Greedy – General Method

- The greedy approach suggests
  - constructing a solution through a sequence of steps
  - each expanding a partially constructed solution obtained so far,
  - Until a complete solution to the problem is reached.

  On each step the choice made must be:
  - ***feasible***, i.e., it has to satisfy the problem's constraints
  - ***locally optimal***, i.e., it has to be the best local choice among all feasible choices available on that step
  - ***irrevocable***, i.e., once made, it cannot be changed on subsequent steps of the algorithm

General method

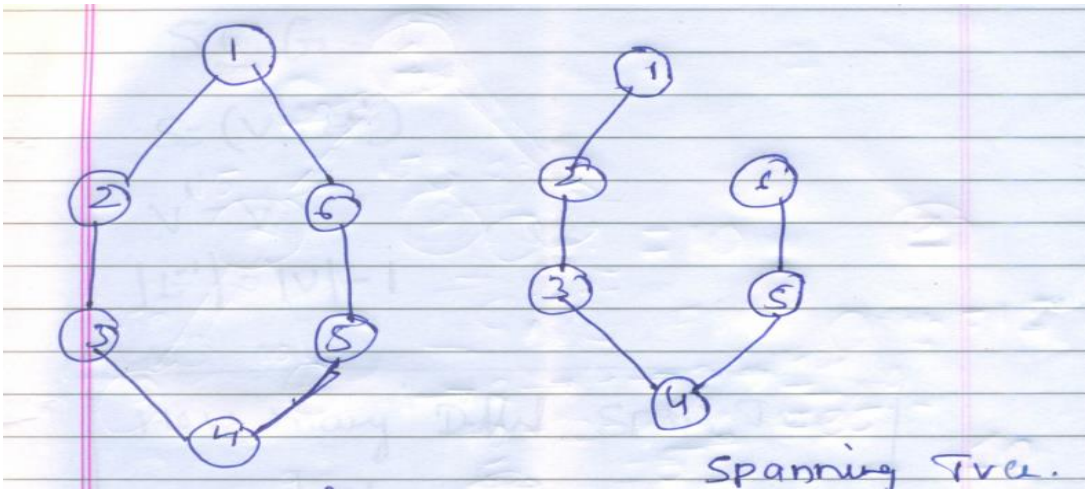
- Given n inputs choose a subset that satisfies some constraints.
- A subset that satisfies the constraints is called a feasible solution.
- A feasible solution that maximizes or minimizes a given (objective) function is said to be optimal.
- Often it is easy to find a feasible solution but difficult to find the optimal solution.

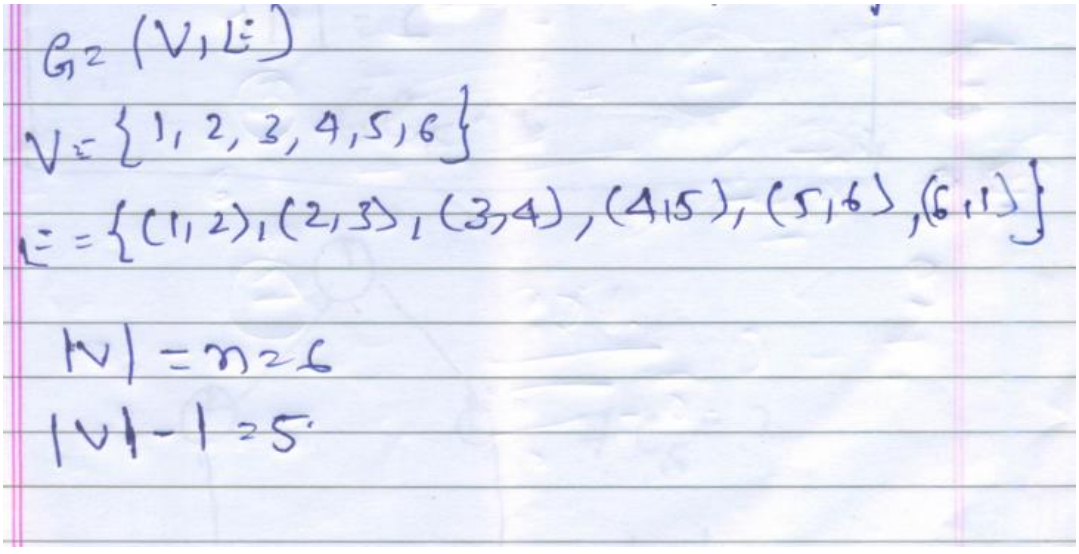
Greedy Method

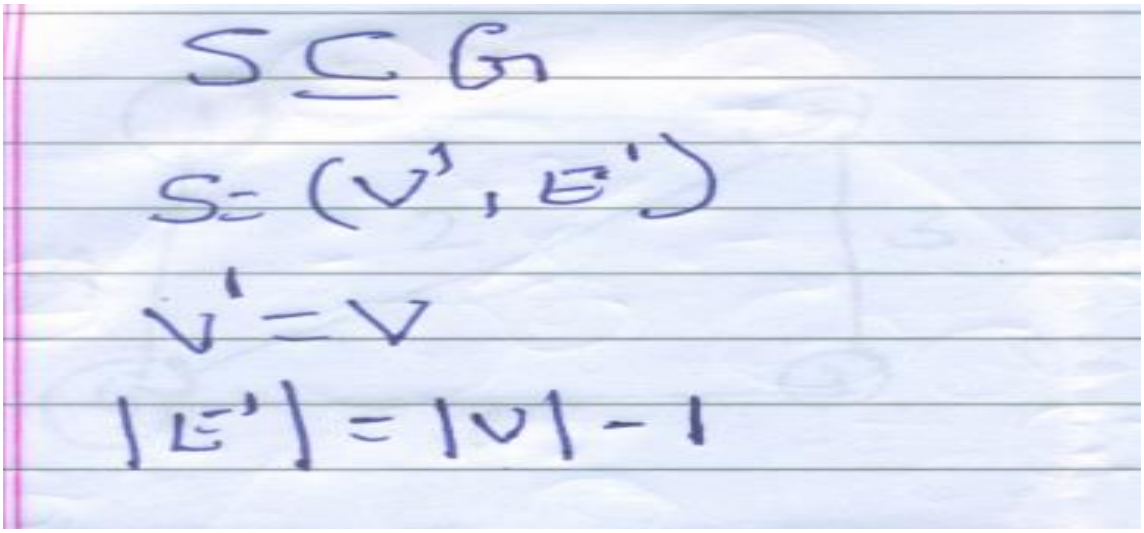
```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
     solution := ∅; // Initialize the solution.
     for i := 1 to n do
     {
          x := Select(a);
          if Feasible(solution, x) then
               solution := Union(solution, x);
     }
     return solution;
}
```

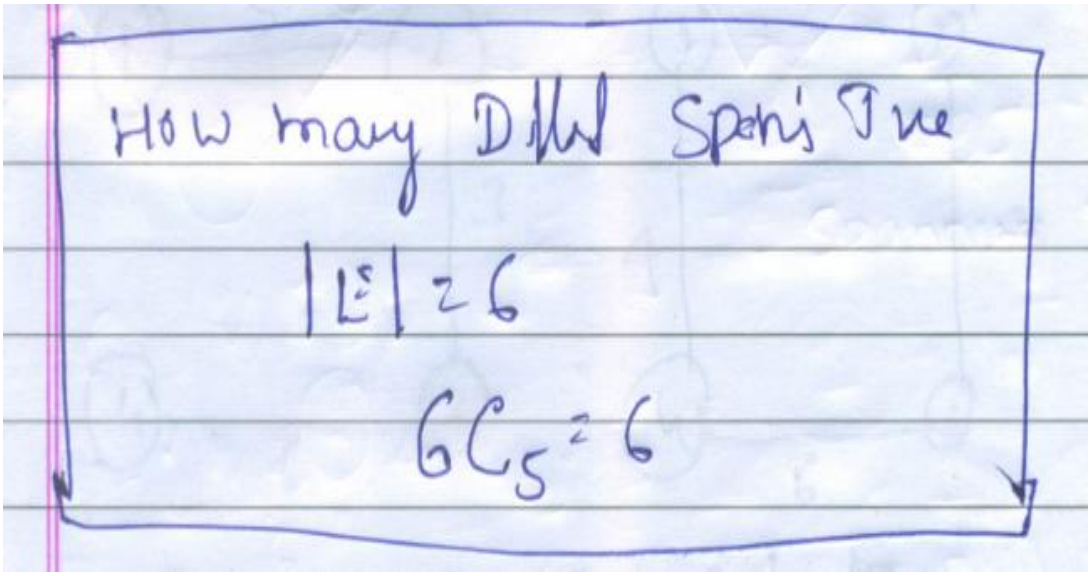
Minimum Spanning TreeMST

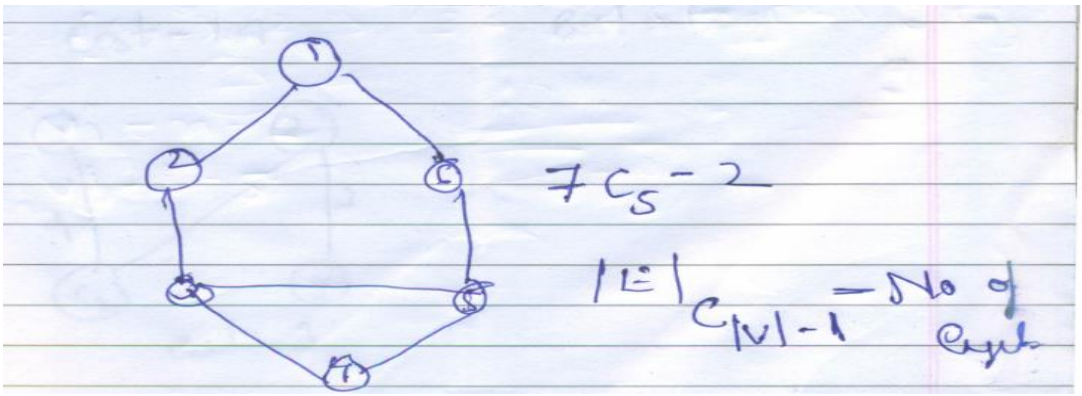
- A **spanning tree** of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.
- A **minimum spanning tree** of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the **weights**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






S ⊆ G
S = (V', E')
V' = V
|E'| = |V| - 1


How many Diff Spans Tree
|E| = 6
6C5 = 6


7C5 = 2
|E| C|V|-1 = No of

MST – Example

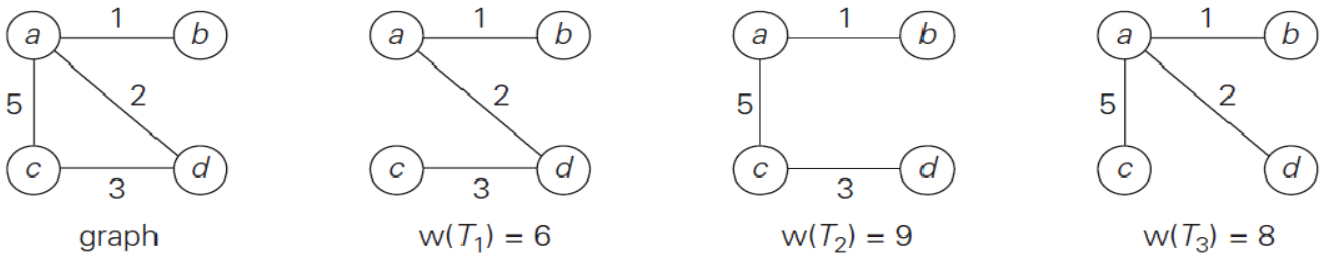
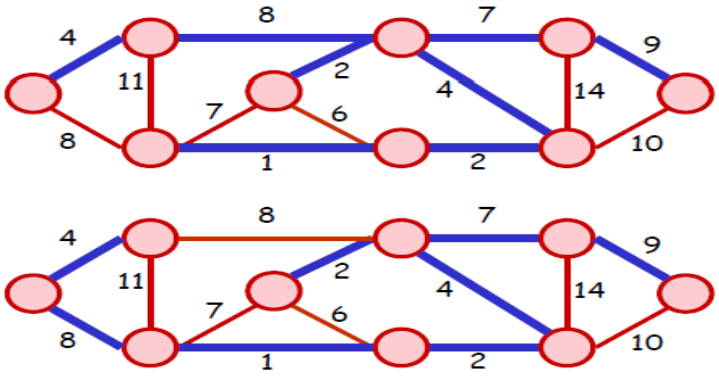
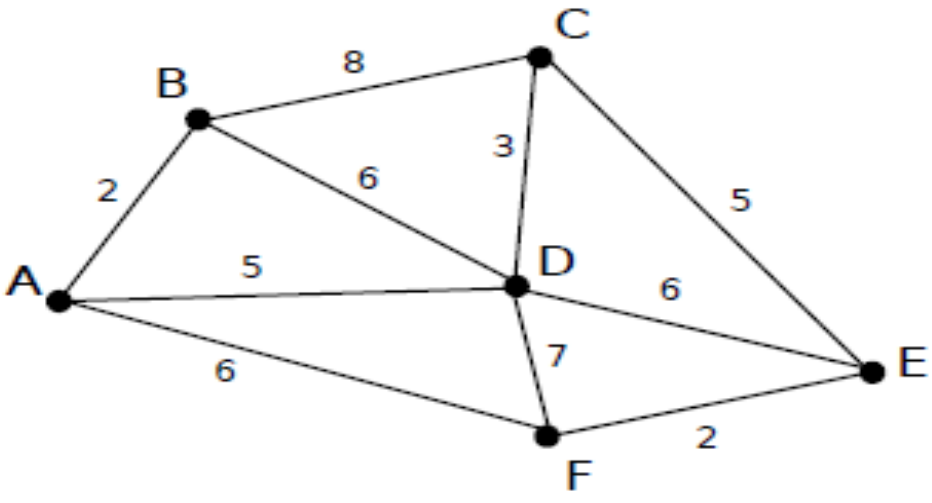


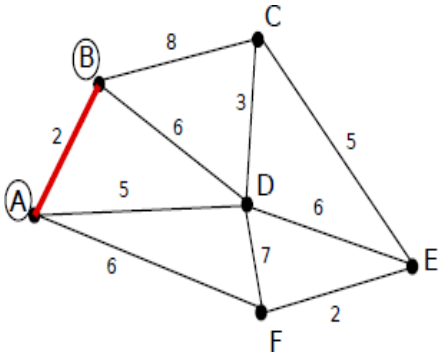
FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

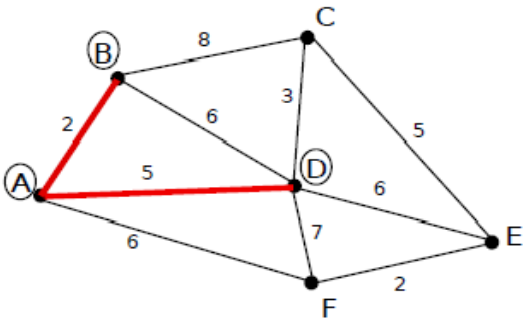
MST of a graph may not be unique

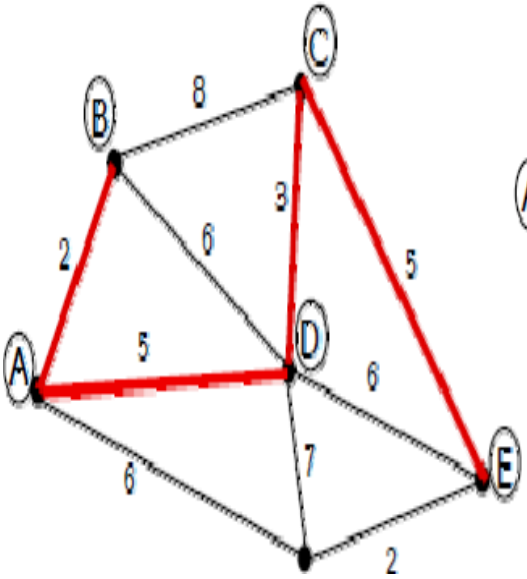


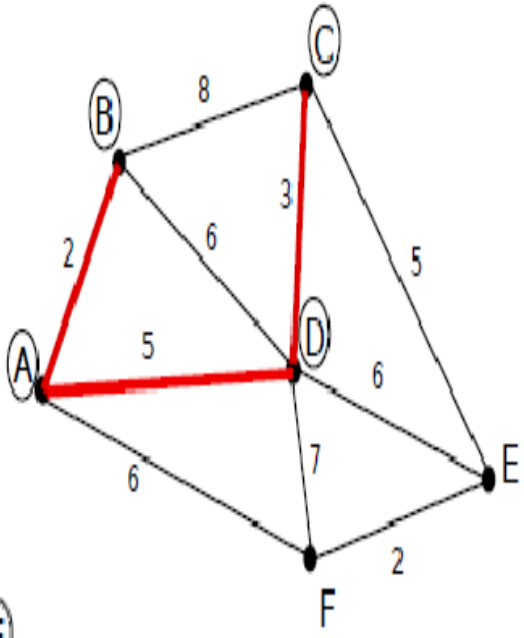
Prims Algorithm-Example

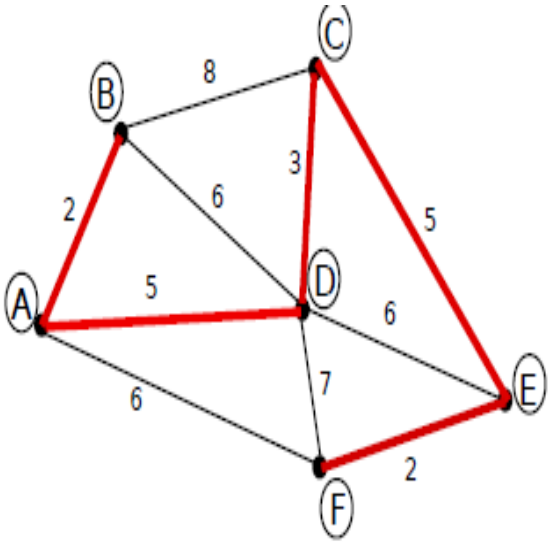



C
B
8
3
2
6
5
A
5
D
6
7
E
6
2
F

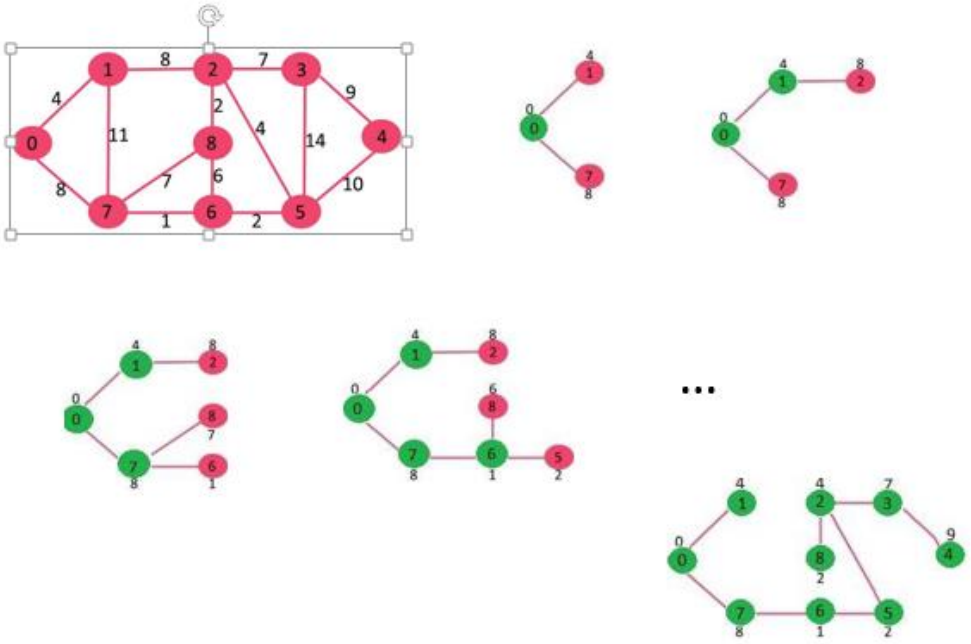

C
B
8
3
2
6
5
A
5
D
6
7
E
6
2
F

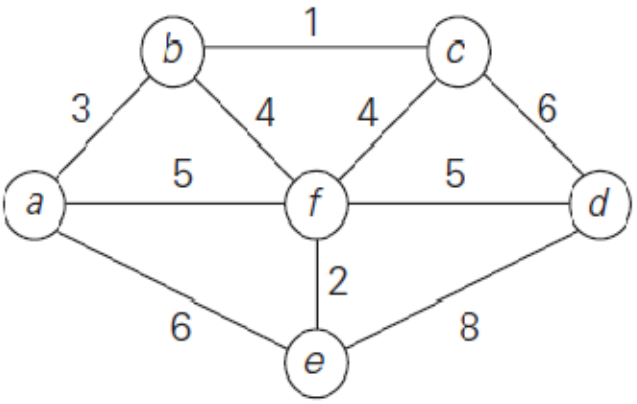

C
B
8
3
2
6
5
A
5
D
6
7
E
6
2

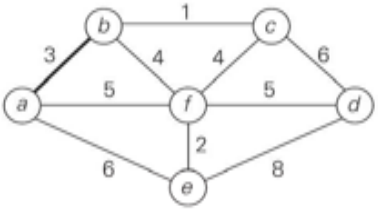
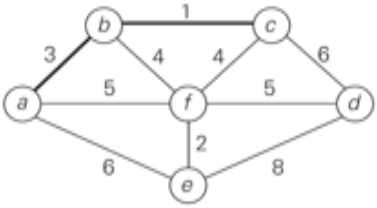
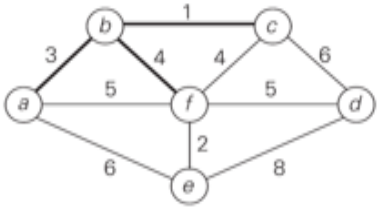
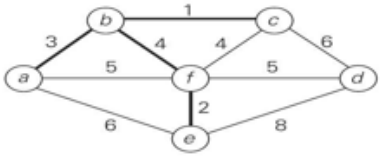
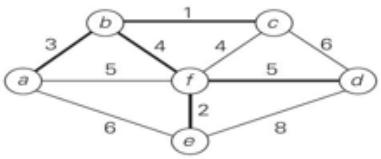

C
B
8
3
2
6
5
A
5
D
6
7
E
6
2
F



Prim's Algorithm - Example





Tree vertices	*Remaining vertices*	*Illustration*
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Prim's Algorithm

ALGORITHM *Prim(G)*

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

Analysis of Efficiency

- Depends on the data structures chosen
  - for the **graph** & for the **priority queue** of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.
- If graph representation - **weight matrix,**
- priority queue - is implemented as an **unordered array**,

  the running time will be in $\Theta(|V|^2)$.
- On each of the $|V|-1$ iterations,
  - the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.
- We can implement the priority queue as a **min-heap**.
  - A min-heap is a complete binary tree in which every element is less than or equal to its children.
  - Deletion of the smallest element from and insertion of a new element into a min-heap of size n are **O(log n)** operations.
- If a graph is represented as **adjacency lists** and the priority queue is implemented as a **min-heap**,

- the running time is in **O(|E| log |V|).**

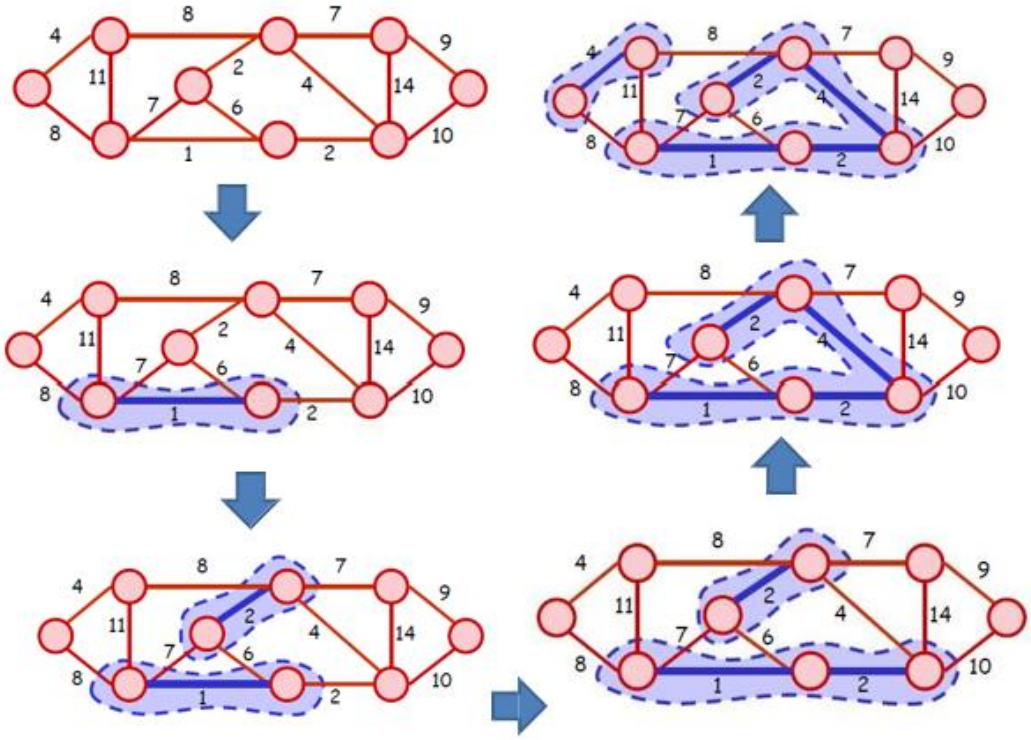
Why running time is in **O(|E| log |V|) ?**

- Algorithm performs |V|−1 deletions of the smallest element and makes |E| verifications
  - and, possibly, changes of an element's priority in a min-heap of size not exceeding |V|.
  - Each of these operations, is a O(log |V|) operation.
- Hence, the running time of this implementation of Prim's algorithm is in
  = (|V| − 1+ |E|) O (log |V |)
  = O(|E| log |V |) because, in a connected graph, |V|− 1≤ |E|.

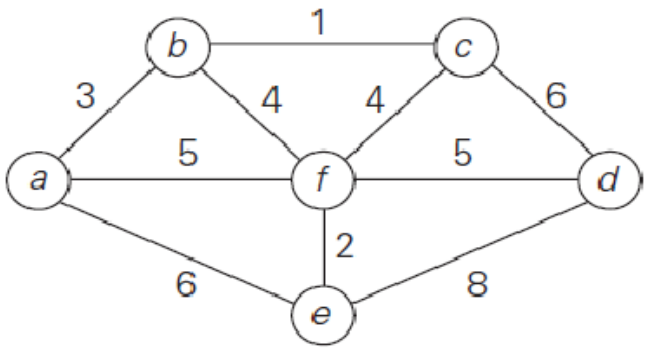
Kruskals Algorithm

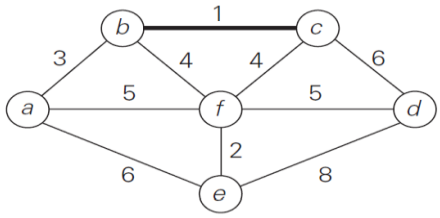
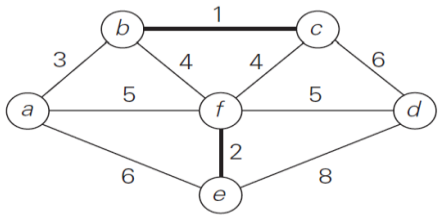
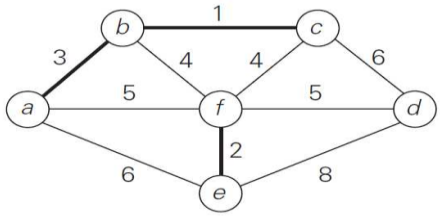
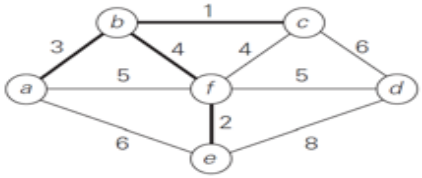
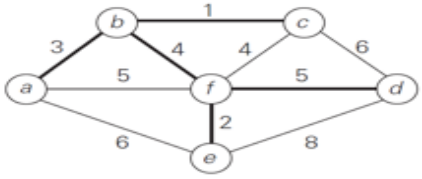
Working

- **Sorting** the graph's edges in **non decreasing** order of their **weights**.
- Then, starting with the empty sub graph,
  - it scans this sorted list **adding the next edge** on the list to the current sub graph if such an inclusion does not create a **cycle** and simply **skipping** the edge otherwise.



Kruskal Algorithm



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

Kruskal's MST Algorithm

ALGORITHM *Kruskal(G)*

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

Analysis

- The crucial check whether two vertices belong to the same tree can be found out using **union-find algorithms.**
- Efficiency of Kruskal's algorithm is based on the time needed for **sorting the edge weights** of a given graph.
- With an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in **O (|E| log |E|).**

Single Source Shortest Path

Problem Statement

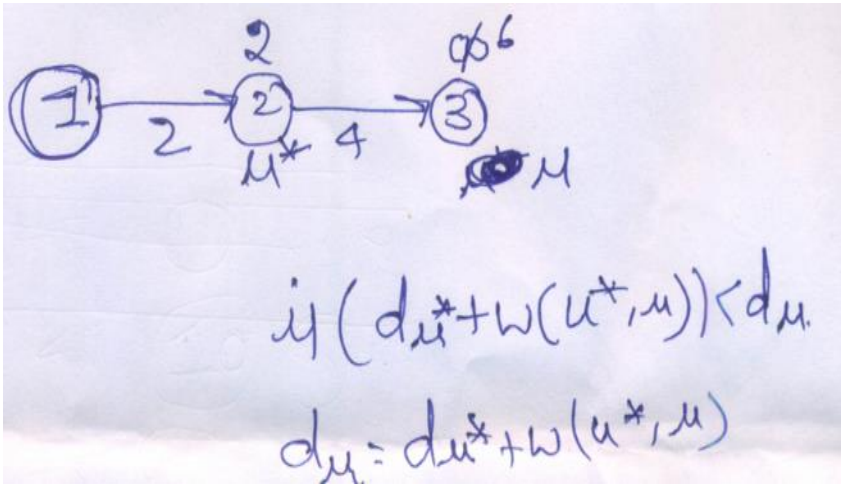
- Given a graph and a source vertex in graph,
- find shortest paths from **source** to **all vertices** in the given graph
- **Dijkstra's** Algorithm is the best-known algorithm

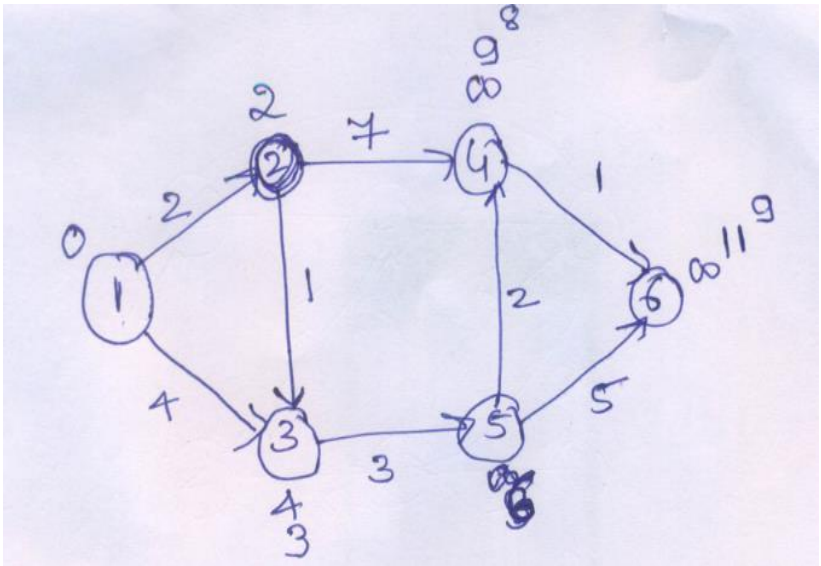
- It is similar to Prim's algorithm
- This algorithm is applicable to undirected and directed graphs with nonnegative weights only.

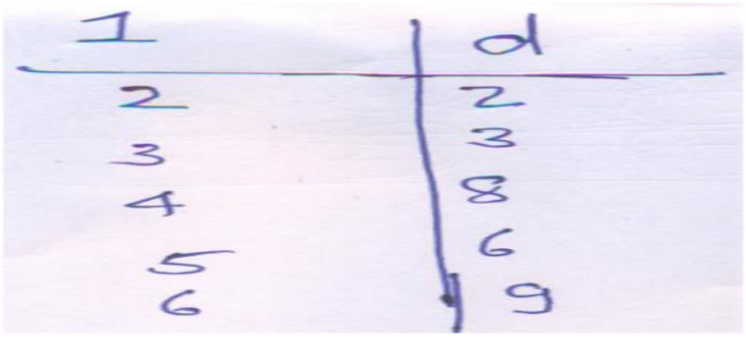
Dijkstra's Algorithm

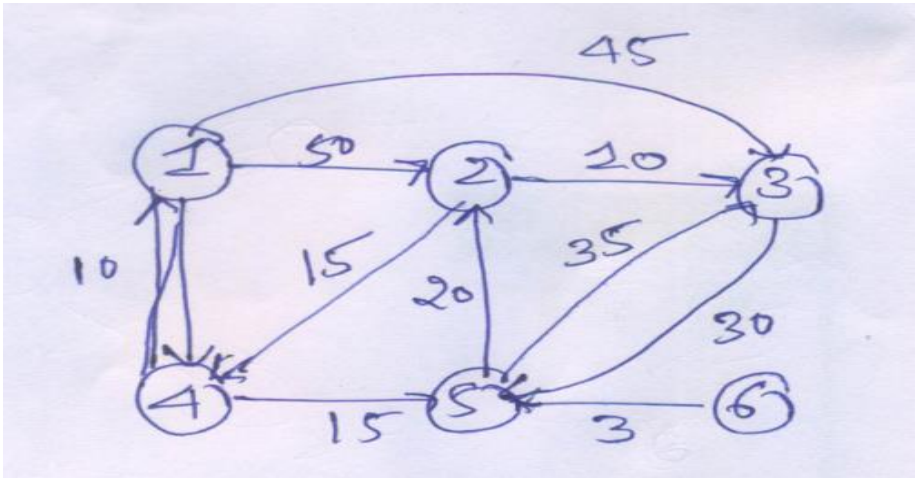
Working

- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on
- In general, before its i^{th} iteration commences, the algorithm has already identified the shortest paths to i-1 other vertices nearest to the source.

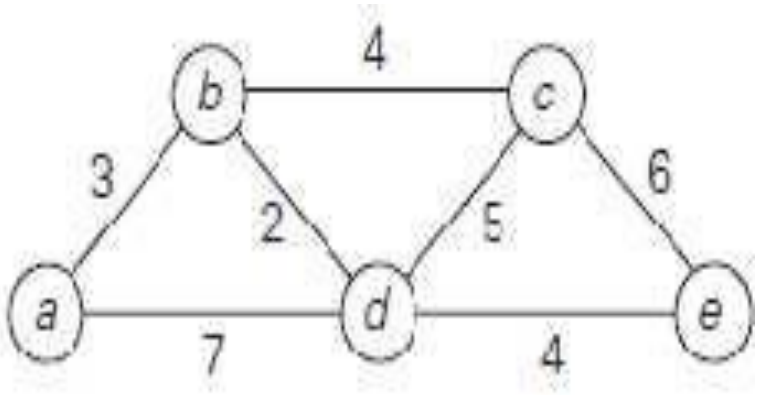


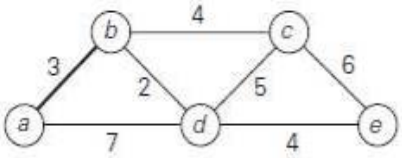
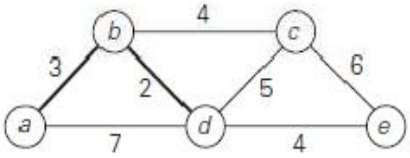
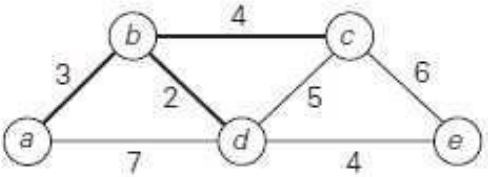
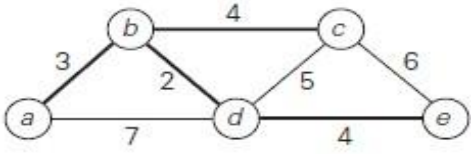



1
d
2
3
4
5
6
2
3
8
6
9


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


Selected vale
2
3
4
5
6
4
50
45
10
∞
∞
5
50
45
20
25
∞
2
45
45
20
25
∞
3
45
45
10
25
∞
6.
45
45
10
25
∞



Tree vertices	Remaining vertices	Illustration
a(–, 0)	**b(a, 3)** c(–, ∞) d(a, 7) e(–, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(–, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b : $a - b$ of length 3

from a to d : $a - b - d$ of length 5

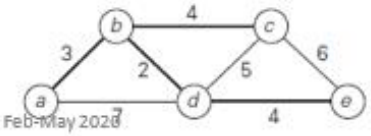
from a to c : $a - b - c$ of length 7

from a to e : $a - b - d - e$ of length 9

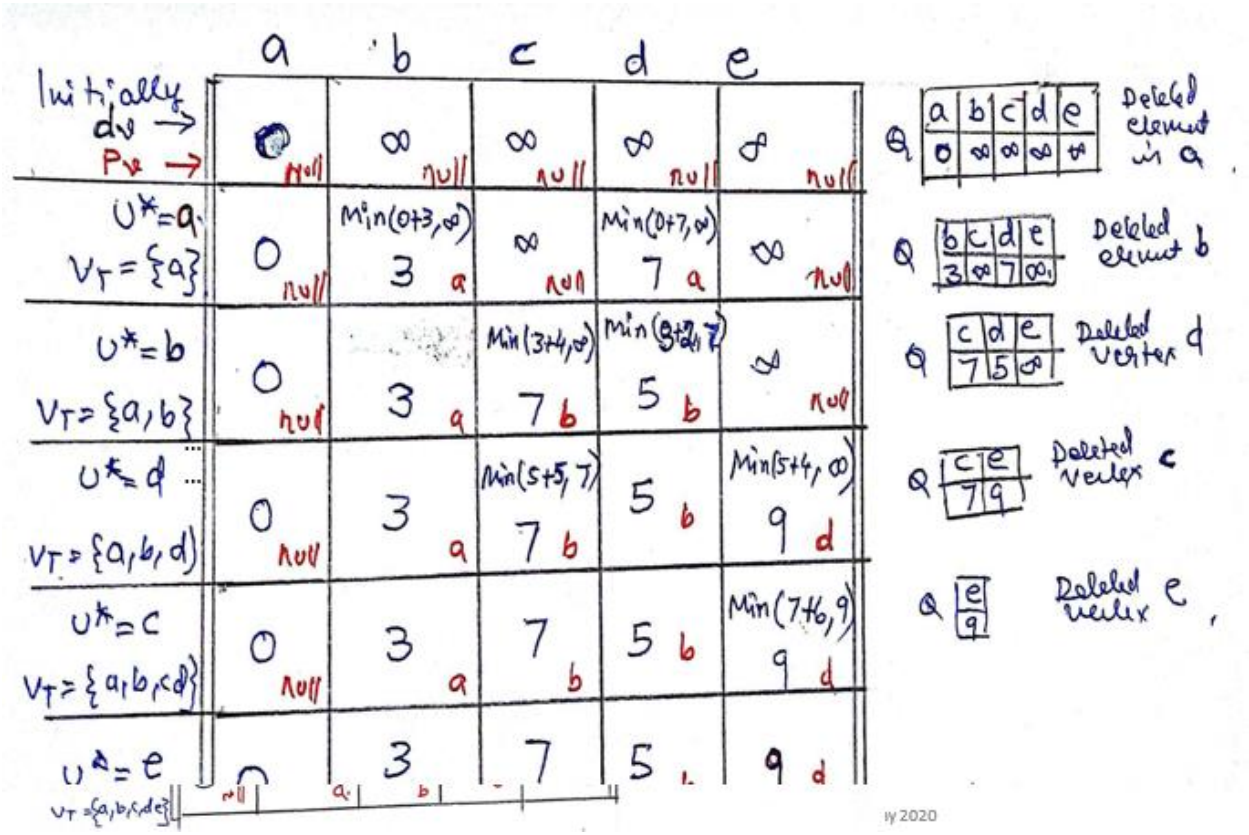
Dijkstra's Algorithm

ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
    $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
    Insert(Q, v, d_v) //initialize vertex priority in the priority queue
$d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \varnothing$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
    $u^* \leftarrow$ *DeleteMin*(Q) //delete the minimum priority element
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in $V - V_T$ that is adjacent to u^* **do**
        if $d_{u^*} + w(u^*, u) < d_u$
            $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
            Decrease(Q, u, d_u)



Feb-May 2020



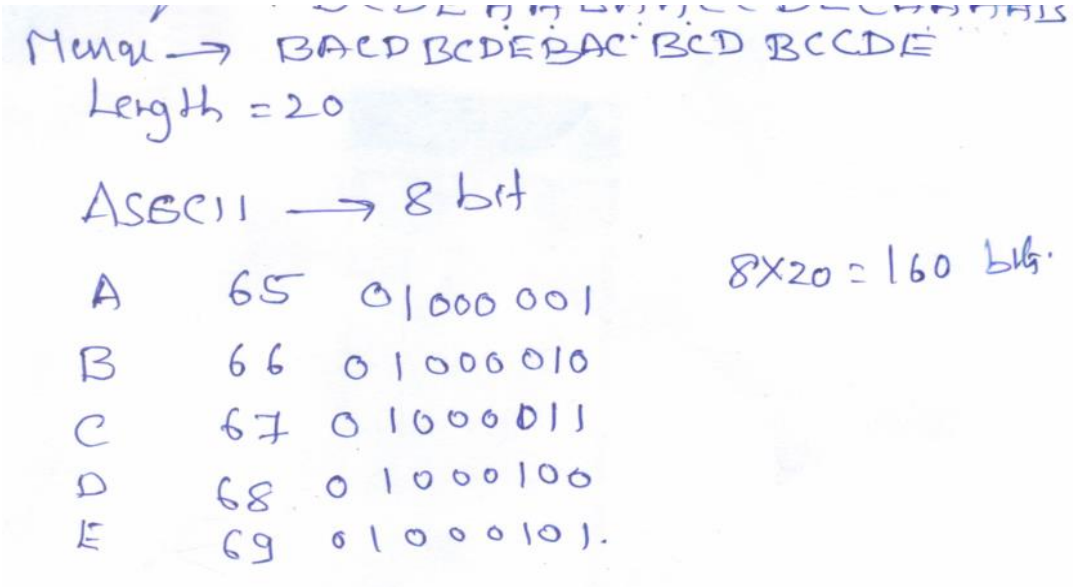
Analysis

- Efficiency is $\Theta(|V|^2)$ for graphs represented by their ***weight matrix*** and the priority queue implemented as an ***unordered array***.
- For graphs represented by their **adjacency lists** and the priority queue implemented as a **min-heap**, it is in **O (|E| log |V|)**
-

Optimal Tree Problem

Background

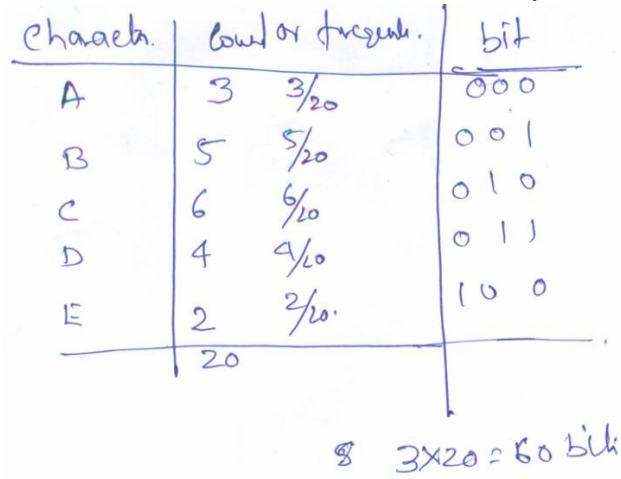
- Suppose we have to encode a text that comprises characters from some n-character alphabet
- Encode by assigning to each of the text's characters some sequence of bits called the **code word.**
- There are two types of encoding: Fixed-length encoding, Variable-length encoding

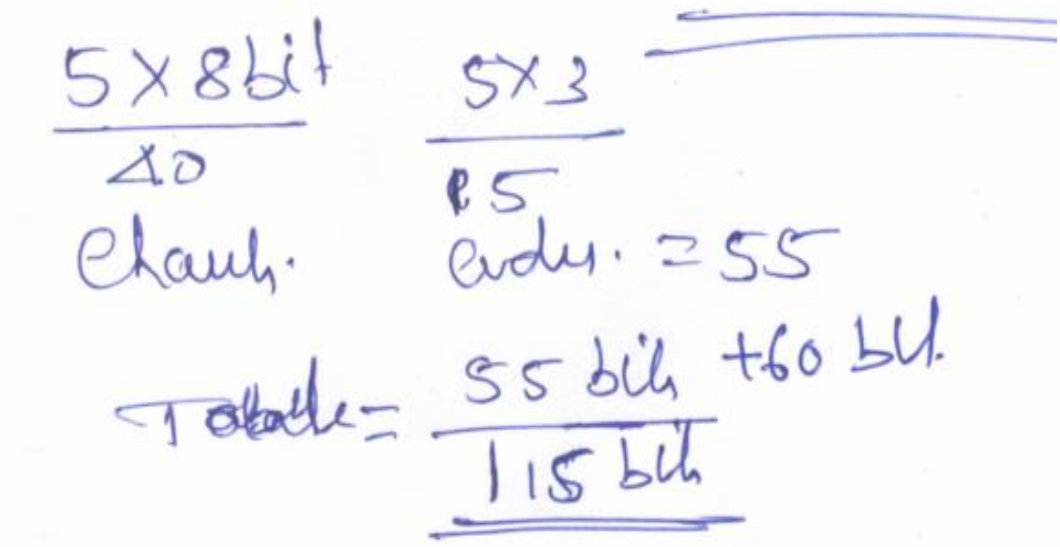


Optimal Tree Problem

Fixed length Coding

- This method assigns to each character a bit string of the same length ***m***.
- Example: ASCII code.
- Drawback?
  - How to overcome the drawback?
  - Assigning shorter code-words to more frequent characters and longer code-words to less frequent characters.





Variable length Coding

- This method assigns code-words of different lengths to different characters
- Prefix-free codes (prefix codes) are used. Here no code word is a prefix of a code word of another character.
- We can simply scan a bit string until we get the first group of bits that is a code word for some character

Variable length Coding

- If we want to create a binary prefix code for some alphabet,
  - it is natural to associate the alphabet's characters with leaves of a binary tree in which all the left edges are labelled by 0 and all the right edges are labelled by 1 (or vice versa).
- Many trees that can be constructed in this manner for a given alphabet

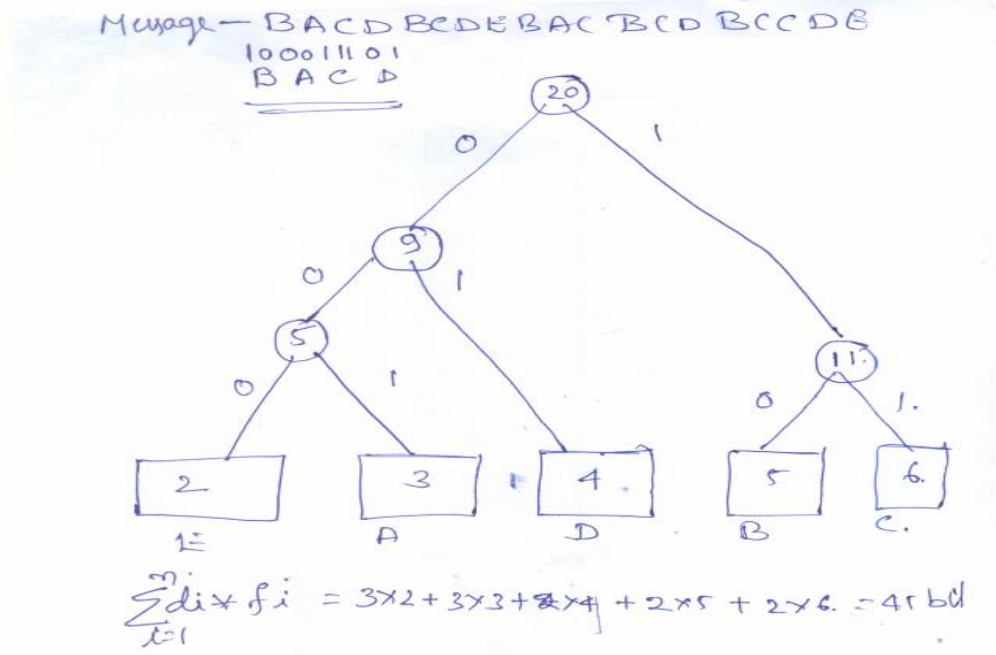
Huffman Trees and Codes

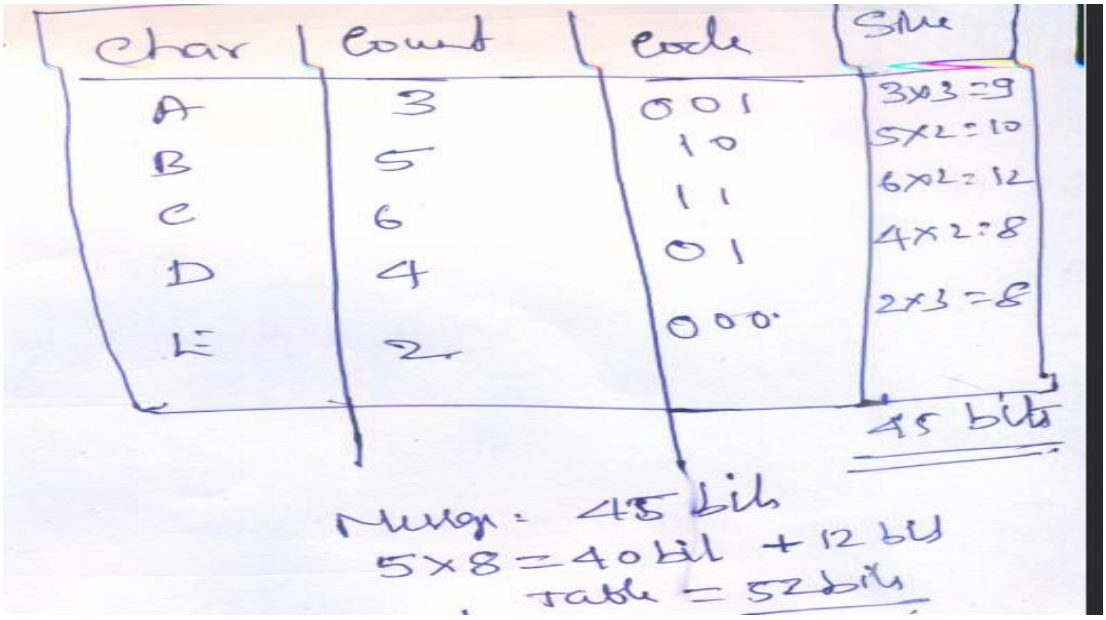
Huffman's Algorithm

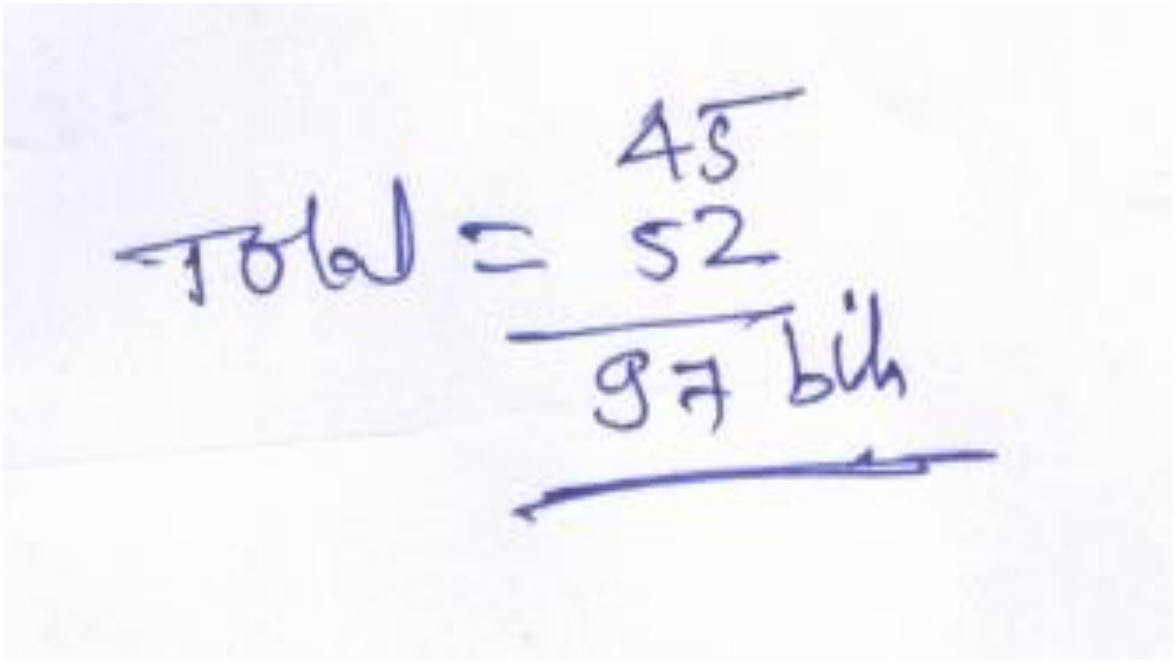
- ***Step 1****:* Initialize *n* one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's *weight.*
- ***Step 2****:* Repeat the following operation until a single tree is obtained.

– Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

- A tree constructed by the above algorithm is called **a *Huffman tree***. It defines-in the manner described-a ***Huffman code***.

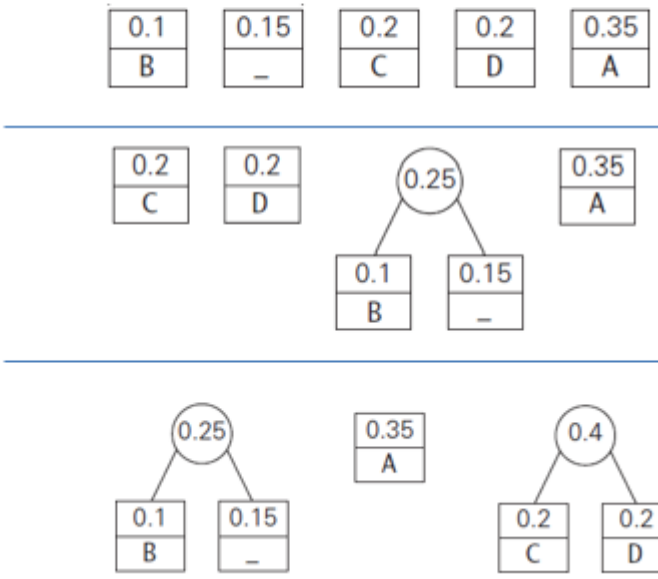


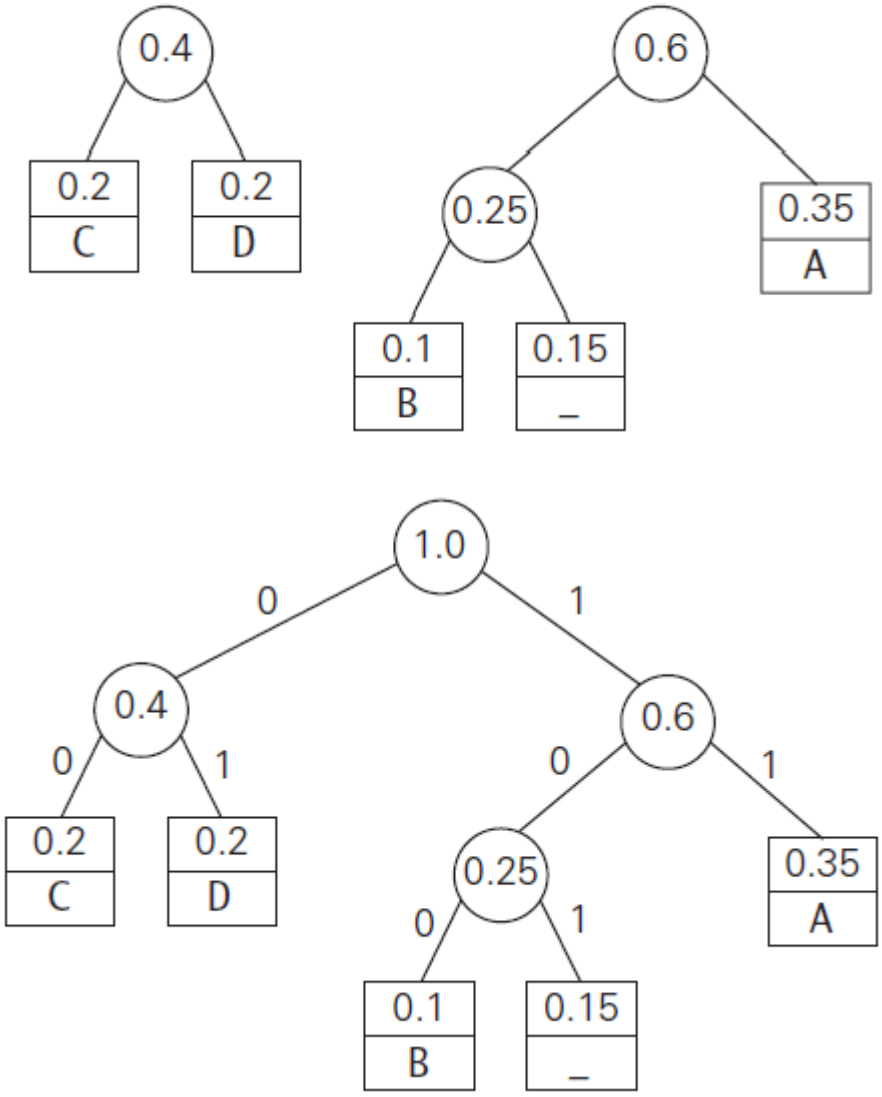




Example

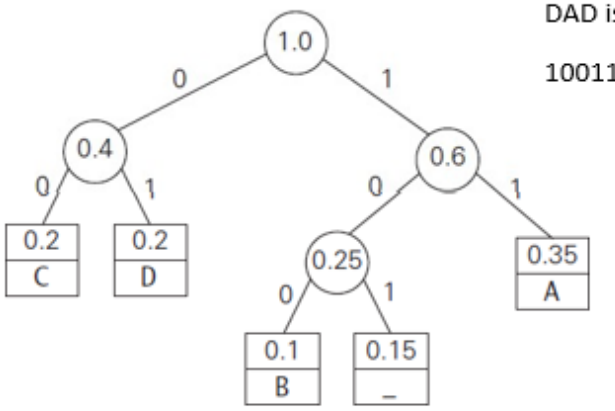
symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15




0.4
0.2
C
0.2
D
0.6
0.25
0.35
A
0.1
B
0.15
_
1.0
0
1
0.4
0.6
0
1
0
1
0.2
C
0.2
D
0.25
0.35
A
0
1
0.1
B
0.15
_

Huffman codes

- The resulting codewords are as follows:



DAD is encoded as

10011011011101 is decoded as

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Analysis

- Average number of bits per symbol in this code

  $2 * 0.35 + 3 * 0.1 + 2 * 0.2 + 2 * 0.2 + 3 * 0.15 = 2.25.$
- Compression ratio$(3-2.25)/3*100 = 25\%$
  In other words, Huffman's encoding of the above text will use 25% less memory than its fixed-length encoding.

DECISION TREE

A decision tree is a type of supervised learning algorithm that is commonly used in machine learning to model and predict outcomes based on input data. It is a tree-like structure where each internal node tests on attribute, each branch corresponds to attribute value and each leaf node represents the final decision or prediction.

Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., k < k'. The node's left subtree contains the information about subsequent comparisons made if k < k', while its right subtree does the same forthe case of k > k'. (For the sake of simplicity, we assume throughout this section that all input items are distinct.) Each leaf represents a possible outcome of the algorithm's run on some input of size n.

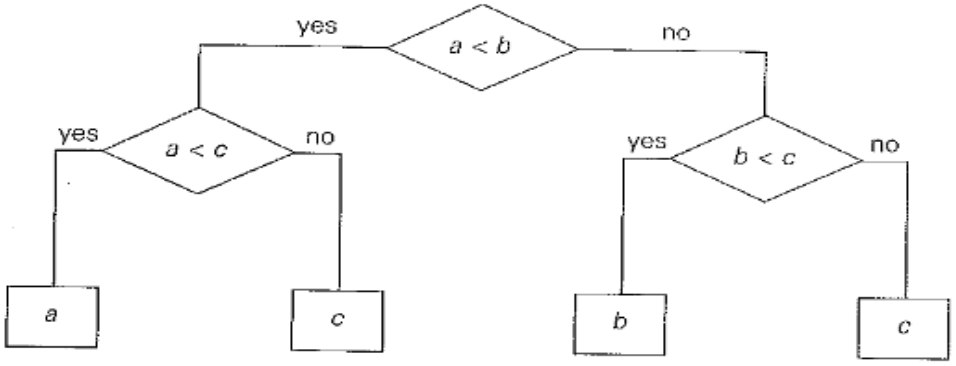


FIGURE 11.1 Decision tree for finding a minimum of three numbers

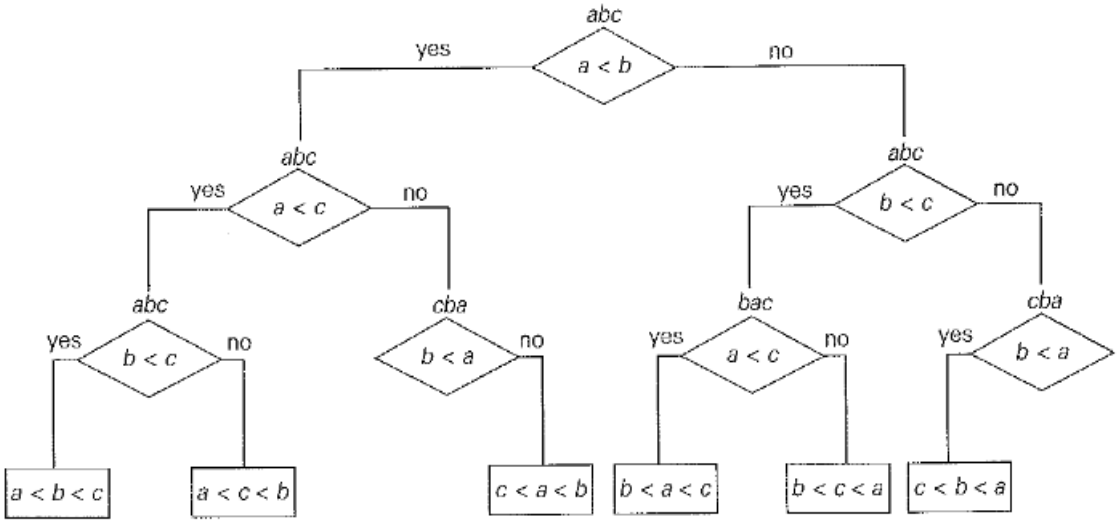


FIGURE 11.2 Decision tree for the three-element selection sort. A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons $b < a$ with a single possible outcome because of the results of some previously made comparisons.

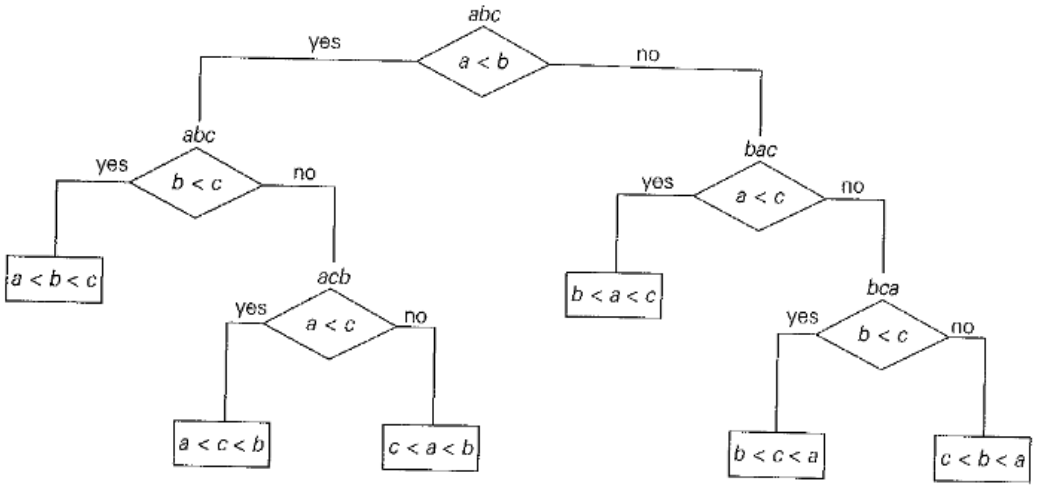


FIGURE 11.3 Decision tree for the three-element insertion sort

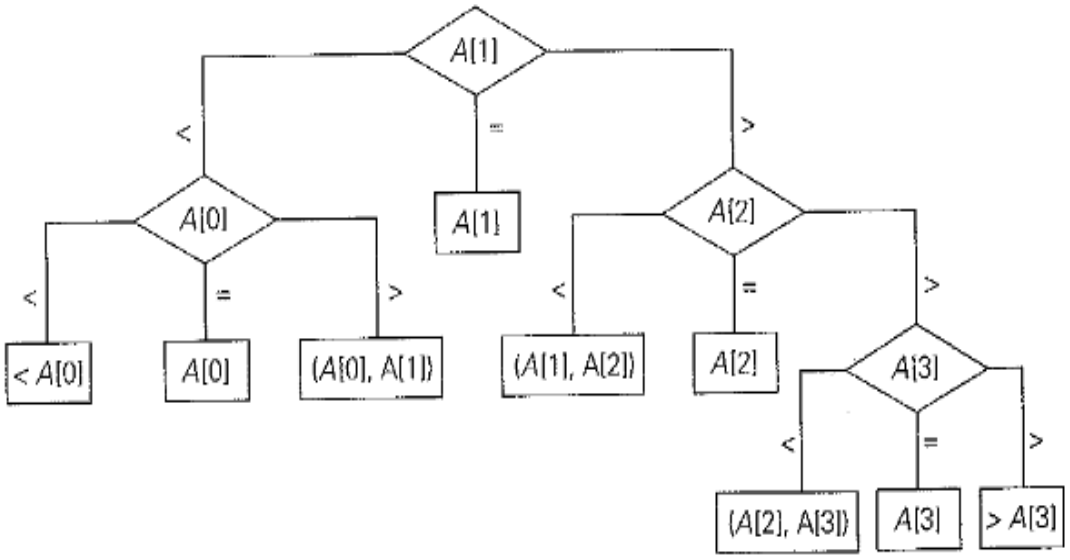


FIGURE 11.4 Ternary decision tree for binary search in a four-element array

LOWER BOUND ARGUMENTS

Given a class of algorithms for solving a particular problem, a lower bound indicates the best possible efficiency any algorithm from this class can have.For example, selection sort, whose efficiency is quadratic, is a reasonably fast algorithm, whereas the algorithm for the Tower of Hanoi problem is very slow because its efficiency is exponential.

The alternative and possibly "fairer" approach is to ask how efficient a particular algorithm is with respect to other algorithms for the same problem. Seen in this light, selection sort has to be considered slow because there are O (n log n) sorting algorithms. The Tower of Hanoi algorithm, on the other hand, turns out to be the fastest possible for the problem it solves.

Trivial lower Bounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem's input that must be processed and the number of output items that need to be produced.

Since any algorithm must at least "read" all the items it needs to process and "write" all its outputs, such a count yields a trivial lower bound.

For example, any algorithm for generating all permutations of n distinct items must be in (n!) because the size of the output is n!. And this bound is tight because good algorithms for generating permutations spend a constant time on each of them except the initial one

As another example, consider the problem of evaluating a polynomial of degree n

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

at a given point x, given its coefficients $a_n, a_{n-1}, \ldots, a_0$. It is easy to see that all the coefficients have to be processed by any polynomial-evaluation algorithm. Indeed, if it were not the case, we could change the value of an unprocessed coefficient, which would change the value of the polynomial at a nonzero point x. This means that any such algorithm must be in $\Omega(n)$. This lower bound is tight because both the right-to-left evaluation algorithm (Problem 2 in Exercises 6.5) and Horner's rule (Section 6.5) are both linear.

Information-Theoretic Arguments

While the approach outlined above takes into account the size of a problem's output, the information-theoretical approach seeks to establish a lower bound based on the amount of information it has to produce.
Consider, as an example,
the well-known game of deducing a positive integer between 1 and n selected by somebody by asking that person questions with yes/no answers.
The amount of uncertainty that any algorithm solving this problem has to resolve can be measured by $[\log_2 n]$ the number of bits needed to specify a particular number among the
n possibilities.
We can think of each question (or, to be more accurate, an answer to each question) as yielding at most one bit of information about the algorithm's output, i.e., the selected number.
Consequently, any such algorithm will need at least [log2 n] such steps before it can determine its output in the worst case.

Adversary Arguments

Let us revisit the same game of "guessing" a number used to introduce the idea of an information-theoretic argument. We can prove that any algorithm that solves this problem must ask at least $[\log_2 n]$ questions in its worst case by playing the role of a hostile adversary who wants to make an algorithm ask as many questions as possible.

The adversary starts by considering each of the numbers between 1 and n as being potentially selected. After each question, the adversary gives an answer that leaves him with the largest set of numbers consistent with this and all the previously given answers. If an algorithm stops before the size of the set is reduced to one, the adversary can exhibit a number that could be a legitimate input the algorithm failed to identify.

This example illustrates the ***adversary method*** for establishing lower bounds. It is based on following the logic of a malevolent but honest adversary: the malevolence makes him push the algorithm down the most time-consuming path, while his honesty forces him to stay consistent with the choices already made. A lower bound is then obtained by measuring the amount of work needed to shrink a set of potential inputs to a single input along the most time-consuming path.

As another example, consider the problem of merging two sorted lists of size n

$$a_1 < a_2 < \ldots < a_n \text{ and } b_1 < b_2 < \ldots < b_n$$

into a single sorted list of size $2n$. For simplicity, we assume that all the a's and b's are distinct, which gives the problem a unique solution. We encountered this problem when discussing mergesort in Section 4.1. Recall that we did merging by repeatedly comparing the first elements in the remaining lists and outputting the smaller among them. The number of key comparisons in the worst case for this algorithm for merging is $2n - 1$.

Problem Reduction

There, we discussed getting an algorithm for problem P by reducing it to another problem Q solvable with a known algorithm. A similar reduction idea can be used for finding a lower bound. To show that problem P is at least as hard as another problem Q with a known lower bound, we need to reduce Q to P (not P to Q).

In other words, we should show that an arbitrary instance of problem Q can be transformed (in a reasonably efficient fashion) to an instance of problem P, so any algorithm solving P would solve Q as well. Then a lower bound for Q will be a lower bound for P.

TABLE 11.1 Problems often used for establishing lower bounds by problem reduction

Problem	Lower bound	Tightness
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness problem	$\Omega(n \log n)$	yes
multiplication of n-digit integers	$\Omega(n)$	unknown
multiplication of square matrices	$\Omega(n^2)$	unknown

P, NP, and NP-complete Problems

DEFINITION 1 We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to O(p(n)) where p(n) is a polynomial of the problem's input size n. Problems that can be solved in polynomial time are called tractable, problems that cannot be solved in polynomial time are called intractable

DEFINITION 2 Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called polynomial

DEFINITION 3 A non-deterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following.

Nondeterministic ("guessing") stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I

Deterministic ("verification") stage: A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I.

DEFINITION 4 Class *NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministic polynomial***.

Most decision problems are in *NP*. First of all, this class includes all the problems in *P*:

$$P \subseteq NP.$$

DEFINITION 5 A decision problem D_1 is said to be ***polynomially reducible*** to a decision problem D_2 if there exists a function t that transforms instances of D_1 to instances of D_2 such that

1. t maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to no instances of D_2;
2. t is computable by a polynomial-time algorithm.

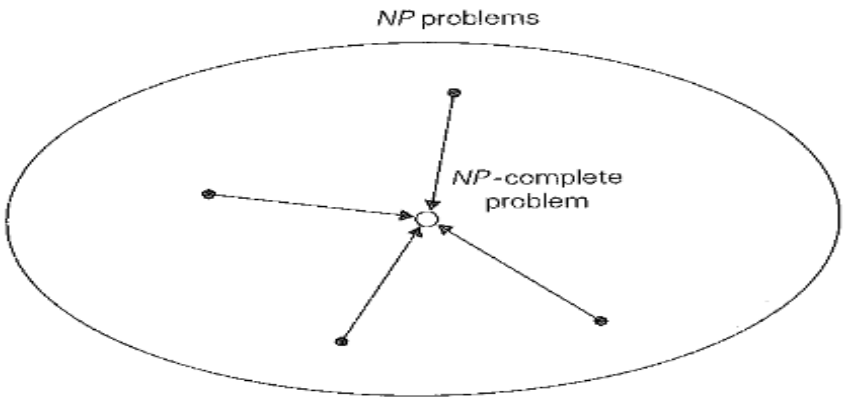


FIGURE 11.6 Notion of an *NP*-complete problem. Polynomial-time reductions of *NP* problems to an *NP*-complete problem are shown by arrows.

DEFINITION 6 A decision problem D is said to be ***NP-complete*** if

1. it belongs to class NP;
2. every problem in NP is polynomially reducible to D.

Challenges of Numerical Algorithms

Numerical analysis is usually described as the branch of computer science concerned with algorithms for solving mathematical problems.

This description needs an important clarification: the problems in question are problems of "continuous" mathematics-solving equations and systems of equations, evaluating such functions as sin x and In x, computing integrals, and so on-as opposed to problems of discrete mathematics dealing with such structures as graphs, trees, permutations, and combinations.

The errors of such approximations are called truncation errors. One of the major tasks in numerical analysis is to estimate the magnitudes of truncation errors

$$e^x \approx 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}. \qquad (11.6)$$

$$|e^x - [1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}]| \leq \frac{M}{(n+1)!}|x|^{n+1}, \qquad (11.8)$$

where $M = \max e^{\xi}$ on the segment with the endpoints at 0 and x. This formula makes it possible to determine the degree of Taylor's polynomial needed to guarantee a predefined accuracy level of approximation (11.6).

The other type of errors, called round-off errors, are caused by the limited accuracy with which we can represent real numbers in a digital computer.

$$\pm .d_1 d_2 \ldots d_p \cdot B^E, \qquad (11.10)$$

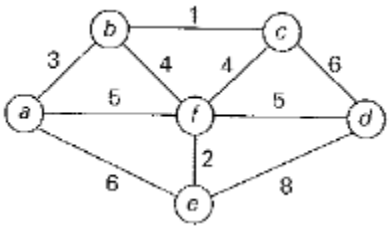
where B is the number base, usually 2 or 16 (or, for unsophisticated calculators, 10); $d_1, d_2, \ldots, d_p$ are digits ($0 \leq d_i < B$ for $i = 1, 2, \ldots, p$ and $d_1 > 0$ unless the number is 0) representing together the fractional part of the number and called its ***mantissa***; E is an integer ***exponent*** with the range of values approximately symmetric about 0.

2 marks Questions

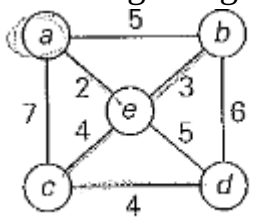
1. What is Greedy problem? List requirements of the solution at each step in greedy approach.
2. Differentiate between Prim's and Kruskal's Algorithm.
3. What is the Prim's algorithm? How it works?
4. What is the approach to solve a problem using Kruskal's algorithm.
5. What is the approach to solve a problem using Dijkstra's algorithm.
6. Write the Complexity of Kruskal's and Prism Algorithms.
7. Define spanning tree and minimum spanning tree.
8. What is dynamic Huffman encoding?
9. Differentiate fixed length encoding and variable length encoding in Huffman tree.
10. Define Huffman tree and Huffman code.
11. What is lower bound Arguments?
12. What are P problems? Write example.
13. What are NP problems? Write example.
14. What are NP Complete problems? Write example.
15. What are Decision Trees? Draw the Decision tree for Maximum of two numbers.

Long Answer Questions **(THREE, FOUR OR FIVE Marks Questions)**

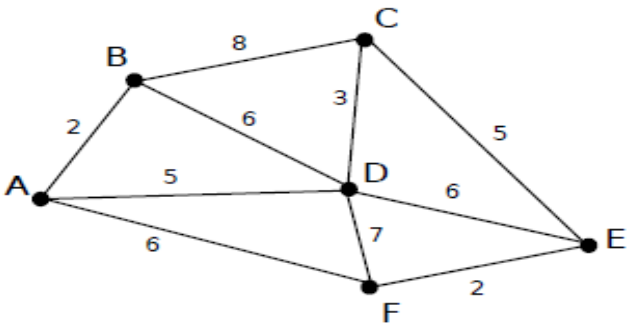
1. Write and explain the Prim's algorithm and find Minimum Spanning tree for the given graph



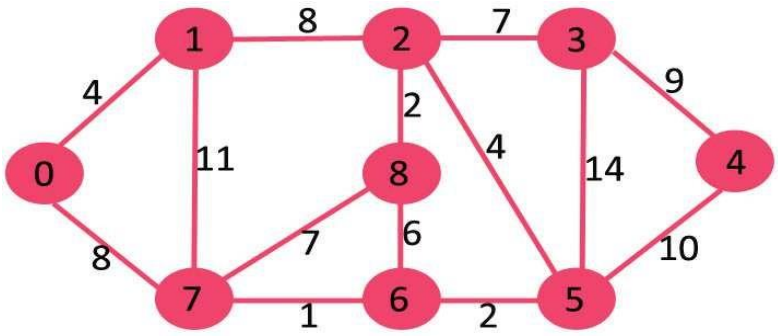
2. Apply Prim's algorithm to the following graph and find Minimum Spanning tree for the given graph



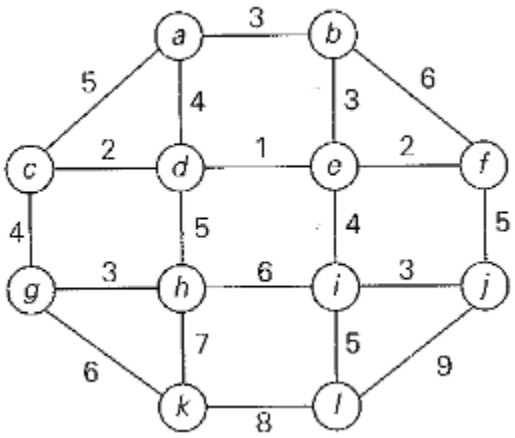
3. Apply Prim's algorithm to the following graph and find Minimum Spanning tree for the given graph



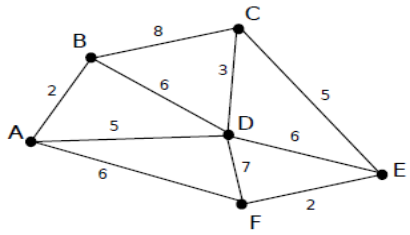
4. Apply Prim's algorithm to the following graph and find Minimum Spanning tree for the given graph



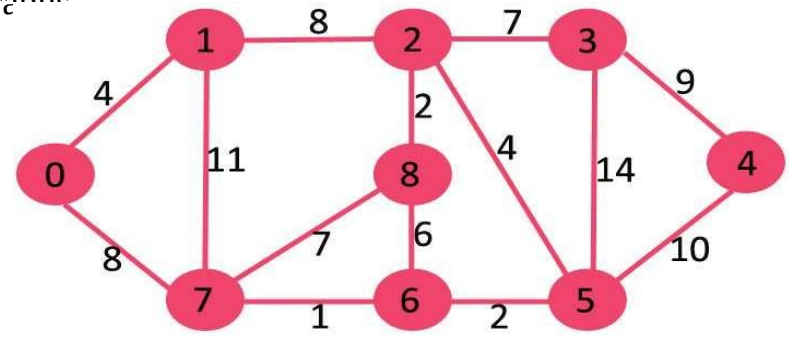
5. Apply Prim's algorithm to the following graph and find Minimum Spanning tree for the given graph



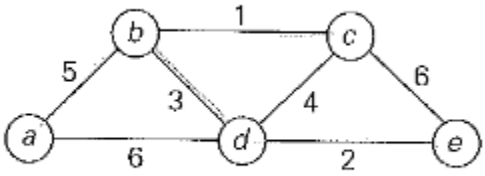
6. Write Kruska's algorithms and Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs.



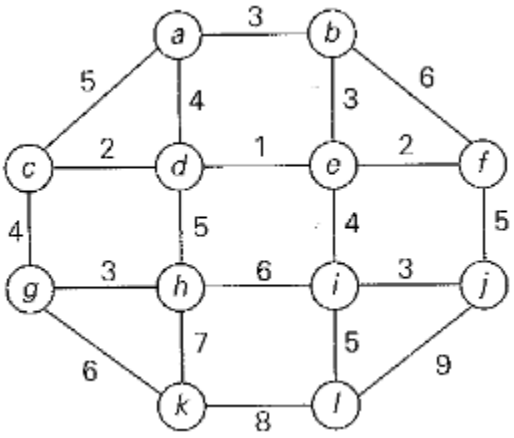
7. Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs



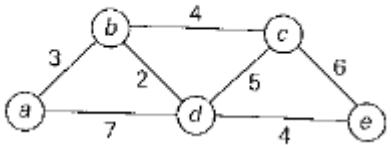
8. Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs.



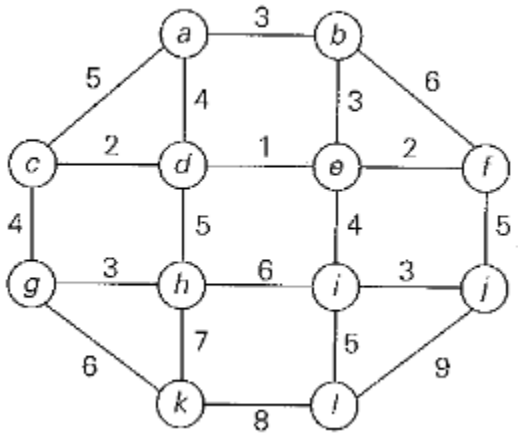
9. Apply Kruskal's algorithm to find a minimum spanning tree of the followinggraphs.



10. Write Dijkstra's Algorithm and solve the following instances of the single-source shortest-paths problem with vertex"a" as the source.



11. Using Dijkstra's Algorithm to solve the following instances of the single-source shortest-paths problem withvertex "a" as the source:



12. Write and explain the Huffman algorithm and construct Huffman coding tree.
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

13. Construct a Huffman code for following data.

character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

14.

a. Construct a Huffman code for the following data:

character	A	B	C	D	_
probability	0.4	0.1	0.2	0.15	0.15

b. Encode the text ABACABAD using the code of question (a).

c. Decode the text whose encoding is 100010111001010 in the code of question (a).

15. Write a note on following
   a. Trivial lower Bounds
   b. Information-Theoretic Arguments

c. Adversary Arguments
d. Problem Reduction

16. Explain Decision Tree and draw the Decision tree for minimum of three numbers,
17. Draw the Decision tree for binary search in a four-element array.
18. Draw the Decision tree for the three-element selection sort.
19. Draw the Decision tree for the three-element insertion sort
20. Explain P, NP, and NP-complete Problems.
21. Write a note on Challenges of Numerical Algorithm