Design and Analysis of Algorithms

Unit-3

Greedy Algorithm

Greedy Algorithm

- A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms are used to solve optimization problems.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.
- We will solve the following problems using greedy algorithm:-
  - Activity selection problem
  - Knapsack problem
  - Minimum spanning tree problem
  - Shortest path problem

Activity selection problem

- Suppose we have a set $S = \{a_1, a_2, a_3, \ldots\ldots, a_n\}$ of n proposed ***activities*** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.
- Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \le s_i < f_i < \infty$.
- Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$.
- In the ***activity-selection problem***, we wish to select a maximum-size subset of mutually compatible activities.

Activity selection problem

Example: For example, consider the following set S of activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Solve this activity selection problem.'

Solution: The subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.

It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities. Another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

Activity selection problem

A recursive greedy algorithm

We assume that the n input activities are already ordered by monotonically increasing finish time. If not, we can sort them into this order in O(nlgn) time, breaking ties arbitrarily. Following algorithm solves the activity selection problem recursively.

```
RECURSIVE-ACTIVITY-SELECTOR$(s, f, k, n)$
m = k + 1
while $m \le n$ and $s[m] < f[k]$        // find the first activity in $S_k$ to finish
    $m = m + 1$
if $m \le n$
    return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, n)$
else return $\emptyset$
```

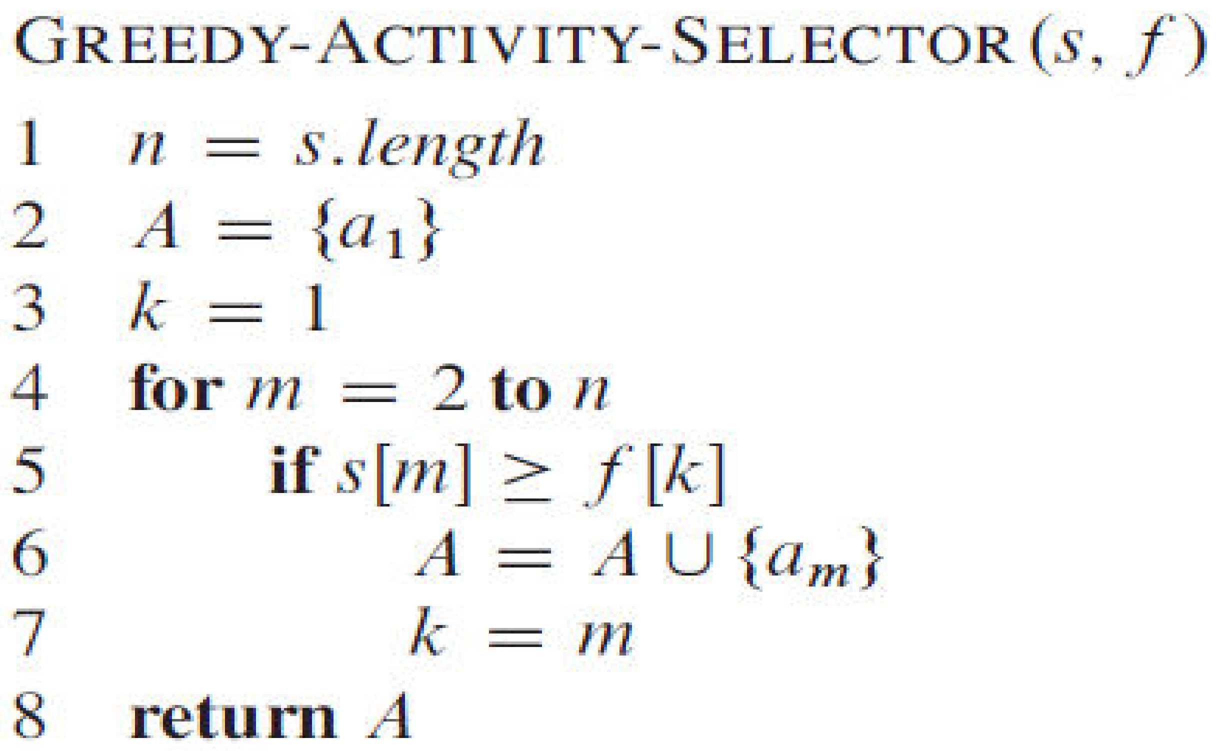
The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n).

Time complexity: the running time of this algorithm is **$\theta(n)$.**

Activity selection problem

An iterative greedy algorithm

We assume that the n input activities are already ordered by monotonically increasing finish time. If not, we can sort them into this order in O(nlgn) time, breaking ties arbitrarily. Following algorithm solves the activity selection problem iteratively



```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

Time complexity: The running time of this algorithm is **$\theta(n)$.**

Greedy algorithm property

Greedy choice property

we can assemble a globally optimal solution by making locally optimal (greedy) choices.

In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Optimal substructure property

A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

Knapsack problem

- A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are n items available in the store and weight of ***i***i^{th} item is $\boldsymbol{w_i}$ and its profit is $\boldsymbol{p_i}$. What items should the thief take?
- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W.
- Based on the nature of the items, Knapsack problems are categorized as
  1. Fractional Knapsack
  2. 0-1 Knapsack

Knapsack problem

Fractional Knapsack

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction $\boldsymbol{x_i}$ of **i**th item.

0-1 Knapsack

In this version of Knapsack problem, we cannot break an item, either pick the complete item or don’t pick it (0-1 property).

Knapsack problem

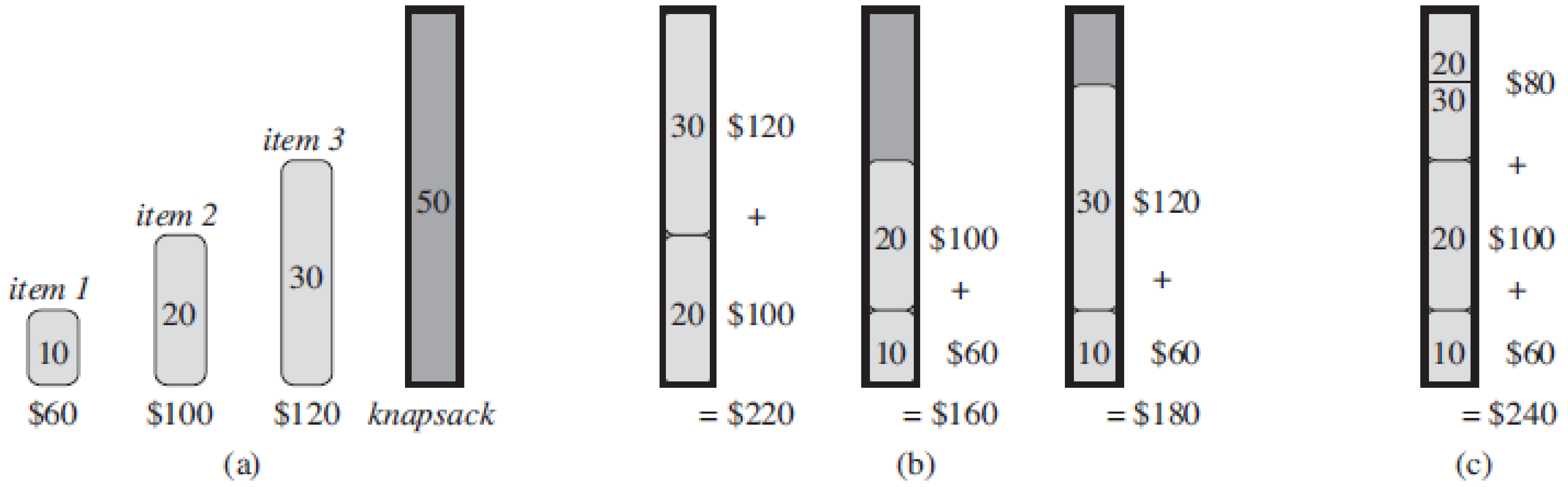
Greedy approach to solve the knapsack problem

- First, we compute value v_i/w_i for each item i.
- Arrange all the items in descending order on the basis of v_i/w_i .
- Put first item in the knapsack fully if weight of first item is less than or equal to W. Put first item in the knapsack partially if weight of first item is greater than W.
- Similarly, put the second item in the knapsack fully if there is a sufficient space in the knapsack. Otherwise, we put second item partially.
- This process continue till Knapsack is filled.

Knapsack problem

Note: Greedy algorithm solves the fractional knapsack but 0-1 Knapsack can not be solved by greedy algorithm.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in following figure:-



Knapsack problem

Example: Consider the following instance for knapsack problem. Find the solution using Greedy method:

N= 8, W=130

W = {21, 31, 43, 53, 41, 63, 65, 75}

V = {11, 21, 31, 33, 43, 53, 65, 65}

Solution: Compute

$$\frac{V}{W} = \{ \frac{11}{21}, \frac{21}{31}, \frac{31}{43}, \frac{33}{53}, \frac{43}{41}, \frac{53}{63}, \frac{65}{65}, \frac{65}{75} \}$$

= { 0.52, 0.68, 0.72, 0.62, 1.05, 0.84, 1, 0.87 }

First arrange all items in descending order of their value per weight i.e. (v_i/w_i).

W' = {41,65,75,63,43,31,53,21}

V' = {43,65,65,53,31,21,33,11}

Descending order of items = item5, item7, item8, item6, item3, item2, item4, item1

Therefore, optimal solution = (0, 0, 0, 0, 1, 0, 1, 24/75)

Optimal value = 43 + 65 + (24/75)*65 = 108 + 20.8 = **128.8**

Knapsack problem

Example: Consider the following instance for knapsack problem. Find the solution using Greedy method:

w = (5, 10, 20, 30, 40),   v = (30, 20, 100, 90,160)

The capacity of knapsack W = 60

Example: Given the six items in the table below and a Knapsack with Weight 100, what is the solution to the Knapsack problem in all concepts. i.e. explain greedy all approaches and find the optimal solution.

ITEM ID	WEIGHT	VALUE	VALUE/WEIGHT
A	100	40	.4
B	50	35	.7
C	40	20	.5
D	20	4	.2
E	10	10	1
F	10	6	.6

Knapsack problem

Algorithm: Assume all the items are arranged in descending order of their value per weight i.e. (v_i/w_i).

```
Fractional_Knapsack(w, v, W)
        n = length[w]
        for i=1 to n
         do      x[i] = 0
        i=1
        weight = 0
        while( I <= n and weight < W)
        do       if(weight+w[i] < = W)
                 then     x[i] = 1
                          weight = weight + w[i]
                 else
                          x[i] =  (W – weight)/$w_i$
                          weight = W
        return x
```

- Time complexity of this algorithm will be O(n) if all the items are arranged in descending order of their value per weight .
- Time complexity of this algorithm will be O(nlogn) if all the items are not arranged in descending order of their value per weight .

Minimum Spanning Tree

Spanning tree:

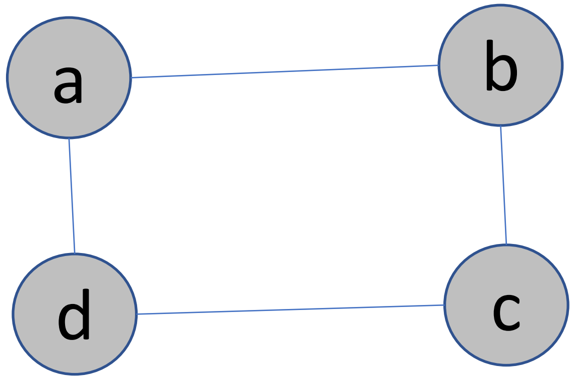
Spanning tree is a non-cyclic sub-graph of a connected and undirected graph G that connects all the vertices together.

General Properties of Spanning Tree

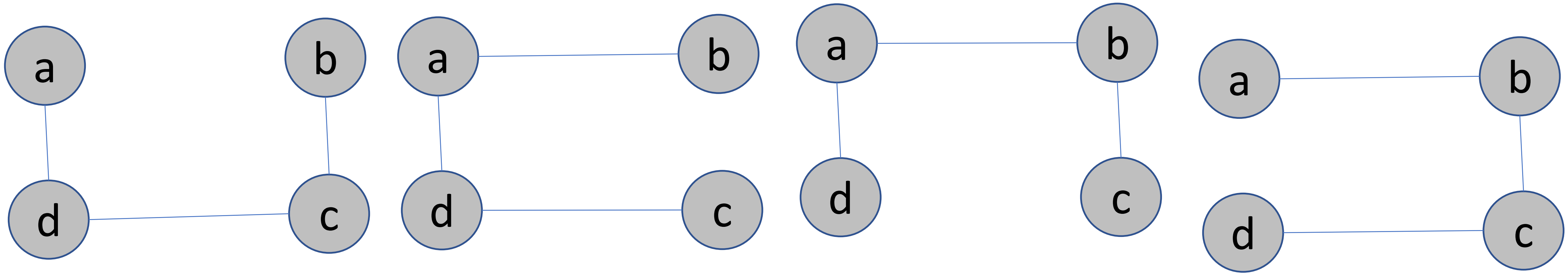
- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop.
- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Minimum Spanning Tree

Example: Consider the following graph:-



All the spanning tree of this graph are the following:-

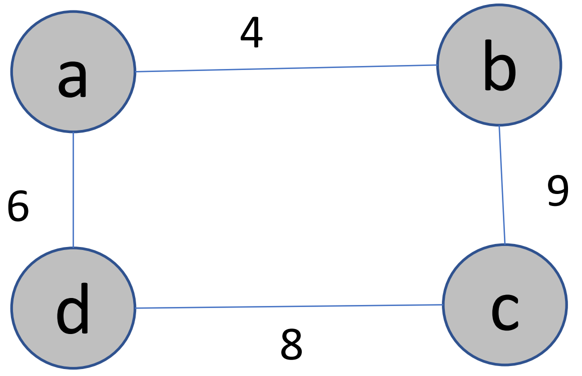


Minimum Spanning Tree

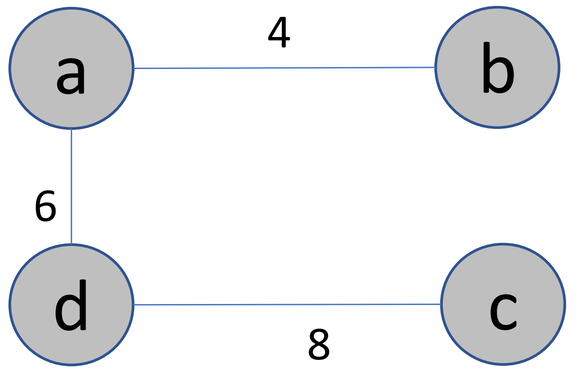
If the given graph is weighted graph, then we define the minimum spanning tree.

Definition: A spanning tree is said to be minimum spanning tree if sum of weights of all the edges in the tree is smallest.

Example: Consider the following graph:-



Graph



Total cost = 18

Minimum spanning tree

Minimum Spanning Tree

- In this chapter, we will study two algorithms to find the minimum spanning tree.

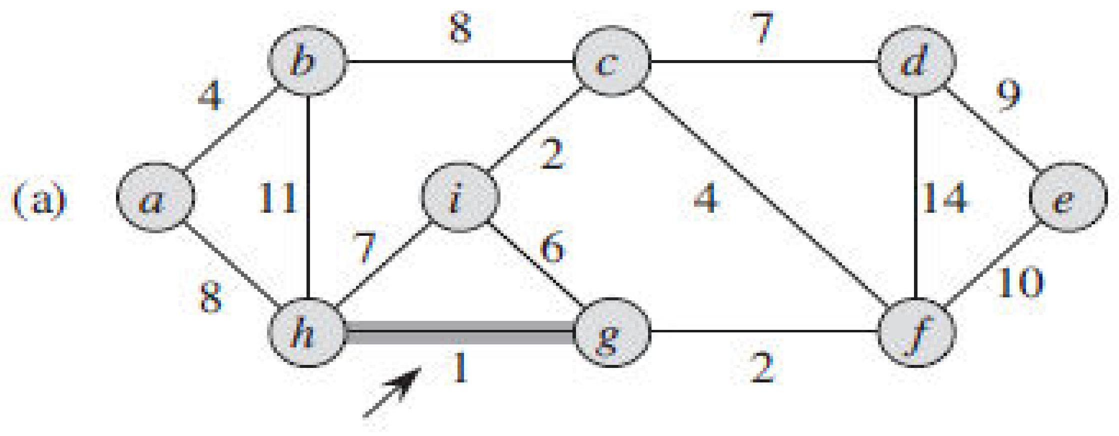
(1) Kruskal’s Algorithm

(2) Prim’s Algorithm

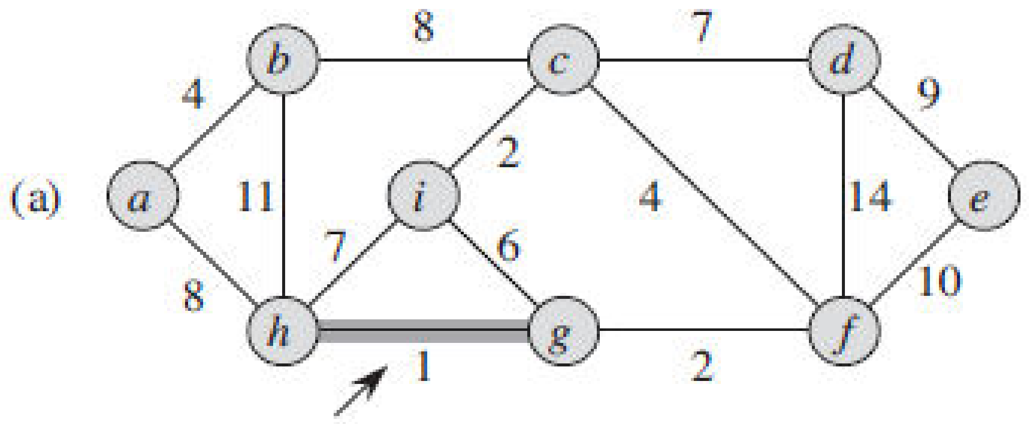
Both algorithms are based on Greedy approach.

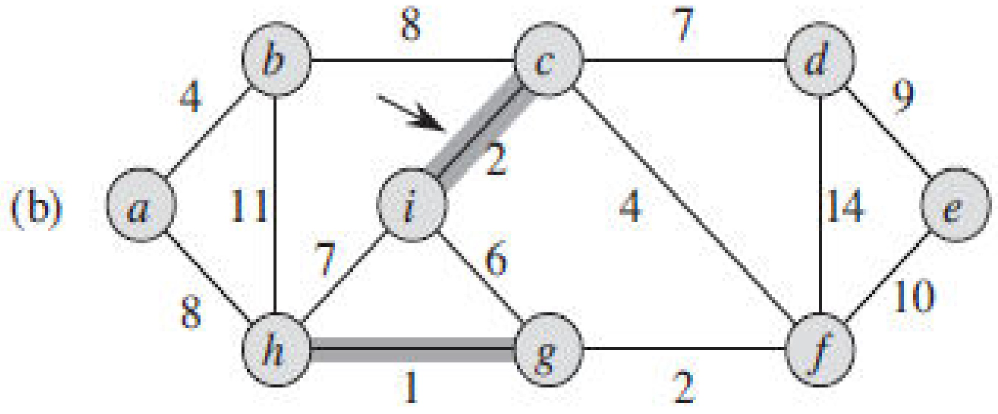
Kruskal's Algorithm

Example: Find the minimum spanning of the following graph using Kruskal algorithm.



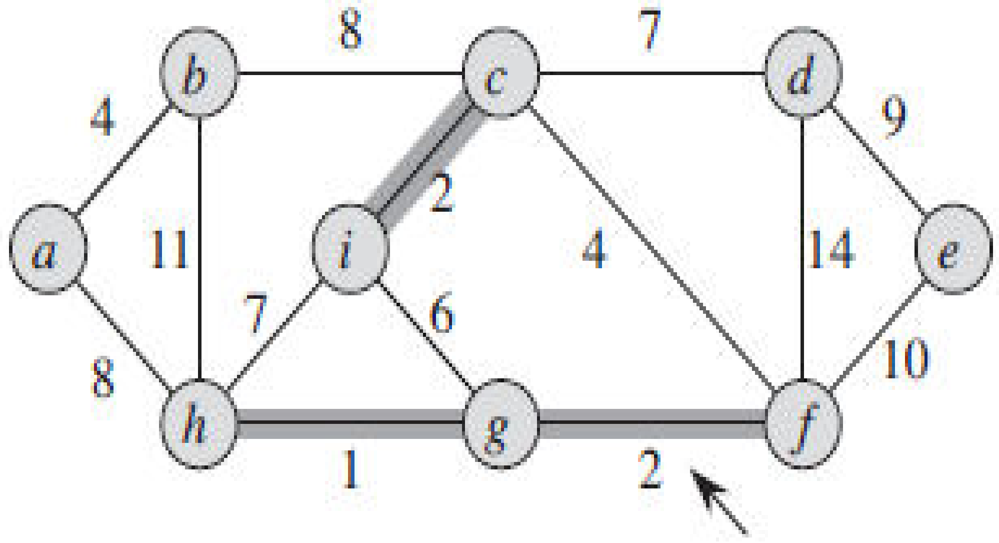
Solution:



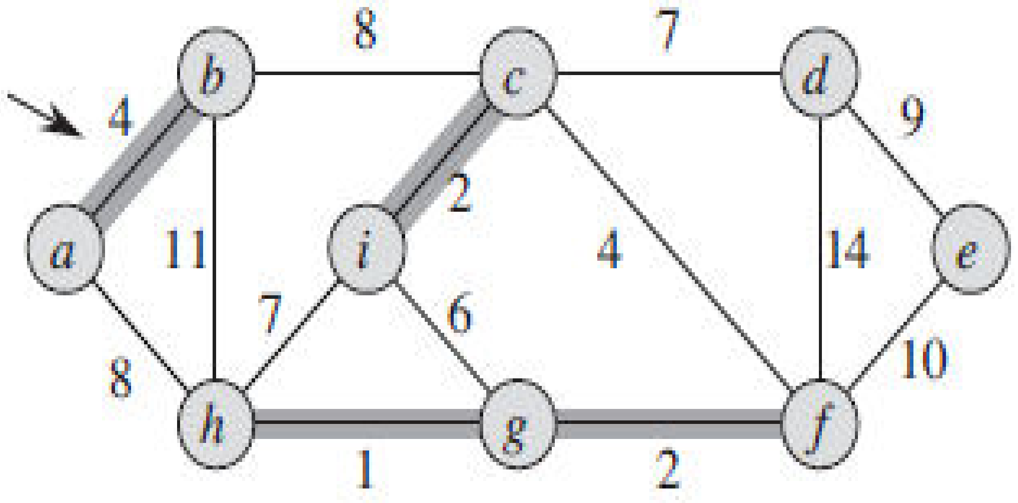


Kruskal’s Algorithm

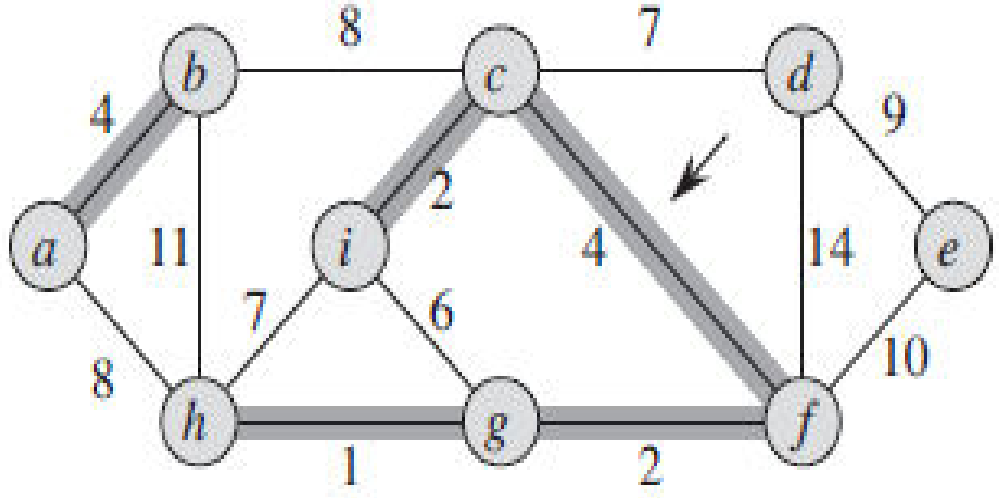
(c)



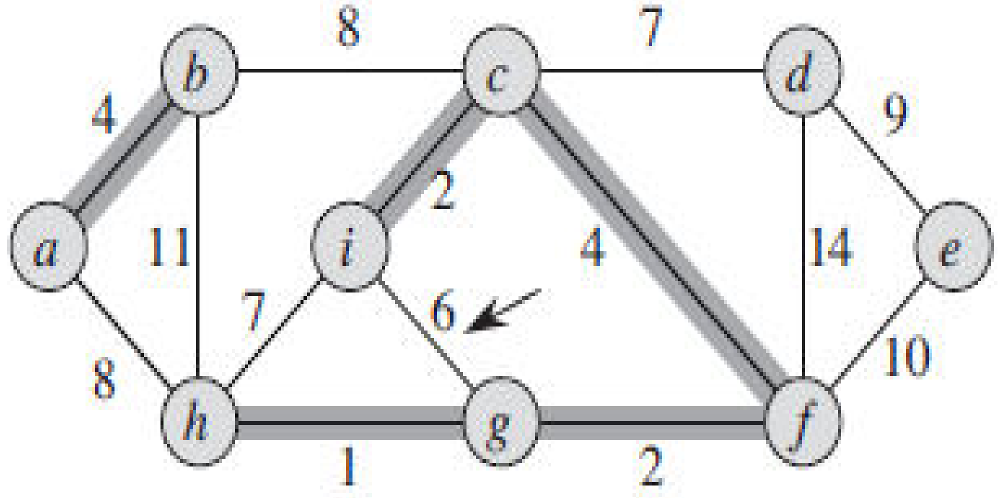
(d)



(e)

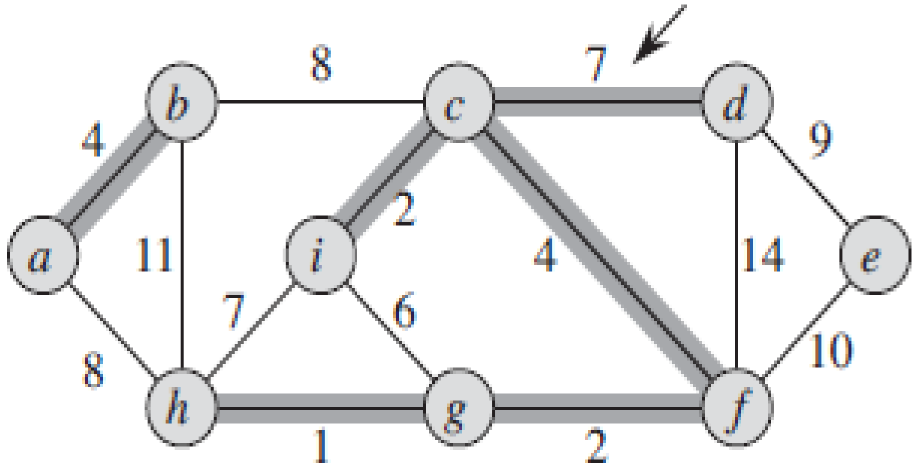


(f)

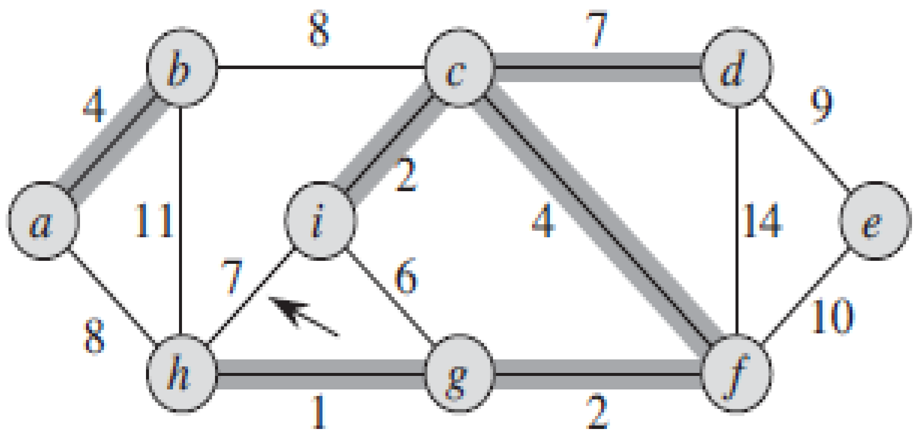


Kruskal’s Algorithm

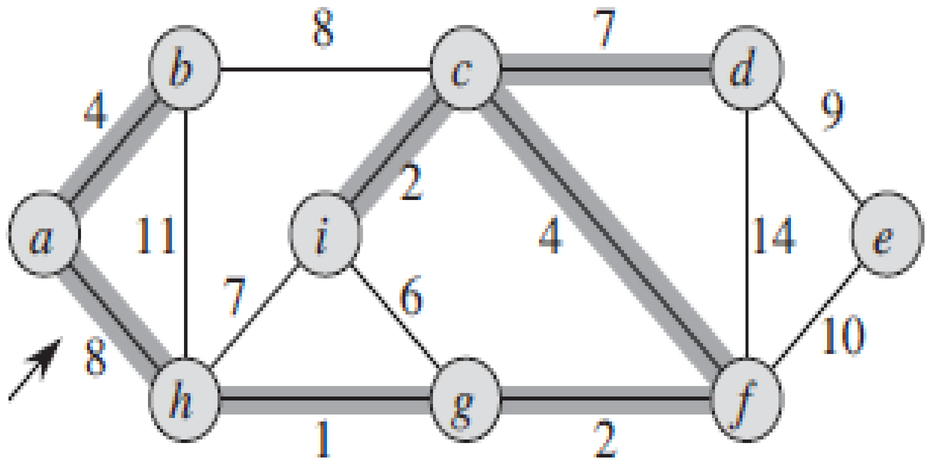
(g)



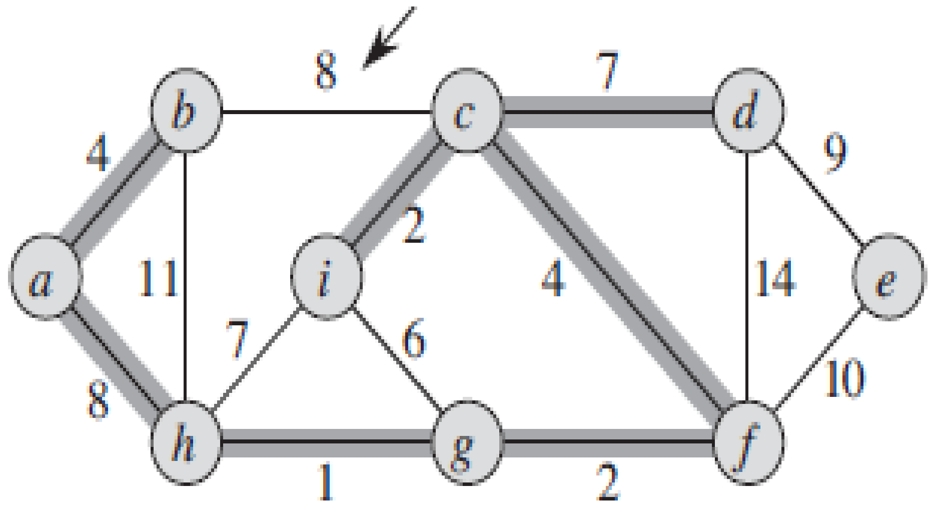
(h)



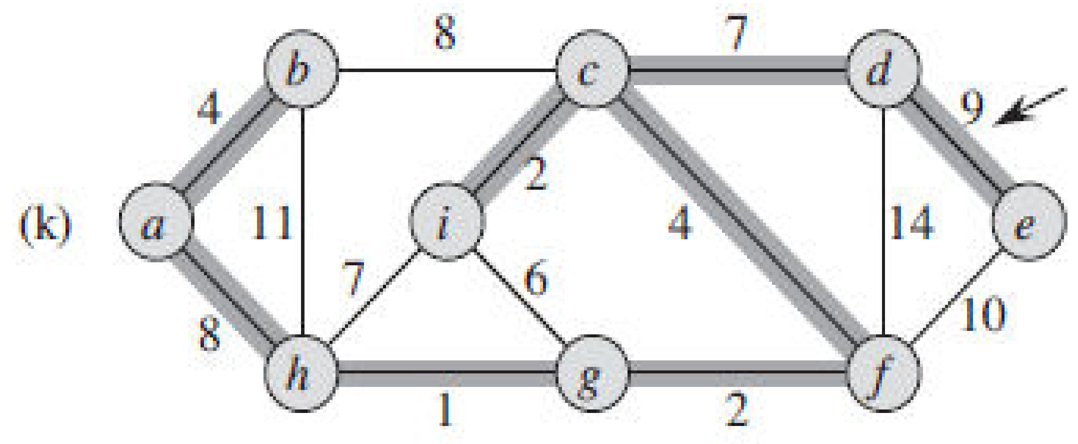
(i)

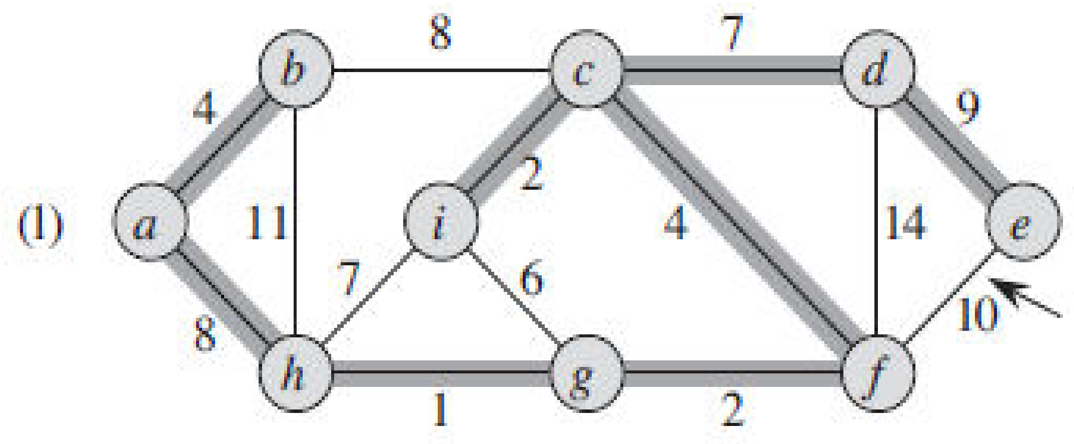


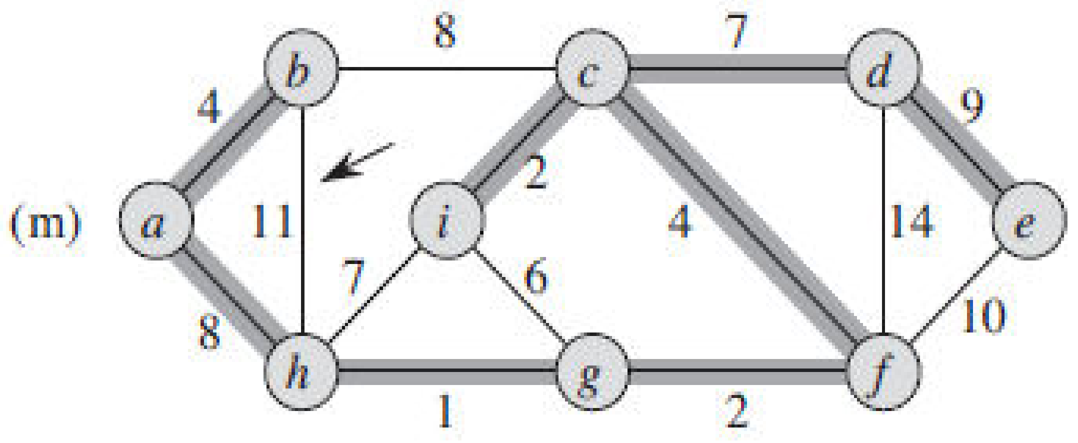
(j)

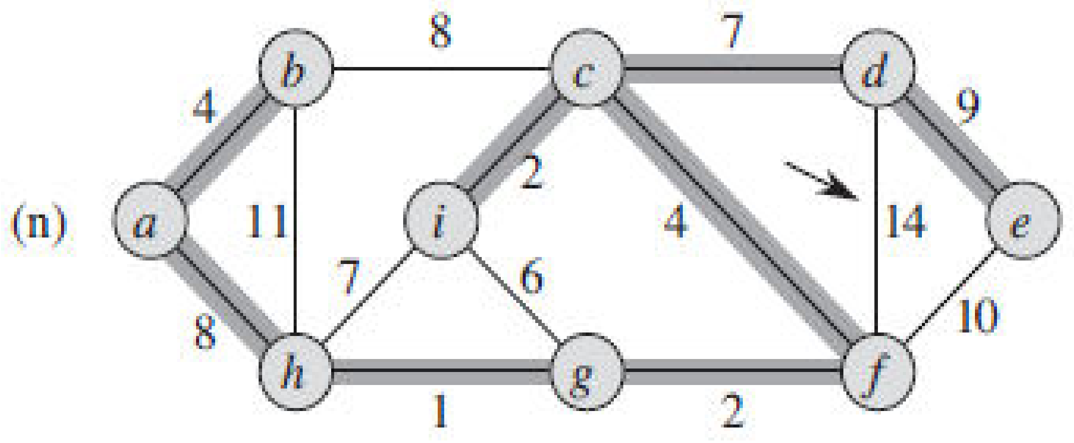


Kruskal's Algorithm





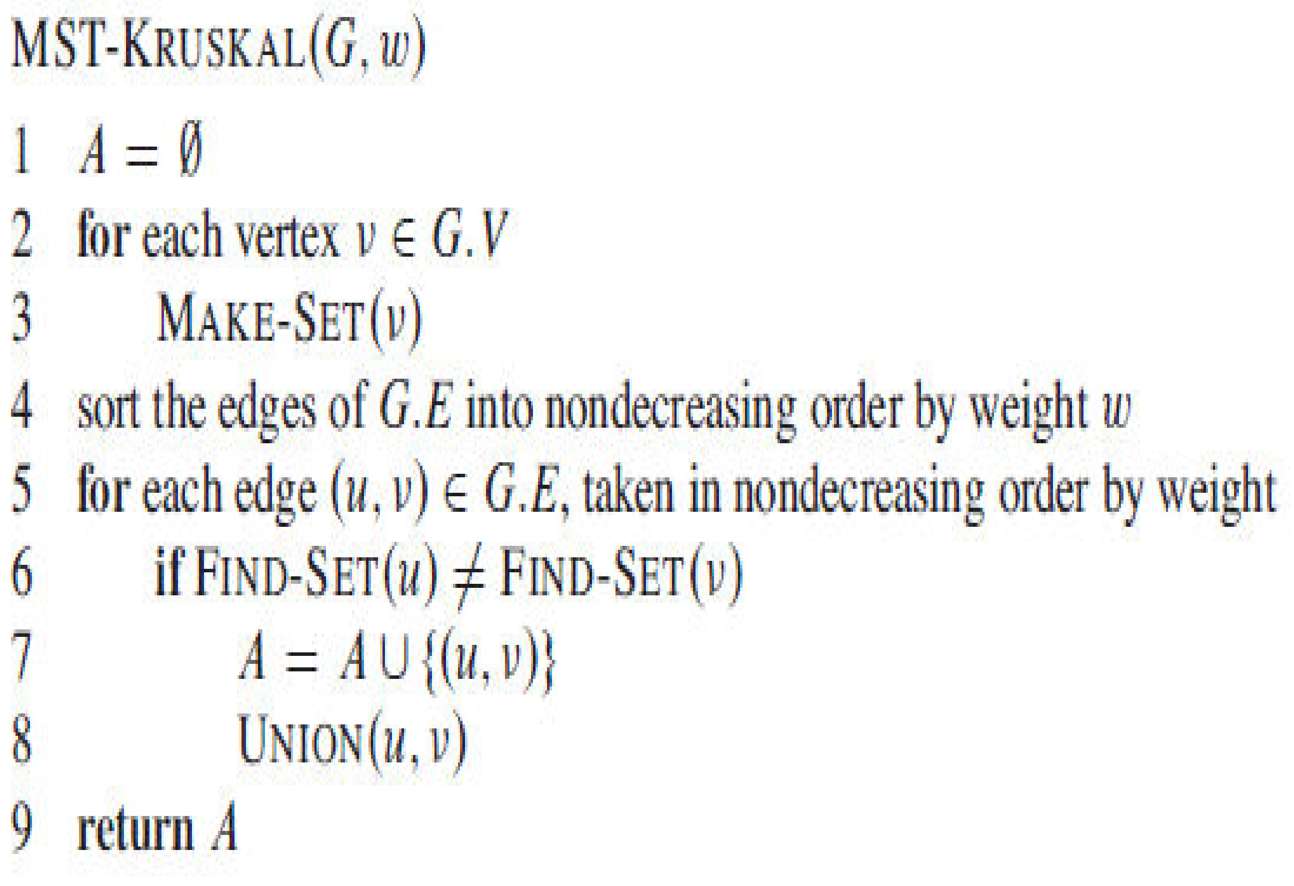




Total cost = 37

Final minimum spanning tree

Kruskal’s Algorithm

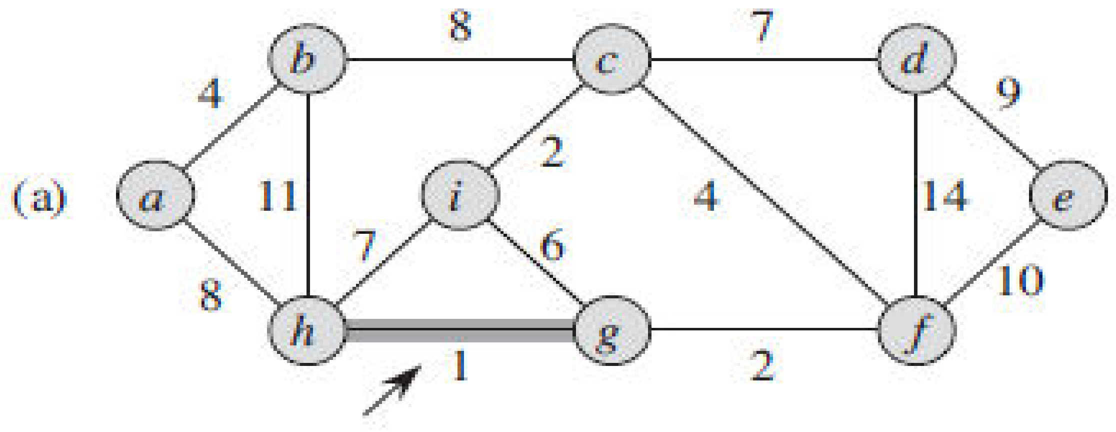


```
MST-KRUSKAL($G, w$)
1  $A = \emptyset$
2  for each vertex $v \in G.V$
3      MAKE-SET($v$)
4  sort the edges of $G.E$ into nondecreasing order by weight $w$
5  for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6      if FIND-SET($u$) $\neq$ FIND-SET($v$)
7          $A = A \cup \{(u, v)\}$
8          UNION($u, v$)
9  return $A$
```

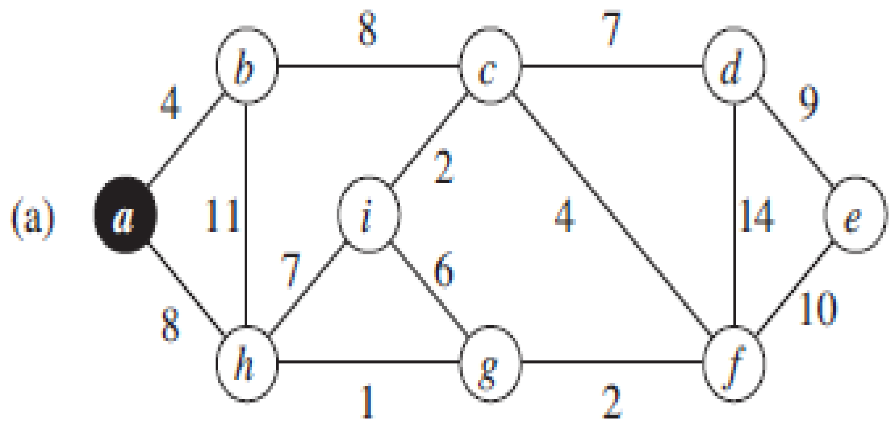
Time complexity of this algorithm is O(E lg E) or O(E lgV).

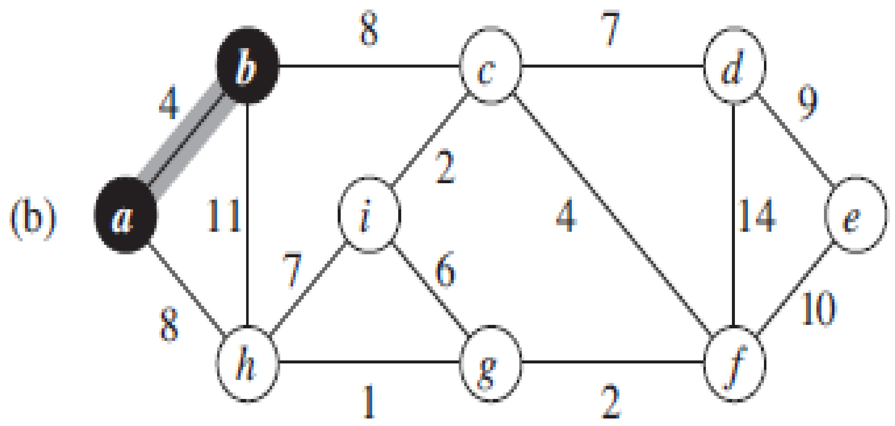
Prim's Algorithm

Example: Find the minimum spanning of the following graph using Prim's algorithm.

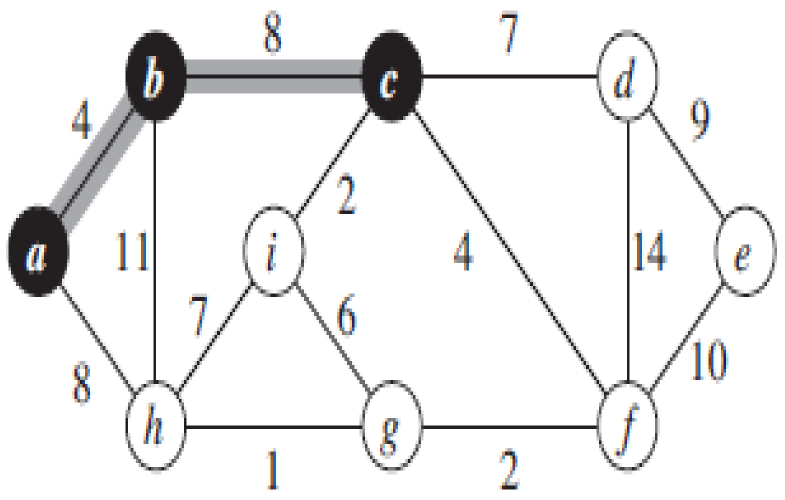


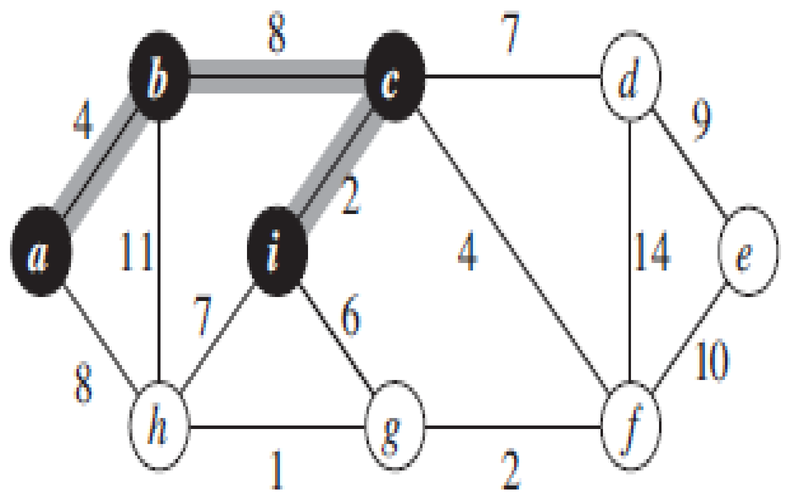
Solution:

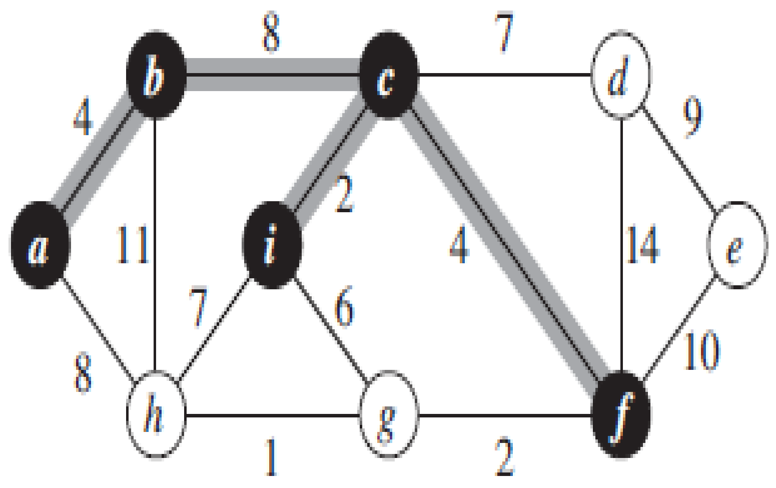


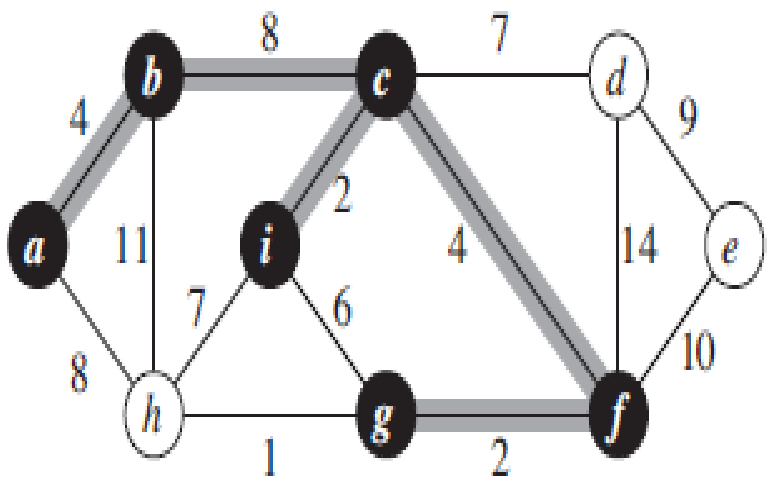


Prim’s Algorithm

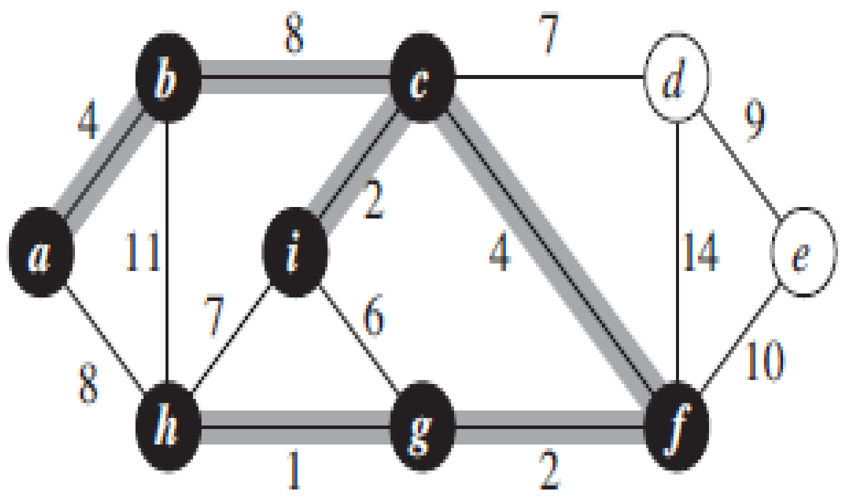
(c)


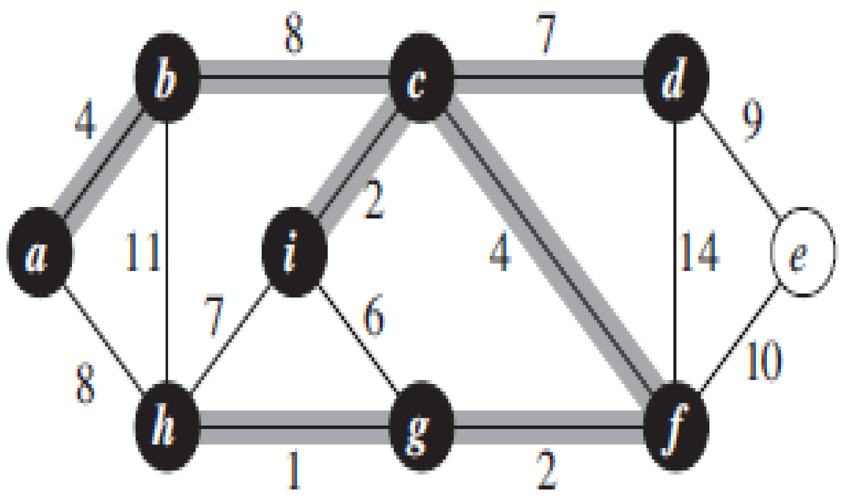
(d)


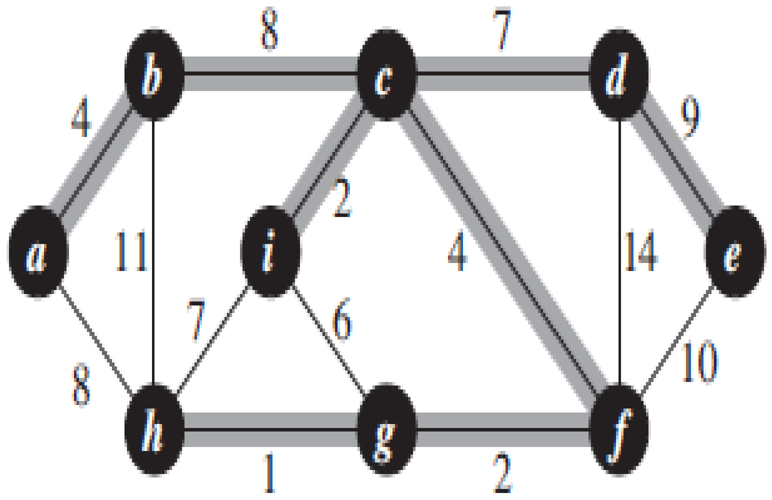
(e)


(f)


Prim’s Algorithm

(g) 

(h) 

(i) 

Total cost = 37

Final minimum spanning tree

Prim’s Algorithm

```
MST-PRIM($G, w, r$)
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi =$ NIL
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN($Q$)
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```

Time complexity of this algorithm is $O(E \lg V + V \lg V) = O(E \lg V)$.

Shortest path algorithm

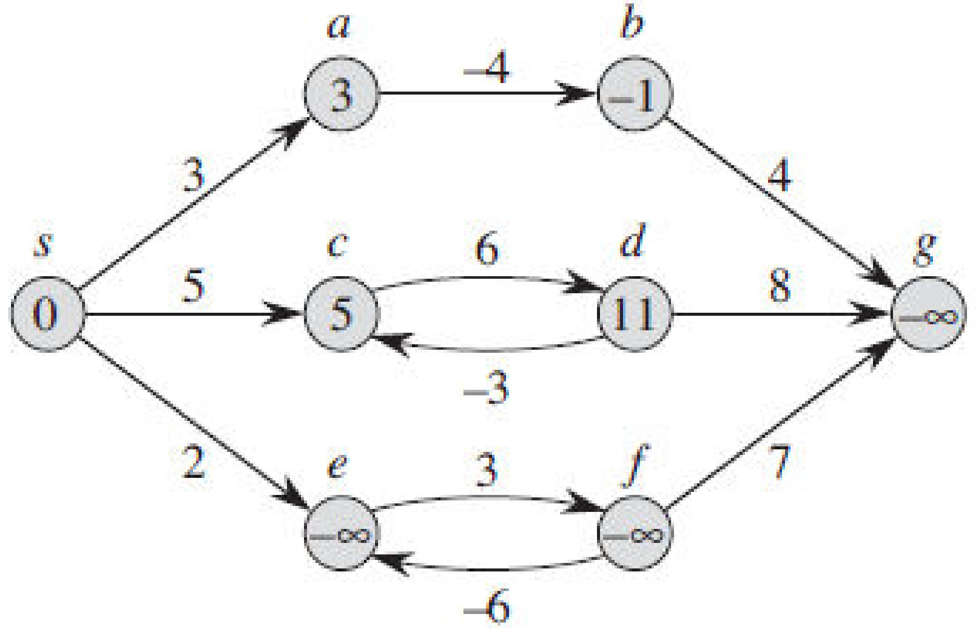
Single-source shortest-paths problem

Given a graph G = (V, E), we want to find a shortest path from a given source vertex s ϵ V to each vertex v ϵ V .

Negative weight cycles

A cycle in the graph is said to be negative weight cycle if the sum of weights of all the edges in the cycle is negative value.

Example: Consider the following graph.



Shortest path algorithm

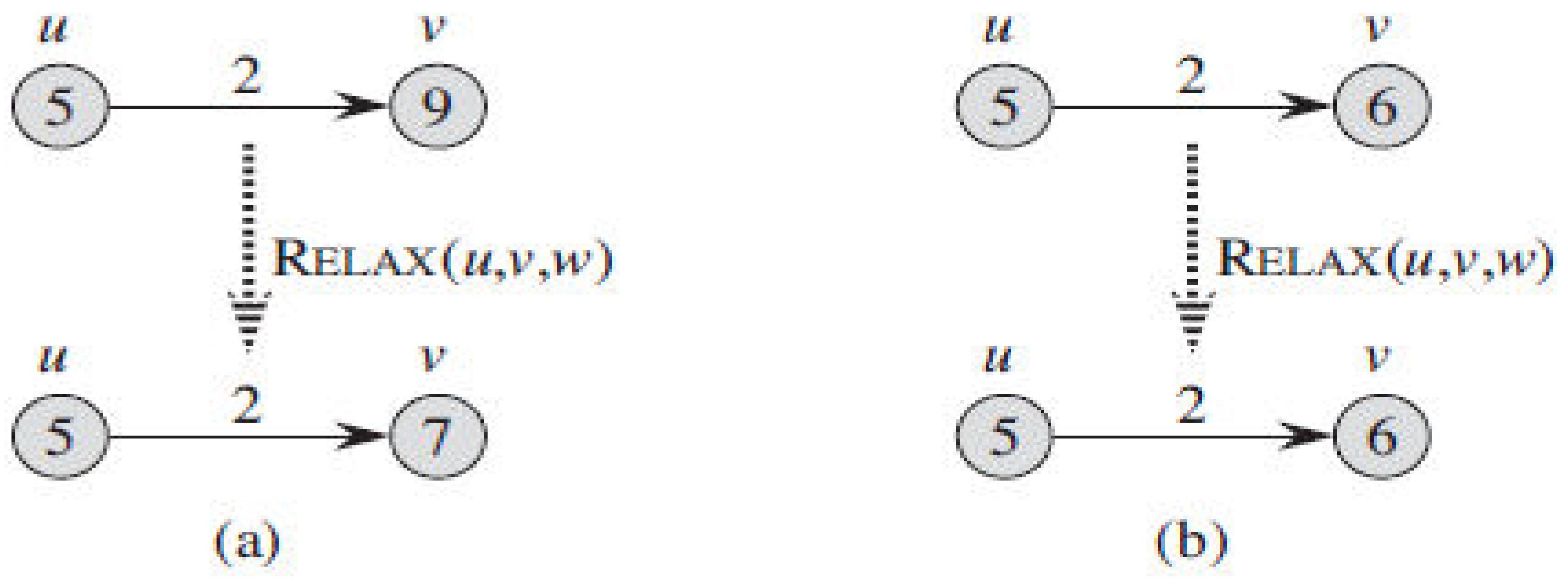
Initialize the single source function

```
INITIALIZE-SINGLE-SOURCE$(G, s)$
1  for each vertex $v \in G.V$
2      $v.d = \infty$
3      $v.\pi = \text{NIL}$
4  $s.d = 0$
```

s is the source vertex.
v.d is the distance of vertex v from s.
v.π is the predecessor vertex of v.

Shortest path algorithm

Relaxation function



```
RELAX$(u, v, w)$
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
```

Shortest path algorithm

Here, we shall study two single source shortest path algorithms. These algorithms are the following:-

1. Bellman-Ford algorithm
2. Dijkastra algrithm

Both algorithms are based on greedy approach.

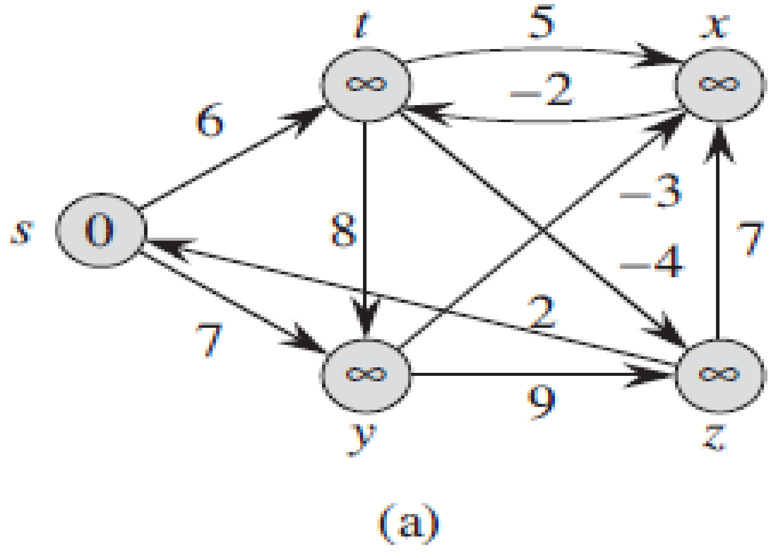
Bellman-Ford algorithm

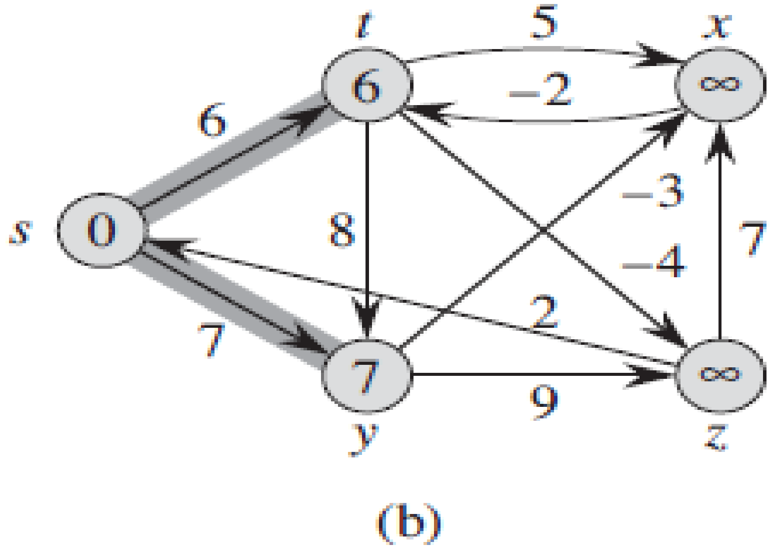
The **Bellman-Ford algorithm** solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow R$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

Bellman-Ford algorithm

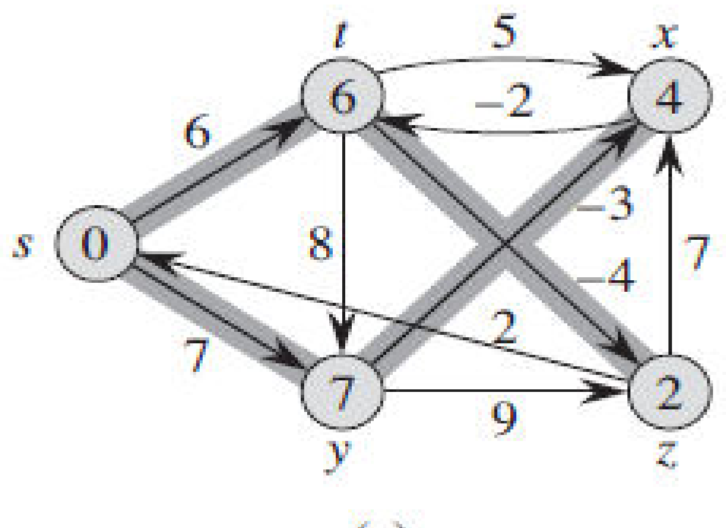
Example:



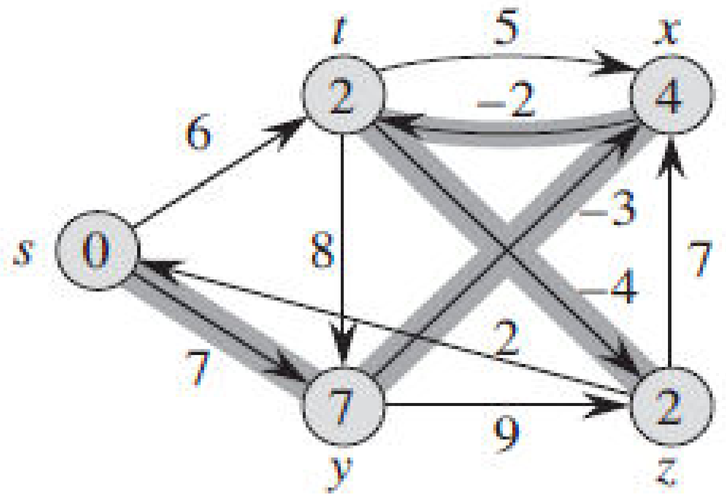


The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then: v.**π** = u. In this particular example, each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y).

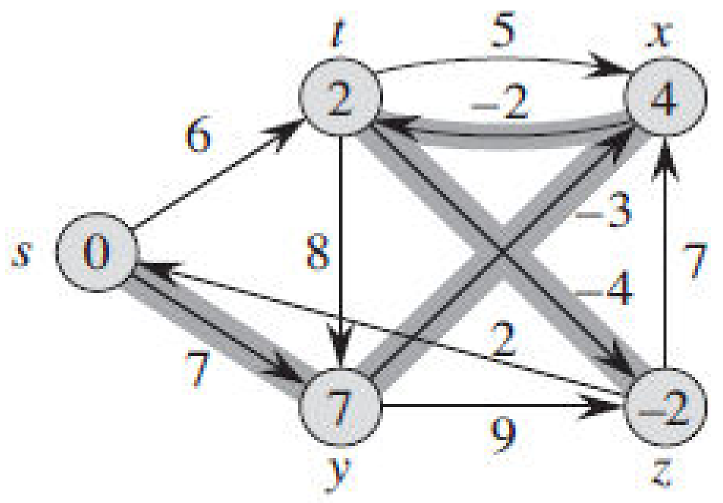
Bellman-Ford algorithm



(c)



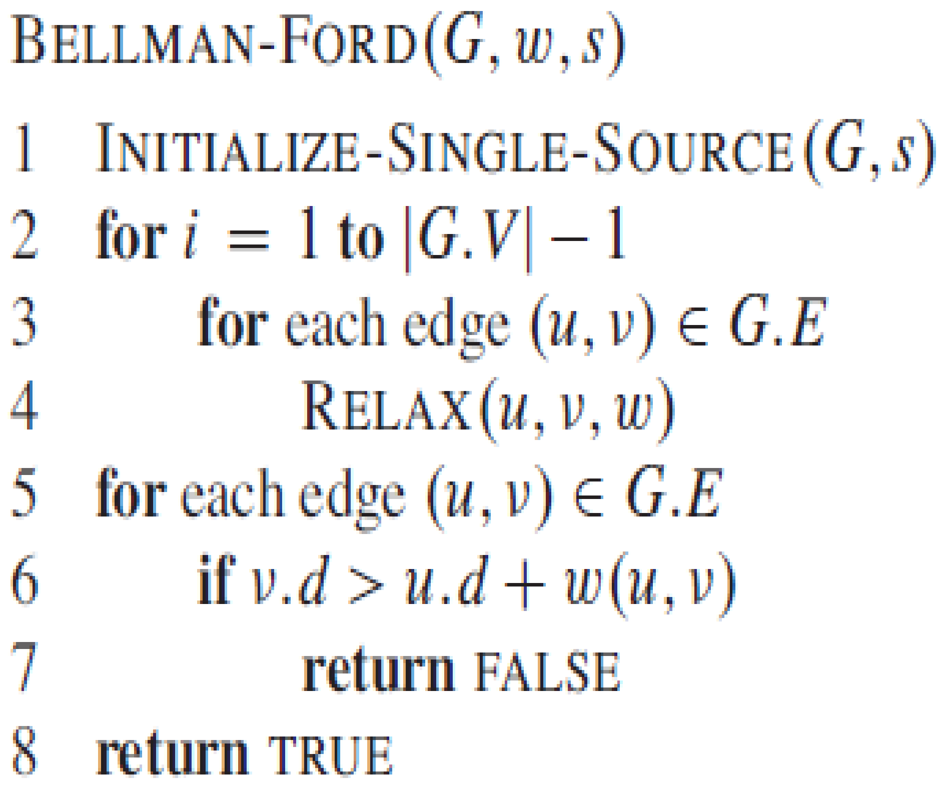
(d)



(e)

Order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y).

Bellman-Ford algorithm



```
BELLMAN-FORD$(G, w, s)$
INITIALIZE-SINGLE-SOURCE$(G, s)$
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX$(u, v, w)$
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

Time complexity of this algorithm is O(VE).

Dijkstra’s algorithm

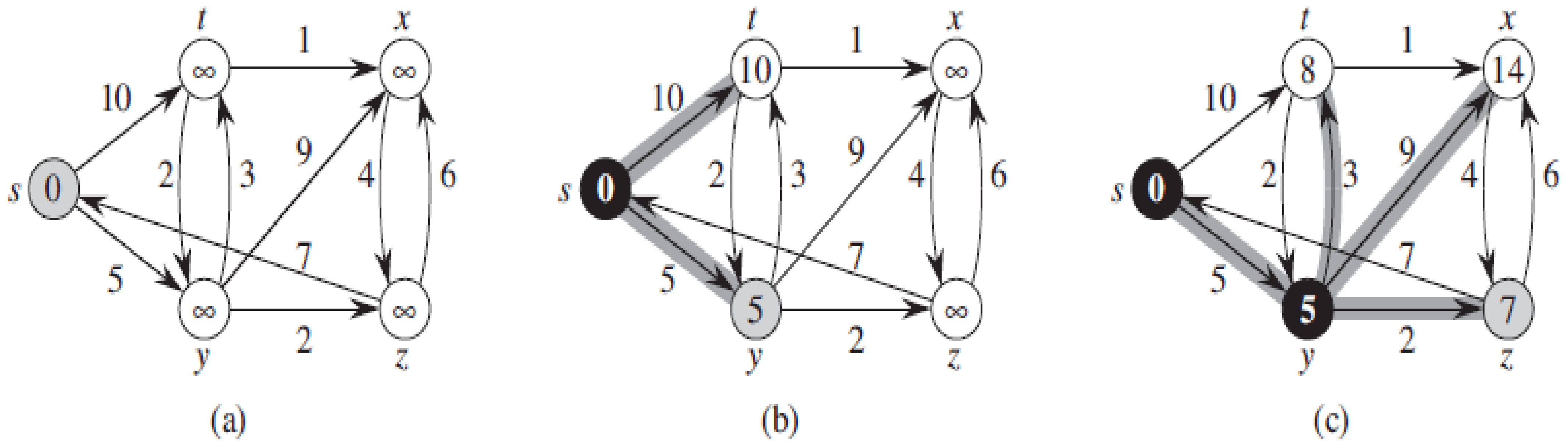
- Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative.

- Dijkstra’s algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u ∈ V-S with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

- In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

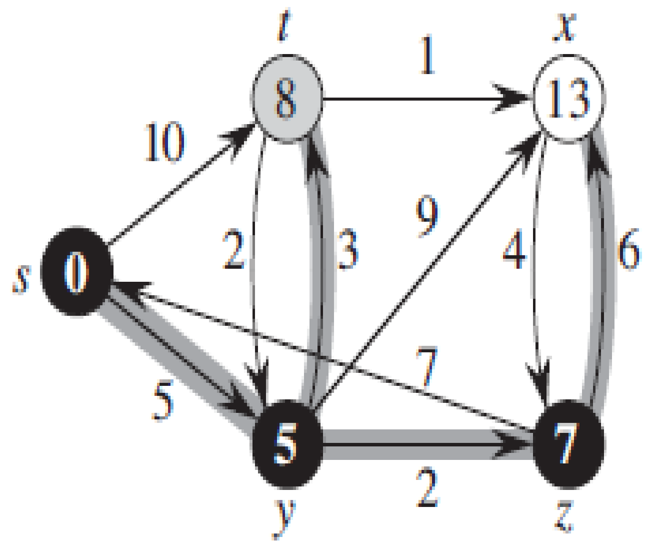
Dijkstra’s algorithm

Example:

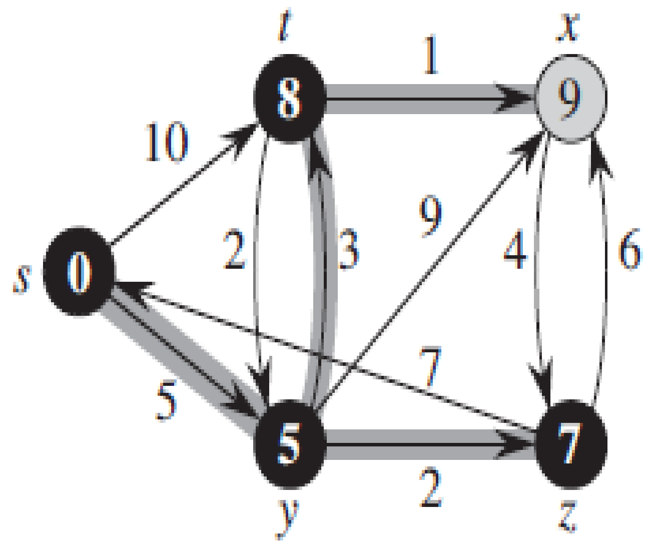


The execution of Dijkstra’s algorithm. s is the source vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue Q = V - S.

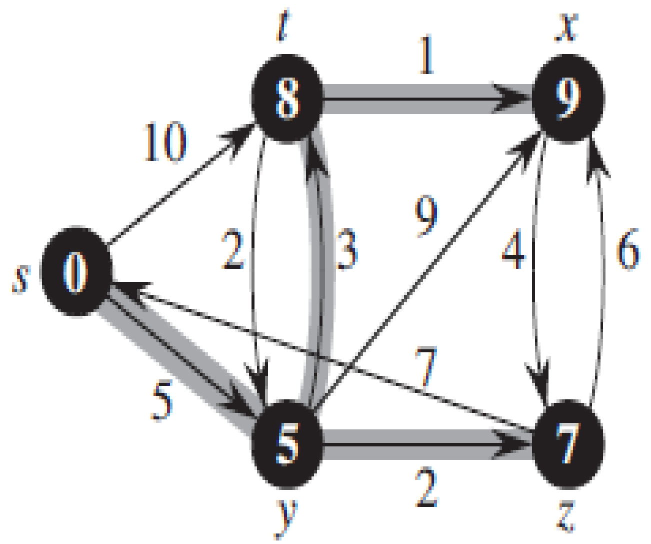
Dijkstra's algorithm



(d)

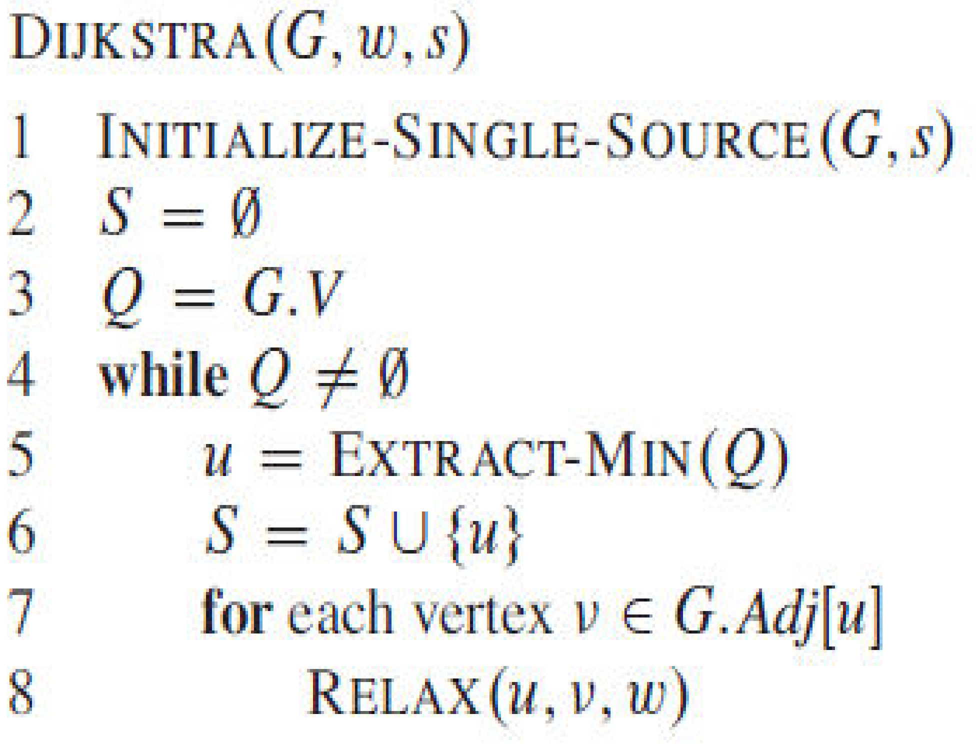


(e)



(f)

Dijkstra’s algorithm



```
DIJKSTRA$(G, w, s)$
1  INITIALIZE-SINGLE-SOURCE$(G, s)$
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN$(Q)$
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX$(u, v, w)$
```

Time complexity of this algorithm is O(E logV).

AKTU Examination Questions

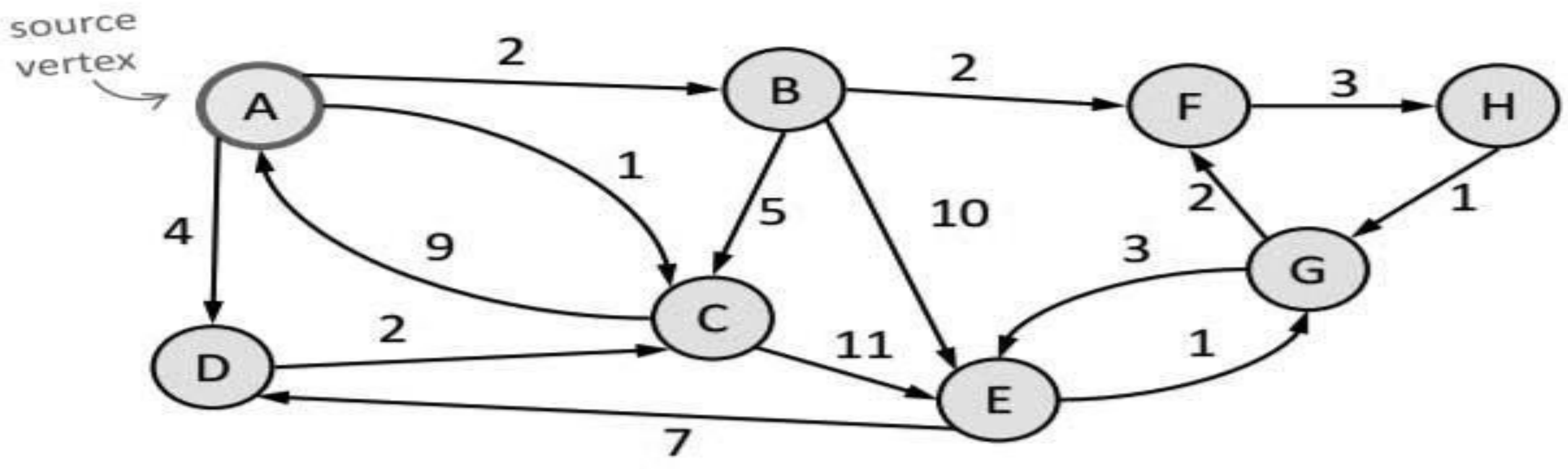
1. Consider the following instance for knapsack problem. Find the solution using Greedy method:

N= 10, W=130

P [] = {21, 31, 43, 53, 41, 63, 65, 75}

V [] = {11, 21, 31, 33, 43, 53, 65, 65}

2. Apply the greedy single source shortest path algorithm on the following graph:

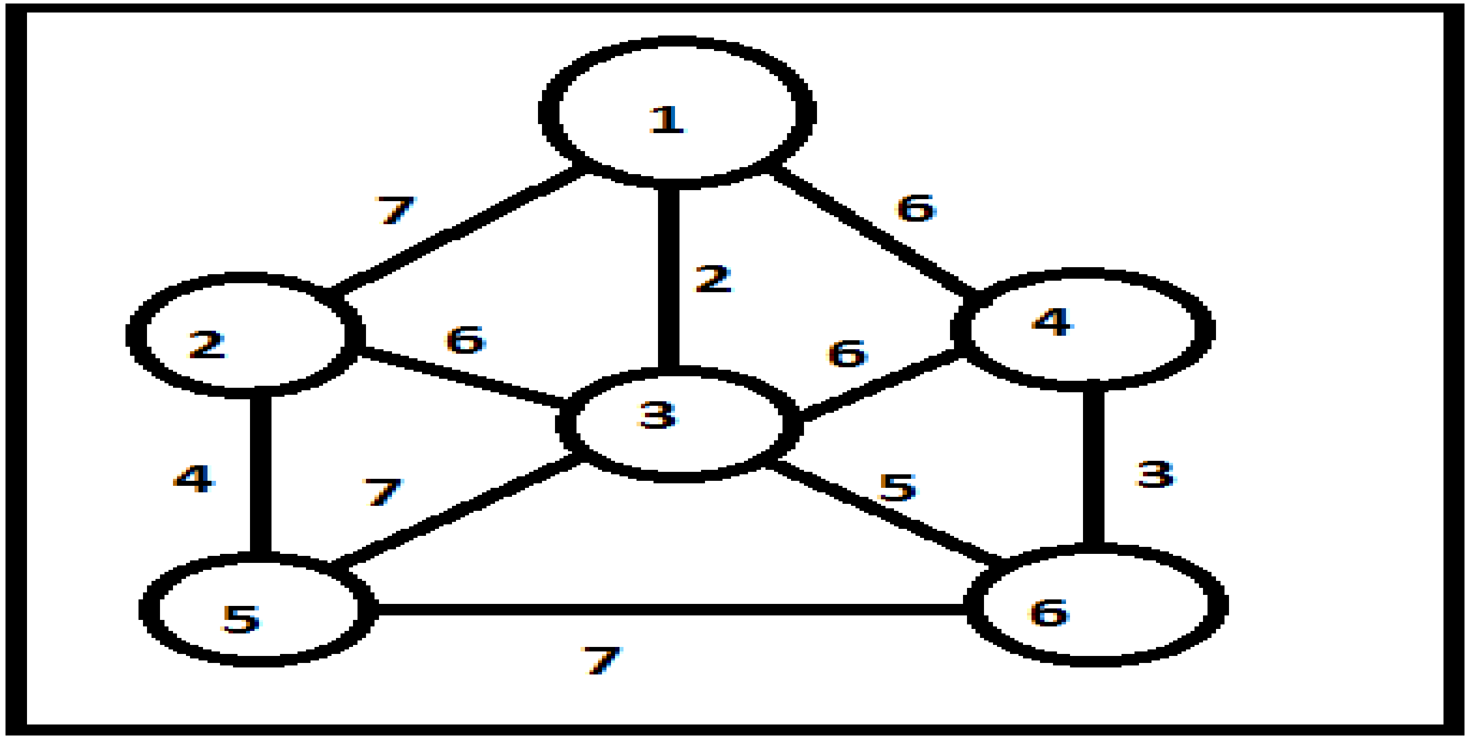


AKTU Examination Questions

3. Describe Activity selection problem.
4. Write an algorithm for minimum spanning tree with example.
5. What are greedy algorithms? Explain their characteristics?
6. Define feasible and optimal solution.
7. Define spanning tree. Write Kruskal's algorithm for finding minimum cost spanning tree. Describe how Kruskal's algorithm is different from Prim's algorithm for finding minimum cost spanning tree.
8. Explain Single source shortest path.

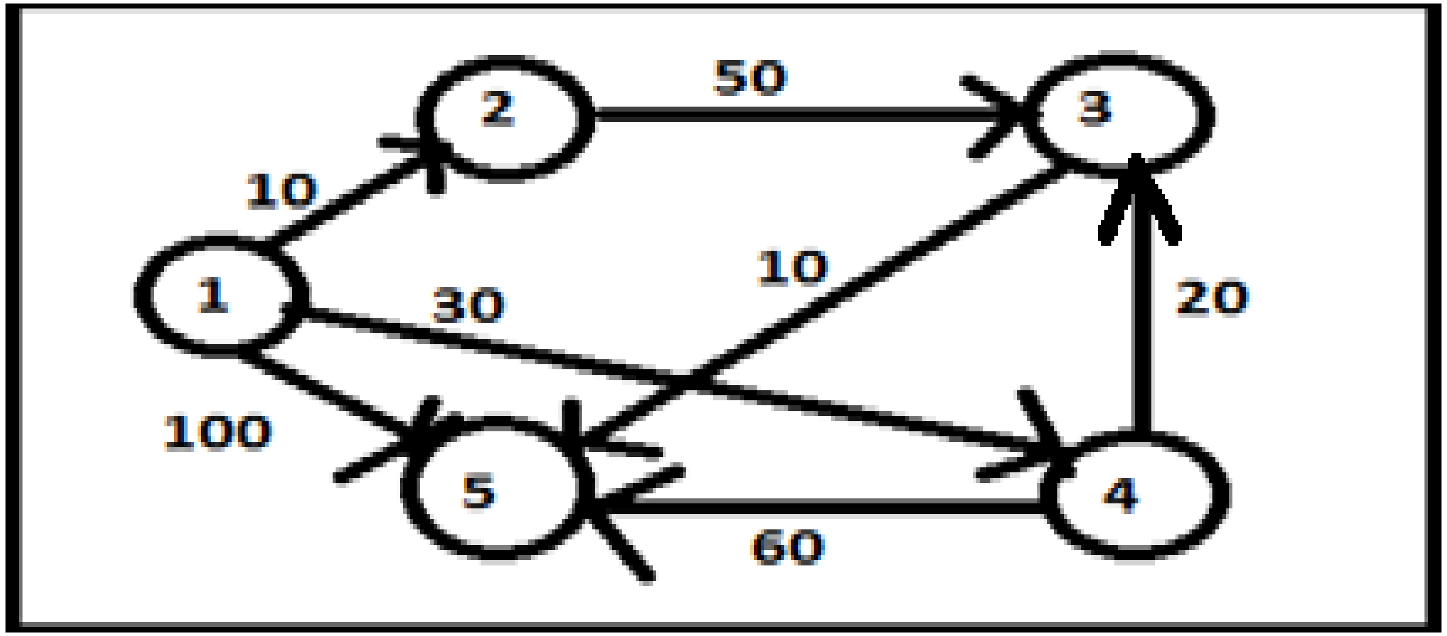
AKTU Examination Questions

9. What is Minimum Cost Spanning Tree? Explain Kruskal’s Algorithm and Find MST of the Graph. Also write its Time-Complexity.



AKTU Examination Questions

10. Find the shortest path in the below graph from the source vertex 1 to all other vertices by using Dijkstra’s algorithm.



AKTU Examination Questions

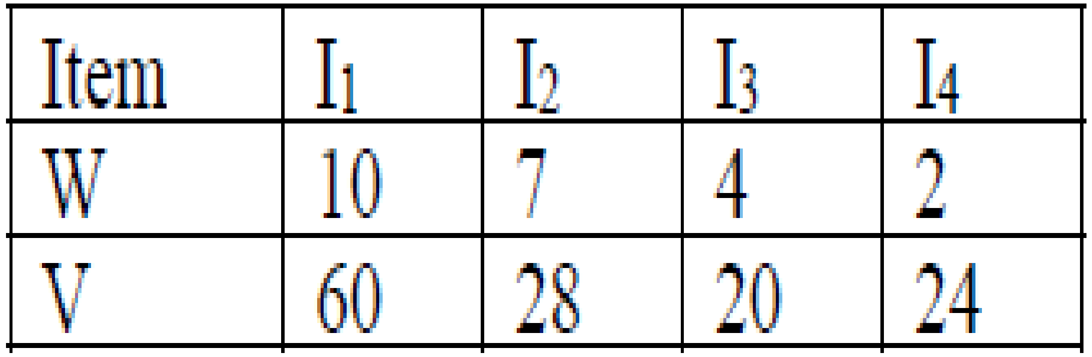
11. Given the six items in the table below and a Knapsack with Weight 100, what is the solution to the Knapsack problem in all concepts. I.e. explain greedy all approaches and find the optimal solution.

ITEM ID	WEIGHT	VALUE	VALUE/WEIGHT
A	100	40	.4
B	50	35	.7
C	40	20	.5
D	20	4	.2
E	10	10	1
F	10	6	.6

12. Prove that if the weights on the edge of the connected undirected graph are distinct then there is a unique Minimum Spanning Tree. Give an example in this regard. Also discuss Prim's Minimum Spanning Tree Algorithm in detail.

AKTU Examination Questions

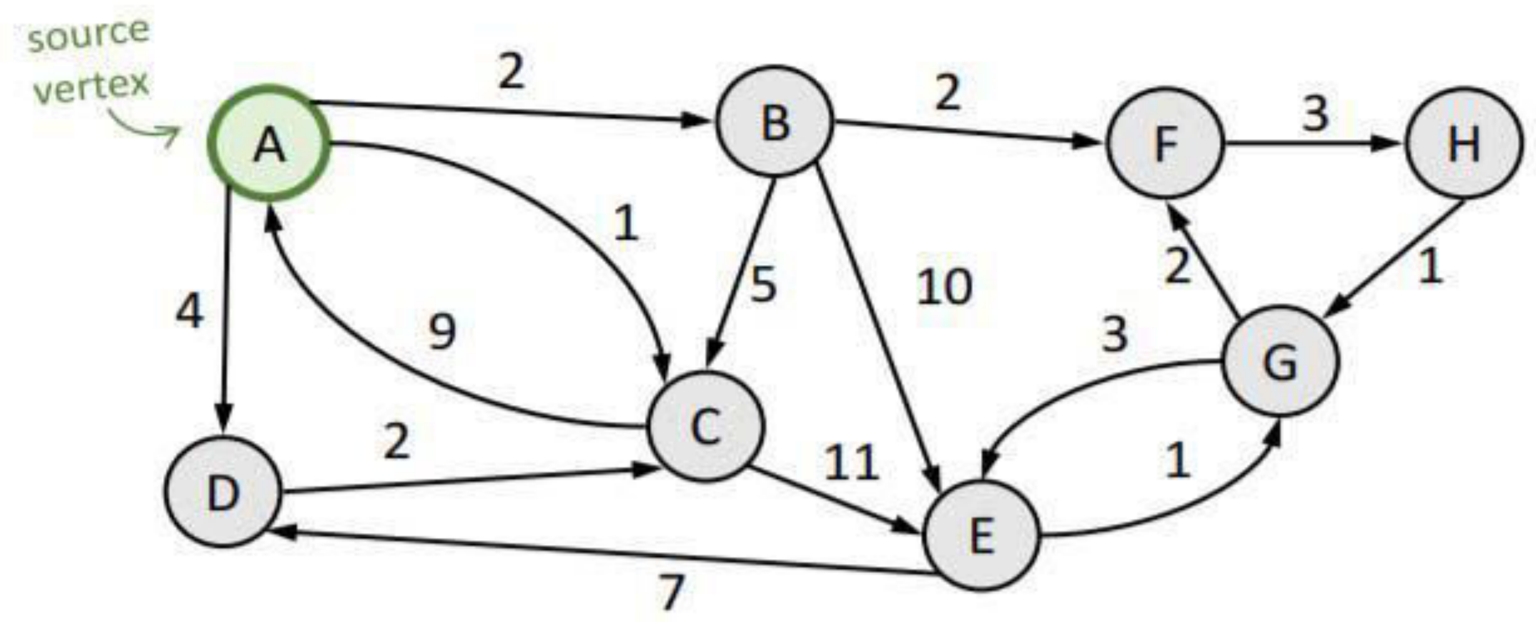
13. Consider the weights and values of items listed below. Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 Kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by V_{opt}. A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by V_{greedy}. Find the value of $V_{opt} - V_{greedy}$.



Item	I_1	I_2	I_3	I_4
W	10	7	4	2
V	60	28	20	24

AKTU Examination Questions

14. When do Dijkstra and the Bellman-Ford algorithm both fail to find a shortest path? Can Bellman ford detect all negative weight cycles in a graph? Apply Bellman Ford Algorithm on the following graph:



Divide and Conquer

Divide and Conquer Approach

- The divide-and-conquer paradigm involves three steps at each level of the recursion:
- **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
- **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub problems in a straightforward manner.
- **Combine** the solutions to the sub-problems into the solution for the original problem.

Divide and Conquer Approach

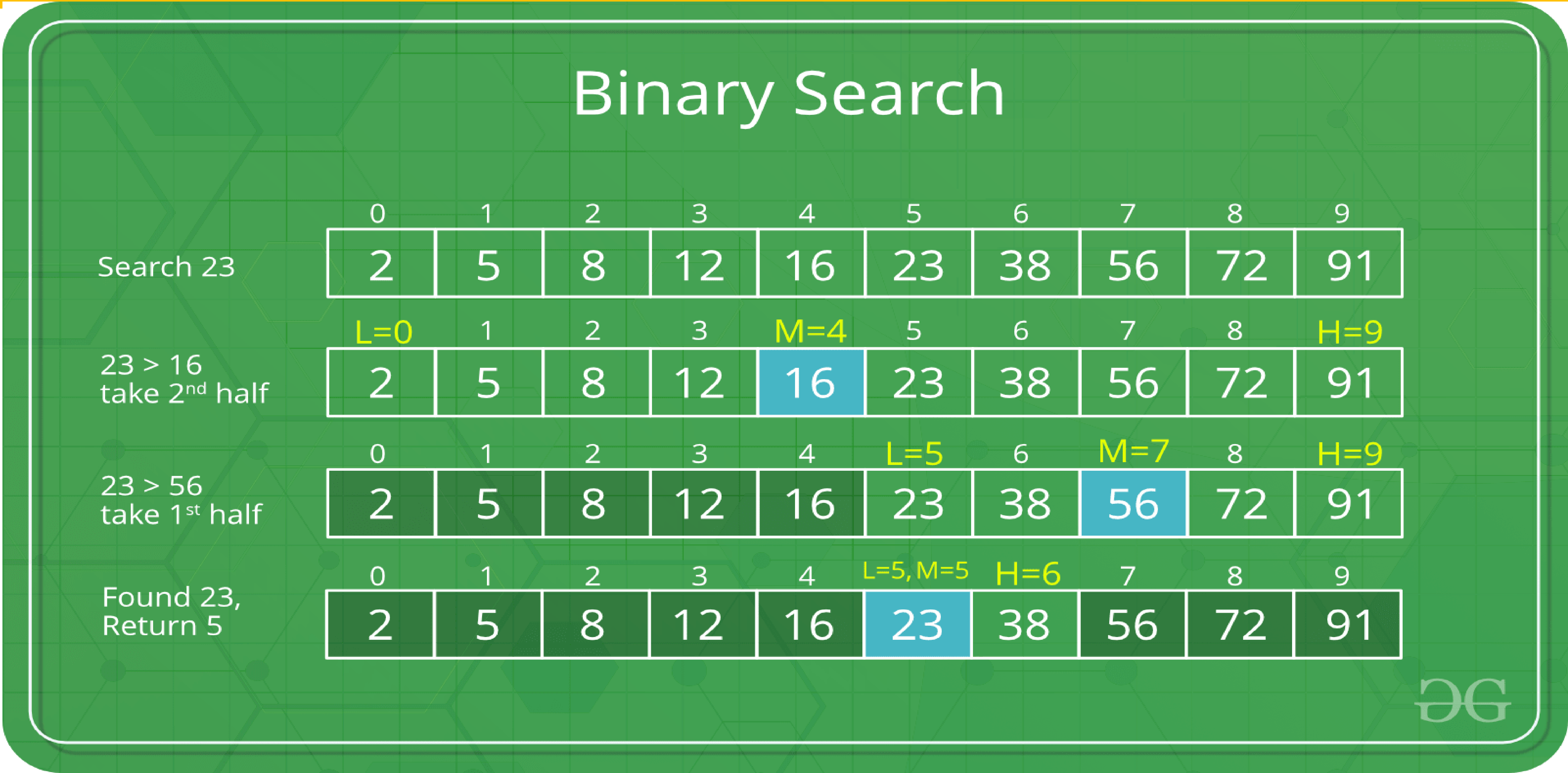
We will solve the following problems using divide and conquer approach :-

- Sorting
  - Merge sort
  - Quick sort
- Searching
  - Binary search
- Matrix Multiplication
- Convex Hull

Binary search

- ❖ **Binary search** is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems.
- ❖ Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be sorted.
- ❖ When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced on the basis of the value that is being searched.

Binary search process



Binary search algorithm

```
Binary-search(A, n, x)
l = 1
r = n
while $l \leq r$
do
        $m = \lfloor (l + r) / 2 \rfloor$
        if $A[m] < x$ then
                $l = m + 1$
        else if $A[m] > x$ then
                $r = m - 1$
        else
                return m
return unsuccessful
```

Time complexity **$T(n) = O(\lg n)$**

Matrix Multiplication (Divide and Conquer Method)

To multiply two matrices A and B of order nxn using **Divide and Conquer approach**, we use to multiply two matrices of order 2x2.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B= \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Where, A_{ij} and B_{ij} are $\frac{n}{2} \text{x} \frac{n}{2}$ matrices for i,j = 1,2.

Resultant matrix C will be

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

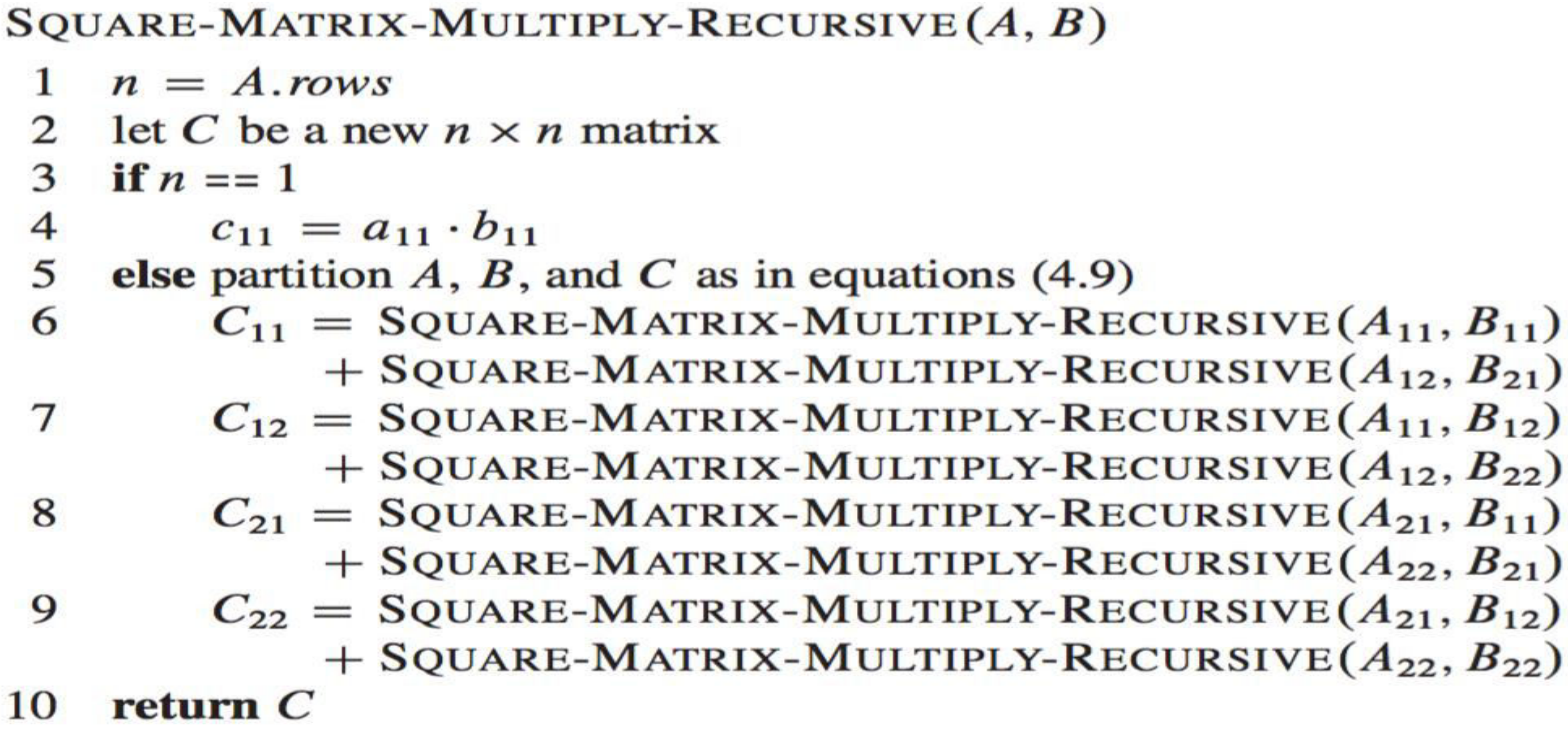
Where,

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$ $\qquad$ $C_{12}= A_{11}B_{21}+ A_{12}B_{22}$

$C_{21}= A_{21}B_{11}+ A_{22}B_{21}$ $\qquad$ $C_{22}= A_{21}B_{21}+ A_{22}B_{22}$

Matrix Multiplication

Clearly, computation of c_{ij} consists of two multiplications of two $\frac{n}{2} \times \frac{n}{2}$ matrices and one addition of two $\frac{n}{2} \times \frac{n}{2}$ matrices. Therefore, the algorithms for this is the following:-



```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    $c_{11} = a_{11} \cdot b_{11}$
else partition A, B, and C as in equations (4.9)
    $C_{11}$ = SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{11}$, $B_{11}$)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{12}$, $B_{21}$)
    $C_{12}$ = SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{11}$, $B_{12}$)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{12}$, $B_{22}$)
    $C_{21}$ = SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{21}$, $B_{11}$)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{22}$, $B_{21}$)
    $C_{22}$ = SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{21}$, $B_{12}$)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{22}$, $B_{22}$)
return C
```

Matrix Multiplication

Time complexity of this algorithm is computed as following:-

$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 8T(n/2) + \theta(n^2) & \text{if } n > 1 \end{cases}$$

After solving this recurrence relation, we get

$$\mathbf{T(n) = \theta(n^3)}$$

Strassen's Matrix Multiplication Algorithm

Strassen's algorithm has three steps:

1) Divide the input matrices A and B into $\frac{n}{2} \text{x} \frac{n}{2}$ sub-matrices.
2) Using the sub-matrices created from the step above, recursively compute seven matrix products P_1, P_2, … P_7. Each matrix P_i is of size $\frac{n}{2} \text{x} \frac{n}{2}$.

   $P1 = A_{11}(B_{12}-B_{22})$ $P2 = (A_{11}+A_{12})B_{22}$

   $P3 = (A_{21}+A_{22})B_{11}$ $P4 = A_{22}(B_{21}-B_{11})$

   $P5 = (A_{11}+A_{22})(B_{11}+B_{22})$ $P6 = (A_{12}-A_{22})(B_{21}+B_{22})$

   $P7 = (A_{11}-A_{21})(B_{11}+B_{12})$
3) Get the desired sub-matrices C_{11}, C_{12}, C_{21}, and C_{22} of the result matrix C by adding and subtracting various combinations of the P_i sub-matrices.

$C_{11} = P_5+P_4-P_2+P_6 C_{12}= P_1+P_2$

$C_{21}= P_3+P_4$ $C_{22}= P_1+P_5-P_3-P_7$

Strassen's Matrix Multiplication Algorithm

```
Strassen_Matrix_Multiplication(A, B, n)
    If n=1   then      return AxB.
    Else
        1.  Compute $A_{11}$, $B_{11}$, ………………., $A_{22}$, $B_{22}$.
        2.  $P_1$ ← Strassen_Matrix_Multiplication($A_{11}$, $B_{12}-B_{22}$, n/2)
        3.  $P_2$ ← Strassen_Matrix_Multiplication($A_{11}+A_{12}$, $B_{22}$, n/2)
        4.  $P_3$ ← Strassen_Matrix_Multiplication($A_{21}+A_{22}$, $B_{11}$, n/2)
        5.  $P_4$ ← Strassen_Matrix_Multiplication($A_{22}$, $B_{21}-B_{11}$, n/2)
        6.  $P_5$ ← Strassen_Matrix_Multiplication($A_{11}+A_{22}$, $B_{11}+B_{22}$, n/2)
        7.  $P_6$ ← Strassen_Matrix_Multiplication($A_{12}-A_{22}$, $B_{21}+B_{22}$, n/2)
        8.  $P_7$ ← Strassen_Matrix_Multiplication($A_{11}-A_{21}$, $B_{11}+B_{12}$, n/2)
        9.  $C_{11} = P_5+P_4-P_2+P_6$
        10. $C_{12} = P_1+P_2$
        11. $C_{21} = P_3+P_4$
        12. $C_{22} = P_1+P_5-P_3-P_7$
        13. return C
    End if
```

Strassen’s Matrix Multiplication Algorithm

Time complexity of this algorithm is computed as following:-

$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 7T(n/2) + \theta(n^2) & \text{if } n > 1 \end{cases}$$

After solving this recurrence relation, we get

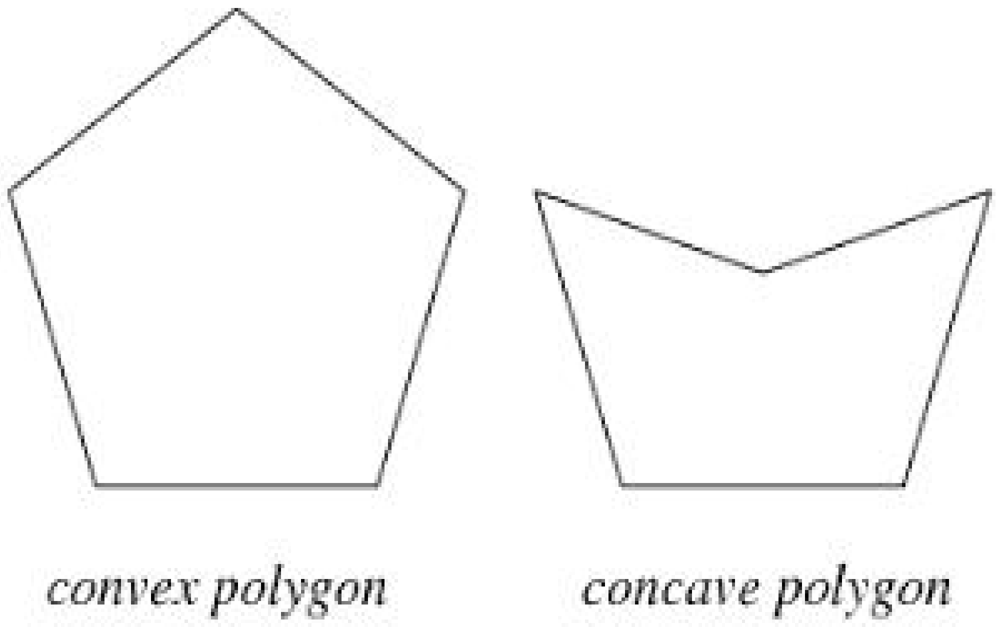
$$\mathbf{T(n) = \theta(n^{2.8})}$$

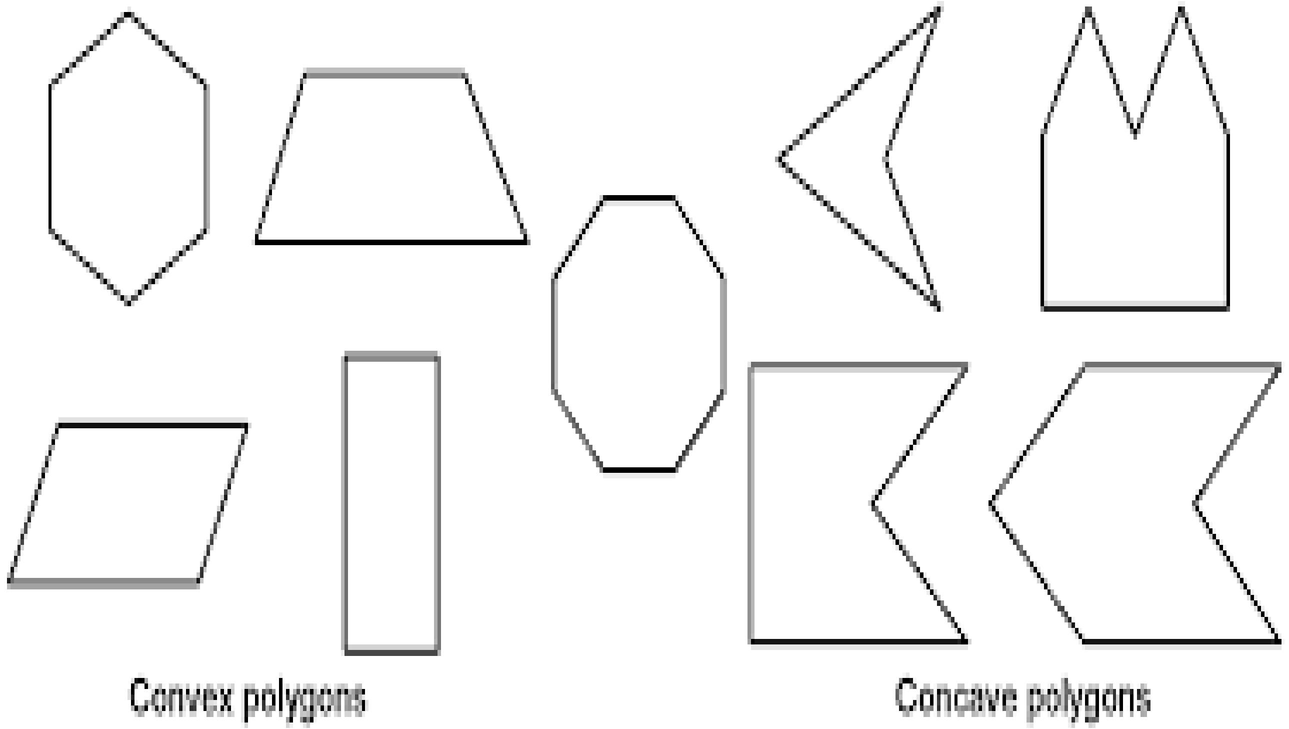
Convex Hull

Convex Polygon:

A polygon P is said to be convex polygon if for any two points p and q on the boundary of P, segment pq lies entirely inside P.

Example:

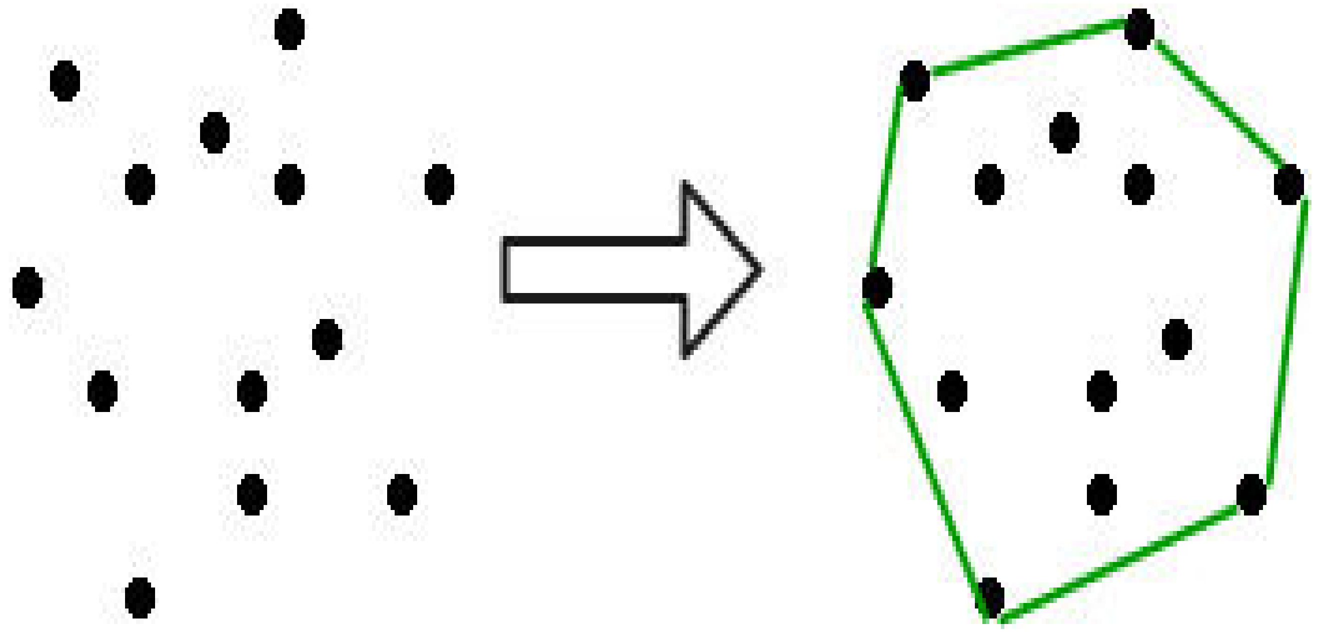




Convex Hull

Convex hull: Convex hull of a set Q of points is the smallest convex polygon P for which each points in Q is either on the boundary of P or in its interior.

Example:



Convex hull

Convex Hull

There are many algorithms for computing convex hull. But here, we shall study only two algorithms.

1. QuickHull
2. Divide and Conquer

QuickHull Algorithm

❖To compute the convex hull of a set X of n points in the plane, an algorithm is designed. This algorithm, called QuickHull, first identifies the two points (call them p_1 and p_2) of X with the smallest and largest x-coordinate values. Assume now that there are no ties. Both p_1 and p_2 are extreme points and part of the convex hull.

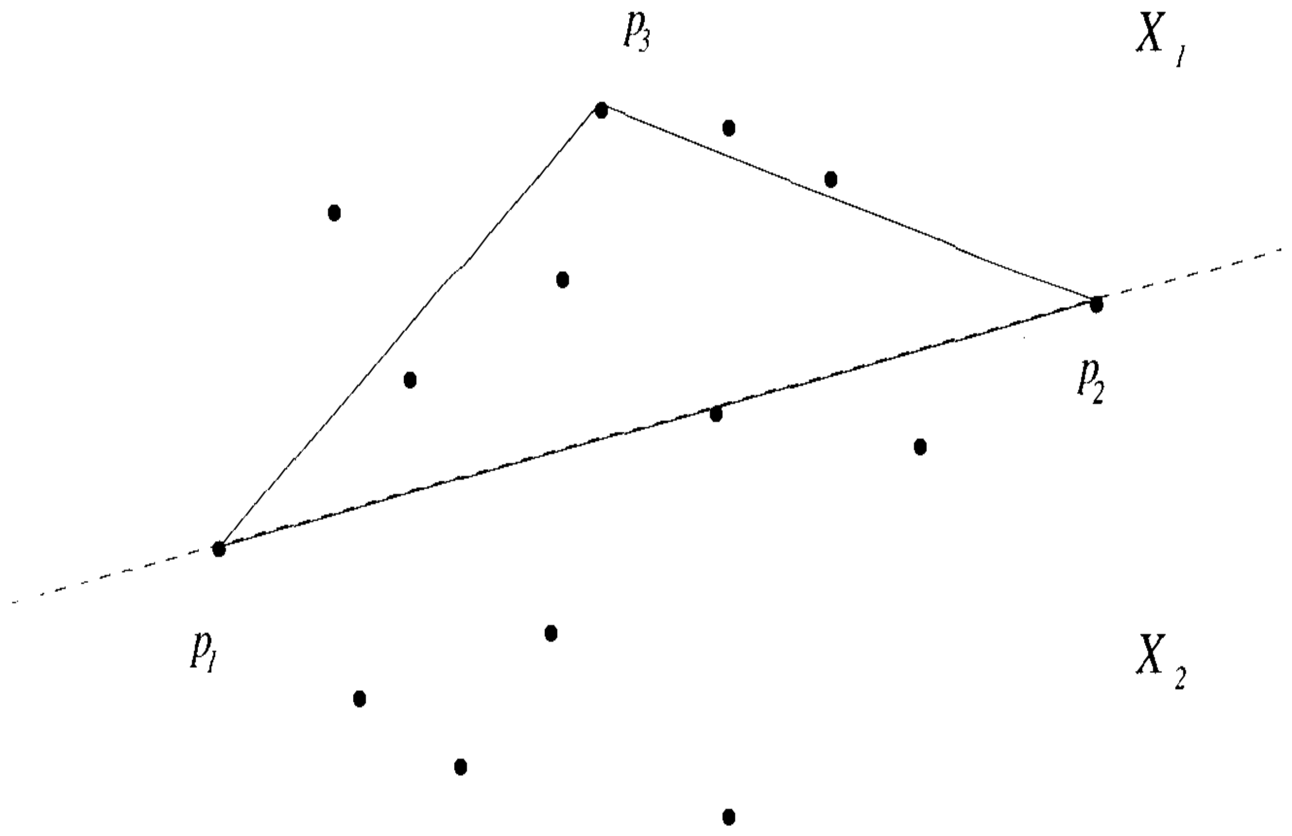
❖The set X is divided into X_1 and X_2 so that X_1 has all the points to the left of the line segment (p_1, p_2) and X_2 has all the points to the right of (p_1,p_2). Both X_1 and X_2 include the two points p_1 and p_2. Then, the convex hulls of X_1 and X_2 (called the upper hull and lower hull, respectively) are computed using a divide-and-conquer algorithm called QuickHull. The union of these two convex hulls is the overall convex hull.

QuickHull Algorithm

Computation of the convex hull of X_1

- ❖We determine a point of X_1 that belongs to the convex hull of X_1 and use it to partition the problem into two independent subproblems.
- ❖Such a point is obtained by computing the area formed by p_1 , p , and p_2 for each p in X_1 and picking the one with the largest (absolute) area.
- ❖Ties are broken by picking the point p for which the angle pp_1p_2 is maximum. Let p_3 be that point.
- ❖Now X_1 is divided into two parts; the first part contains all the points of X_1 that are to the left of (p_1,p_3) (including p_1 and p_3), and the second part contains all the points of X_1 that are to the left of (p_3, p_2) (including p_3 and p_2).
- ❖All the other points are interior points and can be dropped from future consideration.
- ❖The convex hull of each part is computed recursively, and the two convex hulls are merged easily by placing one next to the other in the right order.

QuickHull Algorithm



Running time of algorithm

T(n) = O(nlogn)

Where, n is the number of points in the set.

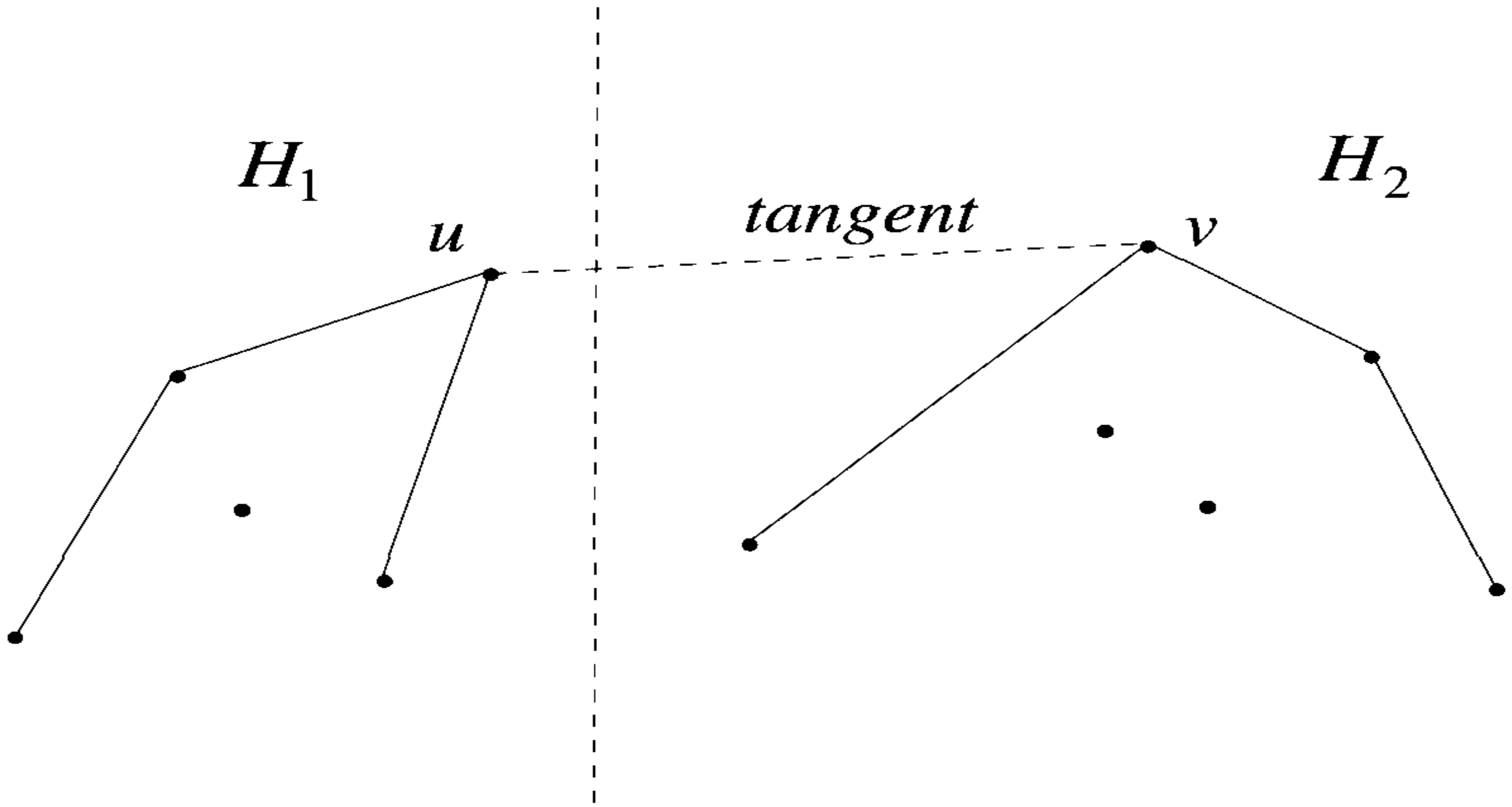
Identifying a point on the convex hull of X_1

Divide and Conquer Algorithm

This algorithm is called DCHull. This algorithm also takes O(nlogn) time. It computes the convex hull in clockwise order.

- ❖Given a set of n points, the problem is reduced to finding the upper hull and the lower hull separately and then putting them together. Since the computations of the upper and lower hulls are very similar, we restrict our discussion to computing the upper hull.
- ❖The divide-and-conquer algorithm for computing the upper hull partitions X into two nearly equal halves. Partitioning is done according to the x-coordinate values of points using the median x-coordinate as the splitter.
- ❖Upper hulls are recursively computed for the two halves. These two hulls are then merged by finding the line of tangent (i.e., a straight line connecting a point each from the two halves, such that all the points of X are on one side of the line).

Divide and Conquer Algorithm



Divide and Conquer Algorithm

- To begin with, the points p_1 and p_2 are identified [where p_1 (p_2) is the point with the least (largest) x-coordinate value].
- All the points that are to the left of the line segment (p_1, p_2) are separated from those that are to the right.
- Sort the input points according to their x-coordinate values.
- Let $q_1, q_2, \ldots \ldots, q_N$ be the sorted order of these points. Now partition the input into two equal halves with $q_1, q_2, \ldots \ldots, q_{N/2}$ in the first half and $q_{N/2+1}, q_{N/2+2}, \ldots \ldots, q_N$ in the second half.
- The upper hull of each half is computed recursively. Let H_1 and H_2 be the upper hulls. Upper hulls are maintained as linked lists in clockwise order.

Divide and Conquer Algorithm

- ❖The line of tangent is then found in O($\log^2 N$) time. If (u, v) is the line of tangent, then all the points of H_1 that are to the right of u are dropped.
- ❖Similarly, all the points that are to the left of v in H_2 are dropped. The
- ❖Remaining part of H_1, the line of tangent, and the remaining part of H_2 form the upper hull of the given input set.

- ❖If T(N) is the run time of the above recursive algorithm for the upper hull on an input of N points, then we have

$$T(N) = 2T(N/2) + \log^2 N$$

The solution of this recurrence relation is $T(N) = O(N)$.

- ❖But, the sorting of points takes O(N logN) time, therefore the running time of whole algorithm is O(N logN).

AKTU Examination Questions

1. Describe convex hull problem.
2. What do you mean by convex hull? Describe an algorithm that solves the convex hull problem. Find the time complexity of the algorithm.
3. Given an integer x and a positive number n, use divide & conquer approach to write a function that computes x^n with time complexity O (log n).