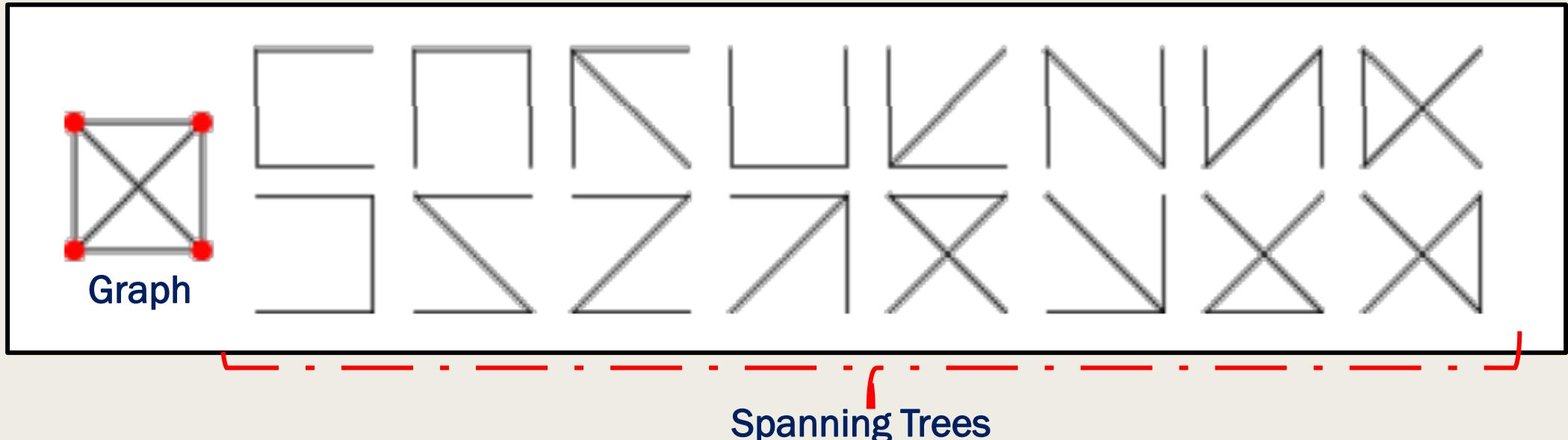
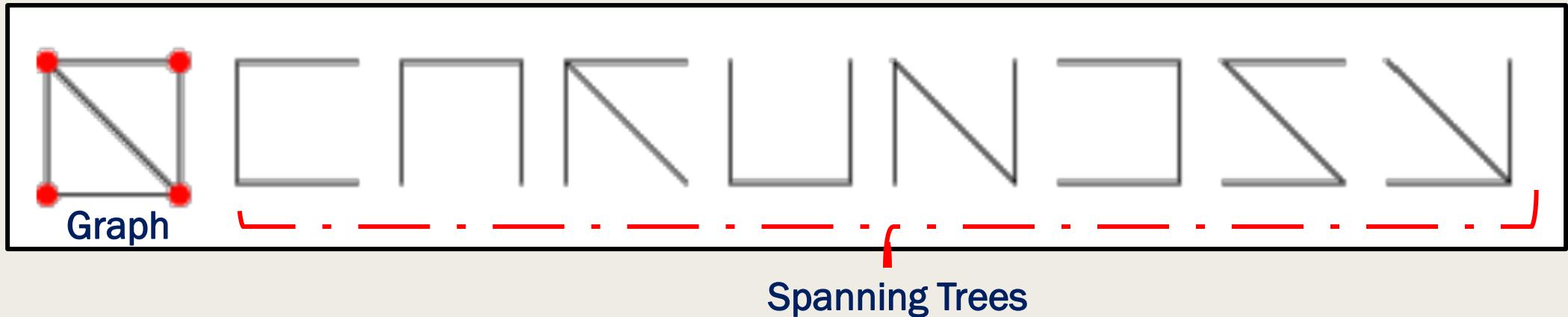
Spanning Tree

Definition

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

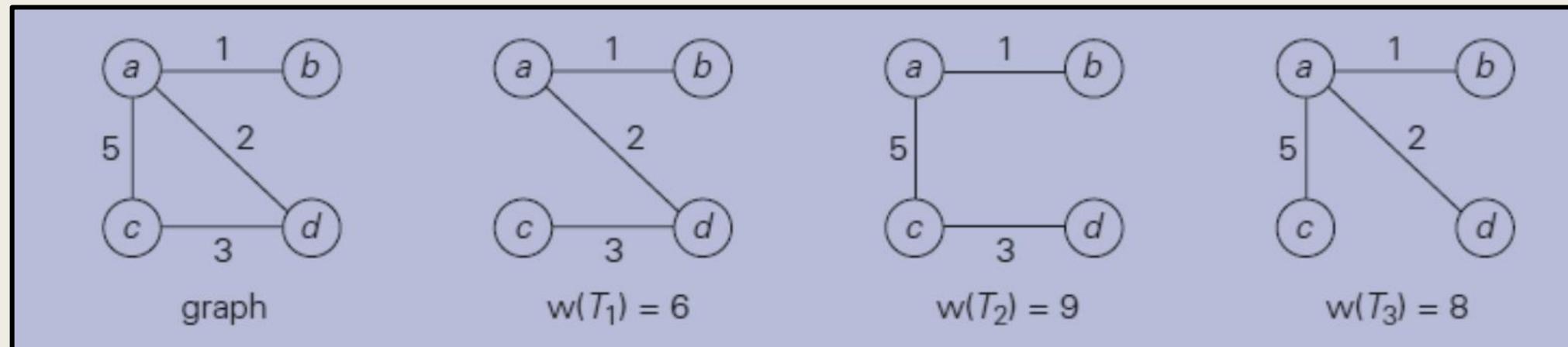


Spanning Tree

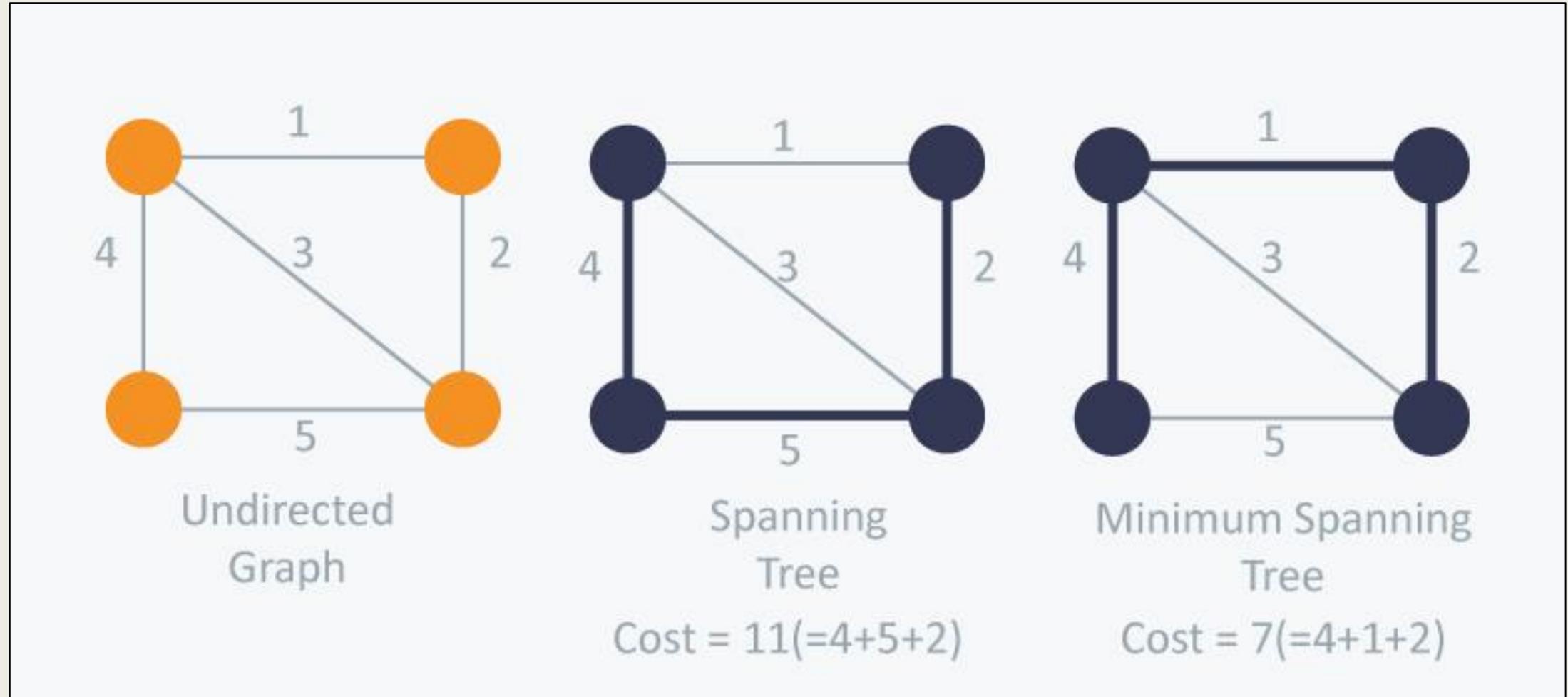


Spanning Tree

- A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.
- For a weighted graph, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges.
- The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.



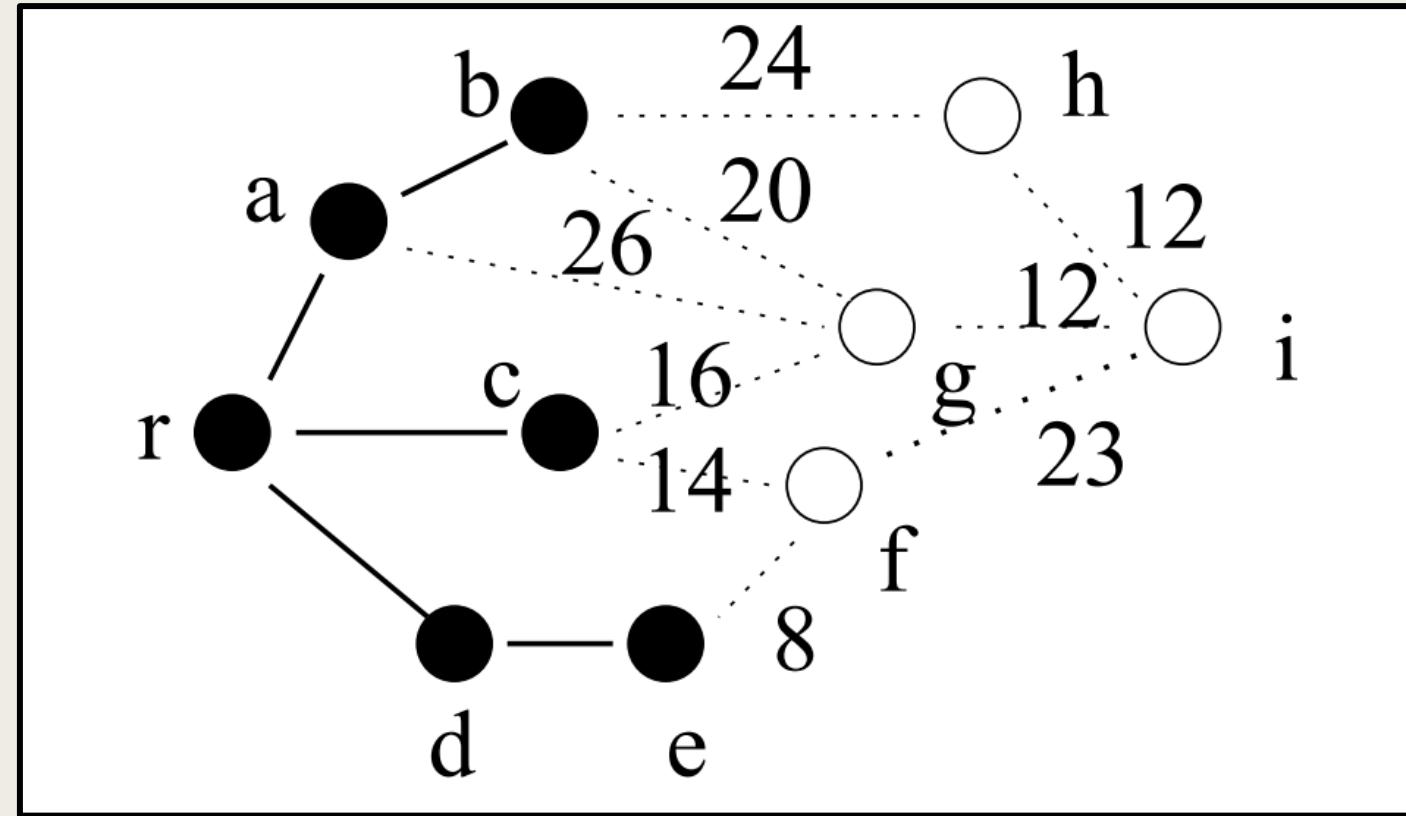
Spanning Tree



Prims Algorithm

- A greedy method to obtain a minimum-cost spanning tree builds this tree **edge by edge**.
- The next edge to include is chosen according to some optimization criterion.
- The simplest such criterion is to choose an edge that results in a **minimum increase** in the sum of the costs of the edges so far included

Prims Algorithm – Illustration



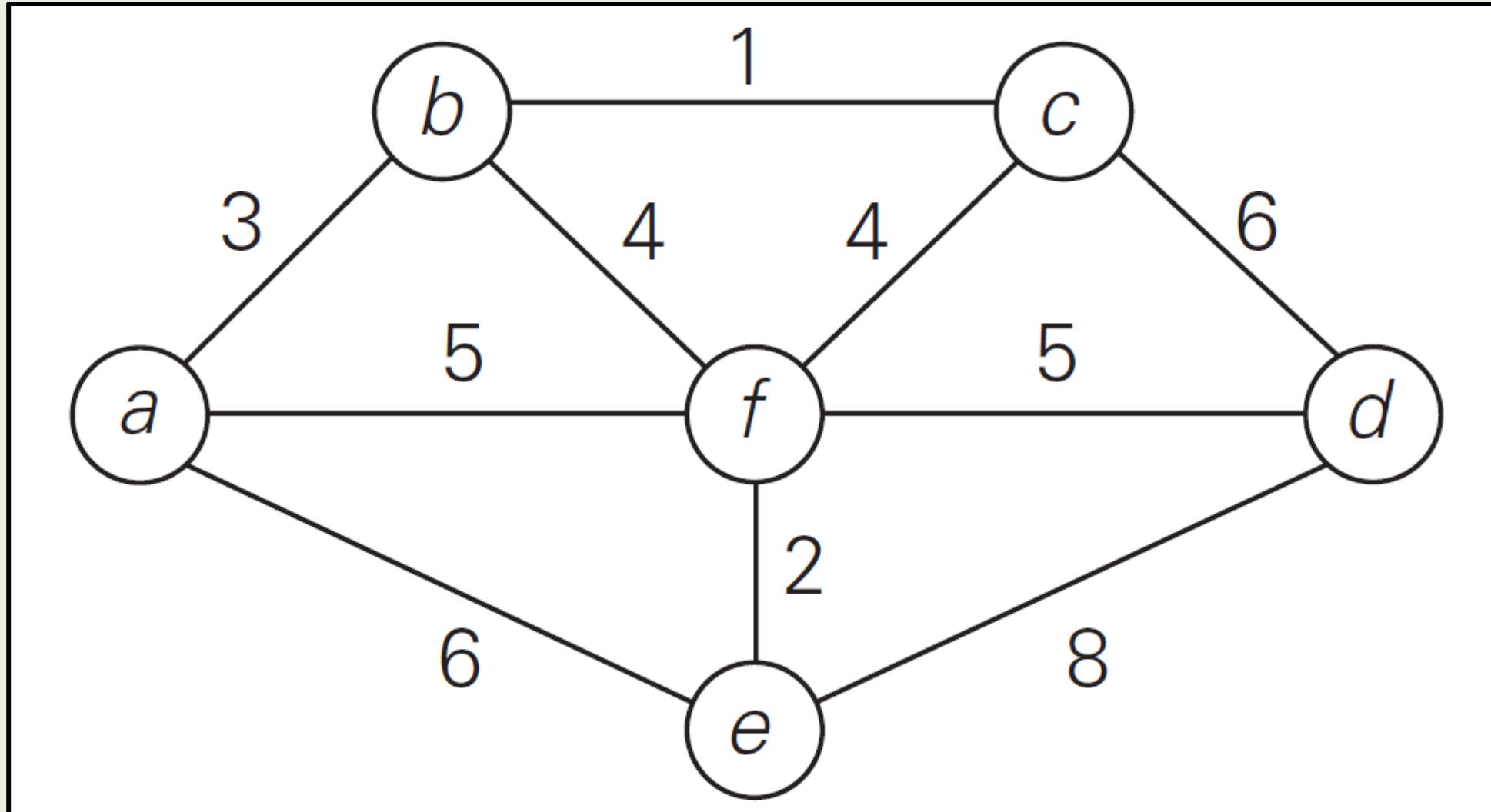
Tree Vertices - { **r, a, b, c, d, e** } – vertices that are part of the spanning tree

Fringe vertices - { **h, g, f** } – vertices not in the tree but adjacent to at least one vertex that is in the tree

Unseen vertices – { **i** } – vertices not yet affected by the algorithm

Prims Algorithm – Problem

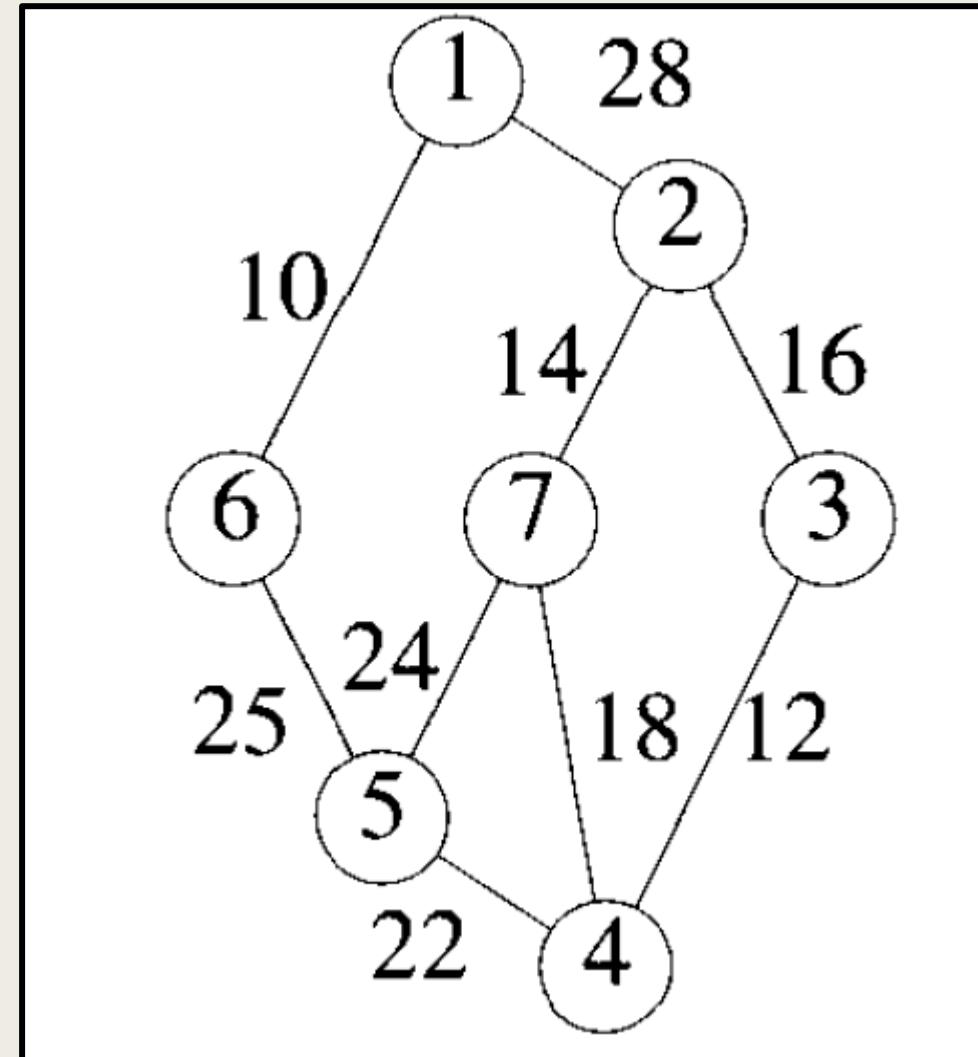
Find the minimum cost spanning tree for the following graph



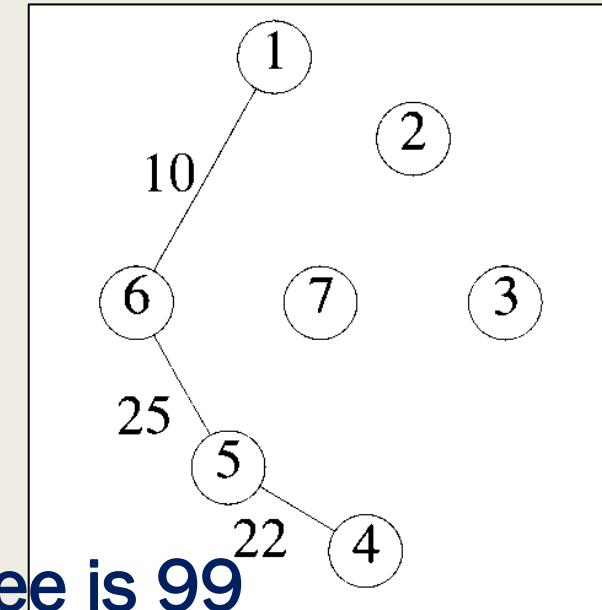
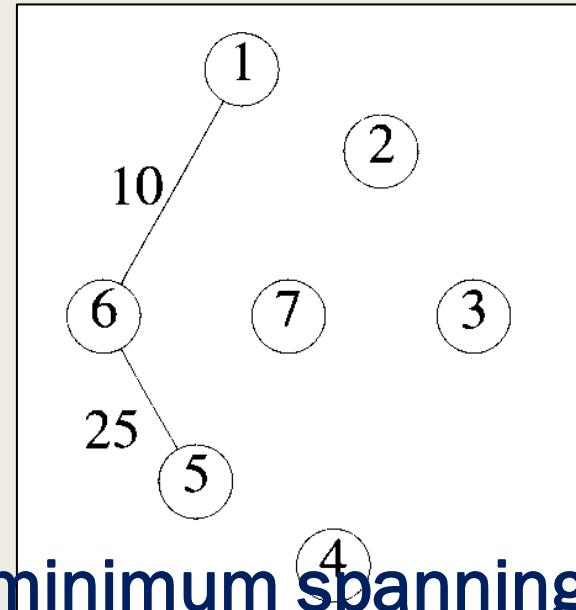
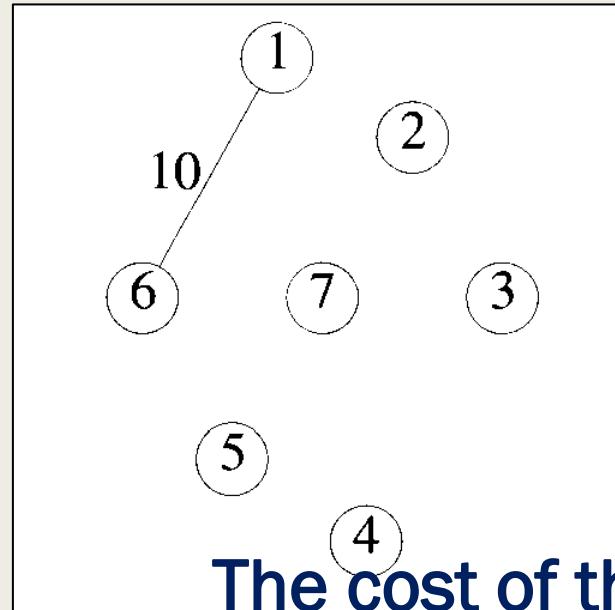
Solution

Prims Algorithm – Problem

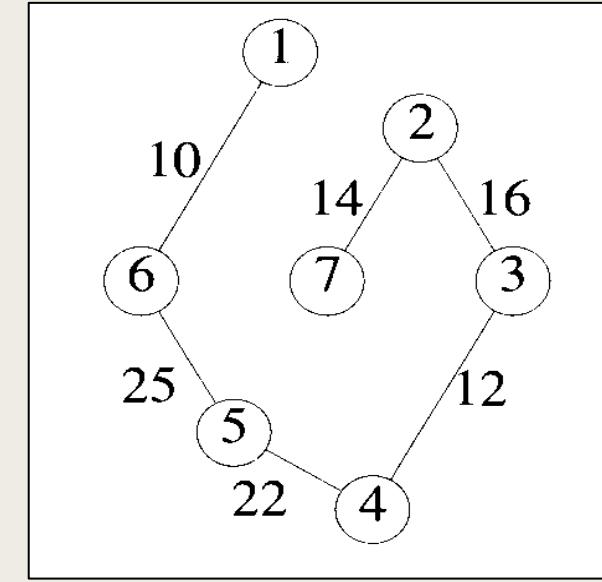
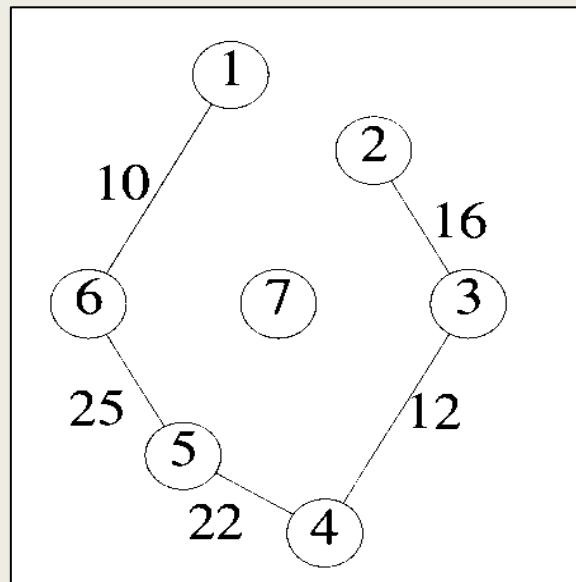
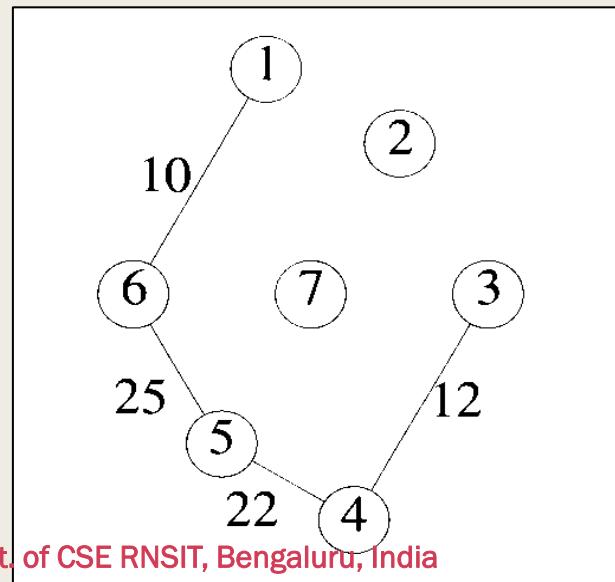
Find the minimum cost spanning tree for the following graph



Prims Algorithm- Example

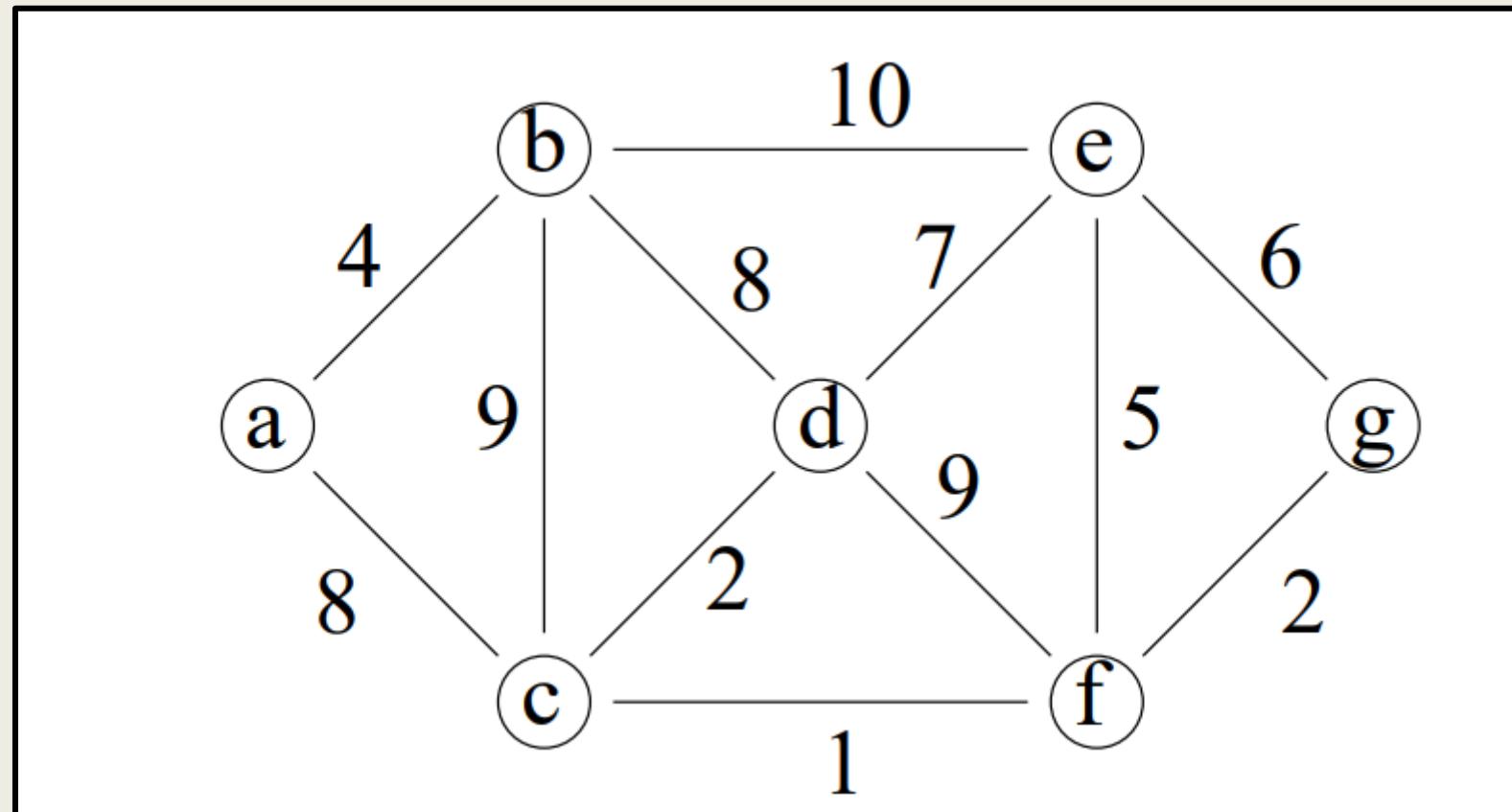


The cost of the minimum spanning tree is 99

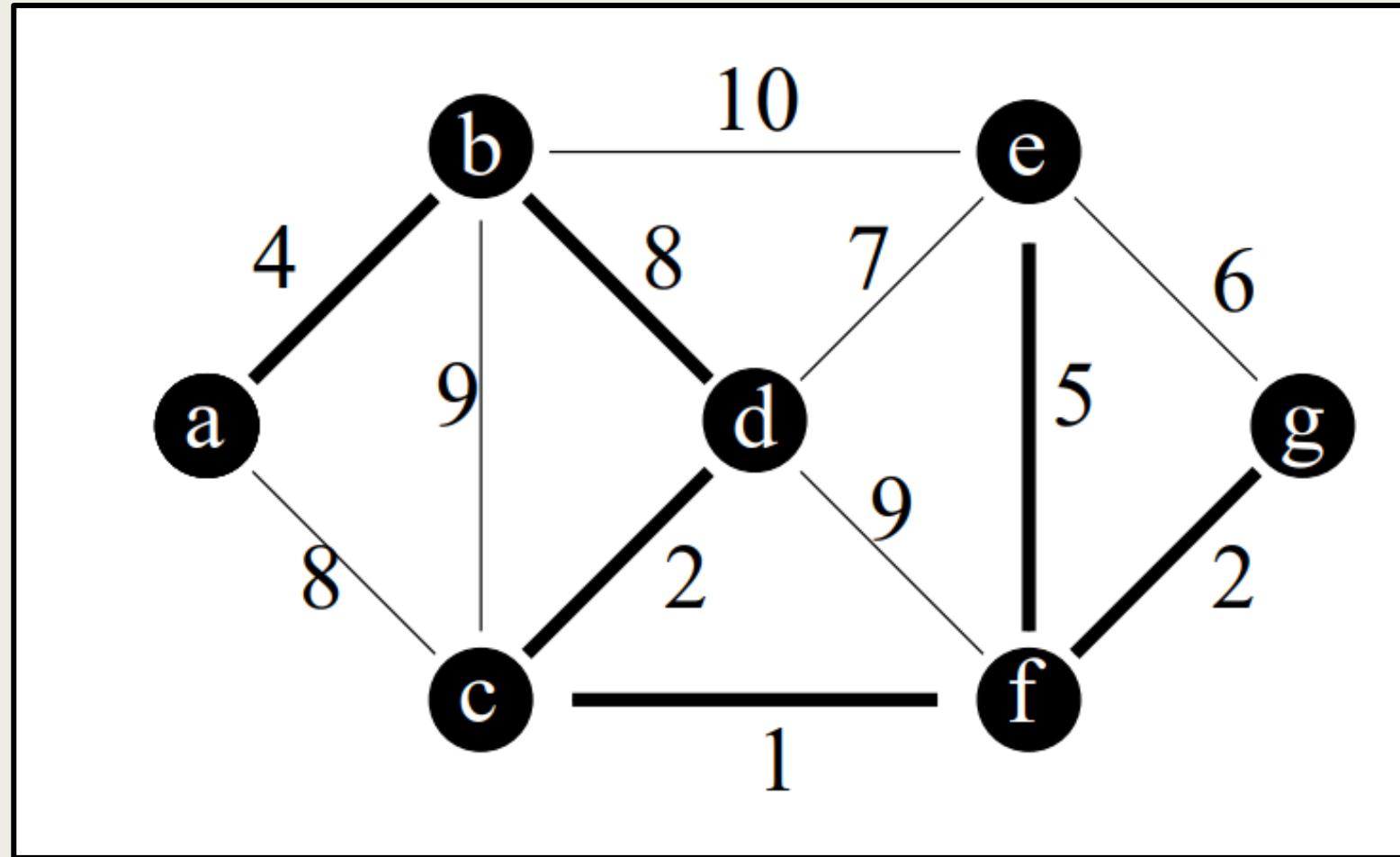


Prims Algorithm – Problem Home work

Find the minimum cost spanning tree for the following graph



Prims Algorithm – Problem Home work – solution



The cost of the minimum spanning tree is 22

Prims Algorithm

ALGORITHM $Prim(G)$

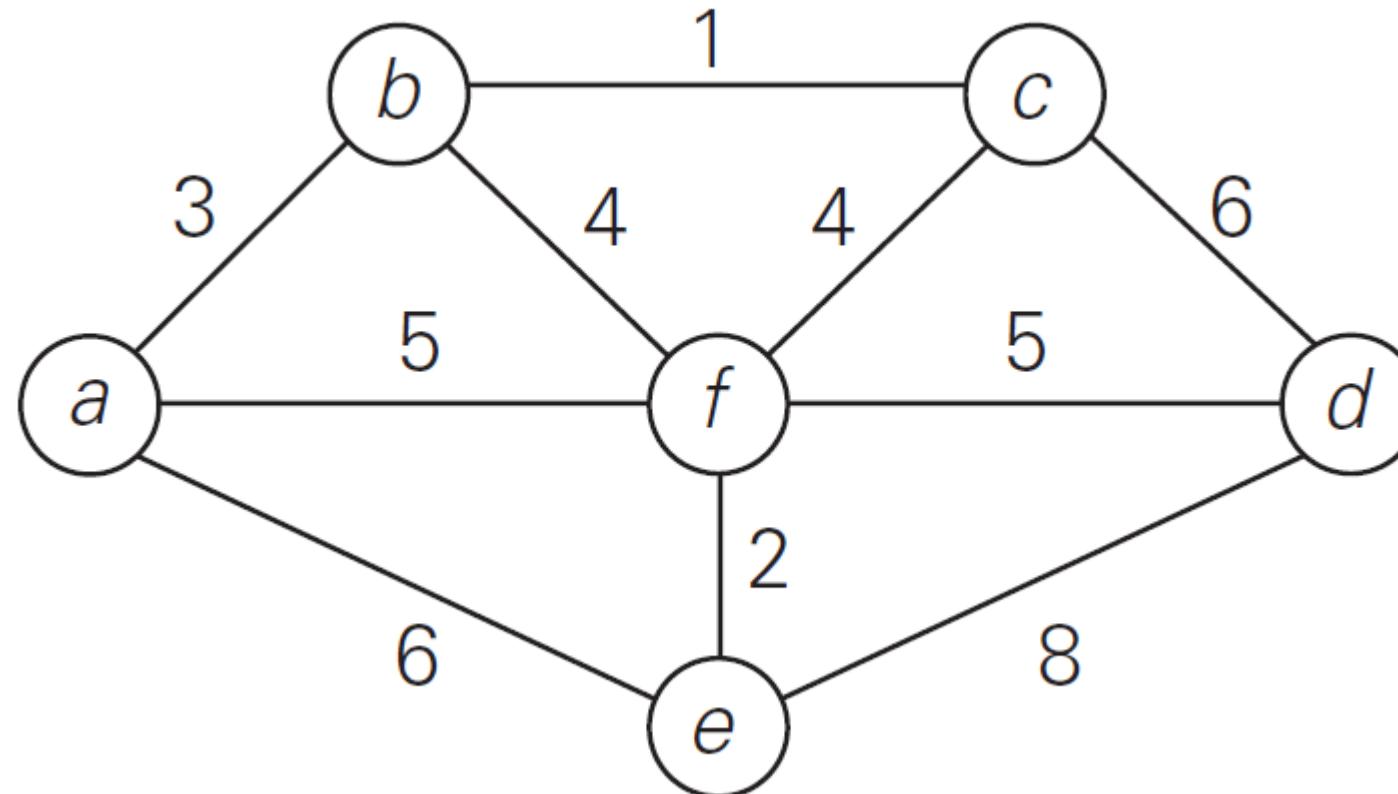
```
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G = \langle V, E \rangle$ 
//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$ 
 $V_T \leftarrow \{v_0\}$  //the set of tree vertices can be initialized with any vertex
 $E_T \leftarrow \emptyset$ 
for  $i \leftarrow 1$  to  $|V| - 1$  do
    find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$ 
    such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$ 
     $V_T \leftarrow V_T \cup \{u^*\}$ 
     $E_T \leftarrow E_T \cup \{e^*\}$ 
return  $E_T$ 
```

Kruskal's Algorithm

- Joseph Kruskal developed an algorithm - **Kruskal's Algorithm** - to find an optimal solution to minimum spanning tree problem when he was a **second year graduate student**
 - Prim's algorithm grew the minimum spanning tree by including nearest vertex to the vertices already in the tree
 - In Prim's algorithm, the subgraphs were always connected in the intermediate stages
- The **Kruskal's algorithm** begins by sorting the edges in **increasing order of their weights**
- Starting with the **empty subgraph**, it scans this sorted list, **adding the next edge** on the list to the current subgraph if such an **inclusion does not create a cycle** and simply **skipping the edge** otherwise

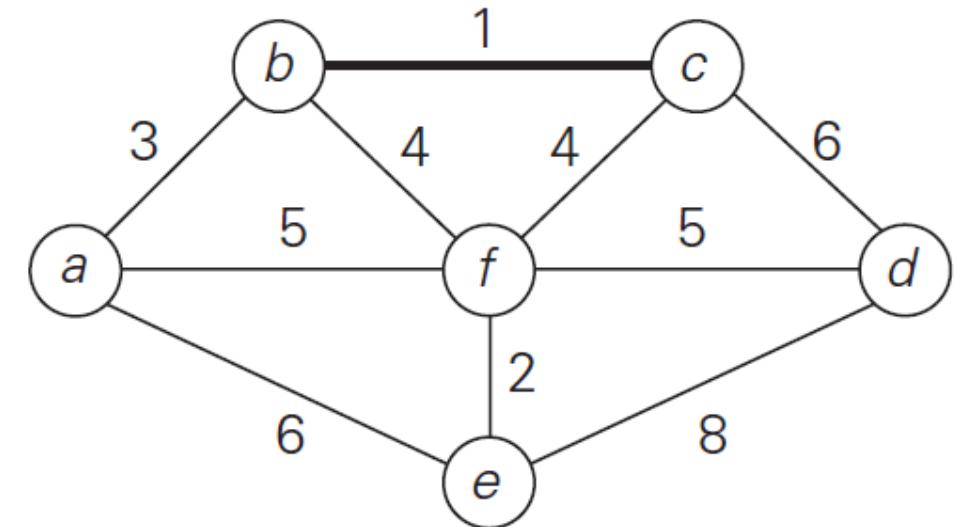
Kruskal's Algorithm - Problem

Find the minimum cost spanning tree for the following graph



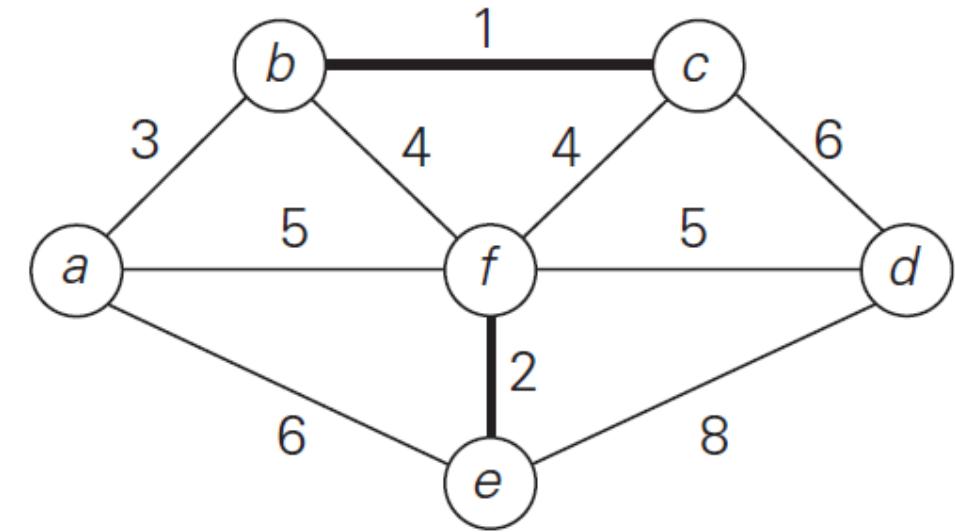
Kruskal's Algorithm - Solution

bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



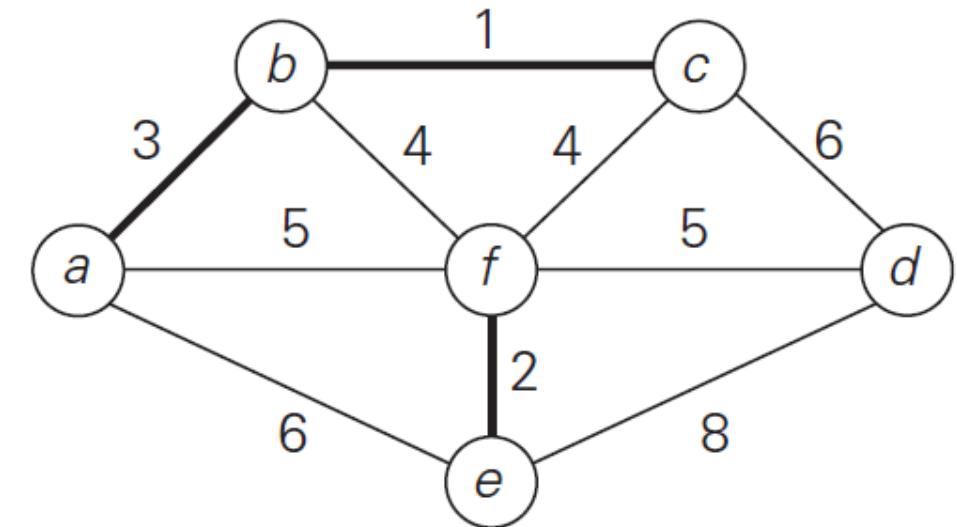
Kruskal's Algorithm - Solution

bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



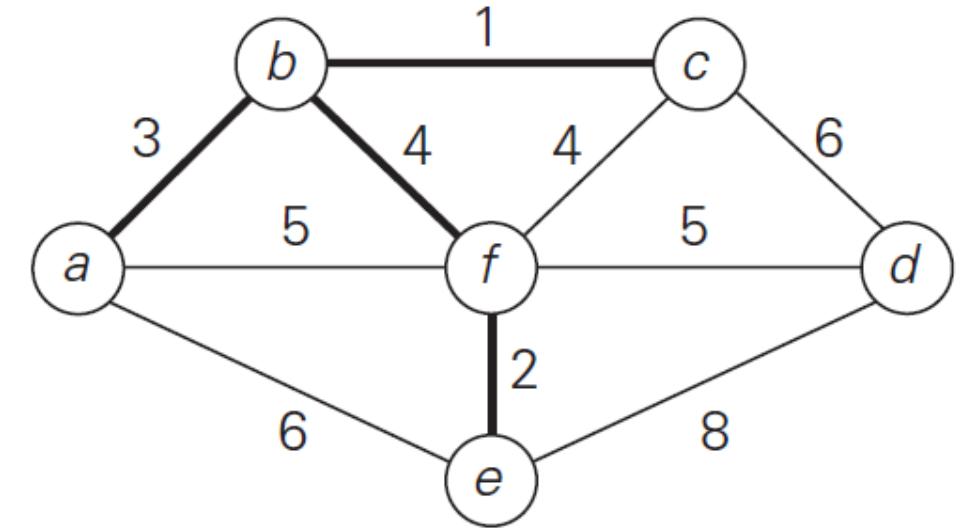
Kruskal's Algorithm - Solution

bc ef **ab** bf cf af df ae cd de
1 2 3 4 4 5 5 6 6 8



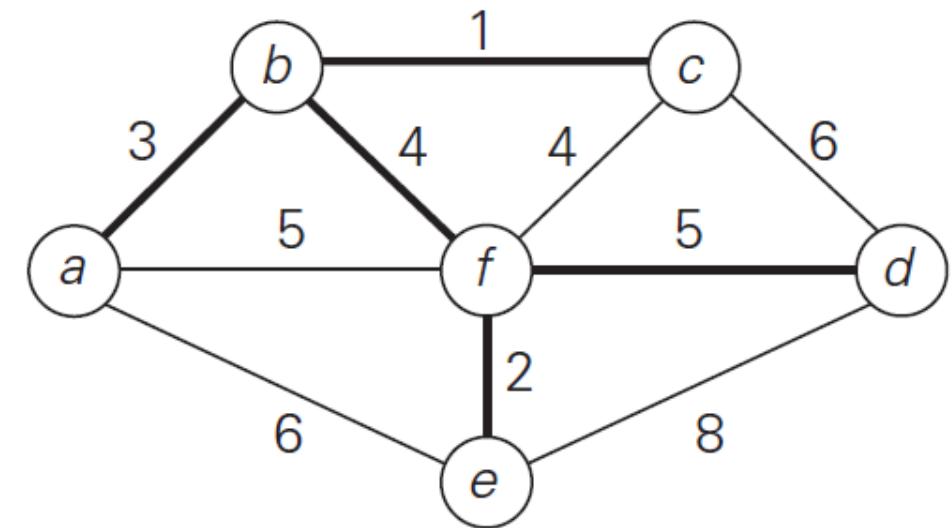
Kruskal's Algorithm - Solution

bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



Kruskal's Algorithm - Solution

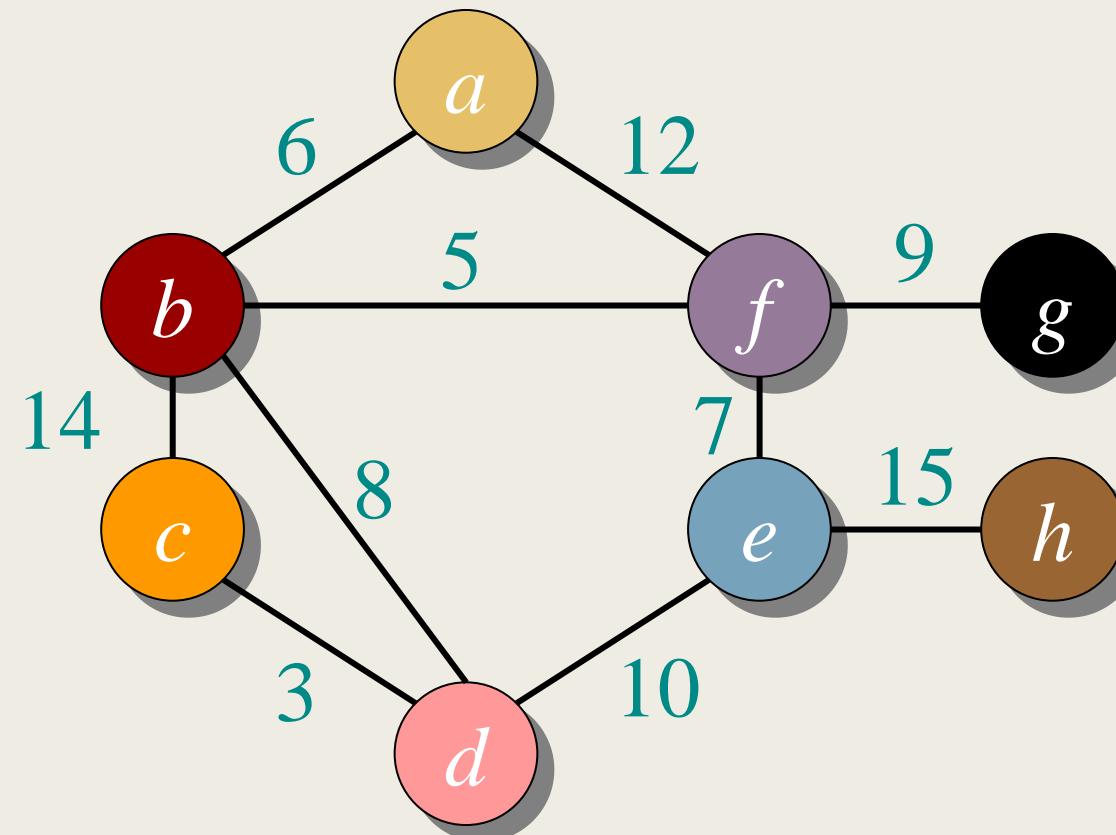
bc ef ab bf cf af **df** ae cd de
1 2 3 4 4 5 5 6 6 8



Total cost of the minimum spanning tree= 15

Kruskal's Algorithm - Problem

- Find the minimum cost spanning tree for the following graph



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

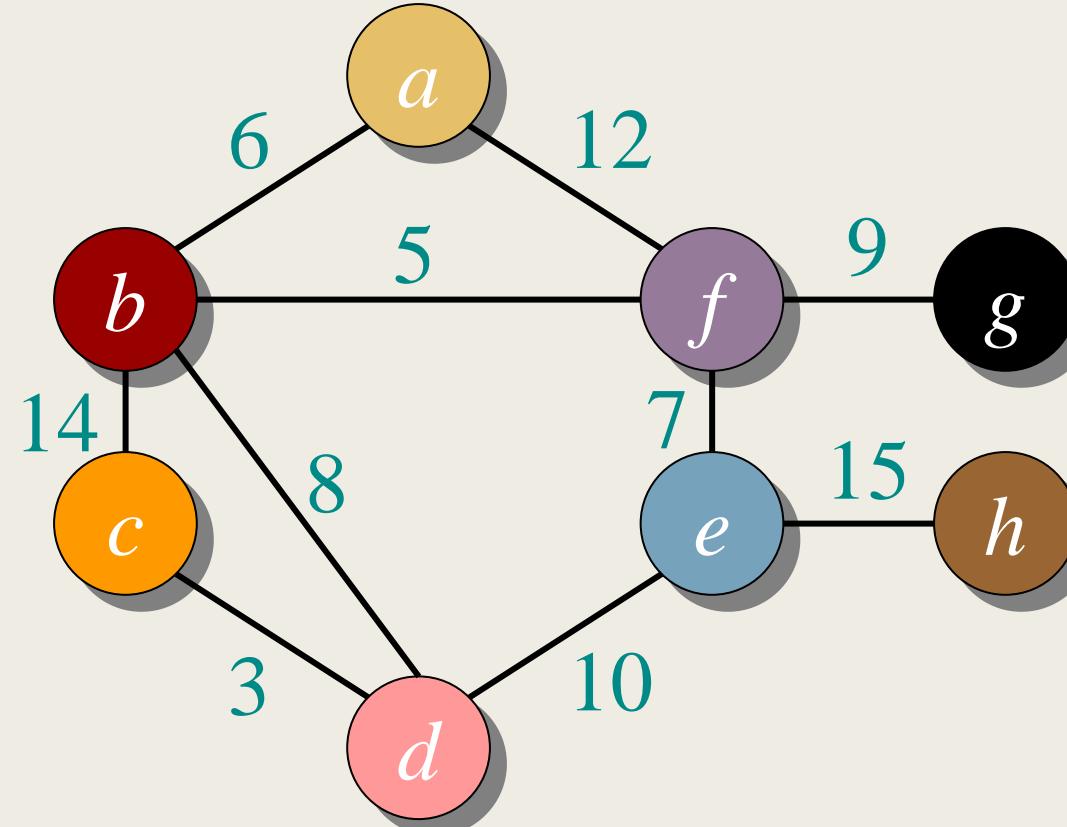
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

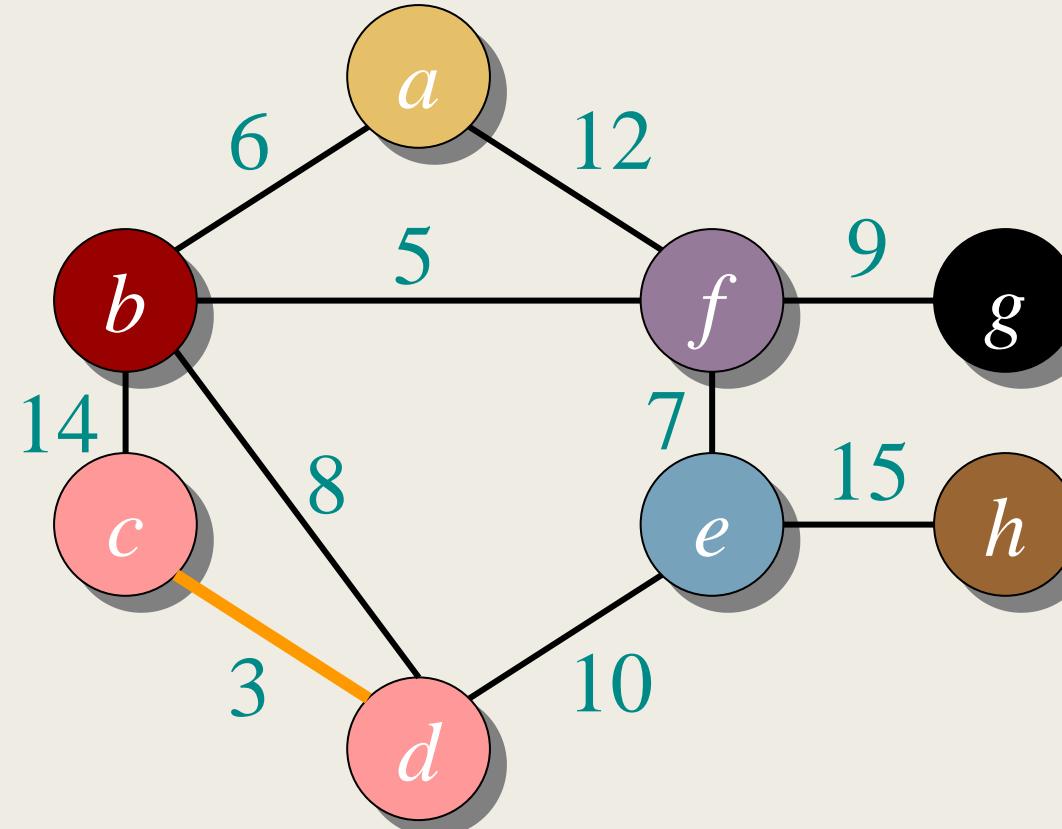
f-g: 9

d-e: 10

a-f: 12

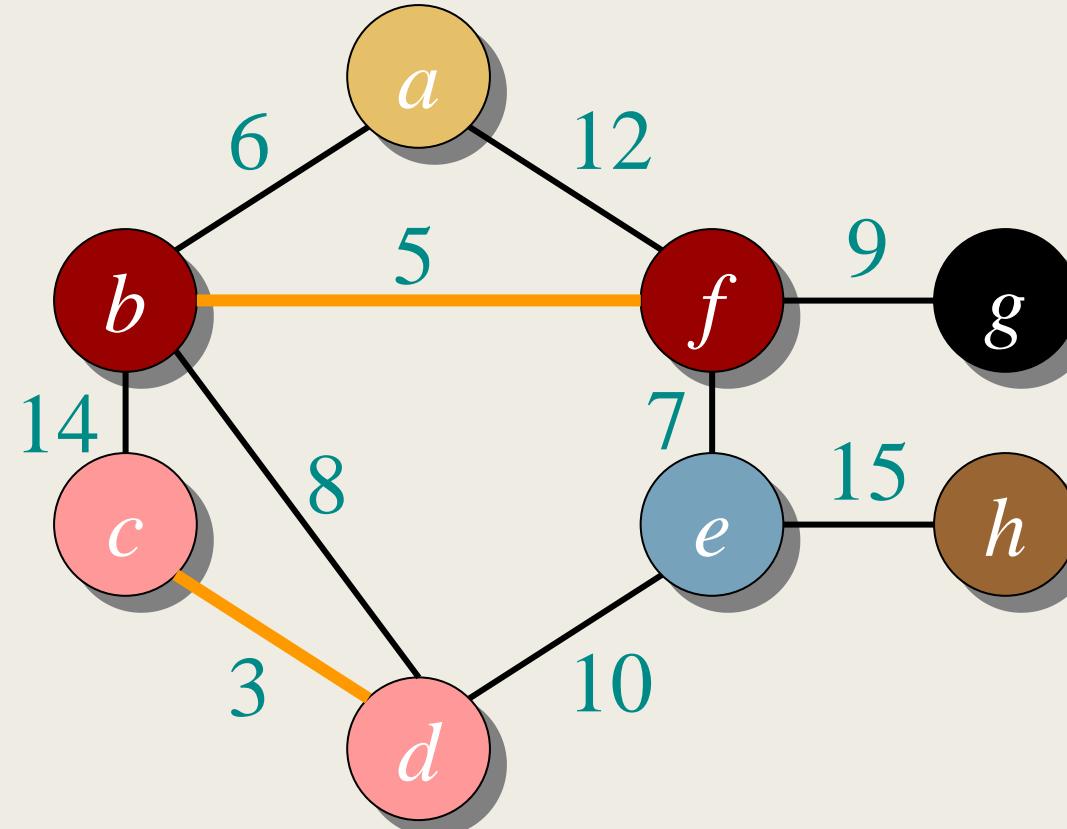
b-c: 14

e-h: 15



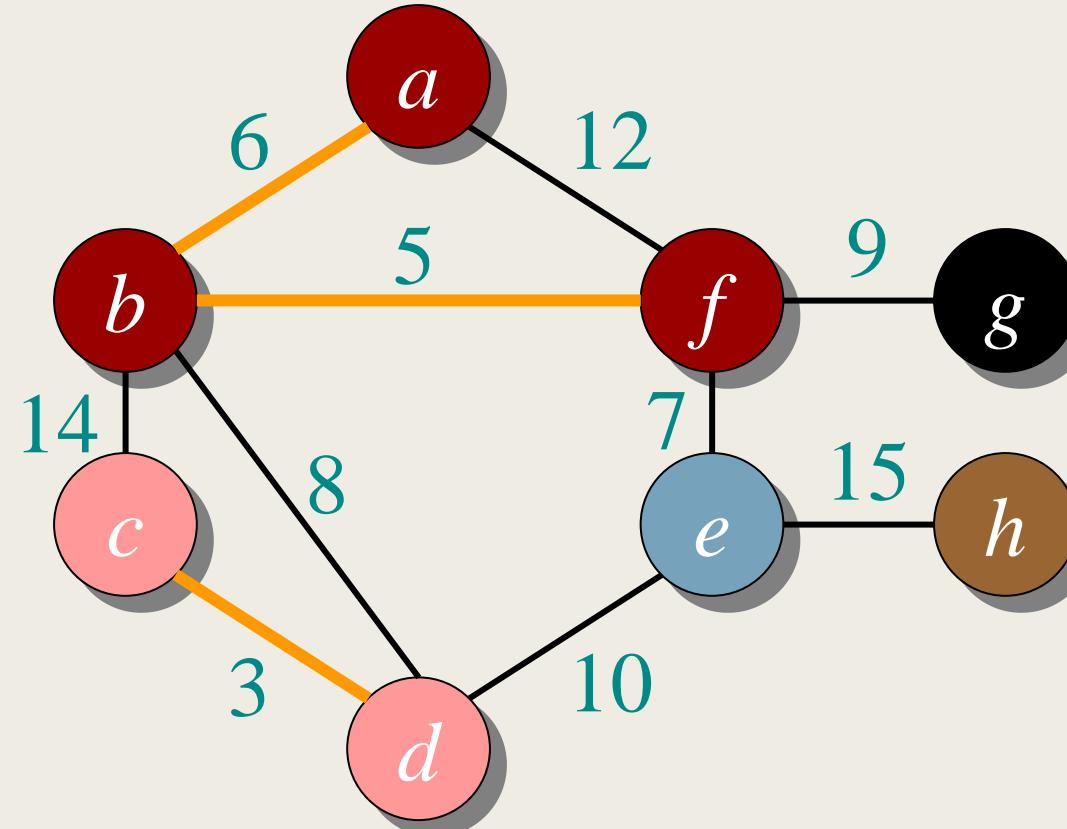
Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



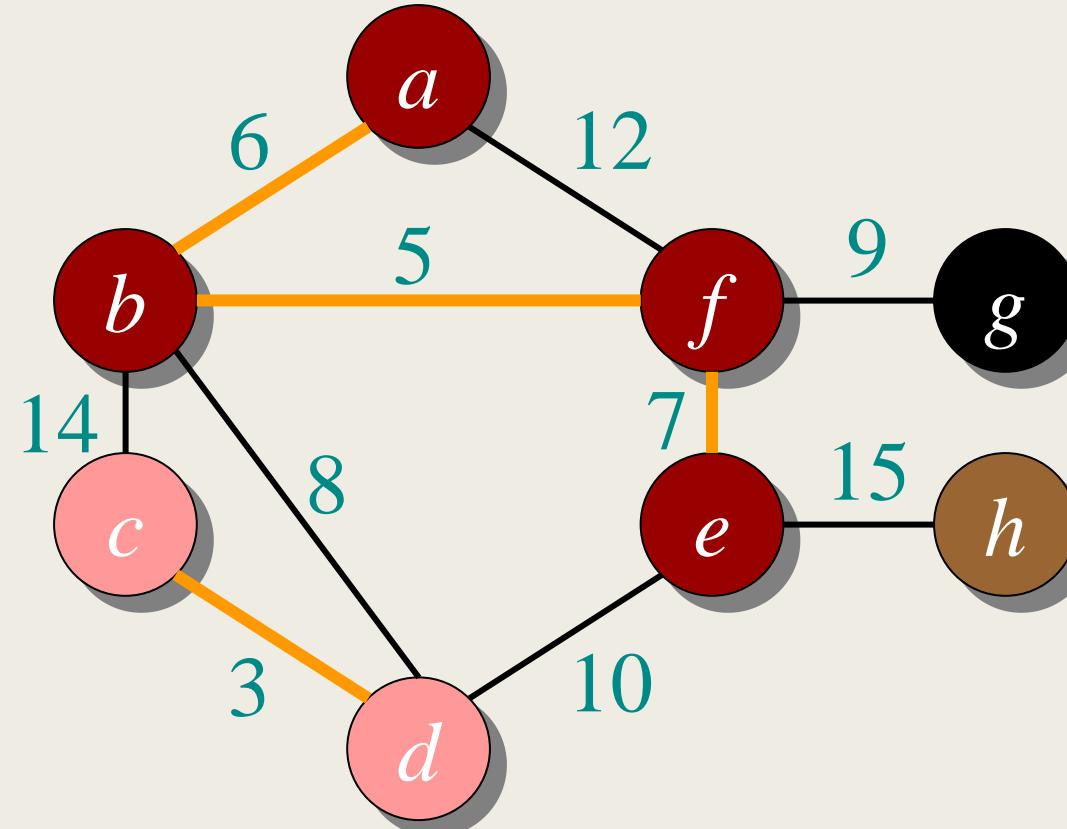
Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



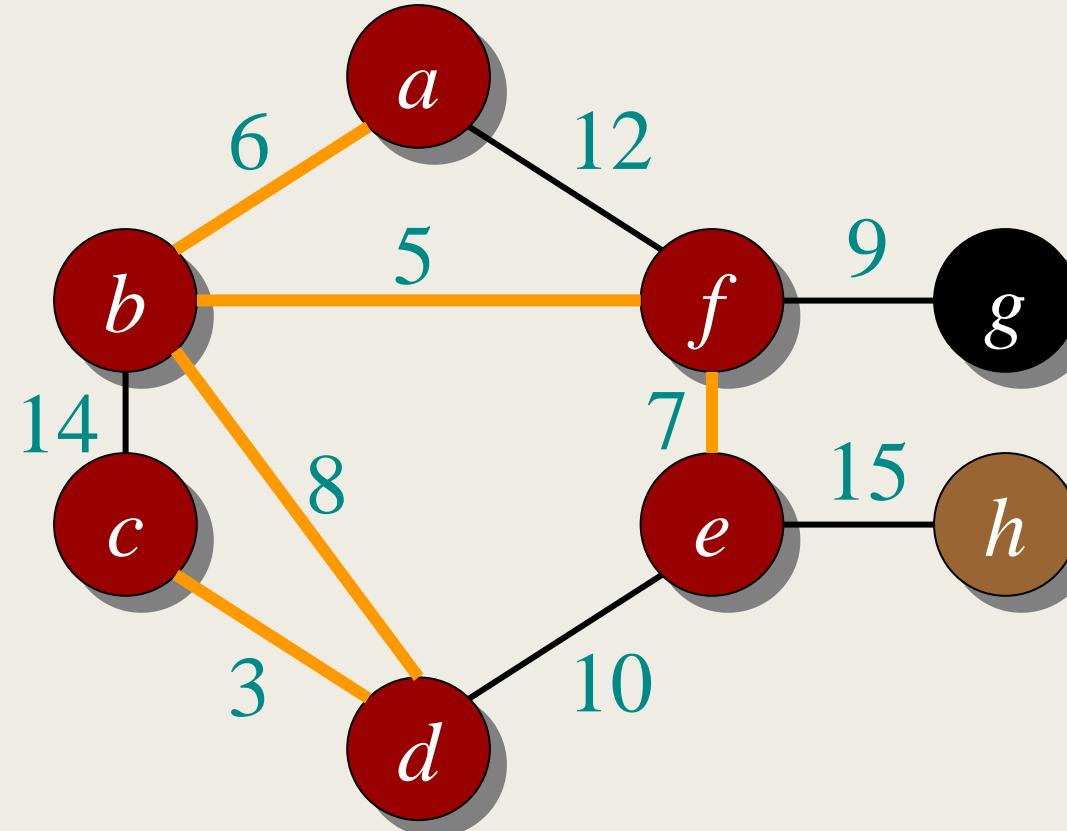
Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



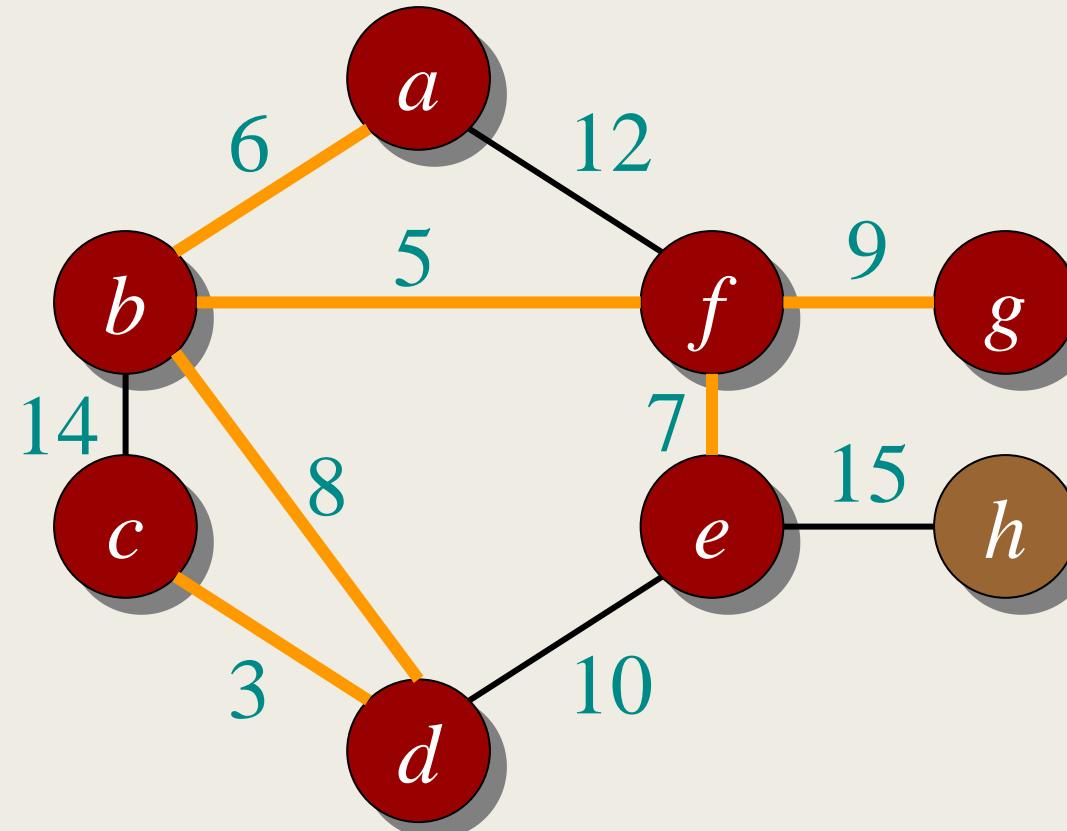
Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



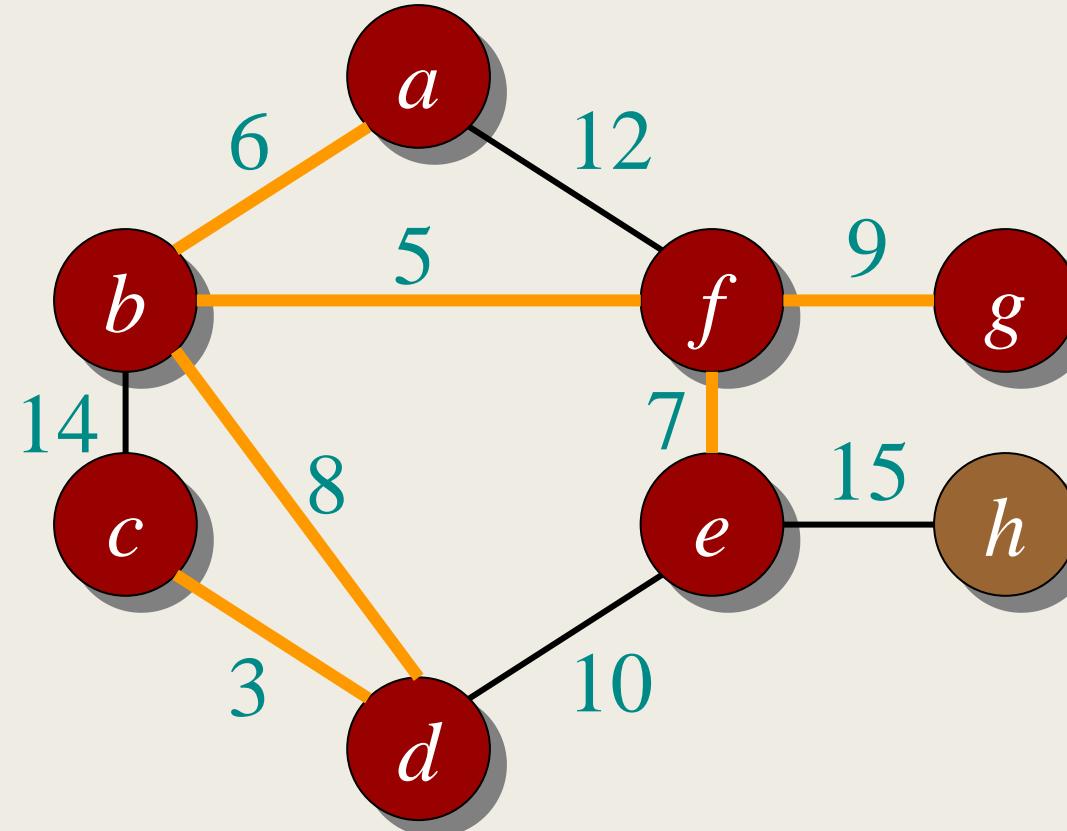
Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



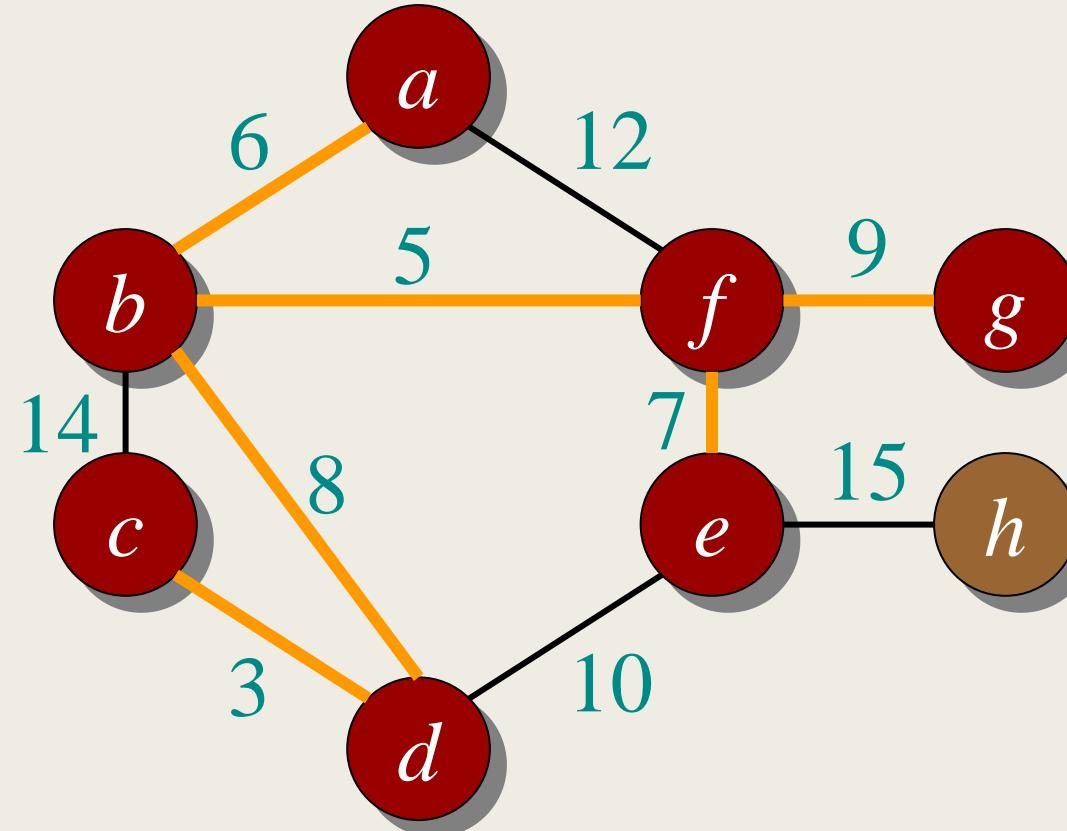
Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



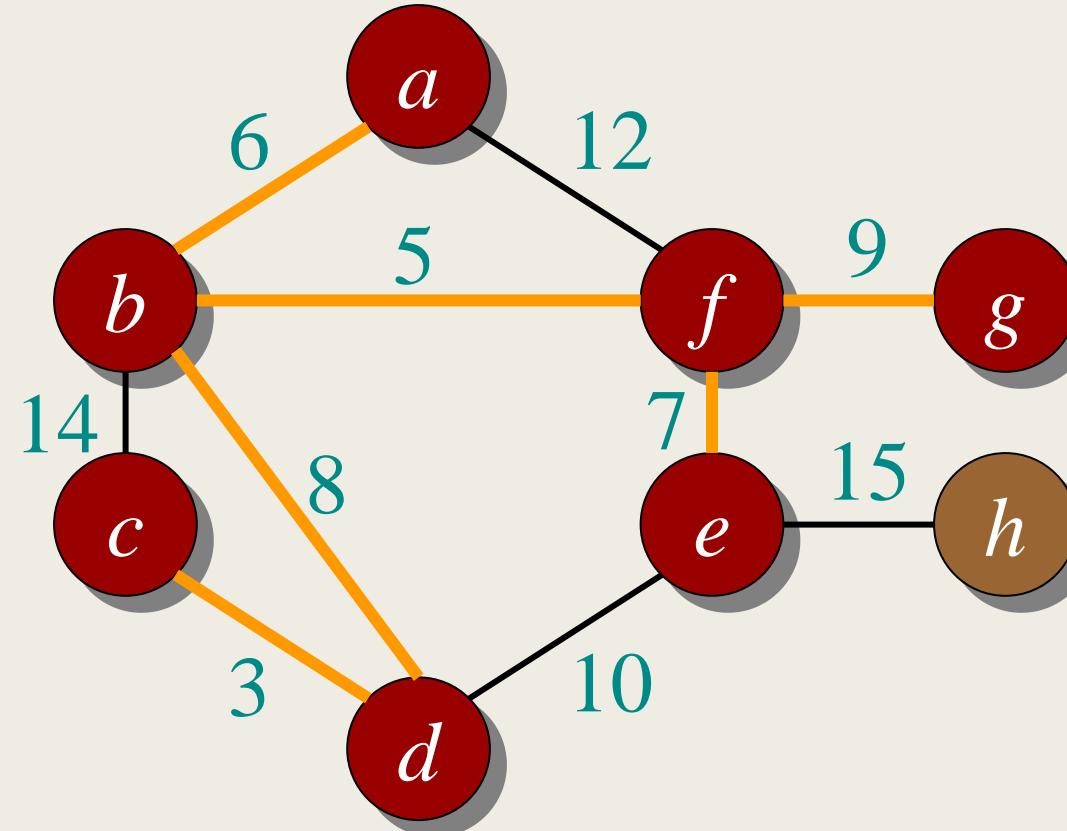
Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



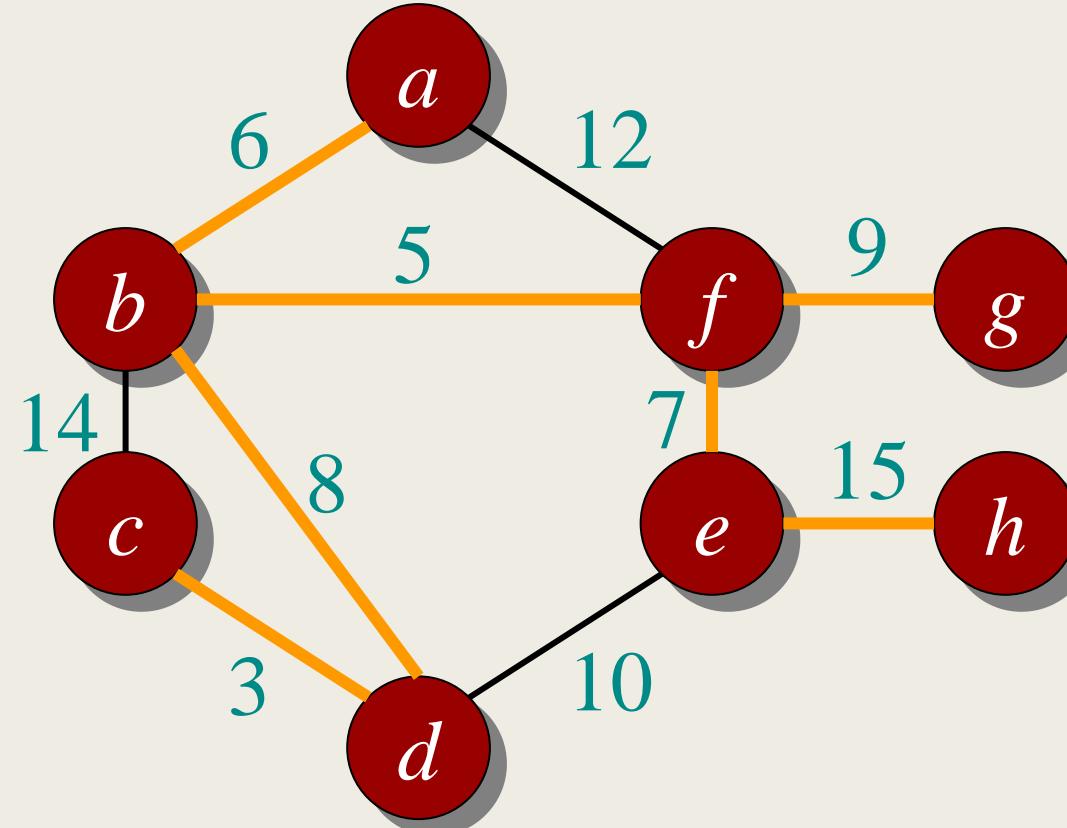
Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



Kruskal's Algorithm - Solution

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



Total cost of the minimum spanning tree= 53

Kruskal's Algorithm

ALGORITHM Kruskal(G)

```

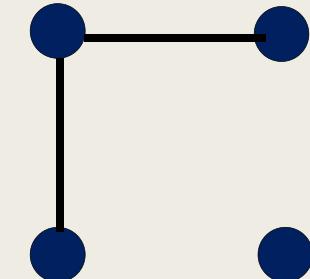
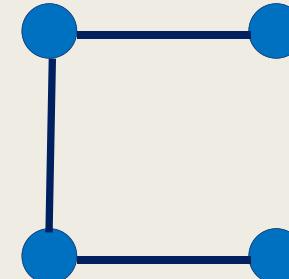
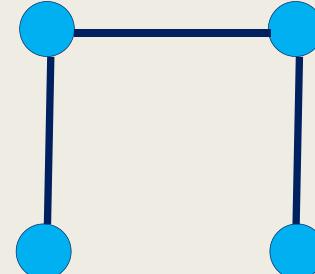
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G = \langle V, E \rangle$ 
//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$ 
sort  $E$  in nondecreasing order of the edge weights  $w(e_{i_1}) \leq \dots \leq w(e_{i_{|E|}})$ 
 $E_T \leftarrow \emptyset$ ;  $eCounter \leftarrow 0$       //initialize the set of tree edges and its size
 $k \leftarrow 0$                           //initialize the number of processed edges
while  $eCounter < |V| - 1$  do
     $k \leftarrow k + 1$ 
    if  $E_T \cup \{e_{i_k}\}$  is acyclic
         $E_T \leftarrow E_T \cup \{e_{i_k}\}$ ;  $eCounter \leftarrow eCounter + 1$ 
return  $E_T$ 

```

Kruskal's Algorithm

The implementation catch

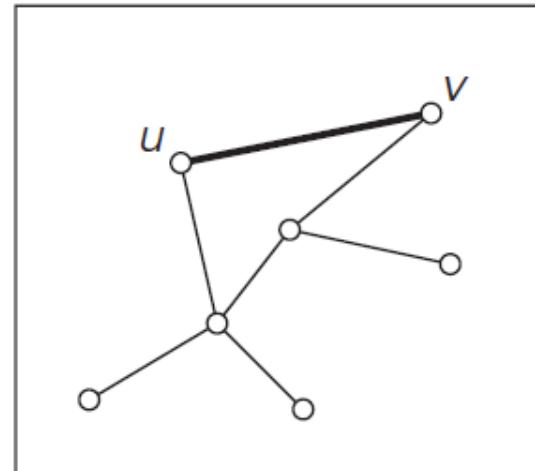
- Kruskal's approach gives away an impression of being very easy to implement. It is really so?
- Each time / iteration the algorithm should check whether adding the next edge to the edges already selected creates a cycle?
- Remember a cycle is created iff the new edge connects two vertices which are already connected by a path



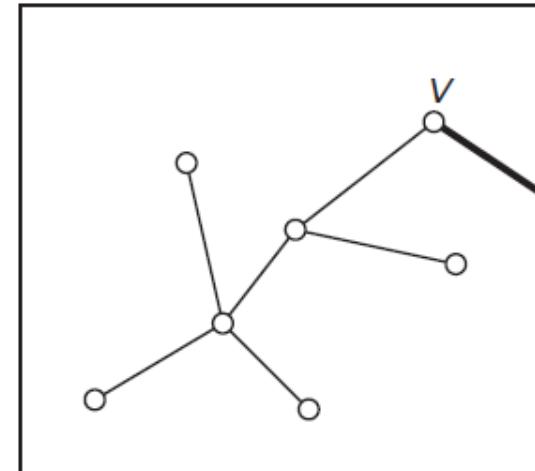
Kruskal's Algorithm

The implementation catch

- New cycle is created if and only if the two vertices belong to the same connected component
- Each connected component of a subgraph generated by Kruskal's algorithm is a tree because it has no cycles.



(a)



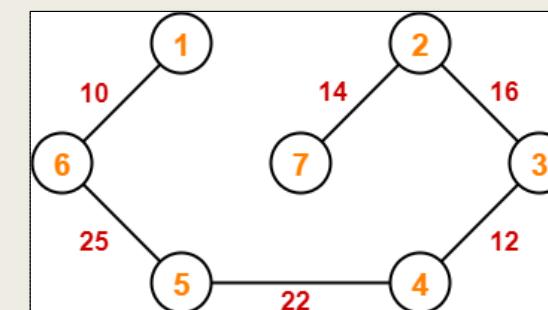
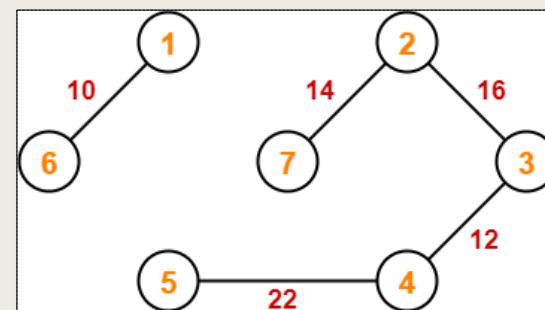
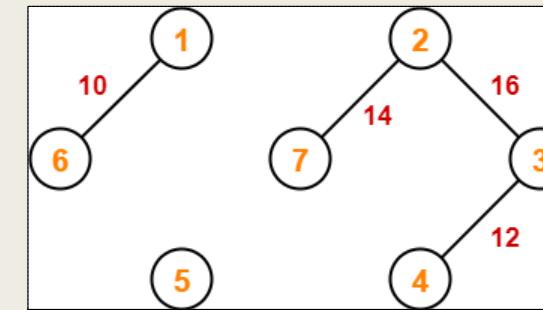
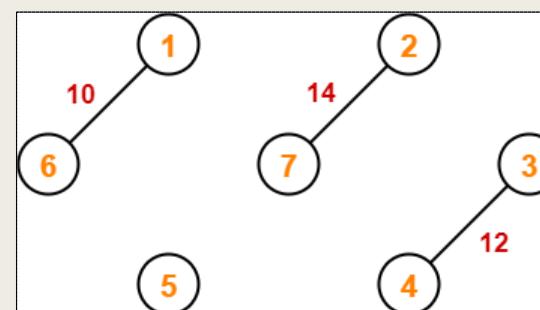
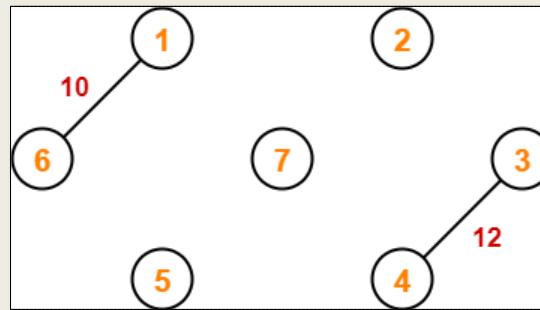
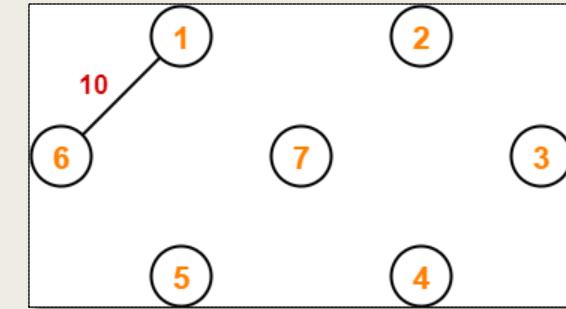
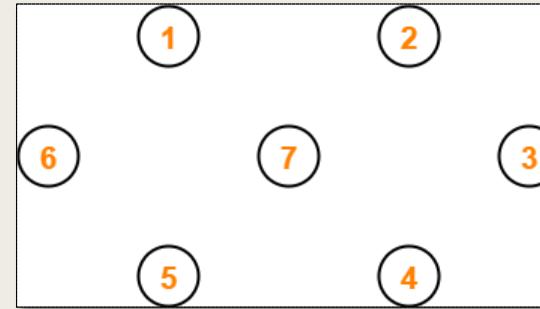
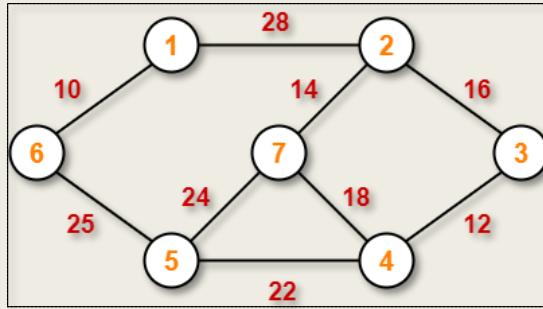
(b)

Kruskal's Algorithm

The implementation catch

- Consider the algorithm's operations as a progression through a series of forests containing all the vertices of a given graph and some of its edges.
- The initial forest consists of $|V|$ trivial trees, each comprising a single vertex of the graph.
- The final forest consists of a single tree, which is a minimum spanning tree of the graph.
- On each iteration, the algorithm takes
 - *the next edge (u, v) from the sorted list of the graph's edges,*
 - *finds the trees containing the vertices u and v , and,*
 - *if these trees are not the same, unites them in a larger tree by adding the edge (u, v) .*

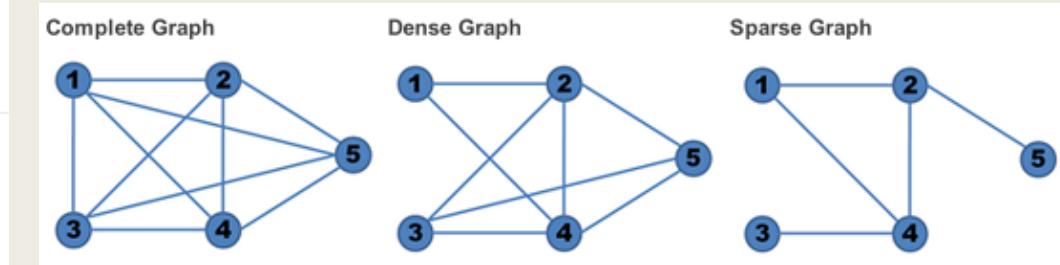
Kruskal's Algorithm



Differences between Prim's and Kruskal's

PRIM'S ALGORITHM	KRUSKAL'S ALGORITHM
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices.	Kruskal's algorithm's time complexity is $O(\log V)$, V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

A **graph** in which the number of edges is much less than the possible number of edges



A **graph** in which the number of edges is close to the possible number of edges.

Job Sequencing with deadlines

- The sequencing of jobs (for execution) on a single processor with deadline constraints is called as Job Sequencing with Deadlines.
- Problem statement

“

Given,

- **set of n jobs**
- **a deadline associated with each job i , $d_i \geq 0$**
- **profit associated with each job i , $p_i > 0$**
- For any job i the profit p_i is earned iff the job is completed by its deadline
- To complete a job, one has to process the job on a machine for one unit of time
- Only one machine is available for processing job

Obtain the sequence of jobs that yields optimal solution (max profit) ”

Job Sequencing with deadlines



What is the feasible solution ?

- A **feasible solution** for this problem is a subset J of jobs such that each Job in this subset can be **completed by its deadline**
- The value of a **feasible solution J** is the sum of the profits of the jobs in J , or $\sum_{i \in J} p_i$
- An **optimal solution** is a feasible solution with **maximum value**.

Job Sequencing with deadlines

Brute force approach

- Solve the following instance of job sequencing problem

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

	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

Job Sequencing with deadlines



Greedy Approach

- *Sort all the given jobs in decreasing order of their profit.*
- *Check the value of maximum deadline.*
- *Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.*
- *Select the jobs one by one.*
- *Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.*

Job Sequencing with deadlines

- Solve the following instance of job sequencing with deadline problem.

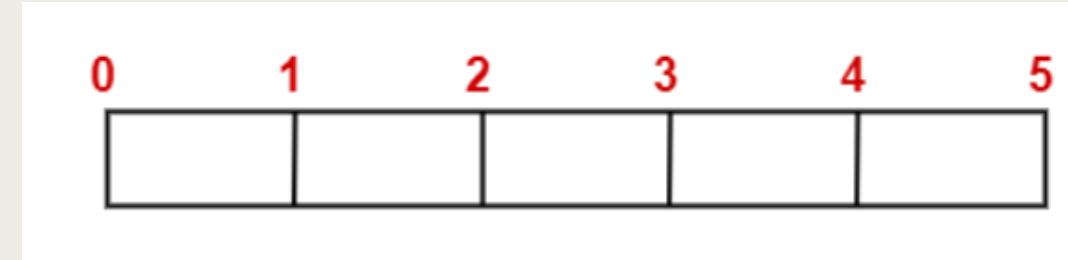
Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

- Solution
 - Sort all the given jobs in decreasing order of their profit.*

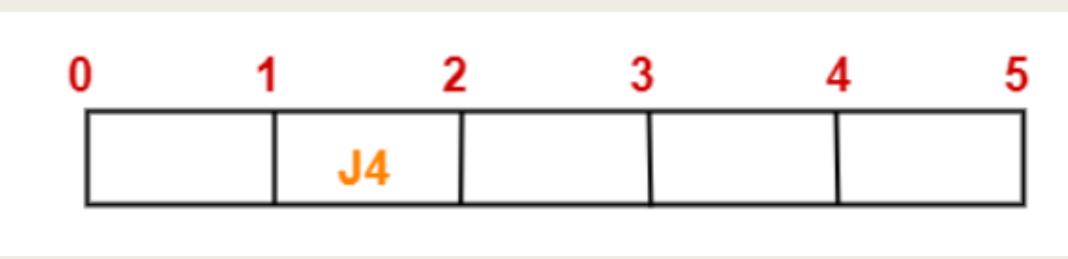
Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Job Sequencing with deadlines

- Value of maximum deadline = 5.
- So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown below

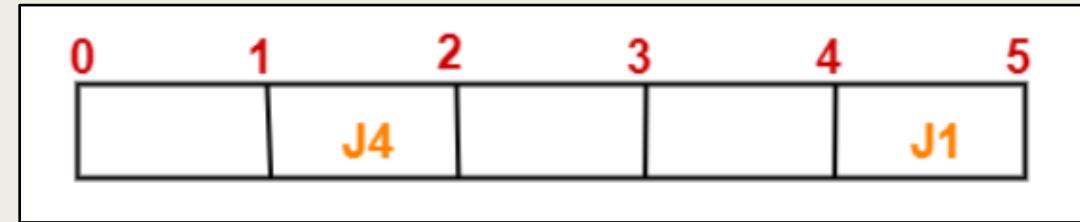


- Select the jobs as per the ordered profits and place them in the chart accordingly
- Select J4 ($d_4=2$)

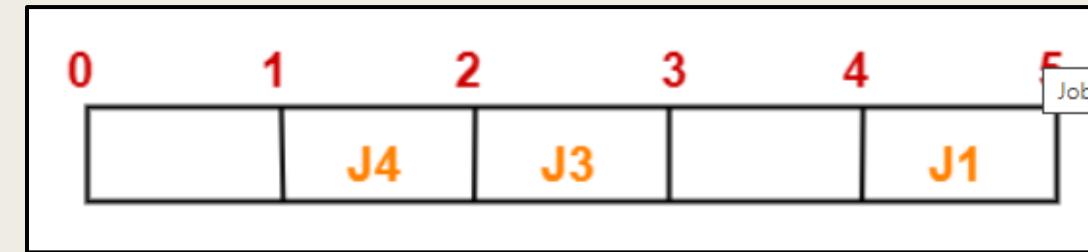


Job Sequencing with deadlines

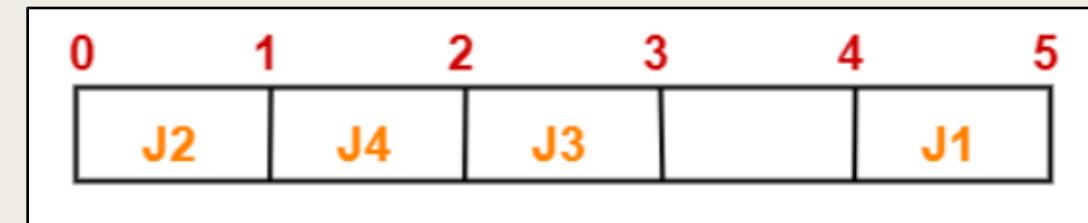
- Select J1 ($d_1=5$)



- Select J3 ($d_3=3$)



- Select J2 ($d_2=3$)



Job Sequencing with deadlines

- Select J5 (d5=4)



- The left over job is J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.
- Hence the optimal sequence of jobs are <J2, J4, J3, J5, J1>
- The profit earned by this sequence is 990 units

Job Sequencing with deadlines

- Solve the following instance of job sequencing problem

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

Solution ?

Huffman coding

Situation

- How will you compactly store a data file of 100000 characters?
- The file consists of only 6 different characters
- Their occurrences (frequencies) are as follows

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

- Designing a **binary character code** (in which each character is represented by a unique binary string, which we call a **Codeword**) is one among multiple solutions

Huffman coding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

■ Fixed length code (e.g. ASCII)

- 3 bits to represent 6 characters
- **a = 000, b = 001, . . . , f = 101.**
- This method requires 300,000 bits to code the entire file
- **Can you do better ?**

Huffman coding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

■ Variable length code (Morse code)

- assign frequent characters short codewords and infrequent characters long codewords
- This code requires $(45.1+13.3+12.3+16.3+9.4+5.4).1,000 = 224,000$ bits
- When compared to fixed length code, the variable length code saved the storage by 25%

Huffman coding

Prefix codes

- A code is called a prefix (free) code if no codeword is a prefix of another one. (typically of variable length)
- Example

A: 00

B: 010

C: 011

D: 10

E: 11

↑
Prefix code

A: 00

B: 010

C: 001 # 011 -> 001

D: 10

E: 11

↑
Not a Prefix code

{3, 11, 22} ← Prefix code

{1, 12, 13} ← Not a Prefix code

Huffman coding

Prefix codes

- Life is easy with prefix code
- Let [1, 2, 33, 34, 50, 61] be the codewords
- Let the sequence of number received be 1611333425012
- Decoding ?

1 61 1 33 34 2 50 1 2

- <https://gist.github.com/joepie91/26579e2f73ad903144dd5d75e2f03d83>
- <https://leimao.github.io/blog/Huffman-Coding/>

Huffman coding

David Huffman invented a greedy algorithm to

“Construct a tree that would assign shorter bit strings to high-frequency symbols and longer ones to low-frequency symbols”

- He invented this when he was a graduate student at MIT
- The two major steps in Huffman algorithm are
 - Building a Huffman Tree from the input characters.
 - Assigning code to the characters by traversing the Huffman Tree.

Huffman coding

1. Create a leaf node for each character of the text.
2. Arrange all the nodes in increasing order of their frequency value.
3. Considering the first two nodes having minimum frequency,
 - a. Create a new internal node.
 - b. The frequency of this new node is the sum of frequency of those two nodes.
 - c. Make the first node as a left child and the other node as a right child of the newly created node.
4. Keep repeating Step-2 and Step-3 until all the nodes form a single tree.

<https://www.gatevidyalay.com/huffman-coding-huffman-encoding/>

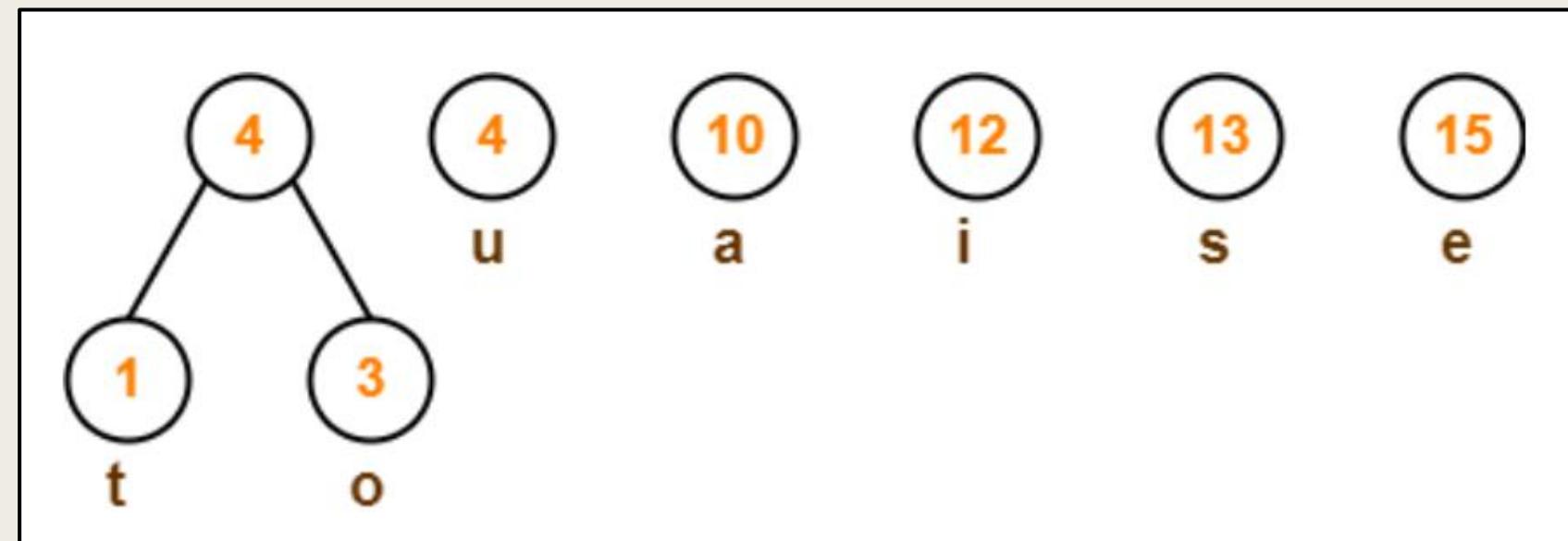
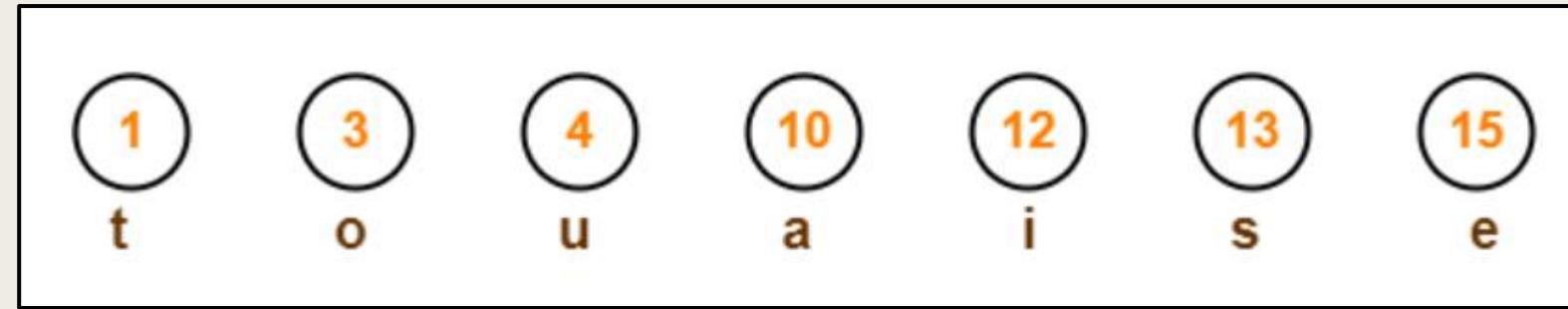
Huffman coding - Problem

- A file contains the following characters with the frequencies as shown. If Huffman Coding is used for data compression, determine-
 - Huffman Code for each character
 - Average code length
 - Length of Huffman encoded message (in bits)

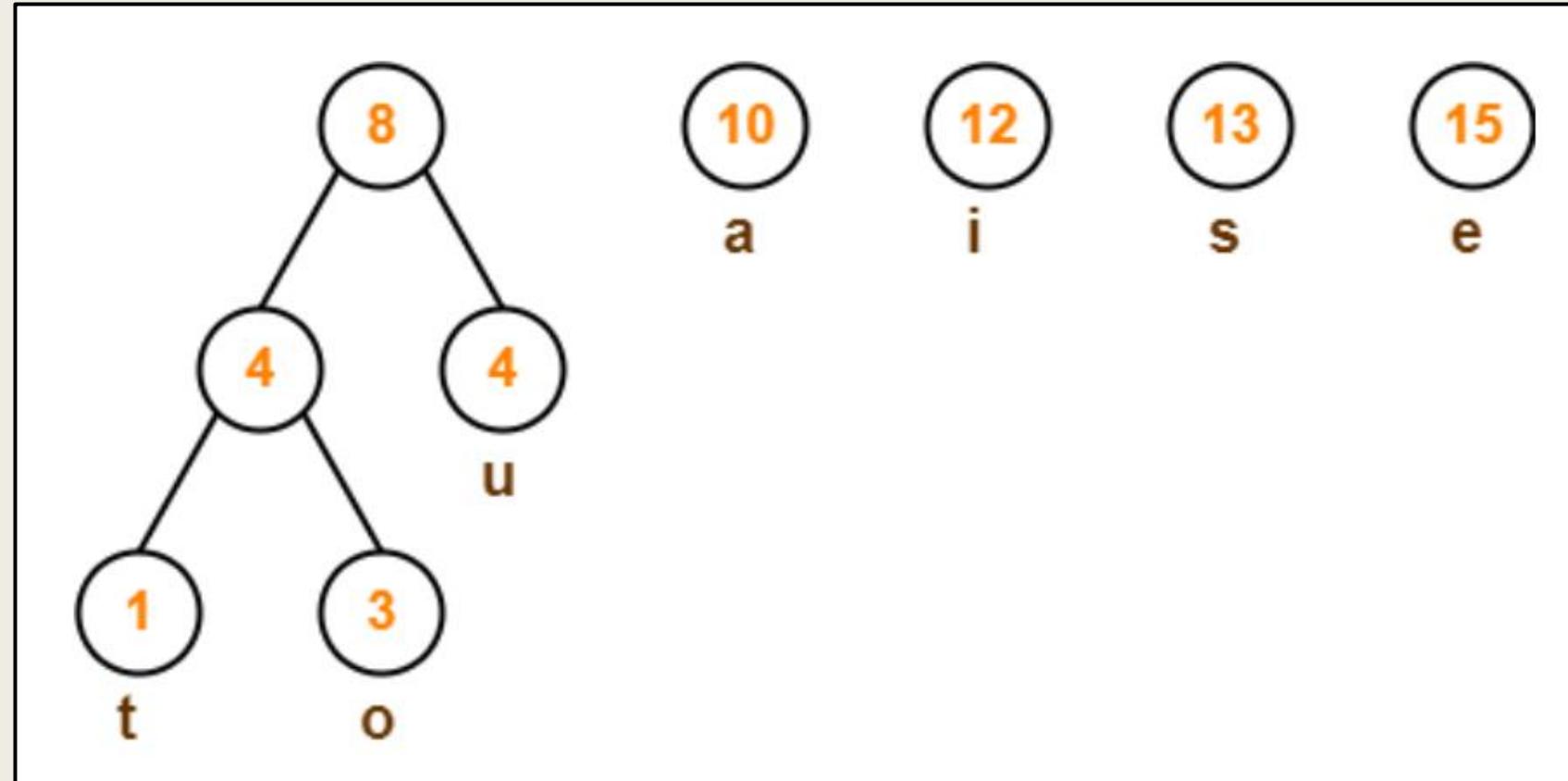
Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Huffman coding

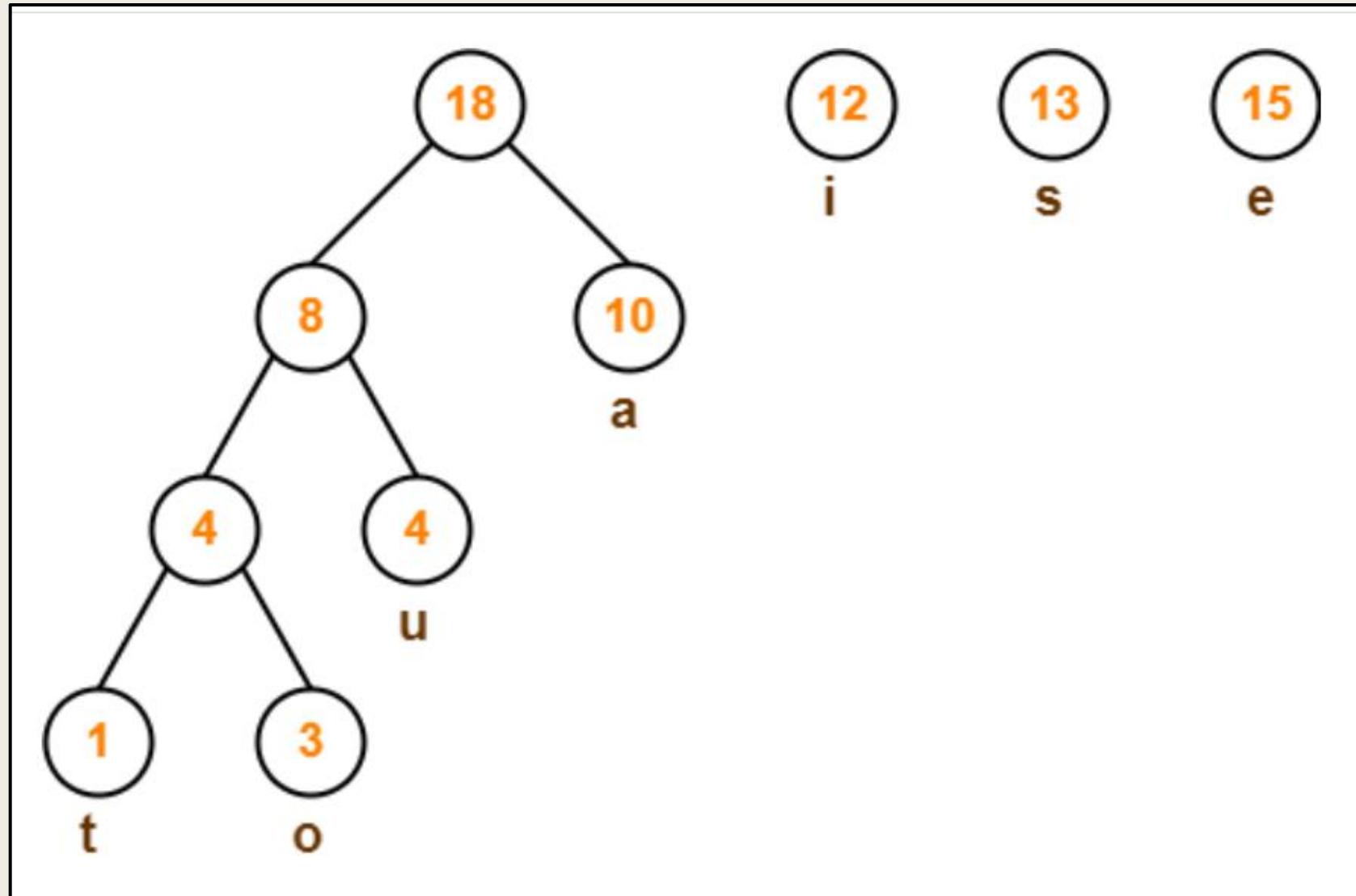
Construction of Huffman tree



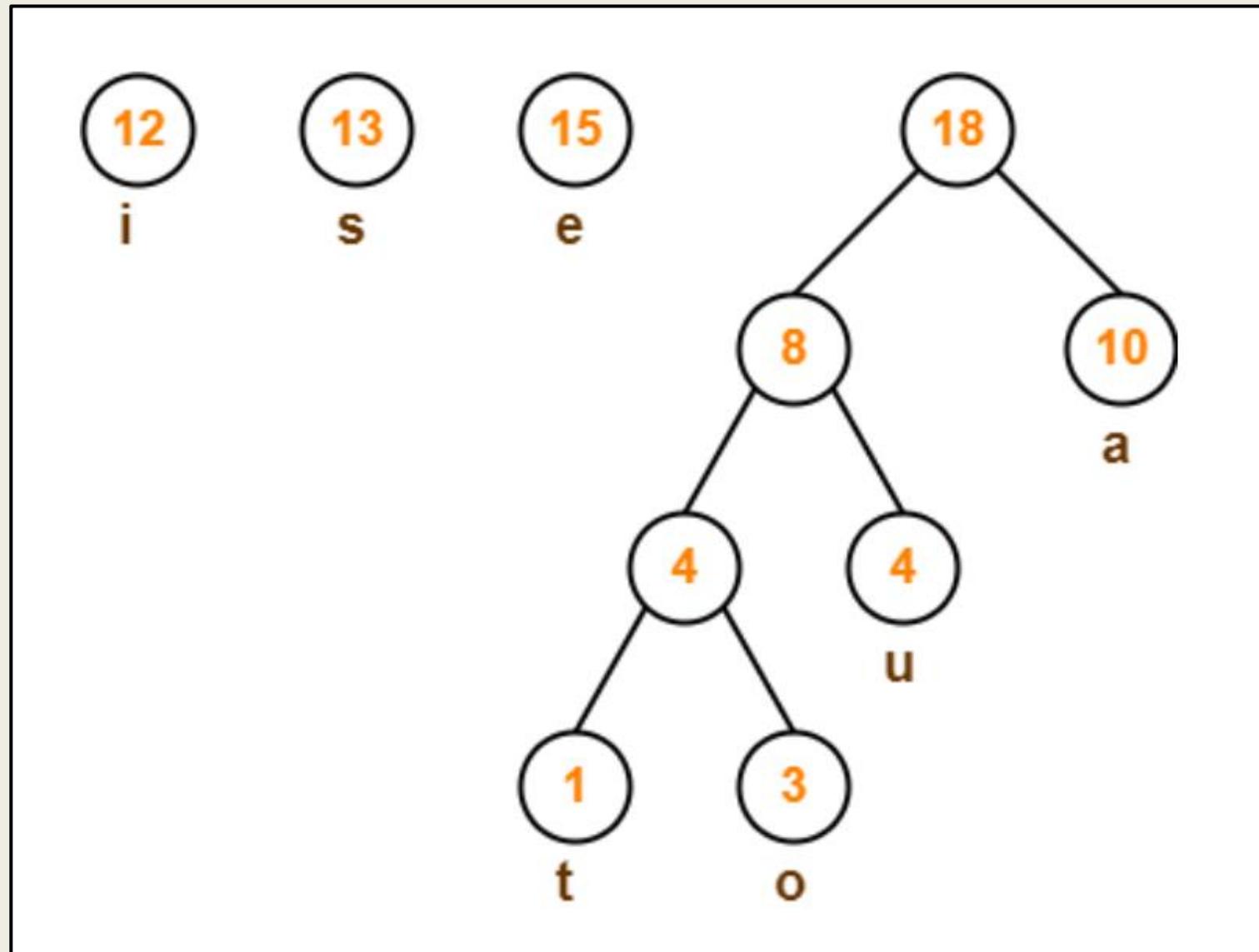
Huffman coding



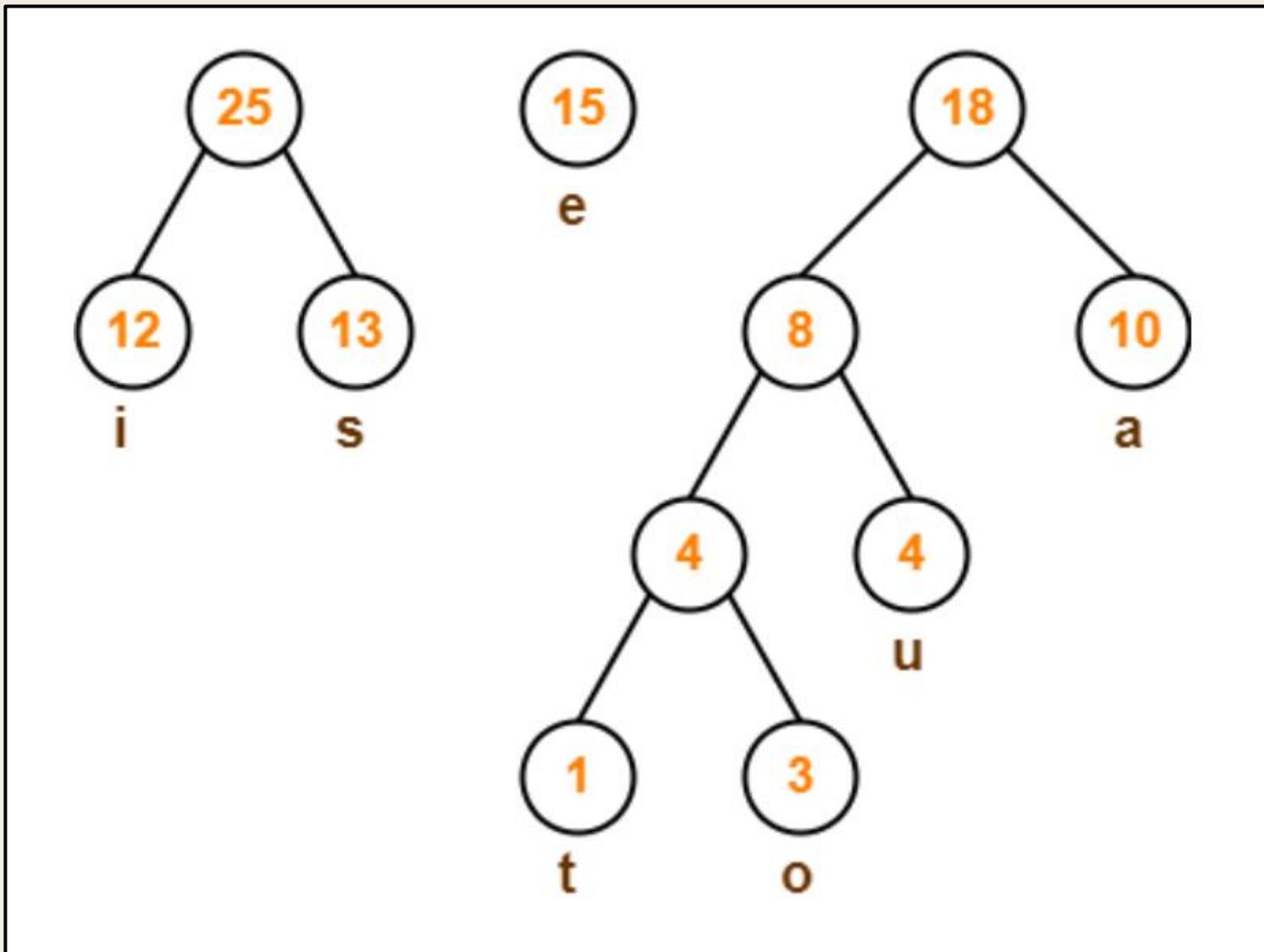
Huffman coding



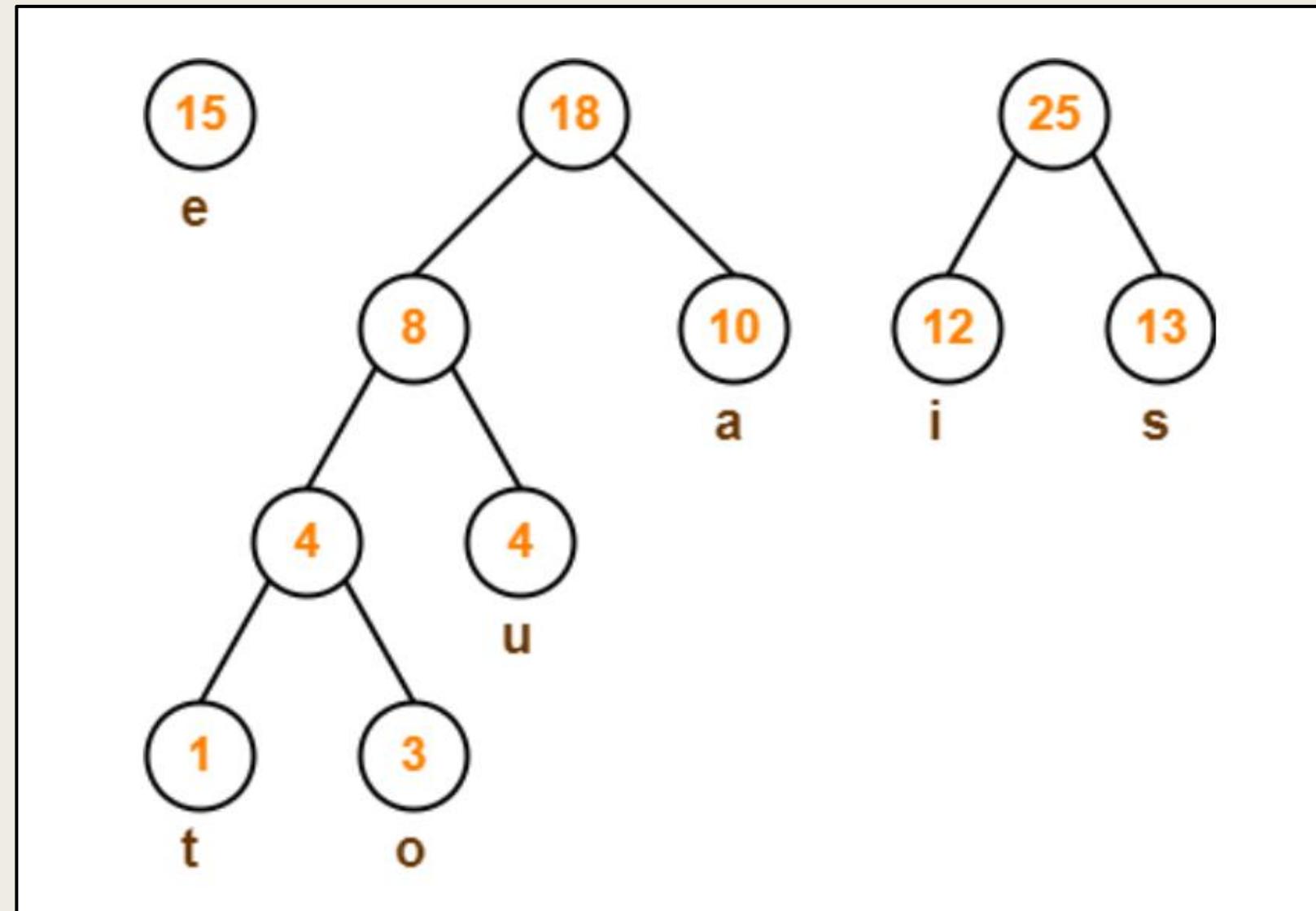
Huffman coding



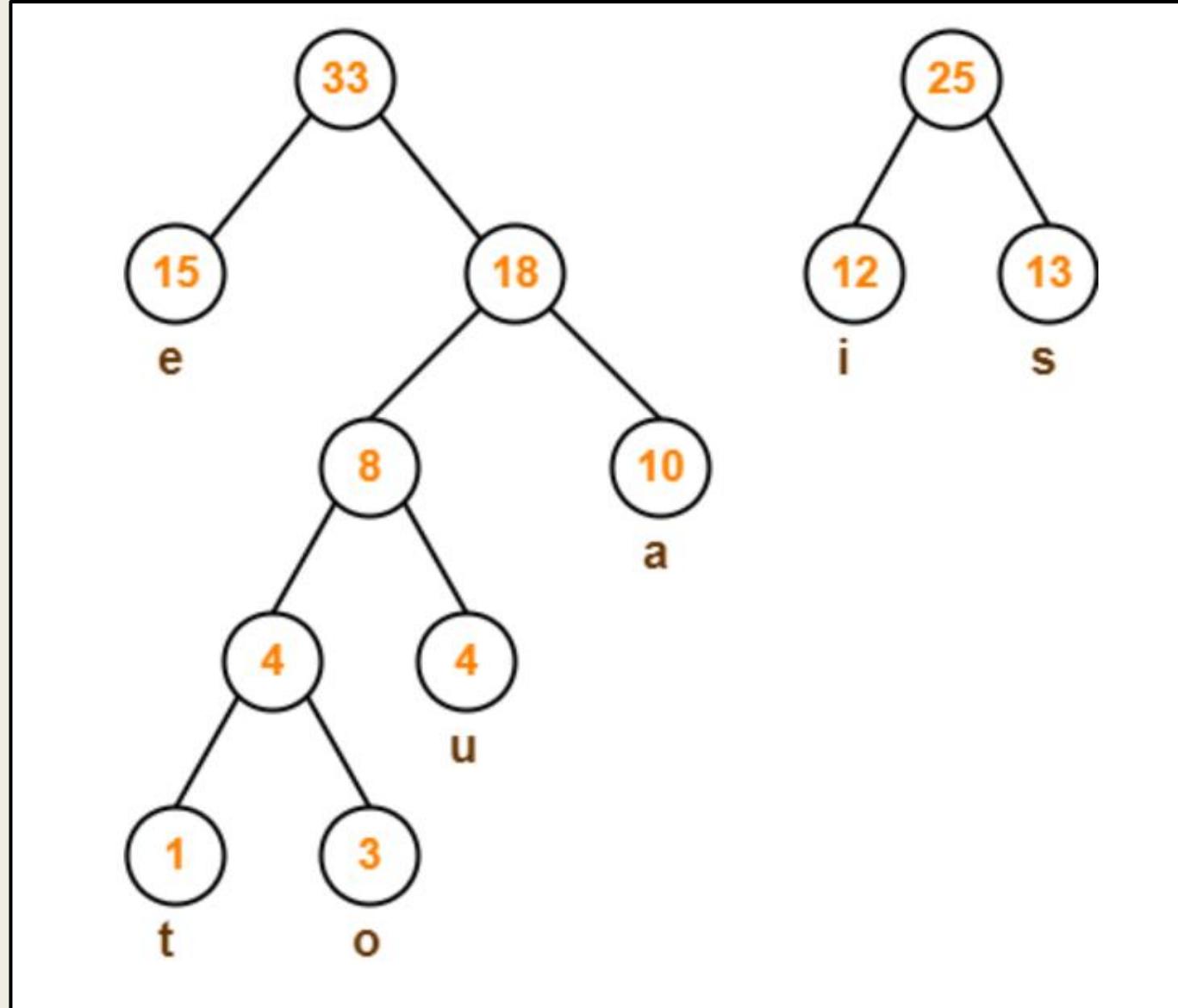
Huffman coding



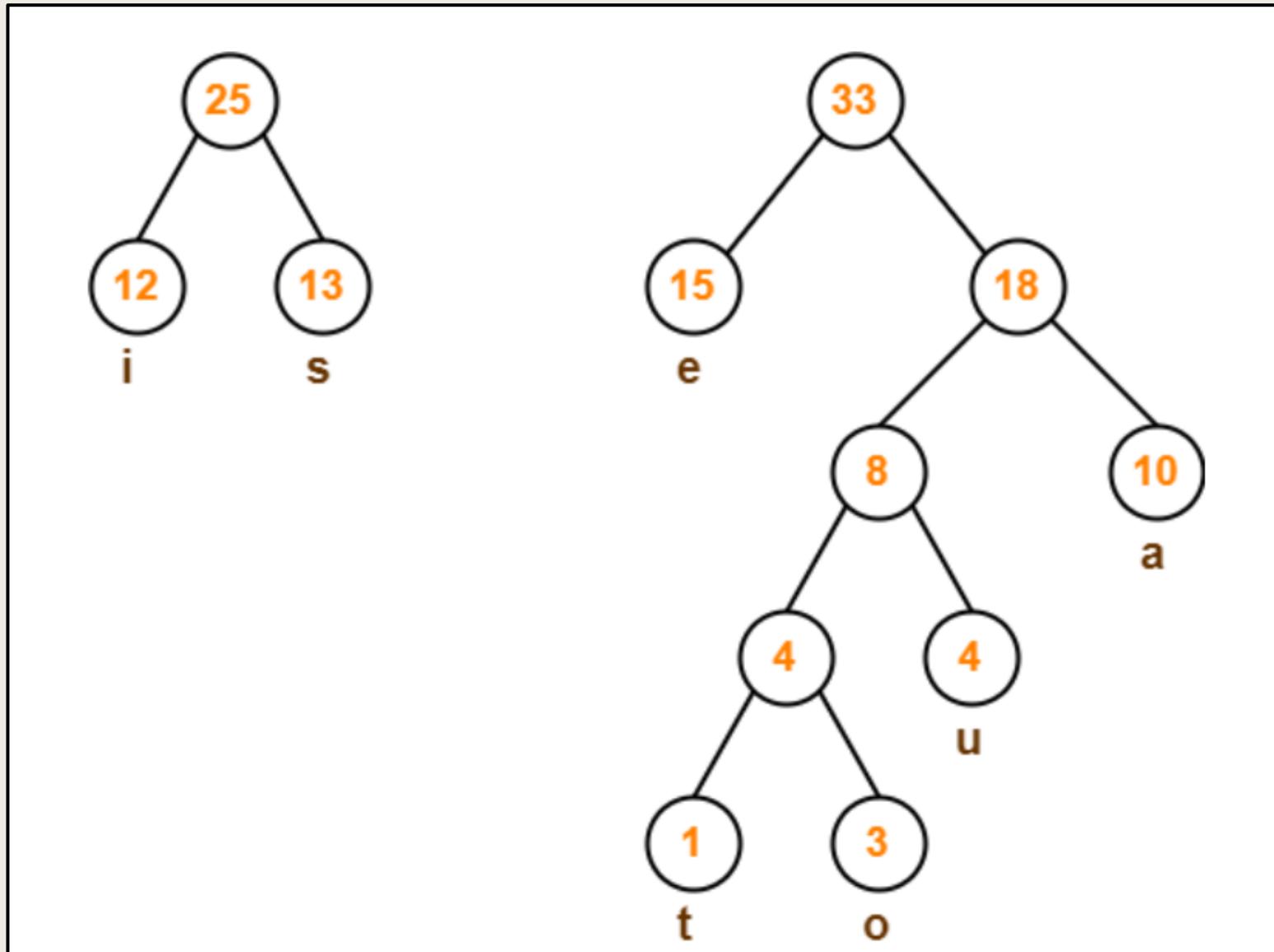
Huffman coding



Huffman coding

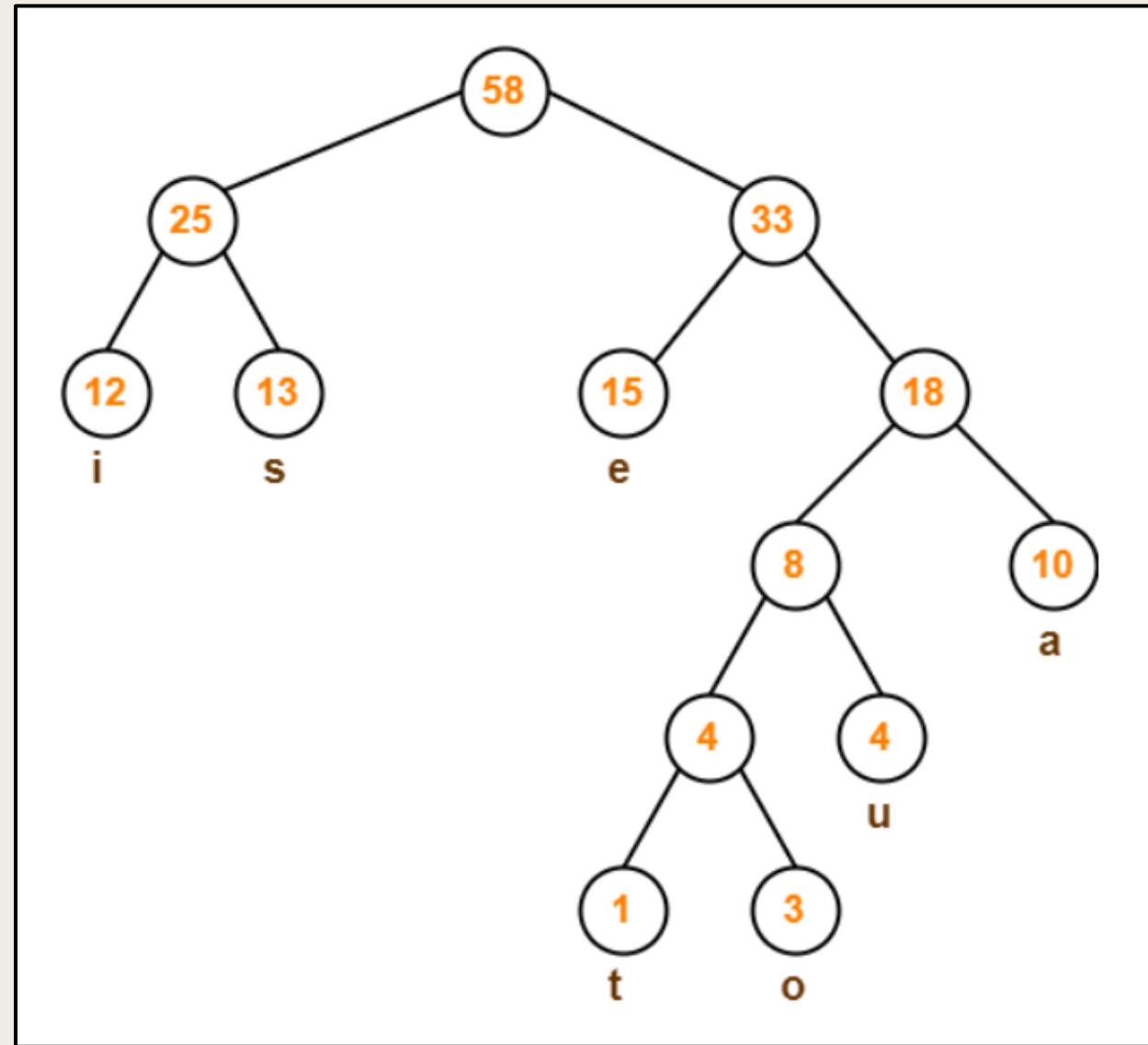


Huffman coding



Huffman coding

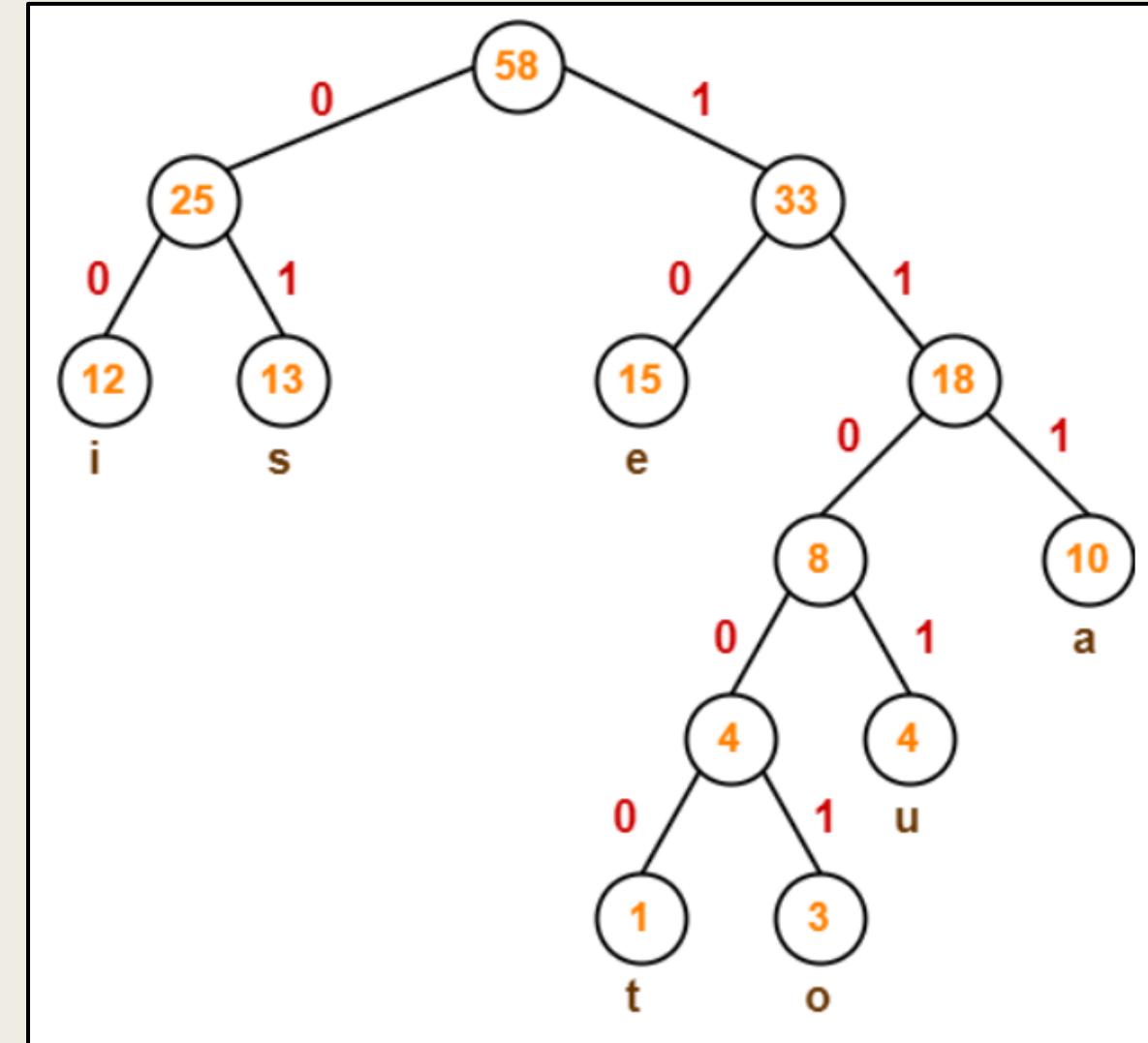
Huffman Tree →



Huffman coding

Assign weight '0' to the left edges and weight '1' to the right edges of the Huffman Tree

a = 111
e = 10
i = 00
o = 11001
u = 1101
s = 01
t = 11000



Huffman coding

- Construct the Huffman tree and hence obtain the codewords for the following population

Value	A	B	C	D	E	F
Frequency	5	25	7	15	4	12

Single source shortest path



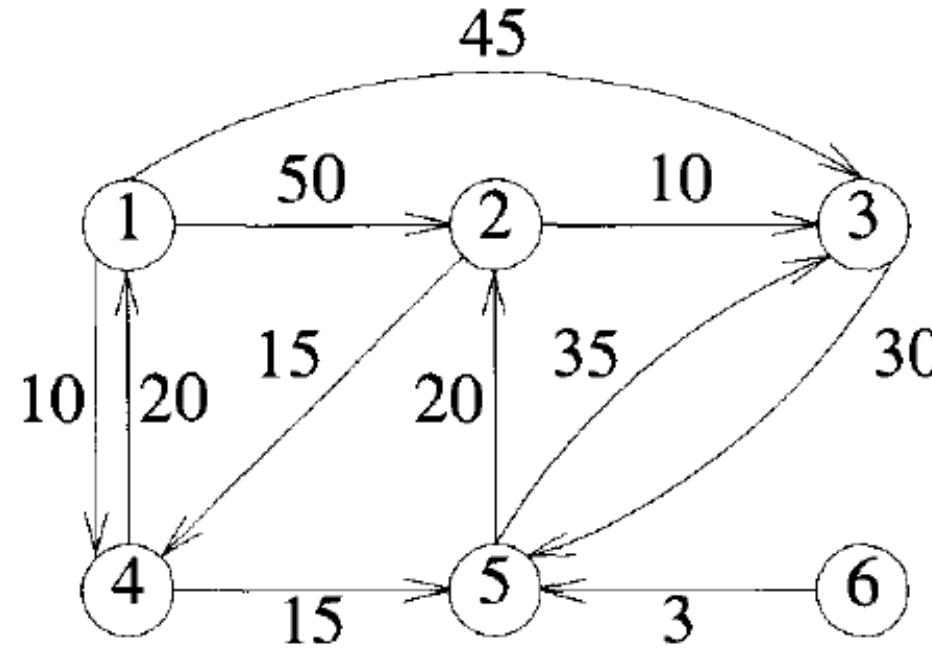
- Graphs can be used to model highway structure of a state/country.
- Vertices representing cities and edges representing the connecting roads
- The edges can be assigned weights which might represent distance/cost/time to drive across the cities
- If Mr. X wants to drive from point A to point B
- Following are the questions raised
- Is there a path from A to B?
- If there are multiple paths then which is the shortest path?

Single source shortest path



- Graphs can be used to model highway structure of a state/country.
- Vertices representing cities and edges representing the connecting roads
- The edges can be assigned weights which might represent distance/cost/time to drive across the cities
- If Mr. X wants to drive from point A to point B
- Following are the questions raised
 - *Is there a path from A to B?*
 - *If there are multiple paths then which is the shortest path?*

Single source shortest path



(a) Graph

<i>Path</i>	<i>Length</i>
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

Single source shortest path

- The problem statement
- Given a weighted connected graph $G=(V, E)$ and a source vertex s , determine the shortest paths from s to all the remaining vertices.
- The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.
- Major applications
 - *Transportation planning*
 - *Packet routing in Internet*
 - *Social networks*
 - *Speech recognition*
 - *Path finding video games*

Single source shortest path

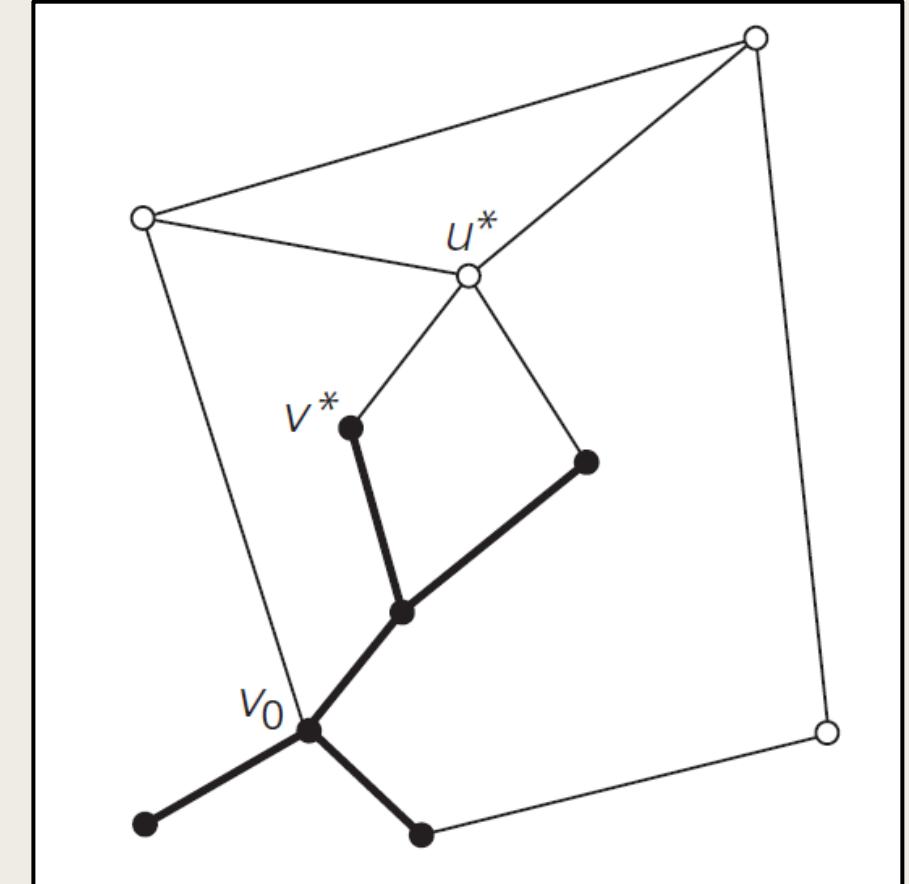


Dijkstra's Algorithm

- **Best-known algorithm for the single-source shortest-paths problem**
- **This algorithm is applicable to undirected and directed graphs with nonnegative weights only**
- Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source
- First, it finds the shortest path from the source to a vertex nearest to it then to a second nearest, and so on
- In general, before its i th iteration commences, the algorithm has already identified the shortest paths to $i - 1$ other vertices nearest to the source.

Single source shortest path

- Working of Dijkstra's algorithm
- Fringe vertices?
 - *The set of vertices adjacent to the vertices in T_i*
- How to identify next nearest vertex? u^*



Single source shortest path

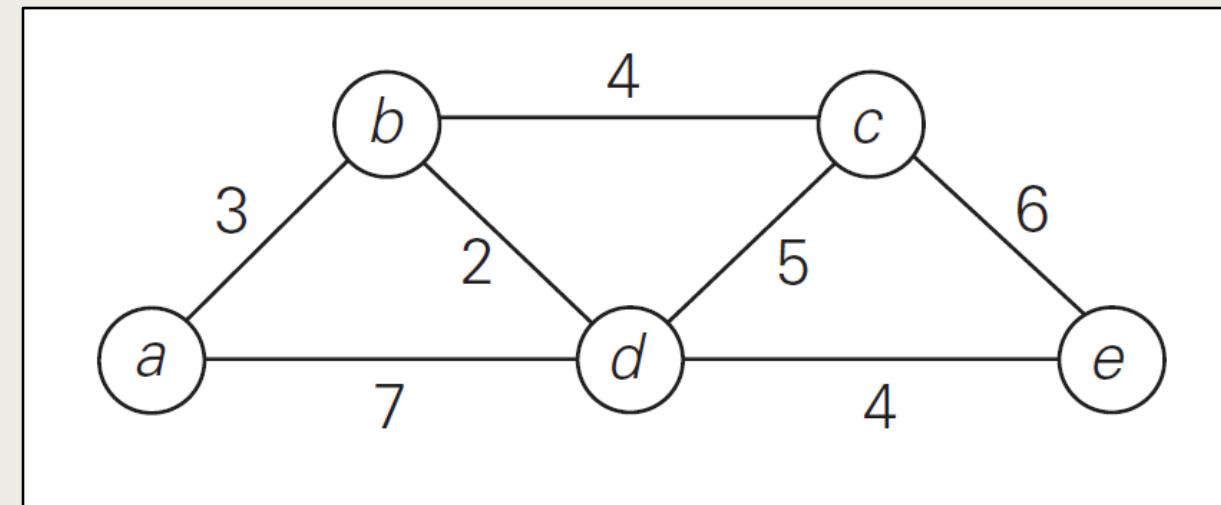
- After identifying do the following
- Move u^* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $du^* + w(u^*, u) < du$, update the labels of u by u^* and $du^* + w(u^*, u)$, respectively.

Single source shortest path

- Find the shortest paths from the source vertex **a** to all other vertices

- Adjacency Matrix

0	3	∞	7	∞
3	0	4	2	∞
∞	4	0	5	6
7	2	5	0	4
∞	∞	6	4	0

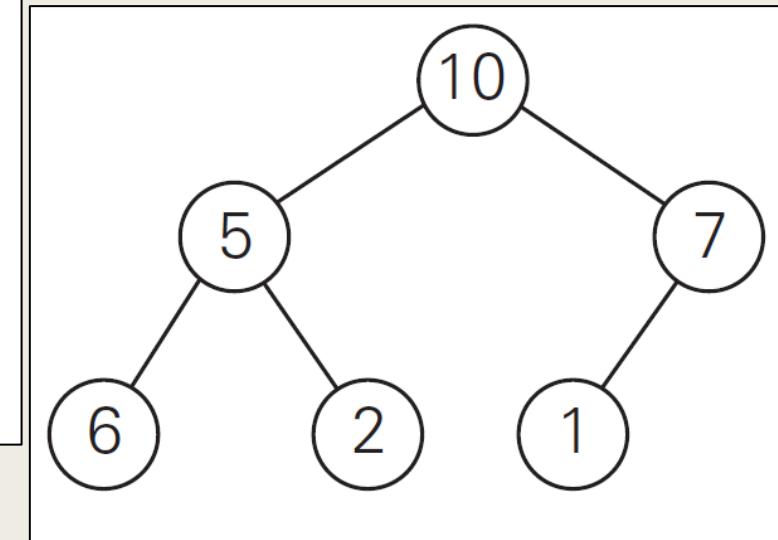
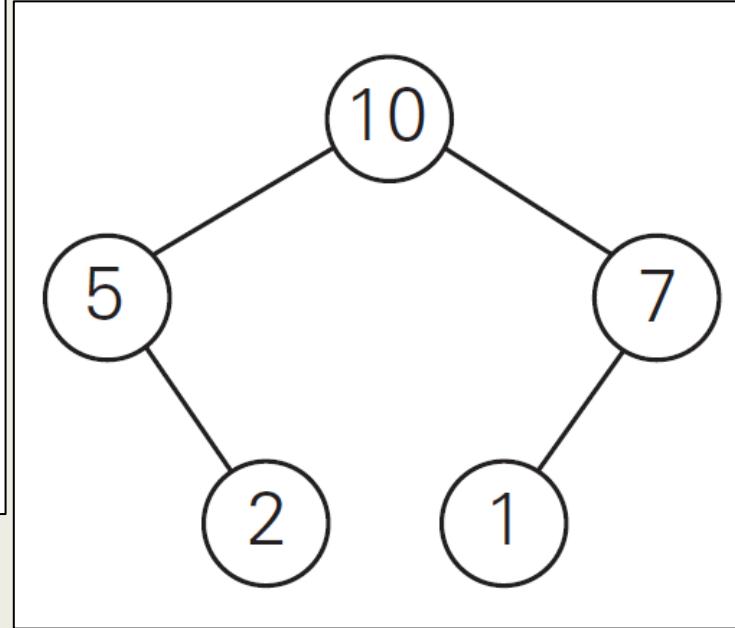
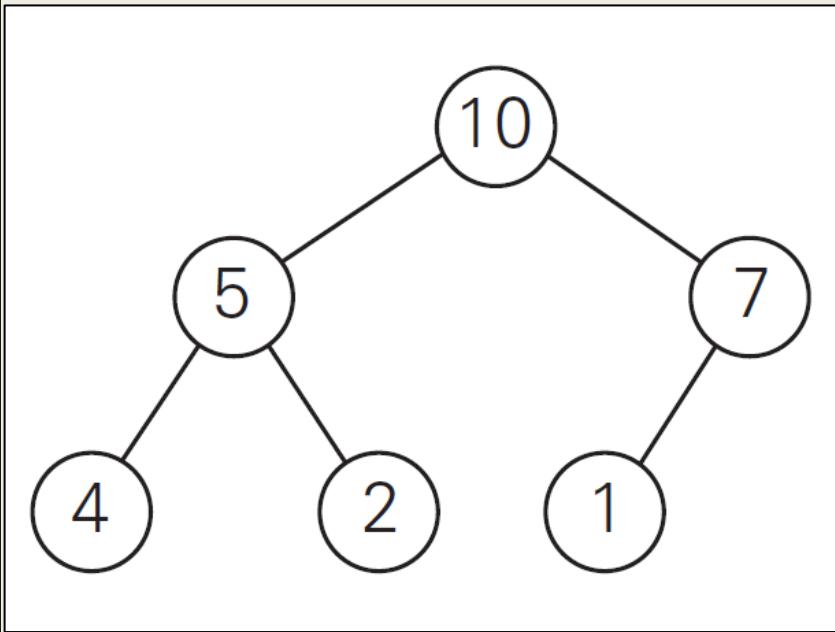


[Solution](#)

Heaps and Heap Sort

- Partially ordered data structure that is especially suitable for implementing priority queues
- A **heap** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided these two conditions are met:
 - **The shape property**
 - The binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
 - **The parental dominance or heap property**
 - The key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

Heaps and Heap Sort



- **Note**
 - key values in a heap are ordered top down
 - there is no left-to-right order in key values;

Heaps and Heap Sort

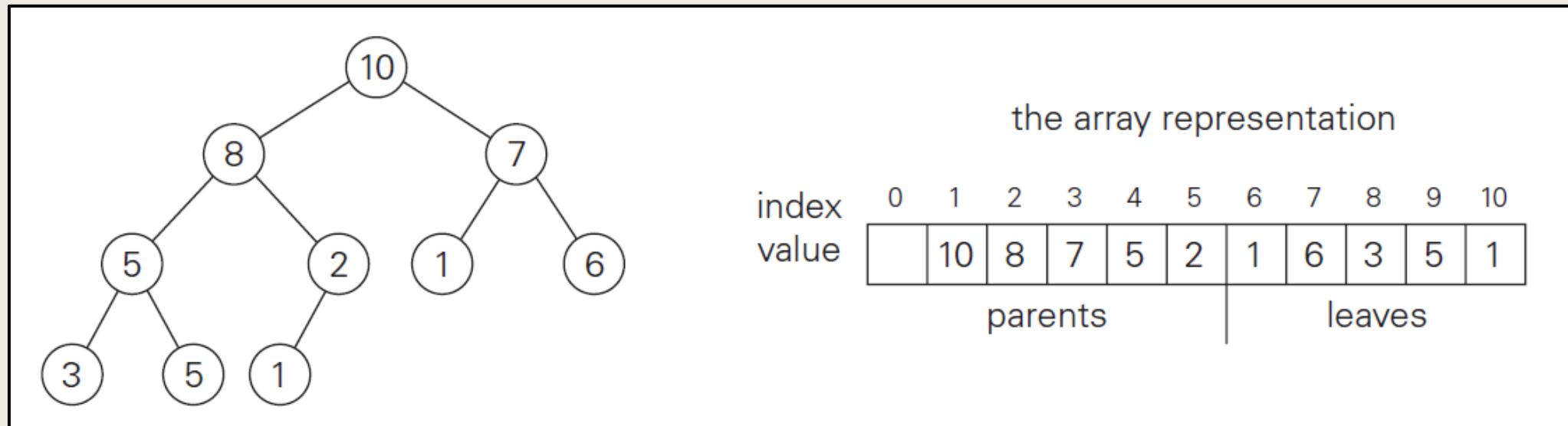
Properties of Heap

- There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\log_2 n$.
- The **root** of a heap always contains its **largest** element.
- A node of a heap considered with all its descendants is also a heap.
- A heap can be implemented as an **array** by recording its elements in the top down, left-to-right fashion
- It is convenient to store the heap's elements in positions 1 through n of such an array, leaving **H[0]** either unused

Heaps and Heap Sort

■ In an array representation of heaps

- the parental node keys will be in the first $[n/2]$ positions of the array, while the leaf keys will occupy the last $[n/2]$ positions;
- the children of a key in the array's parental position $i (1 \leq i \leq n/2)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position $i (2 \leq i \leq n)$ will be in position $[i/2]$



Heaps and Heap Sort

How to construct a heap for a given list of keys?

- **bottom-up heap construction**

- It initializes the essentially complete binary tree with n nodes by placing keys in the order given
 - then “heapifies” the tree

- **top-down heap construction**

- Successive insertions of a new key into a previously constructed heap

Bottom-up heap construction

Example : construct a heap for the following list of numbers

2, 9, 7, 6, 5, 8.

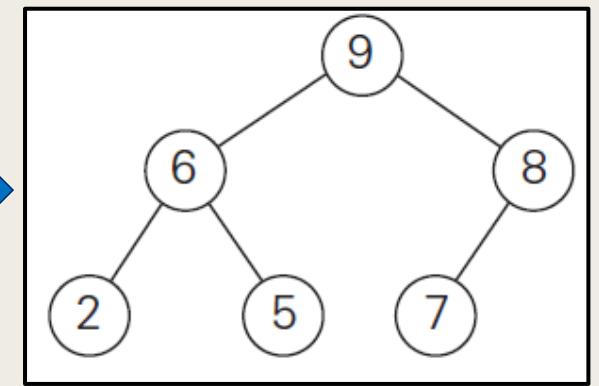
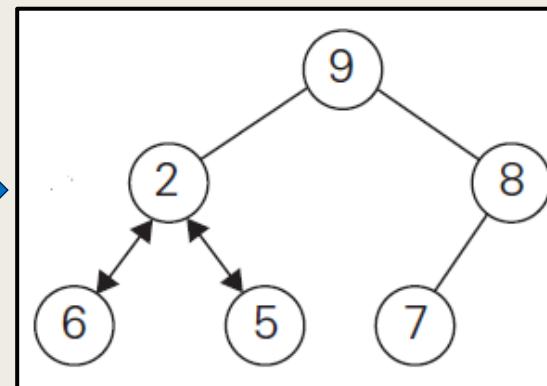
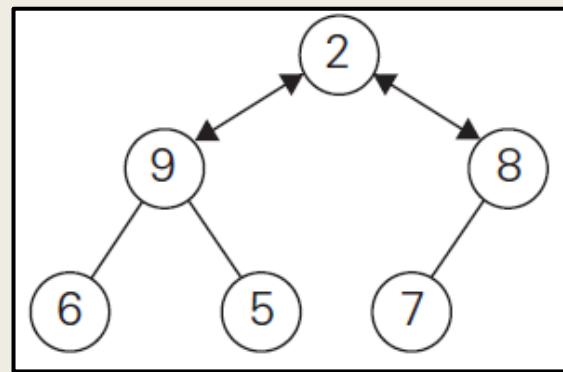
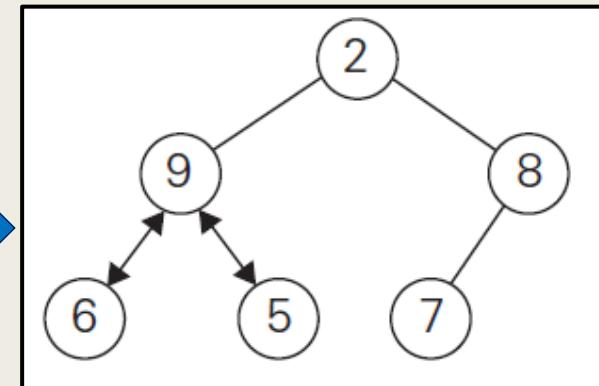
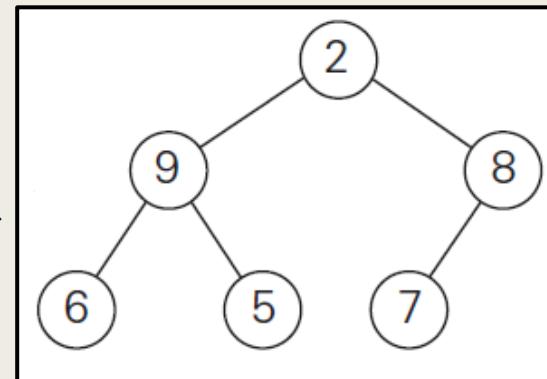
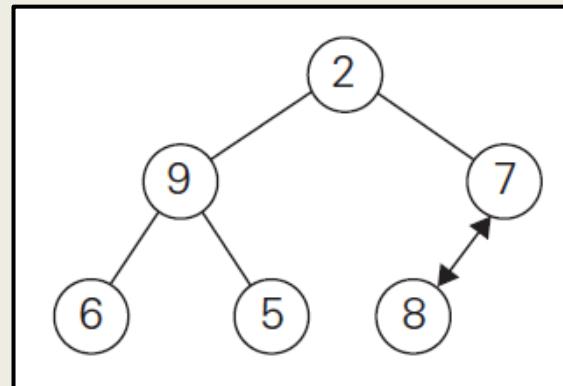
Heaps and Heap Sort

Bottom-up heap construction

- It initializes the essentially complete binary tree with n nodes by placing keys in the order given
- **Heapify**
 - Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node
 - If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position
 - This process continues until the parental dominance for K is satisfied.
 - After completing the “**heapification**” of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor
 - The algorithm stops after this is done for the root of the tree.

Heaps and Heap Sort

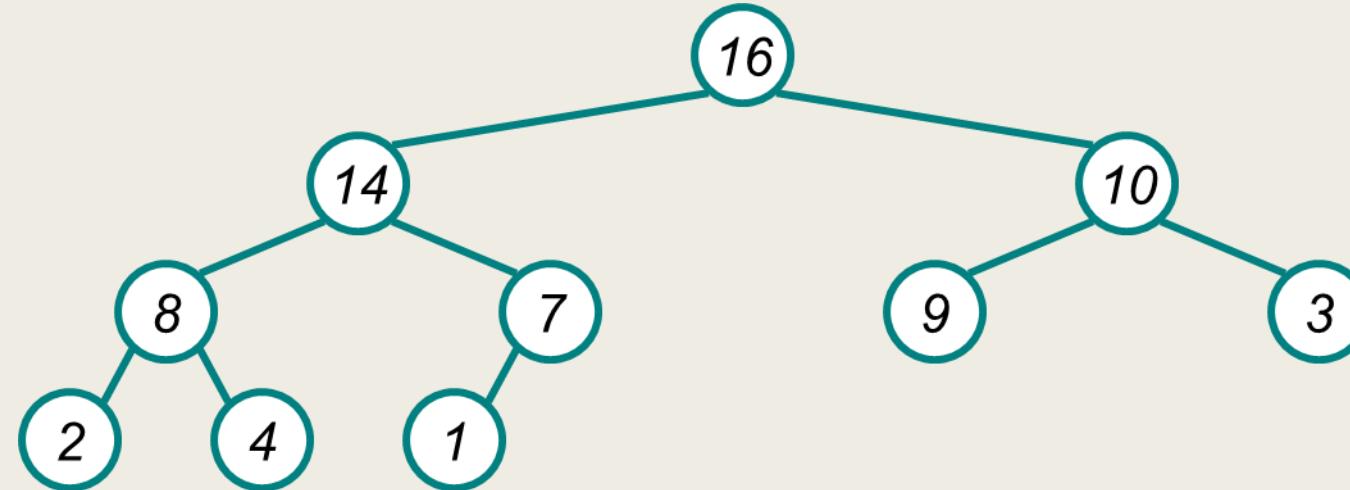
Example : Construct a heap for the following list of numbers
2, 9, 7, 6, 5, 8.



Heaps and Heap Sort

- construct a heap for the following list of numbers

4, 1, 3, 2, 16, 9, 10, 14, 8, 7



Heaps and Heap Sort

ALGORITHM *HeapBottomUp($H[1..n]$)*

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array  $H[1..n]$  of orderable items
//Output: A heap  $H[1..n]$ 
for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1 do
     $k \leftarrow i$ ;  $v \leftarrow H[k]$ 
     $heap \leftarrow \text{false}$ 
    while not  $heap$  and  $2 * k \leq n$  do
         $j \leftarrow 2 * k$ 
        if  $j < n$  //there are two children
            if  $H[j] < H[j + 1]$   $j \leftarrow j + 1$ 
        if  $v \geq H[j]$ 
             $heap \leftarrow \text{true}$ 
        else  $H[k] \leftarrow H[j]$ ;  $k \leftarrow j$ 
     $H[k] \leftarrow v$ 
```

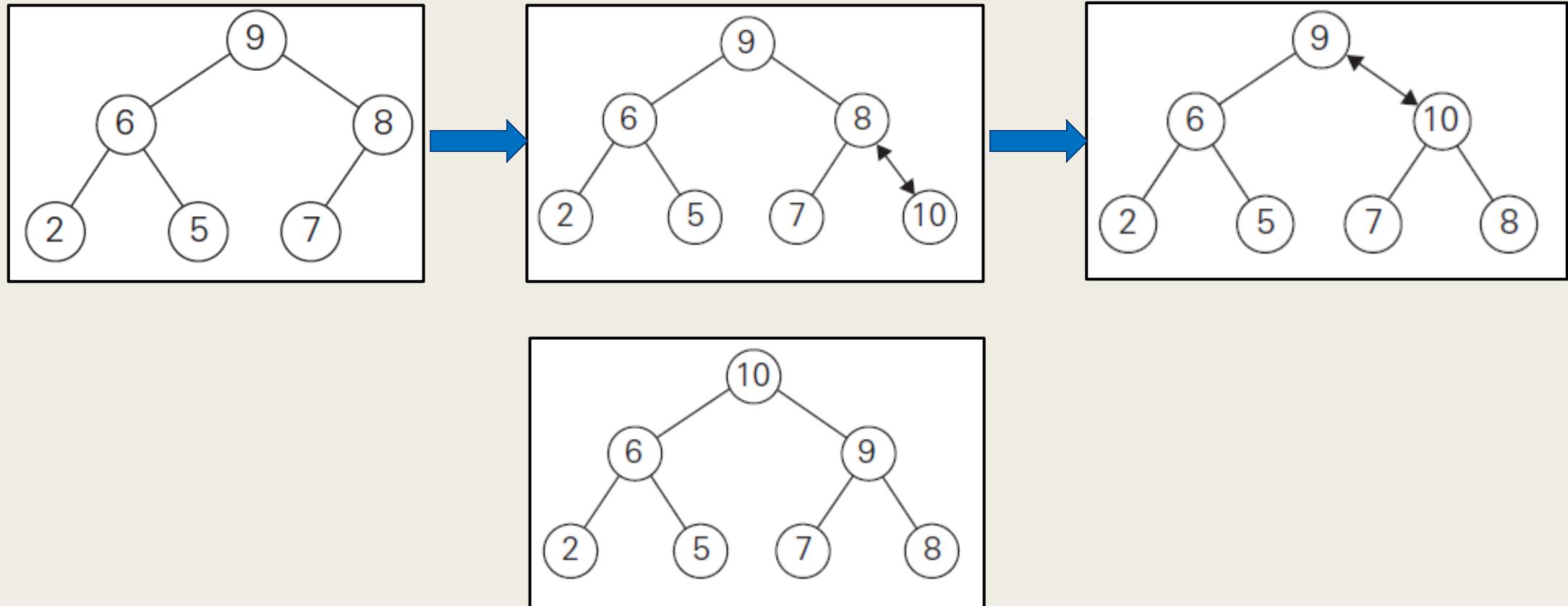
Heaps and Heap Sort

- **Top-down heap construction**
 - Successive insertions of a new key into a previously constructed heap
- **How to insert a new key K into a heap?**
 - First, attach a new node with key K in it after the last leaf of the existing heap.
 - Then shift K up to its appropriate place in the new heap as follows.
 - Compare K with its parent's key: if the latter is greater than or equal to K , stop (the structure is a heap);
 - Otherwise, swap these two keys and compare K with its new parent.
 - This swapping continues until K is not greater than its last parent or it reaches the root

Heaps and Heap Sort

■ top-down heap construction

– Successive insertions of a new key into a previously constructed heap



Heaps and Heap Sort

- How to delete an item form the heap?
- Special case- **Deletion of root**

Maximum Key Deletion from a heap

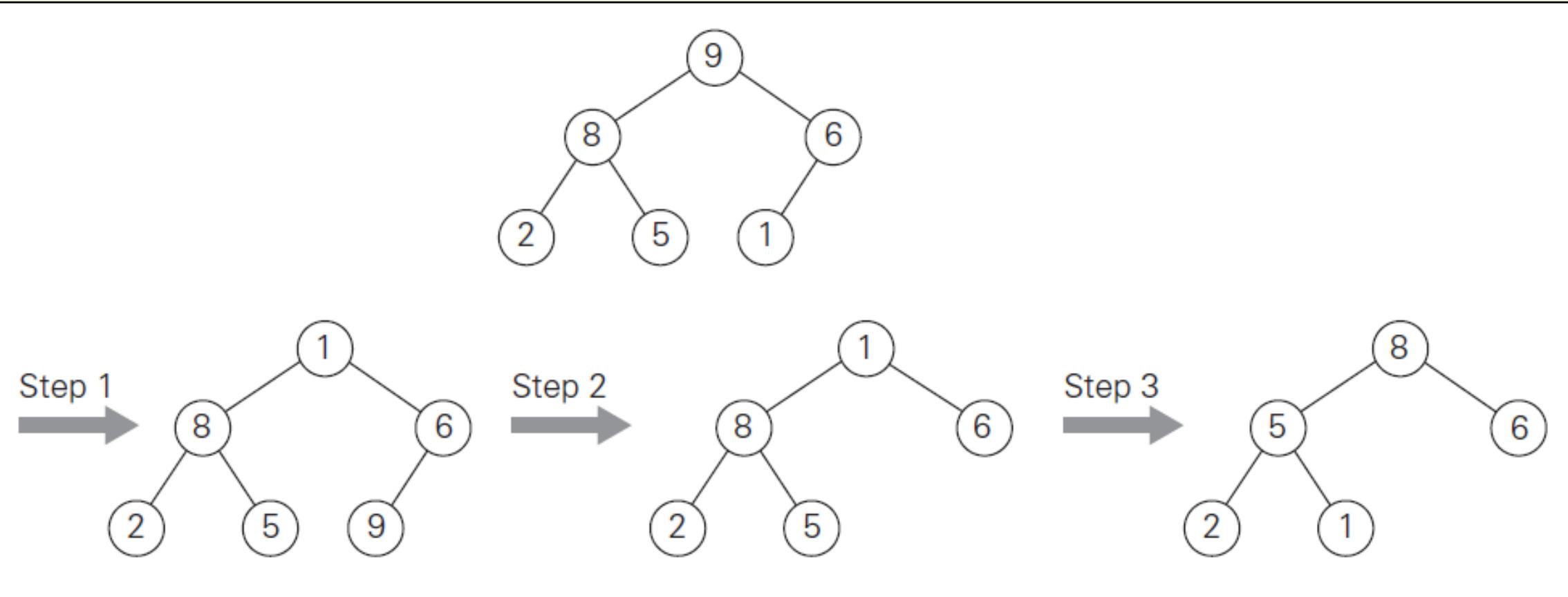
Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 “Heapify” the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K : if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Heaps and Heap Sort

Steps illustrating the deletion of root element from the heap



Heaps and Heap Sort

Heap Sort

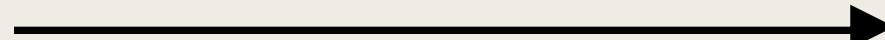
- An interesting sorting algorithm discovered by J. W. J. Williams
- This is a two-stage algorithm that works as follows.
 - Stage 1 (heap construction): Construct a heap for a given array.
 - Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

Heaps and Heap Sort

Sort the following elements using heap sort

2, 9, 7, 6, 5, 8

Stage 1 (heap construction)						
2	9	7	6	5	8	
2	9	8	6	5	7	
2	9	8	6	5	7	
9	2	8	6	5	7	
9	6	8	2	5	7	



Stage 2 (maximum deletions)						
9	6	8	2	5	7	
7	6	8	2	5		9
8	6	7	2	5		
5	6	7	2		8	
7	6	5	2			
2	6	5		7		
6	2	5				
5	2		6			
5	2					
2		5				
						2



GREEDY METHOD