MANIPAL INSTITUTE OF TECHNOLOGY
Manipal – 576 104

DEPARTMENT OF INFORMATION & COMMUNICATION TECHNOLOGY

CERTIFICATE

This is to certify that Ms./Mr. …………………...…………………………………… Reg. No. …..…………………… Section: …. Roll No: ………… has satisfactorily completed the lab exercises prescribed for Algorithm Lab [ICT-2263] of Fourth Semester B. Tech. (IT/CCE) Degree at MIT, Manipal, in the academic year 2019-2020.

Date: ……...................................

Signature	Signature
Faculty in Charge	Head of the Department

CONTENTS

Course Objectives

- To implement basic algorithm designing techniques.
- To understand basic approximation algorithm.
- To apply the appropriate designing technique for a given problem.

Course Outcomes

- Implement an algorithm to find path between any two vertices in the given graph.
- Apply the knowledge of shortest path algorithms for real world problems.
- Implement Greedy, Divide and Conquer, Dynamic Programming, Back tracking and Branch and Bound techniques to solve different problems.
- Implement approximation algorithm for travelling sales person and vertex cover problem.

Evaluation Plan

- Internal Assessment Marks : 60%
  - ✓ Continuous evaluation component (biweekly):10 marks
  - ✓ The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce
- End semester assessment of 2 hour duration: 40%

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the lab manual and the required stationery to every lab session
2. Be in time and follow the institution dress code
3. Must sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

- Follow the instructions on the allotted exercises
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the lab manual
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
    - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
    - Programs should perform input validation (data type, range error, etc.) and give appropriate error messages and suggest corrective actions.

- Comments should be used to give the statement of the problem and every function should indicate the purpose of the function, inputs and outputs.
    - Statements within the program should be properly indented.
    - Use meaningful names for variables and functions.
    - Make use of constants and type definitions wherever needed.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
    - Solved exercise
    - Lab exercises : to be completed during lab hours
    - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- If a student misses a lab class then he/she must ensure that the experiment is completed during the repetition class in case of genuine reason (medical certificate approved by HOD) with the permission of the faculty concerned
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.
- A sample note preparation is given as a model for observation.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

LAB NO: 1 **Date:**

SEARCHING AND SORTING

Objectives:

1. Understand Algorithm and Algorithm Design Techniques
2. Apply the technique to searching and sorting using step count method
3. Analyze the complexity for different input sizes

1. <u>Algorithm and Algorithm Analysis</u>

Algorithm

Algorithm is a sequence of unambiguous instructions for solving a problem i.e for obtaining required output for any legitimate input in a finite amount of time.

It is procedural solution to problem

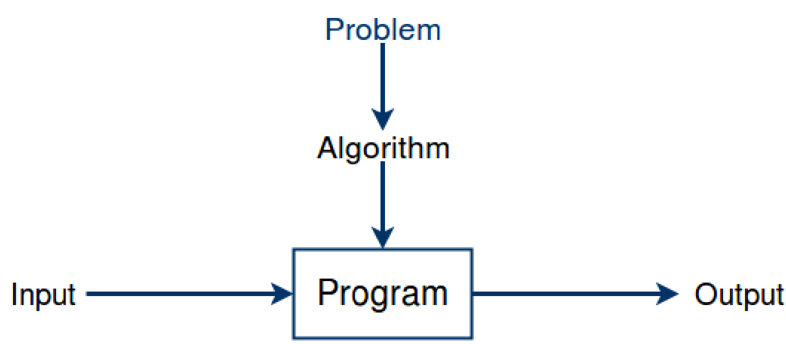


Figure 1: Notion of an Algorithm

2. Algorithm Analysis

Algorithm design technique is a general approach to solving problems algorithmically that is applicable to variety of problems from different areas of computing

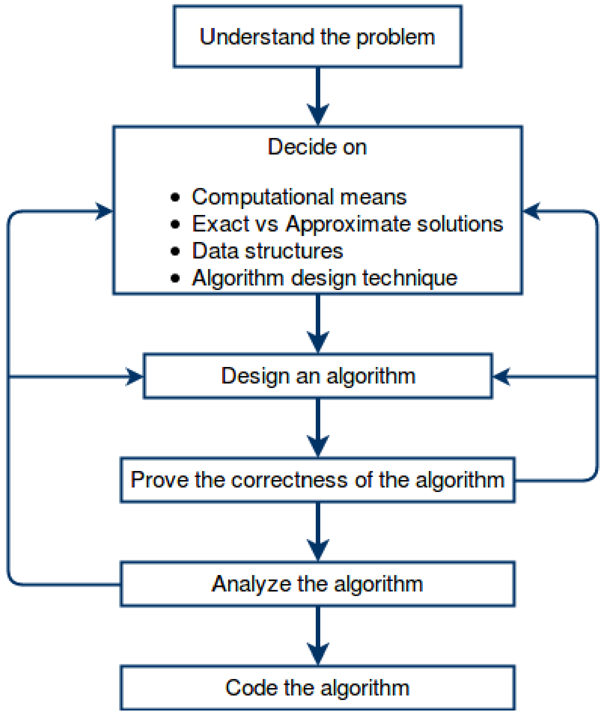


Figure 2: Algorithmic Problem Solving

3. Performance of a Program

Performance analysis of an algorithm depends upon two factors i.e. amount of memory used and amount of compute time consumed on any CPU. Formally they are notified as complexities in terms of:

- Space Complexity
- Time Complexity

Space Complexity of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination. Space needed by any program is the sum of following components:

Fixed Component: This is independent of the characteristics of the inputs and outputs. This part includes: Instruction Space, Space of simple variables, fixed size component variables, and constants variables.

Variable Component: This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like Linked list, heap, trees, graphs etc.

Therefore the total space requirement of any algorithm 'A' can be provided as

Space(A) = Fixed Components(A) + Variable Components(A)

Time complexity: Time Complexity of an algorithm(basically when converted to program) is the amount of computer time it needs to run to completion. The time taken by a program is the sum of the compile time and the run/execution time . The compile time is independent of the instance(problem specific) characteristics. following factors effect the time complexity:

- Characteristics of compiler used to compile the program.
- Computer Machine on which the program is executed and physically clocked
- Multiuser execution system
- Number of program steps.

The number of steps is the most prominent instance characteristics. The number of steps any program statement is assigned depends on the kind of statement like comments count as zero steps, an assignment statement which does not involve any calls to other algorithm is counted as one step, for iterative statements the steps count is considered only for the control part of the statement etc.

Therefore to calculate total number of program steps we use following procedure. For this we build a table in which we list the total number of steps contributed by each statement. This is arrived at by first determining the number of steps per execution of the statement and the frequency of each statement executed. This procedure is explained using Solved Exercise 1.

Solved Exercises:

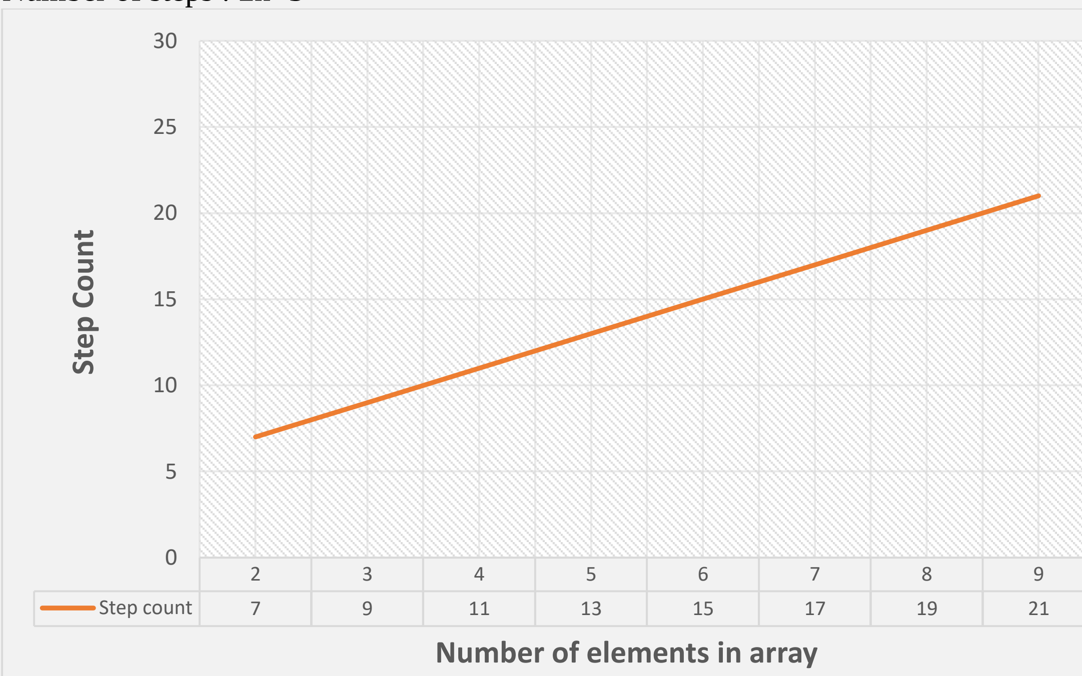
1. **To print step count of sum function (to add elements of an array) and plot number of elements and step counts**

```
#include <iostream>
using namespace std;
int count;

int sum(int arr[], int n)
{
    int i,s=0;
    count++;
    for (i = 0; i < n; i++)
    {
        count++;
        count++;
        s=s+arr[i];

    }
    count++;
    count++;
    return s;
}

int main(void)
{
    int arr[25], n, x;
    count=0;
    cout<<"enter no. of elements";
    cin>>n;
    cout<<"enter "<< n <<" elements";
    for(int i=0;i<n;i++)
    cin>>arr[i];
    int result = sum(arr, n);
    cout<<"Sum of elements "<<result;
    cout<<endl;
    cout<<"Number of steps for sum function "<<count;
   return 0;
}
enter no. of elements 5
enter 5 elements1 2 3 4 5
Sum of elements 15
```

Number of steps for sum function 13

enter no. of elements 8
enter 8 elements 11 12 33 21 43 22 43 67
Sum of elements 252
Number of steps for sum function 19

Number of steps : 2n+3



2. To print the time taken by the sum function

```
#include <iostream>
#include <chrono>
using namespace std;
using namespace std::chrono;
int count;
int sum(int arr[], int n)
{

    int i,s=0;
    count++;
    for (i = 0; i < n; i++)
    {
        count++;
```

```
        count++;
        s=s+arr[i];

    }
    count++;
    count++;

    return s;
}

int main(void)
{
    int arr[1000], n, x;
    count=0;
    cout<<"enter no. of elements";
    cin>>n;
    //cout<<"enter "<< n <<" elements";
    for(int i=0;i<n;i++)
    arr[i]=i;
    auto start = high_resolution_clock::now();
    int result = sum(arr, n);
    auto stop = high_resolution_clock::now();
    cout<<"Sum of elements "<<result;
    cout<<endl;
    auto duration = duration_cast<microseconds>(stop - start);
    cout << "time taken to run sum fucntion:  "<<duration.count() << endl;
    cout<<"Number of steps for sum function "<<count << microseconds;
   return 0;
}
```

Output

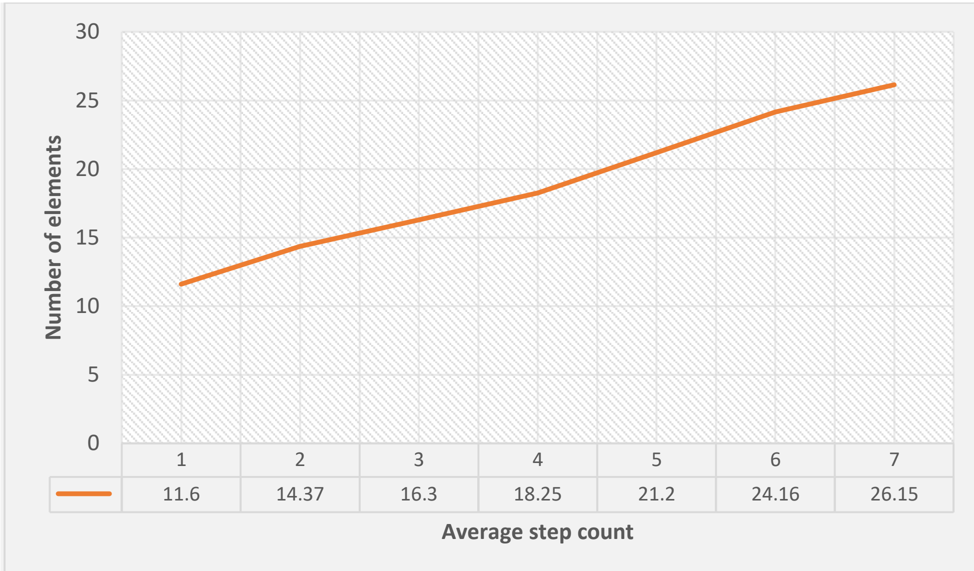
```
enter no. of elements500
Sum of elements 124750
time taken to run sum function:  2 microseconds
Number of steps for sum function 1003
```

3. To print step counts of linear search function and plot average step counts

```
#include <iostream>
using namespace std;
int count;

int search(int arr[], int n, int x)
{
    int i;
    count++;
    for (i = 0; i < n; i++)
    {
        count++;
        count++;
        if (arr[i] == x)
          {
           count++;
            return i;
           }
     }
    count++;
    count++;
    return -1;
}

int main(void)
{
    int arr[25], n, x;
    count=0;
    cout<<"enter no. of elements";
    cin>>n;
    cout<<"enter "<< n <<" elements";
    for(int i=0;i<n;i++)
    cin>>arr[i];
    cout<<"enter the element to be searched";
    cin>> x;
    int result = search(arr, n, x);
   (result == -1)? cout<<"Element is not present in array"
                : cout<<"Element is present at index " <<result;
    cout<<endl;
    cout<<"Number of seteps for search function "<<count;
   return 0;
```

```
}
enter no. of elements 6
enter 6 elements 1 3 2 5 4 6
enter the element to be searched 3
Element is present at index 1
Number of steps for search function 6

enter no. of elements 6
enter 6 elements 1 3  2 5 4 6
enter the element to be searched 4
Element is present at index 4
Number of steps for search function 12

enter no. of elements 6
enter 6 elements 1 3 2 5 4 6
enter the element to be searched 10
Element is not present in array
Number of steps for search function 15

enter no. of elements 6
enter 6 elements 1 3 2 5 4 6
enter the element to be searched56
Element is not present in array
Number of steps for search function 15
```

Number of steps :
If lement found in the ith position : 2(i)+2
If element not found : 2n+3
Hence total number of intsnces : n+1
Average case complexity = $\frac{\sum_{i=1}^{n}(2i+2)+2n+3}{n+1}$



Lab Exercises

Write a Program to perform the following and find the time complexity using step count method

1. Binary Search (both iterative and recursive)
2. Bubble Sort
3. Selection Sort
4. Insertion Sort

Additional Exercises

1. Find the substring in a given string without using String handling functions.
2. Implement rank sort algorithm.

LAB NO: 2 **Date:**

GRAPHS-I

Objectives:

1. Understand the graph representations
2. Understand Breadth First Search (BFS) and Depth-First Search (DFS), graph traversal algorithms.
3. Apply the BFS and DFS to find sequence and analyze the complexities based on the graph representaions

1. Graph Representations

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.

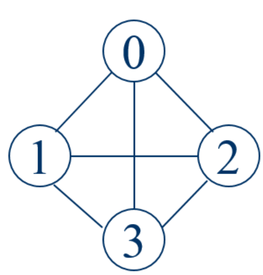
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Following two are the most commonly used representations of a graph.

1. Adjacency Matrix

2. Adjacency List

Adjacency Matrix:

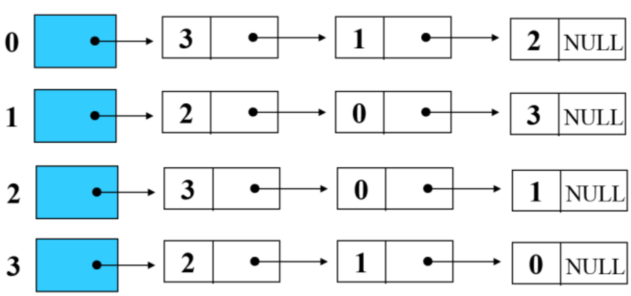


Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

Adjacency List:

An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is adjacency list representation of the above graph.



2. Breadth First Search

.B x.readth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors.

Input: A graph and a starting vertex root of Graph

Output: All vertices reachable from root labeled as explored.

Example:

BFS will visit the sibling vertices before the child vertices using this algorithm:

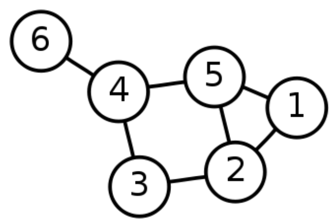
Mark the starting node of the graph as visited and enqueue it into the queue

While the queue is not empty

  Dequeue the next node from the queue to become the current node

  While there is an unvisited child of the current node

    Mark the child as visited and enqueue the child node into the queue



The queue operation enqueue adds to the left and dequeue removes from the right.

Action	Current Node	Queue	Unvisited Nodes	Visited Nodes
Start with node 1		1	2, 3, 4, 5, 6	1
Dequeue node 1	1		2, 3, 4, 5, 6	1
Node 1 has unvisited children nodes 2 and 5	1		2, 3, 4, 5, 6	1
Mark 2 as visited and enqueue into queue	1	2	3, 4, 5, 6	1, 2
Mark 5 as visited and enqueue into queue	1	5, 2	3, 4, 6	1, 2, 5
Node 1 has no more unvisited children, dequeue a new current node 2	2	5	3, 4, 6	1, 2, 5
Mark 3 as visited and enqueue into queue	2	3, 5	4, 6	1, 2, 5, 3
Node 2 has no more unvisited children, dequeue a new current node 5	5	3	4, 6	1, 2, 5, 3
Mark 4 as visited and enqueue into queue	5	4, 3	6	1, 2, 5, 3, 4
Node 5 has no more unvisited children, dequeue a new current node 3	3	4	6	1, 2, 5, 3, 4
Node 3 has no more unvisited children, dequeue a new current node 4	4		6	1, 2, 5, 3, 4
Mark 6 as visited and enqueue into queue	4	6		1, 2, 5, 3, 4, 6

2. Depth-First Search :

Considering a given node as the parent and connected nodes as children, DFS will visit the child vertices before visiting siblings using this algorithm.

Mark the starting node of the graph as visited and push it onto the stack

While the stack is not empty

Peek at top node on the stack (look at the top element on the stack, but do not remove it)

If there is an unvisited child of that node

    Mark the child as visited and push the child node onto the stack

Else

    Pop the top node off the stack

Action	Stack	Unvisited Nodes	Visited Nodes
Start with node 1	1	2, 3, 4, 5, 6	1
Peek at the stack Node 1 has unvisited child nodes 2 and 5	1	2, 3, 4, 5, 6	1
Mark node 2 visited	1, 2	3, 4, 5, 6	1, 2
Peek at the stack Node 2 has unvisited child nodes 3 and 5	1, 2	3, 4, 5, 6	1, 2
Mark node 3 visited	1, 2, 3	4, 5, 6	1, 2, 3
Peek at the stack Node 3 has unvisited child node 4	1, 2, 3	4, 5, 6	1, 2, 3
Mark node 4 visited	1, 2, 3, 4	5, 6	1, 2, 3, 4
Peek at the stack Node 4 has unvisited child node 5	1, 2, 3, 4	5, 6	1, 2, 3, 4
Mark node 5 visited	1, 2, 3, 4, 5	6	1, 2, 3, 4, 5
Peek at the stack Node 5 has no unvisited children	1, 2, 3, 4, 5	6	1, 2, 3, 4, 5
Pop node 5 off stack	1, 2, 3, 4	6	1, 2, 3, 4, 5
Peek at the stack Node 4 has unvisited child node 6	1, 2, 3, 4	6	1, 2, 3, 4, 5
Mark node 6 visited	1, 2, 3, 4, 6		1, 2, 3, 4, 5, 6

Solved Exercise: Representing graph using adjacency matrix and printing indegree of vertices

```
#include <iostream>
using namespace std;
int count;
int indegree(int arr[][10], int p, int n)
{
   int c1=0;

   for (int i = 1; i <= n; i++)
   {
      if(arr[i][p]==1)
      c1++;
   }

   return c1;
}

int main(void)
{
   int A[10][10], n, m,x;
   count=0;
   cout<<"enter no. of vertices";
   cin>>n;
   cout<<"enter number of edges";
   cin>>m;
   for(int i=1;i<=n;i++)
   for(int j=1;j<=n;j++)
   A[i][j]=0;
   int p,q;

   for(int i=1;i<=m;i++)
   {
     cout<<"enter Source";
     cin>> p;
     cout<<"enter destination ";
     cin>>q;
     A[p][q]=1;
   }

   for(p=1;p<=n;p++)
```

```
    {
    int result = indegree(A,p,n);
    cout<<endl;
    cout<<"Indegree of "<< p <<" is:" << result;
     }
    return 0;
}

Output
enter no. of vertices4
enter number of edges5
enter Source 1
enter destination  3
enter Source1
enter destination 4
enter Source3
enter destination 4
enter Source4
enter destination 2
enter Source2
enter destination 1

Indegree of 1 is:1
Indegree of 2 is:1
Indegree of 3 is:1
Indegree of 4 is:2
```

Lab Exercises

Write Programs to perform the following and find the time complexity using step count method:

1. Depth First Search (DFS)
2. Breadth First Search (BFS)
3. To find mother vertex in a graph
4. To find transpose of a given graph

Additional Exercises

Write a Program to input a graph from the user (nodes should be given numbers 1, 2,.. upto n) and perform the following:

1. Display the nodes with even number using BFS.
2. Display the nodes with odd number using DFS.

LAB NO: 3 **Date:**

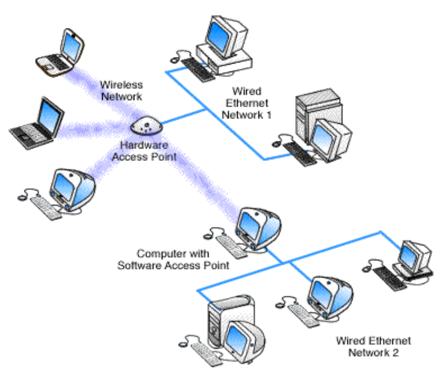
GRAPHS-II

Objectives:

1. Apply graph traversal algorithms to find the path, check the graph connected or not etc.
2. Analyze the time complexity.

1. Applications

In a network of computers it is possible to find whether one machine is communicating with another machine or not. The network of computers can be represented by a graph, with each vertex representing one of the machines in the network and each edge representing a communication wire between two machines. Either a BFS or a DFS can be used to determine whether a path exists between two vertices any two computers ‘u’ and ‘v’.



Lab Exercises

Write a Program to perform the following and find the time complexity using step count method:

1. Finding a path in the graph
2. Finding a cycle in the graph
3. Check whether the given graph is connected or not.

Additional Exercises

1. Write a program to find the components of the given graph also analyze the complexity.

2. Write a program to find the strongly Connected Components of a graph also analyze the complexity.

LAB NO: 4 **Date:**

GREEDY- I

Objectives:

1. Understand optimization problem and Greedy Technique
2. Apply the technique to container loading, find the topological sequence etc..
3. Analyze the time complexity

1. Optimization Problem

In an Optimization problem, we are given a set of constraints and an optimization function. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called an optimal solution.

2. Greedy Technique

A greedy technique, as the name suggests, always makes the choice that seems to be the best at that moment. In greedy algorithm approach, decisions are made from the given solution domain. A decision made in one stage is not changed in later state, so each decision should assure feasibility.

The criterion used to make the greedy decision at each stage is called the greedy criterion.

Some of the applications of this technique are finding the topological sequence, container loading problem etc.,

Solved Exercise:

Machine Scheduling Problem

Given n tasks and an infinite supply of machines on which these tasks can be performed. Each task has start time S_i and finish time F_i, $S_i < F_i$. [S_i , F_i] is the processing interval for task i. Two tasks i & j overlap iff their processing intervals overlap at a point other than start or finish time. Schedule the tasks using minimum number of machines.

Greedy Criterion:

If an old machine becomes available by the start time of the task to be assigned, assign the task to this machine; if not, assign it to a new machine.

```
#include <iostream>
using namespace std;
#include <bits/stdc++.h>
using namespace std;

class Task
{        public: int start, finish;   };

bool Comparetasks(Task t1, Task t2)
{
         return (t1.start < t2.start);  }

void printtasks(Task arr[], int n)
{
         sort(arr, arr+n, Comparetasks);
          int i = 0;
         bool selected[n];
         for(i=0;i<n;i++)
                  selected[i]=false;
         int k=0,j;
         for(i=0;i<n;i++)
         {
              if(selected[i])
                   continue;
                cout<<"=========================================";
              cout <<"\n"<<" Task Allocation : For Machine "<<k+1<<"\n";
             cout << "(" <<arr[i].start<< ", " <<arr[i].finish<<")"<<"\n";

             for (j = i+1; j < n; j++)
                 {
                    if(selected[j])
                       continue;
```

```
                    if (arr[i].finish<= arr[j].start)
                  {
                              i = j;
                        selected[j]=1;
                        cout << "(" <<arr[j].start<<","<<arr[j].finish<<")"<<"\n";
                  }
                        }
          //k++;
          i=++k;
         cout<<"========================================"; }
  cout<<"\n"<<"Total Machines Used "<<k;  }

int main()
{
      Task arr[10];
      int n;
      cout<<"Enter the number of Total Number Tasks";
      cin>>n;
      cout<<"Enter Start and Finish time of each machine";
      for(int i=0;i<n;i++)
      cin>>arr[i].start>>arr[i].finish;
      printtasks(arr, n);
      return 0;
}
```

<u>Output</u>

```
Enter the number of Total Number Tasks7
Enter Start and Finish time of each machine
9 11
7  10
6  8
4  7
3  7
2 5
1 2
========================================
 Task Allocation : For Machine 1
(1, 2)
```

(2,5)
(6,8)
(9,11)
==
 Task Allocation : For Machine 2
(3, 7)
(7,10)
==
 Task Allocation : For Machine 3
(4, 7)
==
Total Machines Used 3

Time Complexity :
Time taken is O(n log n) if the tasks are not in sorted order.

Lab Exercises

Write a Program to perform the following and find the time complexity using step count method:

1. Container Loading Problem
2. 0/1 Knapsack Problem
3. Topological Sorting

Additional Exercises

1. To Check whether a given graph is bipartite or not.
2. To find the minimal cover of a Bipartite graph
3. Implement 0/1 Knapsack problem using profit by density method.

LAB NO: 5 **Date:**

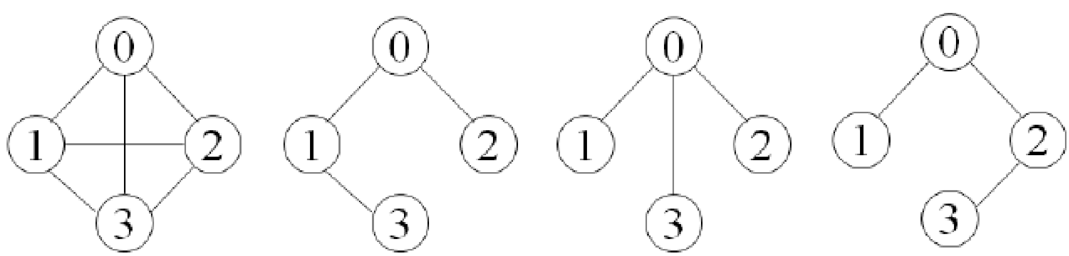
GREEDY - II

Objectives:

1. To Understand spanning tree
2. Apply greedy technique to find the minimum cost spanning tree and shortest path from single source
3. Analyze the time complexity for the same

1. Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.



G_1 Possible spanning trees

Minimum Cost Spanning Tree

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. Following algorithms are used find the minimum cost spanning tree.

A. Prims’s
B. Kruskal;s
C. Sollin’s

Application: For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols. “Shortest” can be in time, distance, cost, etc. It can be applied to numerous applications like Map navigation, Flight itineraries, Circuit wiring, Network routing.

Greedy criterion:

From the remaining edges, select a least cost edge that doesn't result in a cycle when added to a set of already selected edges.

Solved Exercise

Find the minimum cost spanning tree using Kruskal’s algorithm.

```
#include <bits/stdc++.h>
using namespace std;

#define V 6
#define Infinity 5000
int parent[V];

// Find set of vertex i
int find(int i)
{
	while (parent[i] != i)
		i = parent[i];
	return i;
}

// Does union of i and j. It returns  false if i and j are already in the same set.
void union1(int i, int j)
{
	int a = find(i);
	int b = find(j);
	parent[a] = b; }
```

```
// Finds MST using Kruskal's algorithm
void kruskals(int cost[][V])
{
	int mincost = 0; // Cost of min MST.

	// Initialize sets of disjoint sets.
	for (int i = 0; i < V; i++)
		parent[i] = i;

	// Include minimum weight edges one by one
	int edge_count = 0;
	cout<<"Edges Selected and corresponding cost"<<":"<<"\n";
	while (edge_count < V - 1) {
		int min = Infinity, a = -1, b = -1;
		for (int i = 0; i < V; i++) {
			for (int j = 0; j < V; j++) {
				if (find(i) != find(j) && cost[i][j] < min) {
					min = cost[i][j];
					a = i;
					b = j;
						}
					}
		}
	union1(a, b);
	cout<<"("<<a+1<<','<<b+1<<")"<<"\t\t"<<min<<"\n";
	edge_count++;
		mincost += min;
	}
	cout<<"Minimum cost="<<mincost; }
int main()
   {
		int cost[][V] = {
		  { Infinity, 6,1,5,Infinity,Infinity },
		  { 6,Infinity,5, Infinity, 3, Infinity },
		  { 1,5,Infinity, 5,6,4 },
		  { 5,Infinity, 5,Infinity,Infinity,2 },
		  { Infinity, 3,6,Infinity,Infinity,6 },
		  { Infinity, Infinity, 4,2,6,Infinity },
```

```
        };

        // Print the solution
        kruskals(cost);

        return 0;
}



Output:
Edge :(1,3)Cost:1
Edge :(4,6)Cost:2
Edge :(2,5)Cost:3
Edge :(3,6)Cost:4
Edge :(2,3)Cost:5

 Minimum cost= 15
```

Time Complexity

Time complexity is O(V+E). Where V and E represents the number vertices and edges of the given graph respectively.

Lab Exercises

1. Write a Program to find the shortest path from a given vertex to other vertices in a given directed weighted connected using Dijkstra's Algorithm and also analyze the complexity.

2. Write a program to find the minimum cost spanning tree using Prim's algorithm and also analyze the complexity.

Additional Exercises

1. Write a program to find the maximum cost spanning tree for the given connected weighed graph.
2. Write a program to find the minimum cost spanning tree using Sollin's algorithm and also analyze the complexity.
3. Implement Huffman tree construction algorithm.

LAB NO:6 **Date:**

DIVIDE AND CONQUER - I

Objectives:

1. Understand the divide and conquer algorithm design technique
2. Apply this technique to sort the elements of the list
3. Analyze the complexity

1. Basic Concepts :

Divide & conquer is a general algorithm design strategy with a general plan as follows:

- DIVIDE: A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
- RECUR: Solve the sub-problem recursively.
- CONQUER: If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

2. General divide & conquer recurrence:

An instance of size “n” can be divided into “b” instances of size n/b, with “a" of them needing to be solved. [$a \geq 1$; $b \geq 1$]

Assume size n is a power of b. The recurrence for the running time T(n) is as follows:

$T(n) = aT(n/b) + f(n)$

Where f (n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions

Therefore, the order of growth of T(n) depends on the values of the constants a & b and the order of growth of the function f (n).

Solved Exercise

Search an element using in a sorted array using binary search, divide and conquer technique.

```
#include <iostream>
using namespace std;

int binSearch(int arr[], int l, int r, int x)
{
	while (l <= r) {
		int m = l + (r - l) / 2;

		if (arr[m] == x) //Mid element
			return m;  // Element Present
		if (arr[m] < x)  // Search Right half
			l = m + 1;
		else //Search Left half
			r = m - 1;
	}
	return -1; // not found
}

int main(void)
{
   int n,ele,arr[20];
	cout<< "Enter the total number of elements";
	cin>>n;
	cout<<"Enter the elements in ascending order";
	for(int i=0;i<n;i++)
	   cin>>arr[i];
	cout<<"Enter the element to search";
	cin>>ele;
	int result = binSearch(arr, 0, n - 1, ele);
	    if(result == -1){
	       cout << "Element is not present in array";
```

```
            exit(0);}

                        cout << "Element is present at index " << result+1;
        return 0;
}
```

Output

Enter the total number of elements8
Enter the elements 1 6 7 9 10 12 16 18
Enter the element to search 16
Element is present at index 7

Complexity Analysis
Recurrence Relation: $T(n) = T(n/2) + c$
Hence the complexity is (Log n).

Auxiliary Space: O(1) in case of iterative implementation.

Lab Exercises

Write a program to sort the elements of an array using the following and analyze the complexity.

1. Quick Sort

2. Merge Sort

Additional Exercises

1. Given set of strings, find the longest common prefix using Divide and Conquer given set of strings.
2. Write a program to compute a^b using divide and conquer technique and analyze the complexity.

LAB NO: 7 **Date:**

DIVIDE AND CONQUER - II

Objectives:

1. Apply the divide and conquer technique find the maximum and minimum of an array, multiplication of matrices
2. Analyze the time complexity for different input sizes.

Solved Exercise

Program to find the maximum element of an array using divide and conquer technique.

```
#include <iostream>
using namespace std;
        bool search(int mat [][4], int rowBegin, int rowEnd, int colBegin, int colEnd, int
key)
        {
                // Compare with middle
                bool flag=false;
                int r1,c1;
                int i = rowBegin + (rowEnd-rowBegin )/2;
                int j = colBegin + (colEnd-colBegin )/2;
                if (mat[i][j] == key) // If key is present at middle
                  { r1=i;  c1=j;
                   flag=true;}
                else
                {
                        if (i!=rowEnd || j!=colBegin)
                        search(mat,rowBegin,i,j,colEnd,key);

                        if (rowBegin == rowEnd && colBegin + 1 == colEnd)
                        if (mat[rowBegin][colEnd] == key)
                        {
                                r1=rowBegin;  c1=colEnd;
                                flag=true;}

                       // If middle element is less than key
```

```
            /                  //then search lower horizontal matrix and right hand side matrix
                                   if (mat[i][j] < key)
                                   { // search lower horizontal if such matrix exists
                                               if (i+1<=rowEnd)
                                               search(mat, i+1, rowEnd, colBegin, colEnd, key);
                                   }

                                   // If middle element is > key then search left vertical
                                   // matrix and right hand side matrix
                                   else {
                                           if (j-1>=colBegin)
                                               search(mat, rowBegin, rowEnd, colBegin, j-1, key); }
                                           }

                               if(flag)  {
                                                 cout<<"Found "<< key<<" at "<<r1 << " " << c1;
                                                exit(0); }
                                return false;       }
int main() {
    int mat[][4] = { {5, 10, 15, 20}, {12, 16, 20, 24}, {14, 19, 22, 25}, {16, 23, 40, 60}};
      int rows = 4,cols=4,key,r;
     cout<<"Enter the Key Element to search";
     cin>>key;


eturn 0;
            }




Enter the Key Element to search5
Found 5 at 0 0

Enter the Key Element to search4 23
Found 23 at 3 1
Enter the Key Element to search50
Key not found
```

Lab Exercises

1. Write a Program to multiply two matrices using Strassen's method and analyze the time complexity.

2. Write a program find the maximum and minimum of an array using divide and conquer technique.

Additional Exercises

1. Write a program to merge K sorted arrays using divide and conquer technique.

2. Write a program to find the closest pair of points in a given plane and analyze the time complexity.

3. Write a program to find the Maximum Subarray Sum of all sub arrays s using Divide and Conquer technique.

   Example: Input : arr[] = {1, 2, 3}

   Output : 14

   Max of all sub-arrays: {1} - 1
   {1, 2} - 2
   {1, 2, 3} - 3
   {2} - 2
   {2,3} - 3
   {3} - 3
   1 + 2 + 3 + 2 + 3 + 3 = 14

LAB NO: 8 **Date:**

DYNAMIC PROGRAMMING

Objective:

1. Understand the Dynamic Programming algorithm design method
2. Apply the same to find the optimal packing of Knapsack, find the optimal order of multiplication etc.
3. Analyze the time complexity

1. Dynamic Programming

Dynamic programming (usually referred to as DP) is a very powerful technique to solve a particular class of problems. A DP is an algorithmic technique which is usually based on a recurrent formula and one (or some) starting states. A sub-solution of the problem is constructed from previously found ones. DP solutions have a polynomial complexity. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, if you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again. Shortly '*Remember your Past*'. If the given problem can be broken up in to smaller sub-problems and these smaller subproblems are in turn divided in to still-smaller ones, and in this process, if you observe some over-lappping subproblems, then it’s a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem.

It uses bottom-up approach i.e. analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as Dynamic Programming.

Solved Exercise

Generation of Fibonacci Series using Dynamic Programming

```
#include<stdio.h>
#include <bits/stdc++.h>
using namespace std;
void fib_series(int n)
{
int fib[n+2], i;
fib[0] = 0;  // 1st number of the series
fib[1] = 1;  /// 2nd  number of the series
cout<<"Fibonacci Series of length "<<n<<"\n";
cout<<fib[0]<< " " <<fib[1]<<" ";
for (i = 2; i < n; i++)
   {
            fib[i] = fib[i-1] + fib[i-2];
            cout<<fib[i]<< " "; }
   }

int main () {
cout<<"Enter the Value n (Length of the Series) ";
int n;
cin>>n;
fib_series(n);
return 0;  }
```

Output

```
Enter the Value n (Length of the Series) 10
Fibonacci Series of length 10
0 1 1 2 3 5 8 13 21 34
```

Lab exercises

1. Write a Program to implement 0/1 Knapsack using dynamic programming technique and analyze the complexity.

2. Write a program to find the optimal order of multiplication in a chain of matrices and analyze the complexity

3. Write a program to find the shortest path between every pair of vertices in a given graph using dynamic programming technique.

Additional exercises

1. Write program to find the binomial coefficient of any numbers *n* and *k* using dynamic programming technique.

2. Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

   Sample Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9

   Sample Output: True //There is a subset (4, 5) with sum 9.

LAB NO: 9 **Date:**

BACKTRACKING

Objectives:

1. T Understand the bounding functions and working of Backtracking algorithm design technique
2. Apply the technique to find the optimal solution
3. Analyze the complexity with and without bounding functions

Backtracking

Backtracking is a systematic way to search for the solution to a problem. In backtracking we begin by defining the solution space for the problem. This solution space must contain at least one optimal solution to the problem. It has to be organized so that it can be searched easily. Solution space is represented as follows:

- Subspace tree : 0/1 Knapsack, Container loading, Max Clique
- Permutation Tree : Traveling sales person

Once solution space is defined this space is searched in a DFS manner beginning at a start node. Initially start node is both live node and Expansion (E) node. From this E node we try to move to a new node. If we can move to a new node from the current E Node then we move to the new node. This new node becomes both live node and also becomes E node. The old E-node remains live node. If we cannot move to a new node the current E node dies and we move back to the most recently seen live node that remains. This live node becomes new E node. Search terminates, when the answer is found or we run out of live nodes to go back

Solved Exercise

N Queen Problem : The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.

```
#include <iostream>

using namespace std;
#define N 8
//# define N 2
int ld[30] = { 0 };
int rd[30] = { 0 }; // Used to check queen can be placed on right diagonal
int cl[30] = { 0 }; //Used to check whether a queen can be placed in that row or not
 bool check_soln(int board[N][N], int col)
{
	if (col >= N)    // All Queens placed
		return true;

	for (int i = 0; i < N; i++) {  // Consider all rows
		// Check if the queen can be placed on
		//board[i][col]
		//Check for board[row][col]
		//Check left diagonal ld[row-col+n-1]
		// Check right diagonal rd[row+coln] where
		 if ((ld[i - col + N - 1] != 1 && rd[i + col] != 1) && cl[i] != 1) {
			//Possible to place this queen in board[i][col]
			board[i][col] = 1;
			ld[i - col + N - 1] = rd[i + col] = cl[i] = 1;
			if (check_soln(board, col + 1))  // Call for remaining queens
				return true;
			board[i][col] = 0; // BACKTRACK
			ld[i - col + N - 1] = rd[i + col] = cl[i] = 0; }
	    }

	return false; // Not possible to place on any row for the col
   }
  int main() {
         int board[N][N] = { { 0, 0}, { 0, 0} };
          cout<<"Solution for "<<N<<" Queens ";
	 if (check_soln(board, 0) == false) {
		cout<<"Solution does not exist";
		exit(0);	     }
```

```
        else{
                cout<<"\n";
                for (int i = 0; i < N; i++) {
                        for (int j = 0; j < N; j++)
                                cout<<" "<<board[i][j]<<" ";
                          cout<<"\n";

        }
  }
        return 0;
}
```

Output

Solution for 8 Queens

```
1  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0
 0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  1
 0  1  0  0  0  0  0  0
 0  0  0  1  0  0  0  0
 0  0  0  0  0  1  0  0
 0    0  1  0  0  0  0  0
```

Solution for 2 Queens does not exist

Time Complexity:

N- Queens problem time complexity → O(n^2).

Lab Exercises

1. Write a program to find the Maximum clique of a graph using backtracking and analyze the complexity.
2. Implement Travelling Sales Person problem using backtracking technique.
3. Find the optimal packing of the Knapsack using backtracking technique use proper bounding functions.
4. Implement container loading problem using backtracking technique.

Additional Exercises

1. Write a program to find the maximum independent set of a given graph using backtracking technique
2. Implement N-Queens problem using backtracking technique to display all the possible solutions.

LAB NO: 10 **Date:**

BRANCH and BOUND

Objectives:

1. Understand the bounding functions and working of Branch and bound algorithm design technique
2. Apply the technique to find the optimal solution
3. Analyze the complexity with and without bounding functions

Branch and Bound:

Brach and bound is another technique to systematically search solution space. Solution space in branch and bound is searched in BFS manner. The main difference between the backtracking and the branch and bound is the way the E-node (Expansion node) is expanded. In branch and bound each live node becomes E-node exactly once When live E-node becomes a new E-node, all the new nodes that are reached within single move are generated

The generated node that can't lead to feasible / optimized solution are discarded. Remaining nodes are added to the list of live nodes and then one node from the list is selected to become the next E-node. This expansion is continued until the list of live nodes become empty

Solved Exercise:

Implement the 0/1 Knapsack using branch and bound technique

```
#include <iostream>
using namespace std;
#include <bits/stdc++.h>
class Item
{
	public : float weight;
	public : int itemprofit;
};
```

```
class Node
{
	// level : Level of node
	// profit : Profit of nodes on path from root up to this node
	// bound : Upper bound of maximum profit in subtree of this node
	public: int level, profit, bound;
	public : float weight;
};

bool cmp(Item a, Item b)
{
	double r1 = (double)a.itemprofit / a.weight;
	double r2 = (double)b.itemprofit / b.weight;
	return r1 > r2;
}

int bound(Node u, int n, int c, Item arr[])
{
	if (u.weight >= c)
		return 0;
	int profit_bound = u.profit;
	int j = u.level + 1;
	int totweight = u.weight;
	while ((j < n) && (totweight + arr[j].weight <= c))
	{
		totweight += arr[j].weight;
		profit_bound += arr[j].itemprofit;
		j++;
	}

	// If k is not n, include last item partially for
	// upper bound on profit
	if (j < n)
		profit_bound += (c - totweight) * arr[j].itemprofit /
								arr[j].weight;
// Not possiple to load last itemnal
	return profit_bound;
 }
```

```
int knapsack(int c, Item arr[], int n) // Compute maximum profit with capacity c
{
	sort(arr, arr + n, cmp);  // Sort on profit density
	queue<Node> Q;
	Node u, v;
	u.level = -1;
	u.profit = u.weight = 0;
	Q.push(u);
	int maxProfit = 0;
	while (!Q.empty())
	{
		u = Q.front();
		Q.pop();

		if (u.level == -1)
			v.level = 0; // For starting node

		if (u.level == n-1)
			continue;
		v.level = u.level + 1; //not last node
		v.weight = u.weight + arr[v.level].weight;
		v.profit = u.profit + arr[v.level].itemprofit;
		if (v.weight <= c && v.profit > maxProfit)
			maxProfit = v.profit;
		v.bound = bound(v, n, c, arr); //Compute bound
		if (v.bound > maxProfit)
			Q.push(v);
		v.weight = u.weight;
		v.profit = u.profit;
		v.bound = bound(v, n, c, arr);
		if (v.bound > maxProfit)
			Q.push(v);
	}

	return maxProfit;
}

int main()
{
```

```
        int n,c,i,j;
        Item arr[10];
        cout<<"Enter the number of items";
        cin>>n;
        cout<<"Enter the Capacity of the Knapsack";
        cin>>c;
    cout<<"Enter the Weight and Profit of each item";
        for(i=0;i<n;i++)
            cin>>arr[i].weight>>arr[i].itemprofit;
        cout << "Maximum possible profit = "
                << knapsack(c, arr, n);

        return 0;
}
```

Output

```
Enter the number of items 3
Enter the Capacity of the Knapsack 6
Enter the Weight and Profit of each item
2   1
3   2
4   5
Maximum possible profit = 6
```

Time Complexity

Time complexity of 0/1 Knapsack using branch and bound → O (2^n)

Lab exercises

1. Write a program to implement container loading problem using Max-Heap branch and bound technique.

2. Implement Travelling Sales Person problem using branch and bound technique. Use proper bounding functions.

3. Write program to find the Maximum Clique of a graph using branch and bound technique. Use proper bounding functions.

Additional exercises

1. Write a program to generate binary strings of length *N* using branch and bound technique.

2. Implement N-Queens problem using branch and bound technique.

LAB NO: 11 **Date:**

APPROXIMATION ALGORITHMS

Objectives:

1. Understand approximation algorithms
2. Design algorithms to implement vertex cover problem, Travelling Salesperson Problem to find near optimal solutions
3. Analyze the complexity

Approximation Algorithms

There is a strong evidence to support that no NP-Hard problem can be solved in polynomial time. But many NP-Hard problems have great practical importance and it is desirable to solve large instances of these problems in a reasonable amount of time. The best known algorithms for NP-Hard problems have exponential worst case complexity.

The examples of NP-Hard problems are :

- 0/1 Knapsack Problem,
- Sum of subsets Problem
- Maximum Clique Problem etc.,

Backtracking & Branch & Bound algorithmic strategies enable to quickly solve a large instance of a problem provided heuristic works on that instance. However this heuristic approach does not work equally effective on all problem instances. NP-Hard problems, even coupled with heuristics, still show exponential behavior on some set of inputs (instances). The discovery of a sub-exponential algorithm (Approximate) for NP-Hard problem increases the maximum problem size that can be solved.

A low polynomial time complexity, NP-Hard optimization problem P, must always generate a feasible solution instead of exact solution. A feasible solution with value close to the optimal solution is called Approximate Solution and the algorithm that generates approximate solution for the problem P, is called as an Approximation Algorithm.

Hence in the case of NP-Hard problems, approximate solutions have added importance as exact solutions (i,.e. optimal solutions) may not be obtainable in a feasible amount of computing time. One can get approximate solution using Approximation Algorithm for an NP-Hard problem in reasonable amount of computing time.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.

- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

Solved Exercise:

Given a universe U of n elements, a collection of subsets of U say S = {S1, S2…,Sm} where every subset Si has an associated cost. Find a minimum cost subcollection of S that covers all elements of U.

```
#include <iostream>
using namespace std;
#include <cstdlib>
class subset
{
    public: int len;
    public : int cost;
    public : int content[10];
};

int main(){
            int U[10],ele;
             int no_of_subsets,i,j;
             int setI[10]={0};
              cout<<"Enter Total No. of elements of U";
              cin>>ele;
              cout<<"Enter values --> Set U";
              for(i=0;i<ele;i++)
                          cin>>U[i];
               subset s[10];
               cout<<"Enter No. of subsets";
                cin>>no_of_subsets;
                for(i=0;i<no_of_subsets;i++){
```

```
            cout<<"Enter the Cost of subset "<<i+1<<"\n";
            cin>>s[i].cost  ;
            cout<<"Enter the No. of elements for subset "<<i+1<<"\n";
            cin>>s[i].len;
            cout<<"Enter the elements of subset "<<i+1<<"\n";
        for(j=0;j<s[i].len;j++)
             cin>>s[i].content[j];
      }
      cout<<"Entered Data";
      cout<<" Set U "<<"\n";
      for(i=0;i<ele;i++)
      cout<<U[i]<<"  ";

     cout<<" Length, of each subset and elements";

  for(i=0;i<no_of_subsets;i++){
      cout<<"Cost"<<"  "<<s[i].cost<<"\n";
      cout<<"Length"<< " "<<s[i].len<<"\n";
      cout<<"Elements of set "<<i<<"\n";
  for(j=0;j<s[i].len;j++)
      cout<<s[i].content[j]<<"  ";
      }
  float temp,minsum=0.0,min;
  i=0;
  int leni = 0, k, l;
  bool flag[10],flag1[10];
  int selected[no_of_subsets]={0};
  leni=0;
  while(leni<ele){
      min=1000.0;
      for(i=0;i<no_of_subsets;i++){
          if(selected[i]==0){
              for(l=0;l<s[i].len;l++)
                  flag1[l]=true;
              for(k=0;k<leni;k++)
                  for(l=0;l<s[i].len;l++)
                      if(setI[k]==s[i].content[l])
                          flag1[l]=false;
      if(flag1[i]){
```

```
        temp=float(s[i].cost)/abs(s[i].len-leni);
        if(temp<min){
            min=temp;
            j=i;} }
    } }
    selected[j]=1;
    for(k=0;k<s[j].len;k++)
        if(leni==0){
            leni=leni+s[j].len;
            for(i=0;i<s[j].len;i++)
                setI[i]=s[j].content[i];
            }
        else{
            for(l=0;l<s[j].len;l++)
                flag[l]=true;
            for(k=0;k<leni;k++)
                for(l=0;l<s[j].len;l++)
                    if(setI[k]==s[j].content[l])
                        flag[l]=false;
                for(l=0;l<s[j].len;l++)
                    if(flag[l])
                       setI[leni++]=s[j].content[l];
            }
            minsum=minsum+s[j].cost;  }
  cout<<"Minsum="<<minsum<<"\n";
    return 0;
}
```

Output

```
Enter Total No. of elements of U5
Enter values --> Set U1
2
3
4
5
Enter No. of subsets3
Enter the Cost of subset 1
5
Enter the No. of elements for subset 1
```

```
3
Enter the elements of subset 1
4
1
3
Enter the Cost of subset 2
10
Enter the No. of elements for subset 2
2
Enter the elements of subset 2
2
5
Enter the Cost of subset 3
3
Enter the No. of elements for subset 3
4
Enter the elements of subset 3
1
4
3
2
Entered Data Set U
1  2  3  4  5   Length, of each subset and elementsCost  5
Length 3
Elements of set 0
4  1  3  Cost  10
Length 2
Elements of set 1
2  5  Cost  3
Length 4
Elements of set 2
1  4  3  2
Minsum=13
```

Lab exercises:

1. A vertex cover of an undirected graph G = (V, E) is a subset V' of set V such that if (u, v) is an edge of G, then either u Є V' or v Є V' (or both). The size of a vertex

cover is the number of vertices in it. Write a program to find a vertex cover of minimum size in a given undirected graph.

2. Write a program to implement travelling sales person's problem in polynomial time approximate using minimum spanning tree.

Additional exercises:

1. Write a program to determine the minimum number of colors needed to color a planar graph G = (V, E).

2. Let there be n programs and two storage devices, two disks with storage capacity of L each. Let li be the amount of storage needed to store the i^{th} program. Determine the maximum number of these n programs that can be stored on the two disks without splitting a program over the disks.

REFERENCES

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, Prentice-Hall India, 2001.

2. Sartaj Sahni, Data Structures, Algorithms and Applications in C++, 2nd Edition, McGraw-Hill, 2000.

3. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, Pearson Education, 2nd Edition, 2007.