**Part A**

**Ex: 1**

**Title:** Program that uses simple technique to a) search a key b) to sort n elements**.**

**Problem Description:** Run the program for varied values of n>5000 and record the time taken to search/sort. Plot a graph of the time taken v/s n. The elements can be read from a file or can be generated using the random number generator.

**Method:** A Brute Force- Non recursive algorithm implementation. Sequential Search and Selection Sort.

**Theory Reference:** Module 1

***Algorithm***

## Sequential Search Algorithm (Linear Search)

**Input:** An array `a` of size `n`, and a key to search.
**Output:** The index of the key if found, otherwise -1.

**Steps:**

1. **Start**
2. Loop through each element of the array from `i = 0` to `i = n-1`:
   - If `a[i] == key`, return `i` (key found at index `i`).
3. If the loop completes without finding the key, return `-1` (key not found).
4. **End**

## Selection Sort Algorithm

**Input:** An array `a` of size `n`.
**Output:** The sorted array in ascending order.

**Steps:**

1. **Start**
2. Loop through each element from `i = 0` to `n-2`:
   - Assume `i` as the `min`.
   - Loop through the remaining unsorted part from `j = i+1` to `n-1`:
     - If `a[j] < a[min]`, update `min = j`.
   - Swap `a[i]` with `a[min]` (placing the smallest element at the correct position).
3. Repeat until the entire array is sorted.

4. **End**

**ALGORITHM** *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated below in Figure.

| | | | | | | |
|---|---|---|---|---|---|---|
| \| 89 | 45 | 68 | 90 | 29 | 34 | **17** |
| 17 \| | 45 | 68 | 90 | **29** | 34 | 89 |
| 17 | 29 \| | 68 | 90 | 45 | **34** | 89 |
| 17 | 29 | 34 \| | 90 | **45** | 68 | 89 |
| 17 | 29 | 34 | 45 \| | 90 | **68** | 89 |
| 17 | 29 | 34 | 45 | 68 \| | 90 | **89** |
| 17 | 29 | 34 | 45 | 68 | 89 \| | 90 |

Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

The analysis of selection sort is straightforward. The input's size is given by the number of elements n; the algorithm's basic operation is the key comparison A[j] < A[min]. The number of times it is executed depends only on the array's size

***Program***

```
import java.util.Random;
import java.util.Scanner;
public class SeqSort {
   static int searchcount=0;
   static int sortcount=0;
   public static int sequentialSearch(int[] a, int key) {
      int n = a.length;
      for (int i = 0; i < n; i++) {
         searchcount++;
         if (a[i] == key) {
            return i;
         }
      }
      return -1;
   }
   public static void selectionSort(int[] a) {
      int n = a.length;
      for (int i = 0; i < n - 1; i++) {
         int min = i;
         for (int j = i + 1; j < n; j++) {
            sortcount++;
            if (a[j] < a[min]) {
               min = j;
            }
```

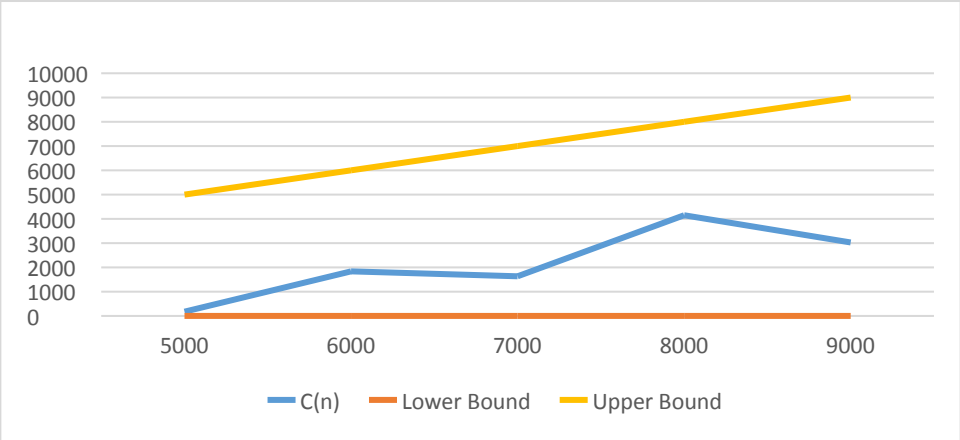
```
            }
            int temp = a[min];
            a[min] = a[i];
            a[i] = temp;
        }
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        Random r = new Random();
        System.out.print("Enter the number of elements (n > 5000): ");
        int n = s.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) {
            a[i] = r.nextInt(10000); // Random numbers between 0 and 9999
        }
        int rnd = r.nextInt(a.length);
        System.out.println("random number is:"+a[rnd]);
        System.out.print("Enter the key to search: ");
        int key = s.nextInt();
        int index = sequentialSearch(a, key);
        if (index != -1) {
            System.out.println("Key found at index: " + index);
        } else {
            System.out.println("Key not found");
        }
        System.out.println("Number of basic operations for searching is: "+searchcount);
        selectionSort(a);
        System.out.println("Sorted numbers are:");
```

```
        for(int i=0;i<5;i++)
            System.out.println(a[i]);

    System.out.println("Number of basic operations for sorting is: "+sortcount);
        }
}
```

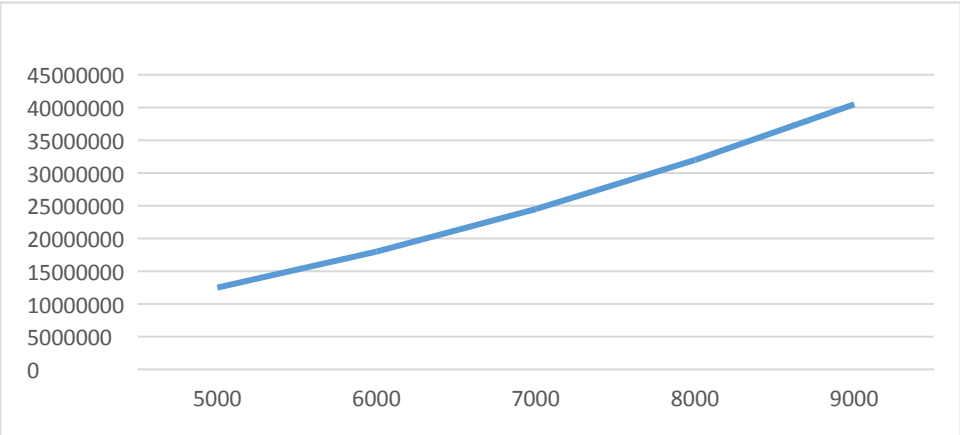
**Sequential Search**

| INPUT SIZE (N) | BASIC OPERATION COUNT | LOWE BOUND (BEST CASE 1) | UPPER BOUND (WORST CASE – n ) |
|---|---|---|---|
| 5000 | 176 | 1 | 5000 |
| 6000 | 1837 | 1 | 6000 |
| 7000 | 1634 | 1 | 7000 |
| 8000 | 4149 | 1 | 8000 |
| 9000 | 3028 | 1 | 9000 |




**Selection Sort**

| INPUT SIZE (N) | BASIC OPERATION COUNT - C(n) – depends only n |
|---|---|
| 5000 | 12497500 |
| 6000 | 17997000 |
| 7000 | 24496500 |
| 8000 | 31996000 |
| 9000 | 40495500 |



## Part A

**Ex: 3**

**Title:** Sorting elements in a list by breaking them into sub-lists.

**Problem Description:** Run the program for varied values of n>5000 and record the time taken to search/sort. Plot a graph of the time taken v/s n. The elements can be read from a file or can be generated using the random number generator.

**Method:** Run the program for varied values of n to demonstrate the behaviour of the algorithm in the Worst, Best and Average Cases. Record the time taken to sort. Plot a graph of the time taken versus *n* on graph sheet. The elements can be read from a file or can be generated using the random number generator for large values of n.

**Theory Reference:** Module 2

***Explanation***

Mergesort sorts a given array $A[0..n-1]$ by

- dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor ..n-1]$,
- sorting each of them recursively, and then
- merging the two smaller sorted arrays into a single sorted one.

The ***merging*** of two sorted arrays can be done as follows.

- Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
- The elements pointed to are compared, and the smaller of them is added to a new array being constructed;
  - the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
  - This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

***Algorithm***

**ALGORITHM** *Mergesort(*$A[0..n-1]$*)*

//Sorts array $A[0..n-1]$ by recursive mergesort

//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in nondecreasing order

**if** $n > 1$

copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$

copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$

Mergesort(B[0.. $\lfloor n/2 \rfloor - 1$])

Mergesort(C[0.. $\lceil n/2 \rceil - 1$])

Merge(B, C, A)

**ALGORITHM** Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])

//Merges two sorted arrays into one sorted array

//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted

//Output: Sorted array A[0..p + q − 1] of the elements of B and C

i ←0; j ←0; k←0

**while** i <p **and** j <q **do**

  **if** B[i]≤ C[j ]

    A[k]←B[i];

    i ←i + 1

  **else**

    A[k]←C[j ];

    j ←j + 1

  k←k + 1

**if** i = p

  copy C[j..q − 1] to A[k..p + q − 1]

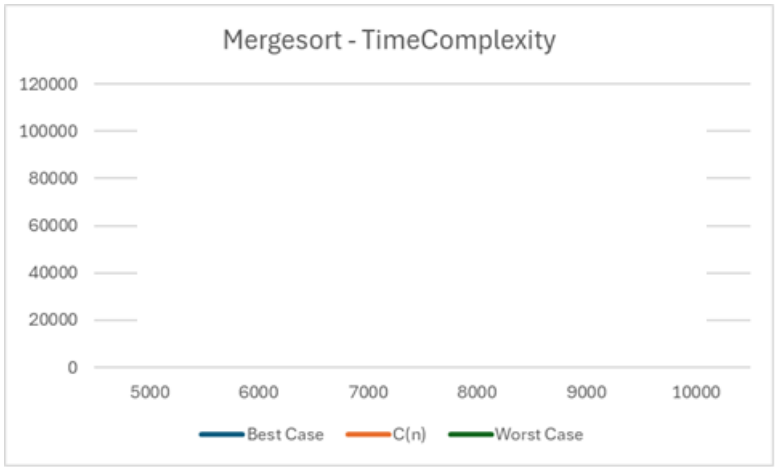
**else**

  copy B[i..p − 1] to A[k..p + q − 1]

***Program***

**Output:**

| INPUT SIZE (N) | LOWER BOUND (BEST CASE: n log n) | BASIC OPERATION COUNT | UPPER BOUND (WORST CASE: n log n ) |
|---|---|---|---|
| 5000 | | | |
| 6000 | | | |
| 7000 | | | |
| 8000 | | | |
| 9000 | | | |



## Part A

**Ex: 4**

**Title:** Program that performs Ordering of Directed Graphs

**Problem Description:** Program to obtain the Topological ordering of vertices in a given digraph with n vertices using source removal method

**Method:** A decrease and conquer approach to perform topological sort.

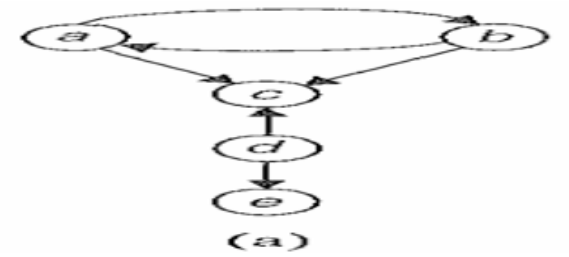
**Theory Reference:** Module 2

***Algorithm***

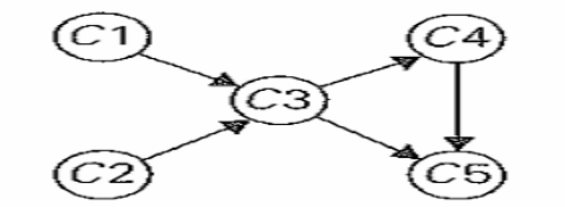
**Iterate to Remove Source Vertices**

- Create an array `removed[n]` initialized to `false` (0), tracking removed vertices.
- Repeat for `n` times:
    1. **Find a Source Vertex (In-degree 0)**
        - Find a vertex `j` such that:
            - `indegree[j] == 0` and
            - `removed[j] == false`
        - If no such vertex is found, **a cycle exists** → Stop and print "Graph is cyclic."
    2. **Print and Remove the Source Vertex**
        - Print vertex `j` (part of topological order).
        - Mark `removed[j] = true`.
        - Remove outgoing edges from vertex `j` by setting `cost[j][k] = 0` for all `k`.

***Output***




```
Enter number of vertices: 5
Enter the cost matrix:
0 1 1 0 0
1 0 1 0 0
0 0 0 0 0
0 0 1 0 1
0 0 0 0 0
Topological ordering is: 3 4 Graph is cyclic, so no solution.
BUILD SUCCESSFUL (total time: 53 seconds)
```




```
Enter number of vertices: 5
Enter the cost matrix:
0 0 1 0 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 1
0 0 0 0 0
Topological ordering is: 0 1 2 3 4 BUILD SUCCESSFUL (total time: 2 minutes 45 seconds)
```

## Part A

**Ex: 5**

**Title:** Urban planning of water supply networks

**Problem Description:** Design an optimized water distribution network that minimizes the total cost of the infrastructure while ensuring reliable water supply to all areas in the given region.

**Method:** Represent the problem in the form of a graph. Use the greedy Technique for a spanning tree which has a total minimum weight.

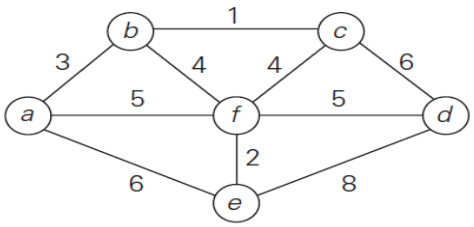
**Theory Reference: Module 3**

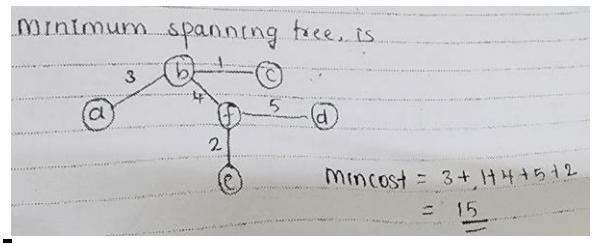
***Algorithm***

**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
$E_T \leftarrow \varnothing$
**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** $E_T$

***Output***



```
run:
Enter the no. of Cities
6
Enter the cost of laying pipelines between the two cities
0 3 0 0 6 5
3 0 1  0 0 4
0 1 0 6 0 4
0 0 6 0 8 5
6 0 0 8 0 2
5 4 4 5 2 0
Enter the source city
0
Pipeline from city 0 to city 1 and the cost 3 units
Pipeline from city 1 to city 2 and the cost 1 units
Pipeline from city 1 to city 5 and the cost 4 units
Pipeline from city 5 to city 4 and the cost 2 units
Pipeline from city 5 to city 3 and the cost 5 units
Minimum cost of urban water supply networks 15 units
BUILD SUCCESSFUL (total time: 1 minute 10 seconds)
```

**Part A**

**Ex: 6**

**Title:** Maximizing Profit/Value while adhering to a capacity constraint.

**Problem Description:** The key principle in this approach is to prioritize items based on their value per unit weight rather than their absolute value or weight alone. Suppose, a thief breaks into a jewelry store and wants to steal the most valuable items while carrying a limited amount of weight. The thief has a knapsack that can hold up to 50 kg. The store has different items, each with a weight and a value. The thief can take fractions of items if necessary.

| Item | Weight (kg) | Value ($) |
|---|---|---|
| Gold Necklace | 10 | 60 |
| Silver Bracelet | 20 | 100 |
| Diamond Ring | 30 | 120 |

**Method:** Use the greedy Technique for solving the fractional knapsack problem.

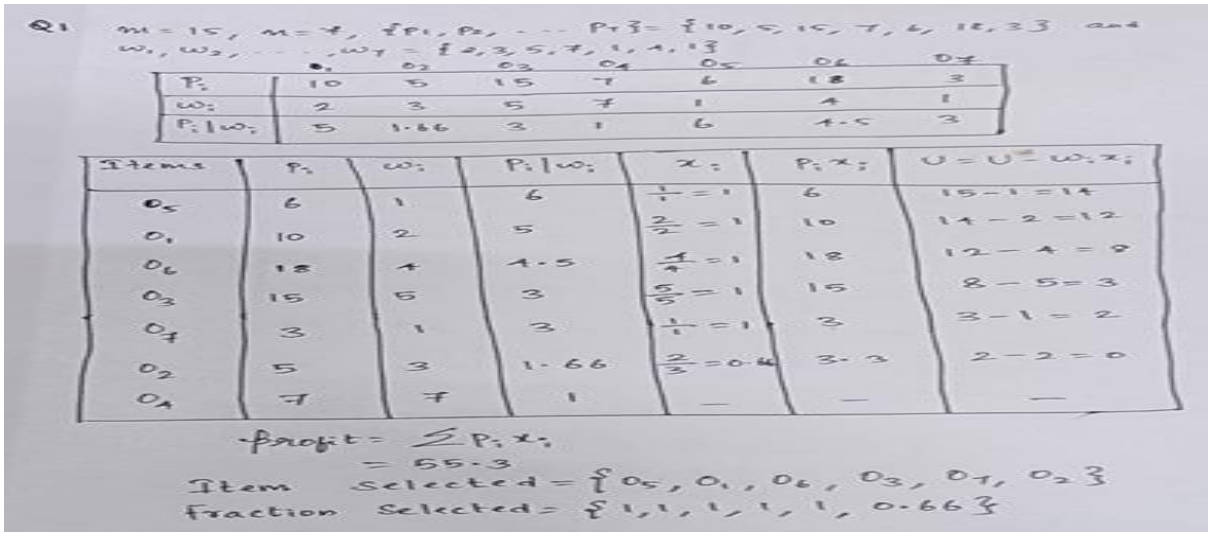
**Theory Reference: Module 2**

***Algorithm***

```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
    for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
    float U = m;
    for (i=1; i<=n; i++) {
       if (w[i] > U) break;
       x[i] = 1.0;
       U -= w[i];
    }
    if (i <= n) x[i] = U/w[i];
}
```

***Output***

```
run:
Enter the no. of items
3
Enter the weights of n items
10 20 30
Enter the profits of n items
60 100 120
Enter the capacity of Knapsack
50
item included is :
1 item is selected
fraction is 1
2 item is selected
fraction is 1
3 item selected
fraction selected is 0.6666667
Knapsack profit = 240.0
BUILD SUCCESSFUL (total time: 50 seconds)
```




```
run:
Enter the no. of items
7
Enter the weights of n items
2 3 5 7 1 4 1
Enter the profits of n items
10 5 15 7 6 18 3
Enter the capacity of Knapsack
15
item included is :
5 item is selected
fraction is 1
1 item is selected
fraction is 1
6 item is selected
fraction is 1
3 item is selected
fraction is 1
7 item is selected
fraction is 1
2 item selected
fraction selected is 0.6666667
Knapsack profit = 55.333332
BUILD SUCCESSFUL (total time: 48 seconds)
```

**Part A**

**Ex: 6**

**Title:** Optimal Road Network

**Problem Description:** A local government wants to improve the road network between towns to enhance connectivity and reduce travel costs. The region consists of five towns that need to be connected by the most efficient road network possible, minimizing the total length of the roads while ensuring that each town is accessible from any other town.

**Method:** A greedy Technique for a spanning tree which has a total minimum weight. Use Union-Find Data Structure to detect cycles during the execution of the algorithm.

**Theory Reference:** Module 3

***Algorithm***

1. **Initialize Parent Array:**

- Create an array `par[]` of size `n` and initialize all elements to `-1`, representing that each node is initially its own set.

2. **Initialize Variables:**

- `mincost` to store the total cost of the Minimum Spanning Tree (MST).
- `ne` (number of edges in MST), initially `0`.
- `a, b, u, v` to store the nodes of the minimum edge.

3. **Repeat Until the MST Contains `n-1` Edges:**

- Find the minimum cost edge in `cost[][]`.
- Store the indices `a` and `b` of the minimum edge.
- Store `u` and `v` (representative nodes of `a` and `b` using the Find operation).

4. **Find the Parent (Find Operation):**

- Use path traversal to find the representative (root) of `u`.
- Use path traversal to find the representative (root) of `v`.

5. **Check for Cyclicity:**

- If `u` and `v` are different, it means adding this edge won't form a cycle.
    - Print the edge and its cost.
    - Update the total minimum cost `mincost`.

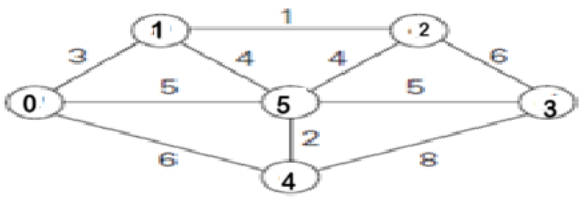
- Perform the **Union Operation**: Merge the sets by updating `par[v] = u` (or vice versa).
- Increment `ne` (edge count in MST).

6. **Mark the Edge as Processed:**

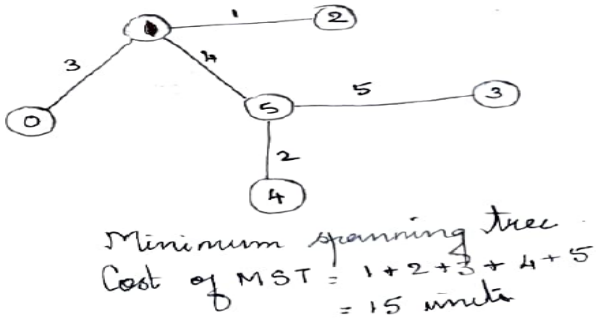
- Set `cost[a][b] = cost[b][a] = Integer.MAX_VALUE` to avoid selecting it again.

7. **Output the Total Cost of the MST.**

***Output***




Edges arranged in ascending order of weights: 1-2, 4-5, 0-1, 1-5, 2-5, 0-5, 3-5, 0-4, 2-3, 3-4.



```
run:
Enter the no. of towns
6
Enter the cost of laying roads between two towns in lakhs
0 3 0 0 6 5
3 0 1 0 0 4
0 1 0 6 0 7
0 0 6 0 8 5
6 0 0 8 0 2
5 4 4 5 2 0
Cost of Laying road from town  1 to town  2 is 1 Lakhs
Cost of Laying road from town  4 to town  5 is 2 Lakhs
Cost of Laying road from town  0 to town  1 is 3 Lakhs
Cost of Laying road from town  1 to town  5 is 4 Lakhs
Cost of Laying road from town  3 to town  5 is 5 Lakhs
Cost of Laying entire Road Network =15 Lakhs
BUILD SUCCESSFUL (total time: 8 seconds)
```

**Part A**

**Ex: 8**

**Title:** Optimal Road Routes

**Problem Description:** You are the chief planner for a transportation department in a country with several cities. The cities are connected by a network of roads, each with a specific travel time (in hours). Your task is to determine the shortest travel time from a designated source city to all other cities.

**Method:** Represent the problem in the form of a graph using adjacency matrix or adjacency list. Using the Single Source Shortest Path Algorithm find the shortest path from the source city to all other cities.

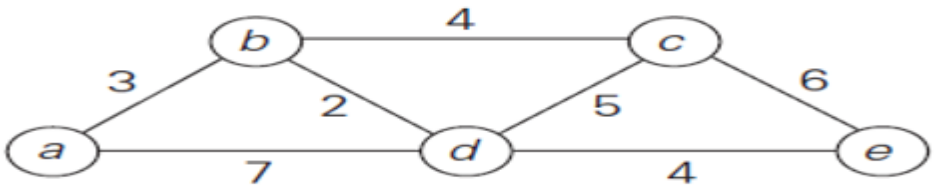
**Theory Reference:** Module 3

***Algorithm***

```
ALGORITHM Dijkstra(G, s)
    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
    //        and its vertex s
    //Output: The length d_v of a shortest path from s to v
                                          for every vertex v in V
    Initialize(Q)  //initialize priority queue to empty
    for every vertex v in V
        d_v ← ∞;   p_v ← null
        Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
    d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
    V_T ← ∅
    for i ← 0 to |V| − 1 do
        u* ← DeleteMin(Q)  //delete the minimum priority element
        V_T ← V_T ∪ {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 ← d_u* + w(u*, u);   p_u ← u*
                Decrease(Q, u, d_u)
```

***Output***

1.




0 3 0 7 0

3 0 4 2 0

0 4 0 5 6

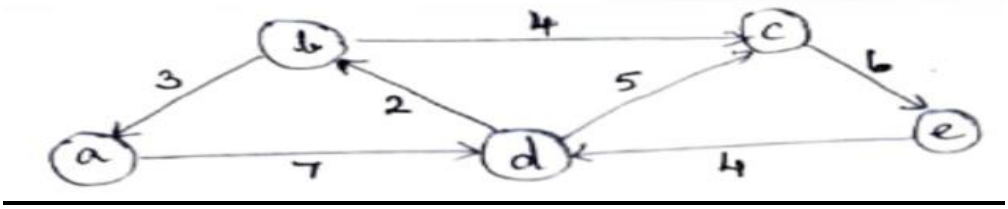
7 2 5 0 4

0 0 6 4 0

```
run:
Enter the no.of cities
5
Enter the travel time in hours between every two cities
0 3 0 7 0
3 0 4 2 0
0 4 0 5 6
7 2 5 0 4
0 0 6 4 0

Enter the source city
0
 The shortest travel time
from 0 city to 0 city is 0
from 0 city to 1 city is 3
from 0 city to 2 city is 7
from 0 city to 3 city is 5
from 0 city to 4 city is 9
BUILD SUCCESSFUL (total time: 20 seconds)
```

2.




0 0 0 7 0

3 0 4 0 0

0 0 0 0 6

0 2 5 0 0

0 0 0 4 0

```
run:
Enter the no.of cities
5
Enter the travel time in hours between every two cities
0 0 0 7 0
3 0 4 0 0
0 0 0 0 6
0 2 5 0 0
0 0 0 4 0

Enter the source city
0
 The shortest travel time
from 0 city to 0 city is 0
from 0 city to 1 city is 9
from 0 city to 2 city is 12
from 0 city to 3 city is 7
from 0 city to 4 city is 18
BUILD SUCCESSFUL (total time: 23 seconds)
```

**Part A**

**Ex: 9**

**Title:** Metropolitan Subway System Optimization

**Problem Description:** A metropolitan city is planning to optimize its subway system to ensure efficient travel for its commuters. The subway system consists of various stations connected by subway lines, and each line has a different travel time. The city's transportation authority wants to find the shortest travel time between all pairs of stations.

**Method:** Use Dynamic programming technique to find the shortest travel time between all pairs of stations using the Floyd- Warshall algorithm

**Theory Reference:** Module 3

***Algorithm***

**ALGORITHM** *Floyd*$(W[1..n, 1..n])$

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

***Output***

```
run:
Enter no. of Stations
4
Enter the travel time between subway lines
0 0 3 0
2 0 0 0
0 7 0 1
6 0 0 0

Shortest Travel Time between all stations
0   10   3   4
2   0   5   6
7   7   0   1
6   16   9   0
BUILD SUCCESSFUL (total time: 16 seconds)
```

**Part B**

**Ex: 10**

**Title:** Optimal Product Selection for a Limited Shelf Space

**Problem Description:** You are a store manager responsible for stocking products on the shelves. The store has limited shelf space, and you want to maximize the total profit by selecting the most valuable products to display. Each product has a specific weight and profit associated with it.

**Method:** Solve the 0/1 Knapsack Problem using dynamic programming or other optimization techniques. The key idea is to build a table that stores the maximum value achievable for different weights and items.

**Theory Reference:** Module 4

***Algorithm***

ALGORITHM DPKnapsack(n,m,p,w)

//function: solves knapsack problem using dynamic programming

//Input: A nonnegative integer 'n' and 'm', where n indicates the number of items and m the knapsack capacity with their profits & weight.

//Output: Optimal profit and items selected.

```
for i←0 to n do
   for j←0 to m do
        if i=0 or j=0 then
                V[i,j] ←0
        else if j<W[i] then
                V[i,j] ←V[i-1,j]
        else
                V[i,j] ←max(V[i-1,j],p[i]+V[i-1,j-W[i]])
 return V
```

***Output***

```
run:
Enter the no. of products
4
Enter the weights of n products
2 1 3 2
Enter the profits of n products
12 10 20 15
Enter the capacity of shelf(Knapsack)
5
0   0   0   0   0   0
0   0   12   12   12   12
0   10   12   22   22   22
0   10   12   22   30   32
0   10   15   25   30   37
OPTIMAL PROFIT IS:37
Products selected for shelf that yields maximum profit are:4  2  1  BUILD SUCCESSFUL (total time: 16 seconds)
```

## Part B

**Ex: 11**

**Title:** Event Budget Allocation

**Problem Description:** An event planner is organizing a conference and has a fixed budget to allocate across various categories such as venue, catering, speakers, and marketing. The planner needs to ensure that the total expenditure does not exceed the budget while maximizing the quality of the event. Set of Expenses (with costs): Venue rental-$5000, Catering $2000, Keynote Speaker-$3000, Marketing-$1500, Audio/Visual Equipment-$1000. Given Fixed budget-$8000

**Method:** Use backtracking technique to solve the problem.

**Theory Reference:** Module 4

***Algorithm***

1) **Mark Current Element as Selected**:

   Set `x[k] = 1`.

2) **Check if Adding w[k] to Current Sum Matches Target**:

   If `s + w[k] == d`, then:

   - A valid subset is found.
   - Increment and print the subset count.
   - Print all elements `i` where `x[i] == 1` (i.e., selected elements and their categories).

3) **Left Subtree (Include Current Element)**:

   If `k + 1` is a valid index AND `s + w[k] + w[k+1] <= d`, then:

   - Recur: `sum_of_subsets(s + w[k], k + 1, rem - w[k])`
   - This explores the case where the current item is included.

4) **Right Subtree (Exclude Current Element)**:

   If `k + 1` is a valid index AND `(s + rem - w[k] >= d)` AND `s + w[k+1] <= d`, then:

   - Set `x[k] = 0` (exclude current element).
   - Recur: `sum_of_subsets(s, k + 1, rem - w[k])`
   - This explores the case where the current item is excluded.

***Output***

```
run:
Enter no of categories
5
Enter the category names followed by their fixed budgets (in increasing order of budget):
Category 1 name: Audio/visual equipment
Category 1 budget: 1000
Category 2 name: Marketing
Category 2 budget: 1500
Category 3 name: Catering
Category 3 budget: 2000
Category 4 name: Speaker
Category 4 budget: 3000
Category 5 name: Venue Rental
Category 5 budget: 5000
Enter the budget  to host the event
8000
subset = 1
Audio/visual equipment : 1000
Catering : 2000
Venue Rental : 5000

subset = 2
Speaker : 3000
Venue Rental : 5000

BUILD SUCCESSFUL (total time: 2 minutes 30 seconds)
```

## Part B

**Ex: 12**

**Title:** Delivery Route Optimization.

**Problem Description:** Imagine a delivery company that operates in a city with five key delivery locations. The company needs to design a route for their delivery vehicle that visits each location exactly once and returns to the starting point without retracing its steps.

**Method:** Given a graph representing the locations as vertices and the roads between them as edges, determine whether a Hamiltonian Circuit exists. If it does, find the circuit.

**Theory Reference:** Module 4

***Algorithm***

**Step 1: Initialize a temporary array `temp`** to store the current path during recursion.

**Step 2: Check base case:**

- If all nodes are visited (`visited == n`) and there is an edge from the current node (`st`) back to the `source`, then:
  - A Hamiltonian circuit is found.
  - Call the function `path(succ)` to process or print this path.
  - Return from the function.

**Step 3: Recursive step:**

- For every vertex `i` from `0` to `n-1`:
  - If there is an edge from the current node `st` to `i` (`a[st][i] == 1`) **and** `i` has not been visited yet (`succ[i] == 0`):
    - Assign `i` as the successor of `st` in the path: `succ[st] = i`
    - Copy the `succ` array to `temp` to preserve path state
    - Recursively call `ckt(i, temp, visited + 1)` to continue the path from node `i`

***Output***

```
run:
Enter the no. of delivery locations in a city/vertices
6
Enter the roads between delivery locations/adjacency matrix
0 1 1 1 0 0
1 0 1 0 0 1
1 1 0 1 1 0
1 0 1 0 1 0
0 0 1 1 0 1
0 1 0 0 1 0

Routes for delivery vehicle/Hamiltonian circuits
1->2->6->5->3->4->1
1->2->6->5->4->3->1
1->3->2->6->5->4->1
1->3->4->5->6->2->1
1->4->3->5->6->2->1
1->4->5->6->2->3->1
BUILD SUCCESSFUL (total time: 13 seconds)
```