UNIT 3
Greedy & Dynamic Programming

Session Outcome:

Idea to apply Both Dynamic Programming and Greedy algorithm to **solve optimization problems**.

Greedy Algorithm

Greedy Method

- "Greedy Method finds out of many options, but you have to choose the best option."
- Greedy Algorithm solves problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:
- **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
- **Optimal substructure:** Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.

Greedy Method (Cont.)

Steps for achieving a Greedy Algorithm are:

- **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
- **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
- **Unalterable:** Once the decision is made, at any subsequence step that option is not altered.

Greedy Method (Cont.)

Example:

- Machine scheduling
- Fractional Knapsack Problem
- Minimum Spanning Tree
- Huffman Code
- Job Sequencing
- Activity Selection Problem

Dynamic Programming

Dynamic Programming

- Dynamic Programming is the most powerful design technique for solving optimization problems.
- Divide & Conquer algorithm partition the problem into disjoint sub-problems solve the sub-problems recursively and then combine their solution to solve the original problems.
- Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

Dynamic Programming (Cont.)

- Dynamic Programming solves each subproblems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.
- Dynamic Programming is a **Bottom-up approach-** we solve all possible small problems and then combine to obtain solutions for bigger problems.
- Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "**principle of optimality**".

Characteristics of Dynamic Programming

Dynamic Programming works when a problem has the following features:-

- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping sub-problems:** When a recursive algorithm would visit the same sub-problems repeatedly, then a problem has overlapping sub-problems.

Characteristics of Dynamic Programming

- If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping sub-problems, then we can improve on a recursive implementation by computing each sub-problem only once.
- If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.
- If the space of sub-problems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming

- **Substructure:** Decompose the given problem into smaller sub-problems. Express the solution of the original problem in terms of the solution for smaller problems.
- **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because sub-problem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
- **Bottom-up Computation:** Using table, combine the solution of smaller sub-problems to solve larger sub-problems and eventually arrives at a solution to complete problem.

Note: Bottom-up means:-

- Start with smallest sub-problems.
- Combining their solutions obtain the solution to sub-problems of increasing size.
- Until solving at the solution of the original problem.

Components of Dynamic Programming

1. **Stages:** The problem can be divided into several sub-problems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
6. There exist a recursive relationship that identify the optimal decisions for stage j, given that stage j+1, has already been solved.
7. The final stage must be solved by itself.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest sub-problems)
4. Construct the optimal solution for the entire problem form the computed values of smaller sub-problems.

Applications of Dynamic Programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

Divide & Conquer Method vs Dynamic Programming

Divide & Conquer Method	Dynamic Programming
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	•**1.**It involves the sequence of four steps:Characterize the structure of optimal solutions. •Recursively defines the values of optimal solutions. •Compute the value of optimal solutions in a Bottom-up minimum. •Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

Greedy vs Dynamic

Feature	Greedy method	Dynamic programming
Feasibility	In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .
Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.
Memorization	It is more efficient in terms of memory as it never look back or revise previous choices	It requires dp table for memorization and it increases it's memory complexity.
Time complexity	Greedy methods are generally faster. For example, Dijkstra's shortest path algorithm takes O(ELogV + VLogV) time.	Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.
Fashion	The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.	Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.
Example	Fractional knapsack .	0/1 knapsack problem

Course Learning Outcome :

Apply Greedy approach to solve Huffman coding

Comparison of Brute force and Huffman method of Encoding

Topics

- Huffman Problem
- Problem Analysis (Real Time Example)
  - Binary coding techniques
  - Prefix codes
- Algorithm of Huffman Coding Problem
- Time Complexity
  - Analysis and Greedy Algorithm
- Conclusion

Huffman Coding using Greedy Algorithm

Using ASCII Code: using Text Encoding

- Our objective is to develop a code that represents a given text as compactly as possible.
- A standard encoding is ASCII, which represents every character using 7 bits

Example

Represent "An English sentence" using ASCII code

– 1000001 (A) 1101110 (n) 0100000 ( ) 1000101 (E)
1101110 (n) 1100111 (g) 1101100 (l) 1101001 (i)
1110011 (s) 1101000 (h) 0100000 ( ) 1110011 (s)
1100101 (e) 1101110 (n) 1110100 (t) 1100101 (e)
1101110 (n) 1100011 (c) 1100101 (e)

= 133 bits ≈ 17 bytes

Refinement in Text Encoding

- Now a better code is given by the following encoding:

  ‹space› = 000, A = 0010, E = 0011, s = 010, c = 0110, g = 0111, h = 1000, i = 1001, l = 1010, t = 1011, e = 110, n = 111

- Then we encode the phrase as

  0010 (A) 111 (n) 000 ( ) 0011 (E) 111 (n) 0111 (g) 1010 (l) 1001 (i) 010 (s) 1000 (h) 000 ( ) 010 (s) 110 (e) 111 (n) 1011 (t) 110 (e) 111 (n) 0110 (c) 110 (e)

- This requires 65 bits ≈ 9 bytes. Much improvement.
- The technique behind this improvement, i.e., Huffman coding which we will discuss later on.

Major Types of Binary Coding

There are many ways to represent a file of information.

Binary Character Code (or **Code**)

- each character represented by a unique binary string.

- **Fixed-Length Code**
  - If $\Sigma = \{0, 1\}$ then
  - All possible combinations of two bit strings

    $\Sigma \times \Sigma = \{00, 01, 10, 11\}$
  - If there are less than four characters then two bit strings enough
  - If there are less than three characters then two bit strings not economical

Fixed Length Code

- **Fixed-Length Code**
  - All possible combinations of three bit strings
    $\Sigma \times \Sigma \times \Sigma = \{000, 001, 010, 011, 100, 101, 110, 111\}$
  - If there are less than nine characters then three bit strings enough
  - If there are less than five characters then three bit strings not economical and can be considered two bit strings
  - If there are six characters then needs 3 bits to represent, following could be one representation.
    a = 000, b = 001, c = 010, d = 011, e = 100, f = 101

Variable Length Code

- Variable-Length Code
  - better than a fixed-length code
  - It gives short code-words for frequent characters and
  - long code-words for infrequent characters
- Assigning variable code requires some skill
- Before we use variable codes we have to discuss prefix codes to assign variable codes to set of given characters

Prefix Code (Variable Length Code)

- A prefix code is a code typically a variable length code, with the “prefix property”
- Prefix property is defined as no codeword is a prefix of any other code word in the set.

Examples

1. Code words {0,10,11} has prefix property
2. A code consisting of {0, 1, 10, 11} does not have, because “1” is a prefix of both “10” and “11”.

Other names

- Prefix codes are also known as prefix-free codes, prefix condition codes, comma-free codes, and instantaneous codes etc.

Why are prefix codes?

- Encoding simple for any binary character code;
- Decoding also easy in prefix codes. This is because no codeword is a prefix of any other.

Example 1

- If a = 0, b = 101, and c = 100 in prefix code then the string: 0101100 is coded as 0·101·100

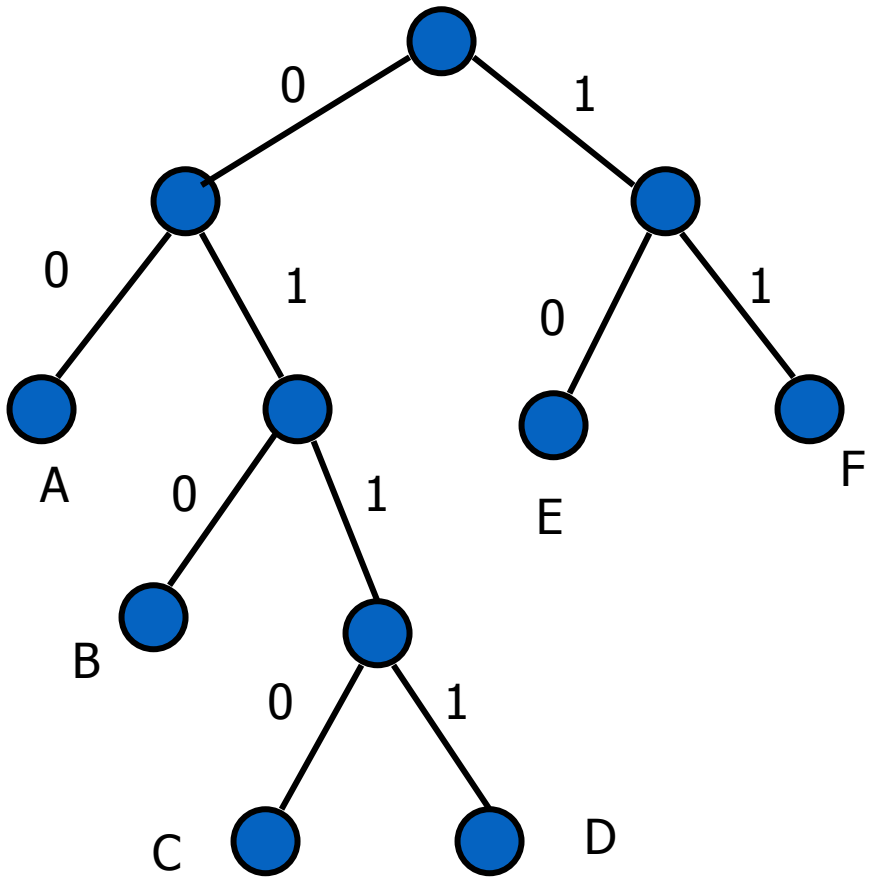
Example 2

- In code words: {0, 1, 10, 11}, receiver reading "1" at the start of a code word would not know whether
  - that was complete code word "1", or
  - prefix of the code word "10" or of "11"

Prefix codes and binary trees

Tree representation of prefix codes

A	00
B	010
C	0110
D	0111
E	10
F	11



Huffman Codes

- In Huffman coding, variable length code is used
- Data considered to be a sequence of characters.
- Huffman codes are a widely used and very effective technique for compressing data
  - Savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.
- Now let us see an example to understand the concepts used in Huffman coding

Example: Huffman Codes

	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

A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated above.

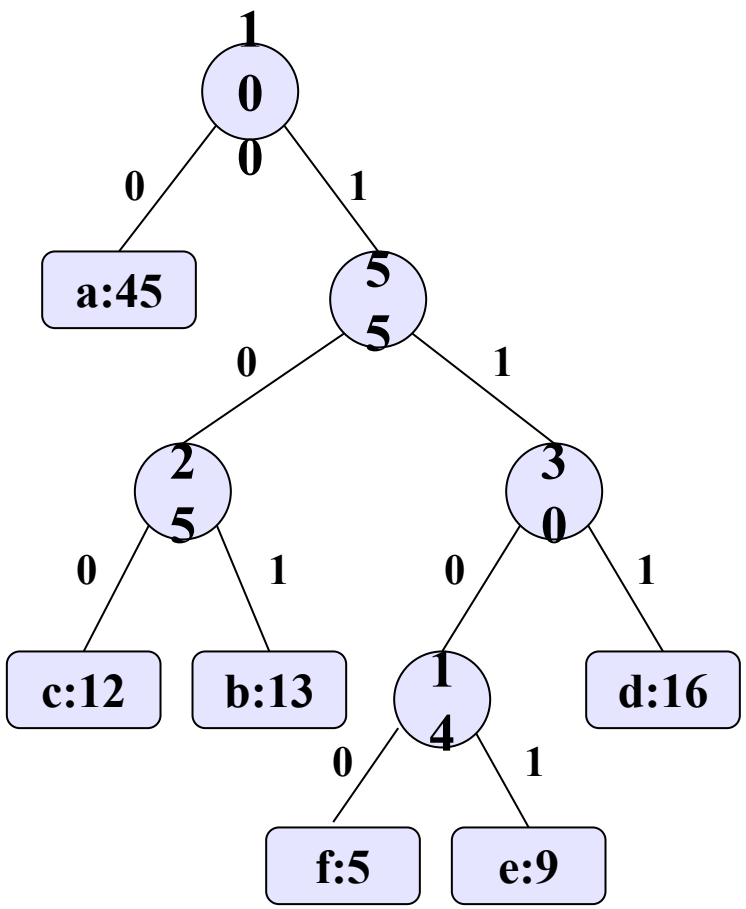
- If each character is assigned a 3-bit **fixed-length codeword**, the file can be encoded in 300,000 bits.
- Using the **variable-length code**

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

which shows a savings of approximately 25%.

Binary Tree: Variable Length Codeword

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



The tree corresponding to the variable-length code is shown for the data in table.

Cost of Tree Corresponding to Prefix Code

- Given a tree T corresponding to a prefix code. For each character c in the alphabet C,
  - let $f(c)$ denote the frequency of c in the file and
  - let $d_T(c)$ denote the depth of c's leaf in the tree.
  - $d_T(c)$ is also the length of the codeword for character c.
  - The number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

  - which we define as the ***cost*** of the tree T.

Algorithm: Constructing a Huffman Codes

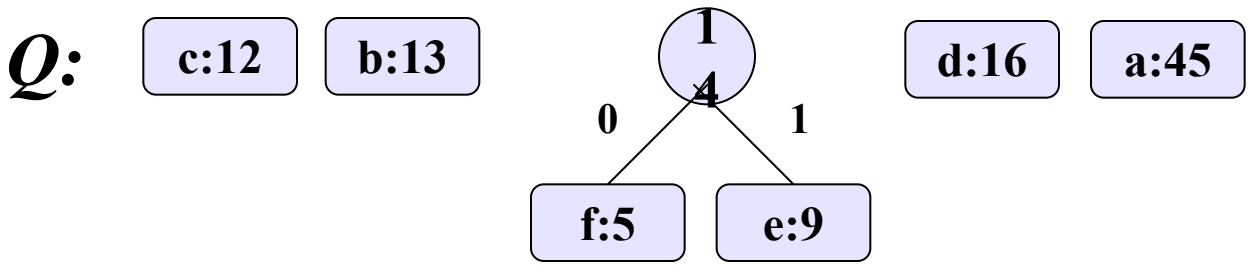
```
Huffman (C)
n ← |C|
Q ← C
for i ← 1 to n - 1
    do allocate a new node z
        left[z] ← x ← Extract-Min (Q)
        right[z] ← y ← Extract-Min (Q)
        f [z] ← f [x] + f [y]
        Insert (Q, z)
return Extract-Min(Q)  □ Return root of the tree.
```

Example: Constructing a Huffman Codes

Q: **f:5** **e:9** **c:12** **b:13** **d:16** **a:45**

The initial set of $n = 6$ nodes, one for each letter.
Number of iterations of loop are 1 to n-1 (6-1 = 5)

Constructing a Huffman Codes



for i ← 1

Allocate a new node z

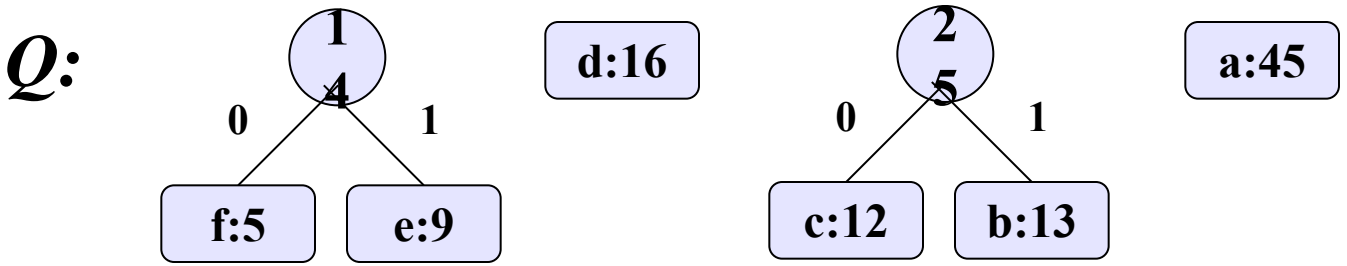
left[*z*] ← *x* ← Extract-Min (*Q*) = f:5

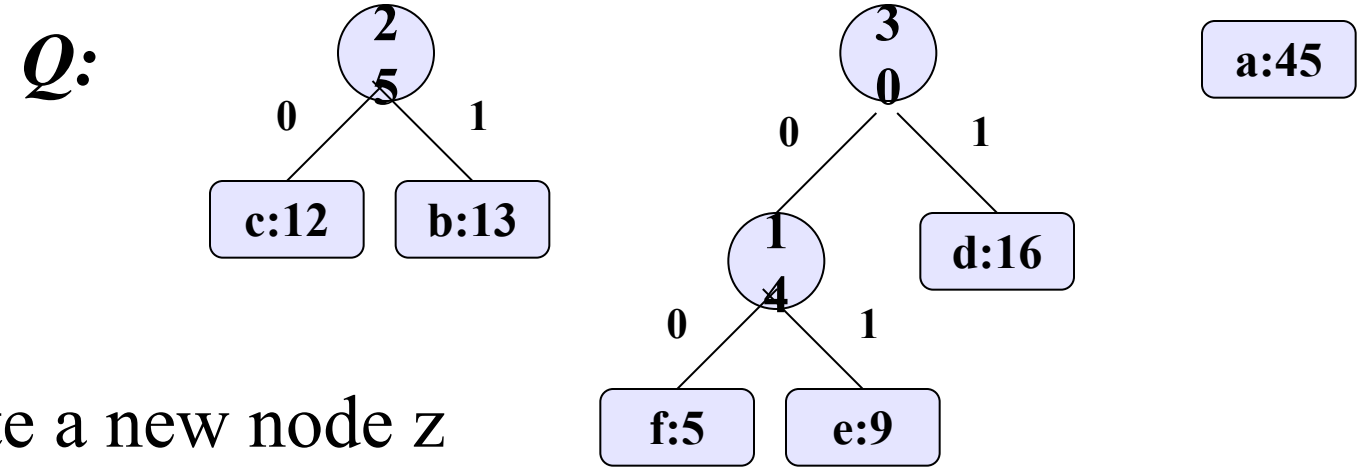
right[*z*] ← *y* ← Extract-Min (*Q*) = e:9

f [*z*] ← *f* [*x*] + *f* [*y*] (5 + 9 = 14)

Insert (*Q*, *z*)

Constructing a Huffman Codes





for i ← 3

Allocate a new node z

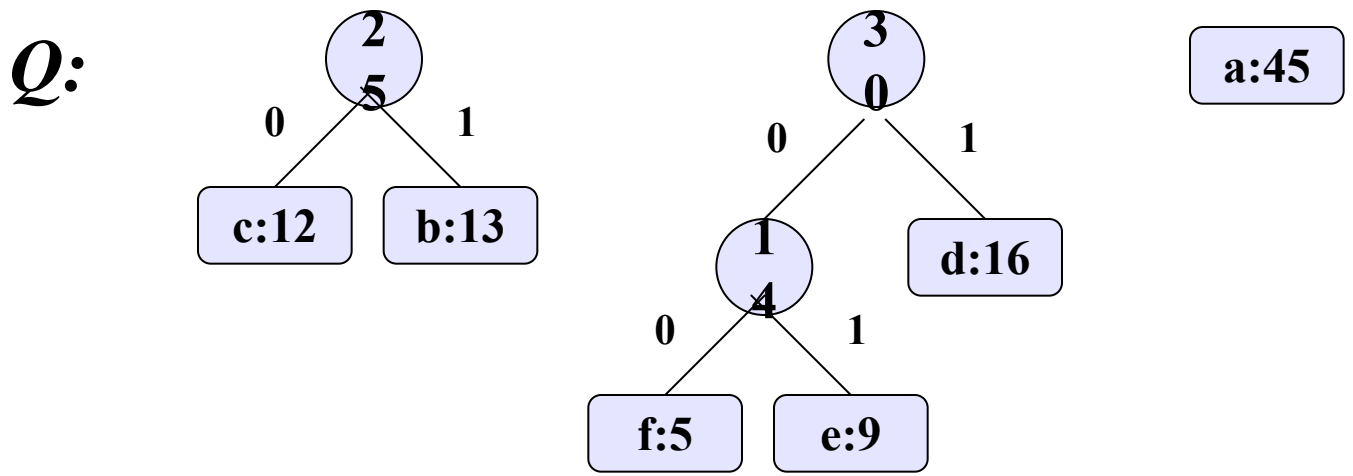
left[*z*] ← *x* ← Extract-Min (*Q*) = z:14

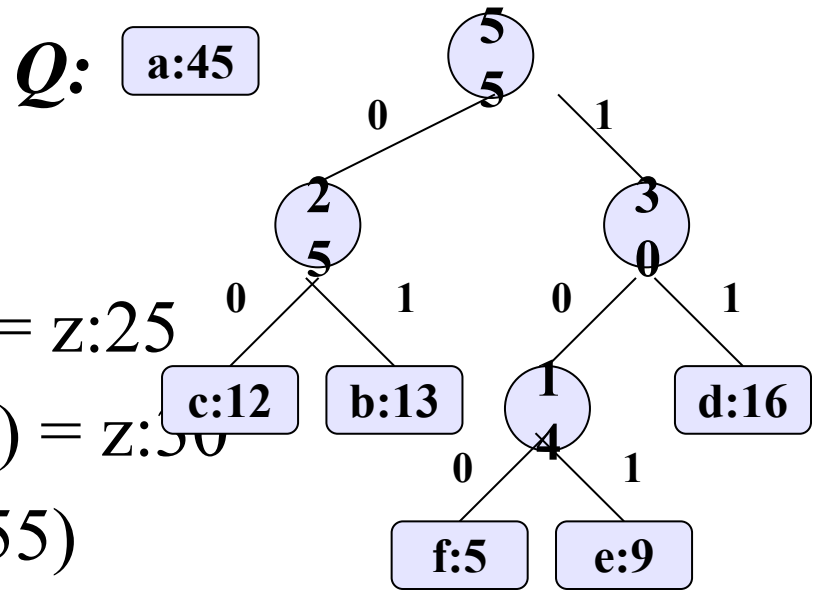
right[*z*] ← *y* ← Extract-Min (*Q*) = d:16

f [*z*] ← *f* [*x*] + *f* [*y*] (14 + 16 = 30)

Insert (*Q*, *z*)

Constructing a Huffman Codes





for i ← 4

Allocate a new node z

left[*z*] ← *x* ← Extract-Min (*Q*) = z:25

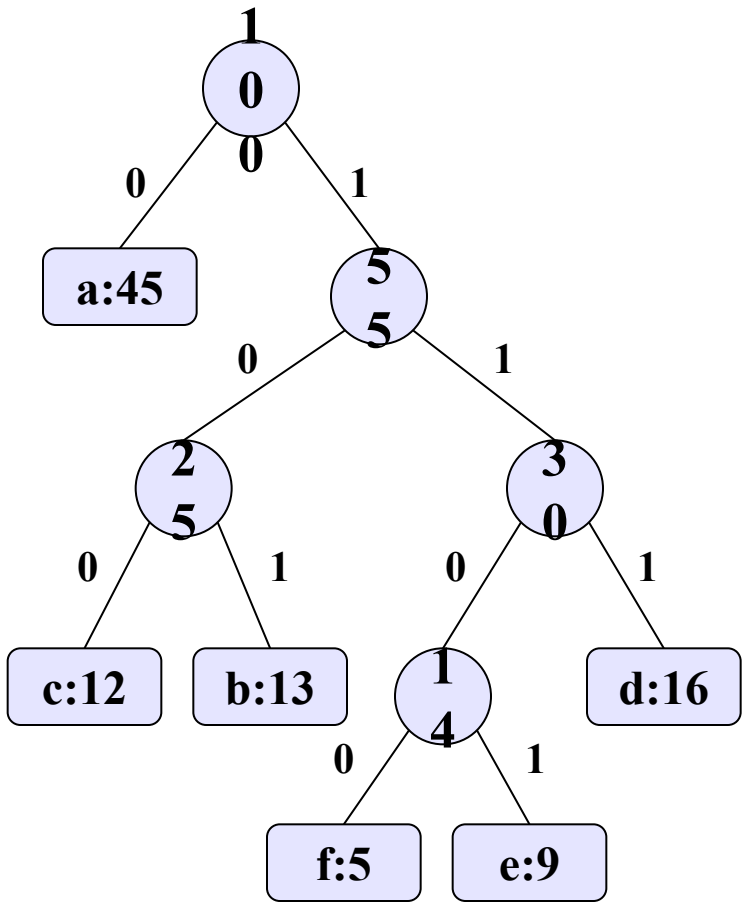
right[*z*] ← *y* ← Extract-Min (*Q*) = z:30

f [*z*] ← *f* [*x*] + *f* [*y*] (25 + 30 = 55)

Insert (*Q*, *z*)

Constructing a Huffman Codes

Q:



for i ← 5

Allocate a new node z

left[*z*] ← *x* ← Extract-Min (*Q*) = a:45

right[*z*] ← *y* ← Extract-Min (*Q*) = z:55

f [*z*] ← *f* [*x*] + *f* [*y*] (45 + 55 = 100)

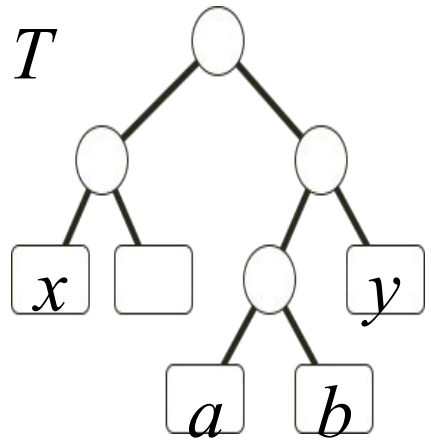
Insert (*Q*, *z*)

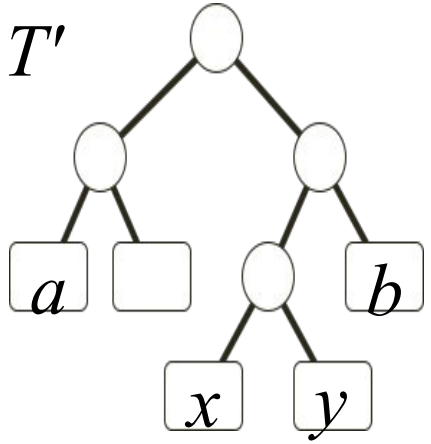
Lemma 1: Greedy Choice

There exists an optimal prefix code such that the two characters with smallest frequency are siblings and have maximal depth in T.

Proof:

- Let x and y be two such characters, and let T be a tree representing an optimal prefix code.
- Let a and b be two sibling leaves of maximal depth in T, and assume with out loss of generality that $f(x) \leq f(y)$ and $f(a) \leq f(b)$.
- This implies that $f(x) \leq f(a)$ and $f(y) \leq f(b)$.
- Let T' be the tree obtained by exchanging a and x and b and y.





Conclusion

- Huffman Algorithm is analyzed
- Design of algorithm is discussed
- Computational time is given
- Applications can be observed in various domains e.g. data compression, defining unique codes, etc. generalized algorithm can used for other optimization problems as well

Two questions

- Why does the algorithm produce the best tree ?
- How do you implement it efficiently ?

Knapsack problem using greedy approach

Session Learning Outcome-SLO

Greedy is used in optimization problems. The **algorithm** makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem

GREEDY INTRODUCTION

- Greedy algorithms are simple and straightforward.
- They are shortsighted in their approach
- A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the sub problems do not have to be known at each stage,
- instead a "greedy" choice can be made of what looks best for the moment.

GREEDY INTRODUCTION

- Many real-world problems are optimization problems in that they attempt to find an optimal solution among many possible candidate solutions.
- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase: You take the best you can get right now, without regard for future consequences. It hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.

APPLICATIONS OF GREEDY APPROACH

1) Job sequenced with deadline.
2) Knapsack problem.
3) Huffman coding .
4) Optimal merge pattern.
5) Min cost spanning tree.
6) Dikstra's algorithm.
7) Bellman ford
8) Breath first traversal(BFT)

FRACTIONAL KNAPSACK PROBLEM

Now applying the greedy method to solve the knapsack problem. We are given with:-

- Objects n
- Weights Wi
- Profits Pi
- Knapsack of capacity 'm'

So here we have n objects and a knapsack of capacity 'm'. Object 'i' has weight 'Wi'. If the fraction Xi , 0 <= Xi <=1, of object 'i' is placed into the knapsack, then a profit of 'PiXi' is earned.

THE OBJECTIVE is to obtain a filling of the knapsack that maximizes the total profit earned. As the knapsack capacity is 'm' , we require the total weight of all objects to be at most 'm'. Formally the problem is stated as:-

Maximize	$\sum_{1<=I<=n} PiXi$	**(1)**
Subject to	$\sum_{i<=i<=n} WiXi <= m$	**(2)**
And	$0<=Xi<=1 , 1<=i<=n$	**(3)**

And a feasible solution is any set satisfying equation (2) and equation (1). An optimal solution is a feasible solution for which equation (1) is maximized.

FRACTIONAL KNAPSACK ALGORITHM

```
Fractional knapsack ( a , p , w , c )

//a is array of items,p is array  for profits, w is array  for weights and
c is array for capacity of knapsack. //

{
   n := length [a] ;                    // finding total number of items given.
   for i := 1 to n
   do
        ratio [ i ]  := p [ i ] / w [ i ] ;
        x [ i ] := 0 ;
  done
   sort ( a, ratio ) ;                 // to arrange item with highest value first
   weight := 0
   i := 1
```

```
while ( i <= n  and  weight < c )
   {
      if ( weight +w [ i ] <= c )
       then
           x [ i ] := i;
           weight := weight + w [ i ] ;
       else
           r := ( c- weight ) / w [ i ] ;
           x [ i ] := r ;
           weight := c ;
     i := i + 1 ;
   }

   output ( x )
}
```

The fractional knapsack problem can be more clearly explained with the help of the following example:

EXAMPLE :-

The total capacity of knapsack is 20.
The total number of items present is 3.

Profit values Pi

P1	P2	P3
25	24	15

Weights values Wi

W1	W2	W3
18	15	10

Now as we go according to the algorithm, we have :

n = 3 p[i] = { 25 , 24 , 15 }
c = 20 w[i] = {18 , 15 , 10 }

Now the ratio array will have values:

ratio [1] = 25 / 18 = 1.4
ratio [2] = 24 / 15 = 1.6
ratio [3] = 15 / 10 = 1.5

simultaneously ,

x [1] = 0
x [2] = 0
x [3] = 0

then we sort the ratio array and it is as follows:-

1.6	1.5	1.4
1(item (2)	**2 (item 3)**	**3 (item 1)**

Weight = 0
i = 1

P

P2	P3	P1
24	15	25

W

W2	W3	W1
15	10	18

Now we check i for 1,2 3 and till weight (0) < 20

For i =1 and 0 < 20
check if (0 + 15 <= 20)
15<= 20
=> x [1] = 1
weight = 15

For i = 2 and 15 < 20
else
=>r = (20 -15) /10 = 5 /10
x [2]=5 /10
weight = 20

For i = 3 and 20 !<20 END

So in this case order of consideration is 2 , 3 , 1.
We choose item 2 first whose weight is 15 and left space in knapsack is 5.
So we will add as much as possible from the next highest order that is item 3.
Which gives us 5/10.

Thus the profit is:

24 + (5/10) * 15 = 31.5

This can also be explained by taking one more example as follows:-

EXAMPLE :

(w1,p1)	(w2,p2)	(w3,p3)	(w4,p4)	(w5,p5)
(120,5)	(150,5)	(200,4)	(150,8)	(140,3)

Total capacity c = 600

The profit / weight ratios are :

p1/w1 = 5/120 = .0417; p2/w2 = 5/150 = .0333; p3/w3 = 4/200 = .0200;

p4/w4 = 8/150 = .0533; p5/w5 = 3/140 = .0214;

Thus the order of consideration is 4, 1, 2, 5, 3

We take all of items 4, 1, 2 and 5 for a total weight of knapsack is

$$150 + 120 + 150 + 140 = 560$$

We now only have room for 40 units of weight, so we take 40/200 = 1/5 of object 3

The profit is thus $5 + 5 + 0.2 \cdot 4 + 8 + 3 = 21.8$

Thus the total profit gained is = 21.8

COMPLEXITY OF FRACTIONAL KNAPSACK PROBLEM

The complexity of fractional knapsack problem is **O (n log (n))**

RUNNING TIME

Given a set of n tasks specified by their start and finish times, Algorithm Task Schedule produces a schedule of the tasks with the minimum number of machines in O(nlogn) time.

- Use heap-based priority queue to store tasks with the start time as the priorities
- Finding the earliest task takes O(logn) time

APPLICATIONS OF FRACTIONAL KNAPSACK PROBLEM

- It is used in automated data mining.
- Used in linear programming problems
- Real life applications of money-time relationships.
- Criteria for choosing feasible project

Home assignment /Questions

Write code for the 0-1 knapsack problem using greedy algorithm.
Output should display:

Number of items = 5
Weights & values for each item:

Weights	Values
2	10
3	12
4	20
5	22
7	40

Capacity of knapsack: 15
Maximum value that can be chosen with capacity of knapsack = 15

Tree Traversals

Session Learning Outcome-SLO

These usage patterns can be divided into the three ways that we access the nodes of the tree.

Tree Traversal

- Traversal is the process of visiting every node once
- Visiting a node entails doing some processing at that node, but when describing a traversal strategy, we need not concern ourselves with what that processing is

Binary Tree Traversal Techniques

- Three <u>recursive</u> techniques for binary tree traversal
- In each technique, the left subtree is traversed recursively, the right subtree is traversed recursively, and the root is visited
- What distinguishes the techniques from one another is the order of those 3 tasks

Preoder, Inorder, Postorder

- In Preorder, the root is visited before (pre) the subtrees traversals
- In Inorder, the root is visited in-between left and right subtree traversal
- In Preorder, the root is visited after (pre) the subtrees traversals

Preorder Traversal:

1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

Inorder Traversal:

1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

Postorder Traversal:

1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

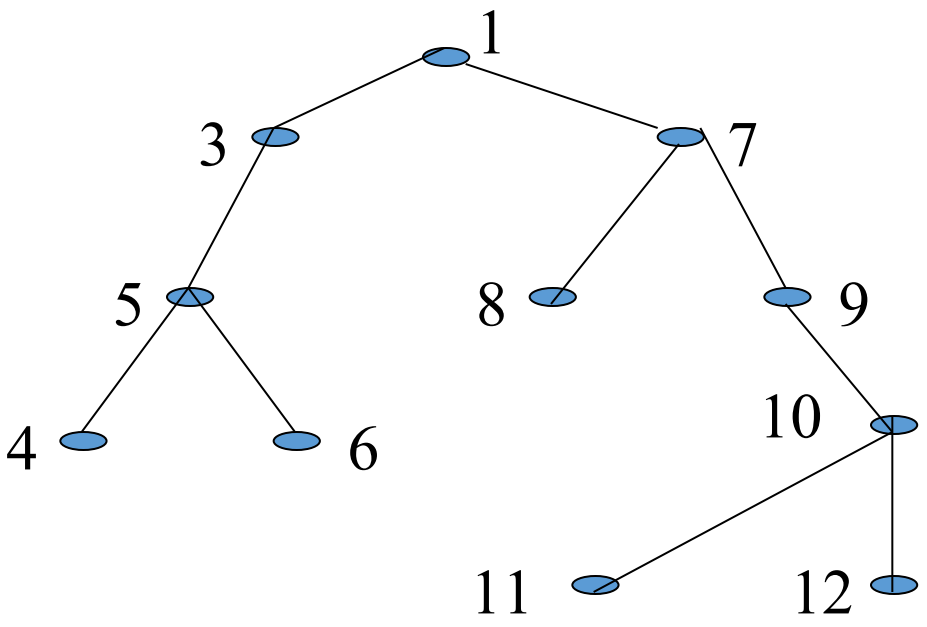
Illustrations for Traversals

- Assume: visiting a node is printing its <u>label</u>
- Preorder:

  1 3 5 4 6 7 8 9 10 11 12
- Inorder:

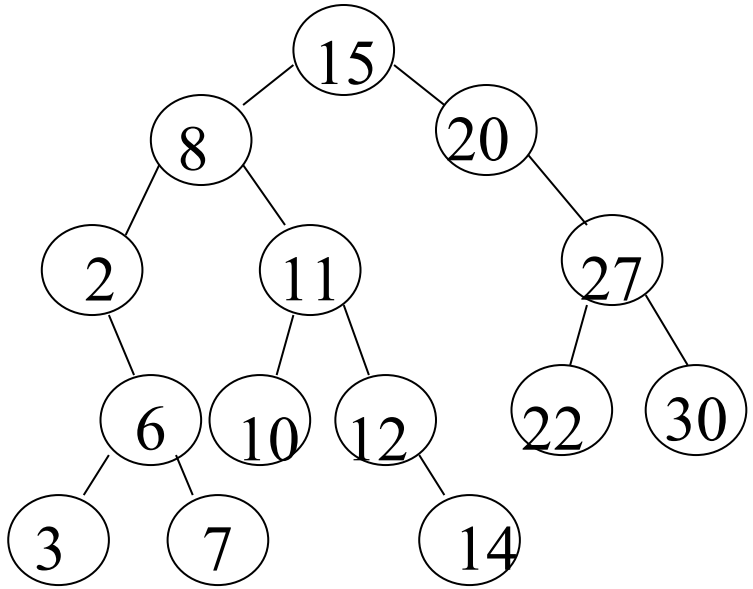
  4 5 6 3 1 8 7 9 11 10 12
- Postorder:

  4 6 5 3 8 11 12 10 9 7 1



Illustrations for Traversals (Contd.)

- Assume: visiting a node is printing its <u>data</u>
- Preorder: 15 8 2 6 3 7 11 10 12 14 20 27 22 30
- Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30
- Postorder: 3 7 6 2 10 14 12 11 8 22 30 27 20 15



Code for the Traversal Techniques

- The code for visit is up to you to provide, depending on the application
- A typical example for visit(…) is to print out the data part of its input node

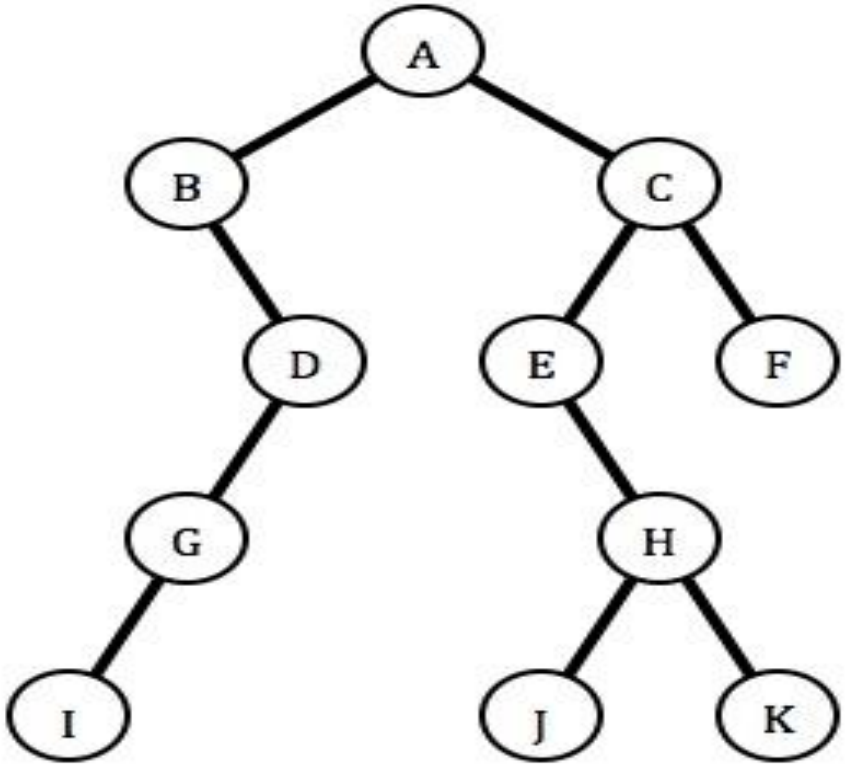
```
void preOrder(Tree *tree){
    if (tree->isEmpty( ))      return;
    visit(tree->getRoot( ));
    preOrder(tree->getLeftSubtree());
    preOrder(tree->getRightSubtree());
}
```

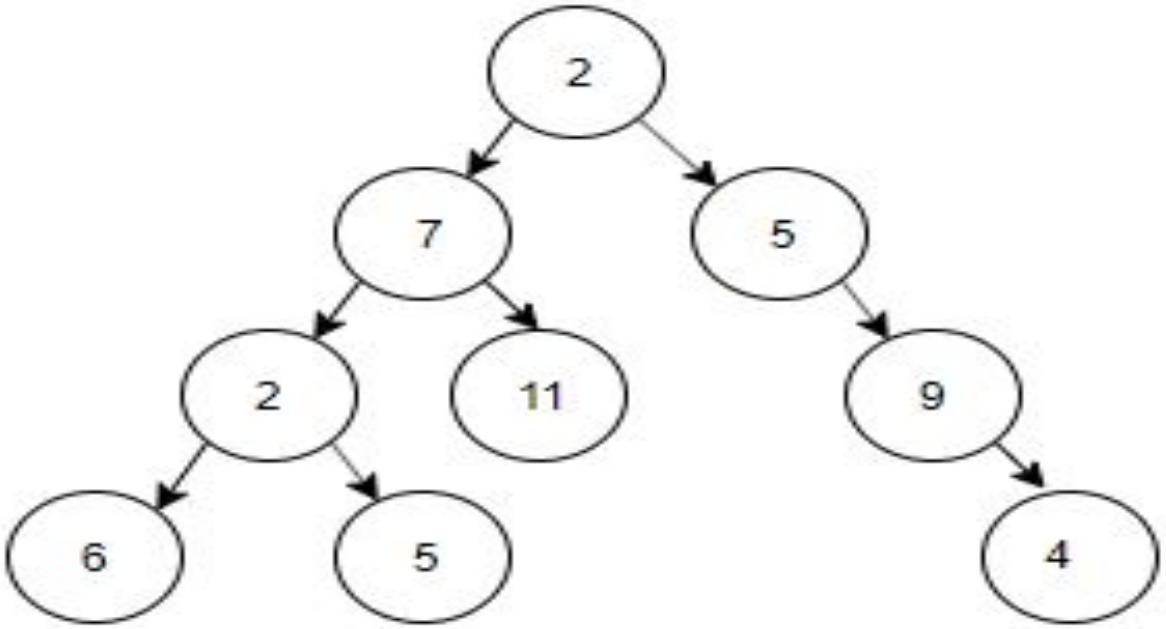
```
void inOrder(Tree *tree){
    if (tree->isEmpty( ))      return;
    inOrder(tree->getLeftSubtree( ));
    visit(tree->getRoot( ));
    inOrder(tree->getRightSubtree( ));
}
```

```
void postOrder(Tree *tree){
    if (tree->isEmpty( ))      return;
    postOrder(tree->getLeftSubtree( ));
    postOrder(tree->getRightSubtree( ));
    visit(tree->getRoot( ));
}
```

Home assignment /Questions

Write the preorder, inorder and postorder traversals of the binary tree shown below.





Minimum Spanning Tree

Agenda

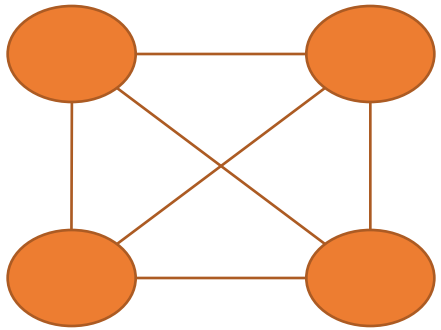
- Spanning Tree
- Minimum Spanning Tree
- Prim’s Algorithm
- Kruskal’s Algorithm
- Summary
- Assignment

Session Outcomes

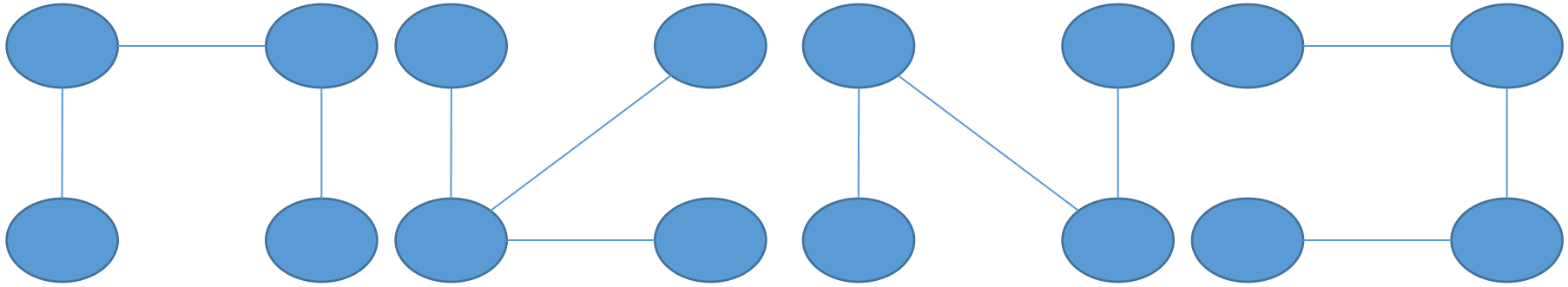
- Apply and analyse different problems using greedy techniques
- Evaluate the real time problems
- Analyse the different trees and find the low cost spanning tree

Spanning Tree

- Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)



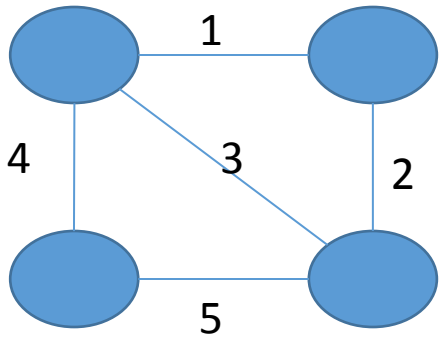
Undirected Graph



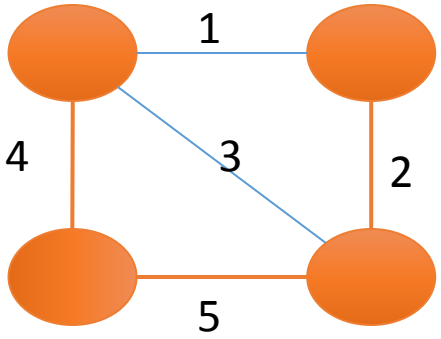
For of the spanning trees of the graph

Minimum Spanning Tree

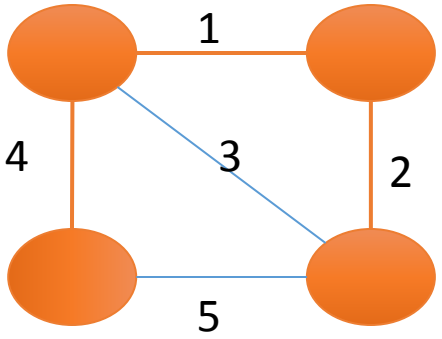
- The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.



Undirected Graph



Spanning Tree Cost=11 (4+5+2)



Minimum Spanning Tree Cost = 7 (4+1+2)

Applications – MST

- Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:
- Cluster Analysis
- Handwriting recognition
- Image segmentation

Prim’s Algorithm

- Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.
- Prim’s Algorithm is preferred when-
  - The graph is dense.
  - There are large number of edges in the graph like $E = O(V^2)$.

Steps for finding MST - Prim's Algorithm

Step 1: Select any vertex

Step 2: Select the shortest edge connected to that vertex

Step 3: Select the shortest edge connected to any vertex already connected

Step 4: Repeat step 3 until all vertices have been connected

Prim's Algorithm

Algorthm Prim(G)

$V_T \leftarrow \{v_0\}$

$E_T \leftarrow \{\}$ // empty set

for $i \leftarrow 1$ **to** $|V|$ **do**

find the minimum weight edge, $e^* = (v^*, u^*)$ such that v^* is in V_T and u is in V- V_T

$V_T \leftarrow V_T$ union $\{u^*\}$

$E_T \leftarrow E_T$ union $\{e^*\}$

return E_T

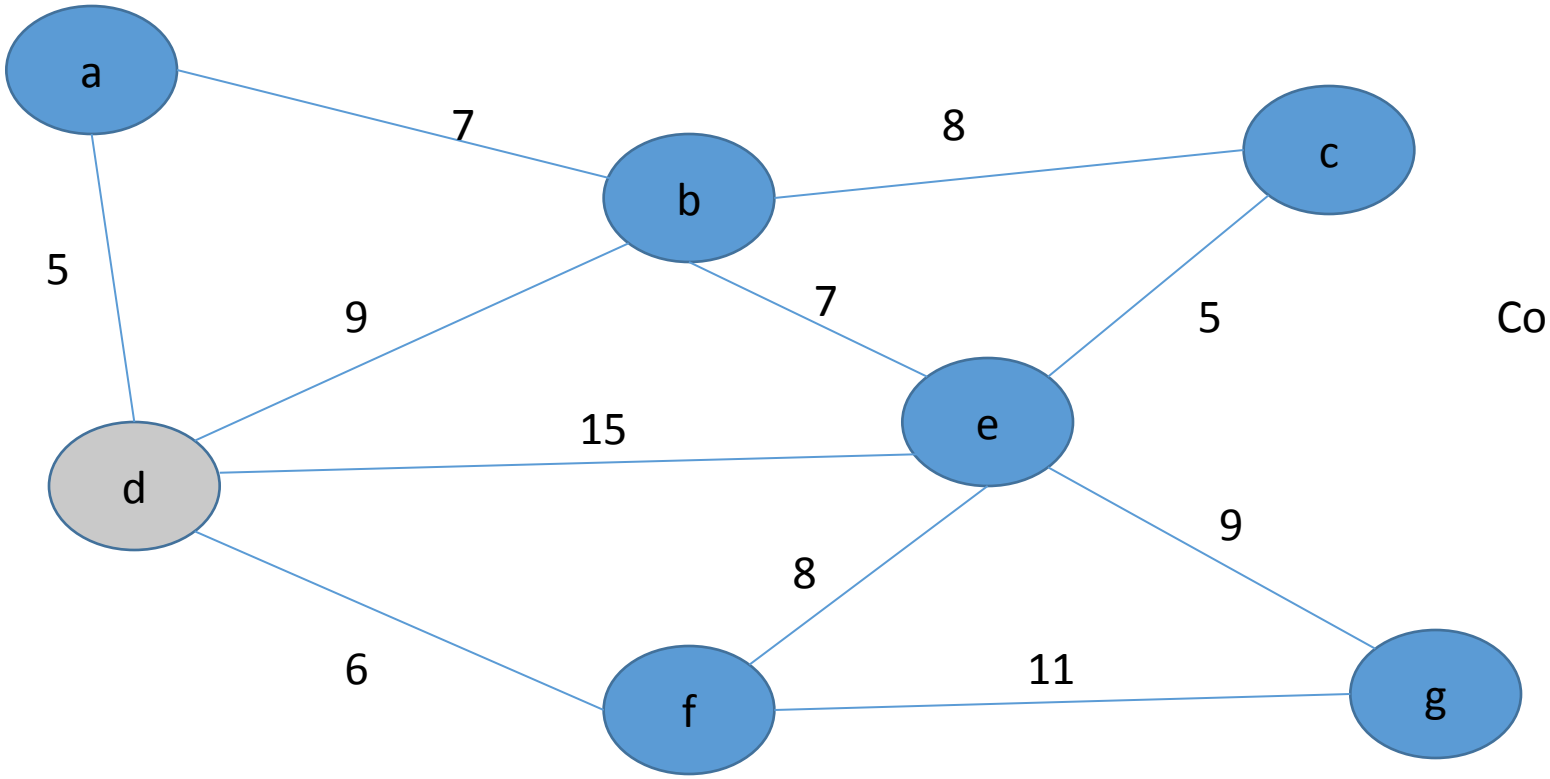
Time Complexity –Prim's Algorithm

- If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in O(V + E) time.
- We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.
- To get the minimum weight edge, we use min heap as a priority queue.
- Min heap operations like extracting minimum element and decreasing key value takes O(logV) time.

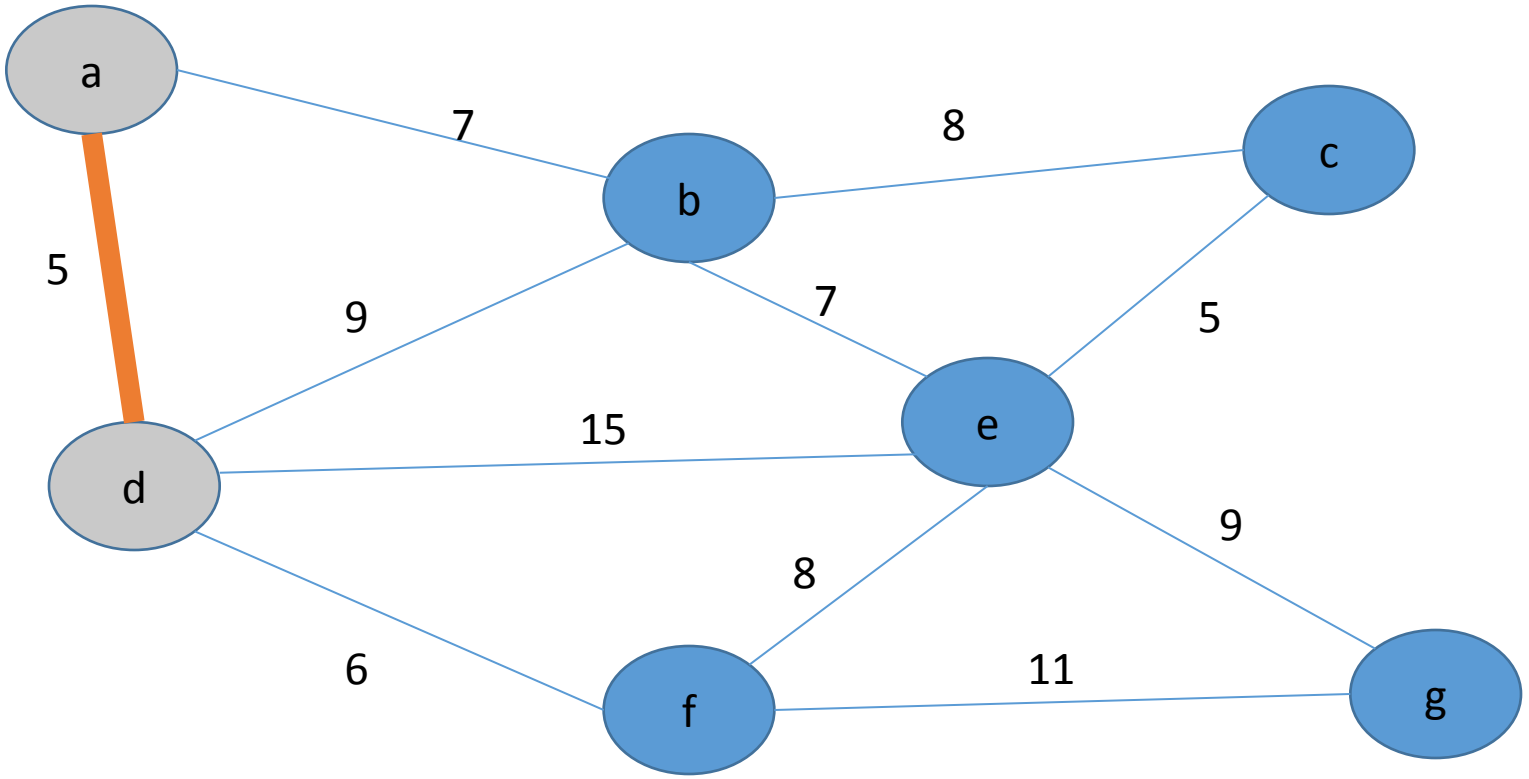
- So, overall time complexity
- = O(E + V) x O(logV)
- = O((E + V)logV)
- = O(ElogV)

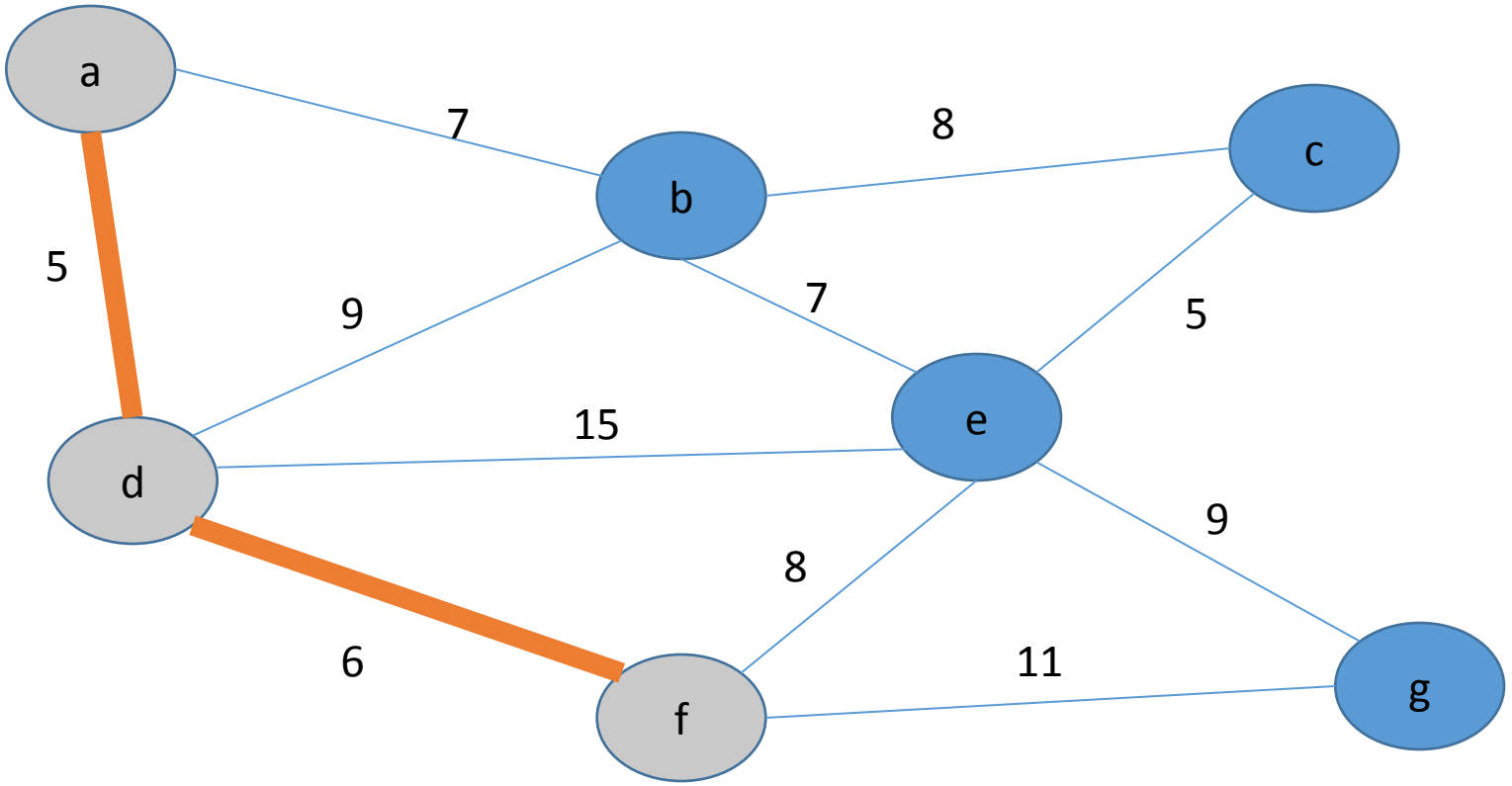
This time complexity can be improved and reduced to O(E + VlogV) using Fibonacci heap.

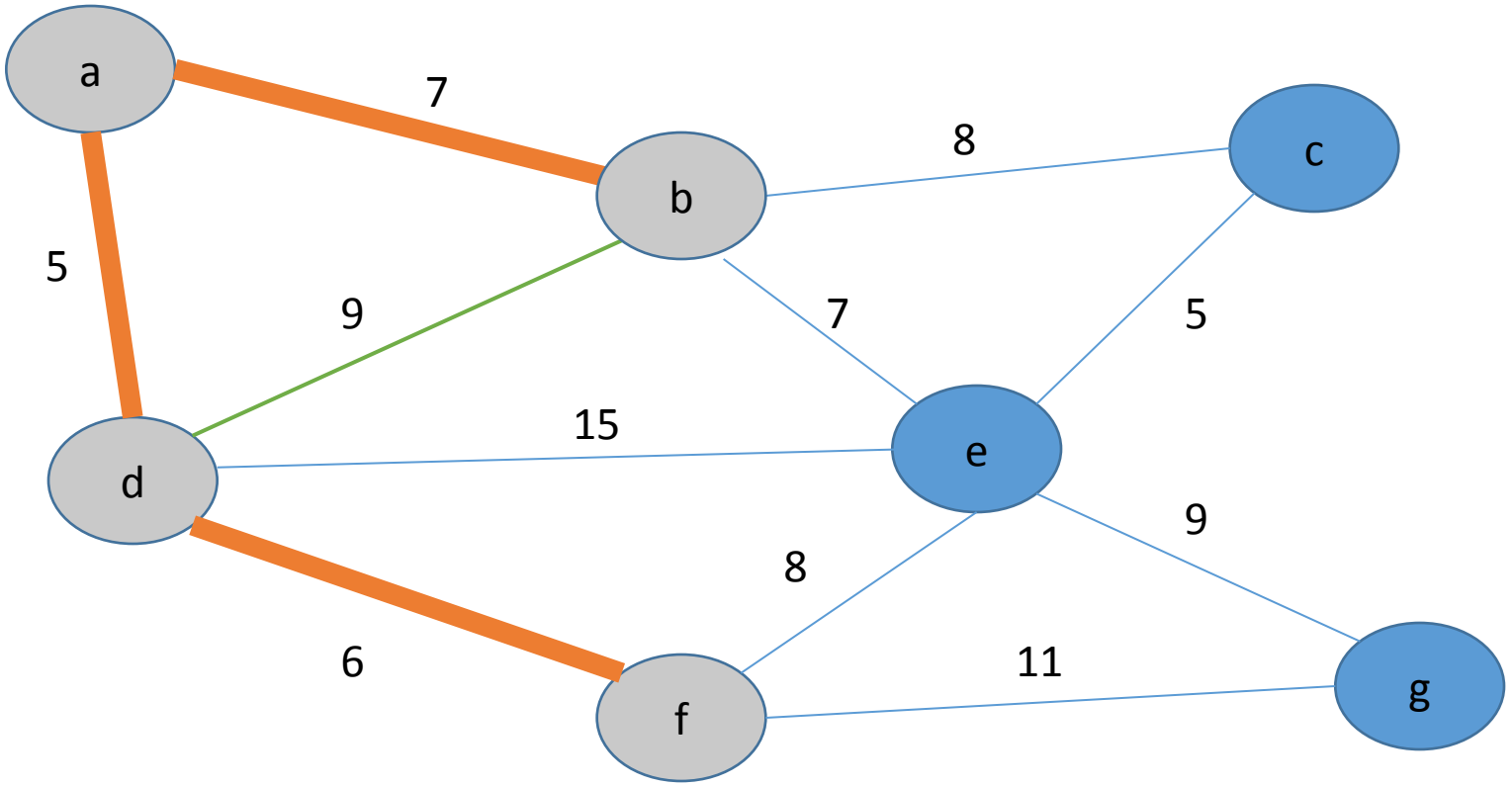
Prim's Algorithm - Example

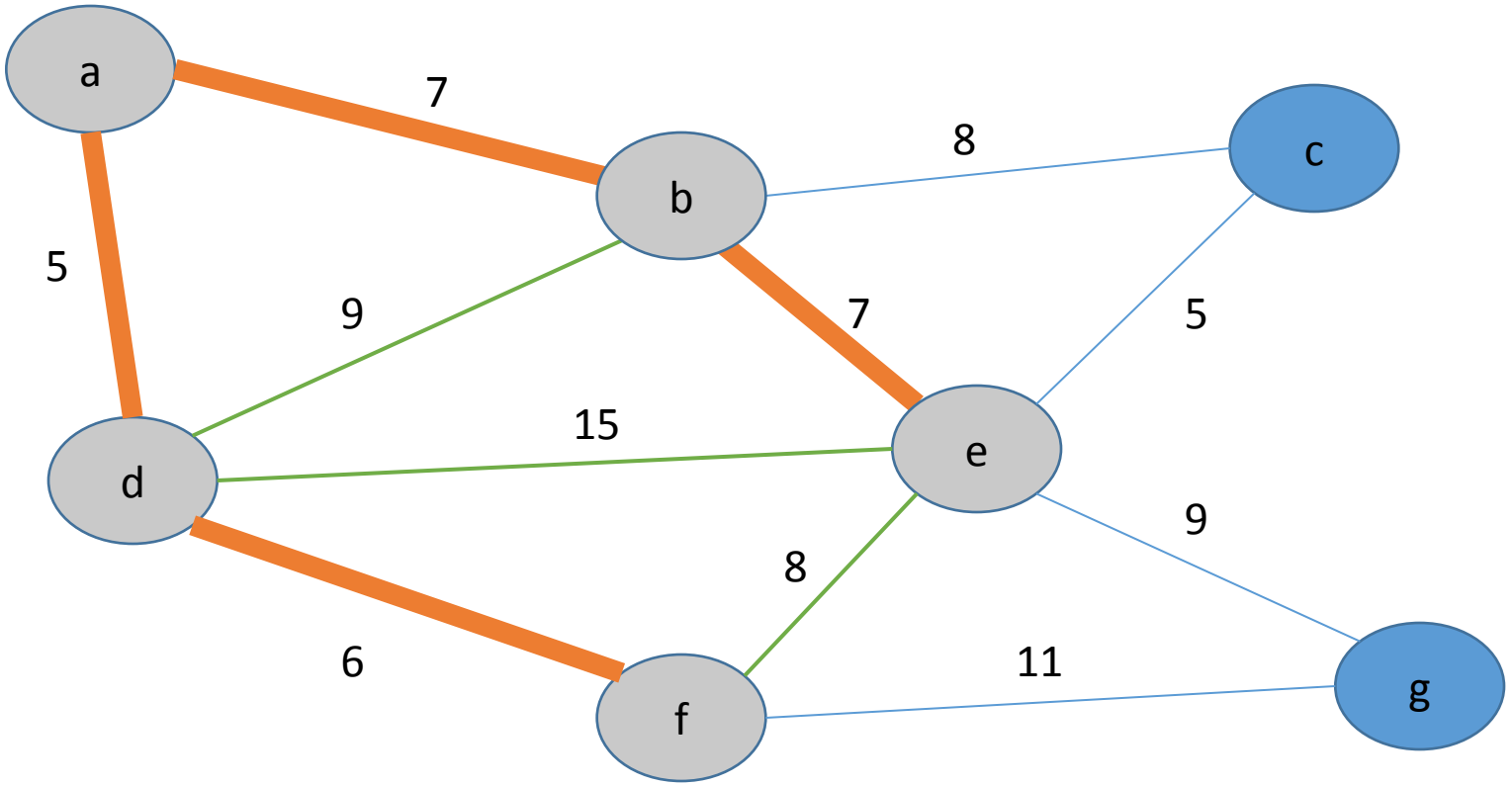


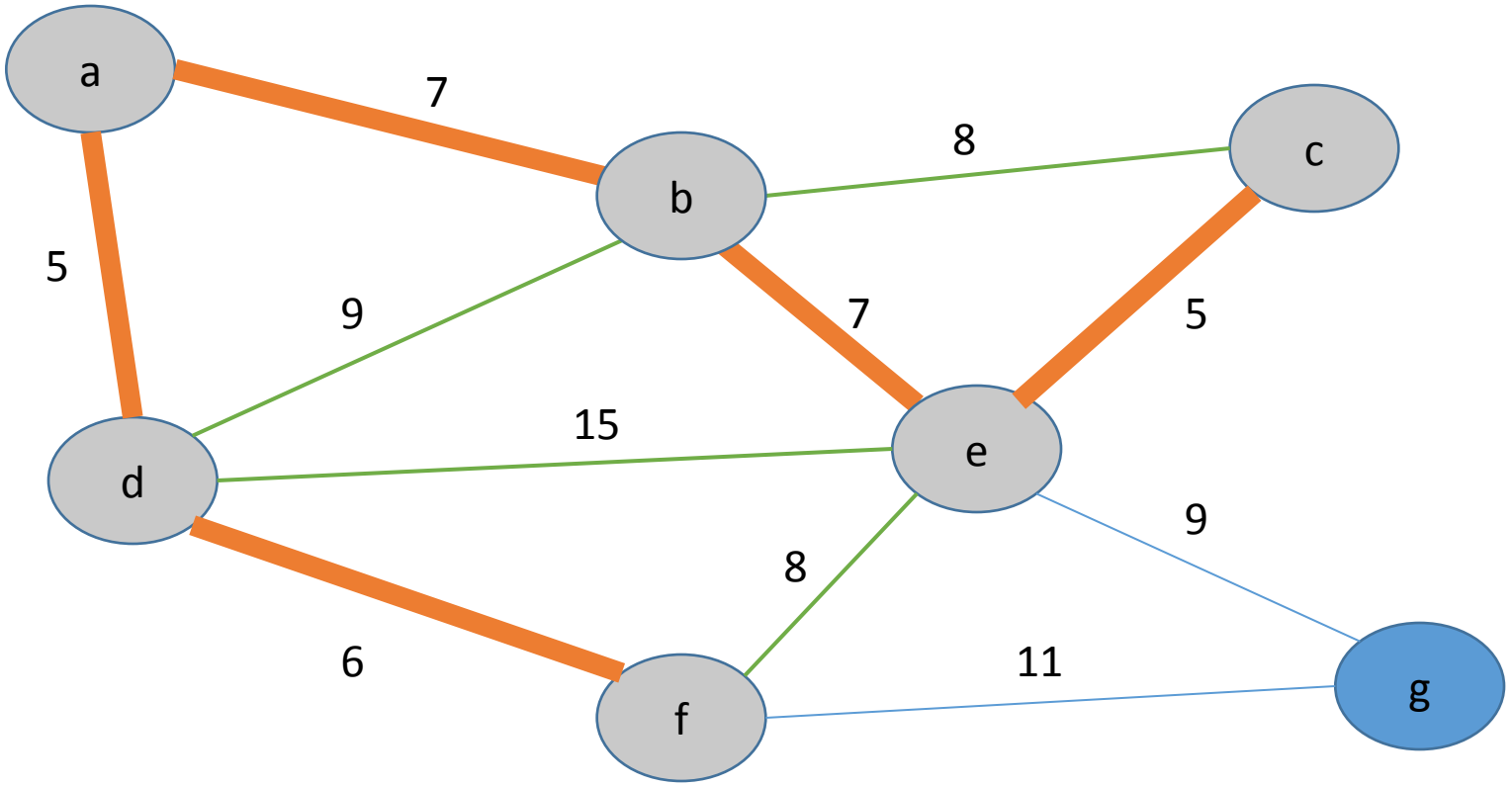
Consider vertex 'd'

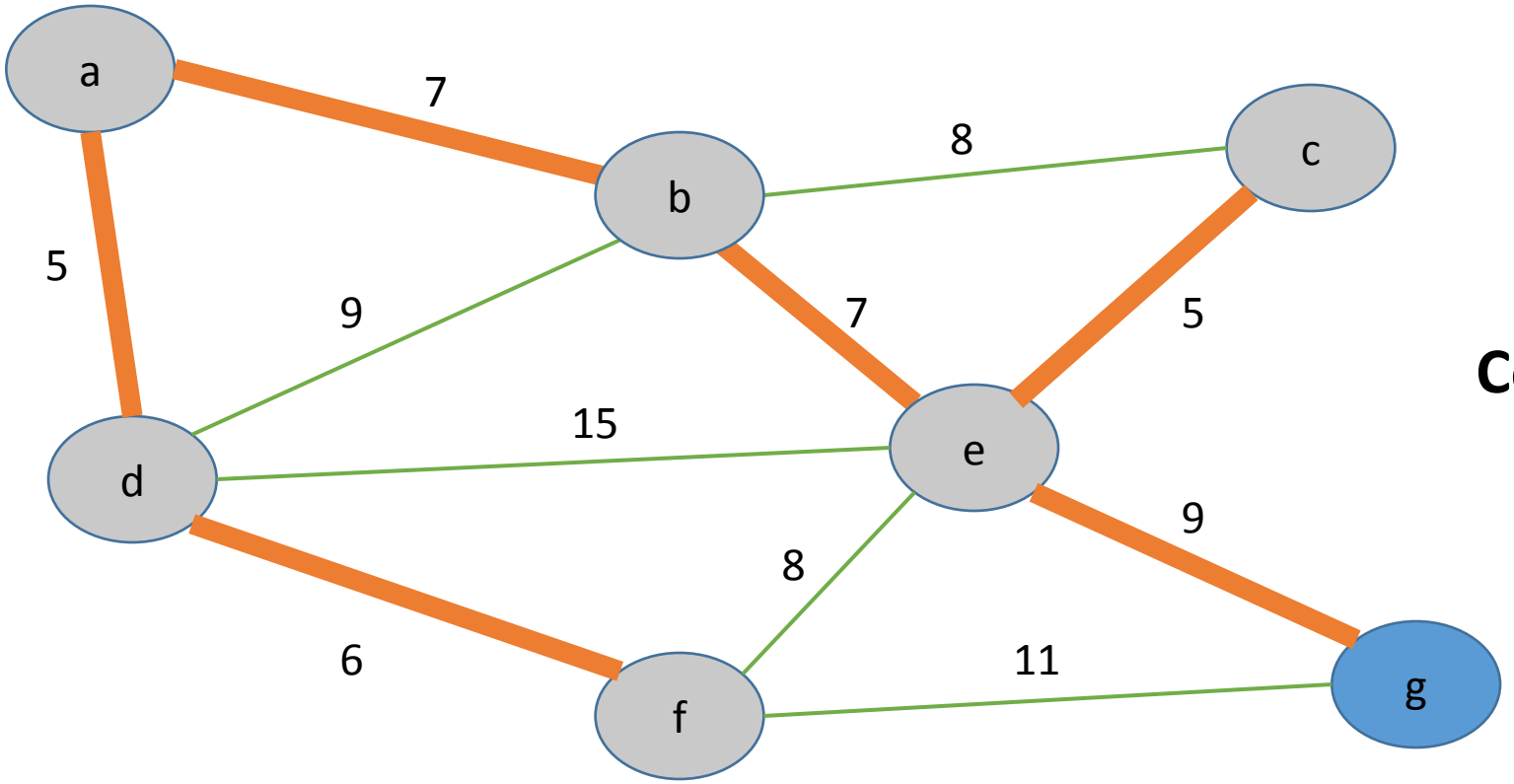

a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11


a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11


a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11


a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11


a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11



Cost = 39

Real-time Application – Prim's Algorithm

- Network for roads and Rail tracks connecting all the cities.
- Irrigation channels and placing microwave towers
- Designing a fiber-optic grid or ICs.
- Travelling Salesman Problem.
- Cluster analysis.
- Path finding algorithms used in AI.
- Game Development
- Cognitive Science

Minimum Spanning tree – Kruskal's Algorithm

- Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree.
- Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.
- Kruskal's Algorithm is preferred when-
  - The graph is sparse.
  - There are less number of edges in the graph like E = O(V)
  - The edges are already sorted or can be sorted in linear time.

Steps for finding MST - Kruskal's algorithm

Step 1: Sort all the edges in non-decreasing order of their weight.

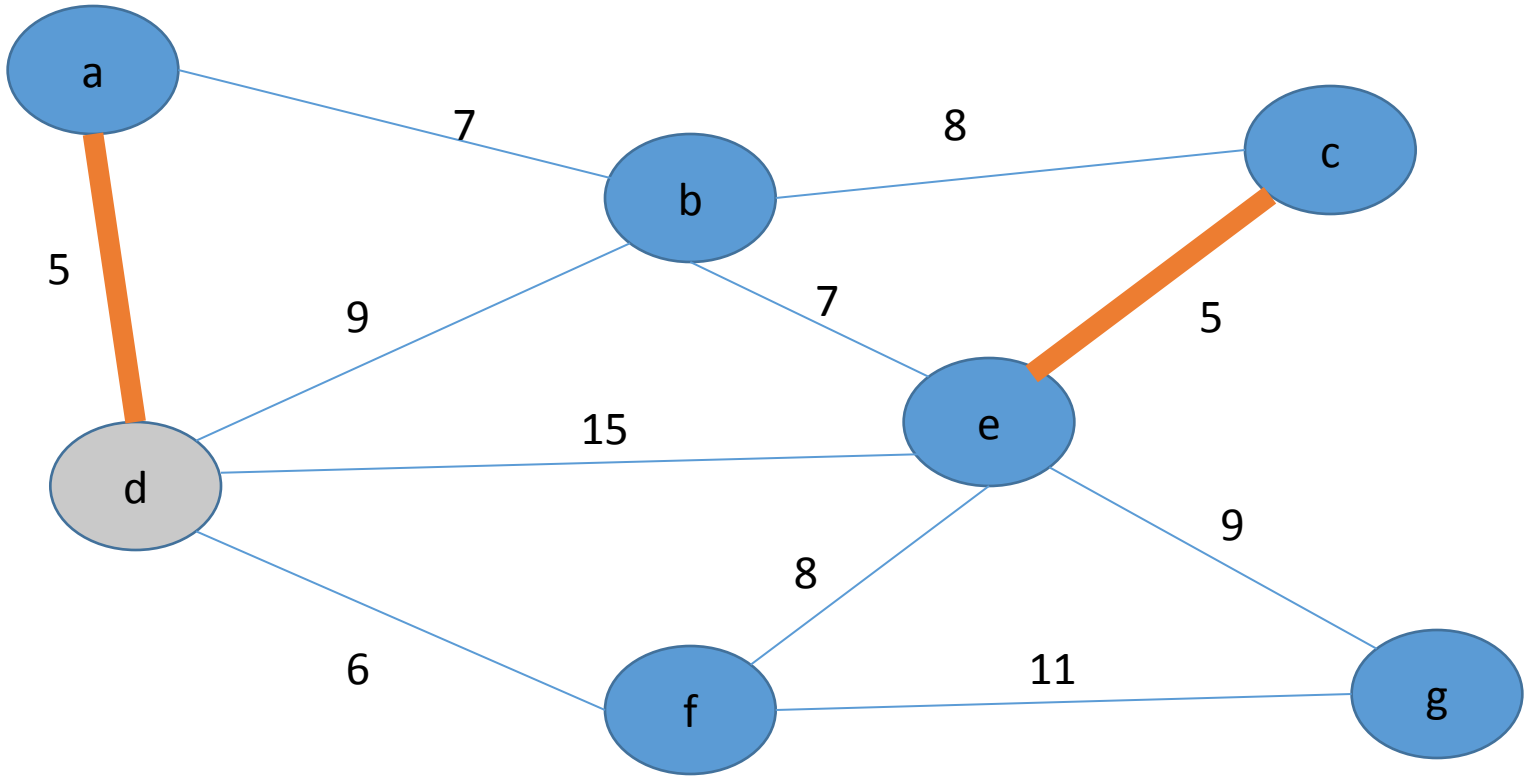
Step 2: Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

Step 3: Repeat step#2 until there are (V-1) edges in the spanning tree.

Kruskal's Algorithm

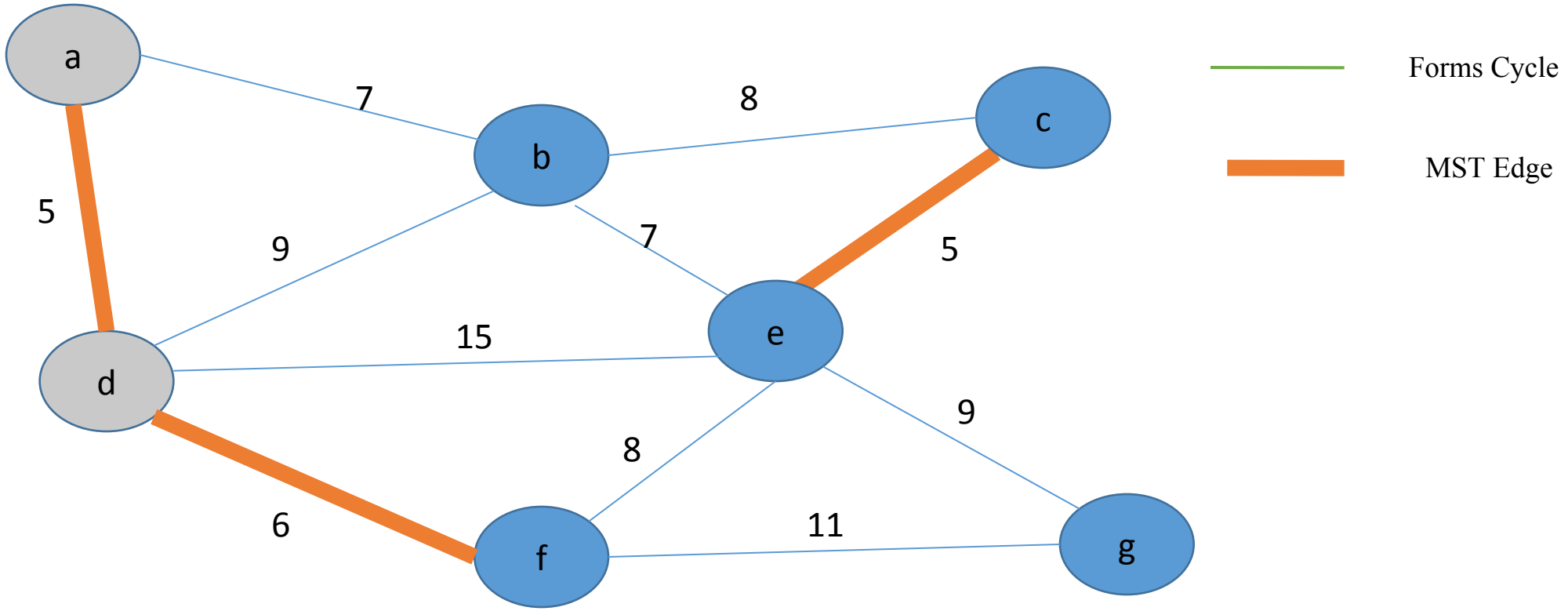
```
algorithm Kruskal(G) is
    F:= ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠
    FIND-SET(v) then
            F:= F ∪ {(u, v)} UNION(FIND-SET(u), FIND-SET(v))
    return F
```

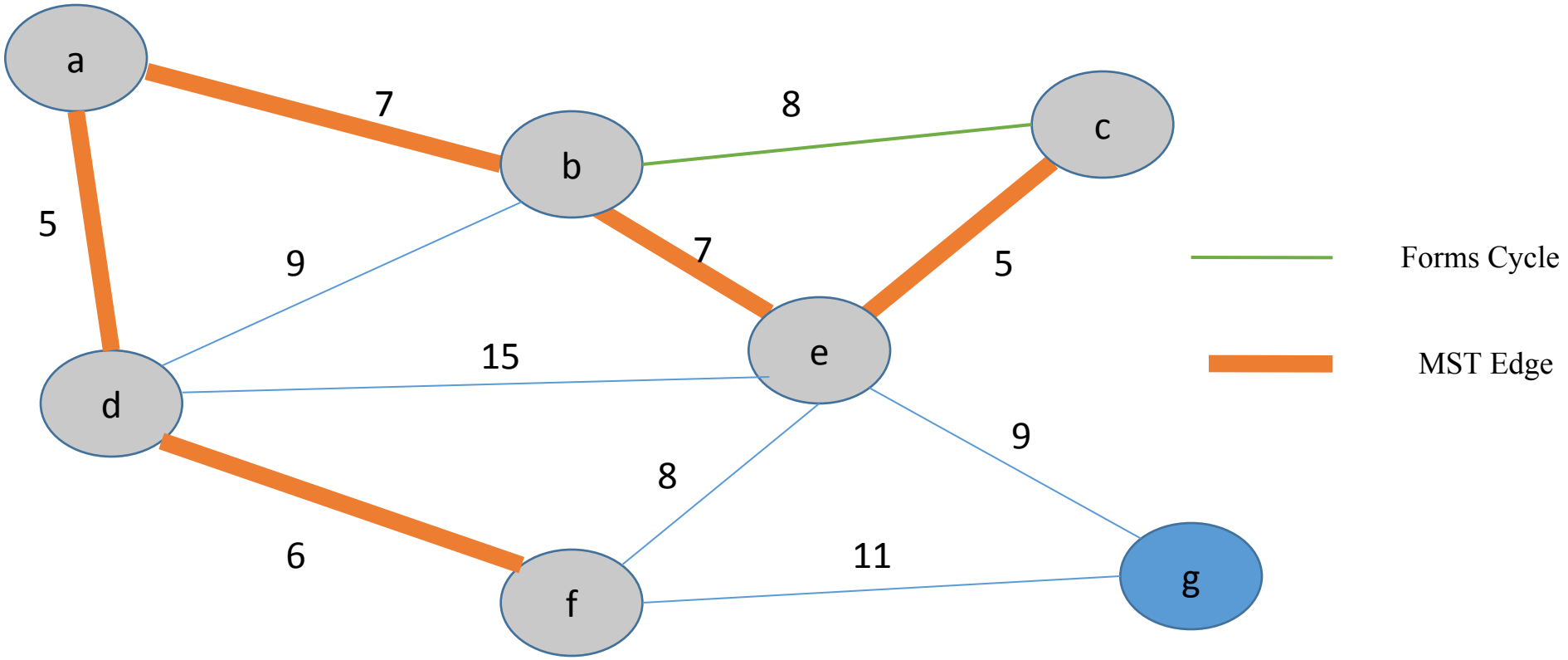
Example – Kruskal's Algorithm

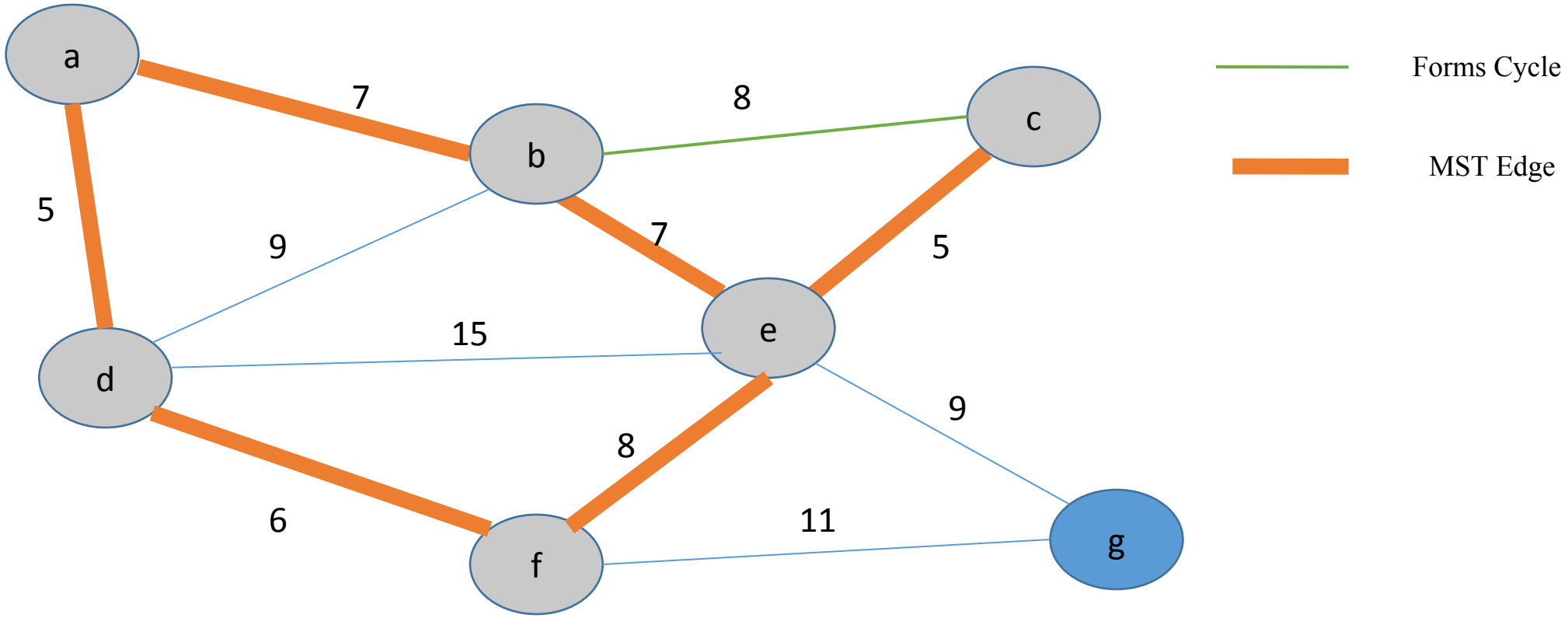


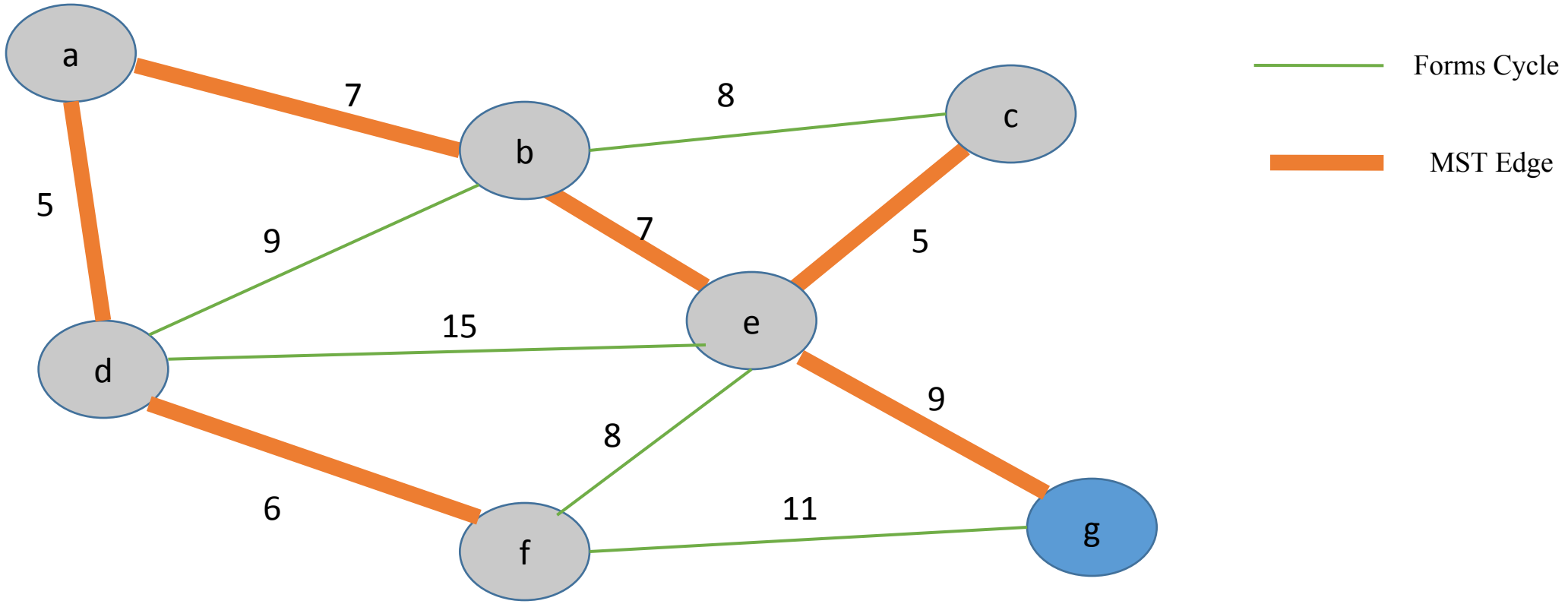
Forms Cycle

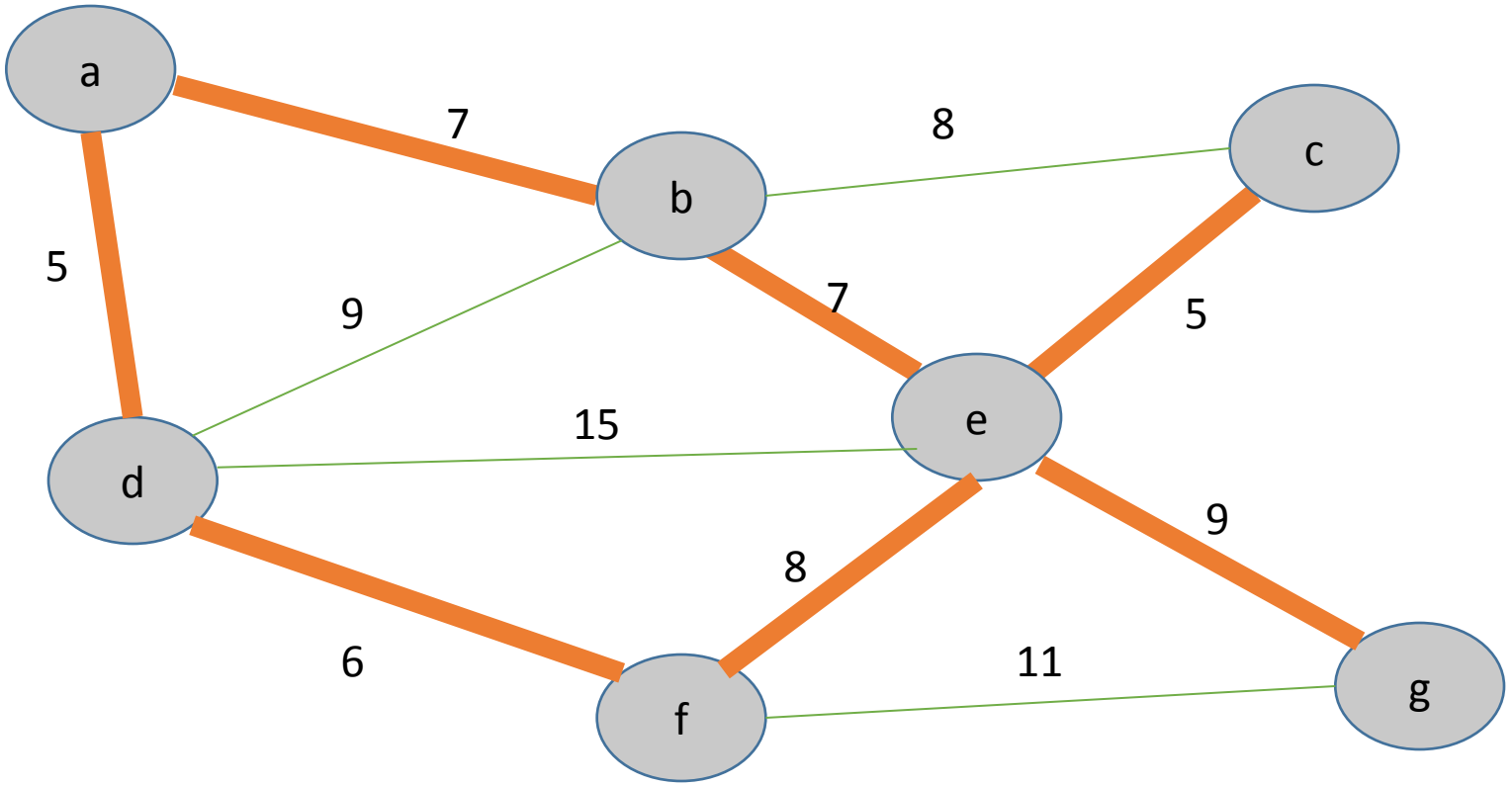
MST Edge


a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11
Forms Cycle
MST Edge


a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11
Forms Cycle
MST Edge


a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11
Forms Cycle
MST Edge


a
b
c
d
e
f
g
7
8
5
9
7
5
15
9
8
6
11
Forms Cycle
MST Edge



Cost = 39

Time Complexity –Kruskal's Algorithm

- Sorting of edges takes O(ELogE) time.
- After sorting, we iterate through all edges and **Time Complexity – Kruskal's Algorithm**
- Apply find-union algorithm. The find and union operations can take atmost O(LogV) time.
- So overall complexity is O(ELogE + ELogV) time. The value of E can be atmost $O(V^2)$, so O(LogV) are O(LogE) same.
- Therefore, overall time complexity is O(ElogE) or O(ElogV)

Real-time Application – Kruskal's Algorithm

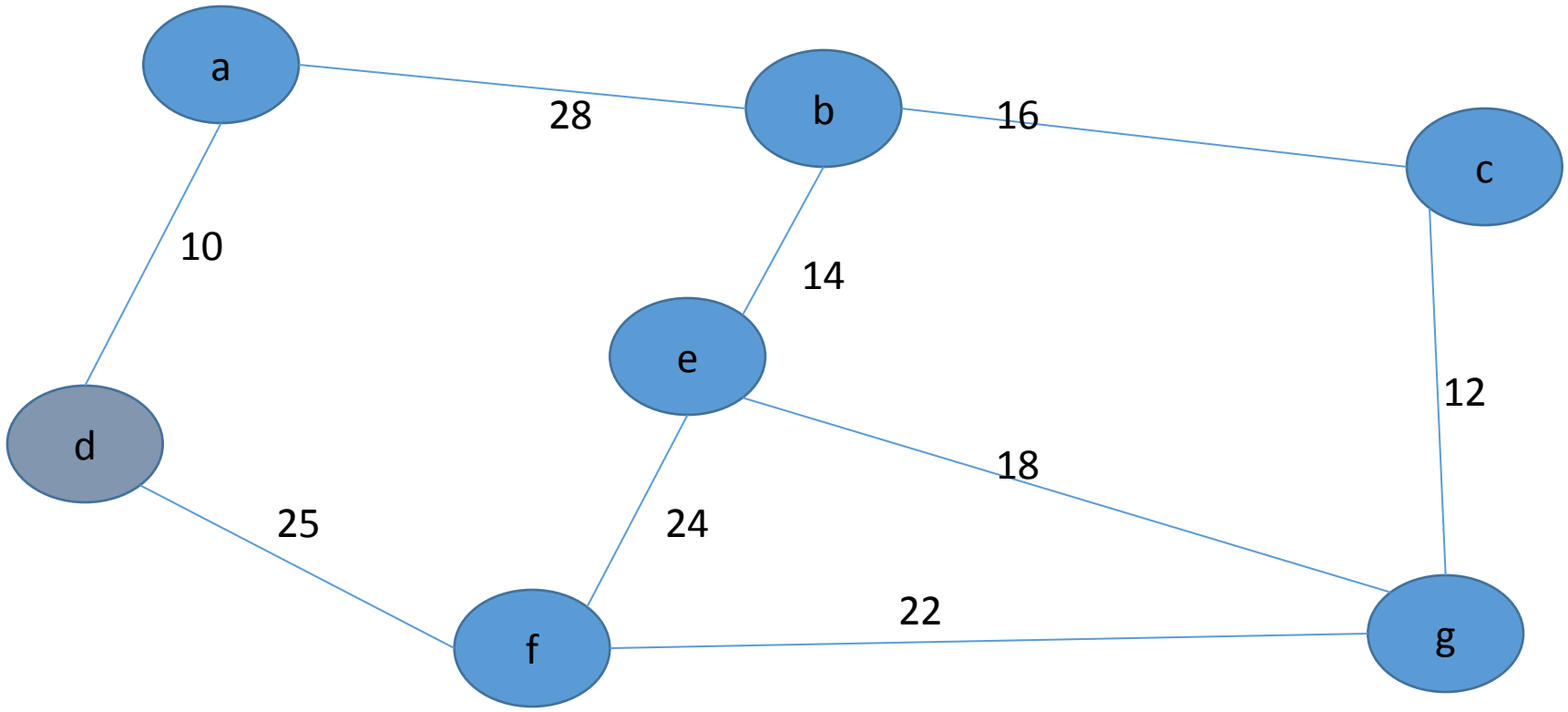
- Landing cables
- TV Network
- Tour Operations
- LAN Networks
- A network of pipes for drinking water or natural gas.
- An electric grid
- Single-link Cluster

Difference between Prim's and Kruskal's Algorithm

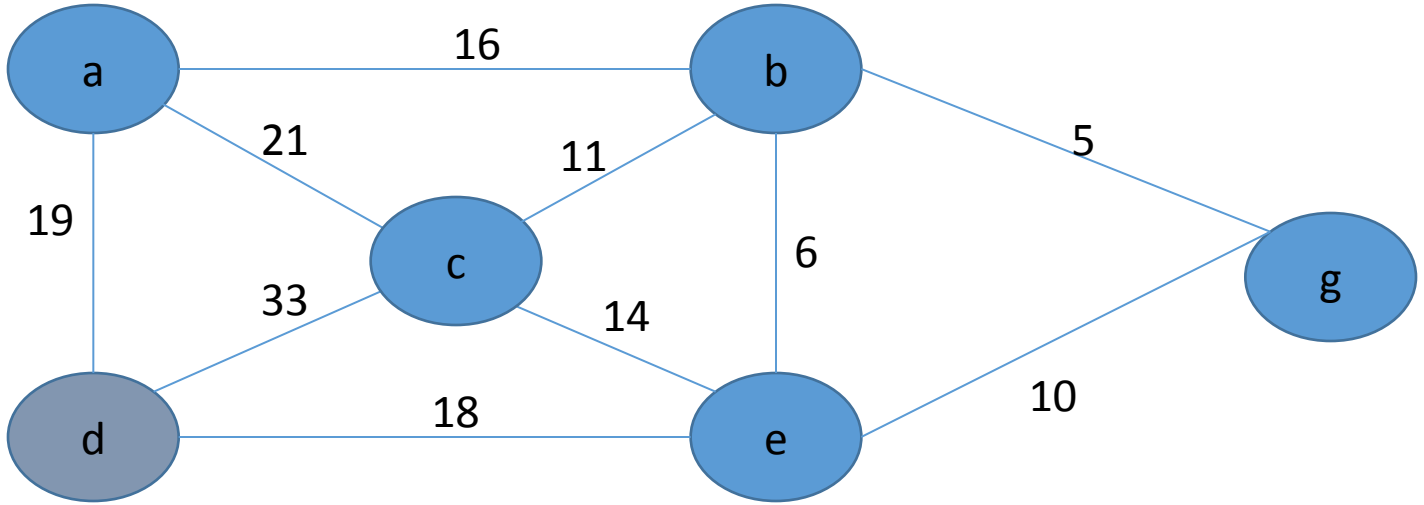
Prim's Algorithm	Kruskal's Algorithm
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

Activity

1. Find the minimum spanning tree of the graph given below using Prim's and Kruskal's Algorithms



2. Find the minimum spanning tree of the graph given below using Prim's and Kruskal's Algorithms



Summary

- Both algorithms will always give solutions with the same length.
- They will usually select edges in a different order.
- Occasionally they will use different edges – this may happen when you have to choose between edges with the same length.

Dynamic Programming

Introduction to Dynamic Programming

Session Learning Outcome-SLO

- Critically analyze the different algorithm design techniques for a given problem.
- To apply dynamic programming types techniques to solve polynomial time problems.

What is Dynamic Programming?

- Dynamic programming is a method of solving complex problems by breaking them down into sub-problems that can be solved by working backwards from the last stage.

- Coined by Richard Bellman who described dynamic programming as the way of solving problems where you need to find the best decisions one after another

Real Time applications

- 0/1 knapsack problem.
- Mathematical optimization problem.
- All pair Shortest path problem.
- Reliability design problem.
- Longest common subsequence (LCS)
- Flight control and robotics control.
- Time sharing: It schedules the job to maximize CPU usage.

Steps to Dynamic Programming

- Every problem is divided into stages
- Each stage requires a decision
- Decisions are made to determine the state of the next stage
- The solution procedure is to find an optimal solution at each stage for every possible state
- This solution procedure often starts at the last stage and works its way forward

0/1 Knapsack Problem

Given n items of known weights $w_1,\ldots,w_n$ and values $v_1,\ldots,v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Example

- Let's illustrate that point with an example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

- **The maximum weight the knapsack can hold is 20.**
- The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$
- BUT the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$.
  - In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$.
    - (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

Recursive formula

- Re-work the way that builds upon previous sub-problems
  - Let **B[k, w]** represent the maximum total value of a subset S_k with weight w.
  - Our goal is to find **B[n, W],** where n is the total number of items and W is the maximal weight the knapsack can carry.

- Recursive formula for subproblems:

$$B[k, w] = B[k - 1, w], \text{ if } w_k > w$$

$$= \max \{ B[k - 1, w], B[k - 1, w - w_k] + v_k \}, \text{ otherwise}$$

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.

First case: $w_k > w$

- Item k can't be part of the solution. If it was the total weight would be > w, which is unacceptable.

Second case: $w_k \leq w$

- Then the item k can be in the solution, and we choose the case with greater value.

Algorithm

```
for w = 0 to W {  // Initialize 1st row to 0's
    B[0,w] = 0
}
for i = 1 to n {  // Initialize 1st column to 0's
    B[i,0] = 0
}
for i = 1 to n {
    for w = 0 to W {
        if w_i <= w {  //item i can be in the solution
            if v_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = v_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        }
        else B[i,w] = B[i-1,w] // w_i > w
    }
}
```

Example Problem

Let's run our algorithm on the following data:

- n = 4 (# of elements)
- W = 5 (max weight)
- Elements (weight, value):

  (2,3), (3,4), (4,5), (5,6)

Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
// Initialize the base cases
for w = 0 to W
        B[0,w] = 0

for i = 1 to n
        B[i,0] = 0
```

Knapsack 0/1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 1$

$w\text{-}w_i = -1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0/1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 2}$

$w - w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i,w] = v_i + B[i-1, w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0/1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 3}$

$w - w_i = 1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0/1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 4}$

$w - w_i = 2$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1,w]$

$\mathbf{B[i,w] = v_i + B[i-1, w- w_i]}$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0/1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 5}$

$w - w_i = 3$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1,w]$

$\mathbf{B[i,w] = v_i + B[i-1, w- w_i]}$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0/1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 1}$

$w - w_i = -2$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i,w] = v_i + B[i-1, w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 2}$

$w - w_i = -1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	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		
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 3}$

$w - w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	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	
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 4}$

$w - w_i = 1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	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					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 5}$

$w - w_i = 2$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1,w]$

$\mathbf{B[i,w] = v_i + B[i-1, w- w_i]}$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	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		
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$w = 1..3$

$w\text{-}w_i = -3..-1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	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	
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 4}$

$w - w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i,w] = v_i + B[i-1, w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	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					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 5}$

$w - w_i = 1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1, w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	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	

$i = 4$

$v_i = 6$

$w_i = 5$

$w = 1..4$

$w\text{-}w_i = -4..-1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0/1 Example

i / w	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

$i = 4$

$v_i = 6$

$w_i = 5$

$\mathbf{w = 5}$

$w - w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i,w] = v_i + B[i-1, w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0/1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	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**

The max possible value that can be carried in this knapsack is ***$7***

Knapsack 0/1 Algorithm - Finding the Items

- This algorithm only finds the max possible value that can be carried in the knapsack
  - The value in B[n,W]

- To know the ***items*** that make this maximum value, we need to trace back through the table.

Knapsack 0/1 Algorithm
Finding the Items

- Let $i = n$ and $k = W$

  if $B[i, k] \neq B[i-1, k]$ then

  mark the ith item as in the knapsack

  $i = i-1$, $k = k-w_i$

  else

  $i = i-1$ // Assume the ith item is not in the knapsack

  // Could it be in the optimally packed knapsack?

Knapsack:

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Finding the Items

i / w	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

$i = 4$
$k = 5$
$v_i = 6$
$w_i = 5$
B[i,k] = 7
$B[i-1,k] = 7$

```
i = n , k = W
while  i, k > 0
    if B[i, k] ≠ B[i-1, k] then
        mark the i^th item as in the knapsack
        i = i-1, k = k-w_i
    else
        i = i-1
```

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

i / w	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

$i = 3$
$k = 5$
$v_i = 5$
$w_i = 4$
B[i,k] = 7
$B[i-1,k] = 7$

```
i = n , k = W
while  i, k > 0
    if B[i, k] ≠ B[i-1, k] then
        mark the i^th item as in the knapsack
        i = i-1, k = k-w_i
    else
        i = i-1
```

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2

i / w	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

$i = 2$
$k = 5$
$v_i = 4$
$w_i = 3$
B[i,k] = 7
$B[i-1,k] = 3$
$k - w_i = 2$

$i = n$, $k = W$
while $i, k > 0$
    if $B[i, k] \neq B[i-1, k]$ then
        mark the ith item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

i / w	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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{B[i,k] = 3}$
$B[i-1,k] = 0$
$k - w_i = 0$

$i = n$, $k = W$
while $i, k > 0$
    if $B[i, k] \neq B[i-1, k]$ then
        mark the ith item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

i / w	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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{B[i,k] = 3}$
$B[i-1,k] = 0$
$k - w_i = 0$

k = 0, it's completed

The optimal knapsack should contain:

Item 1 and Item 2

Complexity calculation of knapsack problem

```
for w = 0 to W
    B[0,w] = 0
```

O(*W*)

```
for i = 1 to n
    B[i,0] = 0
```

O(*n*)

Repeat *n* times

```
for i = 1 to n
    for w = 0 to W
        < the rest of the code >
```

O(*W*)

What is the running time of this algorithm?

O(*nW*)**

Home Assignment

1. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem. How to find out which items are in optimal subset?

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

, capacity $W = 6$.

2. Solve an instance of the knapsack problem by the dynamic programming algorithm.

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

Matrix-chain Multiplication

- Suppose we have a sequence or chain $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied
  - That is, we want to compute the product $A_1 A_2 \ldots A_n$

- There are many possible ways (parenthesizations) to compute the product

Matrix-chain Multiplication …contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication …contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)

Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY($A_{p\times q}$, $B_{q\times r}$)

```
1.  for i ← 1 to p
2.          for j ← 1 to r
3.              C[i, j] ← 0
4.              for k ← 1 to q
5.                  C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6.  return C
```

Scalar multiplication in line 5 dominates time to compute CNumber of scalar multiplications = *pqr*

Matrix-chain Multiplication …contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

Total: 7,500

Total: 75,000

Matrix-chain Multiplication …contd

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for i=1, 2, …, n, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1 A_2 \ldots A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Dynamic Programming Approach

- The structure of an optimal solution
  - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \ldots A_j$
  - An optimal parenthesization of the product $A_1 A_2 \ldots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \leq k < n$
  - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach …contd

- **Key observation**: parenthesizations of the subchains $A_1A_2\ldots A_k$ and $A_{k+1}A_{k+2}\ldots A_n$ must also be optimal if the parenthesization of the chain $A_1A_2\ldots A_n$ is optimal (why?)

- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

Dynamic Programming Approach …contd

- Recursive definition of the value of an optimal solution
  - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
  - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
  - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \leq k < j$

Dynamic Programming Approach …contd

- $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$

- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \qquad for\ i \leq k < j$
- $m[i, i] = 0$ for $i=1,2,\ldots,n$

Dynamic Programming Approach ...contd

- But… optimal parenthesization occurs at one value of k among all possible $i \leq k < j$
- Check all these and select the best one

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

Dynamic Programming Approach …contd

- To keep track of how to construct an optimal solution, we use a table s
- $s[i, j] =$ value of k at which $A_i A_{i+1} \ldots A_j$ is split for optimal parenthesization
- Algorithm: next slide
  - First computes costs for chains of length l=1
  - Then for chains of length l=2,3, … and so on
  - Computes the optimal cost bottom-up

Algorithm to Compute Optimal Cost

Input: Array $p[0 \ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s

```
MATRIX-CHAIN-ORDER(p[ ], n)
    for i ← 1 to n
        m[i, i] ← 0
    for l ← 2 to n
        for i ← 1 to n-l+1
            j ← i+l-1
            m[i, j] ← ∞
            for k ← i to j-1
                q ← m[i, k] + m[k+1, j] + p[i-1] p[k] p[j]
                if q < m[i, j]
                    m[i, j] ← q
                    s[i, j] ← k
    return m and s
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Constructing Optimal Solution

- Our algorithm computes the minimum-cost table m and the split table s
- The optimal solution can be constructed from the split table s
  - Each entry $s[i, j] = k$ shows where to split the product $A_i A_{i+1} \ldots A_j$ for the minimum cost

Example

- Show how to multiply this matrix chain optimally

- Solution on the board
  - Minimum cost 15,125
  - Optimal parenthesization $((A_1(A_2A_3))((A_4 A_5)A_6))$

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Longest Common Subsequence(LCS)

Dynamic Programming

LCS using Dynamic Programming

- SLO 1: To understand the Longest Common Subsequence(LCS) Problem and how it can be solved using Dynamic programming approach.

LCS Problem Statement

- Given two sequences, find the length of longest subsequence present in both of them.

Example

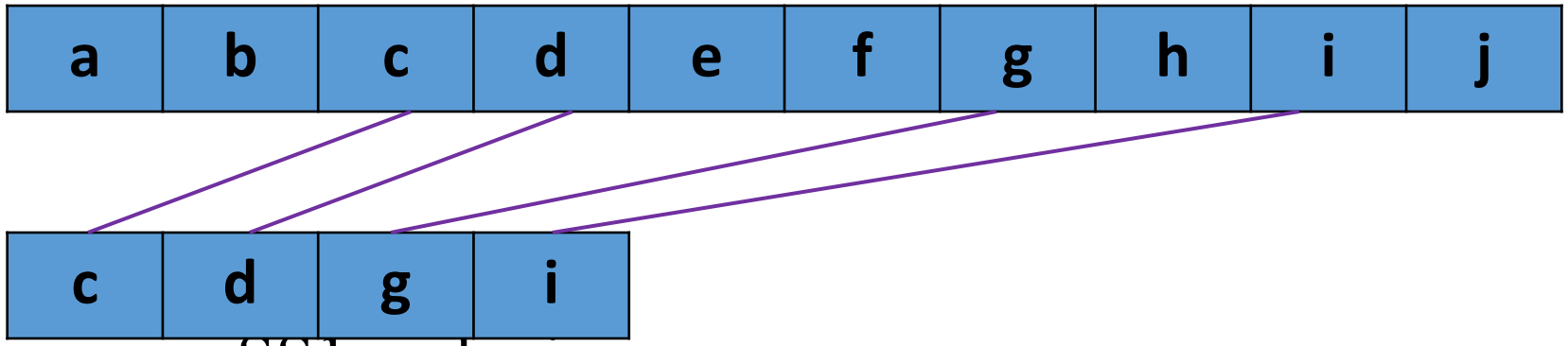
- A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
- Let String S1= a b c d
- Subsequence

ab	bd	~~ca~~	ac	ad	~~db~~	ba	acd	bcd	abcd

- Length of LCS = 4
- Subsequence : abcd

LCS Example

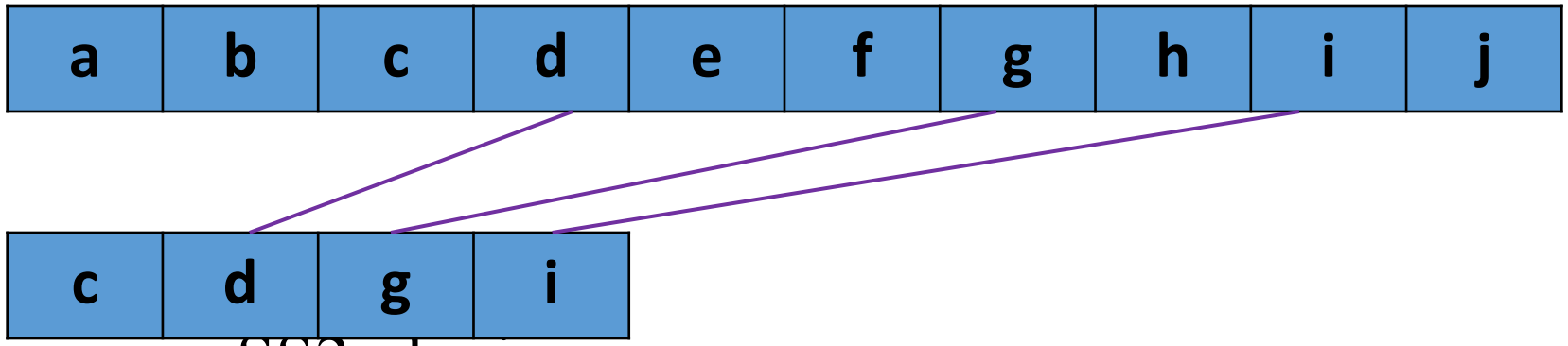
- S1= a b c d e f g h i j
- S2= c d g i
- Subsequence
- S1:



- S2:
- Subsequence, SS1: c d g i

Example

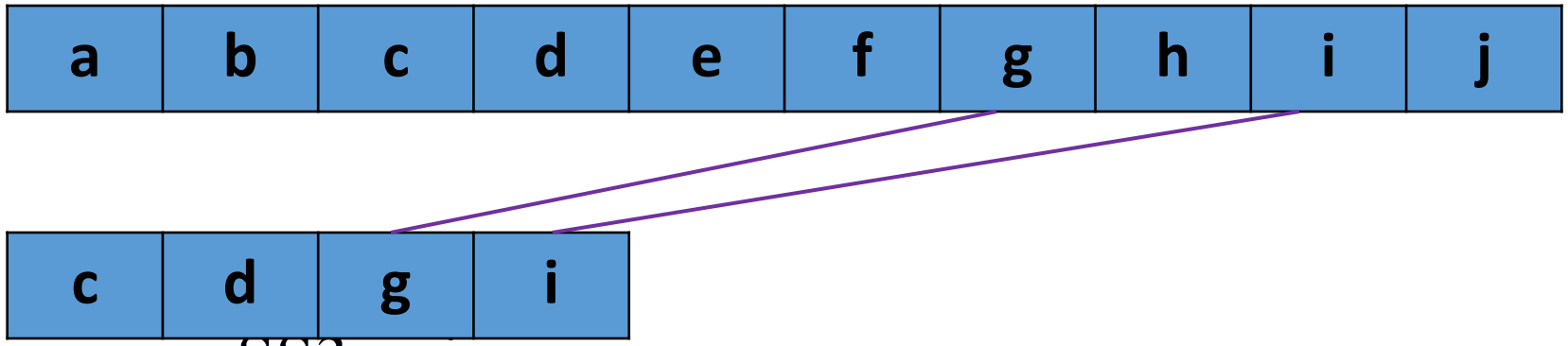
- S1= a b c d e f g h i j
- S2= c d g i
- Subsequence
- S1:



- S2:
- Subsequence, SS2: d g i

Example

- S1= a b c d e f g h i j
- S2= c d g i
- Subsequence
- S1:



- S2:
- Subsequence, SS3: g i

Example

- SS1= c d g i
- SS2: d g i
- SS3: g i
- **Longest Common Subsequence**: subsequence with maximum length.

  maximum(length(subsequence1),
  length(subsequence2)….
  length(subsequence n)).

Longest Common Subsequence :

maximum(length(c d g i), length(d g i),length(g i)).

- Therefore LCS (S1,S2) is c d g i with length 4

Activity

- Give the LCS for the following strings

- X: ABCBDAB
  Y: BDCABA

Motivation

Brute Force Approach-Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

Analysis

- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Hence, the runtime would be exponential !

Example:

Consider S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA…..

S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA….

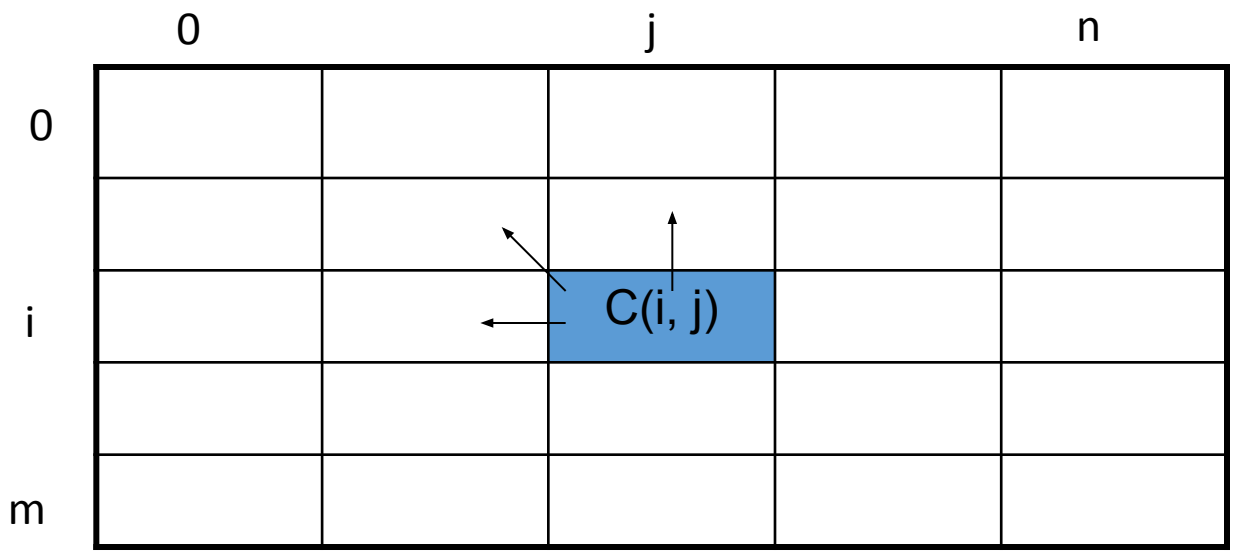
Towards a better algorithm: a Dynamic Prpgramming strategy

- Key: optimal substructure and overlapping sub-problems
- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.

LCS Problem Formulation using DP Algorithm

- Key: find out the correct order to solve the sub-problems
- Total number of sub-problems: $m * n$

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max\{c[i-1,j],\, c[i,j-1]\} & \text{otherwise.} \end{cases}$$



LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the LCS of X and Y?

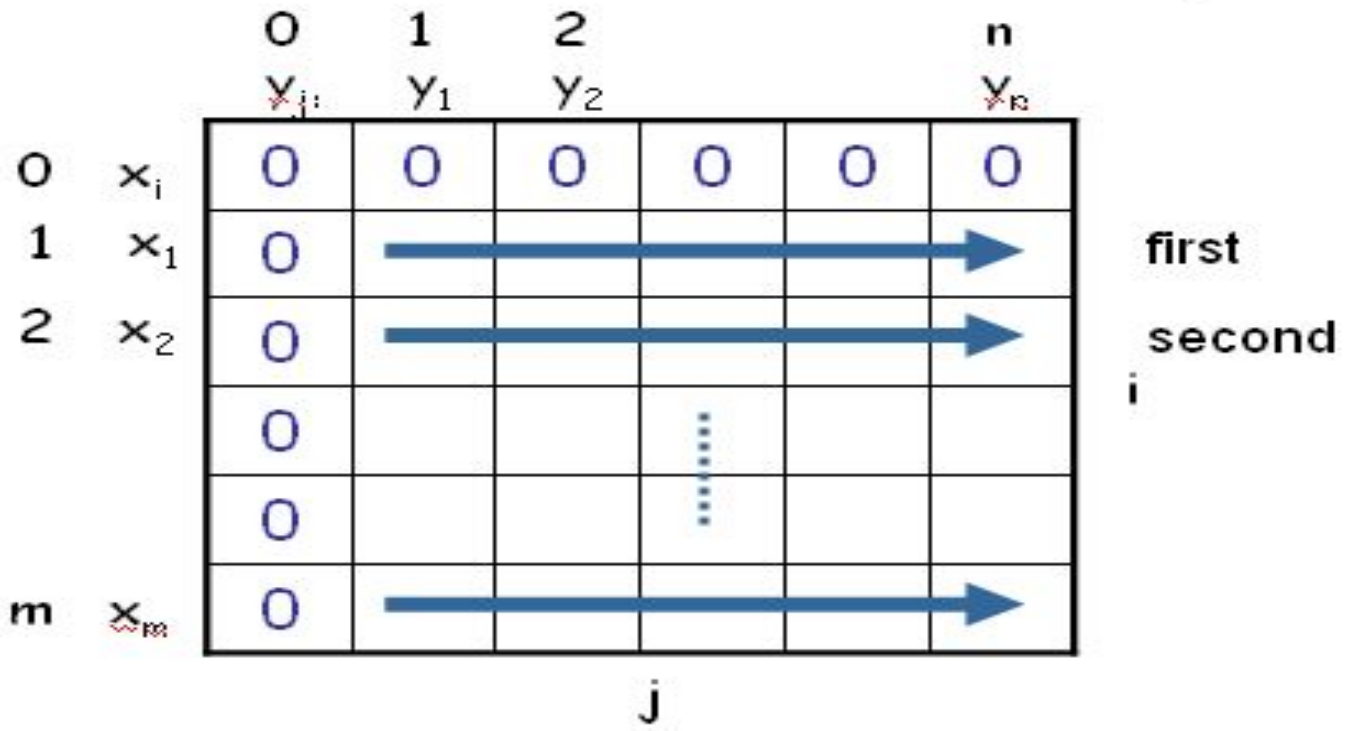
LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

Computing the Length of the LCS

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$



LCS Example (0)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m c[i,0] = 0
for j = 1 to n c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (3)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

    c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (6)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB

BDCAB

i	j	0	1	2	3	4	5
	Y[j]		B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (8)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

    c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (9)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (10)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (11)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (13)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

```
if ( X_i == Y_j )
      c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

Finding LCS

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Time for trace back: $O(m+n)$.

Finding LCS (2)(Bottom Up approach)

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

B C B

LCS: BCB

Algorithm steps for solving LCS problem using Dynamic Programming

- Step 1: Characterizing a longest common subsequence
- Step 2: A recursive solution
- Step 3: Computing the length of an LCS
- Step 4: Constructing an LCS

Step 1: Characterizing a longest common subsequence

- Optimal substructure of LCS: a problem is said to have **optimal substructure** if an optimal solution can be constructed from optimal solutions of its sub problems.

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step 2: A recursive solution

- Optimal substructure of LCS problem gives the recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

Step 3: Computing the length of an LCS

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Step 4: Constructing an LCS

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Time Complexity Analysis of LCS Algorithm

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

Asymptotic Analysis:

Worst case time complexity: O(n*m)
Average case time complexity: O(n*m)
Best case time complexity: O(n*m)
Space complexity: O(n*m)

Real Time Application

- **Molecular biology**. DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four sub molecules forming DNA.
  - To find a new sequences by generating the LCS of existing similar sequences
  - To find the similarity of two DNA Sequence by finding the length of their longest common subsequence.
- **File comparison**. The Unix program "diff" is used to compare two different versions of the same file, to determine what changes have been made to the file.
  - It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed.
- **Screen redisplay**. Many text editors like "emacs" display part of a file on the screen, updating the screen image as the file is changed.
  - Can be a sort of common subsequence problem (the common subsequence tells you the parts of the display that are already correct and don't need to be changed)

Summary

- Solving LCS problem by brute force approach requires $O(2^m)$.
- Applying Dynamic Programming to solve LCS problem reduces the time complexity to O(nm).

Review Questions

1. What is the time complexity of the brute force algorithm used to find the longest common subsequence?
2. What is the time complexity of the dynamic programming algorithm used to find the longest common subsequence?
3. If X[i]==Y[i] what is the value stored in c[i,j]?
4. If X[i]!=Y[i] what is the value stored in c[i,j]?
5. What is the value stored in zeroth row and zeroth column of the LCS table?

Home Assignment

- Determine the LCS of (1,0,0,1,0,1,0,1) and (0,1,0,1,1,0,1,1,0).

OPTIMAL BINARY SEARCH TREE

- Session Learning Outcome-SLO
- Motivation of the topic
- Binary Search Tree
- Optimal Binary Search Tree
- Example Problem
- Analysis
- Summary
- Activity /Home assignment /Questions

INTRODUCTION

Session Learning Outcome:

- To Understand the concept of Binary Search tree and Optimal Binary Search Tree
- how to construct Optimal Binary Search Tree with optimal cost

OPTIMAL BINARY SEARCH TREE(OBST)

- Binary Search tree(BST) which is mainly constructed for searching a key from it
- For searching any key from a given BST, it should take optimal time.
- For this, we need to construct a BST in such a way that it should take optimal time to search any of the key from given BST
- To construct OBST, frequency of searching of every key is required
- With the help of frequencies , construction of OBST is possible

BINARY SEARCH TREE:

A binary search tree is a special kind of binary tree. In binary search tree, the elements in the left and right sub-trees of each node are respectively lesser and greater than the element of that node. Fig. 1 shows a binary search tree.

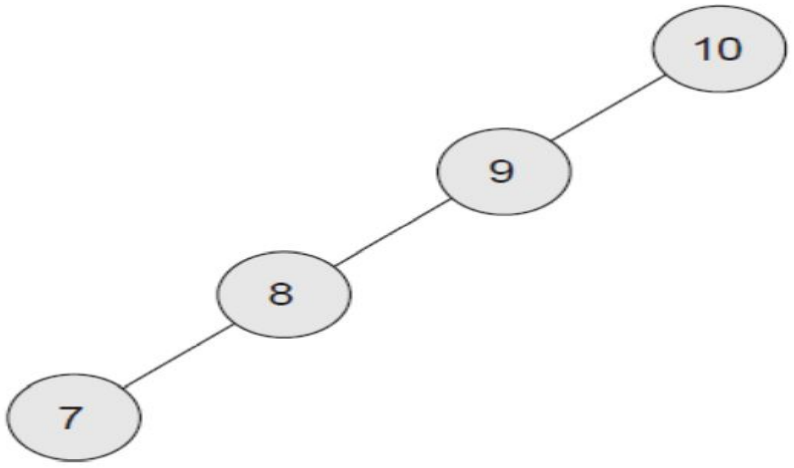


Fig. 1: Skewed Binary Search Tree

Fig. 1 is a binary search tree but is not balanced tree. On the other hand, this is a skewed tree where all the branches on one side. The advantage of binary search tree is that it facilitates search of a key easily. It takes O(n) to search for a key in a list. Whereas, search tree helps to find an element in logarithmic time.

How an element is searched in binary search tree?

- Let us assume the given element is x. Compare x with the root element of the binary tree, if the binary tree is non-empty. If it matches, the element is in the root and the algorithm terminates successfully by returning the address of the root node. If the binary tree is empty, it returns a NULL value.

  - If x is less than the element in the root, the search continues in the left sub-tree
  - If x is greater than the element in the root, the search continues in the right sub-tree.

- This is by exploiting the binary search property. There are many applications of binary search trees. One application is construction of dictionary.

- There are many ways of constructing the binary search tree. Brute force algorithm is to construct many binary search trees and finding the cost of the tree. How to find cost of the tree? The cost of the tree is obtained by multiplying the probability of the item and the level of the tree. The following example illustrates the way of find this cost of the tree.

HOW TO FIND THE COST OF BINARY SEARCH TREE

Example 1: Find the cost of the tree shown in Fig. 2 where the items probability is given as follows:
a1 = 0.4 , a2= 0.3 , a3 = 0.3

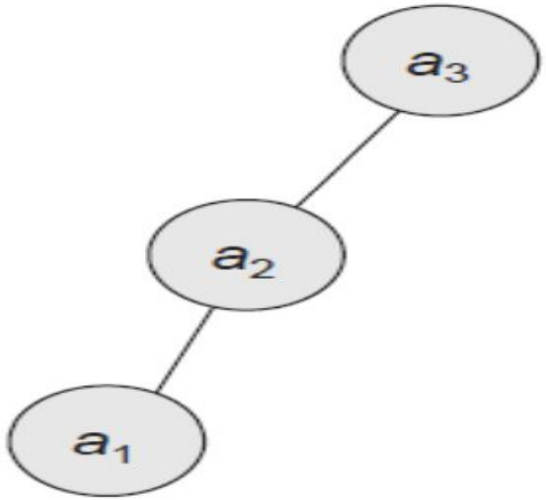


Fig. 2: Sample Binary Search Tree

Solution

As discussed earlier, the cost of the tree is obtained by multiplying the item probability and the level of the tree. The cost of the tree is computed as follows;
a1 level=3, a2 level=2, a3 level=1

$$\text{Cost of BST} = 3(0.4) + 2(0.3) + 1(0.3) = 2.1$$

It can be observed that the cost of the tree is 2.1.

OPTIMAL BINARY SEARCH TREE:

- What is an optimal binary search tree? An optimal binary search tree is a tree of optimal cost. This is illustrated in the following example

Example 2: Construct optimal binary search tree for the three items a1 = 0.4 , a2= 0.3 , a3 = 0.3 ?

Solution:

- There are many ways one can construct binary search trees. Some of the constructed binary search trees and its cost are shown in Fig. 3.

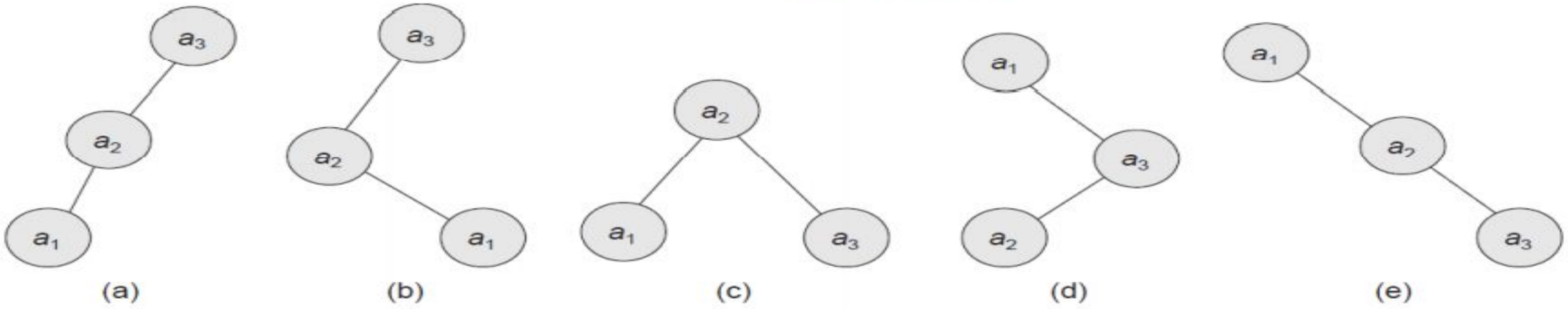


Fig. 3. : Some of the binary search trees

- It can be seen the cost of the trees are respectively, 2.1, 1.3, 1.6, 1.9 and 1.9. So the minimum cost is 1.3. Hence, the optimal binary search tree is (b) Fig. 3.

OPTIMAL BINARY SEARCH TREE(Contd..)

How to construct optimal binary search tree? The problem of optimal binary search tree is given as follows:

- Given sequence K = k1 < k2 < kn of n sorted keys, with a search probability pi for each key ki . The aim is to build a binary search tree with minimum expected cost.

- One way is to use brute force method, by exploring all possible ways and finding the expected cost. But the method is not practical as the number of trees possible is Catalan sequence. The Catalan number is given as follows:

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

If the nodes are 3, then the Catalan number is

$$C_3 = \frac{1}{3+1}\binom{6}{3} = 5$$

OPTIMAL BINARY SEARCH TREE(Contd..)

Hence, five search trees are possible. In general, $\Omega(4^n/n^{3/2})$ different BSTs are possible with n nodes. Hence, alternative way is to explore dynamic programming approach.

Dynamic programming Approach:

- The idea is to One of the keys in a1, …,an , say ak, where 1 ≤ k ≤ n, must be the root. Then , as per binary search rule, Left sub tree of ak contains a1,...,ak-1 and right subtree of ak contains ak+1, ...,an.

- So, the idea is to examine all candidate roots ak , for 1 ≤ k ≤ n and determining all optimal BSTs containing a1,...,ak-1 and containing ak+1,...,an

The informal algorithm for constructing optimal BST based on [1,3] is given as follows:

Step 1: Read n symbols with probability p_i.
Step 2: Create the table $C[i, j]$, $1 \leq i \leq j + 1 \leq n$.
Step 3: Set $C[i, i] = p_i$ and $C[i - 1, j] = 0$ for all $i \in [n]$.
Step 4: Recursively compute the following relation:

$$C[i, j] = C[1 \ldots k + 1] + C[k + 1 \ldots j] + \sum_{m=1}^{n} p_m, \quad \text{for all i and j}$$

Step 5: Return $C[1 \ldots n]$ as the maximum cost of constructing a BST.
Step 6: End.

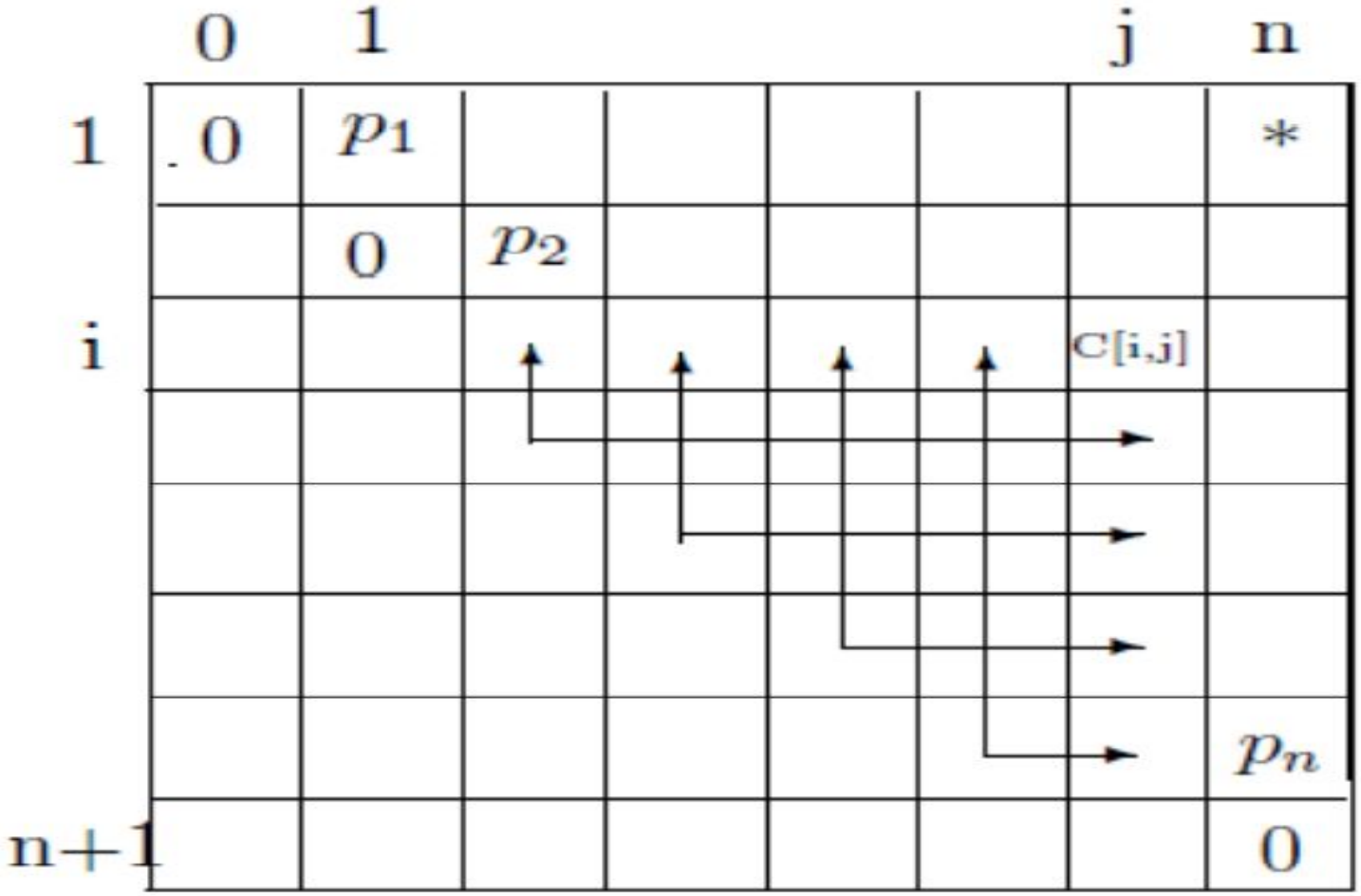
The idea is to create a table as shown in below [2]

Table 2: Constructed Table for building Optimal BST

OPTIMAL BINARY SEARCH TREE(Contd..)

The idea is to create a table as shown in below [2]

Table 2: Constructed Table for building Optimal BST



OPTIMAL BINARY SEARCH TREE(Contd..)

- The aim of the dynamic programming approach is to fill this table for constructing optimal BST.
- What should be entry of this table? For example, to compute C[2,3] of two items, say key 2 and key 3, two possible trees are constructed as shown below in Fig. 4 and filling the table with minimum cost.

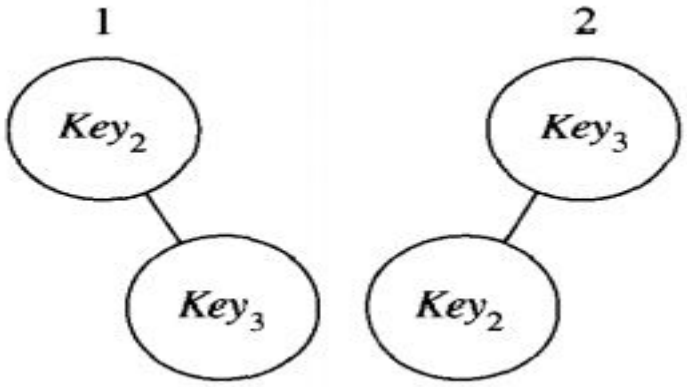


Fig. 4: Two possible ways of BST for key 2 and key 3.

OPTIMAL BINARY SEARCH TREE(Contd..)

Example 3: Let there be four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability 2/7 1/7 3/7 1/7 . **Apply dynamic programming approach and construct optimal binary search trees?**

Solution:

The initial Table is given below in Table 1,

Table 1: Initial table

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

It can be observed that the table entries are initial probabilities given. Then, using the recursive formula, the remaining entries are calculated.

OPTIMAL BINARY SEARCH TREE(Contd..)

$$C[1,2] = \min \begin{cases} C[1,0] + C[2,2] + p_1 + p_2 & \text{when } k = 1 \\ C[1,1] + C[3,2] + p_1 + p_2 & \text{when } k = 2 \end{cases}$$

$$C[2,3] = \min \begin{cases} C[2,1] + C[3,3] + p_1 + p_3 & \text{when } k = 2 \\ C[2,2] + C[4,3] + p_1 + p_3 & \text{when } k = 3 \end{cases}$$

$$C[3,4] = \min \begin{cases} C[3,2] + C[4,4] + p_3 + p_4 & \text{when } k = 3 \\ C[3,3] + C[5,4] + p_3 + p_4 & \text{when } k = 4 \end{cases}$$

The updated entries are shown below in Table 2.

OPTIMAL BINARY SEARCH TREE(Contd..)

Table 2: Updated table

	0	1	2	3	4
1	0	2/7	4/7		
2		0	1/7	6/7	
3			0	3/7	5/7
4				0	1/7
5					0

Similarly, the other entries are obtained as follows:

OPTIMAL BINARY SEARCH TREE(Contd..)

$$C[1,3] = \min \begin{cases} C[1,0] + C[2,3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 1 \\ C[1,1] + C[3,3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 2 \\ C[1,2] + C[4,3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 3 \end{cases}$$

$$= \min\{{}^{12}/_{7}, {}^{11}/_{7}, {}^{10}/_{7}\} = {}^{10}/_{7}$$

$$C[2,4] = \min \begin{cases} C[2,3] + C[5,4] + p_2 + p_3 + p_4, \text{when } k = 2 \\ C[2,2] + C[4,4] + p_2 + p_3 + p_4, \text{when } k = 3 \\ C[2,3] + C[5,4] + p_2 + p_3 + p_4 \text{ when } k = 4 \end{cases}$$

$$= \min\{{}^{11}/_{7}, {}^{7}/_{7}, {}^{11}/_{7}\} = {}^{7}/_{7}$$

OPTIMAL BINARY SEARCH TREE(Contd..)

The updated table is given in Table 3.

	0	1	2	3	4
1	0	2/7	4/7	10/7	
2		0	1/7	6/7	7/7
3			0	3/7	5/7
4				0	1/7
5					0

Table 3: Updated table

The procedure is continued as

OPTIMAL BINARY SEARCH TREE(Contd..)

$$C[1,4] = \min \begin{cases} C[1,0] + C[2,4] + P_1 + P_2 + P_3 + P_4, \text{ when } k = 1 \\ C[1,1] + C[3,4] + P_1 + P_2 + P_3 + P_4, \text{ when } k = 2 \\ C[1,2] + C[4,4] + P1 + P_2 + P_3 + P_4, \text{ when } k = 3 \\ C[1,3] + C[4,4] + P_1 + P_2 + P_3 + P_4\} \text{when } k = 4 \end{cases}$$

$$= \min\{{}^{14}/_{7}, {}^{14}/_{7}, {}^{12}/_{7}, {}^{18}/_{7}\} = {}^{12}/_{7}$$

The updated final table is given as shown in Table 4.

	0	1	2	3	4
1	0	2/7	4/7	10/7	12/7
2		0	1/7	6/7	7/7
3			0	3/7	5/7
4				0	1/7
5					0

Table 4: Final Table

OPTIMAL BINARY SEARCH TREE(Contd..)

- It can be observed that minimum cost is 12/7. What about the tree structure? This can be reconstructed by noting the minimum k in another table as shown in Table 5

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

Table 4: Minimum k

- It can be seen from the table 5 that C(1,4) is 3. So the item 3 is root of the tree. Continuing this fashion, one can find the binary search tree as shown in Fig. 5.

OPTIMAL BINARY SEARCH TREE(Contd..)

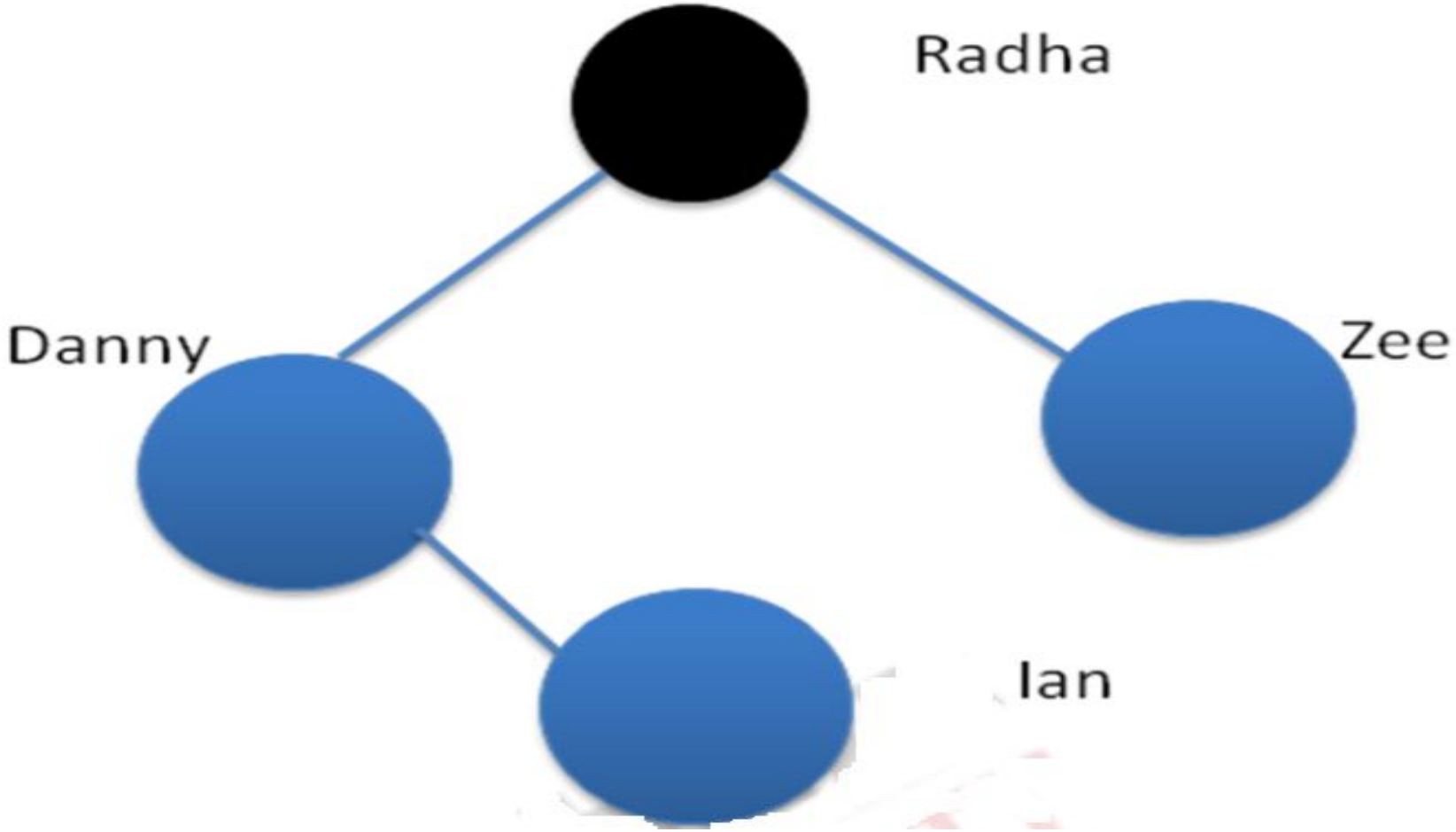


Fig. 5: Constructed Optimal BST

It can be seen that the constructed binary tree is optimal and balanced. The formal algorithm for constructing optimal BST is given as follows:

ALGORITHM : OPTIMAL BINARY SEARCH TREE

```
%% Initialize the table for C
for i = 1 to n do
    C[i, i - 1] = 0
    C[i, i] = pi
End for
C[n + 1,n]= 0
```

```
%% Initialize the table for R
for i = 1 to n do
    R[i, i - 1] = 0
    R[i, i] = i
End for
C[n + 1, n] = 0
```

ALGORITHM : OPTIMAL BINARY SEARCH TREE

```
for diag = 1 to n - 1 do
    for i = 1 to n - diag do
        j = i + diag
```

$$C[i,j] = \min_{i<k\leq j} C[1 \ldots k-1] + C[k+1 \ldots j] + \sum_{m=1}^{n} p_m$$

```
        R[i,j] = k
    End for
End for
Return C[1,n]
```

Complexity Analysis :

The time efficiency is Θ (n 3) but can be reduced to Θ(n 2) by taking advantage of monotonic property of the entries. The monotonic property is that the entry R[i,j] is always in the range between R[i,j-1] and R[i+1,j]. The space complexity is Θ(n 2) as the algorithm is reduced to filling the table.

SUMMARY

- Dynamic Programming is an effective technique.
- DP can solve LCS problem effectively.
- Edit Distance Problem can be solved effectively by DP approach.

HOME ASSIGNMENT:

Construct the Optimal Binary Search Tree for the given values

Node No :	0	1	2	3
Nodes :	10	12	16	21
Freq :	4	2	6	3

References

1. Thomas H Cormen, Charles E Leiserson, Ronald L Revest, Clifford Stein, Introduction to Algorithms, 3rd ed., The MIT Press Cambridge, 2014.
2. https://www.ics.uci.edu/~goodrich/teach/cs260P/notes/LCS.pdf
3. https://www.youtube.com/watch?v=sSno9rV8Rhg
4. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2006
5. Ellis Horowitz, Sartajsahni, Sanguthevar, Rajesekaran, Fundamentals of Computer Algorithms, Galgotia Publication, 2010
6. Anany Levitin, "Introduction to the Design and Analysis of Algorithms", Third Edition, Pearson Education, 2012.
7. S.Sridhar , Design and Analysis of Algorithms , Oxford University Press, 2014.
8. A.Levitin, Introduction to the Design and Analysis of Algorithms, Pearson Education, New Delhi, 2012.
9. T.H.Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA 1992.
10. www.javatpoint.com