LAB MANUAL

Of

Analysis Of Algorithm

Academic Year	2018-2019
Semester	I
Department	Computer Engineering
Syllabus Revision Year	2016
Last Update On	02-01-2019

RIZVI EDUCATION SOCIETY

Rizvi College of Engineering

Off Carter Road, Bandra West, Mumbai- 400050

Program Objectives

PO1	**Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem analysis:** Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Objectives

PSO1	**Professional & Problem-Solving Skills:** The ability to understand and analyse the problem and develop algorithms & programs for the same, with efficient design and varying complexity using lifelong learning and principles of computer engineering in the fields of multimedia, web design, data management & analytics, networking & security, machine learning & artificial intelligence etc. To apply standard practices and strategies in software & hardware project development using open-source programming environments to deliver a quality product for business success.
PSO2	**Successful Career and Entrepreneurship:** Ability to acquire logical thinking capability and problem solving skill in computer engineering as well as diverse fields to achieve better overall prospects of the employment or to be a successful entrepreneur and work as an individual and as well as in a team to achieve solution within the budget and to communicate effectively with the engineering community and the society and also to have a zest for higher studies.

➢ Course Objectives

To provide mathematical approach for Analysis of Algorithms
To solve problems using various strategies
To analyse strategies for solving problems not solvable in polynomial

➢ Course Outcomes

CO1	Analyze the running time and space complexity of algorithms.
CO2	Describe, apply and analyze the complexity of divide and conquer strategy.
CO3	Describe, apply and analyze the complexity of greedy strategy.
CO4	Describe, apply and analyze the complexity of dynamic programming strategy.
CO5	Explain and apply backtracking, branch and bound and string matching techniques to deal with some hard problems.
CO6	Describe the classes P, NP, and NP-Complete and be able to prove that a certain problem is NP-Complete.

INDEX

Sr. No.	Name of Experiment	Page No.
1	Write a Program To implement selection sort	5-7
2	Write a Program To implement binary search	7-12
3	Write a Program To implement merge sort	13-18
4	Write a Program To implement quick sort	19-24
5	Write a Program To implement Insertion sort	25-29
6	Write a Program To implement knapsack algorithm	30-35
7	Write a Program To implement Prim’s algorithm	36-41
8	Write a Program To implement n- queens problem	41-46
9	Write a Program To implement Longest common Sequence	47-51
10	Write a Program To implement Travelling Sales Man Problem	51-55
11	Write a Program to implement Job Sequencing	56-59
12	Write a Program to implement naïve string matching	60-63
13	Write a Program to implement Rabin Karp	64-67
14	Write a program to implement to Sort Elements in Lexicographical Order (Dictionary Order)	68-72
15	Write a program to perform Bellman Ford's Algorithm	73-76

Attainment of PO’s & PSO’s

Sr. No.	Name of Experiment	PO’s Attained	PSO’s Attained
1	Write a Program To implement selection sort	PO3,PO4	PSO1
2	Write a Program To implement binary search	PO3	PSO1
3	Write a Program To implement merge sort	PO3,PO2	PSO1
4	Write a Program To implement quick sort	PO3	PSO1
5	Write a Program To implement Insertion sort	PO5	PSO1 ,PSO2
6	Write a Program To implement knapsack algorithm	PO3,P012	PSO1
7	Write a Program To implement Prim’s algorithm	PO3,PO12	PSO1, PSO2
8	Write a Program To implement n- queens problem	PO2,PO3	PSO2
9	Write a Program To implement Longest common Sequence	PO2,PO4,PO3	PSO1, PSO2
10	Write a Program To implement Travelling Sales Man Problem	PO3,PO4,POP6	PSO1, PSO2
11	Write a Program to implement Job Sequencing	PO2,PO4,PO3	PSO1, PSO2
12	Write a Program to implement naïve string matching	PO3,PO4,POP6	PSO2
13	Write a Program to implement Rabin Karp	PO2,PO4,PO3	PSO1
14	Write a program to implement to Sort Elements in Lexicographical Order (Dictionary Order)	PO2,PO3	PSO1, PSO2
15	Write a program to perform Bellman Ford's Algorithm	PO3,P012,PO11	PSO1, PSO2

Experiment No. 1.A

Title: *Write a Program to implement Selection Sort*

Date : ____ / ____ / ________

Subject In-charge Sign:

…………………………………..

Experiment No. 1.A

Aim: Write a Program to implement Selection Sort

Language Used: C

Prerequisite: Knowledge of Sorting

Theory:

Selection sort algorithm starts by comparing first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, leave the elements as it is. Then, again first element and third element are compared and swapped if necessary. This process goes on until first and last element of an array is compared. This completes the first step of selection sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, comparison starts from second element because after first step, the required number is automatically placed at the first (i.e, In case of sorting in ascending order, smallest element will be at first and in case of sorting in descending order, largest element will be at first.). Similarly, in third step, comparison starts from third element and so on.

Procedure/Algorithm:

```
procedure selection sort
  list  : array of items
  n     : size of list


  for i = 1 to n - 1
  /* set current element as minimum*/
    min = i


    /* check the element to be minimum */


    for j = i+1 to n
      if list[j] < list[min] then
```

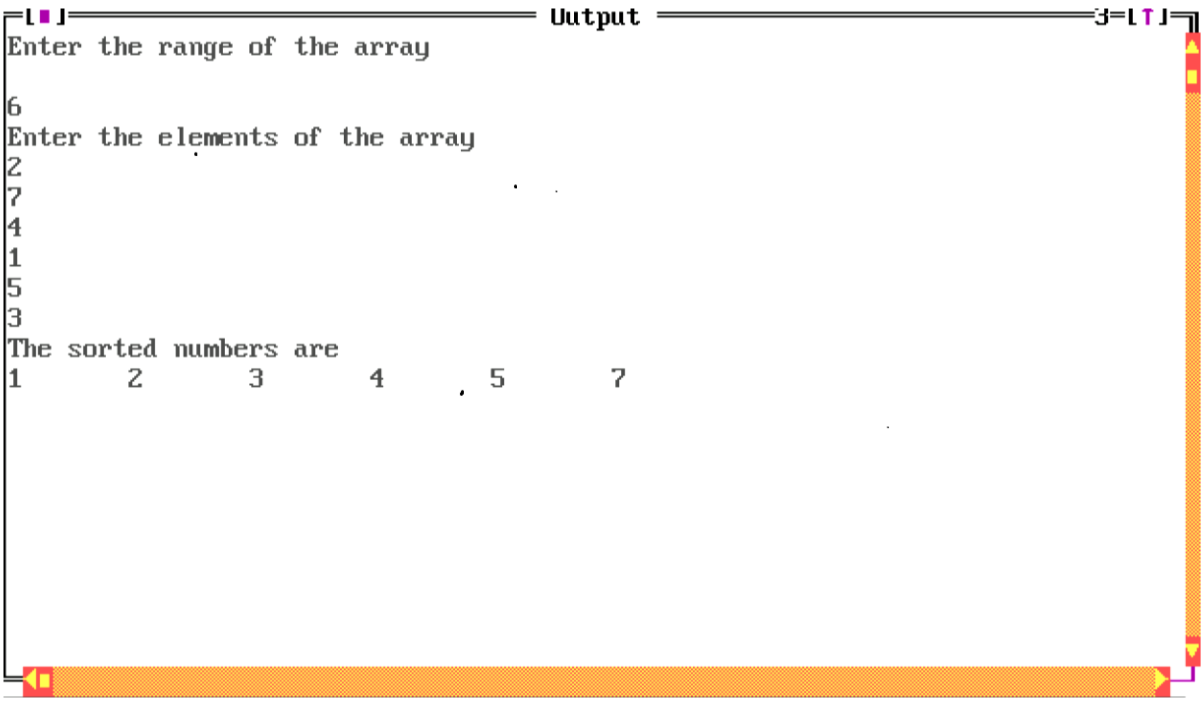
```
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Program:

```
#include<stdio.h>
void SS(int a[],int n)
{
		int i,j,min,temp;
		for(i=0;i<n-1;i++)
		{
				min=i;
				for(j=(i+1);j<n;j++)
				{
						if(a[j]<a[min])
						{
								min=j;
						}
				}
				temp=a[i];
				a[i]=a[min];
				a[min]=temp;
		}
		printf("The sorted numbers are\n");
		for(i=0;i<n;i++)
		{
				printf("%d\t",a[i]);
		}
}
void main()
{
		int a[20],i,n;

		printf("Enter the range of the array\n");
```

```
            scanf("%d",&n);
            printf("Enter the elements of the array\n");
            for(i=0;i<n;i++)
            {
                        scanf("%d",&a[i]);
            }
            SS(a,n);
}
```

Result/Output:



```
Enter the range of the array

6
Enter the elements of the array
2
7
4
1
5
3
The sorted numbers are
1       2       3       4       5       7
```

Pre Lab Questions:

Question 1: Explain Sorting?
Question 2: What are the various types of sorting?

Post Lab Questions:

Question 1: Define Selection sort**?**
Question 2: What is the algorithm for Selection sort**?**

Conclusion:

Thus, We have studied the algorithm of Selection Sort and implemented it

Experiment No. 1.B

***Title** : Write a Program To implement binary search .*

***Date :** ____ / ____ / ________*

Subject In-charge Sign:

……………………………..

.

Experiment No. 1.B

Aim: Write a Program To implement binary search

Language Used: C

Prerequisite: Knowledge of Tree

Theory:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Procedure/Algorithm:

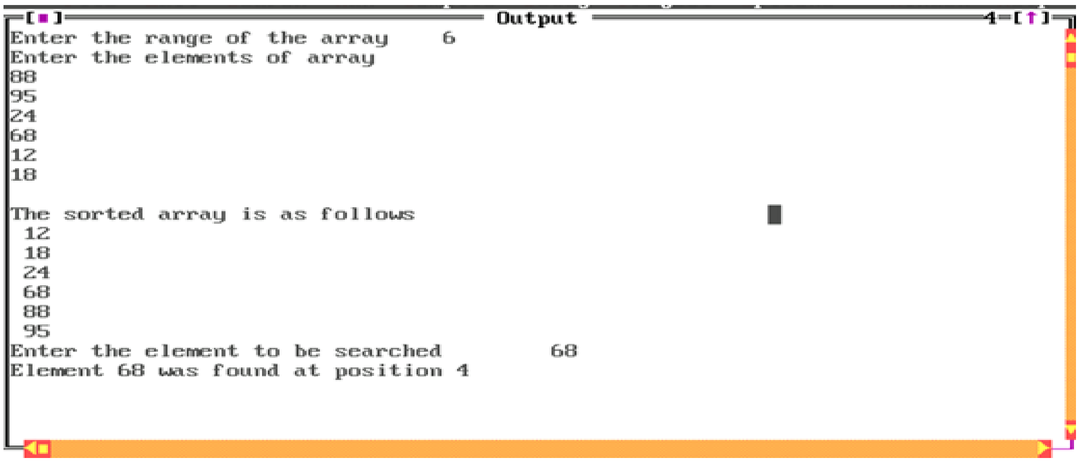
```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
```

```
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

Program:

```
#include<stdio.h>

int binary(int x[],intn,int key)
{
        inti,mid,low=0,high=n-1;
        while(low<=high)
        {
                mid=(low+high)/2;
                if(key==x[mid])
                {
                        return mid+1;
                }
                else if(key>x[mid])
                {
                        low=mid+1;
                }
                else
                {
                        high=mid-1;
                }
        }
        return -1;
}

void sort(intn,int a[20])
{
        inti,j,t;
        for(i=1;i<n;i++)
        {
                t=a[i];
                j=i-1;
                while(t<a[j]&&j>=0)
                {
                        a[j+1]=a[j];
                        j--;
                }
                a[j+1]=t;
        }
}

void main()
{
        int p, r,a[20],i,key;

          printf("Enter the range of the array\t");
          scanf("%d",&r);
          printf("Enter the elements of array\n");
        for(i=0;i<r;i++)
        {
                scanf("%d",&a[i]);
        }
        sort(r,a);
          printf("\nThe sorted array is as follows\n");
        for(i=0;i<r;i++)
        {
                printf(" %d\n",a[i]);
        }
           printf("Enter the element to be searched\t");
           scanf("%d",&key);
           p=binary(a,r,key);
        if(p==-1)
        {
                printf("Element %d was not found",key);
        }
        else
        {
                printf("Element %d was found at position %d",key,p);
        }

}
```

Result/Output:



```
Enter the range of the array    6
Enter the elements of array
88
95
24
68
12
18

The sorted array is as follows
 12
 18
 24
 68
 88
 95
Enter the element to be searched        68
Element 68 was found at position 4
```

Pre Lab Questions:

Question : Explain Searching?

Question 2: What are the various types of trees?

Post Lab Questions:

1. Question 1: Define binary search?
2. Question 2: What is the algorithm for binary search**?**

Conclusion:

Thus, we have studied and implemented Binary Search

Experiment No. 1.C

Title: *Write a Program to implement merge sort*

Date : ____ / ____ / ________

Subject In-charge Sign:

…………………………………

Experiment No. 1.C

Aim: Write a Program to implement merge sort.

Language Used: C

Prerequisite: Knowledge of Sorting

Theory:

In computer science, **merge sort** (also commonly spelled **mergesort**) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945.A detailed description and analysis of bottom-up mergesort appeared in a report byGoldstine and Neumann as early as 1948.

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list

Example :

```
Start       : 3--4--2--1--7--5--8--9--0--6
Select runs : 3--4  2  1--7  5--8--9  0--6
Merge       : 2--3--4  1--5--7--8--9  0--6
Merge       : 1--2--3--4--5--7--8--9  0--6
Merge       : 0--1--2--3--4--5--6--7--8--9
```

Procedure/Algorithm:

```
// Array A[] has the items to sort; array B[] is a work array.
TopDownMergeSort(A[],B[],n)
{
TopDownSplitMerge(A,0,n,B);
}

// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
TopDownSplitMerge(A[],iBegin,iEnd,B[])
{
if(iEnd-iBegin<2)// if run size == 1
return;//   consider it sorted
// recursively split runs into two halves until run size == 1,
// then merge them and return back up the call chain
iMiddle=(iEnd+iBegin)/2;// iMiddle = mid point
TopDownSplitMerge(A,iBegin,iMiddle,B);// split / merge left  half
TopDownSplitMerge(A,iMiddle,iEnd,B);// split / merge right half
TopDownMerge(A,iBegin,iMiddle,iEnd,B);// merge the two half runs
CopyArray(B,iBegin,iEnd,A);// copy the merged runs back to A
}

//  Left half is A[iBegin :iMiddle-1].
// Right half is A[iMiddle:iEnd-1   ].
TopDownMerge(A[],iBegin,iMiddle,iEnd,B[])
{
i=iBegin,j=iMiddle;

// While there are elements in the left or right runs...
for(k=iBegin;k<iEnd;k++){
// If left run head exists and is <= existing right run head.
if(i<iMiddle&&(j>=iEnd||A[i]<=A[j])){
B[k]=A[i];
i=i+1;
}else{
B[k]=A[j];
j=j+1;
}
}
}

CopyArray(B[],iBegin,iEnd,A[])
{
for(k=iBegin;k<iEnd;k++)
A[k]=B[k];
}
```

Program:

```
#include<stdio.h>
void merge (int[], int, int, int);

void part (int[], int, int);

void
main ()
{

intarr[30];

inti, size;
```

```
printf ("\n\t------- Merge sorting method -------\n\n");

printf ("Enter total no. of elements : ");

scanf ("%d", &size);

for (i = 0; i < size; i++)

    {

printf ("Enter %d element : ", i + 1);

scanf ("%d", &arr[i]);

}

part (arr, 0, size - 1);

printf ("\n\t------- Merge sorted elements -------\n\n");

for (i = 0; i < size; i++)

printf ("%d ", arr[i]);

getch ();

return;

}


void
part (intarr[], intmin, int max)
{

int mid;

if (min < max)

    {

mid = (min + max) / 2;

part (arr, min, mid);

part (arr, mid + 1, max);

merge (arr, min, mid, max);

}
```

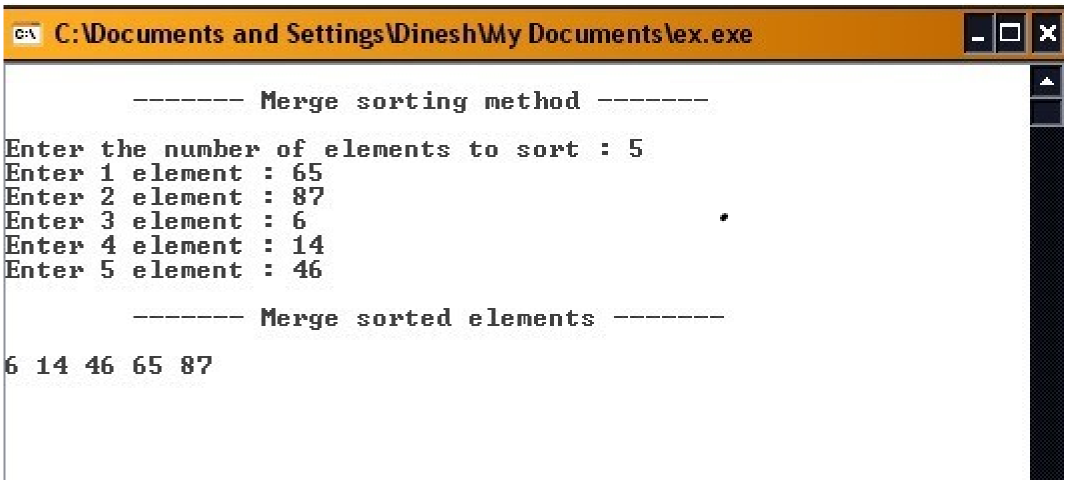
```
}


void
merge (intarr[], intmin, intmid, int max)
{

inttmp[30];

inti, j, k, m;

j = min;

m = mid + 1;

for (i = min; j <= mid && m <= max; i++)
   {

if (arr[j] <= arr[m])

            {

tmp[i] = arr[j];

j++;

}
    else
            {

tmp[i] = arr[m];

m++;

}

}

if (j > mid)

   {

for (k = m; k <= max; k++)

            {

tmp[i] = arr[k];

i++;
}

}
 else
   {

for (k = j; k <= mid; k++)

            {

tmp[i] = arr[k];

i++;

}

}

for (k = min; k <= max; k++)

arr[k] = tmp[k];

}
```

Analysis:

In sorting n objects, merge sort has an average and worst-case performance of $O(n \log n)$. If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.
In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than $(n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1)$, which is between $(n \lg n - n + 1)$ and $(n \lg n + n + O(\lg n))$.

Selection	Comparison
1. BEST CASE COMPLEXITY:	O(nlog n)
2. AVERAGE CASE COMPLEXITY:	O(nlog n)
3. WORST CASE COMPLEXITY:	O(nlog n)

Result/Output:



```
        ------- Merge sorting method -------

Enter the number of elements to sort : 5
Enter 1 element : 65
Enter 2 element : 87
Enter 3 element : 6
Enter 4 element : 14
Enter 5 element : 46

        ------- Merge sorted elements -------

6 14 46 65 87
```

Pre Lab Questions:

Question 1 : Explain Sorting?

Question 2 : What are the various types of sorting?

Post Lab Questions:

1. Question 1: Define Merge sort**?**
2. Question 2: What is the algorithm for Merge sort**?**

Conclusion: Thus, we have studied and implemented Merge Search

Experiment No. 2.A

Title: *Write a Program to implement quick sort*

Date : ____ / ____ / ________

Subject In-charge Sign:

…………………………………..

Experiment No. 2.A

Aim: Write a Program to implement quick sort

Language Used: C

Theory:

Quicksort (sometimes called **partition-exchange sort**) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959, with his work published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heap sort.
Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.
Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons, though this behavior is rare.

Conceptually, a Quick sort works as follows:
Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

1. Pick an element, called a **pivot**, from the array.
2. **Partitioning**: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. **Recursively** apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Example:

Input :	[13 81 92 65 43 31 57 26 75 0]
Pivot :	65
Partition :	[13 0 26 43 31 57] **65** [92 75 81]
Pivot :	31 81
Partition :	[13 0 26] **31** [43 57] **65** [75] **81** [92]
Pivot :	13
Partition :	[0] **13** [26] **31** [43 57] **65** [75] **81** [92]
Combine:	[0 13 26] **31** [43 57] **65** [75 81 92]
Combine:	[0 13 26 31 43 57] **65** [75 81 92]
Combine:	[0 13 26 31 43 57 65 75 81 92]

Procedure/Algorithm:

```
Quick Sort:
Step 1: Initialize x as array containing values to be sorted
Step 2: Initialize 'first' and 'last' as starting and ending index of array
Step 3: if (first >= last) Go to Step 9
Step 4: Set pivot and i as first, j as last
Step 5:  while (i < j) do the following:
        Step 5.1: while x[i] <= x[pivot] and i < last, increment i by 1
        Step 5.2: while x[j] > x[pivot] , decrement j by 1
        Step 5.3: if (i < j) , Swap x[i] and x[j]
Step 6: Swap x[pivot] and x[j]
Step 7: CALL quicksort with 'first' as 'first' and 'last' as j-1
Step 8: CALL quicksort with 'first' as 'j+1 and 'last' as 'last'
Step 9: Stop
        }
}
```

Program:

```
#include<stdio.h>
void quicksort(int x[10], int first, int last) {
        int pivot, j, temp, i;
        if (first < last) {
                pivot = i = first;
                j = last;
                while(i < j) {
                        while(x[i] <= x[pivot] && i < last)
                                i++;
                        while(x[j] > x[pivot])
                                j--;
                        if (i < j) {
                                temp = x[i];
                                x[i] = x[j];
                                x[j] = temp;
                        }
                }
                temp = x[pivot];
                x[pivot] = x[j];
                x[j] = temp;
                quicksort(x, first, j-1);
                quicksort(x, j+1, last);
        }
}
void main() {
        int x[20], size, i;
        printf("Enter size of the array: ");
        scanf("%d", &size);
        printf("Enter %d elements: ",size);
        for (i=0;i<size;i++)
                scanf("%d", &x[i]);

        printf("Entered Array is : \n");
        for (i = 0;i < size; i++)
                printf("\t%d", x[i]);
        quicksort(x, 0, size-1);

        printf("\nSorted elements: \n");
        for (i = 0;i < size; i++)
                printf("\t%d", x[i]);

}
```

Analysis:

QUICK SORT	Comparison
4. BEST CASE COMPLEXITY:	O(nlog(n))
5. AVERAGE CASE COMPLEXITY:	O(nlog(n))
6. WORST CASE COMPLEXITY:	$O(n^2)$

Result/Output:

```
Enter the no. of Cities: 4

 Enter the Cost if path Exist Between cities.:{c1,c2}.Else Enter 0

Cities   Cost

        0 to 1: 10
        0 to 2: 6
        0 to 3: 3
        1 to 2: 11
        1 to 3: 8
        2 to 3: 5

        Possible Path 1: 0 1 2 3 : Cost=29
        Possible Path 2: 0 1 3 2 : Cost=29
        Possible Path 3: 0 2 1 3 : Cost=28
        Possible Path 4: 0 2 3 1 : Cost=29
        Possible Path 5: 0 3 1 2 : Cost=28
        Possible Path 6: 0 3 2 1 : Cost=29
                Minimum Cost is 28 and the Paths are

        Minimum Path: 0 2 1 3
        Minimum Path: 0 3 1 2
```

Pre Lab Questions:

Question 1 : Explain Sorting?

Question 2 : What are the various types of sorting?

Post Lab Questions:

1. Question 1: Define Quick sort**?**
2. Question 2: What is the algorithm for Quick sort**?**

Conclusion:

Thus, we have studied and implemented Quick sort

Experiment No. 2.B

Title: *Write a Program to implement Insertion sort*

Date : ____ / ____ / ________

Subject In-charge Sign:

……………………………….

Experiment No. 2.B

Aim: Write a Program to implement Insertion sort

Language Used: C

Theory:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2), where **n** is the number of items.

Procedure/Algorithm:

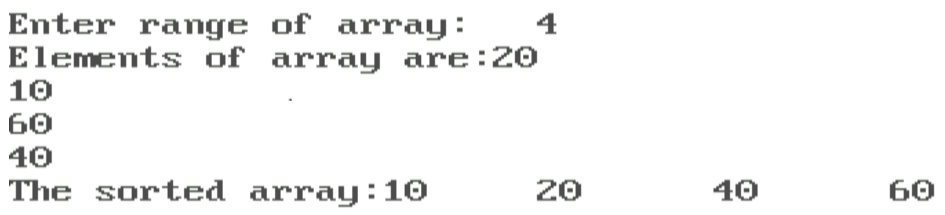
```
// Sort an arr[] of size n
insertionSort(arr, n)
Loop from i = 1 to n-1.
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]
```

Program:

```
#include<stdio.h>

void main()
{
		int i,j,n,temp,x[50];
		printf("Enter range of array:\t");
		scanf("%d",&n);
		printf("Elements of array are:");
		for(i=0;i<n;i++)
		{
			scanf("%d",&x[i]);
		}
		for(i=0;i<n;i++)
		{
			temp=x[i];
			j=i-1;
			while((temp<x[j])&& (j>=0))
			{
				x[j+1]=x[j];
```

```
                        j=j-1;
                }
                x[j+1]=temp;
        }
        printf("The sorted array:");
        for(i=0;i<n;i++)
        {
                printf("%d\t",x[i]);
        }
}
```

Result/Output:



```
Enter range of array:   4
Elements of array are:20
10
60
40
The sorted array:10     20      40      60
```

Pre Lab Questions:

Question 1 : Explain Sorting?

Question 2 : What are the various types of sorting?

Post Lab Questions:

1. Question 1: Define insertion sort**?**
2. Question 2: What is the algorithm for insertion sort**?**

Conclusion:

Thus, we have studied and implemented insertion sort

Experiment No. 3

Title: *Write a Program To implement knapsack algorithm*

Date : ____ / ____ / ________

Subject In-charge Sign:

………………………………..

Experiment No. 3

Aim : Write a Program To implement knapsack algorithm

Language Used: C

Theory:

The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography, applied mathematics, and daily fantasy sports.

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of mathematician Tobias Dantzig (1884–1956), and refers to the commonplace problem of packing your most valuable items without overloading your luggage.

Conceptually, a Knapsack Problem can be solved by:

George Dantzig proposed a greedy approximation algorithm to solve the unbounded knapsack problem. His version sorts the items in decreasing order of value per unit of weight, **v_i / w_i**. It then proceeds to insert them into the sack, starting with as many copies as possible of the first kind of item until there is no longer space in the sack for more. Provided that there is an unlimited supply of each kind of item, if **m** is the maximum value of items that fit into the sack, then the greedy algorithm is guaranteed to achieve at least a value of **m/2** . However, for the bounded problem, where the supply of each kind of item is limited, the algorithm may be far from optimal.

Example:

Consider 5(**n**) items and Total Capacity(**m**) as 15

Item have following Weights (**W**)and Profits(**P**):

W_i	12	1	4	1	2
P_i	4	2	10	1	2

Sorting the Profit and Weights according to **P/W** ratio

W_i	4	1	1	2	12
P_i	10	2	1	2	14
P_i/W_i	2.5	2	1	1	0.33

We can calculate the fractional Knapsack as Follows:

Sr. No	Remaining Weight	x (Fraction taken from weight)	Profit
1	15 – 4 = **11**	**1**	**10**
2	11 – 1 = **10**	**1**	**2**
3	10 – 1 = **9**	**1**	**1**
4	9 – 2 = **7**	**1**	**2**
5	7 – 12*(7/12) = **0**	**7/12**	**14**

Profit can be calculated as:

1.1.1 **Σ(x*p)** = 1*10 + 1*2 + 1*1 + 1*2 + (7/12)*14

Maximum Profit = 23.1667

Procedure/Algorithm:

```
Knapsack(wt, profit, capacity) {
// w = Array containing weight
// v = Array containing Profit
// capacity  = Total Capacity
        for i = 1 to n do
                x[i] = 0
        weight = 0
        i = 0;
        total_profit = 0;
        while weight < capacity do
                if (weight + wt[i] ≤ capacity) then
                        x[i] = 1
                        weight = weight + wt[i]
                else
                        x[i] = (wt[i] – weight) / wt[i]
                        weight = capacity
                endif
                total_profit = total_profit + profit[i]*x[i];
        endwhile
print total_profit
}
```

Program:

```
#include<stdio.h>
#include<math.h>
void knapsack(int n, float weight[], float profit[], float capacity)
{
        float x[20], tp = 0;
        int i, j, u;
        u = capacity;
        for (i = 0; i < n; i++)
                x[i] = 0.0;
        for (i = 0; i < n; i++) {
                if (weight[i] > u)
                        break;
                x[i] = 1.0;
                tp = tp + profit[i];
                u = u - weight[i];
        }

        if (i < n)
                x[i] = u / weight[i];
        tp = tp + (x[i] * profit[i]);
        printf("\nThe result vector is: ");
        for (i = 0; i < n; i++)
                printf("%f\t", x[i]);
        printf("\nMaximum profit is: %f", tp);
}
void main()
{
        float weight[20], profit[20], capacity;
        int num, i, j;
        float ratio[20], temp;
        printf("\nEnter the no. of objects: ");
        scanf("%d", &num);
        printf("\nEnter the weights and profit of each object:\n");
        for (i = 0; i < num; i++)
                scanf("%%f", &weight[i], &profit[i]);
        printf("\nEnter the capacity of knapsack: ");
        scanf("%f", &capacity);
        for (i = 0; i < num; i++)
                ratio[i] = profit[i] / weight[i];
        for (i = 0; i < num; i++)
                for (j = i + 1; j < num; j++)
                        if (ratio[i] < ratio[j]) {
                                temp = ratio[j];
                                ratio[j] = ratio[i];
                                ratio[i] = temp;
                                temp = weight[j];
                                weight[j] = weight[i];
                                weight[i] = temp;
                                temp = profit[j];
                                profit[j] = profit[i];
                                profit[i] = temp;
                        }
        knapsack(num, weight, profit, capacity);
}
```

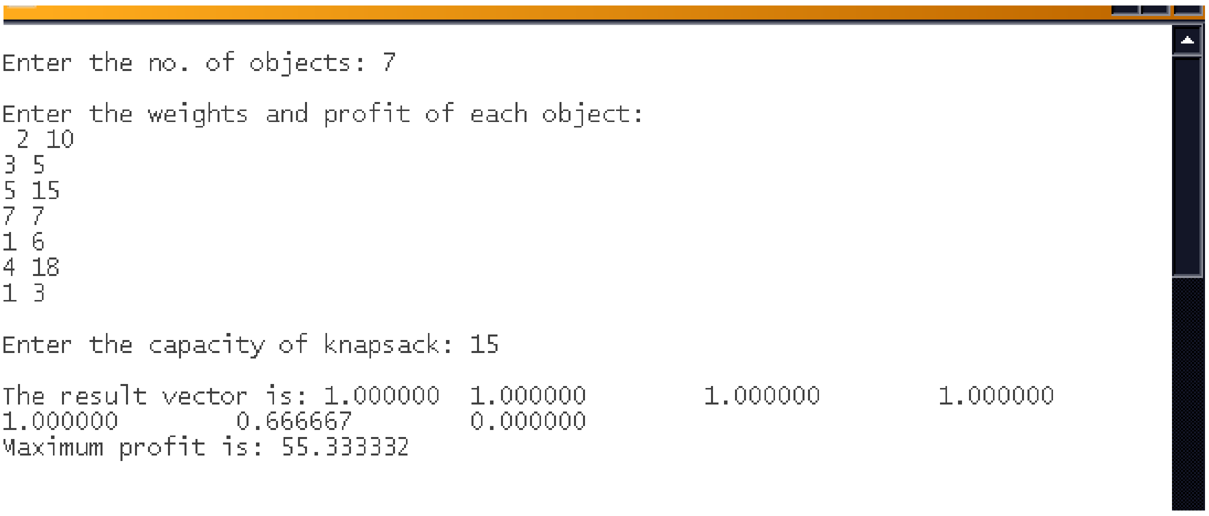
Analysis:

Given an instance of Knapsack Problem with set S of n items, we can construct a maximum benefit subset of S (allowing for fractional amount of items) in O(nlog(n)) time

O(n^2) is time complexity taken for sorting P/W Ratio,

Selection	Comparison
1. BEST CASE COMPLEXITY:	O(nlog(n))
2. AVERAGE CASE COMPLEXITY:	O(nlog(n))
3. WORST CASE COMPLEXITY:	O(nlog(n))

Result/Output:



```
Enter the no. of objects: 7

Enter the weights and profit of each object:
 2 10
3 5
5 15
7 7
1 6
4 18
1 3

Enter the capacity of knapsack: 15

The result vector is: 1.000000  1.000000        1.000000        1.000000
1.000000        0.666667        0.000000
Maximum profit is: 55.333332
```

Pre Lab Questions:

Question 1: The Knapsack problem is an example of ?

Question 2: . Which of the following methods can be used to solve the Knapsack problem?

Post Lab Questions:

Question 1: You are given a knapsack that can carry a maximum weight of 60.
There are 4 items with weights {20, 30, 40, 70} and values {70, 80, 90, 200}.
What is the maximum value of the items you can carry using the knapsack?

Question 2 : What is the time complexity of the brute force algorithm used to solve the Knapsack problem?

Conclusion: Thus, we have studied and implemented Knapsack

Experiment No . 4

Title: *Write a Program to implement Prim's algorithm*

Date : ____ / ____ / ________

Subject In-charge Sign:

……………………………….

Experiment No. 4

Aim: Write a Program To implement Prim's algorithm

Language Used: C

Prerequisite: GRAPHS

Theory:

Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST. A group of edges that connects two set of vertices in a graph is called cut in graph theory. *So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*

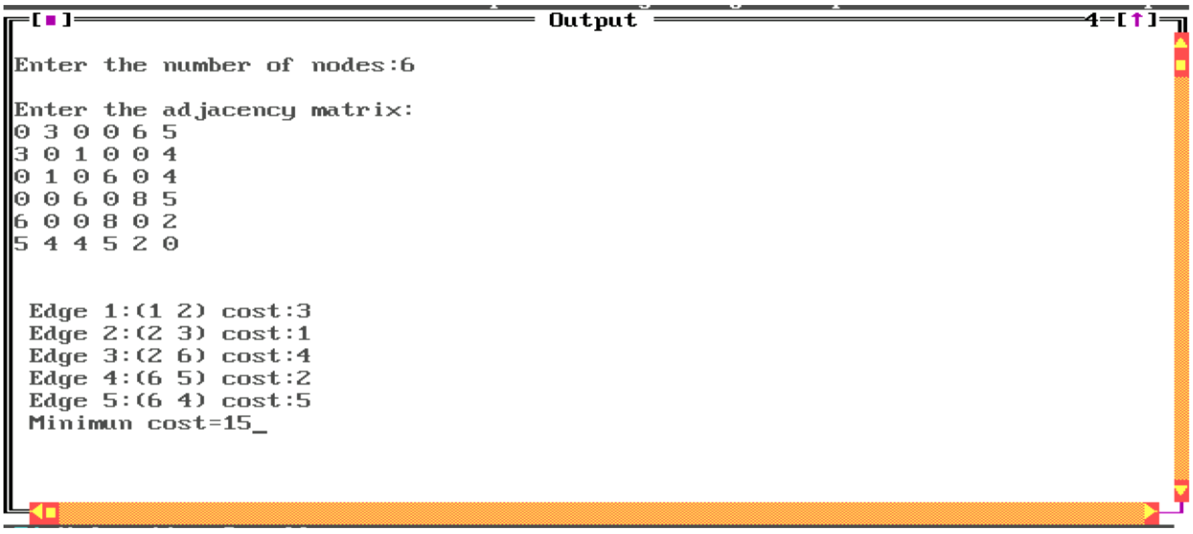
Procedure/Algorithm:

1) Create a set *mstSet* that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3) While mstSet doesn't include all vertices
….**a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
….**b)** Include *u* tomstSet.
….**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as weight of *u*-*v*
The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Program:

```
#include<stdio.h>
int a,b,u,v,n,i,j,ne=1;
 int visited[10]={0},min,mincost=0,cost[10][10];
 void main()
 {

        printf("\nEnter the number of nodes:");
        scanf("%d",&n);
        printf("\nEnter the adjacency matrix:\n");
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
                scanf("%d",&cost[i][j]);
                if(cost[i][j]==0)
                        cost[i][j]=999;
        }
        visited[1]=1;

        printf("\n");

        while(ne < n)

        {

                for(i=1,min=999;i<=n;i++)

                for(j=1;j<=n;j++)

                if(cost[i][j]< min)

                if(visited[i]!=0)

                {

                        min=cost[i][j];

                        a=u=i;

                        b=v=j;

                }

                if(visited[u]==0 || visited[v]==0)

                {

                        printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);

                        mincost+=min;

                        visited[b]=1;

                }

                cost[a][b]=cost[b][a]=999;

        }
        printf("\n Minimun cost=%d",mincost);}
```

Result/Output:

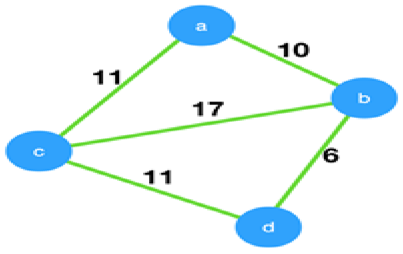

```
Enter the number of nodes:6

Enter the adjacency matrix:
0 3 0 0 6 5
3 0 1 0 0 4
0 1 0 6 0 4
0 0 6 0 8 5
6 0 0 8 0 2
5 4 4 5 2 0


 Edge 1:(1 2) cost:3
 Edge 2:(2 3) cost:1
 Edge 3:(2 6) cost:4
 Edge 4:(6 5) cost:2
 Edge 5:(6 4) cost:5
 Minimun cost=15_
```

Pre Lab Questions:

Question 1: 1. Which of the following is true?

a) Prim's algorithm initialises with a vertex
b) Prim's algorithm initialises with a edge
c) Prim's algorithm initialises with a vertex which has smallest edge
d) Prim's algorithm initialises with a forest

Question 2: Consider the given graph. What is the weight of the minimum spanning tree using the Prim's algorithm,starting from vertex



Post Lab Questions:

Question 1: Worst case is the worst case time complexity of Prim's algorithm if adjacency matrix is used?

Question 2 : Prim's algorithm is a _______

Conclusion:

Thus, we have studied and implemented Prims

Experiment No. 5

Title: *Write a Program to implement n-queens problem*

Date : _____ / _____ / __________

Subject In-charge Sign:

…………………………………..

Experiment No. 5

Aim: Write a Program to implement n- queens problem

Language Used: C
Theory:

The **eight queens puzzle** is the problem of placing eight chessqueens on an 8×8 chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general ***n*-queens problem** of placing *n* queens on an ***n*×*n*** chessboard, where solutions exist for all natural numbers *n* with the exception of *n*=2 and *n*=3.

Chess composerMax Bezzel published the eight queens puzzle in 1848. Franz Nauck published the first solutions in 1850Nauck also extended the puzzle to the *n*-queens problem, with *n* queens on a chessboard of $n \times n$ squares.

Since then, many mathematicians, including Carl Friedrich Gauss, have worked on both the eight queens puzzle and its generalized *n*-queens version.

The problem can be quite computationally expensive as there are 4,426,165,368 (*i.e.*, $_{64}C_8$) possible arrangements of eight queens on an 8×8 board, but only 92 solutions. It is possible to use shortcuts that reduce computational requirements or rules of thumb that avoids brute-force computational techniques. For example, just by applying a simple rule that constrains each queen to a single column (or row), though still considered brute force, it is possible to reduce the number of possibilities to just 16,777,216 (that is, 8^8) possible combinations. Generating permutations further reduces the possibilities to just 40,320 (that is, 8!), which are then checked for diagonal attacks.

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

We can solve nQueen's Problem by:

1) Start in the leftmost column
2) If all queens are placed
return true
3) Try all rows in the current column. Do following for every tried row.
a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
b) If placing queen in [row, column] leads to a solution
then return true.

c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
4) If all rows have been tried and nothing worked, return false to trigger backtracking.

Procedure/Algorithm:

```
NQueens(k,n) {
        // Usingbacktracing, this procedure prints all possible placements of n queens on an
nxn chessboard so that they are non attacking.
        For i = 1 to n do {
                If (Place(k, i)) then
                        x[k] = I;
                        if (k == n)
                                write(x[l : n]);
                        else
                                NQueens(k+l, n);
                endif
        }
}
        }
```

Program:

```
#include<stdio.h>
void nqueen(int r)
{
int i,p,arr[10][10]={0};

if(r%2==0)
  p=1;
else
  p=0;

for(i=0;i<r;i++)
{
 arr[i][p]=1;
 if(r%2==1)
      p=(p+2)%r;
 else
 {
     if(p==r-1)
          p=0;
     else
          p=p+2;
 }
}
for(i=0;i<r;i++)
{
  for(p=0;p<r;p++)
  {
     printf("%d\t",arr[i][p]);
  }
  printf("\n");
 }
 }
void main()
{
int r;
printf("\n enter the value of r:-");
scanf("%d",&r);
nqueen(r);
}
```

Analysis:

The place() function have the complexity **O(n),** as there is only one for loop recurring for n times.
The place() is called **O(n)** time by function Queen()
But Queen is a recursive function and is called **n*T(n-1)** Times, which makes the Time Complexity as

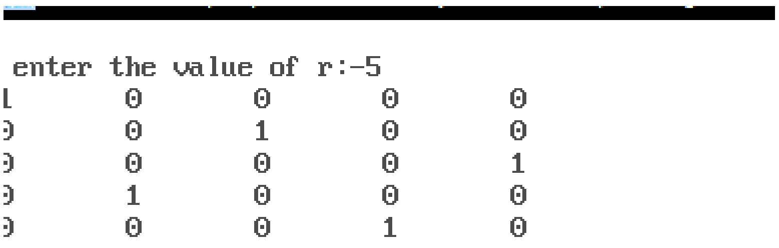
T(n) = **n*T(n-1) + O(n^2)**

T(n) = **O(n!)**

Therefore, Time complexity for N-Queen Problem is **n!**

Selection	Comparison
1. BEST CASE COMPLEXITY:	O(n!)
2. AVERAGE CASE COMPLEXITY:	O(n!)
3. WORST CASE COMPLEXITY:	O(n!)

Result/Output:



```
 enter the value of r:-5
l       0       0       0       0
ɔ       0       1       0       0
ɔ       0       0       0       1
ɔ       1       0       0       0
ɔ       0       0       1       0
```

Pre Lab Questions:

Question 1: In how many directions do queens attack each other?

Question 2: Placing n-queens so that no two queens attack each other is called?

Post Lab Questions:

Question 1: Where is the n-queens problem implemented?

Question 2 : How many possible solutions exist for an 8-queen problem

Conclusion*:*

Thus, we have studied and implemented NQueens

Experiment No.6

Title: *Write a Program to implement longest common Sequence*

Date : ____ / ____ / ________

Subject In-charge Sign:

………………………………..

Experiment No.6

Aim: Write a Program to implement Longest common Sequence

Language Used: C

Prerequisite: STRING

Theory:

The **longest common subsequence** (**LCS**) **problem** is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff utility, and has applications in bioinformatics. It is also widely used by revision control systems such as Git for reconciling multiple changes made to a revision-controlled collection of files.

Example:

Calculating the LCS of a row of the LCS table requires only the solutions to the current row and the previous row. Still, for long sequences, these sequences can get numerous and long, requiring a lot of storage space. Storage space can be saved by saving not the actual subsequences, but the length of the subsequence and the direction of the arrows, as in the table below.

Storing length, rather than sequences						
	Ø	**A**	**G**	**C**	**A**	**T**
Ø	0	0	0	0	0	0
G	0	↑←0	↖1	←1	←1	←1
A	0	↖1	↑←1	↑←1	↖2	←2
C	0	↑1	↑←1	↖2	↑←2	↑←2

The actual subsequences are deduced in a "traceback" procedure that follows the arrows backwards, starting from the last cell in the table. When the length decreases, the sequences must have had a common element. Several paths are possible when two arrows are shown in a cell. Below is the table for such an analysis, with numbers colored in cells where the length is about to decrease.

Traceback example

	Ø	A	G	C	A	T
Ø	0	**0**	0	0	0	0
G	0	↑←0	↖**1**	←**1**	←**1**	←1
A	0	↖1	↑←1	↑←1	↖**2**	←**2**
C	0	↑1	↑←1	↖2	↑←2	↑←**2**

The final result is that the last cell contains all the longest subsequences common to (**AGCAT**) and (**GAC**); these are (**AC**), (**GC**), and (**GA**). The bold numbers

Procedure/Algorithm:

```
        // X = First String
        // Y = Second String
        int c[10][10];
        for i = 0 to m-1 do
          C[i][0] = 0
        for j = 0 to n-1 do
        C[0][j] = 0
        for i = 0 to m-1
                for j = 0 to n-1
                        if X[i] = Y[j]
                                C[i][j] = C[i-1][j-1] + 1
                        else
                                C[i][j] = max(C[i][j-1], C[i-1][j])
        return C[m][n]
}
```

```
Algorithm backtrack(C[0..m][0..n], X[1..m], Y[1..n], i, j) {
      // C = array of LCS Length
      // X = First String
      // Y = Second String
      // i = current index of string 1
      // j = current index of string 2
      if i = 0 or j = 0
            return ""
      else if  X[i] = Y[j]
            print X[i]
            return backtrack(C, X, Y, i-1, j-1)
      else
            if C[i,j-1] > C[i-1,j]
                  return backtrack(C, X, Y, i, j-1)
            else
                  return backtrack(C, X, Y, i-1, j) }
```

Program:

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

```
#include<string.h>
void LCS(char x[20],char[20]);
voiddisp();
int L[20][20], n, m;
int A[20][20];        // Arrow(0=Corner,1=Top,2=Left)
char x[20];
void main()
{
        char y [20];

        printf("Enter String 1: ");
        scanf("%s",x);
        n = strlen(x);
        printf("\nEnter String 2: ");
        scanf("%s",y);
        m = strlen(y);
        LCS(x, y);
        disp();

}
voiddisp_LCS(int, int);

voiddisp()
{
        inti,j;
        for(i = 0;i <= n-1; i++) {
                for(j=0;j <= m-1;j++)
                        printf("%d\t", L[i][j]);
                printf("\n");
        }
        printf("\nLCS is: ");
        disp_LCS(n-1, m-1);
}

voiddisp_LCS(int i, int j)
{
        if (i == -1 || j == -1)
                return;
        if (A[i][j] == 0) {   // Corner
                disp_LCS(i-1, j-1);
                printf("%c", x[i]);
        } else if (A[i][j] == 1) { // Top
                disp_LCS(i-1, j);
        } else {
                disp_LCS(i, j-1);
        }
}

void LCS(char x[20],char y[20])
{
```

```
inti,j;
for (i = 0; i <= n-1; i++)
          for (j = 0; j <= m-1; j++) {
                    L[i][j] = 0;
                    A[i][j] = -1;
          }
for (i = 0; i <= n-1; i++) {
          for (j = 0;j <= m-1;j++) {
                    if (x[i] == y[j]) {
                              A[i][j] = 0;
                              if (i == 0 || j == 0)
                                        L[i][j] = 1;
                              else
                                        L[i][j]=L[i-1][j-1]+1;
                    } else {
                              intcheck_i = 0;
                              intcheck_j = 0;
                              if (i != 0)
                                        check_i = L[i-1][j];
                              if (j != 0)
                                        check_j = L[i][j-1];
                              if(check_i>= check_j) {
                                        if (i == 0)
                                                  L[i][j] = 0;
                                        else
                                                  L[i][j]=L[i-1][j];
                                        A[i][j] = 1;
                              } else {
                                        if (j == 0)
                                                  L[i][j] = 0;
                                        else
                                                  L[i][j]=L[i][j-1];
                                        A[i][j] = 2;
                              }
                    }}}}
```

Analysis:

For the general case of an arbitrary number of input sequences, the problem is NP-hard.[1] When the number of sequences is constant, the problem is solvable in polynomial time by dynamic programming (see *Solution* below). Assume you have N sequences of lengths $n_1, \ldots, n_N$. A naive search would test each of the 2^{n_1}subsequences of the first sequence to determine whether they are also subsequences of the remaining sequences; each subsequence may be tested in time linear in the lengths of the remaining sequences, so the time for this algorithm would be

$$O\left(2^{n_1}\sum_{i>1} n_i\right).$$

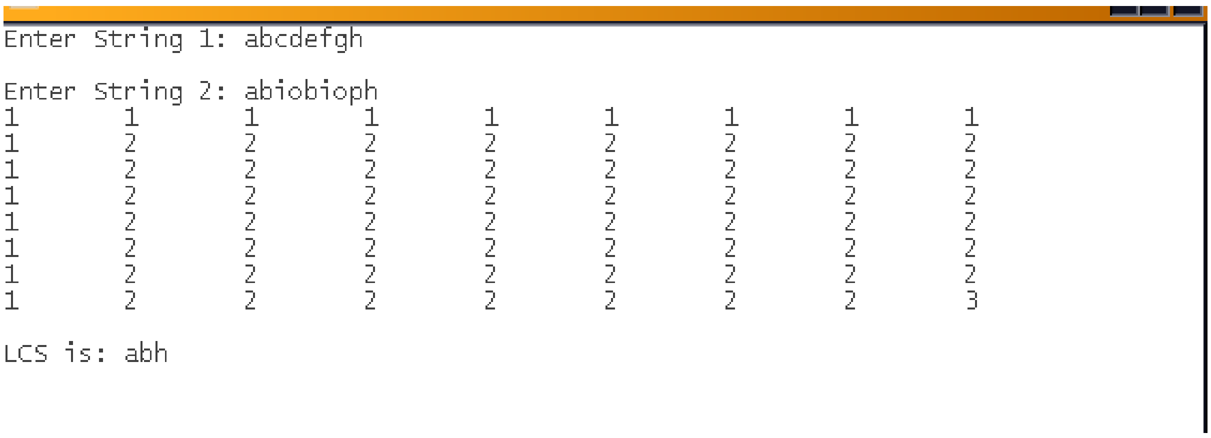
For the case of two sequences of n and m elements, the running time of the dynamic programming approach is O($n \times m$). For an arbitrary number of input sequences, the dynamic programming approach gives a solution in

$$O\left(N\prod_{i=1}^{N} n_i\right).$$

There exist methods with lower complexity, which often depend on the length of the LCS, the size of the alphabet, or both.

Notice that the LCS is not necessarily unique; for example the LCS of "ABC" and "ACB" is both "AB" and "AC". Indeed, the LCS problem is often defined to be finding *all* common subsequences of a maximum length. This problem inherently has higher complexity, as the number of such subsequences is exponential in the worst case, even for only two input strings.

Result/Output:



```
Enter String 1: abcdefgh

Enter String 2: abiobioph
1       1       1       1       1       1       1       1       1
1       2       2       2       2       2       2       2       2
1       2       2       2       2       2       2       2       2
1       2       2       2       2       2       2       2       2
1       2       2       2       2       2       2       2       2
1       2       2       2       2       2       2       2       2
1       2       2       2       2       2       2       2       2
1       2       2       2       2       2       2       2       3

LCS is: abh
```

Pre Lab Questions:

Question 1: Consider the strings “PQRSTPQRS” and “PRATPBRQRPS”. What is the length of the longest common subsequence?

Question 2: Longest common subsequence is an example of

Post Lab Questions:

Question 1: What is the time complexity of the brute force algorithm used to find the longest common subsequence?

Question 2 : What is the time complexity of the above dynamic programming implementation of the longest common subsequence problem where length of one string is “m” and the length of the other string is “n”?

Conclusion: Thus, we have studied and implemented LCS

Experiment No. 7

Title: *Write a program to solve Travelling Salesman Problem*

Date : ____ / ____ / ________

Subject In-charge Sign:

……………………………….

Experiment No. 7

Aim: Write a program to solve Travelling Salesman Problem

Language Used: C

Prerequisite: GRAPHS

Theory:

The **travelling salesman problem (TSP)**asks the following question: *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?* It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

TSP is a special case of the travelling purchaser problem and the vehicle routing problem.

In the theory of computational complexity, the decision version of the TSP (where, given a length *L*, the task is to decide whether the graph has any tour shorter than *L*) belongs to the class of NP-complete problems. Thus, it is possible that the worst-caserunning time for any algorithm for the TSP increases superpolynomially (perhaps, specifically, exponentially) with the number of cities.

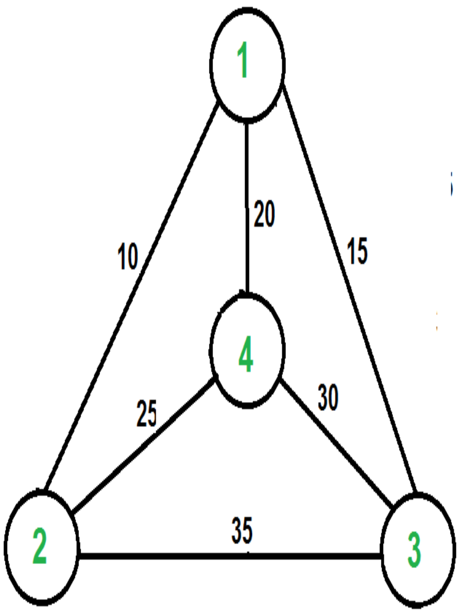
The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.[1]

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimise the time spent slewing the telescope between the sources. In many applications, additional constraints such as limited resources or time windows may be imposed.

TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (*i.e.* each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

Example:

Consider the graph as shown:



Starting city will be **City1**

Step 1: We have to choose minimum path and city that has not been visited yet, which is **2** , as cost of 1-2 is **(Cost= 10)**

Step 2: We now have City 4 and 3 as a choice to visit, city 4 have the minimum cost**(Cost = 25),** thus, we select city 4.

Step 3: We now have City 1 and 3 as choice to visit, we cannot visit city 1, as it is start point and city 3 has not been visited yet, so we choose city 3 **(Cost = 30)**

Step 4: Only city 1 is remaining, and we have to go back to start city. Therefore we choose city 1 **(Cost = 15)**

Route Followed: 1-2-4-3-1

Cost: 10+25+30+15 = 80

Procedure/Algorithm:

```
AlgorithmTSP(k,n){
// k is current city
// n is total number of cities
          for x[k] = 1 to n-1 do {
                    if (path_ok(k,n)) {
                    // path_ok returns 0 if salesman cannot travel to city k
                              used[x[k]] = 1; // City Visited
                              if (k == n-1)  then // If last city, then print the path
                                        sum = 0;
                                        for i = 0 to n-1 do {
                                                  print x[i] // City
                                                  path[c][i] = x[i];
                                                  sum += adj[x[i]][x[i+1]];     // Cost
                                        }
                                        print sum// sum is total cost of city
                                        wght[c++] = sum;  // Save weight of current path
                                        if (c==0 || sum<min) // Saves the minimum cost
                                                  min=sum;
                                        used[x[k]]=0;
                                        getch();
                              else
                                        CALL TSP(k+1,n);
                              used[x[k]]=0;
                    }
          }
```

}

Program:

```
#include<stdio.h>

int x[15], used[15];
intadj[15][15] = { 0 };
int path[15][15], wght[15];
int c, min;

intpath_ok(intk,int n)
{
        if (used[x[k]])
                return 0;
        if (k < n-1)
                return(adj[x[k-1]][x[k]]);
        else
                return(adj[x[k-1]][x[k]] &&adj[x[k]][x[0]]);
}
void TSP(intk,int n)
{
        inti,sum;
        for (x[k] = 1; x[k] < n; x[k]++) {
                if (path_ok(k,n)) {
                        used[x[k]] = 1;
                        if (k == n-1) {
                                sum = 0;
                                printf("\n\n\tPossible Path %d: ",c+1);
                                for (i = 0; i < n; i++) {
                                        printf("%d ", x[i]);
                                        path[c][i] = x[i];
                                        sum += adj[x[i]][x[i+1]];
                                }
                                printf(": Cost=%d",sum);
                                wght[c] = sum;
                                if (c==0 || sum<min)
                                        min=sum;
                                c++;
                                used[x[k]]=0;
                                getch();
                        } else
                                TSP(k+1,n);
                        used[x[k]]=0;
                }
        }
}

voidfindmin(int n)
{
        inti,j;
```

```
		for (i = 0;i < c; i++)
			if (wght[i] == min) {
				printf("\n\n\tMinimum Path: ");
				for (j = 0; j < n; j++)
					printf("%d ", path[i][j]);
			}
}

void main()
{
		inti,n,j;
		intedg;

		printf("\n\n\t\tTravelling Salesman Problem\n\n");
		printf("\nEnter the no. of Cities: ");
		scanf("%d",&n);
		printf("\n\n Enter the Cost if path Exist Between cities.:{c1,c2}.Else Enter 0\n\n");
		printf("\n\Cities\t\Cost\n\n");
		for (i = 0; i < n; i++)
		for (j = i+1; j < n; j++) {
			printf("\n\t%d to %d: ", i, j);
			scanf("%d", &edg);
			if (edg> 0)
				adj[i][j] = adj[j][i] = edg;
		}
		used[0] = 1;
		TSP(1, n);
		if (c == 0)
			printf("\n\n\t\tNo Path Found to Cover all the Cities\n\n");
		else {
			printf("\n\n\t\tMinimum Cost is %d and the Paths are \n\n", min);
			findmin(n);
		}

}
```

Analysis:

Let **N** be the number of **g (i, S)**'s that have to be computed before **i**can be used to compute **g (1, V – {1}).** For each value of **|S|** there are **n-1** choices for **i**. The number of distinct sets **S** of size **k** not include **1** and **i**$\binom{n-2}{k}$

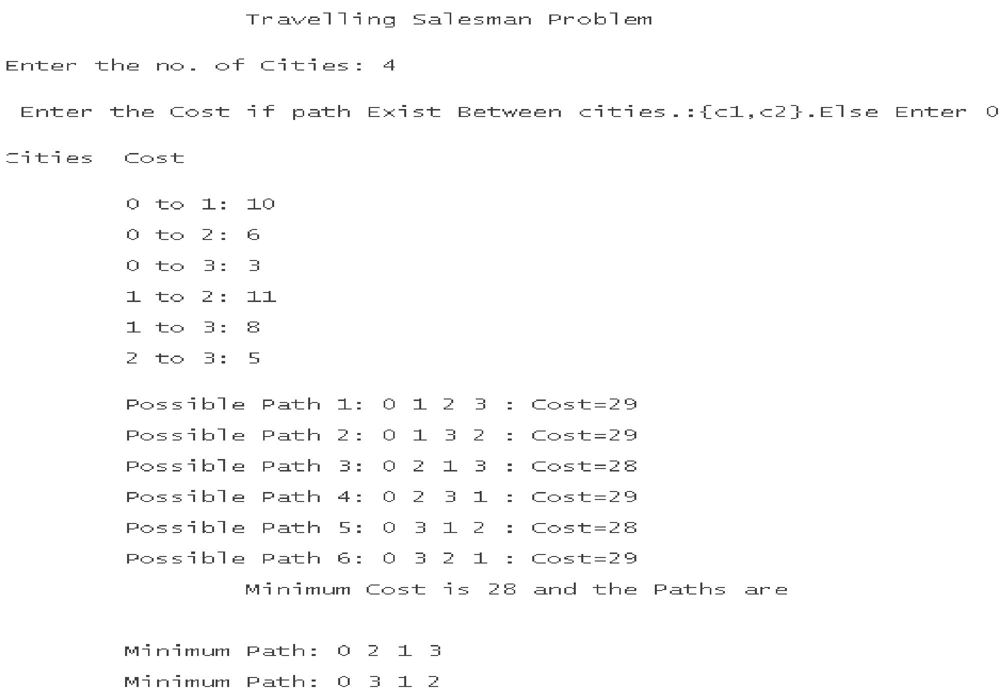
Hence, **N** = $\sum_{k=0}^{n-2}(n-1)\binom{n-2}{k}$ = **(n-1) 2^{n-2}**

This Algorithm will require **O($n^2 2^n$)** time complexity.

The most serious drawback of this dynamic programming solution is space needed **O($n2^n$)**

Selection	Comparison
1. BEST CASE COMPLEXITY:	$O(n^2 2^n)$
2. AVERAGE CASE COMPLEXITY:	$O(n^2 2^n)$
3. WORST CASE COMPLEXITY:	$O(n^2 2^n)$

Result/Output:



```
                 Travelling Salesman Problem

Enter the no. of Cities: 4

 Enter the Cost if path Exist Between cities.:{c1,c2}.Else Enter 0

Cities   Cost

        0 to 1: 10
        0 to 2: 6
        0 to 3: 3
        1 to 2: 11
        1 to 3: 8
        2 to 3: 5

        Possible Path 1: 0 1 2 3 : Cost=29
        Possible Path 2: 0 1 3 2 : Cost=29
        Possible Path 3: 0 2 1 3 : Cost=28
        Possible Path 4: 0 2 3 1 : Cost=29
        Possible Path 5: 0 3 1 2 : Cost=28
        Possible Path 6: 0 3 2 1 : Cost=29
                Minimum Cost is 28 and the Paths are

        Minimum Path: 0 2 1 3
        Minimum Path: 0 3 1 2
```

Pre Lab Questions:

Question 1:Explain Graph Theory ?
Question 2: What is vertex and Edges?

Post Lab Questions:

Question 1: Explain the algorithm of Tsp?
Question 2 :What is the type of TSP?

Conclusion:

Thus, we have studied and implemented Travelling Salesman Problem.

Experiment No. 8

Title: *Write a program to solve Job Sequencing Problem*

Date : _____ / _____ / __________

Subject In-charge Sign:

……………………………….

Experiment No. 8

Aim : Write a program to solve Job Sequencing Problem

Language Used: C

Theory :

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Examples:
Input: Four Jobs with following
deadlines and profits

JobID	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Output: Following is maximum
profit sequence of jobs
c, a

Input: Five Jobs with following
deadlines and profits

JobID	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output: Following is maximum
profit sequence of jobs
c, a, e

Steps for performing job sequencing with deadline using greedy approach is as follows:

1. Sort all the jobs based on the profit in an increasing order.
2. Let α be the maximum deadline that will define the size of array.
3. Create a solution array S with d slots.
4. Initialize the content of array S with zero.
5. Check for all jobs.
   a. If scheduling is possible a lot i^{th} slot of array s to job i.
   b. Otherwise look for location (i-1), (i-2)...1.
   c. Schedule the job if possible else reject.
6. Return array S as the answer.
7. End.

Algorithm for job sequencing

```
Begin
    Sort all the jobs based on profit Pi so
    P1 > P2 > P3 …………………………….>=Pn
    d = maximum deadline of job in A
    Create array S[1,…………………,d]
    For i=1 to n do
    Find the largest job x
    For j=I to 1
    If ((S[j] = 0) and (x deadline<= d1))
    Then
    S[x] = I;
    Break;
    End if
    End for
    End for
    End
```

Time Complexity of the above solution is $O(n^2)$.

Program :

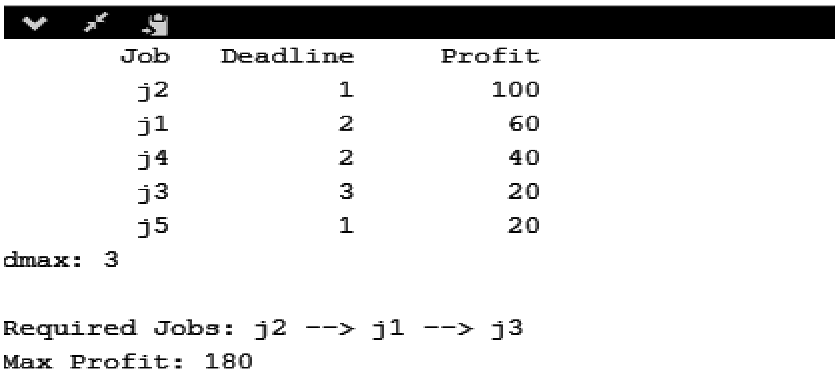
```
#include <stdio.h>

#define MAX 100

typedef struct Job {
  char id[5];
  int deadline;
  int profit;
} Job;

void jobSequencingWithDeadline(Job jobs[], int n);

int minValue(int x, int y) {
  if(x < y) return x;
  return y;
```

```
}

int main(void) {
  //variables
  int i, j;

  //jobs with deadline and profit
  Job jobs[5] = {
    {"j1", 2,  60},
    {"j2", 1, 100},
    {"j3", 3,  20},
    {"j4", 2,  40},
    {"j5", 1,  20},
  };

  //temp
  Job temp;

  //number of jobs
  int n = 5;

  //sort the jobs profit wise in descending order
  for(i = 1; i < n; i++) {
    for(j = 0; j < n - i; j++) {
      if(jobs[j+1].profit > jobs[j].profit) {
        temp = jobs[j+1];
        jobs[j+1] = jobs[j];
        jobs[j] = temp;
      }
    }
  }

  printf("%10s %10s %10s\n", "Job", "Deadline", "Profit");
  for(i = 0; i < n; i++) {
    printf("%10s %10i %10i\n", jobs[i].id, jobs[i].deadline, jobs[i].profit);
  }

  jobSequencingWithDeadline(jobs, n);

  return 0;
}

void jobSequencingWithDeadline(Job jobs[], int n) {
  //variables
  int i, j, k, maxprofit;

  //free time slots
  int timeslot[MAX];

  //filled time slots
```

```
int filledTimeSlot = 0;

//find max deadline value
int dmax = 0;
for(i = 0; i < n; i++) {
  if(jobs[i].deadline > dmax) {
    dmax = jobs[i].deadline;
  }
}

//free time slots initially set to -1 [-1 denotes EMPTY]
for(i = 1; i <= dmax; i++) {
  timeslot[i] = -1;
}

printf("dmax: %d\n", dmax);

for(i = 1; i <= n; i++) {
  k = minValue(dmax, jobs[i - 1].deadline);
  while(k >= 1) {
    if(timeslot[k] == -1) {
      timeslot[k] = i-1;
      filledTimeSlot++;
      break;
    }
    k--;
  }

  //if all time slots are filled then stop
  if(filledTimeSlot == dmax) {
    break;
  }
}

//required jobs
printf("\nRequired Jobs: ");
for(i = 1; i <= dmax; i++) {
  printf("%s", jobs[timeslot[i]].id);

  if(i < dmax) {
    printf(" --> ");
  }
}

//required profit
maxprofit = 0;
for(i = 1; i <= dmax; i++) {
  maxprofit += jobs[timeslot[i]].profit;
}
printf("\nMax Profit: %d\n", maxprofit);
```

}

Output :



```
        Job    Deadline      Profit
         j2           1         100
         j1           2          60
         j4           2          40
         j3           3          20
         j5           1          20
dmax: 3

Required Jobs: j2 --> j1 --> j3
Max Profit: 180
```

Pre Lab Questions:

Question 1: Explain what are processors?
Question 2: Write a code to arrange jobs in descending order?

Post Lab Questions:

Question 1: Explain the algorithm for Job Sequencing?
Question 2 :What is the Complexity of Job Sequencing?

Conclusion:

Thus, we have studied and implemented Job Sequencing.

Experiment No. 9

Title: *Write a program to implement naïve string matching*

Date : ____ / ____ / ________

Subject In-charge Sign:

……………………………..

Experiment No. 9

Aim : Write a program to implement naïve string matching

Language Used: C

Theory :

The Naive String Matching Algorithm is one of the simplest methods to check whether a string follows a particular pattern or not.

It is simple of all the algorithm but is highly inefficient. It checks where the string matches the input pattern one by one with every character of the string.

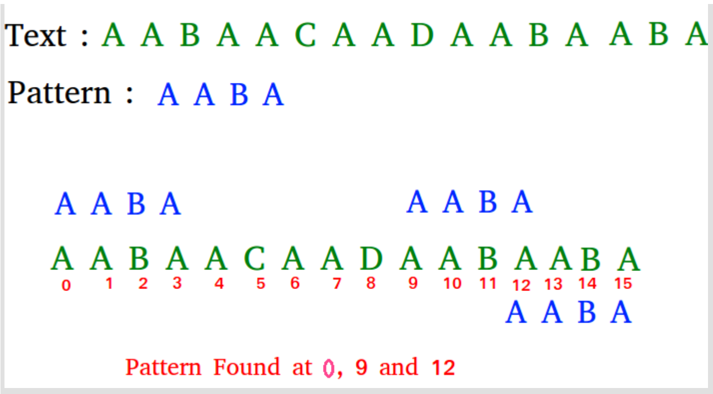
The complexity of the Naive String Search Algorithm for average case scenario is **O(n +m)** whereas for worst case scenario is **O(nm)**.
The following Naive String Search program takes a string from the user and then a pattern that user wants to find in the string.

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that *n > m*.

Examples:

Input: txt[] = "THIS IS A TEST TEXT"
pat[] = "TEST"
Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA"
pat[] = "AABA"
Output: Pattern found at index 0
Pattern found at index 9
Pattern found at index 12


Text : A A B A A C A A D A A B A A B A
Pattern : A A B A
A A B A
A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A A B A
Pattern Found at 0, 9 and 12

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

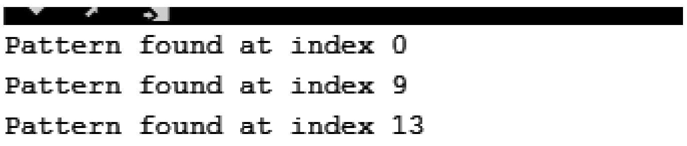
Naive Pattern Searching:

Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

Program :

```c
 #include <stdio.h>
#include <string.h>

void search(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++) {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;

        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            printf("Pattern found at index %d \n", i);
    }
}

/* Driver program to test above function */
int main()
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";
    search(pat, txt);
    return 0;
}
```

Output:



```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Complexity :

What is the best case?

The best case occurs when the first character of the pattern is not present in text at all.

txt[] = "AABCCAADDEE";

pat[] = "FAA";

The number of comparisons in best case is O(n).

What is the worst case ?
The worst case of Naive Pattern Searching occurs in following scenarios.
1) When all characters of the text and pattern are same.
filter_none

brightness_4
txt[] = "AAAAAAAAAAAAAAAAAA";

pat[] = "AAAAA";

2) Worst case also occurs when only the last character is different.

filter_none

brightness_4
txt[] = "AAAAAAAAAAAAAAAAAB";

pat[] = "AAAAB";

The number of comparisons in the worst case is O(m*(n-m+1)). Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts). The KMP matching algorithm improves the worst case to O(n). We will be covering KMP in the next post. Also, we will be writing more posts to cover all pattern searching algorithms and data structures.

Pre Lab Questions:

Question 1: Which package is used for string?
Question 2: How do you define a string?

Post Lab Questions:

Question 1: Explain the algorithm for naïve string matching?
Question 2 :What is the Complexity of naïve string matching?

Conclusion:

Thus, we have studied and implement naïve string matching

Experiment No. 9

Title: *Write a program to implement Rabin Karp*

Date : _____ / _____ / __________

Subject In-charge Sign:

…………………………………..

Experiment No. 9

Aim : Write a program to implement naïve string matching

Language Used: C

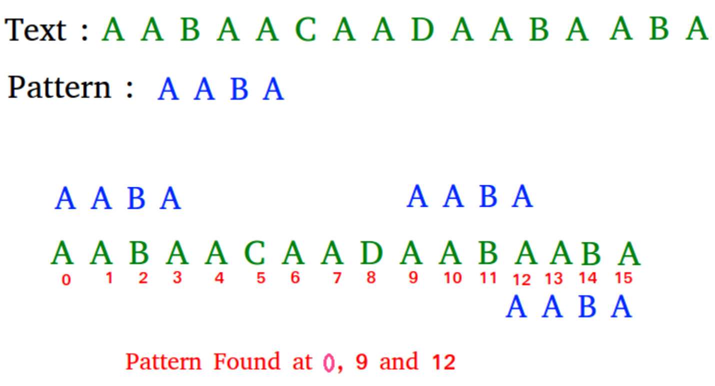
Theory :

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that n > m.

Examples:

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
Output: Pattern found at index 10

Input:  txt[] =  "AABAACAADAABAABA"
        pat[] =  "AABA"
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```


Text : A A B A A C A A D A A B A A B A
Pattern : A A B A
A A B A
A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A A B A
Pattern Found at 0, 9 and 12

The Naive String Matching algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match. Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.
2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property.
Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1]))* and rehash must be O(1) operation.
The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

*hash(txt[s+1 .. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q*
hash(txt[s .. s+m-1]) : Hash value at shift *s*.
hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift *s*+1)
d: Number of characters in the alphabet
q: A prime number
h: d^(m-1)

RABIN-KARP-MATCHER (T, P, d, q)Algorithm

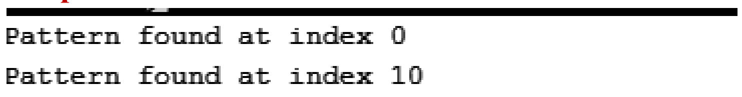
```
n ← length [T]
m ← length [P]
h ← dm-1 mod q
p ← 0
t0 ← 0
for i ← 1 to m
do p ← (dp + P[i]) mod q
t0 ← (dt0+T [i]) mod q
for s ← 0 to n-m
do if p = ts
then if P [1.....m] = T [s+1.....s + m]
then "Pattern occurs with shift" s
If s < n-m
then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q
```

Program :

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

/* pat -> pattern
```

```
    txt -> text
    q -> A prime number
*/
void search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M-1; i++)
        h = (h*d)%q;

    // Calculate the hash value of pattern and first
    // window of text
    for (i = 0; i < M; i++)
    {
        p = (d*p + pat[i])%q;
        t = (d*t + txt[i])%q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++)
    {

        // Check the hash values of current window of text
        // and pattern. If the hash values match then only
        // check for characters on by one
        if ( p == t )
        {
            /* Check for characters one by one */
            for (j = 0; j < M; j++)
            {
                if (txt[i+j] != pat[j])
                    break;
            }

            // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if (j == M)
                printf("Pattern found at index %d \n", i);
        }

        // Calculate hash value for next window of text: Remove
        // leading digit, add trailing digit
        if ( i < N-M )
        {
            t = (d*(t - txt[i]*h) + txt[i+M])%q;
            // We might get negative value of t, converting it
            // to positive
            if (t < 0)
            t = (t + q);
        }
    }
}

/* Driver program to test above function */
int main()
{
    char txt[] = "GEEKS FOR GEEKS";
    char pat[] = "GEEK";
    int q = 101; // A prime number
    search(pat, txt, q);
    return 0;
}
```

Output :



```
Pattern found at index 0
Pattern found at index 10
```

Complexity:

The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

Pre Lab Questions:

Question 1: Which package is used for string?
Question 2: How do you define a string?

Post Lab Questions:

Question 1: Explain the algorithm for Rabin Karp?
Question 2 :What is the Complexity of Rabin Karp?

Conclusion:

Thus, we have studied and implement Rabin Karp

Experiment No. 10

Title: *Write a program to implement to Sort Elements in Lexicographical Order (Dictionary Order)*

Date : ____ /____ /________

Subject In-charge Sign:

……………………………..

Experiment No. 10

Aim: Write a program to implement to Sort Elements in Lexicographical Order (Dictionary Order)

Language Used: C

Theory :

A string is a data type used in programming, such as an integer and floating point unit, but is used to represent text rather than numbers. It is comprised of a set of characters that can also contain spaces and numbers. For example, the word "hamburger" and the phrase "I ate 3 hamburgers" are both strings. Even "12345" could be considered a string, if specified correctly. Typically, programmers must enclose strings in quotation marks for the data to recognized as a string and not a number or variable name.

if (Option1 == Option2) then ...

Option1 and Option2 may be variables containing integers, strings, or other data. If the values are the same, the test returns a value of true, otherwise the result is false. In the comparison:

if ("Option1" == "Option2") then ...

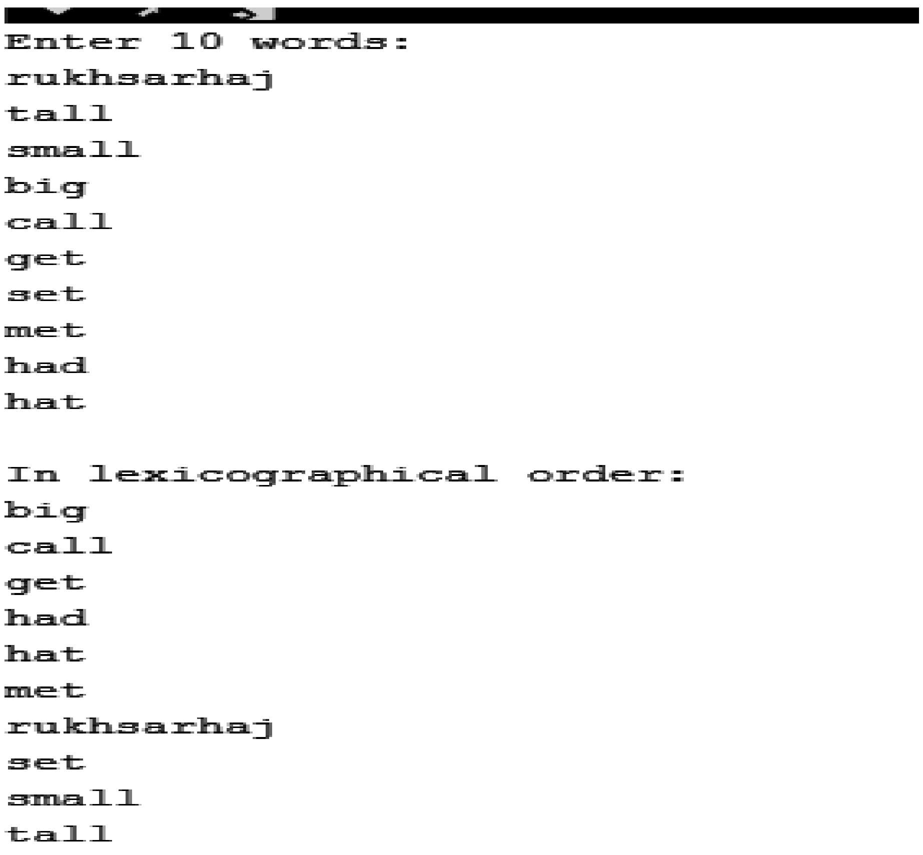
Option1 and Option2 are being treated as strings. Therefore the test is comparing the words "Option1" and "Option2," which would return false. The length of a string is often determined by using a null character.

Program:

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

int main()
{
   int i, j;
   char str[10][50], temp[50];

   printf("Enter 10 words:\n");

   for(i=0; i<10; ++i)
      scanf("%s[^\n]",str[i]);


   for(i=0; i<9; ++i)
      for(j=i+1; j<10 ; ++j)
      {
         if(strcmp(str[i], str[j])>0)
         {
            strcpy(temp, str[i]);
```

```
                strcpy(str[i], str[j]);
                strcpy(str[j], temp);
            }
        }

    printf("\nIn lexicographical order: \n");
    for(i=0; i<10; ++i)
    {
        puts(str[i]);
    }

    return 0;
}
```

Output:



```
Enter 10 words:
rukhsarhaj
tall
small
big
call
get
set
met
had
hat

In lexicographical order:
big
call
get
had
hat
met
rukhsarhaj
set
small
tall
```

Pre Lab Questions:

Question 1: Which package is used for string?
Question 2: How do you define a string?

Post Lab Questions:

Question 1: Explain the algorithm to sort elements in lexicographical order?
Question 2 :What is the complexity to sort elements lexicographical order ?

Conclusion:

Thus, we have Sorted Elements in Lexicographical Order (Dictionary Order)

Experiment No. 11

Title: *Write a program to perform Bellman Ford's Algorithm*

Date : ____ / ____ / ________

Subject In-charge Sign:

……………………………….

Experiment No. 11

Aim: Write a program to perform Bellman Ford's Algorithm

Language Used: C

Theory:

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

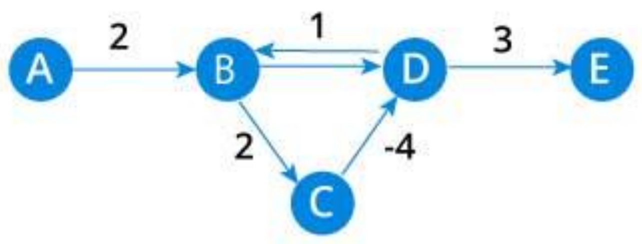
Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, heat released/absorbed in a chemical reaction etc.

For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.

If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

WHY WE NEED TO BE CAREFUL WITH NEGATIVE WEIGHTS?

Negative weight edges can create negative weight cycles i.e. a cycle which will reduce the total path distance by coming back to the same point.



Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

How Bellman Ford's algorithm works

Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

By doing this repeatedly for all vertices, we are able to guarantee that the end result is optimized.

Bellman Ford Pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

```
function bellmanFord(G, S)
     for each vertex V in G
                    distance[V] <- infinite
                    previous[V] <- NULL
     distance[S] <- 0
     for each vertex V in G
          for each edge (U,V) in G
                    tempDistance <- distance[U] + edge_weight(U, V)
                    if tempDistance < distance[V]
                       distance[V] <- tempDistance
                       previous[V] <- U

     for each edge (U,V) in G
          If distance[U] + edge_weight(U, V) < distance[V}
                    Error: Negative Cycle Exists

     return distance[], previous[]
```

Program :

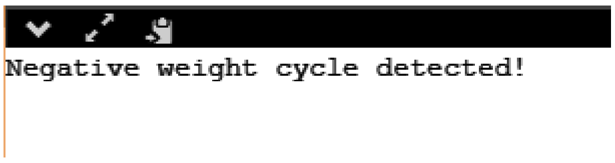
```
#include <stdio.h>
#include <stdlib.h>
#define INFINITY 99999
//struct for the edges of the graph
struct Edge {
        int u;     //start vertex of the edge
        int v;     //end vertex of the edge
        int w;     //weight of the edge (u,v)
```

```
};
//Graph - it consists of edges
struct Graph {
        int V;      //total number of vertices in the graph
        int E;      //total number of edges in the graph
        struct Edge *edge; //array of edges
};
void bellmanford(struct Graph *g, int source);
void display(int arr[], int size);
int main(void) {
        //create graph
        struct Graph *g = (struct Graph*)malloc(sizeof(struct Graph));
        g->V = 4;           //total vertices
        g->E = 5;           //total edges

        //array of edges for graph
        g->edge = (struct Edge*)malloc(g->E * sizeof(struct Edge));

        //------- adding the edges of the graph
        /*
                edge(u, v)
                where   u = start vertex of the edge (u,v)
                                v = end vertex of the edge (u,v)

                w is the weight of the edge (u,v)
        */

        //edge 0 --> 1
        g->edge[0].u = 0;
        g->edge[0].v = 1;
        g->edge[0].w = 5;

        //edge 0 --> 2
        g->edge[1].u = 0;
        g->edge[1].v = 2;
        g->edge[1].w = 4;
        //edge 1 --> 3
        g->edge[2].u = 1;
        g->edge[2].v = 3;
        g->edge[2].w = 3;
        //edge 2 --> 1
        g->edge[3].u = 2;
        g->edge[3].v = 1;
        g->edge[3].w = -6;
        //edge 3 --> 2
        g->edge[4].u = 3;
        g->edge[4].v = 2;
        g->edge[4].w = 2;

        bellmanford(g, 0);          //0 is the source vertex
        return 0;
}
void bellmanford(struct Graph *g, int source) {
        //variables
        int i, j, u, v, w;
        //total vertex in the graph g
        int tV = g->V;

        //total edge in the graph g
        int tE = g->E;

        //distance array
        //size equal to the number of vertices of the graph g
        int d[tV];

        //predecessor array
        //size equal to the number of vertices of the graph g
        int p[tV];

        //step 1: fill the distance array and predecessor array
        for (i = 0; i < tV; i++) {
                d[i] = INFINITY;
                p[i] = 0;
        }

        //mark the source vertex
        d[source] = 0;

        //step 2: relax edges |V| - 1 times
        for(i = 1; i <= tV-1; i++) {
                for(j = 0; j < tE; j++) {
                        //get the edge data
                        u = g->edge[j].u;
                        v = g->edge[j].v;
                        w = g->edge[j].w;

                        if(d[u] != INFINITY && d[v] > d[u] + w) {
                                d[v] = d[u] + w;
                                p[v] = u;
                        }
                }
        }

        //step 3: detect negative cycle
        //if value changes then we have a negative cycle in the graph
        //and we cannot find the shortest distances
        for(i = 0; i < tE; i++) {
                u = g->edge[i].u;
                v = g->edge[i].v;
```

```
                        w = g->edge[i].w;
                        if(d[u] != INFINITY && d[v] > d[u] + w) {
                                printf("Negative weight cycle detected!\n");
                                return;
                        }
                }

                //No negative weight cycle found!
                //print the distance and predecessor array
                printf("Distance array: ");
                display(d, tV);
                printf("Predecessor array: ");
                display(p, tV);
}
void display(int arr[], int size) {
                int i;
                for(i = 0; i < size; i ++) {
                        printf("%d ", arr[i]);
                }
                printf("\n");
}
```

Time Complexity: O(VE)

Output:



```
Negative weight cycle detected!
```

Pre Lab Questions:

Question 1: What are edges?
Question 2: How do you idemtify negative weights?

Post Lab Questions:

Question 1: Explain the how will find the best path?
Question 2: What is the complexity of bellman ford ?

Conclusion:

Thus, we hav performed Bellman Ford's Algorithm