MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

CERTIFICATE

This is to certify that Ms./Mr. ……..…… Reg. No. …..…… Section: …… Roll No: ……... has satisfactorily completed the lab exercises prescribed for **Algorithms Lab** [**CSE 2242**] of Second Year B. Tech. Degree at MIT, Manipal, in the academic year 2023-24.

Date: …….....

Signature
Faculty in Charge

CONTENTS

Course Objectives

- Review the fundamental data structures.
- Design and Implementation of various classes of algorithms
- Analysing the efficiency of various algorithms.

Course Outcomes

At the end of this course, students will have the

1. Demonstrate the fundamental data structures.
2. Application of different algorithm design techniques
3. Categorizing algorithms efficiency classes

Evaluation plan

- Internal Assessment Marks : 60 marks

  - ✓ Two evaluations of 20 mark each and One midterm assessment of 20 mark
  - ✓ The assessment will depend on punctuality, designing right algorithm, converting algorithm into an efficient program, maintaining the observation note and answering the questions in viva voce

- End semester assessment of 2 hour duration: 40 marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book 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 record
- 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.
    - Programs should include necessary time analysis part (Operation count /Step count method)
- 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
- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition lab 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.
- You may write scripts/programs to automate the experimental analysis of algorithms and compare with the theoretical result.
- You may use spreadsheets to plot the graph.

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

REVIEW OF FUNDAMENTAL DATA STRUCTURES

Objectives:

In this lab, student will be able to:

- recall the concepts learnt in Data Structures Lab
- implement basic data structures

Description: Data Structures specify the structure of data storage in a program. Various data structures namely **arrays, stacks, queues, linked lists, trees** are used for storing data. Each data structure is different from the other in its fundamental way of storing. In **arrays**, a contiguous piece of memory is allocated for storing data. **Static** and **Dynamic arrays** are two types of array which differ by the instance at which memory is allocated. In **static arrays**, memory is allocated at compile time of the program whereas in **Dynamic arrays** memory is allocated at run time. **Dynamic arrays** overcome the problem of unnecessary wastage of memory space. **Stack** is a data structure in which the insertion to the **Stack** (called as push) and removal from the **stack** (called as pop) operations are performed in Last In First Out (LIFO) order. LIFO specifies that the last item to be pushed is the first one to be popped. **Queue** is a data structure in which the insertion to the queue (enque) and removal of element from the queue (Dequeue) happen in the same order. This means it follows First In First Out (FIFO) order. Linked lists store data in non-linea manner. A node of a **linked list** is created at run time and is used to store data element. Nodes of a **linked list** will be allocated memory at run time and the nodes can be anywhere in memory. **Singly linked** list and **doubly linked** lists are two broad types of **linked lists**. **Single linked list** has a single pointer to the next node whereas doubly linked list has two pointers one to the left node and other to the right node. A special value NULL will be used to denote the non-existence of node. **Trees** are very useful in specific storage requirements of graphs, dictionaries etc. **Binary tree** is a special form of trees in which every node can have maximum two children.

I. SOLVED EXERCISE:

1) Write an algorithm and program to implement a doubly linked list which supports the following operations
   i. Create the list by adding each node at the front.
   ii. Insert a new node to the left of the node whose key value is read as an input.
   iii. Delete all occurrences of a given key, if it is found. Otherwise, display appropriate message.
   iv. Search a node based on its key value.
   v. Display the contents of the list.

Description : Doubly linked list is a data structure in which the data elements are stored in nodes and the nodes are connected by two links. Out of two links one link points to the neighboring node in the left direction and the other link to the node in the right direction. Addresses of nodes will be used to represent the node. A special value NULL is used to represent the absence of a node. Creating the doubly linked list, insertion of an element to the left/right of any node, deletion of all nodes with specific node content and displaying all nodes are the operations commonly performed on a doubly linked list. Each of the operations consumes certain amount of time and memory. Hence they differ in time and space efficiency.

Algorithm: Doubly Linked list
Define a structure to hold list node
Define two links within the node one for left link and the other for rlink

```
CreateList(int val)
begin
        if head == NULL then
                node = allocate memory for a Node
                node ->llink = node->rlink = NULL
                node->val = val
                head=node
        else
                print "List is already created …"
        end if
end

insertIntoList(int before, int val)
begin
        node=head
        while node->val != before
                node = node->rlink
        end while
        if node != NULL then
                newNode = allocate memory for a node
                newNode->val = val
                if node->llink != NULL then
                        node->llink->rlink = newNode
                        newNode->llink = node->llink
                        newNode->rlink = node
                        node->llink = newNode
                else
                        newNode->rlink = node
```

```
                    node->llink = newNode
                        head = newNode
                end if
        else
                print "Unable to insert, node with value " val "not found"
                return
        end if
end

deleteAll(int delVal)
begin
        node = head
        while node != NULL
                if node->val == delVal
                        if node->llink != NULL then
                                node -> llink ->rlink = node -> rlink
                                if node->rlink != NULL then
                                        node->rlink->llink = node->llink
                                        node = node->rlink
                                else
                                        node->llink->rlink = NULL
                                        node=NULL
                                end if
                         else
                                if node->rlink != NULL then
                                        node ->rlink->llink = NULL
                                        head = node->rlink
                                        node = head
                                        release memory for node
                                else
                                        head = node = NULL
                                        release memory for node
                                end if
                        end if
                else
                        node = node->rlink
                end if
        end while
end
```

```
searchNode(int searchVal)
begin
        node=head
        while node != NULL
        do
                if node->val == searchVal then
                        print "Node is found with key ", searchVal
                end if
                node = node->rlink
        end do
end

displayAll()
begin
        node = head
        while node != NULL
        do
                print "Node    with val  ", node->val
                node = node->rlink
        end do
end
```

Time Complexity:

For creating list θ(1) is the time complexity.

For Insertion, Search, Delete, Display All operations the complexity is O(n) where n is the number of nodes.

Program

```
#include<stdio.h>
#include<stdlib.h>

struct node {
int val;
struct node *llink,*rlink;
};

typedef struct node *NODE;
NODE head=NULL;
```

```
void CreateList(int val)
{
        NODE nd;
        if (head == NULL) {
                nd = (NODE) malloc(sizeof(struct node));
                nd->llink = nd->rlink = NULL;
                nd->val = val;
                head=nd;
        }
        else {
                printf("List is already created ….\n");
        }
}

void insertIntoList(int before, int val)
{
        NODE nd, newnd;
        nd=head;
        while (nd != NULL && nd->val != before)
                nd = nd->rlink;
        if (nd != NULL) {
                newnd = (NODE)malloc(sizeof(struct node));
                newnd->llink = newnd->rlink = NULL;
                newnd->val = val;
                if (nd->llink != NULL) {
                        nd->llink->rlink = newnd;
                        newnd->llink = nd->llink;
                        newnd->rlink = nd;
                        nd->llink = newnd;
                }
                else {
                        newnd->rlink = nd;
                        nd->llink = newnd;
                        head=newnd;
                }
        }
        else
                printf( "Unable to insert, node with value  %d not found", val);

}
void deleteAll(int delVal)
{
```

```
	NODE nd,nxtNode;
	nd = head;

	while (nd != NULL) {
		if (nd->val == delVal) {
			if (nd->llink != NULL) {
				nd-> llink ->rlink = nd -> rlink;
				if (nd->rlink != NULL) {
					nd->rlink->llink = nd->llink;
					nxtNode = nd->rlink;
					free(nd);
					nd = nxtNode;
				}
				else {
					nd->llink->rlink = NULL;
					free(nd);
					nd=NULL;
				}

			}
			else {
				if (nd->rlink != NULL) {
					nd ->rlink->llink = NULL;
					head = nd->rlink;
					free(nd);
					nd = head;

				}
				else {
					free(nd);
					head = nd = NULL;
				}
			}
		}
		else
			nd = nd->rlink;
	}
}


void searchNode(int searchVal) {
	NODE nd;
```

```
        nd=head;
        while (nd != NULL) {
                if (nd->val == searchVal)
                        printf( "Node is found with key %d\n", searchVal);
                nd = nd->rlink;
        }
}

void displayAll()
{
        NODE nd;
        nd = head;
        while (nd != NULL) {
                printf("Node    with val  %d\n", nd->val);
                nd = nd->rlink;
        }
}
int main() {
        int choice, val,before;
        do {
                printf("1. Create List\n");
                printf("2. Insert into List\n");
                printf("3. Delete all by value\n");
                printf("4. Search by value\n");
                printf("5. Display all\n");
                printf("6. Exit\n");
                printf("Enter your choice    :");
                scanf("%d", &choice);
                switch(choice) {
                        case 1: printf("Please enter the node value");
                                scanf("%d", &val);
                                CreateList(val);
                                break;
                        case 2: printf("Please enter the node value to insert ");
                                scanf("%d", &val);
                                printf("Please enter the node value before which new
node has to be inserted ");
                                scanf("%d", &before);
                                insertIntoList(before, val);
                                break;
                        case 3:printf("Enter the node value to be deleted ");
                                scanf("%d", &val);
```

```
                        deleteAll(val);
                        break;
                case 4:printf("Enter the node value to be searched ");
                        scanf("%d", &val);
                        searchNode(val);
                        break;
                case 5:displayAll();
                        break;
                case 6:
                        break;
                default:printf("Invalid choice ");
                         break;
                 }
        }while(choice != 6);
        return 0;
}
```

Sample Input and Output:

```
1. Create List
2. Insert into List
3. Delete all by value
4. Search by value
5. Display all
6. Exit
Enter your choice    :1
Please enter the node value5
1. Create List
2. Insert into List
3. Delete all by value
4. Search by value
5. Display all
6. Exit
Enter your choice    :2
Please enter the node value to insert 3
Please enter the node value before which new node has to be inserted 5
1. Create List
2. Insert into List
3. Delete all by value
4. Search by value
5. Display all
6. Exit
Enter your choice    :3
```

```
Enter the node value to be deleted 3
1. Create List
2. Insert into List
3. Delete all by value
4. Search by value
5. Display all
6. Exit
Enter your choice    :4
Enter the node value to be searched 5
Node is found with key 5
1. Create List
2. Insert into List
3. Delete all by value
4. Search by value
5. Display all
6. Exit
Enter your choice    :5
Node    with val  5
1. Create List
2. Insert into List
3. Delete all by value
4. Search by value
5. Display all
6. Exit
Enter your choice    :6
```

II. LAB EXERCISES

1). Write a program to construct a binary tree to support the following operations. Assume no duplicate elements while constructing the tree.

   i. Given a key, perform a search in the binary search tree. If the key is found then display "key found" else insert the key in the binary search tree.

   ii. Display the tree using inorder, preorder and post order traversal methods

2). Write a program to implement the following graph representations and display them.

   i. Adjacency list

   ii. Adjacency matrix

III. ADDITIONAL EXERCISES:

1). Solve the problem given in solved exercise using singly linked list.
2). Write a program to implement Stack and Queue using circular doubly linked list.
3) Write a program to convert a Binary tree to a Doubly linked list

---

LAB NO: 2

FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

Objectives:

In this lab, student will be able to:

- Familiarize with fundamentals of problem solving with the help of algorithms.
- Realize that for a problem there can be multiple solutions with different complexities.
- Determine the time complexity associated with algorithms.

Description: A solution to the problem is obtained after understanding clearly the problem, nature of input and output. The detailed step by step solution to the problem is called an algorithm. For a single problem, there can be multiple ways in which solution is found. Hence, a problem may be solvable using multiple algorithms. To measure the efficiency of an algorithm, measurement along time required by the algorithm and space required is used. To measure time, an operation called basic operation is identified.

I. SOLVED EXERCISE:

1) Write an algorithm for finding the Greatest Common Divisor (GCD) of two numbers using Euclid's algorithm and implement the same. Determine the time complexity of the algorithm.

Description: Greatest Common Divisor(GCD) of two numbers is the largest divisor which divides the two numbers. E.g. GCD(36,8) = 4. Euclid's algorithm is one of the oldest algorithm for calculating GCD. The algorithm is significant because the solution is obtained by reducing the problem space with irregular count. We take the modulus of first number when divided by the second number and we shrink the first number with the second number and the second number with the modulus. This we continue until the second number is not zero. When the second number is zero, the GCD is the first number. Time complexity of this algorithm is in log(n) where n is the second number.

ALGORITHM *EuclidGCD(m, n)*
//Computes gcd*(m, n)* by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers *m* and *n*
//Output: Greatest common divisor of *m* and *n*
while $n \neq 0$ **do**
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return *m*

Time Complexity: O(log n). The worst case for this algorithm is when the inputs are two consecutive Fibonacci numbers. We can plot the graph of (m+n) vs the step count (opcount is shown in the sample code), where m and n are the two inputs to function EuclidGCD.

Program

```
#include<stdio.h>
unsigned int EuclidGCD(unsigned int m, unsigned int n) {
    unsigned int r;
    int  opcount = 0; // variable to count how many times the basic operation executes.
    while(n!=0) {
            opcount++;
             r = m %n;
            m = n;
            n=r;
    }
    printf("Operation count= %d\n", opcount);
    return m;
}
int  main() {
            unsigned int m,n;
            printf("Enter the two numbers whose GCD has to be calculated");
            scanf("%d", &m);
            scanf("%d", &n);
            printf("GCD of two numbers using Euclid's method is %d",
                    EuclidGCD(m,n));  return 0;
}
```

Sample Input and Output:

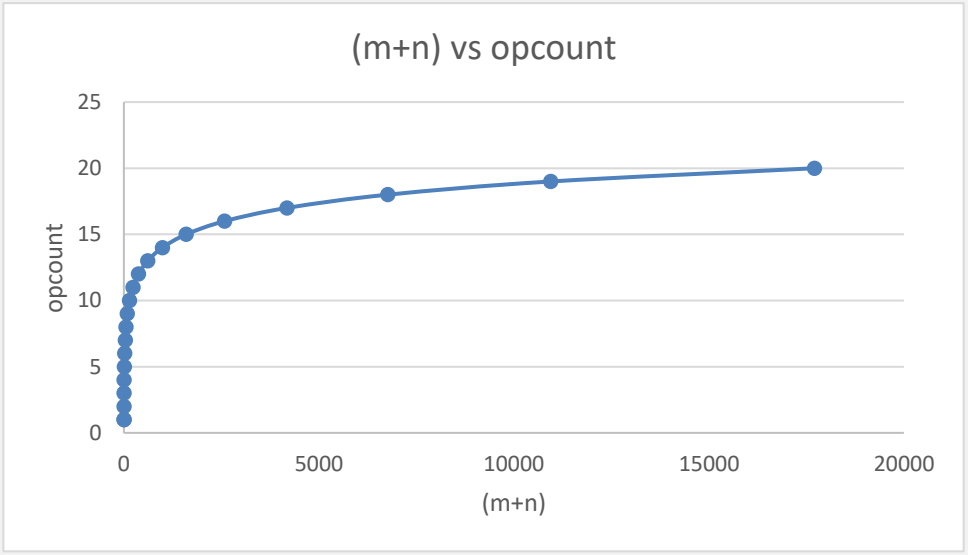
Enter the two numbers whose GCD has to be calculated 8 13

Operation count= 6

GCD of two numbers using Euclids method is 1

Tabulate the values of (m+n), opcount as shown below and plot the graph.

(m+n)	1	8	13	21	.....	6765	10946	17711
opcount	1	4	5	6	....	18	19	20



II. LAB EXERCISES

1). Write a program to find GCD using consecutive integer checking method and analyze its time efficiency.

2). Write a program to find GCD using middle school method and analyze its time efficiency.

III. ADDITIONAL EXERCISES

1) Write a program for computing $\lfloor \sqrt{n} \rfloor$ for any positive integer and analyze its time efficiency. Besides assignment and comparison, your program may only use the four basic arithmetic operations.

2) Write a program to implement recursive solution to the Tower of Hanoi puzzle and analyze its time efficiency.

3) Write a program to compute the n^{th} Fibonacci number recursively and analyze its time efficiency.

4) Write a program to delete strong numbers from an array using recursion [A strong number is such that the sum of it's factorial is the number itself]

LAB NO: 3

BRUTE FORCE TECHNIQUE - I

Objectives:

In this lab, student will be able to:

- Understand brute-force design technique
- Apply this technique for sorting, searching etc.
- Determine the time complexity associated with brute-force algorithms

Description: Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. Brute force is applicable to a very wide variety of problems. For some important problems e.g. sorting, searching, matrix multiplication, string matching the brute-force approach yields reasonable algorithms of value with no limitation on instance size. The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.

I. SOLVED EXERCISE:

1) Write a program to sort a set of integers using selection sort algorithm and analyze its time efficiency. Obtain the experimental result of order of growth. Plot the result for the best and worst case.

Description: In selection sort, we scan the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n-1 elements and exchange it with the second element, putting the second smallest element in its final position. This is repeated for all positions.

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 nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$
      $min \leftarrow j$
    end if

```
  end for
  swap A[i] and A[min]
end for
```

Time Complexity:

$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

The worst case occurs when elements are given in decreasing order.

$C_{worst}(n) = \sum_{i=0}^{n-2} n - 1 - i - 1 + 1$

$= \sum_{i=0}^{n-2} n - i - 1$

$=$ n-1 + n-2 … +1

$= \frac{(n-1)(n-1+1)}{2}$

$= \frac{(n-1)(n)}{2}$

$= \theta(n^2)$

This can be observed by repeating the experiment with the worst case inputs for different array sizes say 10, 15, 20, 35, 100. A plot of number of elements in the array vs opcount (opcount is shown in the sample code) gives a quadratic curve.

Program

```
#include<stdio.h>
#include<stdlib.h>
void selectionSort(int *a, unsigned int n)
{
 unsigned int i,j,min;
 int temp;
 int opcount=0; // introduce opcount
 for(i= 0;i<n-1;i++)
 {
      min=i;
      for(j=i + 1;j<n;j++)
      {
         ++opcount;       // increment opcount for every comparison
         if(a[j ]<a[min])
            min=j;
      }
      //swap A[i] and A[min]
      temp = a[i];
      a[i] = a[min];
      a[min]=temp;
  }
  printf("\nOperation Count %d\n",opcount);
```

```
}
int main() {
	int *a;
	int i,n = 5;
	// generate worst case input of different  input size
	for (int j=0; j < 4; j++) // repeat experiment for 4 trials
	{
		a = (int *)malloc(sizeof(int)*n);
		for (int k=0; k< n; k++)
		    a[k] = n-k; // descending order list
		printf("Elements are ");
		for(i=0;i<n;i++)
			printf("%d   ",a[i]);
		selectionSort(a,n);
		printf("Elements after selection sort ");
		for(i=0;i<n;i++)
			printf("%d   ",a[i]);
		printf("\n");
		free(a);
		n = n + 5;  // try with a new input size
	}
	return 0;
}
```

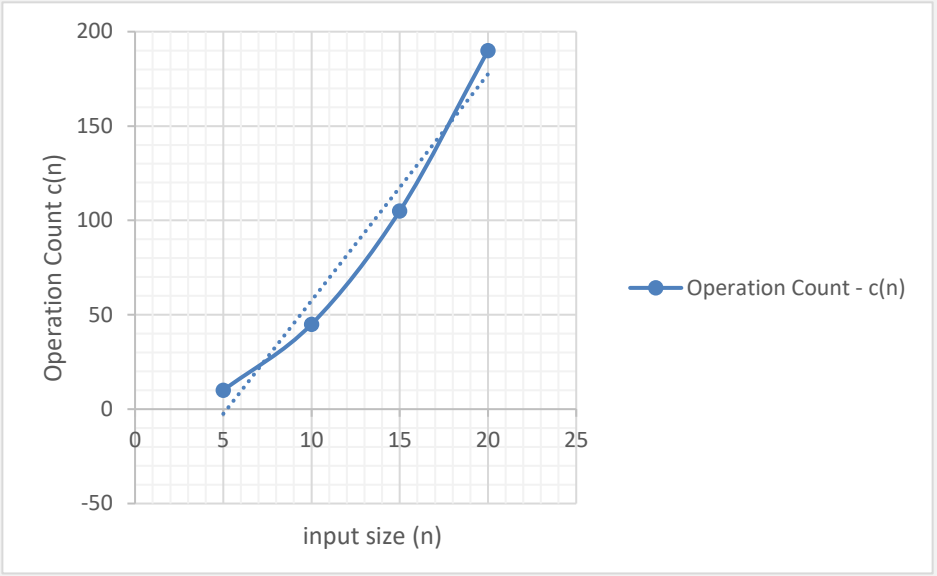
Sample Input and Output :

Elements are 5 4 3 2 1
Operation Count 10
Elements after selection sort 1 2 3 4 5
Elements are 10 9 8 7 6 5 4 3 2 1
Operation Count 45
Elements after selection sort 1 2 3 4 5 6 7 8 9 10
Elements are 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Operation Count 105
Elements after selection sort 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Elements are 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Operation Count 190
Elements after selection sort 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Tabulate the values of n and opcount as shown below and plot the graph.

input size - (n)	5	10	15	20

Operation Count - c(n)	10	45	105	190



II. LAB EXERCISES

1). Write a program to sort set of integers using bubble sort. Analyze its time efficiency. Obtain the experimental result of order of growth. Plot the result for the best and worst case.

2). Write a program to implement brute-force string matching. Analyze its time efficiency.

3). Write a program to implement solution to partition problem using brute-force technique and analyze its time efficiency theoretically. A partition problem takes a set of numbers and finds two disjoint sets such that the sum of the elements in the first set is equal to the second set. [Hint: You may generate power set]

III. ADDITIONAL EXERCISES

1). Write a program to implement matrix multiplication using brute-force technique and analyze its time efficiency. Obtain the experimental result of order of growth. Plot the result for the best and worst case.

2). Write a program in C for finding maximal clique in a graph by brute-force approach. Clique is a maximal complete subgraph in a graph.

3). Write a program to implement solution to partition problem using recursion.

4) Write a program to sort a set of strings using Bubble Sort. Analyze its time efficiency.

LAB NO: 4

BRUTE FORCE TECHNIQUE - II

Objectives:

In this lab, student will be able to:

- Apply brute-force design technique to few combinatorial and graph problems
- Determine the time complexity associated with brute-force combinatorial and graph algorithms

I. SOLVED EXERCISE:

1). Write a program to implement Knapsack problem using brute-force design technique and analyze its time efficiency. Obtain the experimental result of order of growth and plot the result. Knapsack Problem: Given n items of known weights w_1, w_2, ..w_n values v_1, v_2, ...v_n and a knapsack of capacity B, find the most valuable subset of items that fit into the knapsack.

Description: The exhaustive search approach to Knapsack problem leads to generating all the subsets of the set of n items given computing the total weight of each subset in order to identify feasible subsets and finding a subset of the largest value among them.

ALGORITHM *Knapsack(W*[0..*n* − 1],*V[0..n-1],B)*

//Determine the Knapsack for a capacity B

//Input: An array W[0..*n* − 1] of weights, V[0..*n* − 1] of values, capacity B

//Output: An array K[0..m-1] of Knapsack items

```
maxVal ← 0
while there is a unique bit array of size n
begin
        0 in the bit array at position i implies element vi is absent in Knapsack
        1 in the bit array at position i implies element vi is present in Knapsack
        Add all the weights corresponding to 1 in the bit array
        if total weight < B then
                Add all the Values corresponding to 1 in the bit array
                if total value > maxVal then
                        maxVal ← total value
                end if
        end if
end while
```

Time Complexity :
The basic operation in this is comparison inside loop. This is repeated as many times as the iterations of the loop. The loop is repeated 2^n times.

Hence, $C(n)= \theta(2^n)$

This can be observed by repeating the experiment for knapsack problems with number of items as 3, 4, 5, 10, 20….. by randomly generating the weight set and the profit set. A plot of number of items, n, vs opcount (opcount is shown in the sample code) gives an exponential curve.

Program

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int Knapsack(unsigned int *w, unsigned int *v, unsigned int n, unsigned int B)
{
        unsigned int i,temp;
        unsigned  int maxVal=0, maxSequence=0;
        unsigned int totalWeight, totalValue;
        int opcount=0; // Intialize the opcount variable
        unsigned int index=0;
         //Generate bit array
         for(i=1;i<pow(2,n);i++)
        {
            ++opcount; //increment opcount for every possible subsets
              //Compute total weight
            temp = i;
            totalWeight=totalValue=0;
               index=0;
            while(temp) {
                  if(temp & 0x1) {
                        totalWeight = totalWeight + w[index];
                       totalValue = totalValue + v[index];
                    }
                        index++;
                   temp = temp >> 1;
             }
            if(totalWeight <= B && totalValue > maxVal) {
                  maxVal = totalValue;
                 maxSequence = i;
```

```
            }
        }
    printf("\n Operation count = %d\n",opcount);
      return maxSequence;
}
int main() {
              unsigned int *v,*w, i,n,knaps, B;
              printf("Enter the number of elements ");
              scanf("%d", &n);
              v= (unsigned int *)calloc(n, sizeof(unsigned int));
              w = (unsigned int *) calloc(n, sizeof(unsigned int));
              printf("Please enter the weights");
              for(i=0;i<n;i++)
                        scanf("%d",&w[i]);
             printf("Please enter the values");
             for(i=0;i<n;i++)
                        scanf("%d",&v[i]);
                       printf("Please enter the Knapsack capacity");
             scanf("%d", &B);
             knaps = Knapsack(w,v,n,B);
             printf("Knapsack contains the following items \n");
             i=0;
             while(knaps) {
                        if(knaps & 0X1)
                                  printf("item %u        value %u", (i+1), v[i]);
                        i++;
                        knaps = knaps >> 1;
             }
 return 0;
}
```

Sample Input and Output :

```
Enter the number of elements 3
Please enter the weights1 2 4
Please enter the values2 4 8
Please enter the Knapsack capacity5
Operation count = 7
Knapsack contains the following items
item 1        value 2item 3        value 8
```

II. LAB EXERCISES

1). Write a program for assignment problem by brute-force technique and analyze its time efficiency. Obtain the experimental result of order of growth and plot the result.
2). Write a program for depth-first search of a graph. Identify the push and pop order of vertices.
3). Write a program for breadth-first search of a graph.

III. ADDITIONAL EXERCISES

1). Write a program to check whether a graph is bipartite or not using
   i. DFS to check for bipartite
   ii. BFS to check for bipartite

   A graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y.

2). Write a program to construct a graph for the following maze. One can model a maze by having a vertex for a starting point, a finishing point, dead ends, and all the points in the maze where more than one path can be taken, and then connecting the vertices according to the paths in the maze. Also find the solution to the maze using Depth-First-Search.

3) Write a program to implement the Traveling Salesman Problem using Brute Force Method

---

LAB NO: 5

DECREASE AND CONQUER

Objectives:

In this lab, student will be able to:

- Understand decrease and conquer design technique
- Apply this technique to examples like Topological sorting, diameter of a graph
- Determine the time efficiency

Description : The decrease-and-conquer technique is based on exploiting the relationship between solution to a given instance of a problem and a solution to its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up. The former leads to a recursive implementation. The bottom-up version is implemented iteratively. There are three major variations of decrease and conquer. Decrease by constant, decrease by a constant factor and variable size decrease.

I. SOLVED EXERCISE:

1). Write a program to sort a set of numbers using insertion sort and analyze its time efficiency. Obtain the experimental result of order of growth and plot the result.

Description: We assume that smaller problem of sorting the array A[0,...n-2] has already been solved to give us a sorted array of size n-1 : A[0] ≤ A[1]... ≤A[n-2] has already been solved to give us a sorted array of size n-1. The new element A[n-1] need to be inserted into the right position. We compare each element starting from A[n-2] down to A[0] to check the right position for A[n-1]. Once found, we insert into that position.

ALGORITHM *InsertionSort(A*[0..*n* − 1]*)*
//Sorts a given array by insertion sort
//Input: An array *A*[0..*n* − 1] of *n* orderable elements
//Output: Array *A*[0..*n* − 1] sorted in nondecreasing order
for *i* ←1 **to** *n* − 1 **do**
    v ←*A*[*i*]
    j ←*i* − 1
    while *j* ≥ 0 **and** *A*[*j*]> *v* **do**
        A[*j* + 1]←*A*[*j*]
        j ←*j* – 1
    end while
    A[*j* + 1]←*v*
end for

Time Complexity :

$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$

The worst case occurs when elements are given in decreasing order.

In such cases, j will range from 0 to i-1.

$C_{worst}(n) = \sum_{i=1}^{n-1} i - 1 - 1 + 1$

$= \sum_{i=1}^{n-1} i - 1$

$= n-2 + n-3 \ldots +0$

$= \frac{(n-2)(n-2+1)}{2}$

$= \frac{(n-2)(n-1)}{2}$

$= \theta(n^2)$

This can be observed by repeating the experiment with the worst-case inputs for different array sizes say 10, 15, 20, 35, 100. A plot of number of elements in the array vs opcount (opcount is shown in the sample code) gives a quadratic curve.

Program :

```
#include<stdio.h>
#include<stdlib.h>
void insertionSort(int *a, unsigned int n)
{
        int i, j, v;
        int opcount=0;
        for(i=1;i<n;i++)
        {
                v=a[i];
                j=i-1;
                // increment opcount whenever there is an element comparison
                while(++opcount && j>=0 && a[j]> v)
                {
                        a[j+1]=a[j];
                        j=j-1;
                }
                a[j+1]=v;
        }
  printf("\n Operation count %d\n",opcount);
}
int main() {
      int *a;
      int i,n = 5;
      // generate worst case input of different  input size
```

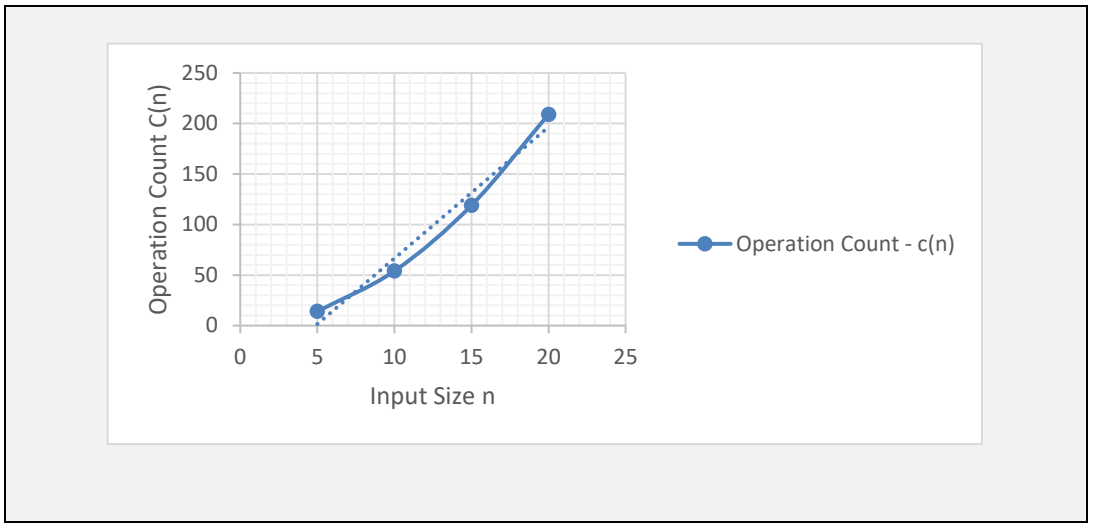
```
    for (int j=0; j < 4; j++) // repeat experiment for 4 trials
    {
        a = (int *)malloc(sizeof(int)*n);
        for (int k=0; k< n; k++)
            a[k] = n-k; // descending order list
        printf("Elements are ");
        for(i=0;i<n;i++)
            printf("%d   ",a[i]);
        insertionSort(a,n);
        printf("Elements after selection sort ");
        for(i=0;i<n;i++)
            printf("%d   ",a[i]);
        printf("\n");
        free(a);
        n = n + 5;  // try with a new input size
    }
    return 0;
}
```

Sample Input and Output :

Elements are 5 4 3 2 1
Operation count 14
Elements after selection sort 1 2 3 4 5
Elements are 10 9 8 7 6 5 4 3 2 1
Operation count 54
Elements after selection sort 1 2 3 4 5 6 7 8 9 10
Elements are 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Operation count 119
Elements after selection sort 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Elements are 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Operation count 209
Elements after selection sort 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Tabulate the values of n and opcount as shown below and plot the graph.

Input Size (n)	5	10	15	20
Operation Count - c(n)	14	54	119	209

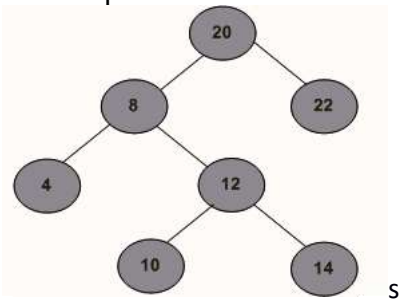


II. LAB EXERCISES

1).Write a program to determine the Topological sort of a given graph using

i. Depth-First technique
ii. Source removal technique

2). Write a C program to find Closest Common Ancestor (CCA) in a Binary Tree. The CCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root [i.e., closest to n1 and n2].
For Example: Consider the following BT



s

Input: CCA of 10 and 14
Output: 12
Input: CCA of 8 and 14
Output: 8

III. ADDITIONAL EXERCISES

1). A student management system should be developed by a software company. There are certain dependent and independent modules that must be developed by K teams. The unit testing and integration testing is done before the deployment of the product. Design an algorithm using decrease and conquer technique such that it gives a schedule of tasks in the order in which it must be executed.

2). Modify the algorithm such that it checks whether there is any task dependency between the teams.

3) Write a program in C to find gcd of two numbers using Euclid's algorithm employing the Decrease and Conquer strategy.

LAB NO: 6

DIVIDE AND CONQUER

Objectives:

In this lab, student will be able to:

- Understand divide and conquer algorithm design technique
- Apply this technique to merge sort, quick sort etc.
- Determine the time complexity associated with divide and conquer algorithms

Description : In divide-and-conquer a problem is divided into several sub-problems of the same type, ideally of about equal size. The sub-problems are solved and if necessary, the solutions to the sub-problems are combined to get a solution to the original problem.

I. SOLVED EXERCISE:

1). Write a program to determine the height of a binary search tree and analyze its time efficiency.

Description : A binary search tree is a data structure in which orderable elements are stored. It is a special form of binary tree in which root will have larger value than all the children on the left and root will have lesser value than all the children on the right. This condition is applicable for all the nodes of a binary search tree.

ALGORITHM *Height(T)*
//Computes recursively the height of a binary tree
//Input: A binary tree *T*
//Output: The height of *T*
if *T* = NULL **then**
  return −1
else
  return max{*Height(Tleft), Height(Tright)*} + 1
end if

Time Complexity : The time complexity of this algorithm is $\theta(n)$ considering the basic operation as comparison (or addition**).** This can be observed by repeating the experiment for trees with different number of nodes as 5, 7, 10, 12, 20….. A plot of number of nodes, n, vs opcount (opcount is shown in the sample code) gives a linear curve.

Program

```
#include<stdio.h>
#include<stdlib.h>

#define MAX(a,b) ((a) > (b) ? a : b)

 int opcount=0;  //initialize the opcount variable

struct node{
        int val;
        struct node *left, *right;
};

typedef struct node *NODE;

NODE root=NULL;

NODE insert(int ele, NODE node)
{
        NODE temp;
        if(node == NULL) {
                temp= (NODE)malloc(sizeof(struct node));
                temp->val=ele;
                temp->left = temp->right=NULL;
                if(root == NULL)
                        root=temp;
                return temp;
        }
        else if(ele < node->val) {
                node->left = insert(ele, node->left);
                return node;
        }
        else if(ele > node->val) {
                node->right = insert(ele,node->right);
                return node;
        }
        else {
                printf("Duplicate element found while inserting. Insertion
failed\n");
                return NULL;
        }
}
```

```
int height(NODE node) {
        opcount++;  // increment opcount for the comparison statement
        if(node==NULL)
                return -1;
        else
                return MAX(height(node->left), height(node->right))+1;
}
void main() {
        int choice,ele;
        do {
                printf("1. Insert an element\n");
                printf("2. Find Height of BST\n");
                printf("3. Exit\n");
                printf("Please enter your choice");
                scanf("%d",&choice);
                switch(choice) {
                        case 1: printf("Insertion : Please enter an element\n");
                                scanf("%d", &ele);
                                insert(ele,root);
                                break;
                        case 2: printf("Height of BST : %d\n",height(root));
                                printf("Opcount=" %d);
                                break;
                        case 3: break;
                        default: printf("Invalid choice. Please enter valid
choice\n");
                                break;
                }
        }while(choice != 3);
}
```

Sample Input and output:

```
1. Insert an element
2. Find Height of BST
3. Exit
Please enter your choice: 1
Insertion : Please enter an element 5
1. Insert an element
2. Find Height of BST
3. Exit
Please enter your choice: 1
```

```
Insertion : Please enter an element 3
1. Insert an element
2. Find Height of BST
3. Exit
Please enter your choice: 1
Insertion : Please enter an element 1
1. Insert an element
2. Find Height of BST
3. Exit
Please enter your choice: 2

1. Insert an element
2. Find Height of BST
3. Exit
Please enter your choice: 3
```

II. LAB EXERCISE:

Write a program to:

1) Find total number of nodes in a binary tree and analyze its efficiency. Obtain the experimental result of order of growth and plot the result.
2) Sort given set of integers using Quick sort and analyze its efficiency. Obtain the experimental result of order of growth and plot the result.
3) Sort the given set of integers using Merge sort and display the number of inversions performed during the merging step. Obtain the experimental result of order of growth by analysing its efficiency and plot the result.

III. ADDITIONAL EXERCISES:

1) Write a program in C to find a^n where $n > 0$ using divide and conquer strategy.
2) Using divide and conquer strategy, count the number of ways to tile the given "2 x n" board. The tiles are of size "2 x 1" and it can either be placed horizontally or vertically.
3) Given an image, find the defective region in the image using divide and conquer technique. The defective region is denoted by 0 and the non-defective region is denoted by 1.
4) Write a C program to find the convex hull of a given set of points using divide and conquer strategy

LAB NO: 7

TRANSFORM AND CONQUER - I

Objectives:

In this lab, student will be able to:

- Understand **Transform and Conquer** design technique
- Apply this technique to examples like AVL tree, 2-3 tree, etc.
- Determine the time complexity associated with this technique

Description: The technique is called as **Transform and Conquer** because these methods work as two-stage procedures. First in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then in the second or conquering stage it is solved.

There are three major variations of this idea that differ by what we transform a given instance.

- Transformation to a simpler or more convenient instance of the same problem-we call it **instance simplification**.
- Transforming to a different representation of the same instance-we call it **representation change**.
- Transformation to an instance of a different problem for which an algorithm is already available-we call it **problem reduction**.

I. SOLVED EXERCISE:

1). Write a program to create a binary search tree and display its elements using all the traversal methods and analyse its time efficiency.

Description: Binary search tree is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it. Note that this transformation from a set to a binary search tree is an example of the representation-change technique.

To create and maintain the information stored in a binary search tree, we need an operation that inserts new nodes into the tree. We use the following insertion approach. A new node is always inserted into its appropriate position in the tree *as a leaf.* We write a function that inserts an item into the tree, given a pointer to the root of the whole tree: create(tree, item). We compare the item to the data of the (current) root node and then call the function to insert the item into the correct subtree-the left subtree if item's key is less than the key of the root node, and the right subtree if item's key is greater than the key of the root node. The argument 'tree', to the create(), is a reference parameter. The case where tree is NULL is the base case where a node will be created and the element will be stored. The recursive call will pass the reference to the appropriate left or right

subtree depending on the item to be inserted. The important point to remember is that passing a pointer by *value* allows the function to change what the caller's pointer points to; passing a pointer by *reference* allows the function to change the caller's pointer as well as to change what the pointer points to.

Algorithm: Create binary search tree
Step1 : If tree is NULL, then allocate a new leaf to contain item.
Step2: If item < tree->info, then recursively call Insert(tree->llink, item).
Step3: Else If item > tree->info, then recursively call lnsert(tree->rlink, item).
Step4: Else Write("Error: duplicate item")

Time Efficiency:

T(n)=O(n), where n is number of nodes

Program

```
#include<stdio.h>
typedef struct node
{
        int info;
        struct node *left,*right;
}NODE;

NODE* create(NODE *bnode,int x)
{
        NODE *getnode;
        if(bnode==NULL)
        {
                bnode=(NODE*) malloc(sizeof(NODE));
                bnode->info=x;
                bnode->left=bnode->right=NULL;
        }
        else if(x>bnode->info)
                bnode->right=create(bnode->right,x);
        else if(x<bnode->info)
                bnode->left=create(bnode->left,x);
        else
        {
                printf("Duplicate node\n");
                exit(0);
```

```
        }
        return(bnode);
}
void inorder(NODE *ptr)
{
        if(ptr!=NULL)
        {
                inorder(ptr->left);
                printf("%5d",ptr->info);
                inorder(ptr->right);
        }
}
void postorder(NODE *ptr)
{
        if(ptr!=NULL)
        {
                postorder(ptr->left);
                postorder(ptr->right);
                printf("%5d",ptr->info);
        }
}
void preorder(NODE *ptr)
{
        if(ptr!=NULL)
        {
                printf("%5d",ptr->info);
                preorder(ptr->left);
                preorder(ptr->right);
        }
}
void main()
{
        int n,x,ch,i;
        NODE *root;
        root=NULL;
        while(1)
        {
                printf("**********Output**********\n\n");
                printf("----------Menu----------\n");
                printf(" 1. Insert\n 2. All traversals\n 3. Exit\n");
                printf("Enter your choice:");
                scanf("%d",&ch);
```

```
                switch(ch)
                {
                        case 1: printf("Enter node (do not enter duplicate nodes):\n");
                                scanf("%d",&x);
                                root=create(root,x);
                                break;
                        case 2: printf("\nInorder traversal:\n");
                                inorder(root);
                                printf("\nPreorder traversal:\n");
                                preorder(root);
                                printf("\nPostorder traversal:\n");
                                postorder(root);
                  printf("\n\n**********************************************");
                                break;
                        case 3: exit(0);
                }
        }
}
```

Sample Input and Output

```
**********Output**********
----------Menu----------
 1. Insert
 2. All traversals
 3. Exit
Enter your choice:1
Enter node (do not enter duplicate nodes):
234
**********Output**********

----------Menu----------
 1. Insert
 2. All traversals
 3. Exit
Enter your choice:1
Enter node (do not enter duplicate nodes):
22
**********Output**********
```

```
----------Menu----------
 1. Insert
 2. All traversals
 3. Exit
Enter your choice:1
Enter node (do not enter duplicate nodes):
65
**********Output**********

----------Menu----------
 1. Insert
 2. All traversals
 3. Exit
Enter your choice:1
Enter node (do not enter duplicate nodes):
25
**********Output**********
----------Menu----------
 1. Insert
 2. All traversals
 3. Exit
Enter your choice:2

Inorder traversal:
  22   25   65  234
Preorder traversal:
  234   22   65   25
Postorder traversal:
  25   65   22  234
**********************************************
```

II. LAB EXERCISES:

1) Write a program to create the AVL tree by iterative insertion.
2) Using the AVL created in question 1, display the successor (next greater key) and predecessor (next smaller key) of a given key.

III. ADDITIONAL EXERCISES:

1) For the AVL tree created in exercise 1 above, insert an element 6
2) Write a program to create a 2-3 tree for a set of integers.
3) For the 2-3 tree created in exercise 2 above, insert an element 6.

LAB NO: 8

TRANSFORM AND CONQUER - II

Objectives:

In this lab, student will be able to:

- Understand **construction of heap** using **Transform and Conquer** design technique
- Apply heap concepts to sort set of elements.
- Determine the time complexity associated with this technique.

Description: A **heap** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

1. The shape property—the binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The parental dominance or heap property—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

I. SOLVED EXERCISE:

1). Write a program to construct a heap for a given list of keys using bottom-up heap construction algorithm.

Description: The bottom-up heap construction algorithm initializes the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows. Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node. If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position. This process continues until the parental dominance for K is satisfied. (Eventually, it has to because it holds automatically for any key in a leaf.) After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor. The algorithm stops after this is done for the root of the tree.

ALGORITHM HeapBottomUp(H[1..n])
//Constructs a heap from the elements of a given array
//by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]

for i ← $\left\lfloor \frac{n}{2} \right\rfloor$ downto 1 do
    k ← i; v ← H[k]
    heap ← false
    while not heap and 2*k≤n do
        j ← 2*k
        if j<n //there are two children
            if H[j] < H[j+1] j ← j+1
        if v≥H[j]
            heap ← true
        else H[k] ← H[j]; k ← j
    H[k] ← v

Time Efficiency:

$T_{worst}(n)=2(n-\log_2(n+1))$, where n is number of elements

Program

```
#include<stdio.h>
#include<conio.h>
void heapify(int h[],int n)
 {
  int i,k,v,heapify,j;
  for(i=(n/2);i>=1;i--)
    {
        k=i;v=h[k];heapify=0;
        while(heapify==0&&2*k<=n)
          {
              j=2*k;
              if(j<n)
                if(h[j]<h[j+1])
                    j=j+1;
              if(v>=h[j])
                 heapify=1;
              else
```

```
                    {
                      h[k]=h[j];
                          k=j;
                    }
                }
              h[k]=v;
            }
     return;
  }
void main()
  {
    int h[20],i,n;
    clrscr();
    printf("\nEnter the number of Elements:");
    scanf("%d",&n);
    printf("\nEnter the Elements:");
    for(i=1;i<=n;i++)
      scanf("%d",&h[i]);
    printf("\ndisplay the array:");
    for(i=1;i<=n;i++)
    {
      printf("%d\t",h[i]);
    }
    heapify(h,n);
    printf("\nThe heap created:");
    for(i=1;i<=n;i++)
    {
      printf("%d\t",h[i]);
    }
    getch();
  }
```

Sample Input and Output

Enter the number of Elements
6
Enter the Elements
2 9 7 6 5 8
The heap created
9 6 8 2 5 7

II. LAB EXERCISES:

1) Write a program to create a heap for the list of integers using top-down heap construction algorithm and analyze its time efficiency. Obtain the experimental results for order of growth and plot the result.
2) Write a program to sort the list of integers using heap sort with bottom up max heap construction and analyze its time efficiency. Prove experimentally that the worst case time complexity is O(n log n)

ADDITIONAL EXERCISES:

1) Write a program to check whether an array H[1..n] is a heap or not
2) Write a program for finding and deleting an element of a given value in a heap.
3) Write a program to find and delete the smallest element in the max heap.

LAB NO: 9

SPACE AND TIME TRADEOFFS

Objectives:

In this lab, student will be able to:

- Understand **Space and Time tradeoffs** design technique
- Apply this technique in input enhancement & pre-structuring techniques.
- Determine the time complexity associated with this technique**.**

Description: In time and space tradeoffs technique if time is at premium, we can pre-compute the function's values and store then in a table. In somewhat more general terms, the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. We call this approach input enhancement. The other type of technique that exploits space-for-time trade-offs simply uses extra space to facilitate faster and/or more flexible access to the data. We call this approach pre-structuring.

I. SOLVED EXERCISE:

1) Write a program to sort set of integers using comparison counting algorithm.

Description: Comparison counting sort uses the input-enhancement technique. One rather obvious idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element and record the results in a table. These numbers will indicate the positions of the elements in the sorted list: e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.

```
ALGORITHM ComparisonCountingSort(A[0..n − 1])
 //Sorts an array by comparison counting
 //Input: An array A[0..n − 1] of orderable elements
 //Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
 for i ←0 to n − 1 do Count[i]←0
 for i ←0 to n − 2 do
     for j ←i + 1 to n − 1 do
        if A[i]<A[j ]
            Count[j ]←Count[j ]+ 1
        else Count[i]←Count[i]+ 1
 for i ←0 to n − 1 do S[Count[i]]←A[i]
 return S
```

Time Efficiency:

$T(n)=O(n^2)$, where n is number of integers

Program

```
#include <stdio.h>
void counting_sort(int A[])
{
   int i, j;
   int S[15], C[100];
   for (i = 0; i <= n-1; i++)
      C[i] = 0;
   for (i = 0; i <= n-2; i++)
     {
    for (j = i+1; j <= n-1; j++)
       {
            if A[i]<A[j ]
                        Count[j ]←Count[j ]+ 1;
                     else Count[i]←Count[i]+ 1;
      }
      }
   for (i = 0; i <= n-1; i++)
      S[c[i]]=A[i];
   printf("The Sorted array is : ");
      for (i = 0; i <= n-1; i++)
      printf("%d ", S[i]);
}
int main()
{
   int n, k = 0, A[15], i;
       printf("Enter the number of integers : ");
   scanf("%d", &n);
   printf("\nEnter the integers to be sorted :\n");
   for (i = 1; i <= n; i++)
      scanf("%d", &A[i]);
  counting_sort(A);
  printf("\n");
  return 0;
}
```

Sample Input and Output

Enter the number of integers :

6

Enter the integers to be sorted :

62 31 84 96 19 47

The Sorted array is :

19 31 47 62 84 96

II. LAB EXERCISES:

1) Write a program to implement Horspool's algorithm for String Matching and find the number of key comparisons in successful search and unsuccessful search.
2) Write a program to construct the Open hash table. Find the number of key comparisons in successful search and unsuccessful search. This should be done by varying the load factor of the hash table. You may consider varying the number of keys for a fixed value of hash table size say m=10 and n=50, 100, and 200. This should be repeated for at least four different hash table sizes say m= 20, m=50.
3) Write a program to construct the closed hash table. Find the number of key comparisons in successful search and unsuccessful search.

III. ADDITIONAL EXERCISES:

1). Write a program to implement Boyer-Moore algorithm for String Matching and find the number of key comparisons in successful search and unsuccessful search.
2). Write a program to sort the elements using distribution counting method.

LAB NO: 10

DYNAMIC PROGRAMMING

Objectives:

In this lab, student will be able to:

- Understand **Dynamic Programming** design technique
- Apply this technique to examples like Warshall's and Floyd's algorithm.
- Determine the time complexity associated with this technique**.**

Description: **Dynamic programming** is a technique for solving problems with overlapping sub-problems. Typically, these sub-problems arise from a recurrence relating a solution to a given problem with solutions to its smaller sub-problems of the same type. Rather than solving overlapping sub-problems again and again, dynamic programming suggests solving each of the smaller sub-problems only once and recording the results in a table from which we can then obtain a solution to the original problem.

I. SOLVED EXERCISE:

1) Write a program to find the Binomial Co-efficient using Dynamic Programming.

Description: Computing a binomial coefficient is a standard example of applying dynamic programming to a nonoptimization problem. You may recall from your studies of elementary combinatorics that the *binomial coefficient,* denoted C(n, *k)* is the number of combinations (subsets) of *k* elements from an n-element set (0 <= *k* <= *n).* The name "binomial coefficients" comes from the participation of
these numbers in the binomial formula.
So,
$C(n, k) = C(n-1, k-1) + C(n-1, k)$ for $n > k > 0$
and
$C(n, 0) = C(n, n) = 1$.

Time Efficiency:

T(n)=O(nk) for binomial coefficient of any number n & k

ALGORITHM Binomial(n,k)
// Computes C(n,k) by the dynamic programming algorithm
//Input: A pair of nonnegative integers $n \geq k \geq 0$
//Output: The value of C(n,k)
for i ← 0 **to** n **do**
  for j ← 0 to min(i,k) **do**

```
        if j=0 or j=i
            C[i,j] ← C[i-1,j-1] + C[i-1,j]
  return C[n,k]
```

Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int c[20][20];
void binomial(int n,int k)
{
  int i,j;
  for(i=0;i<=n;i++)
  {
    for(j=0;j<=min(i,k);j++)
    {
      if(j==0||j==i)
            c[i][j]=1;
      else
           c[i][j]=c[i-1][j-1]+c[i-1][j];
    }
  }
}
void main()
{
  int n,k,i,j;
  clrscr();
  printf("Enter the value of n\n");
  scanf("%d",&n);
  printf("Enter the value of k\n");
  scanf("%d",&k);
  if(n<k)
    printf("Invalid input: n cannot be less than k\n");
  else if(k<0)
    printf("Invalid input: k cannot be less than 0\n");
  else
  {
    binomial(n,k);
    printf("Computed matrix is \n");
    for(i=0;i<=n;i++)
    {
```

```
    for(j=0;j<=min(i,k);j++)
        printf("%d\t",c[i][j]);
    printf("\n");
   }
   printf("binomial coefficient c[%d,%d]=%d\n",n,k,c[n][k]);
  }
  getch();
}
```

Sample Input and Output

```
Enter the value of n
6
Enter the value of k
3
Computed matrix is
 1
 1 1
 1 2 1
 1 3 3 1
 1 4 6 4
 1 5 10 10
 1 6 15 20
binomial coefficient c[6 3]=20
```

II. LAB EXERCISES:

1). Write a program to compute the transitive closure of a given directed graph using Warshall's algorithm and analyse its time efficiency. Obtain the experimental results for order of growth and plot the result.

2). Write a program to implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem for any given graph and analyse its time efficiency. Obtain the experimental results for order of growth and plot the result.

3). Write a program to implement 0/1 Knapsack problem using bottom-up dynamic programming

III. ADDITIONAL EXERCISES:

1). Write a program to implement 0/1 Knapsack problem using memory function.

2). Write a function to find the composition of an optimal subset from the table generated by the bottom-up dynamic programming algorithm for the knapsack problem.

LAB NO: 11

GREEDY TECHNIQUE

Objectives:

In this lab, student will be able to:

- Understand **Greedy Technique** design technique
- Apply this technique to examples like Prim's, Kruskal's and Dijkstra's Algorithm
- Determine the time complexity associated with this technique

Description: The **greedy** approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step-and this is a central point of this technique-the choice made must be

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

I. SOLVED EXERCISE:

1) Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

Description: A *spanning tree* of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a *minimum spanning tree* is its spanning tree of the smallest weight, where the *weight* of a tree is defined as the sum of the weights on all its edges. The *minimum spanning tree problem* is the problem of finding a minimum spanning tree for a given weighted connected graph.

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.) The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n -1, where n is the number of vertices

in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

ALGORITHM Prim(G)
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G=<V,E>
//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
$E_T \leftarrow \Phi$
for i ← 1 **to** |V| -1 **do**
find a minimum-weight edge e*=(v*,u*) among all the edges (u,v)
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

Time Efficiency:

T(n)=O(|E| log|V|), for a graph with E edges and V vertices, represented as adjacency list

Program

```
#include<stdio.h>
#include<conio.h>
int a[50][50],t[50][50],root[50],parent[50],n,i,j,value,e=0,k=0;
int ivalue,jvalue,cost=0,mincost=0,TV[50],count=0,present=0;
main()
{
        clrscr();
        printf("\n\t\t\t PRIMS ALGORITHM\n");
        TV[++count]=1;
        read_cost();
        prims();
        display();
        getch();
}
read_cost()
{
     printf("\n Enter the number of vertices:");
```

```
        scanf("%d",&n);
        printf("\n Enter cost adjacency matrix\n");
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        if(i<j)
        {
                printf("(%d,%d):",i,j);
                scanf("%d",&value);
                a[i][j]=value;
                if(value!=0)
                        e++;
        }
        else if(i>j)
                a[i][j]=a[j][i];
        else
                a[i][j]=0;
}
prims()
{
        while(e && k<n-1)
        {

                for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                {
                        if(a[i][j]!=0)
                        {
                                int x,y;
                                x=check_reach(i);
                                y=check_reach(j);
                                if((x==1) && (y==0))
                                {
                                        present=1;
                                        if((a[i][j] < cost) || (cost==0))
                                        {
                                                cost=a[i][j];
                                                ivalue=i;
                                                jvalue=j;
                                        }
                                }
                        }
                }
```

```
                if(present==0) break;
                a[ivalue][jvalue]=0;
                a[jvalue][ivalue]=0;
                e--;
                TV[++count]=jvalue;
                t[ivalue][jvalue]=cost;
                k++;
                present=cost=0;
        }
}
display()
{
        if(k==n-1)
        {
                printf("\n Minimum cost spanning tree is\n");
                for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                {
                        if(t[i][j]!=0)
                        printf("\n(%d,%d):%d",i,j,t[i][j]);
                        mincost=mincost+t[i][j];
                }
                printf("\n Cost of this spanning tree:%d",mincost);
        }
        else
                printf("\n Graph is not connected");
}
int check_reach(int v)
{
        int p;
        for(p=1;p<=count;p++)
        if(TV[p]==v)  return 1;
        return 0;
}
```

Sample Input and Output

PRIMS ALGORITHM

Enter the number of vertices:

6

Enter cost adjacency matrix

1 2 3

1 3 0

1 4 0

1 5 6

1 6 5

2 3 1

2 4 0

2 5 0

2 6 4

3 4 6

3 5 0

3 6 4

4 5 8

4 6 5

5 6 2

Minimum cost spanning tree is

1 2 3

2 3 1

2 6 4

6 5 2

6 4 5

Cost of this spanning tree

15

II. LAB EXERCISES:

1). Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm and analyse its time efficiency.

2) Given a weighted connected graph, write a program to implement Dijkstra's algorithm to find the shortest path from a given vertex to other vertices in the graph. Also, analyze the time efficiency of the algorithm.

3) Write a program to implement Huffman tree construction algorithm.

III. ADDITIONAL EXERCISES:

1). Write a program to find a maximum spanning tree – a spanning tree with the largest possible edge weight of a weighted connected graph.

2). Write a program to implement the greedy algorithm for the change-making problem, with an amount n and coin denominations $d_1 > d_2 > ... > d_m$ as its input.

LAB NO: 12

BACKTRACKING & BRANCH AND BOUND

Objectives:
In this lab, student will be able to:

- Understand **Backtracking & Branch and Bound** design technique
- Apply this technique to examples like subset-sum problem and knapsack problem**.**

Description: The principle idea in *Backtracking* is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first legitimate option for the next component. If there is no legitimate option for the next component, no alternative for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with the next option. *Branch-and-bound* requires two additional items, compared to backtracking:

1) A way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node.
2) The value of the best solution seen so far.

If this information is available, we can compare a node's bound value with the value of the best solution seen so far. If the bound value is not better than the value of the best solution seen so far-i.e., not smaller for a minimization problem and not larger for a maximization problem-the node is nonpromising and can be terminated (some people say the branch is "pruned"). Indeed, no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:
1) The value of the node's bound is not better than the value of the best solution seen so far.
2) The node represents no feasible solutions because the constraints of the problem are already violated.
3) The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

I. SOLVED EXERCISE:

1) Write a program for n-Queens problem using backtracking technique.

Description: The **n-queens problem** is to place '*n'* Queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. For $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$. So let us consider the 4-Queens problem and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board. We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem.

ALGORITHM Backtrack(X[1..i])
//Gives a template of a generic backtracking algorithm
//Input: X[1..i] specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions

if X[1..i] is a solution **write** X[1..i]
else /* This algorithm works correctly only if no solution is a prefix to another solution to the problem. The pseudocode needs to be changed to work correctly for such problems as well. */
  for each element x ε S_{i+1} consistent with X[1..i] and the constraints **do**
    X[i+1] ← x
    Backtrack(X[1..i+1])
  end for
end if

Time Efficiency:

Size of state space tree

Program

```
int place(int[],int);
void printsolution(int,int[]);
void main()
{
    int n;
   clrscr();
   printf("Enter the no.of queens: ");
   scanf("%d",&n);
   nqueens(n);
   getch();
}
void nqueens(int n)
{
   int x[10],count=0,k=1;
   x[k]=0;
   while(k!=0)
   {
      x[k]=x[k]+1;
     while(x[k]<=n&&(!place(x,k)))
     x[k]=x[k]+1;
     if(x[k]<=n)
    {
       if(k==n)
      {
        count++;
        printf("\nSolution %d\n",count);
        printsolution(n,x);
      }
       else
       {
          k++;
          x[k]=0;
       }
     }
   else
   {
      k--; //backtracking
    }
  }
return;
```

```
}
int place(int x[],int k)
{
      int i;
      for(i=1;i<k;i++)
      if(x[i]==x[k]||(abs(x[i]-x[k]))==abs(i-k))
      return 0;
      return 1;
}
void printsolution(int n,int x[])
{
      int i,j;
      char c[10][10];
      for(i=1;i<=n;i++)
      {
         for(j=1;j<=n;j++)
        c[i][j]='X';
      }
     for(i=1;i<=n;i++)
     c[i][x[i]]='Q';
     for(i=1;i<=n;i++)
    {
       for(j=1;j<=n;j++)
       {
          printf("%c\t",c[i][j]);
       }
       printf("\n");
    }
}
```

Sample Input and Output

Enter the no. of queens: 4

Solution 1

X Q X X

X X X Q

Q X X X

X X Q X

Solution 2

X X Q X

Q X X X

X X X Q

X Q X X

II. LAB EXERCISES:

1). Write a program to find the solution to the subset-sum problem using backtracking. Consider the test case S={1, 2, 5, 6, 8} and d=9, to verify your answer.

2). Write a program to implement Knapsack problem using branch and bound technique.

III. ADDITIONAL EXERCISES:

1). Write a program for finding Hamiltonian circuit for the graph, using backtracking.

2). Write a program to implement assignment problem using Branch and Bound.

3). Solve job assignment problem using Hungarian method.

References:

1. Anany Levitin, *Introduction to The Design and Analysis of Algorithms*, 3rd Edition, Pearson Education, India, 2012.
2. Ellis Horowitz and Sartaj Sahni, *Computer Algorithms/C++*, Second Edition, University Press, 2007.
3. Thomas H. Cormen, Charles E. Leiserson, Ronal L, Rivest, Clifford Stein, *Introduction to Algorithms*, PHI, 2nd Edition, 2006.