DAA – Unit III

Dr.S.Prasanna Devi
Professor & Head, CSE Department

SYLLABUS

Duration (hour)		15	15	15	15	15
S-1	SLO-1	*Introduction-Algorithm Design*	*Introduction-Divide and Conquer*	*Introduction-Greedy and Dynamic Programming*	*Introduction to backtracking - branch and bound*	*Introduction to randomization and approximation algorithm*
	SLO-2	*Fundamentals of Algorithms*	*Maximum Subarray Problem*	*Examples of problems that can be solved by using greedy and dynamic approach*	*N queen's problem - backtracking*	*Randomized hiring problem*
S-2	SLO-1	*Correctness of algorithm*	*Binary Search*	*Huffman coding using greedy approach*	*Sum of subsets using backtracking*	*Randomized quick sort*
	SLO-2	*Time complexity analysis*	*Complexity of binary search*	*Comparison of brute force and Huffman method of encoding*	*Complexity calculation of sum of subsets*	*Complexity analysis*
S-3	SLO-1	*Insertion sort-Line count, Operation count*	*Merge sort*	*Knapsack problem using greedy approach*	*Graph introduction*	*String matching algorithm*
	SLO-2	*Algorithm Design paradigms*	*Time complexity analysis*	*Complexity derivation of knapsack using greedy*	*Hamiltonian circuit - backtracking*	*Examples*
S 4-5	SLO-1 SLO-2	*Lab 1: Simple Algorithm-Insertion sort*	*Lab 4: Quicksort, Binary search*	*Lab 7: Huffman coding, knapsack and using greedy*	*Lab 10: N queen's problem*	*Lab 13: Randomized quick sort*
S-6	SLO-1	*Designing an algorithm*	*Quick sort and its Time complexity analysis*	*Tree traversals*	*Branch and bound - Knapsack problem*	*Rabin Karp algorithm for string matching*
	SLO-2	*And its analysis-Best, Worst and Average case*	*Best case, Worst case, Average case analysis*	*Minimum spanning tree - greedy Kruskal's algorithm - greedy*	*Example and complexity calculation. Differentiate with dynamic and greedy*	*Example discussion*
S-7	SLO-1	*Asymptotic notations Based on growth functions.*	*Strassen's Matrix multiplication and its recurrence relation*	*Minimum spanning tree - Prims algorithm*	*Travelling salesman problem using branch and bound*	*Approximation algorithm*
	SLO-2	*O,o,Θ, ω, Ω*	*Time complexity analysis of Merge sort*	*Introduction to dynamic programming*	*Travelling salesman problem using branch and bound example*	*Vertex covering*
S-8	SLO-1	*Mathematical analysis*	*Largest sub-array sum*	*0/1 knapsack problem*	*Travelling salesman problem using branch and bound example*	*Introduction Complexity classes*
	SLO-2	*Induction, Recurrence relations*	*Time complexity analysis of Largest sub-array sum*	*Complexity calculation of knapsack problem*	*Time complexity calculation with an example*	*P type problems*
S 9-10	SLO-1 SLO-2	*Lab 2: Bubble Sort*	*Lab 5: Strassen Matrix multiplication*	*Lab 8: Various tree traversals, Krukshall's MST*	*Lab 11: Travelling salesman problem*	*Lab 14: String matching algorithms*
S-11	SLO-1	*Solution of recurrence relations*	*Master Theorem Proof*	*Matrix chain multiplication using dynamic programming*	*Graph algorithms*	*Introduction to NP type problems*
	SLO-2	*Substitution method*	*Master theorem examples*	*Complexity of matrix chain multiplication*	*Depth first search and Breadth first search*	*Hamiltonian cycle problem*
S-12	SLO-1	*Solution of recurrence relations*	*Finding Maximum and Minimum in an array*	*Longest common subsequence using dynamic programming*	*Shortest path introduction*	*NP complete problem introduction*
	SLO-2	*Recursion tree*	*Time complexity analysis-Examples*	*Explanation of LCS with an example*	*Floyd-Warshall Introduction*	*Satisfiability problem*
S-13	SLO-1	*Solution of recurrence relations*	*Algorithm for finding closest pair problem*	*Optimal binary search tree (OBST)using dynamic programming*	*Floyd-Warshall with sample graph*	*NP hard problems*
	SLO-2	*Examples*	*Convex Hull problem*	*Explanation of OBST with an example.*	*Floyd-Warshall complexity*	*Examples*
S 14-15	SLO-1 SLO-2	*Lab 3: Recurrence Type-Merge sort, Linear search*	*Lab 6: Finding Maximum and Minimum in an array, Convex Hull problem*	*Lab 9: Longest common subsequence*	*Lab 12: BFS and DFS implementation with array*	*Lab 15: Discussion over analyzing a real time problem*

INTRODUCTION TO GREEDY PROGRAMMING

- The greedy method is a straight forward method, because the decision for the solution is taken based on the information that is available. Here the solution is constructed through a sequence of steps, each steps are expanding a partially constructed solution obtained so far. This procedure is repeated until complete solution to the problem is reached. At each step the choices are made to be
- **Feasible :** It has to satisfy the problem's constraints.
- **Locally Optimal :** It has to be the best local choice among all feasible choices available on that step.
- **Irrevocable :** once made, it cannot be changed on subsequent steps of the algorithm.
- In general, greedy algorithms gives the optimal solution.

HUFFMAN CODING USING GREEDY APPROACH

Example:

- Consider the five character alphabet {A, B, C, D, -} with the following occurrence probabilities:

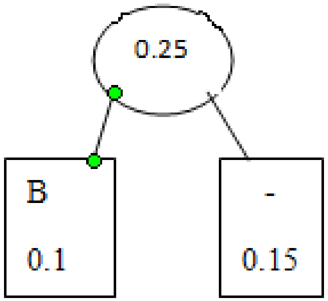
Character	A	B	C	D	-
Probability	0.35	0.1	0.2	0.2	0.15

Step1 : List the probability in ascending order.

Character	B	-	C	D	A
Probability	0.1	0.15	0.2	0.2	0.35

Step 2: Tie the Smallest 2 probability.

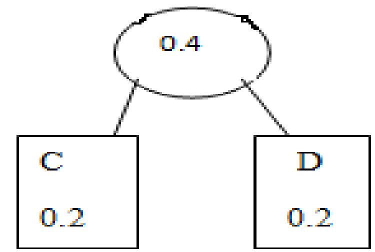
Character	B	-	C	D	A
Probability	0.1	0.15	0.2	0.2	0.35



HUFFMAN CODING USING GREEDY APPROACH

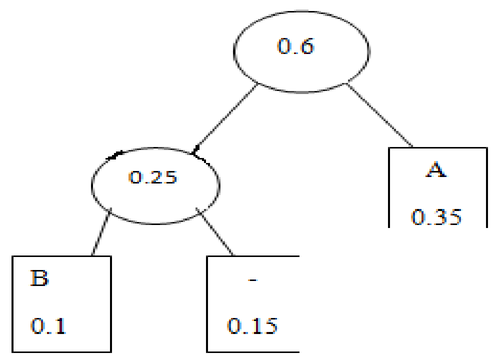
Step 3: Consider the new root as new probability.

C	D	New1		A
0.2	**0.2**	**0.25**		**0.35**
		B	-	
		0.1	0.15	



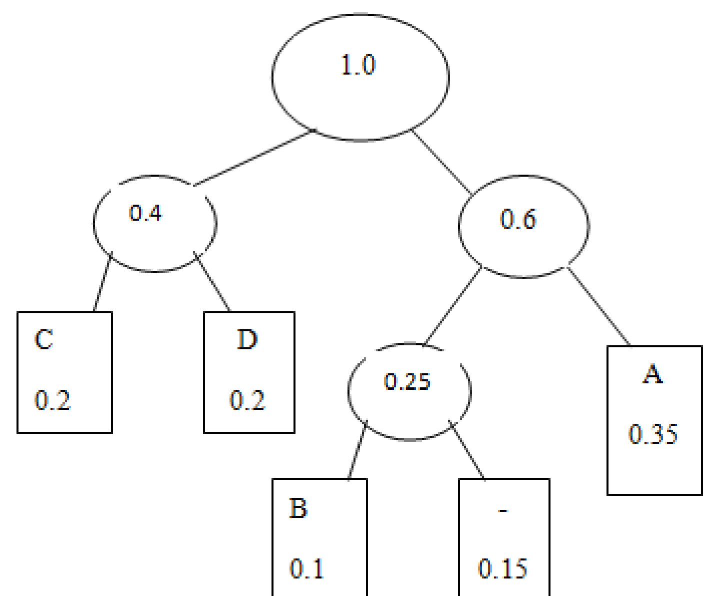
Step 4:

New1		A	New2	
0.25		**0.35**	**0.4**	
B	-		C	D
0.1	0.15		0.2	00.2



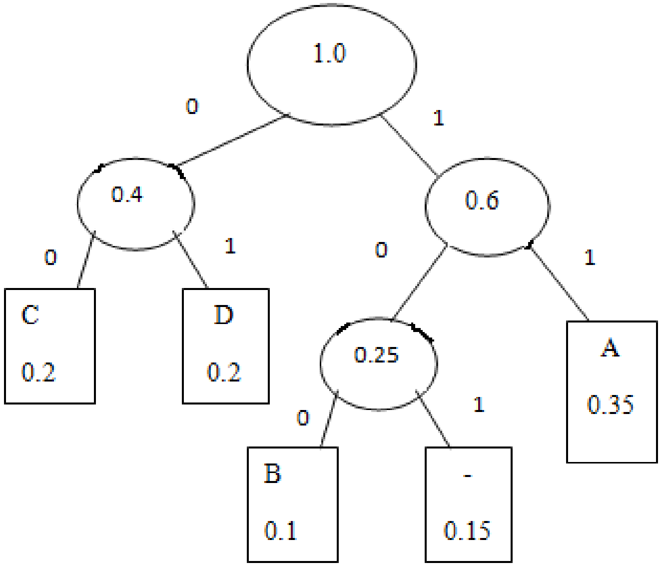
Step 5:

New2		New3		
0.4		**0.6**		
C	D	0.25		A
0.2	0.2	B	-	0.35



HUFFMAN CODING USING GREEDY APPROACH

Final Tree & Code Table



Character	Probability	Codeword
A	0.35	11
B	0.1	100
C	0.2	00
D	0.2	01
-	0.15	101

- Its encoding is used in file compression algorithm.
- It is used in transmission of data in the form of encoding.
- This encoding is used in game playing method in which decision trees need to be formed

$$\text{No.of bits per character} = \sum_{i=1}^{n} (\text{length of codeword} * \text{frequency of corresponding character})$$

HUFFMAN CODING USING GREEDY APPROACH

Huffman(C)

$n \leftarrow |C|$

$Q \leftarrow C$

O(n log n) for creating a priority Queue

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

do allocate a new node z

$left[z] \leftarrow x \leftarrow$ Extract-Min(Q)

$right[z] \leftarrow y \leftarrow$ Extract-Min(Q)

$f(z) \leftarrow f(x) + f(y)$

Insert(Q, z)

return Extract-Min(Q)

Complexity: $O(n \lg n)$

$$\text{Average code length per character} = \frac{\Sigma\,(\text{frequency}_i \times \text{code length}_i)}{\Sigma\,\text{frequency}_i}$$

$$= \Sigma\,(\text{probability}_i \times \text{code length}_i)$$

COMPARISON OF BRUTE FORCE AND HUFFMAN METHOD OF ENCODING

Message : BCCABBDDAECCBBAEDDCC

1000010,1000011, 1000011,10000001....

ASCII Alphabet			
A	1000001	N	1001110
B	1000010	O	1001111
C	1000011	P	1010000
D	1000100	Q	1010001
E	1000101	R	1010010
F	1000110	S	1010011
G	1000111	T	1010100
H	1001000	U	1010101
I	1001001	V	1010110
J	1001010	W	1010111
K	1001011	X	1011000
L	1001100	Y	1011001
M	1001101	Z	1011010

Brute Force

Char	Count	Code
A	3	001
B	5	10
C	6	11
D	4	01
E	2	000

Huffman Encoding

Total = 20 letters * 8 bits = 160 bits.

Message Size = (3*3 + 5*2 + 6*2+ 4*2+2*3) = 45 +

Code table length = 5 chars * 8bits (ascii)= 40 +

Code lengths = 3+2+2+2+3 = 12;

Total size = 45+40+12 = 97.

KNAPSACK USING GREEDY APPROACH

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
    do x[i] = 0
  weight = 0
for i = 1 to n
            if weight + w[i] ≤ W then
            x[i] = 1
            weight = weight + w[i]
            else x[i] = (W - weight) / w[i]
            weight = W - weight
            break
return x
```

FRACTIONAL KNAPSACK PROBLEM

Capacity=15

Objects i	1	2	3	4	5	6	7	8
Profit p	10	9	5	8	3	12	9	15
Weight w	3	5	1	2	1	9	6	4
Prof/wt. p/w	3.33	1.8	5	4	3	1.33	1.5	3.75

Sort the array in decreasing order according to the ratio of profit/weight

Objects i	1	2	3	4	5	6	7	8
Prof/wt. p/w	5	4	3.75	3.33	3	1.8	1.5	1.33
Profit p	5	8	15	10	3	9	9	12
Weight w	1	2	4	3	1	5	6	9

Capacity	Total Profit	Profit/weight
15	0	0
15-1= 14	5	5
14-2= 12	5+8=13	4
12-4 = 8	13+15=28	3.75
8- 3 = 5	28+10=38	3.33
5 – 1 = 4	38+ 3= 41	3

Next item has weight 5 but capacity is 4, so we place a fraction of it

Capacity	Total Profit	Profit/weight
0	41 + 1.8*4=48.2	1.8

~algoskills

COMPLEXITY DERIVATION OF FRACTIONAL KNAPSACK USING GREEDY APPROACH

- Time taken for calculating Pi/Wi = O(n)
- Time taken for sorting Pi/Wi = O(n log n)
- Total complexity = O(n) + O(n logn)= O(n log n)

TREE TRAVERSALS

LEVEL ORDER TRAVERSAL

$T(n) = O(n)$

$S(n) = O(n/2)$ – Avg/Worst Case

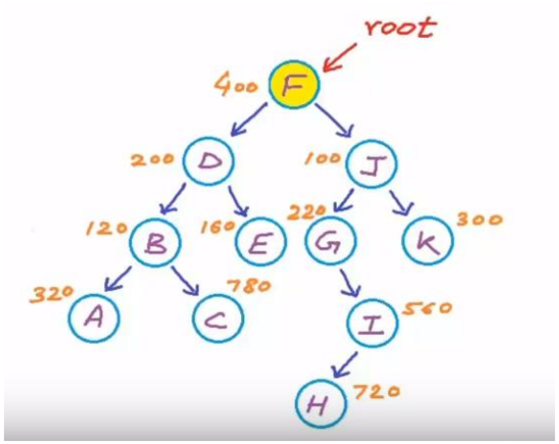
$S(n)= O(1)$ – Best Case

FDJBEGKACIH

Queue

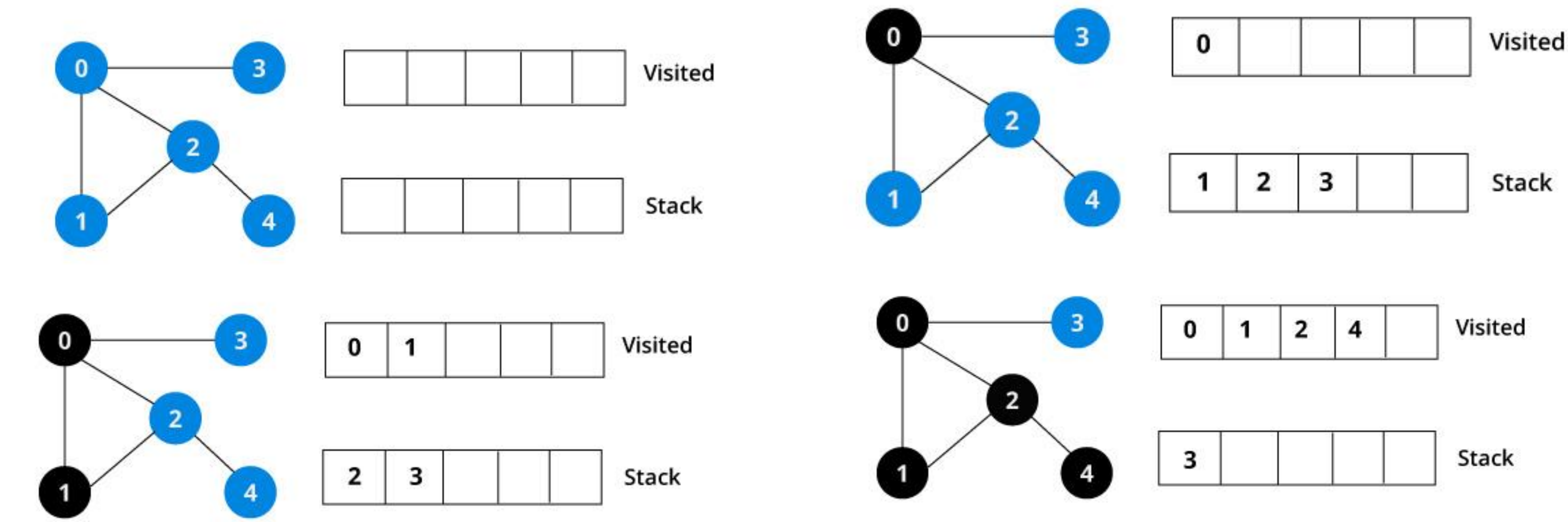
400			

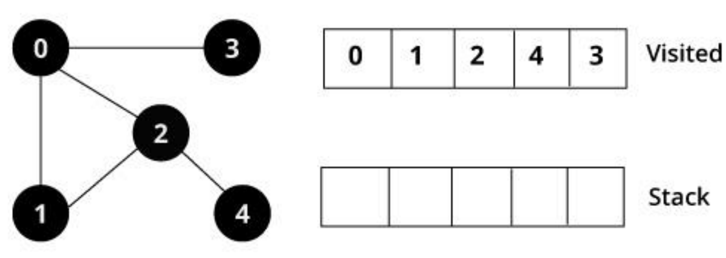
200	100		




```
Struct Node
{
Char data;
Node * left;
Node * right;
};
Void LevelOrder(Node * root)
{
        if (root==NULL) return;
        queue<Node *> Q;   //Queue with ptr to Node
        Q.push(root);
        while(! Q.empty())
        {
                Node * current = Q.front();
                print current->data;
                if(current ->left !=NULL) Q.push(current ->left)
                if(current->right! = NULL) Q.push(current ->right);
                Q.pop();
        }
}
```

Tree Traversal - Depth First Search





Pseudocode

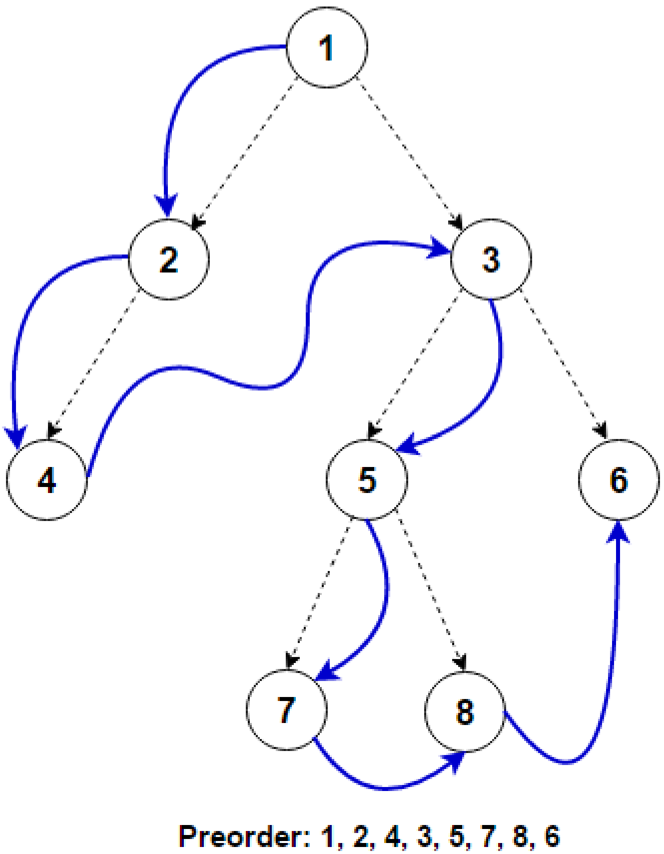
```
init()
{
        For each u ∈ G
        u.visited = false
        For each u ∈ G
        DFS(G, u)
}
DFS(G, u)
{
        u.visited = true
        for each v ∈ G.Adj[u]
         if v.visited == false
         DFS(G,v)
}
```

Time complexity:

T(n) = O(n)

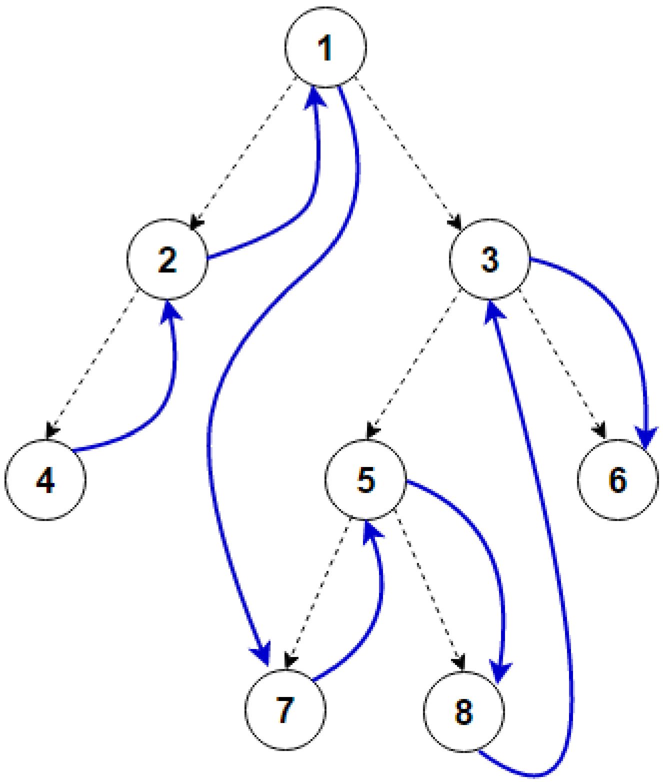
Pre-order Tree Traversal - Depth First Search

- Preorder : N, L, R




```
// Recursive function to perform pre-order traversal of the tree
void preorder (Node *root)
{
        // if the current node is empty
        if (root == nullptr)
                return;

        // Display the data part of the root (or current node)
        print  (root->data );

        // Traverse the left subtree
        preorder(root->left);

        // Traverse the right subtree
        preorder(root->right);
}
```

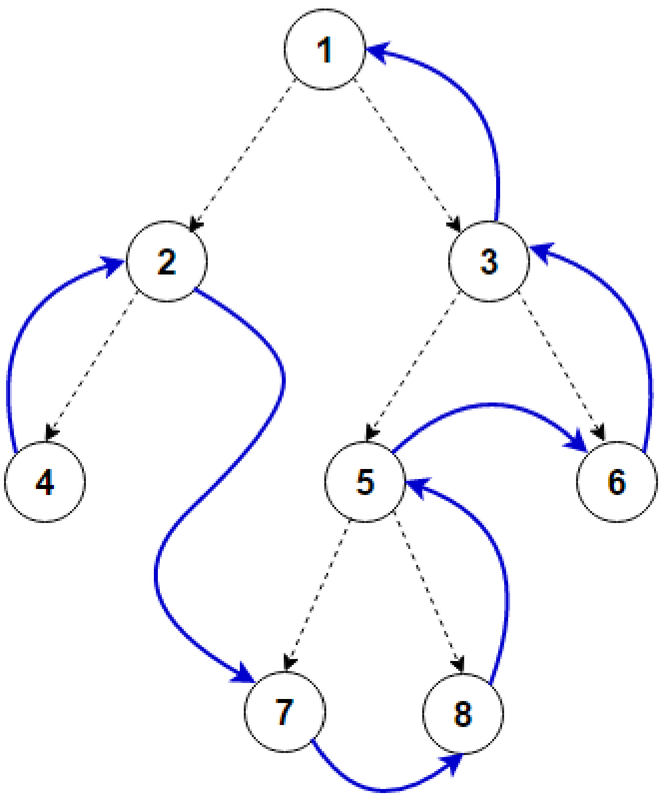
In-order Tree Traversal - Depth First Search



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

```
// Recursive function to perform in-order
traversal of the tree
void inorder(Node *root)
{
        // return if the current node is empty
        if (root == nullptr)
                return;

        // Traverse the left subtree
        inorder(root->left);

        // Display the data part of the root
(or current node)
        print (root->data);

        // Traverse the right subtree
        inorder(root->right);
}
```

Post-order Tree Traversal - Depth First Search

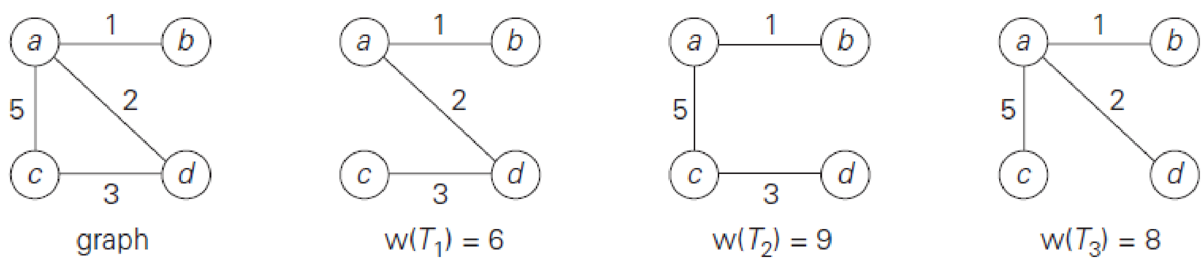


Postorder: 4, 2, 7, 8, 5, 6, 3, 1

```
// Recursive function to perform post-order
    traversal of the tree
void postorder(Node *root)
{
    // if the current node is empty
    if (root == nullptr)
        return;

    // Traverse the left subtree
    postorder(root->left);

    // Traverse the right subtree
    postorder(root->right);

    // Display the data part of the root (or
    current node)
    print (root->data);
}
```

MINIMUM SPANNING TREE

- **Spanning Tree:** A spanning tree of a connected graph is its connected a cyclic sub-graph (i,e; a tree) that contains all the vertices of the graphs.
- **Minimum Spanning Tree:** A Minimum Spanning Tree of a weighted connected graph is its spanning tree of the smallest weight.
- **Weight:** The weight of the 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.



MST - KRUSKAL'S ALGORITHM

Kruskal's algorithm used for solving minimum spanning tree (MST) problem. This algorithm is discovered by a student Joseph Kruskal. In this algorithm, we always select the minimum cost edge but it is not necessary when selected edge is with optimum one in adjacent.

Procedure:

- Initially there are |V| single node tree. Each vertex is initially in its own set.
- Selected the edges (u, v) in the order of smallest weight and accepted if it does not cause the cycle.
- Adding an edge merges 2 trees into one.
- Repeat step 2 until the tree contains all the n vertices.

MST - KRUSKAL'S ALGORITHM Psuedocode and Time Complexity

- T(n) = O(1) + O(V) + O(E log E) + O(V log V)
- T(n) = O(E log E) + O(V log V)
- **T(n) = E log E**

MST-KRUSKAL(G, w)

O(1)

$A = \emptyset$

O(V)

for each vertex $v \in G.V$
    MAKE-SET(v)

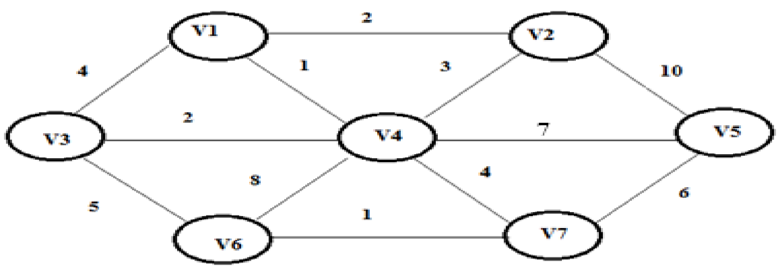
O(E logE)

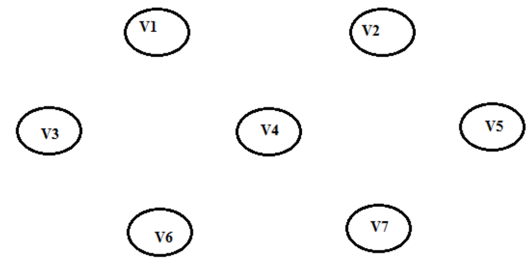
sort the edges of $G.E$ into nondecreasing order by weight w

O(V logV)

for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
    if FIND-SET(u) $\neq$ FIND-SET(v)
        $A = A \cup \{(u, v)\}$
        UNION(u, v)
return A

MST - KRUSKAL'S ALGORITHM Example



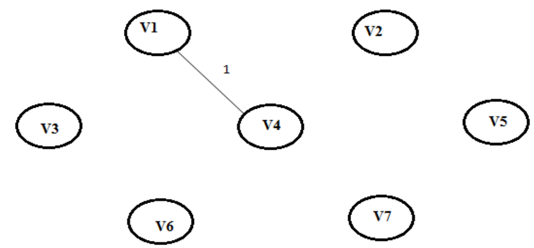


V1V4	V6V7	V1V2	V3V4	V2V4	V1V3	V4V7	V3V6	V5V7	V4V5	V4V6	V2V5
1	1	2	2	3	4	4	5	6	7	8	10

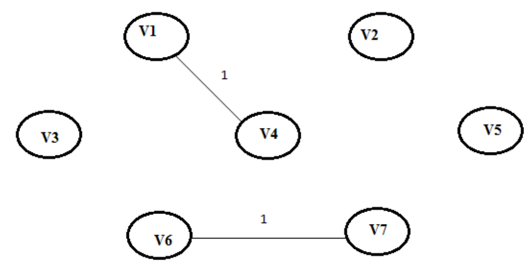
MST - KRUSKAL’S ALGORITHM
Example (Contd..)

V1V4	V6V7	V1V2	V3V4	V2V4	V1V3	V4V7	V3V6	V5V7	V4V5	V4V6	V2V5
1	1	2	2	3	4	4	5	6	7	8	10

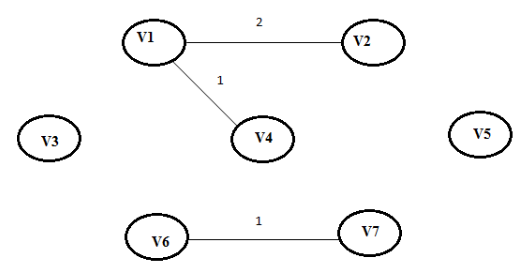
Select the first smallest edge V1V4, both the nodes are different sets it does not form cycle.



Select the next smallest edge V6-V7. Those 2 vertexes are different set it does not form a cycle including in the MST.



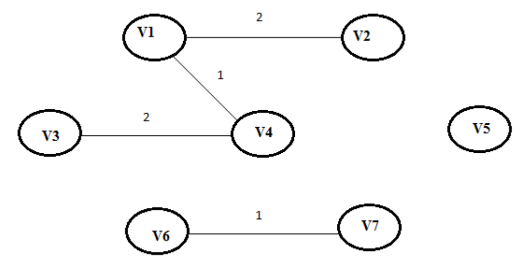
Select the next smallest edge V1-V2, both are different sets so it is included in the tree.



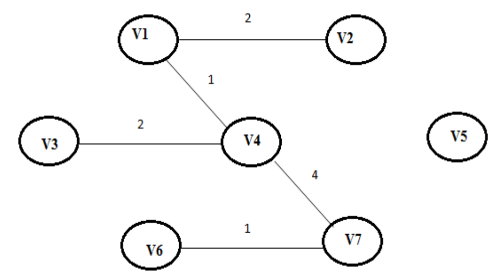
MST - KRUSKAL'S ALGORITHM
Example (Contd..)

V1V4	V6V7	V1V2	V3V4	V2V4	V1V3	V4V7	V3V6	V5V7	V4V5	V4V6	V2V5
1	1	2	2	3	4	4	5	6	7	8	10

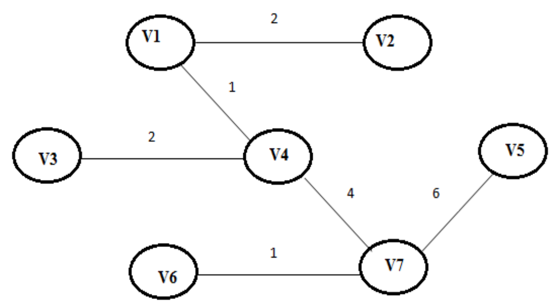
Select the next smallest edge V3—V4, both are different sets so it is included in the tree.



Select the next smallest edge V2-V4, both V2 & V4 are same set. It form cycle so V2-V4 edge is rejected.
Select the next smallest edge V1-V3, it form cycle so V1-V3 edge is rejected.
Select the next smallest edge V4-V7. It is included in the tree.



Select the next smallest edge V3-V6, it form cycle so V3-V6 edge is rejected.
Select the next smallest edge V5-V7, both V5 & V7 are different set. So it is included in the spanning tree.



All the nodes are included. The cost of minimum spanning tree is = 2 + 1 + 2 + 4 +1 + 6 =**16**

MST – PRIMS ALGORITHM

Procedure:

- Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub trees.
- The initial sub tree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices.
- On each iteration, we can expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree.
- The algorithm stops after all the graph's vertices have been included in the tree being constructed.
- Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n-1, where n is the number of vertices in the graph.
- The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

Time Complexity

This algorithm takes its most of time in selecting the edges with minimum length.

Time complexity of Prim's algorithm in case of binary heap is

$$\theta(|E| \log_2 |V|)$$

Where

E – Total number of edges.

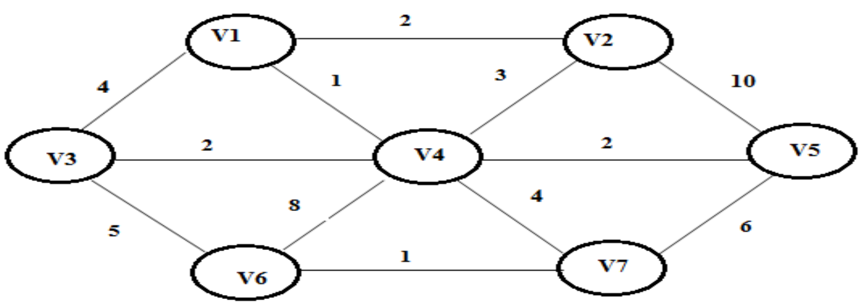
V – Total number of vertices.

Applications of Spanning trees:

Spanning trees are very important in designing efficient routing algorithms.

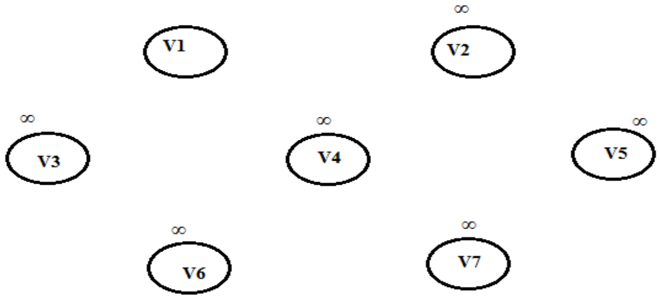
Spanning trees have wide applications in many areas such as network design.

MST – PRIMS ALGORITHM- Example



V	Know	d_v	P_v
V1	0	0	0
V2	0	∞	0
V3	0	∞	0
V4	0	∞	0
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

Let us select V1 as initial node in the spanning tree and construct initial configuration of the table.



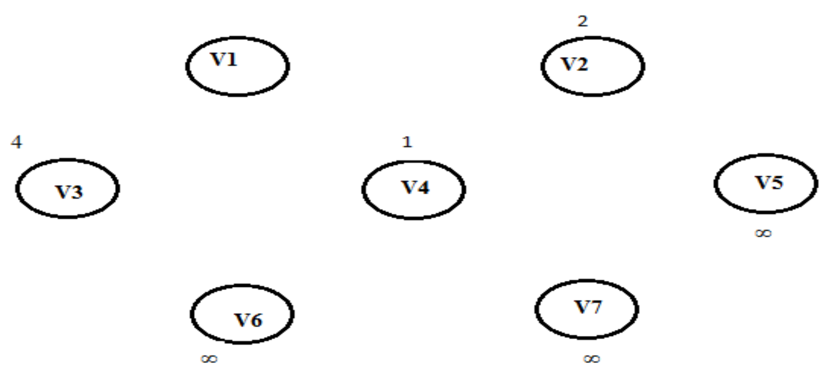
Now V1 is declared as known vertex. Then its adjacent vertices V2V3V4 are updated.

T[V2].dist = Min (T[V2].dist (V1V2)) = Min (∞, 2) = 2

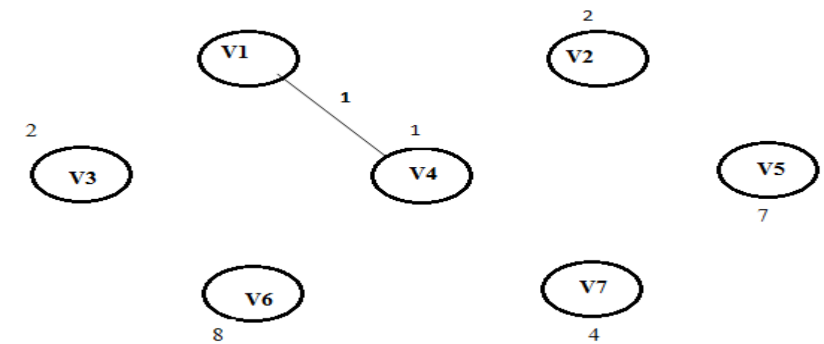
T[V3].dist = Min (T[V3].dist (V1V3)) = Min (∞, 4) = 4

T[V4].dist = Min (T[V4].dist (V1V4)) = Min (∞, 1) = 1

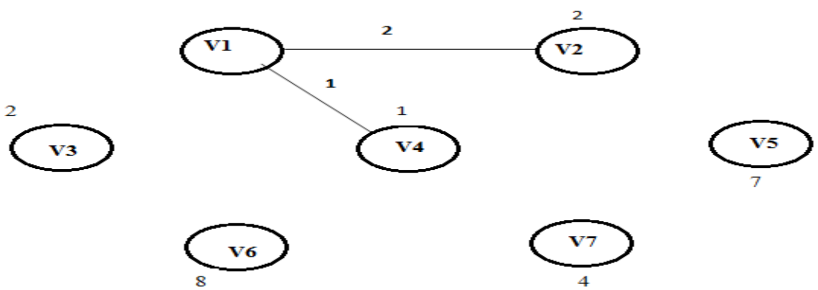
V	Know	d_v	P_v
V1	1	0	0
V2	0	2	V1
V3	0	4	V1
V4	0	1	V1
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0



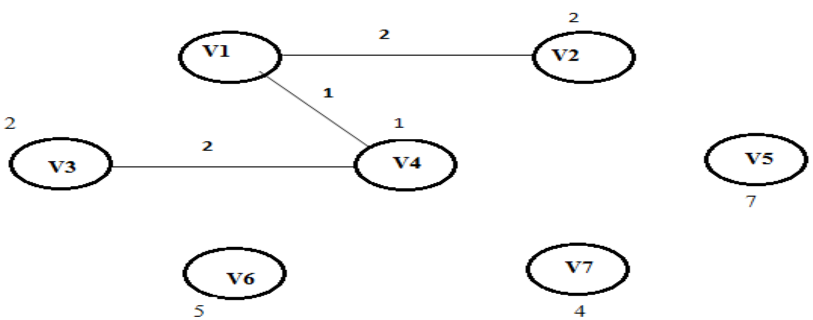
MST – PRIMS ALGORITHM- Example (Contd)..



V	Know	d_v	P_v
V1	1	0	0
V2	0	2	V1
V3	0	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	8	V4
V7	0	4	V4

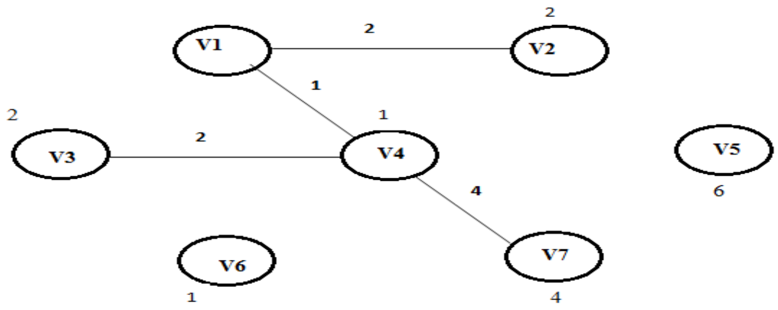


V	Know	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	0	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	8	V4
V7	0	4	V4

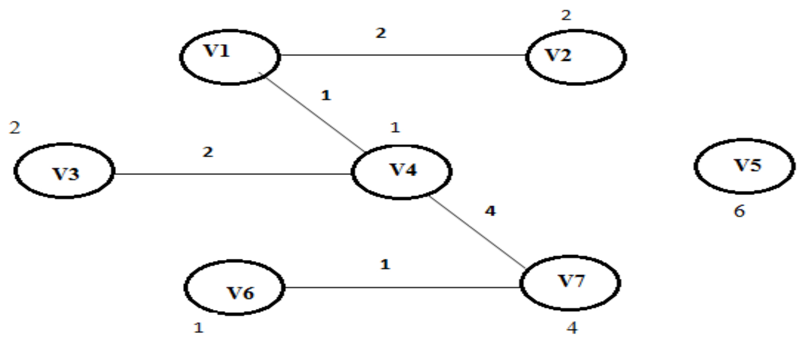


V	Know	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	0	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	5	V3
V7	0	4	V4

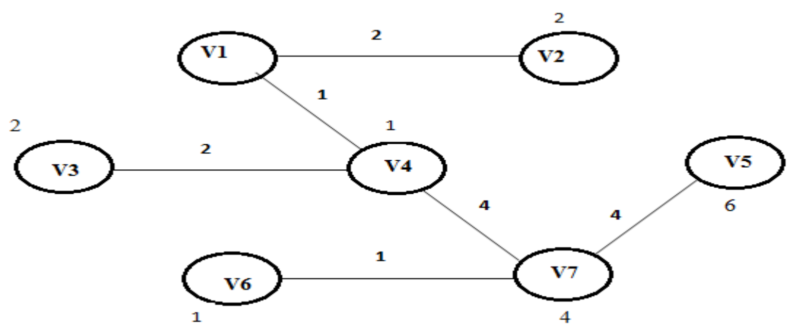
MST – PRIMS ALGORITHM- Example (Contd)..



V	Know	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	6	V7
V6	0	1	V7
V7	1	4	V4



V	Know	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	6	V7
V6	0	1	V7
V7	1	4	V4



V	Know	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	6	V7
V6	0	1	V7
V7	1	4	V4

- **The minimum cost of spanning tree is 16.**

MST – PRIMS ALGORITHM

ALGORITHM *Prim(G)*
//Problem Description: *Prim's algorithm for constructing a minimum spanning tree*
//Input: *A weighted connected graph G = <V, E>*
//Output: *E_T, the set of edges composing a minimum spanning tree of G*
VT←{v_0} //the set of tree vertices can be initialized with any vertex
ET←∅
for *i ←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*

INTRODUCTION TO DYNAMIC PROGRAMMING

- **What is dynamic programming?**
- Dynamic programming is an algorithm design technique for optimizing multistage decision processes. This technique is invented by a prominent U.S. mathematician, Richard Bellman, in 1950s. The word “programming” in the name of this technique stands for “planning” and does not refer to computer programming.
- Dynamic programming is a technique for solving problems with overlapping sub-problems. It suggests solving each of the smaller sub-problems only once and recording the results in a table from which a solution to the original problem.
- **General Method:**
- Dynamic programming is typically applied to optimize.
- For a given problem, we may get any number of solutions. From all those solution we seek for optimum solution (Minimum value or Maximum value solution). And an optimal solution becomes the solution to the given problem.
- **Principles of optimality:**
- The dynamic Programming makes use of principle of optimality when finding solution to given problem. The principle of optimality states that “In an optimal sequence of choices or decisions, each subsequence used also be optimal.”
- When it is not possible to apply principle of optimality, it is almost impossible to obtain the solution using dynamic programming approach.
- For example: while constructing optimal binary search tree we always select the value of K which is obtained for minimum last. Thus it follows principle of optimality.

PROBLEMS SOLVED USING DYNAMIC PROGRAMMING

- **Dynamic programming can solve the following problems:**

There are various problems that can be solved using dynamic programming. They are

  - For computing nth Fibonacci number
  - Computing binomial coefficient
  - Warshall's algorithm
  - Floyd's algorithm
  - Optimal binary search trees

0/1 KNAPSACK PROBLEM using DP Psuedocode

Objective

Max ∑piwi

ST constraint

∑wixi <= M

1<=i<=n;

Xi = 0 or 1.

Steps:

1. Compute S^i = (pi, wi)
2. Assume S^0 = (0,0)
3. S^{i+1} = {Merge (S^i, S_1^i)
4. Purging Rule: Take two pairs in S^i

   (pj, wj), (pk, wk); implies (pj < pk), (wj < wk), if not, remove (pj,wj) from the set.
5. Find Xi:
   1. Xn = 0 if (p,w) ε S^{n-1}
   2. Else Xn=1 and (P,W) = (P-pn, W-wn);
   3. n = n-1;

COMPLEXITY OF KNAPSACK ALGORITHM

- **Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.**

V[i,w]	w=0	1	2	3	···	···	W
i= 0	0	0	0	0	...	...	0
1							
2							
⋮							
n							

bottom

up

0/1 KNAPSACK PROBLEM using DP Problem

- Refer CW

Portions for CT2
Until this topic only

PROBLEMS SOLVED USING DYNAMIC PROGRAMMING - computing nth Fibonacci number

Example:

Let us consider **Fibonacci Number** for our discussion. Fibonacci serious is identified by European Mathematician Leonardo Fibonacci in 1202. We consider the Fibonacci numbers, a famous sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ----------(1)

That can be defined by the simple recurrence

F(n) = F(n-1) + F(n-2) for n > 1 ----------(2)

And two initial conditions **F(0) = 0** and **F(1) = 1** **----------(3)**

If we try to use recurrence (1) directly to compute the *n*th Fibonacci number F(n), then we have to recompute the same values of this function many times like the tree given below

Fig: tree of recursive calls for computing 5th Fibonacci number:

ALGORITHM *F(n)*

//Problem Description:*Computes the nth Fibonacci number recursively by using its definition*

//Input: *A nonnegative integer n*

//Output: *The nth Fibonacci number*

if *n ≤ 1*

return *n*

else

return *F(n − 1) + F(n − 2)*

This algorithm is based on **bottom up dynamic programming** approach, for instance, to compute F(5), we have to compute many smaller sub instances such as F(4), F(3), F(2), F(1) and F(0). For each sub program the solutions are collected, combined and then we get solution to original problem F(5). This is a dynamic programming approach which is used to compute nth Fibonacci number.

MATRIX CHAIN MULTIPLICATION USING DYNAMIC PROGRAMMING

COMPLEXITY OF MATRIX CHAIN MULTIPLICATION

LONGEST COMMON SUBSEQUENCE USING DP

LONGEST COMMON SUBSEQUENCE USING DP - EXAMPLE

OPTIMAL BINARY SEARCH TREE USING DP

OPTIMAL BINARY SEARCH TREE USING DP - EXAMPLE