IMS Engineering College, Ghaziabad

Department of Computer Science and Engineering

Session 2016-17

LAB MANUAL

Design and Analysis of Algorithms

(NCS-551)

IMS Engineering College, Ghaziabad

Department of Computer Science & Engineering

Subject Name: Design and Analysis of Algorithm Lab
Subject Code: NCS-551
Year and Branch: B.Tech 3rd Year CSE

As Per the University Syllabus

S.N.	Objectives	Outcomes	PO's	PSO's
1.	Aim: To implement Linear Search in C language	Outcome: To understand Linear Search strategy.	1,2,3,4,5,6,11,12	1,2,3,4
2	Aim: To implement Recursive Binary Search in C language.	Outcome: To understand the divide and conquer Strategies.	1,2,3,4,5,6,11,12	1,2,3,4
3.	Aim: To implement Heap Sort in C language	Outcome: The student will able to understand the working of heap sort algorithm.	1,2,3,4,5,11,12	1,2,3,4
4.	Aim: To implement Merge Sort in C language	Outcome: The student will able to understand the working of Merge sort algorithm.	1,2,3,4,5,6,11,12	1,2,3,4
5.	Aim: To implement Selection Sort in C language	Outcome: The student will able to understand the working of selection sort algorithm.	1,2,3,4,5,6,11,12	1,2,3,4
6.	Aim: To implement Insertion Sort in C language	Outcome: The student will able to understand the working of insertion sort algorithm.	1,2,3,4,5,6,11,12	1,2,3,4
7.	Aim : To implement Quick Sort in C language	Outcome: Be able to understand the how the divide and conquer based quick sort algorithm works.	1,2,3,4,5,6,11,12	1,2,3,4
8.	Aim: To study the NP-Complete theory and its application.	Outcome: Be able to understand the concepts of NP-Completeness.	1,2,3,4,5,11,12	1,2,3,4
9.	Aim: To study the Cook's theorem and its application.	Outcome: Gain knowledge of Cook's theorem.	1,2,3,4,5,11,12	1,2,3,4
10.	Aim: To study the Sorting network and its application.	Outcome : Be familiar with the Sorting networks	1,2,3,4,5,11,12	1,2,3,4

Experiment beyond syllabus

E. No	**Objectives**	**Outcomes**	**Po's**	**PSO's**
1.	Aim: Give implementation of Greedy algorithm.	Outcome: To learn and understand Greedy Algorithm.	1,2,3,5,4,6,11,12	1,2,3,4
2	Aim: Give implementation of Dynamic algorithm.	Outcome: To learn and understand Dynamic algorithm.	1,2,3,5,4,6,11,12	1,2,3,x4

Prepared By **Evaluated by**

List of Experiments

1	Implement a C program Linear Search and analyze its complexity
2	Implement a C program to search an element in a sorted set of elements using recursive binary search and analyze its complexity
3	Implement a C program to sort a set of integers using heap sort and analyze its complexity
4	Implement a C program to sort a set of integers using merge sort and analyze its complexity
5	Implement a C program to sort a set of integers using Selection sort and analyze its complexity
6	Implement a C program to sort a set of integers using insertion sort and analyze its complexity
7	Write C program to sort a set of integers using quick sort and analyze its complexity Implement
8	Study Of NP-Complete Theory
9	Study Of Cook's theorem.
10	Study of Sorting network

Experiments Beyond Syllabus:

B1.	Implement a C program to solve fractional knapsack problem using greedy method.
B2.	Implement a C program to solve matrix chain multiplication problem using dynamic programming.

Experiment # 1

Experiment Title: - To Implement Linear Search in C language.

Objective:

1. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
2. Ensures that students understand how the worst-case/best case/average time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms,
3. Differentiate between various algorithms for sorting (e.g., insertion, merge, quick-sort, and heap sort), searching (e.g., linear and binary search), and selection (e.g., min, max) and when to use them.
4. Know about data structures arrays to support specific applications.

Outcome:

To Understand Linear Search Strategy.

THEORY: It is also known as sequential search. It is very simple and works as follows: We keep on comparing each element with the element to search until the desired element is found or list ends. Linear search in c language for multiple occurrences and using function.

Mapping with PO & PSO

PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
2	3	2	2	1	1	-	-	-	-	1	2

PSO1	**PSO2**	**PSO3**	**PSO4**
3	2	1	1

ALGORITHM:

Step 1: Set i to 1

```
Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit
```

IMPLEMENTATION CODE:

```
#include <stdio.h>

int main()
{
  int array[100], search, c, n;

  printf("Enter the number of elements in array\n");
  scanf("%d",&n);

  printf("Enter %d integer(s)\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  printf("Enter the number to search\n");
  scanf("%d", &search);

  for (c = 0; c < n; c++)
  {
    if (array[c] == search)     /* if required element found */
    {
      printf("% d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d is not present in array.\n", search);

  return 0;
}
```

OUTPUT:

```
Enter the Number of Elements in array

5

Enter 5 integer(s)

8

3

5

1

2

Enter the number of search 1

1 is present in location 4
```

ANALYSIS:

Worst case time complexity – O(n)

Average case time complexity – O(n)

References:

1. Horowitz and Sahani, ―Fundamentals of Data Structures, Galgotia Publications Pvt Ltd Delhi India
2. Thomas H. Cormen et al.- Introduction to Algorithms
2. G.S. Baluja, ― Data Structure using C.
3. Lipschutz, ―Data Structures‖ Schaum's Outline Series, Tata Mcgraw-hill Education (India) Pvt. Ltd .
4. Reema Thareja- Data Structure Using C, Oxford Edition

Experiment # 2

Experiments Title: To implement C program for recursive binary search

Objective

5. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
6. Ensures that students understand how the worst-case time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms,
7. Differentiate between various algorithms for sorting (e.g., insertion, merge, quick-sort, and heap sort), searching (e.g., linear and binary search), and selection (e.g., min, max) and when to use them.
8. Augment various data structures (trees and arrays) to support specific applications.

Outcomes:-

To understand the divide and conquer strategy

Mapping with PO & PSO

PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
2	3	2	2	1	1	-	-	-	-	1	2

PSO1	**PSO2**	**PSO3**	**PSO4**
3	2	1	1

Algorithm: BINARY SEARCH

```
Step 1:    if(begin<=end) the go to Step 2 else go to Step
Step 2:    mid = (begin+end)/2
Step 3:    if(key element = a[mid]) then successful search else go to step 4
           /* search the upper half of the array */
Step 4:    If(k<a[mid]) then recursively call binary(begin, mid-1)
           /* search the lower half of the array */
Step 5:    if(k>a[mid]) then recursively call binary (mid+1,end)
```

Implementation:

```
#include<stdio.h>
#include<conio.h>
int a[20] , n , k ;                          /* variable declaration */
int bsearch ( int begin , int end ) ;   /* Function declaration */

void main ()
{
    int i , flag = 0 ;
    clrscr () ;
    printf ( " \n Enter size of the array n : " ) ;
    scanf ( "%d" , &n ) ;
    printf ( " \n Enter elements of array in ascending order : " ) ;
    for ( i = 0 ; i < n ; i++ )
            scanf ( "%d" , &a[i] ) ;
    printf ( "\n Enter the key element : " ) ;
    scanf ( "%d" , &k ) ;
    flag = bsearch ( 0, n - 1 ) ;
    if ( flag == 1 )
            printf ( " \n Successful search , key element is present " ) ;
    else
            printf ( " \n Unsuccessful search " ) ;
    getch () ;
    }

    int bsearch ( int begin , int end )
    {
            int mid ;
            if ( begin <= end )
            {
                    mid = ( begin + end ) / 2 ;
                    if ( k == a[mid] )
                            return 1 ;
                    if ( k < a[mid] )
                            return bsearch ( begin , mid - 1 ) ;
                    if ( k > a[mid] )
                            return bsearch ( mid + 1, end ) ;
            }
            return 0 ;
    }
```

Result:

```
Enter size of the array n : 5

 Enter elements of array in ascending order :
1
2
3
4
5

 Enter the key element: 2

 Successful search, key element is present

  Enter the key element: 6

 Unsuccessful search
```

Result Analysis:

The procedure is applicable only for sorted array .Binary search procedure takes O(logn) time

Experiment # 3

Experiments Title: To implement a C program to sort a given set of elements using the Heap Sort method and analyze complexity

Objective

1. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
2. Ensures that students understand how the worst-case time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms,
3. Use different computational models (e.g., divide-and-conquer), order notation and various complexity measures (e.g., running time, disk space) to analyze the complexity/performance of different algorithms.
4. Differentiate between various algorithms for sorting (e.g., insertion, merge, quick-sort, and heap sort), searching (e.g., linear and binary search), and selection (e.g., min, max) and when to use them.
5. Augment various data structures (trees and arrays) to support specific applications.

Course Outcomes

CO1. Understand asymptotic notations to analyze the performance of algorithms

CO2. Identify the differences in design techniques and apply to solve optimization problems.

CO3. Apply algorithms for performing operations on graphs and trees.

CO4. Solve novel problems, by choosing the appropriate algorithm design technique for their solution and justify their selection

Mapping with PO & PSO

PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
2	3	2	2	1	1	-	-	-	-	1	2

PSO1	**PSO2**	**PSO3**	**PSO4**
3	2	1	1

Algorithm:

/* Constructs a heap from the elements of a given array */

```
/* Input : An array H[1..n] of orderable items
/* Output : A heap H[1..n]
Step 1:     for i = n/2  downto 1 do Step 2
Step 2:     k = i ; v = H[k] and heap = false
Step 3:     while not heap and 2 * k <= n do
Step 4:     j = 2 * k
Step 5:     if j < n // there are two children
Step 6:     if H[j] < H[j+1] j = j + 1
Step 7:     if v >= H[j]     heap = true
Step 8:     else go to Step 9
Step 9:     H[k] = H[j] ; k = j
Step 10:    H[k] = v
```

Implementation:

```
#include<stdio.h>
#include<conio.h>
#define max 100
void heapify () ;
void heapsort ()  ;
int maxdel ();
int a[max] , b[max] ,n ;

void main ()
{
        int i ;
        int m ;
        clrscr () ;
        printf ( " \n Enter array size : " ) ;
        scanf ( "%d" , &n ) ;
        m = n ;
        printf ( " \n Enter elements : \n " ) ;
        for ( i = 1 ; i <= n ; i++ )
                scanf ( "%d" , &a[i] ) ;
        heapsort () ;
        printf ( " \n The sorted array is : \n " ) ;
        for ( i = 1 ; i <= m ; i++ )
                printf ( "\n%d" , b[i] ) ;
        getch () ;
}

void heapsort ()
{
        int i ;
        heapify () ;
```

```
          for ( i = n ; i >= 1 ; i-- )
                    b[i] = maxdel () ;
}

void heapify ()
{
          int i , e , j ;
          // start from middle of a and move up to 1
          for ( i = n/2 ; i >= 1 ; i-- )
          {
                    e = a[i] ; //save root of subtree
                    j = 2 * i ;   //left child and c+1 is right child
                    while ( j <= n )
                    {
                              if ( j < n && a[j] < a[j+1] )
                                        j++ ;    //pick larger of children
                              if ( e >= a[j] )
                                        break;  // is a max heap
                              a[j/2] = a[j] ;
                              j = j * 2 ;         //go to the next level
                    }
                    a[j/2] = e ;
          }
}

int maxdel ()
{
          int x , j , e , i ;
          if ( n == 0 )
                    return -1 ;
          x = a[1] ;         //save the maximum element
          e = a[n] ;         //get the last element
           n-- ;
          // Heap the structure again
          i = 1 ;
          j = 2;
          while ( j <= n )
          {
                    if ( j < n && a[j] < a[j+1] )
                              j++ ;    //Pick larger of two childrean
                    if ( e >= a[j] )
                              break ; //subtree is heap
                    a[i] = a[j] ;
                    i = j ;
                    j = j * 2 ;         // go to the next level
```

```
                }
                a[i] = e ;
                return x ;
        }
```

Result:

```
    Enter array size : 5

    Enter elements :
    7
   1
   9
   3
   5

    The sorted array is :

   1
   3
   5
   7
   9
```

Result Analysis:

Complexity of HeapSort is O(nlogn) in all cases.

.

Experiment # 4

Experiment Title: - Implement Merge Sort by C program and analyze its complexity

Objective

1. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
2. Ensures that students understand how the worst-case time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms,
3. Differentiate between various algorithms for sorting (e.g., insertion, merge, quick-sort, and heap sort), searching (e.g., linear and binary search), and selection (e.g., min, max) and when to use them.
4. Augment various data structures (trees and arrays) to support specific applications.

Course Outcomes

CO1. Understand asymptotic notations to analyze the performance of algorithms

CO3. Apply algorithms for performing operations on graphs and trees.

CO4. Solve novel problems, by choosing the appropriate algorithm design technique for their solution and justify their selection

Mapping with PO & PSO

PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
2	2	3	2	1	1	-	-	-	-	1	3

PSO1	**PSO2**	**PSO3**	**PSO4**
2	1	3	1

Theory: Using divide and Conquer approach in merge sort method, we have to sort n number of data.

For Example: If we have input data like
38 27 43 3 9 82 10
Stepwise solution will be:
38 27 43 3 | 9 82 10
38 27 | 43 3 | 9 82 | 10
38 | 27 | 43 |3 |9 |82 | 10

27 38 | 3 43 | 9 82 | 10
3 27 38 43 | 9 10 82
3 9 10 27 38 43

Algorithm:

```
Algorithm Mergesort(low,high)
{
If(low<high) then //if there are more then one element
    {
    //Divide P into subproblems.
    //Find where to split the set.
    Mid:=(low+high)/2;
    //Solve the subproblems.
    Mergesort(low,mid);
    Mergesort(mid+1,high);
    //Combine the solution.
    Merge(low,mid,high);
    }
}

Algorithm Merge(low,mid,high)
{
    h:=low;i:=low;j:=mid+1;
    while((h<=mid) and (j<=high)) do
    {
        if(a[h]<=a[j]) then
        {
        b[i]:=a[h];h:=h+1;
        }
        Else
        {
        b[i]:=a[j];j:=j+1;
    }
    i:=i+1;
if(h>mid)then
    for k:=j to high do
    {
        b[i]:=a[k];i:=i+1;
    }
Else
    For k:=h to mid do
    {
    B[i]:=a[k];i:=i+1;
    }
    For k:=low to high do a[k]:=b[k];
}
```

Implementation:

```
#include<iostream.h>
#include<conio.h>
void merge(int *,int,int,int); //shows how data will be divided, getting sorted and merged.
void mergesort(int *,int ,int );//Divide data into subparts and merge them
void main()
{
	clrscr();
	int n, a[10], i;//a[]will store input.
	cout<<"Enter the number of elemets =";
	cin>>n;
	cout<<"Enter the elements of arry for sorting = ";
	for(i=0;i<n;i++)
	{
		cin>>a[i];
	}
	mergesort (a,0,n-1);
	cout<<"Sorted elements : ";
	for(i=0;i<n;i++)
	cout<<a[i]<<' ';
	getch();
}
void ms(int *a, int low, int high)
{
//low indicates index of first data and high indicates index of last data.
	if(low<high)
	{
		int mid=(low+high)/2;
		mergesort (a,low,mid);
		mergesort (a,mid+1,high);
		merge(a,low,mid,high);
	}
}
void merge(int *a, int low, int mid,int high)
{
	int temp[10];
	int h=low,i=low,j=mid+1;
	while(i<=mid&&j<=high)
	{
		if(a[i]<=a[j])
		{
			temp[h++]=a[i++];
		}
		else
```

```
                    temp[h++]=a[j++];
        }
        if(i>mid)
        {
                for(int k=j;k<=high;k++)
                    temp[h++]=a[k];
        }
        else
        {
                for(int k=i;k<=mid;k++)
                    temp[h++]=a[k];
        }
        for(int k=low;k<=high;k++)
        {
                a[k]=temp[k];
        }
}
```

Result:-

```
Enter the number of elements =5
Enter the elements of arrays for sorting:
2 3 1 5 4
Sorted elements:
1 2 3 4 5
```

Result Analysis:-

Time Complexity Of Merge Sort in best case is(when all data is already in sorted form):O(n)
Time Complexity Of Merge Sort in worst case is: O(nlogn)
Time Complexity Of Merge Sort in average case is: O(nlogn)

Application:

Merge sort type algorithms allowed large data sets to be sorted on early computers that had small random access memories by modern standards. Records were stored on magnetic tape and processed on banks of magnetic tape drives. *Merge Sort* is useful for sorting linked lists in O(nlogn) time.

Experiment # 5

Experiment Title:- To implement selection sort in C.

Objective:

1. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
2. Ensures that students understand how the worst-case/best case/average time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms,
3. Differentiate between various algorithms for sorting (e.g., insertion, merge, quick-sort, and heap sort), searching (e.g., linear and binary search), and selection (e.g., min, max) and when to use them.
4. Know about data structures arrays to support specific applications.

Outcome:

To Understand Selection Sort Strategy.

Mapping with PO & PSO

PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
2	3	2	2	1	1	-	-	-	-	1	2

PSO1	**PSO2**	**PSO3**	**PSO4**
3	2	1	1

THEORY:- Selection sort is a simple sorting algorithm. This sorting algorithm is a in-place comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.

Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes part of sorted array. This process continues moving unsorted array boundary by one element to the right.

ALGORITHM:-

```
Step 1 − Set MIN to location 0
```

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

IMPLEMENTATION CODE:

```
#include <stdio.h>

int main()
{
  int array[100], n, c, d, position, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for ( c = 0 ; c < n ; c++ )
    scanf("%d", &array[c]);

  for ( c = 0 ; c < ( n - 1 ) ; c++ )
  {
    position = c;

    for ( d = c + 1 ; d < n ; d++ )
    {
      if ( array[position] > array[d] )
        position = d;
    }
    if ( position != c )
    {
      swap = array[c];
      array[c] = array[position];
      array[position] = swap;
    }
  }

  printf("Sorted list in ascending order:\n");

  for ( c = 0 ; c < n ; c++ )
    printf("%d\n", array[c]);

  return 0;
}
```

OUTPUT:

Result:

```
Enter the number of elements =5
Enter the elements of arrays for sorting:
2 3 1 5 4
Sorted elements:
1 2 3 4 5
```

ANALYSIS:

Best Case: Ω(n^2)
Average case: Θ(n^2)

Experiment # 6

Experiment Title: - Implement a C program to Sort a given set of elements using Insertion sort and analyze complexity.

Objective:

1. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
2. Ensures that students understand how the worst-case/best case/average time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms,
3. Differentiate between various algorithms for sorting (e.g., insertion, merge, quick-sort, and heap sort), searching (e.g., linear and binary search), and selection (e.g., min, max) and when to use them.
4. Know about data structures arrays to support specific applications.

Outcome:

To Understand Insertion Sort Strategy.

Mapping with PO & PSO

PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
2	3	2	2	1	1	-	-	-	-	1	2

PSO1	**PSO2**	**PSO3**	**PSO4**
3	2	1	1

Algorithm:

```
Insertionsort(A [ 0…n-1 ] )
//Input: An array A [0….n-1] of orderable elements
//Output: Array A [ 0…..n-1 ] sorted in increasing order
Step 1:     for i = 1 to n – 1 do {
Step 2:     v = A[i]
Step 3:     j = i – 1
Step 4:     while j >= 0 and A[j] > v do {
Step 5:         A[j+1] = a[j]
Step 6:         j = j – 1
            } //end while
Step 7:     A[j+1] = v
            } //end for
```

Implementation:

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

int a[20] , n ;

void main ()
{
        void insertionsort() ;
        int i ;
        clrscr () ;
        printf ( " \n Enter size of the array : " ) ;
        scanf ( "%d" , &n ) ;
        printf ( " \n Enter the elements : \n " ) ;
        for ( i = 0 ; i < n ; i++ )
                scanf ( "%d" , &a[i] ) ;
        insertionsort () ;
        printf ( " \n The sorted elements are : " ) ;
        for ( i = 0 ; i < n ; i++ )
                printf ( "\n%d" , a[i] ) ;
        getch () ;
}

void insertionsort ()
{
        int i , j , min ;
        for ( i = 0 ; i < n ; i++ )
        {
                min = a[i] ;
                j = i - 1 ;
                while ( j >= 0 && a[j] > min )
```

```
                    {
                              a[j + 1] = a[j] ;
                              j = j - 1 ;
                    }
                    a[j + 1] = min ;
          }
}
```

Result:

```
 Enter size of the array : 5

 Enter the elements :
 9
2
8
5
1

 The sorted elements are :
1
2
5
8
9
```

Result Analysis:

This algorithm takes O(n) time if array elements are already sorted. In worst case and in average it takes O(n^2) time

Experiment # 7

Experimental Title: -To write a C program to perform Quick sort using the divide and conquer technique and analyze complexity

Objective

1. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
2. Ensures that students understand how the worst-case time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms,
3. Use different computational models (e.g., divide-and-conquer), order notation and various complexity measures (e.g., running time, disk space) to analyze the complexity/performance of different algorithms.
4. Differentiate between various algorithms for sorting (e.g., insertion, merge, quick-sort, and heap sort), searching (e.g., linear and binary search), and selection (e.g., min, max) and when to use them.
5. Augment various data structures (arrays) to support specific applications.

Course Outcomes

CO1. Understand asymptotic notations to analyze the performance of algorithms

CO2. Identify the differences in design techniques and apply to solve optimization problems.

CO3. Apply algorithms for performing operations on graphs and trees.

CO4. Solve novel problems, by choosing the appropriate algorithm design technique for their solution and justify their selection

Mapping with PO & PSO

PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
2	2	3	2	1	1	-	-	-	-	1	3

PSO1	**PSO2**	**PSO3**	**PSO4**
2	1	3	1

Theory:

38 81 22 48 13 **69** 93 14 45 58 79 72
69 81 22 48 13 38 93 14 45 58 79 72
69 58 22 48 13 38 93 14 45 81 79 72
69 58 22 48 13 38 45 14 93 81 79 72
14 58 22 48 13 38 45 **69** 93 81 79 72

Algorithm:

Step 1: Start the process.
Step 2: Declare the variables.
Step 3: Enter the list of elements to be sorted using the get()function.
Step 4: Divide the array list into two halves the lower array list and upper array list using the merge sort function.
Step 5: Sort the two array list.
Step 6: Combine the two sorted arrays.
Step 7: Display the sorted elements using the dis() function.
Step 8: Stop the process.

Implementation:

```
#include<stdio.h>
#include<conio.h>
swap(int &a,int &b)
{
        int t;
        t=a;
        a=b;
        b=t;
}
median(int a[],int left,int right)
{
        int centre=(left+right)/2;
        if(a[left]<a[centre]
                swap(a[left],a[centre]);
        if(a[right]<a[left])
                swap(a[right],a[left]);
        if(a[right]<a[centre])
                swap(a[centre],a[right]);
        swap(a[centre],a[right-1]);
        return(a[right-1]);
}
```

```
median(int a[], int left,int right)
{
        int centre=(left+right)/2;
        if(a[left]<a[centre]
        swap(a[left],a[centre]);
        if(a[right]<a[left])
                swap(a[right],a[left]);
        if(a[right]<a[centre])
                swap(a[centre],a[right]);
        swap(a[centre],a[right-1]);
        return(a[right-1]);
}
quicksort(int a[],int left,int right)
{
        if(left<right)
        {
                int pivot=median(a,left,right);
                int i=left;
                int j=right-1;
                for(;;)
                {
                        while(a[++i]<pivot);
                        while(pivot<a[--j]);
                        if(i<j)
                                swap(a[i],a[j]);
                        else
                                break;
                }
                if(l<right)
                        swap(a[i],a[right-1]);
        qsort(left,i-1);
        qsort(i+1,right);
        }
}
void main()
{
int a[10], n, i;
printf(”\n\nQuick sort\n\n”);
printf(”Enter the number of element :\n”);
scanf(“%d”,n);
printf(“\nEnter the elements :\n”);
for(i=0;i<n;i++)
scanf(“%d”,a[i]);
quicksort(a,0,n-1);
printf(”\nSorted elements :\n”);
for(i=0;i<n;i++)
printf(“%d”,a[i]);
getch();
}
```

Result:

```
Enter the number of elements: 6
Enter the elements:
2
7
1
6
8
9
Sorted elements:
1 2 6 7 8 9
```

Result Analysis:

Quicksort runs in $O(n^2)$ time in the worst case, but that it runs in O(nlog(n)) time in the expected case, assuming the input is randomly ordered.

Applications:

Most commercial applications would use quick sort for its better average performance: they can tolerate an occasional long run (which just means that a report takes slightly longer to produce on full moon days in leap years) in return for shorter runs most of the time.

8. Study Title:- Study of NP-Complete Theory

I. Introduction:-A problem is said to be polynomial if there exists an algorithm that solves the problem in time $T(n)=O(n^c)$, where c is a constant.

Course Objective: - Know various advanced design and analysis techniques such as greedy algorithms, dynamic programming & Know the concepts of tractable and intractable problems and the classes P, NP and NP-complete problems.

Outcome:- Analyze deterministic and nondeterministic algorithms to solve complex Problems

Mapping with PO & PSO

PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
3	2	3	2	1	-	-	-	-	-	1	2

PSO1	PSO2	PSO3	PSO4
2		1	1

- Examples of polynomial problems:
  - Sorting: $O(n \log n) = O(n^2)$
  - All-pairs shortest path: $O(n^3)$
  - Minimum spanning tree: $O(E \log E)= O(E^2)$
- A problem is said to be exponential if no polynomial-time algorithm can be developed for it and if we can find an algorithm that solves it in $O(n^{u(n)})$, where u(n) goes to infinity as n goes to infinity.
- The world of computation can be subdivided into three classes:

1. Polynomial problems (P)
2. Exponential problems (E)
3. Intractable (non-computable) problems (I)

There is a very large and important class of problems that

0. we know how to solve exponentially,
1. we don't know how to solve polynomials, and
2. we don't know if they can be solved polynomials at all

This class is a gray area between the P-class and the E-class. It will be studied in this chapter.

II. Definition of NP

- Definition 1 of NP: A problem is said to be Nondeterministically Polynomial (NP) if we can find a nondeterminsitic Turing machine that can solve the problem in a polynomial number of nondeterministic moves.
- For those who are not familiar with Turing machines, two alternative definitions of NP will be developed.
- Definition 2 of NP: A problem is said to be NP if
    1. its solution comes from a finite set of possibilities, and
    2. it takes polynomial time to verify the correctness of a candidate solution
- Remark: It is much easier and faster to "grade" a solution than to find a solution from scratch.
- We use NP to designate the class of all nondeterministically polynomial problems.
- Clearly, P is a subset of NP
- A very famous open question in Computer Science:

$$P = NP\ ?$$

- To give the 3rd alternative definition of NP, we introduce an imaginary, non-implementable instruction, which we call "choose()".
- Behavior of "choose()":
    1. if a problem has a solution of N components, choose(i) magically returns the i-th component of the CORRECT solution in constant time
    2. if a problem has no solution, choose(i) returns mere "garbage", that is, it returns an uncertain value.
- An NP algorithm is an algorithm that has 2 stages:
    1. The first stage is a guessing stage that uses choose() to find a solution to the problem.
    2. The second stage checks the correctness of the solution produced by the first stage. The time of this stage is polynomial in the input size n.
- Template for an NP algorithm:

```
begin
  /* The following for-loop is the guessing stage*/
  for i=1 to N do
    X[i] := choose(i);
  endfor

  /* Next is the verification stage */
  Write code that does not use "choose" and that
  verifies if X[1:N] is a correct solution to the
  problem.
end
```

- Remark: For the algorithm above to be polynomial, the solution size N must be polynomial in n, and the verification stage must be polynomial in n.
- Definition 3 of NP: A problem is said to be NP if there exists an NP algorithm for it.
- Example of an NP problem: The Hamiltonian Cycle (HC) problem
    1. Input: A graph G
    2. Question: Goes G have a Hamiltonian Cycle?
- Here is an NP algorithm for the HC problem:

```
begin
  /* The following for-loop is the guessing stage*/
  for i=1 to n do
    X[i] := choose(i);
  endfor


  /* Next is the verification stage */
  for i=1 to n do
  for j=i+1 to n do
    if X[i] = X[j] then
            return(no);
    endif
  endfor
  endfor
  for i=1 to n-1 do
  if (X[i],X[i+1]) is not an edge then
            return(no);
  endif
  endfor
  if (X[n],X[1]) is not an edge then
  return(no);
  endif

  return(yes);
end
```

- The solution size of HC is O(n), and the time of the verification stage is O(n^2). Therefore, HC is NP.
- The K-clique problem is NP
    1. Input: A graph G and an integer k
    2. Question: Goes G have a k-clique?
- Here is an NP algorithm for the K-clique problem:

```
begin
```

```
  /* The following for-loop is the guessing stage*/
  for i=1 to k do
     X[i] := choose(i);
  endfor

  /* Next is the verification stage */
  for i=1 to k do
       for j=i+1 to k do
          if (X[i] = X[j] or (X[i],X[j]) is not an edge) then
               return(no);
          endif
       endfor
  endfor

  return(yes);
end
```

- The solution size of the k-clique is O(k)=O(n), and the time of the verification stage is O(n^2). Therefore, the k-clique problem is NP.

III. Focus on Yes-No Problems

- Definition: A yes-no problem consists of an instance (or input I) and a yes-no question Q.
- The yes-no version of the HC problem was described above, and so was the yes-no version of the k-clique problem.
- The following are additional examples of well-known yes-no problems.
- The subset-sum problem:
  - Instance: a real array a[1:n]
  - Question: Can the array be partitioned into two parts that add up to the same value?
- The satisfiability problem (SAT):
  - Instance: A Boolean Expression F
  - Question: Is there an assignment to the variables in F so that F evaluates to 1?
- The Treveling Salesman Problem

  The original formulation:
  - Instance: A weighted graph G
  - Question: Find a minimum-weight Hamiltonian Cycle in G.

  The yes-no formulation:

  - Instance: A weighted graph G and a real number d
  - Question: Does G have a Hamiltonian cycle of weight <= d?

IV. Reductions and Transforms

- Notation: If P stands for a yes-no problem, then
    - I_P: denotes an instance of P
    - Q_P: denotes the question of P
    - Answer(Q_P,I_P): denotes the answer to the question Q_P given input I_P
- Let P and R be two yes-no problems
- Definition: A transform (that transforms a problem P to a problem R) is an algorithm T such that:
    1. The algorithm T takes polynomial time
    2. The input of T is I_P, and the output of T is I_R
    3. Answer(Q_P,I_P)=Answer(Q_R,I_R)

Definition: We say that problem problem P reduces to problem R if there exists a transform from P to R.

V. NP-Completeness

- Definition: A problem R is NP complete if
    1. R is NP
    2. Every NP problem P reduces to R
- An equivalent but casual definition: A problem R is NP-complete if R is the "most difficult" of all NP problems.
- Theorem: Let P and R be two problems. If P reduces to R and R is polynomial, then P is polynomial.
- Proof:
    - Let T be the transform that transforms P to R. T is a polynomial time algorithm that transforms I_P to I_R such that

$$\text{Answer}(Q_P,I_P) = \text{Answer}(Q_R,I_R)$$

    - Let A_R be the polynomial time algorithm for problem R. Clearly, A takes as input I_R, and returns as output Answer(Q_R,I_R)
    - Design a new algorithm A_P as follows:

```
Algorithm $A_P$(input: $I_P$)
begin
    $I_R$ := T($I_P$);
    x := $A_R$($I_R$);
    return x;
end
```

    - Note that this algorithm A_P returns the correct answer Answer(Q_P,I_P) because x = A_R(I_R) = Answer(Q_R,I_R) = Answer(Q_P,I_P).
    - Note also that the algorithm A_P takes polynomial time because both T and A_R

The intuition derived from the previous theorem is that if a problem P reduces to problem R, then R is at least as difficult as P.

Theorem: A problem R is NP-complete if

0. R is NP, and
1. There exists an NP-complete problem R_0 that reduces to R

Proof:

- Since R is NP, it remain to show that any arbitrary NP problem P reduces to R.
- Let P be an arbitrary NP problem.
- Since R_0 is NP-complete, it follows that P reduces to R_0
- And since R_0 reduces to R, it follows that P reduces to R (by transitivity of transforms).

The previous theorem amounts to a strategy for proving new problems to be NP complete. Specifically, to prove a new problem R to be NP-complete, the following steps are sufficient:

4. Prove R to be NP
5. Find an already known NP-complete problem R_0, and come up with a transform that reduces R_0 to R.

For this strategy to become effective, we need at least one NP-complete problem. This is provided by Cook's Theorem below.

Cook's Theorem: SAT is NP-complete.

VI. NP-Completeness of the k-Clique Problem

- The k-clique problem was laready shown to be NP.
- It remain to prove that an NP-complete problem reduces to k-clique
- Theorem: SAT reduces to the k-clique problem
- Proof:
  - Let F be a Boolean expression.
  - F can be put into a conjunctive normal form: $F=F_1F_2...F_r$
    where every factor F_i is a sum of literals (a literal is a Bollean variable or its complement)
  - Let k=r and G=(V,E) defined as follows:
    $V=\{<x_i,F_j> \mid x_i$ is a variable in $F_j\}$
    $E=\{(<x_i,F_j> , <y_s,F_t>) \mid j$!=t and x_i != $y_s'\}$
    where y_s' is the complement of y_s
  - We prove first that if F is satisfiable, then there is a k-clique.
  - Assume F is satisfiable
  - This means that there is an assignment that makes F equal to 1
  - This implies that $F_1=1$, $F_2=1$, ... , $F_r=1$
  - Therefore, in every factor F_i there is (at least) one variable assigned 1. Call that variable z_i

  - As a result, $<z_1,F_1>$, $<z_2,F_2>$, ... , $<z_k,F_k>$ is a k-clique in G because they are k distinct nodes, and each pair ($<z_i,F_i>$, $<z_j,F_j>$) forms an edge since the endpoints come from different factors and z_i != z_j' due to the fact that they are both assigned 1.
  - We finally prove that if G has a k-clique, then F is satistiable
  - Assume G has a k-clique $<u_1,F_1>$, $<u_2,F_2>$, ... , $<u_k,F_k>$ which are pairwise adjacent
  - These k nodes come the k fifferent factors, one per factor, becuae no two nodes from the same factor can be adjacent
  - Furthermore, no two u_i and u_j are complements because the two nodes $<u_i,F_i>$ and $<u_j,F_j>$ are adjacent, and adjacent nodes have non-complement first-components.
  - As a result, we can consistently assign each u_i a value 1.
  - This assignment makes each F_i equal to 1 because u_i is one of the additive literals
  - Consequently, F is equal to 1.
- An illustration of the proof will be carried out in class on $F=(x_1 + x_2)(x_1' + x_3)(x_2 + x_3')$

9. Study of Cook's Theorem:-

Course Objective: - Know various advanced design and analysis techniques such as greedy algorithms, dynamic programming & Know the concepts of tractable and intractable problems and the classes P, NP and NP-complete problems.

Outcome: - Analyze deterministic and nondeterministic algorithms to solve complex

Problems

Mapping with PO & PSO

PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
3	2	3	2	1	-	-	-	-	-	1	2

PSO1	PSO2	PSO3	PSO4
2	3	1	1

Cook's Theorem Cook's theorem shows that the satisfiability problem is NP-complete. Without loss of generality, we assume that languages in NP are over the alphabet {0, 1} ∗ .

Lemma 1, useful for the proof, states that we can restrict the form of a computation of a NTM that accepts languages in NP. Lemma 1 If L ∈ NP, then L is accepted by a 1-tape NTM N with alphabet {0, 1} such that for some polynomial p(n), the following properties hold. • N's computation is composed of two phases, the guessing phase and the checking phase. • In the guessing phase, N nondeterministically writes a string ty directly after the input string, and in the checking phase, N behaves deterministically. • N uses at most p(n) tape cells, never moves its head to the left of w, and takes exactly p(n) steps in the checking phase. A Boolean formula f over variable set V is in conjunctive normal form (CNF) if f = ∧ m i=1 ∨ ki j=1 li,j for some values of m and ki , 1 ≤ i ≤ m, where literal li,j is either x or x¯ for some x ∈ V . For each i, the term ∨ ki j=1li,j is called a clause of the formula. f is satisfiable if there exists a truth assignment to the variables in V that sets f to true. CNFSAT is the set of satisfiable Boolean formulas in CNF.

Theorem 1 (Cook's Theorem) CNFSAT is NP-complete. Proof sketch: It is not hard to show that CNFSAT ∈ NP. To prove that CNFSAT is NP complete, we show that for any language L ∈ NP, L ≤p m CNFSAT. Let L ∈ NP and let N be a NTM accepting L that satisfies the properties of Lemma 1. Let the transition function of N be δ. Let the states of N be q0, ..., qr. Let s0, s1, s2 denote 0, 1, t, respectively. Assume that the tape cells are numbered consecutively from the left end of the input, starting at 0. On input w of length n, we show how to construct a formula in CNF form fw, which is satisfiable if and only if w is accepted by N. The variables of fw are as follows: Variables Range Meaning Q[i, k] 0 ≤ i ≤ p(n) At step i of the checking phase, 0 ≤ k ≤ r the state of N is qk. H[i, j] 0 ≤ i ≤ p(n) At step i of the checking phase, 0 ≤ j ≤ p(n) the head of N is on tape square j. S[i, j, l] 0 ≤ i ≤ p(n) At step i of the checking phase, 0 ≤ j ≤ p(n) the symbol in square j is sl . 0 ≤ l ≤ 2 A computation of N naturally corresponds to an assignment of truth values to the variables. Other assignments to the variables may be meaningless. For example, an assignment with Q[i, k] = Q[i, k0] = true, k 6= k 0 , would imply that N is in two different states at step i, which is impossible. Our goal is to construct fw so that it is satisfied only by assignments to the variables that correspond to accepting computations of N on w. The clauses of fw are constructed to ensure that the following conditions are satisfied: 1. At each step i of the checking phase, N is in exactly 1 state. 2. At each step i, the head is on exactly one tape square. 3. At each step i, there is exactly 1 symbol in each tape square. 4. At step 0 of the checking phase, the state is the initial state of N in its checking phase, and the tape contents are w t y for some y. 5. At step p(n) of the checking phase, N is in an accepting state. 6. The configuration of N at the (i + 1)st step follows from that at the i th step, by applying the transition function of N. Consider condition 1. For each i, we have the following clause: Q[i, 0] ∨ Q[i, 1] ∨ ... ∨ Q[i, r]. This clause ensures that the machine is in at least 1 state at step i. We also need clauses to ensure that N is not both in state qj and qj 0: Q[i, j] ∨ Q[i, j0] for each j 6= j 0 , 0 ≤ j, j0 ≤ r. Conditions 2 and 3 are handled similarly. Conditions 4 and 5 are quite easy. Finally, consider condition 6. For each (i, j, k, l) we add clauses that ensure the following: If at step i, the tape head of N is pointing to the j th tape cell, N is in state qk, sl is the symbol under the tape head, and (qk, sl , qk 0, sl 0, X) ∈ δ, where X ∈ {L, R} then at step i + 1, the tape head is pointing to the (j + y) th tape cell where y = 1 if X = R and y = −1 if X = L, N is in state qk 0 and the symbol in cell j is sl 0. The following clauses ensure this: Q[i, k] ∨ H[i, j] ∨ S[i, j, l] ∨ Q[i + 1, k0] Q[i, k] ∨ H[i, j] ∨ S[i, j, l] ∨ H[i + 1, j + y] Q[i, k] ∨ H[i, j] ∨ S[i, j, l] ∨ S[i + 1, j, l0] All of the clauses for condition 1 to 6 can be computed in polynomial time (how many clases are there?). Moreover, w is accepted by N if and only if F_w is satisfiable.

10. Study of Sorting Network:-

Mapping with PO & PSO

PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
3	2	3	2	1	-	-	-	-	-	1	2

PSO1	PSO2	PSO3	PSO4
2	3	1	1

Single-source shortest paths in directed acyclic graphs

By relaxing the edges of a weighted dag (directed acyclic graph) *G = (V, E)* according to a topological sort of its vertices, we can compute shortest paths from a single source in O*(V + E)* time. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

The algorithm starts by topologically sorting the dag to impose a linear ordering on the vertices. If there is a path from vertex *u* to vertex *v*, then *u* precedes *v* in the topological sort. We make just one pass over the vertices in the topologically sorted order. As we process each vertex, we relax each edge that leaves the vertex.

```
DAG-SHORTEST-PATHS(G,w, s)
1 topologically sort the vertices of G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 for each vertex u, taken in topologically sorted order
4     do for each vertex v ∈ Adj[u]
5         do RELAX(u, v,w)
```

Figure shows the execution of this algorithm. The running time of this algorithm is easy to analyze. As shown in, the topological sort of line 1 can be performed in $O(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $O(V)$ time. There is one iteration per vertex in the **for** loop of lines 3–5. For each vertex, the edges that leave the vertex are each examined exactly once. Thus, there are a total of $|E|$ iterations of the inner **for** loop of lines 4–5. (We have used an aggregate analysis here.) Because each iteration of the inner **for** loop takes $O(1)$ time, the total running time is $O(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.
Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum Shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA(G,w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 S ← ∅
3 Q ← V[G]
4 while Q _= ∅
5     do u ← EXTRACT-MIN(Q)
6         S ← S ∪ {u}
7         for each vertex v ∈ Adj[u]
8             do RELAX(u, v,w)
```

Dijkstra's algorithm relaxes edges as shown in Line 5 performs the usual initialization of d and π values, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue Q to contain all the vertices in V; since $S = \emptyset$ at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, a vertex u is extracted from $Q = V - S$ and added to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 7–8 relax each edge (u, v) leaving u,
thus updating the estimate $d[v]$ and the predecessor $\pi[v]$ if the shortest path to v can be improved by going through u. Observe that vertices are never inserted into Q after line 3 and that each vertex is extracted from Q and added to S exactly once, so that the **while** loop of lines 4–8 iterates exactly $|V|$ times.
Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal

results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths. The key is to show that each time a vertex u is added to set S, we have $d[u] = \delta(s, u)$.

Experiment # B1

Experiment Title:-

Implement a C program to solve fractional knapsack problem using greedy method.

Objective

1. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
2. How a number of algorithms exists for fundamental problems in computer science and engineering work and compare with one another, and how there are still some problems for which it is unknown whether there exist efficient algorithms, and how to design efficient algorithms.
3. Augment various data structures (trees and arrays) to support specific applications.
4. Know various advanced design and analysis techniques such as greedy algorithms, dynamic programming & Know the concepts of tractable and intractable problems and the classes P, NP and NP-complete problems.

Course Outcomes

CO2. Identify the differences in design techniques and apply to solve optimization problems.

CO3. Apply algorithms for performing operations on graphs and trees.

CO4. Solve novel problems, by choosing the appropriate algorithm design technique for their solution and justify their selection

CO5. Analyze deterministic and nondeterministic algorithms to solve complex Problems

Mapping with PO & PSO

PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
2	2	3	2	1	1	-	-	-	-	1	3

PSO1	PSO2	PSO3	PSO4
2	1	3	1

Theory:

Greedy method or technique is used to solve Optimization problems. A solution that can be maximized or minimized is called Optimal Solution. In Knapsack problem we are given:

1) n objects
2) Knapsack with capacity m,
3) An object i is associated with profit W_i,
4) An object i is associated with profit P_i ,
5) when an object i is placed in knapsack we get profit P_i
Here objects can be broken into pieces (X_i Values). The Objective of Knapsack problem is to maximize the profit .

Algorithm:

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

Implementation:

```
#include<stdio.h>
#include<conio.h>
main()
{
        clrscr();
        int n,m,i,u;
        int p[20],w[20];
        float x[20];
        float optimal=0.0;
        printf("Enter number of objects:");
        scanf("%d",&n);
        printf("Enter capacity of KnapSack:");
        scanf("%d",&m);
        printf("Enter profits in decreasing order of Pi/Wi:");
        for(i=1;i<=n;i++)
                scanf("%d",&p[i]);
        printf("Enter Weights in decreasing order of Pi/Wi:");
        for(i=1;i<=n;i++)
```

```
                scanf("%d",&w[i]);
        for(i=1;i<=n;i++)
                x[i]=0.0;
        u=m;
        for(i=1;i<=m;i++)
        {
                if(w[i]>u)
                        break;
                else
                        x[i]=1.0;
                u=u-w[i];
        }
        if(i<=n)
                x[i]=(float)u/w[i];
        printf("The x values are\n");
        for(i=1;i<=n;i++)
                printf("%f\t",x[i]);
        for(i=1;i<=n;i++)
                optimal=optimal+p[i]*x[i];
        printf("\nOptimal Solution is %f",optimal);
        getch();
}
```

Result:

```
Enter the number of objects: 3
Enter the capacity of Knapscak      :       20
Enter profits in decreasing order of pi/wi : 16 15 14
Enter weights in decreasing order of pi/wi : 10 10 10
The x-values are: 1.000000 1.000000 0.000000
Optimal Solution is : 31.000000
```

The optimal solution for a Knapsack problem is obtained.

Experiment # B2

Experiment Title:-

Implement a C program to solve matrix chain multiplication problem using dynamic programming

Objective

1. To analyze and design algorithms and to appreciate the impact of algorithm design in practice.
2. Ensures that students understand how the worst-case time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms,
3. How a number of algorithms exists for fundamental problems in computer science and engineering work and compare with one another, and how there are still some problems for which it is unknown whether there exist efficient algorithms, and how to design efficient algorithms.
4. Use different computational models (e.g., divide-and-conquer), order notation and various complexity measures (e.g., running time, disk space) to analyze the complexity/performance of different algorithms.
5. Augment various data structures (trees and arrays) to support specific applications.
6. Know various advanced design and analysis techniques such as greedy algorithms, dynamic programming & Know the concepts of tractable and intractable problems and the classes P, NP and NP-complete problems.

Course Outcomes

CO2. Identify the differences in design techniques and apply to solve optimization problems.

CO3. Apply algorithms for performing operations on graphs and trees.

CO4. Solve novel problems, by choosing the appropriate algorithm design technique for their solution and justify their selection

CO5. Analyze deterministic and nondeterministic algorithms to solve complex Problems

Mapping with PO & PSO

PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
2	2	3	2	1	1	-	-	-	-	1	3

PSO1	PSO2	PSO3	PSO4
2	1	3	1

Theory:

Let A,B &C are 3 matrices , since matrix multiplication is associative (AB)C=A(BC).The objective in matrix chain multiplication is to find out the paranthasization with less number of operations.

```
A=|1 2|
  |3 4|
B=|2 2 1|
  |1 1 1|
C=|4 1|
  |1 1|
  |1 1|

(AB)C
AB = | 4   4  3|
     |10 10  7|
(AB)C=|23 11|
      |57 27|
A(BC)
BC==|11 5|
    |  6 3|
A(BC)=|23 11|
      |57 27|
```

Paranthasization cost calculation:

Total cost:

(AB)C = (2*2*3)+(2*3*2)= 24

A(BC) = (2*3*2)+(2*2*2) =20

TechReg-Edu

26

Tree diagram for (AB)C

Tree diagram for A(BC)

Algorithm:

```
// n is the n umber of matrices
// l is chain length
// Matrix A_i has the dimension p[i-1] x p[i]
Matrix-Chain-Order(int p[])
{
        n = p.length-1;
        for (i = 1; i <= n; i++)
                m[i,i] = 0;
        for (l=2; l<=n; l++)
        {
                for (i=1; i<=n-l+1; i++)
                {
                        j = i+l-1;
                        m[i,j] = MAXINT;
                        for (k=i; k<=j-1; k++)
                        {
                                q = m[i,k]+ m[k+1,j] + p[i-1]*p[k]*p[j];
                                if (q < m[i,j])
                                {
                                        m[i,j] = q;
                                        s[i,j] = k;
                                }
                        }
                }
        }
}
```

Implementation:

```
#include<stdio.h>
#include<conio.h>
#define MAX 15
#define INF 4294967295
int num,p[MAX+1],n;
void print(int [][MAX],int,int);
void matrixchainorder();
void setdata();
void printorder();
void print(int s[MAX][MAX],int i,int j)
{
if(i==j)
printf("A%d",num++);
else
{
```

```
printf("(");
print(s,i,s[i][j]);
printf(" x ");
print(s,s[i][j]+1,j);
printf(" ) ");
}
}
void matrixchainorder()
{
unsigned int q;
unsigned long m[MAX][MAX]={0};
int s[MAX][MA
X]={0};
int l,j,i,k;
for(l=2;l<=n;l++)
for(i=1;i<=n-l+1;i++)
{
j=i+l-1;
m[i][j]=INF;
for(k=i;k<j;k++)
{
q=m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
if(q < m[i][j])
{
m[i][j]=q;
s[i][j]=k;
}
}
}
printf("Number of Multiplications are %d ",m[1][n]);
num=1;
printf("Order of Multiplication is: ");
print(s,1,n);
}
void setdata()
{
int i;
printf("\nEnter number of matrices: ");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
printf("Enter %d matrix size:",i);
scanf("%d%d",&p[i-1],&p[i]);
}
}
```

```
void printorder()
{
int i,j;
matrixchainorder();
}
void main()
{
setdata();
printorder();
}
```

Result:

```
Enter number of matrices: 3
Enter 1 matrix size:2 2
Enter 2 matrix size:2 3
Enter 3 matrix size:3 2
Number of Multiplications are 20 Order of Multiplication is: (A1 x (A2 x A3 ) )
```

Matrix Chain Multiplication is implemented

References:

1. Horowitz and Sahani, ―Fundamentals of Data Structures, Golgotia's Publications Pvt Ltd Delhi India
2. Thomas H. Cormen et al. - Introduction to Algorithms
3. G.S. Baluja, ― Data Structure using C.
4. Reema Thareja- Data Structure Using C, Oxford Edition

5. https://cprogrammingcodes.blogspot.com