NAME: **Meet Raut**

DIV: **S21**

ROLL NO: **2201084**

EXPERIMENT – 1

- **AIM:** To study and implement Selection sort & Insertion sort algorithms.
- ***THEORY:***

1. Selection Sort Algorithm: In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is **$O(n^2)$**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

Now, let's see the algorithm of selection sort.

Algorithm

```
SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
Step 4: EXIT

SMALLEST (arr, i, n, pos)
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
    SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
Step 4: RETURN pos
```

Working of Selection sort Algorithm:

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

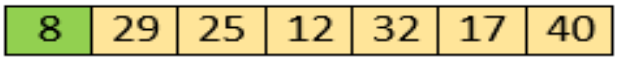
Let the elements of array are –

12	29	25	8	32	17	40

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

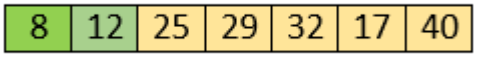
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

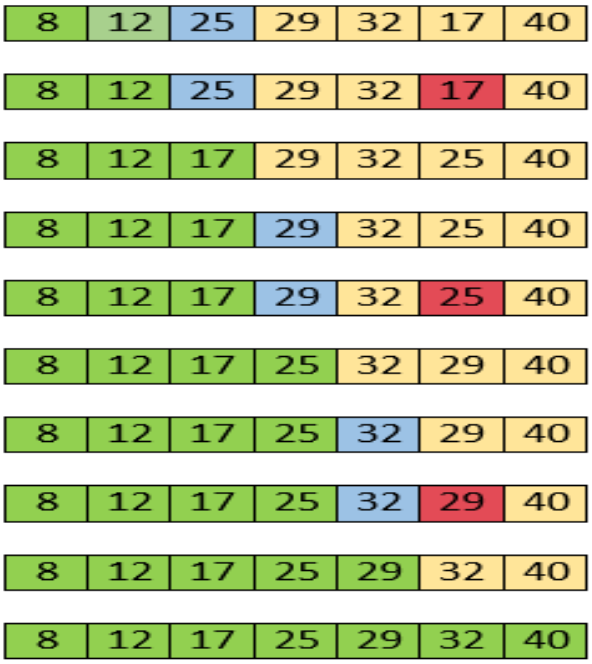


For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

Selection sort complexity:

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

i. Time Complexity:

	Case Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **$O(n^2)$**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **$O(n^2)$**.

ii. Space Complexity:

Space Complexity	O(1)
Stable	YES

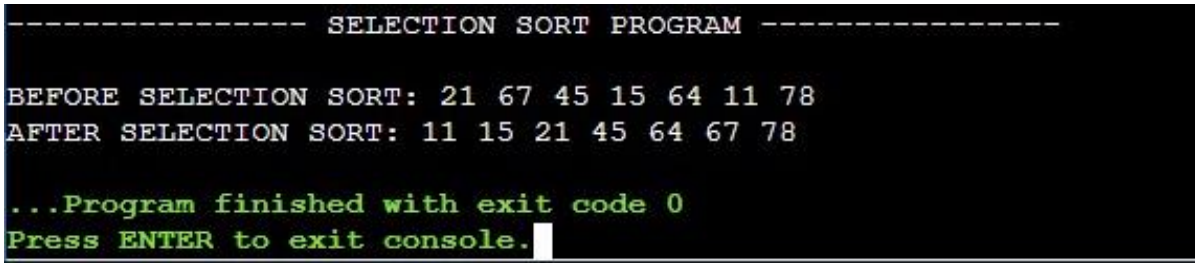
- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

- **C PROGRAM (SELECTION SORT) :**

```
//SELECTION SORT
#include <stdio.h>
int main()
{
   int arr[] = {21,67,45,15,64,11,78};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("---------------- SELECTION SORT PROGRAM ----------------\n");
   printf("\nBEFORE SELECTION SORT: ");
   printArr(arr,n);
   selSort(arr,n);
   printf("\nAFTER SELECTION SORT: ");
   printArr(arr,n);
}

void swap (int *x, int *y) {
   int temp = *x;
   *x = *y;
   *y = temp;
}
void selSort(int arr[], int n)
```

```
{
   int i,j,min;

   for (i=0;i<n-1;i++)
   {
      min = i;

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

      if (arr[j] < arr[min])
      min = j;

      if(min != i)
          swap (&arr[min], &arr[i]);

    }

}

void printArr(int arr[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
}
```

- **OUTPUT:**



2. Insertion Sort Algorithm: Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **$O(n^2)$**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as –

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

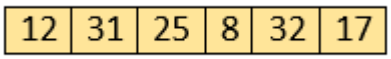
Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm:

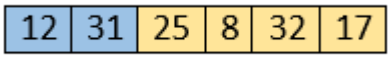
Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

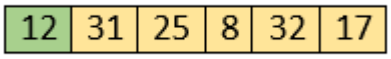
Let the elements of array are -



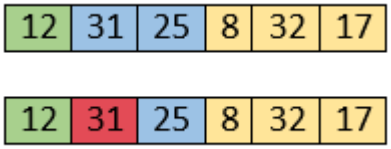
Initially, the first two elements are compared in insertion sort.



Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

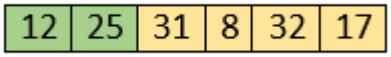


Now, move to the next two elements and compare them.

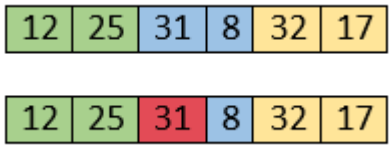


Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

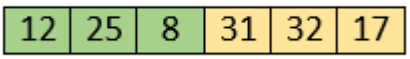
For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.



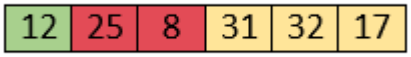
Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



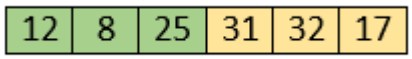
Both 31 and 8 are not sorted. So, swap them.



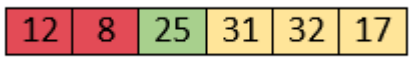
After swapping, elements 25 and 8 are unsorted.



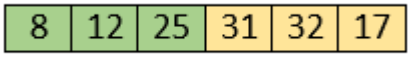
So, swap them.



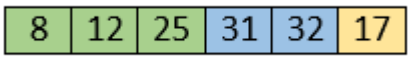
Now, elements 12 and 8 are unsorted.



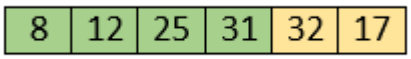
So, swap them too.



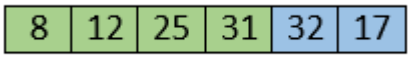
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



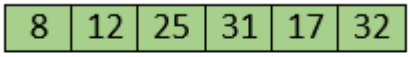
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



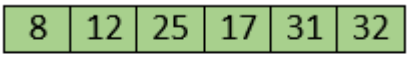
Move to the next elements that are 32 and 17.



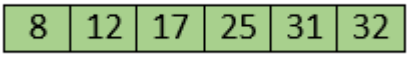
17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

Insertion sort complexity:

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

i. Time Complexity:

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

ii. Space Complexity:

Space Complexity	O(1)
Stable	YES

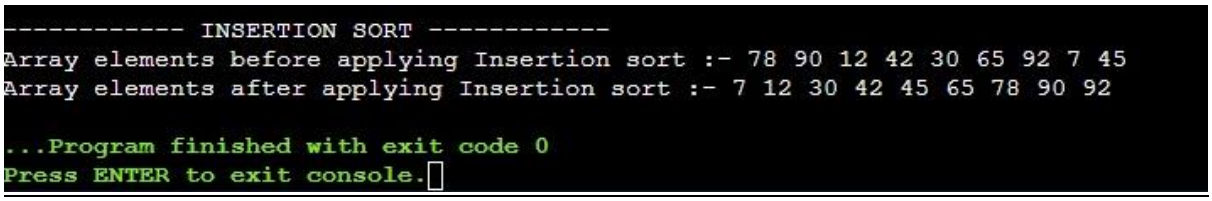
- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

- **C PROGRAM (INSERTION SORT) :**

```
//INSERTION SORT

#include <stdio.h>

int main(){

   int arr[] = { 78, 90, 12, 42, 30, 65, 92, 7, 45 };
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("------------ INSERTION SORT ------------");
   printf("\nArray elements before applying Insertion sort :- ");
   printArr(arr, n);

   insSort(arr, n);

   printf("\nArray elements after applying Insertion sort :- ");
   printArr(arr, n);

   return 0;
}

void printArr(int arr[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
}

void insSort(int arr[], int n)
{
      int i, key, j;
   for (i = 1; i < n; i++)
   {
      key = arr[i];
      j = i - 1;

      while (j >= 0 && arr[j] > key)
      {
```

```
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

- **OUTPUT:**



- ***CONCLUSIONS:*** We have successfully implemented Selection sort & Insertion sort algorithms.

NAME: Meet Raut

DIV: S21

ROLL.NO: 2201084

Experiment 2:

- **Aim: To study and implement Quick Sort & Merge Sort algorithms.**

- **Theory:**

1. **Quick Sort Algorithm:** Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

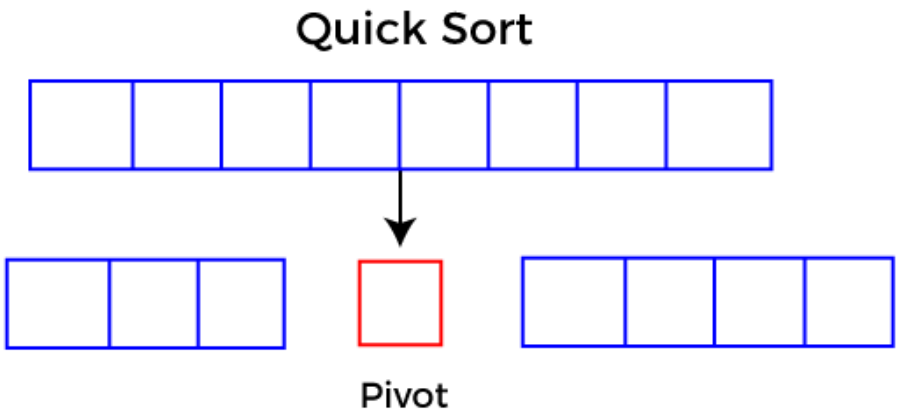
   Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

   Conquer: Recursively, sort two subarrays with Quicksort.

   Combine: Combine the already sorted array.

   Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element.

   In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot. After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



- **Choosing the pivot:**

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows –

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm:

```
QUICKSORT (array A, start, end)
{
 1 if (start < end)
 2 {
3 p = partition(A, start, end)
4 QUICKSORT (A, start, p - 1)
5 QUICKSORT (A, p + 1, end)
6 }
}
```

The partition algorithm rearranges the sub-arrays in a place.

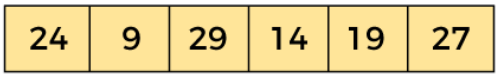
```
PARTITION (array A, start, end)
{
 1 pivot ? A[end]
 2 i ? start-1
 3 for j ? start to end -1 {
 4 do if (A[j] < pivot) {
 5 then i ? i + 1
 6 swap A[i] with A[j]
 7 }}
 8 swap A[i+1] with A[end]
 9 return i+1
}
```

➢ Working of Quick Sort Algorithm:

Now, let's see the working of the Quicksort Algorithm.

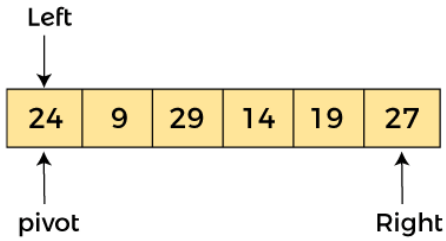
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear.

Let the elements of array are -

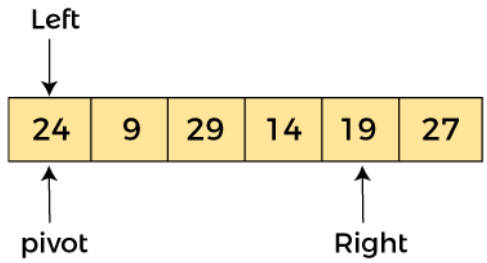


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

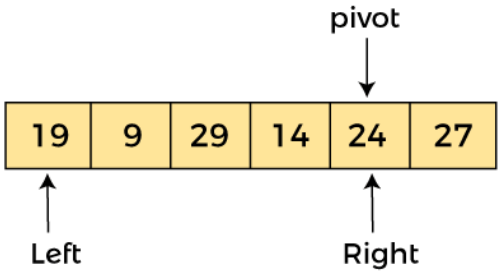


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. –



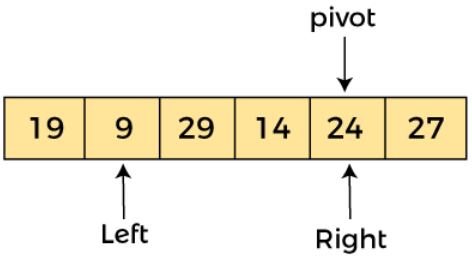
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

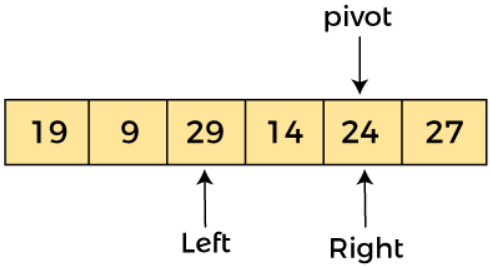


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

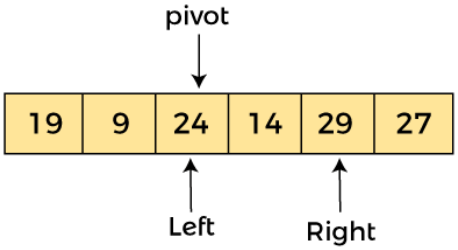
As a[pivot] > a[left], so algorithm moves one position to right as -



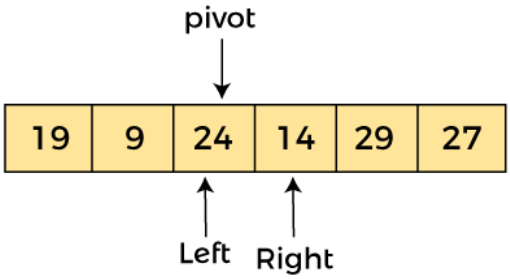
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



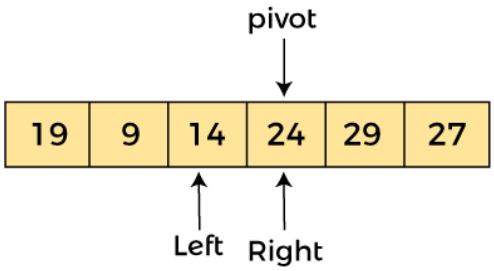
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



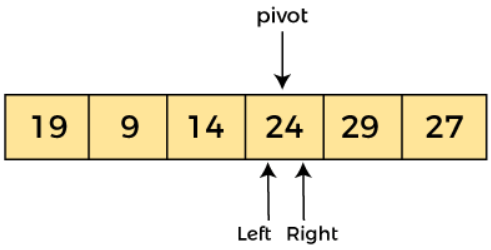
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



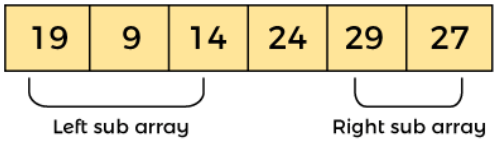
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



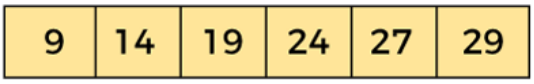
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



➢ Quicksort complexity:

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

- **Best Case Complexity** - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is O(n*logn).
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is O(n*logn).
- **Worst Case Complexity** - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is O(n2).

Though the worst-case complexity of quicksort is more than other sorting algorithms such as Merge sort and Heap sort, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

2. Space Complexity

Space Complexity	O(n*logn)
Stable	NO

- The space complexity of quicksort is O(n*logn).

➢ C PROGRAM (Quick Sort):

```
#include <stdio.h>

int partition (int a[], int low, int high)
{
   int pivot = a[high]; // pivot element
   int i = (low - 1);

   for (int j = low; j <= high - 1; j++)
   {
      // If current element is smaller than the pivot
      if (a[j] < pivot)
      {
         i++; // increment index of smaller element
         int t = a[i];
         a[i] = a[j];
```

```
            a[j] = t;
        }
    }

    int t = a[i+1];
    a[i+1] = a[high];
    a[high] = t;
    return (i + 1);
}

/* function to implement quick sort */
void quick(int a[], int low, int high)
{
    if (low < high)
    {
        int p = partition(a, low, high); //p is the partitioning index
        quick(a, low, p - 1);
        quick(a, p + 1, high);
    }
}

/* function to print an array */
void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}


int main()
{
    int a[] = { 24, 9, 19, 34, 15, 27, 1, 56 };
    int n = sizeof(a) / sizeof(a[0]);

    printf("\nBefore sorting array elements are - \n");
    printArr(a, n);
```

```
    quick(a, 0, n - 1);
    printf("\n\nAfter sorting array elements are - \n");
    printArr(a, n);

    return 0;
}
```

OUTPUT:

```
Before sorting array elements are -
24 9 19 34 15 27 1 56

After sorting array elements are -
1 9 15 19 24 27 34 56

...Program finished with exit code 0
Press ENTER to exit console.
```

---

2. **Merge Sort Algorithm:** Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging. The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Now, let's see the algorithm of merge sort.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
```

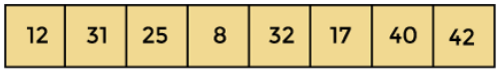
The important part of the merge sort is the MERGE function. This function performs the merging of two sorted sub-arrays that are A[beg…mid] and A[mid+1…end], to build one sorted array A[beg…end]. So, the inputs of the MERGE function are A[], beg, mid, and end.

➢ Working of Merge sort Algorithm:

Now, let's see the working of merge sort Algorithm.

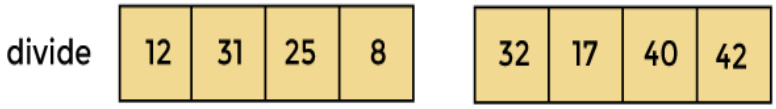
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

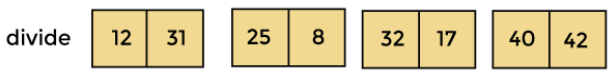


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

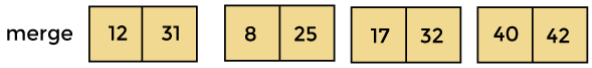


Now, again divide these arrays to get the atomic value that cannot be further divided.

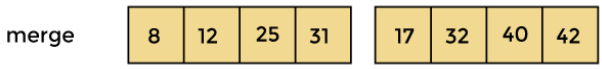
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

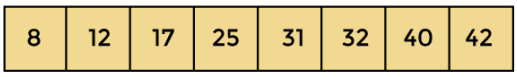
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

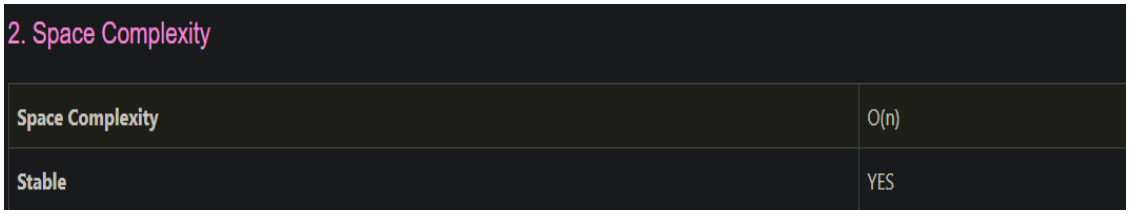
➢ MergeSort complexity:

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is O(n*logn).
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is O(n*logn).
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is O(n*logn).



The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

- **C PROGRAM (Merge Sort):**

```
#include <stdio.h>


void merge(int a[], int beg, int mid, int end)
{
   int i, j, k;
   int n1 = mid - beg + 1;
   int n2 = end - mid;

   int LeftArray[n1], RightArray[n2]; //temporary arrays

   /* copy data to temp arrays */
   for (int i = 0; i < n1; i++)
   LeftArray[i] = a[beg + i];
   for (int j = 0; j < n2; j++)
   RightArray[j] = a[mid + 1 + j];

   i = 0;               /* initial index of first sub-array */
   j = 0;               /* initial index of second sub-array */
   k = beg;             /* initial index of merged sub-array */

   while (i < n1 && j < n2)
   {
      if(LeftArray[i] <= RightArray[j])
      {
         a[k] = LeftArray[i];
         i++;
```

```
        }

        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }

    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}
```

```
/* Function to print the array */
void printArray(int a[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", a[i]);
   printf("\n");
}


int main ()
{
   int a[] = { 12, 31, 15, 8, 32, 27, 4, 42, 1, 45, 60 };
   int n = sizeof(a) / sizeof(a[0]);
   printf("\nBefore sorting array elements are - \n");
   printArray(a, n);
   mergeSort(a, 0, n - 1);
   printf("\n\nAfter sorting array elements are - \n");
   printArray(a, n);
   return 0;
}
```

OUTPUT:

```
Before sorting array elements are -
12 31 15 8 32 27 4 42 1 45 60


After sorting array elements are -
1 4 8 12 15 27 31 32 42 45 60


...Program finished with exit code 0
Press ENTER to exit console.
```

Basis for comparison	Quick Sort	Merge Sort
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	In the merge sort, the array is parted into just 2 halves (i.e. n/2).
Worst case complexity	O(n^2)	O(nlogn)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor
Major work	The major work is to partition the array into two sub-arrays before sorting them recursively.	Major work is to combine the two sub-arrays after sorting them recursively.
Division of array	Division of an array into sub-arrays may or may not be balanced as the array is partitioned around the pivot.	Division of an array into sub array is always balanced as it divides the array exactly at the middle.
Method	Quick sort is in- place sorting method.	Merge sort is not in – place sorting method.

➢ Quick Sort vs Merge Sort :

1. **Partition of elements in the array:** In the merge sort, the array is parted into just 2 halves (i.e. n/2). whereas in case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.

2. **Worst case complexity:** The worst case complexity of quick sort is O(n^2) as there is need of lot of comparisons in the worst condition. whereas In merge sort, worst case and average case has same complexities O(n log n).

3**. Usage with datasets:** Merge sort can work well on any type of data sets irrespective of its size (either large or small). whereas the quick sort cannot work well with large datasets.

4. **Additional storage space requirement:** Merge sort is not in place because it requires additional memory space to store the auxiliary arrays. whereas the quick sort is in place as it doesn't require any additional storage.

5. **Efficiency:** Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets. whereas Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.

6. **Sorting method:** The quick sort is internal sorting method where the data is sorted in main memory. whereas the merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.

7. **Stability:** Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array. whereas Quick sort is unstable in this scenario. But it can be made stable using some changes in code.

8. **Preferred for:** Quick sort is preferred for arrays. whereas Merge sort is preferred for linked lists.

9. **Locality of reference:** Quicksort exhibits good cache locality and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).

- **<u>CONCLUSION</u>: Hence, we have successfully implemented Quick Sort & Merge Sort algorithm; LO1, LO2.**

Name: Meet Raut

Batch: S21

Roll number: 2201084

EX. 3:- IMPLEMENTATION OF BINARY SEARCH ALGORITHM

LO Mapped:- LO1, LO2

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Conditions for when to apply Binary Search in a Data Structure:

To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

Binary Search Algorithm:

In this algorithm,

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
- If the key is smaller than the middle element, then the left side is used for next search.
- If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

Consider an array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target = 23**.

First Step: *Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.*

- *Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.*

Binary Search Algorithm : Compare key with 16

- *Key is less than the current mid 56. The search space moves to the left.*

Binary Search Algorithm : Compare key with 56

Second Step: *If the key matches the value of the mid element, the element is found and stop search.*

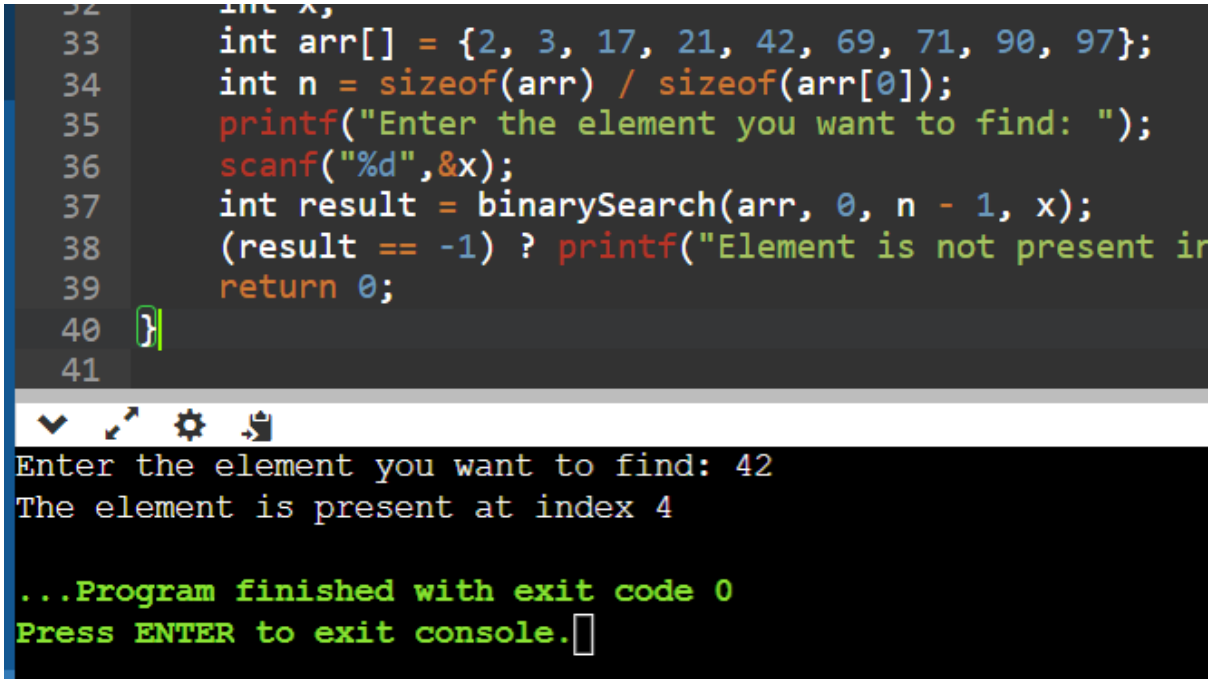
Binary Search Algorithm : Key matches with mid

CODE:

//Binary Search

```
#include<stdio.h>
int binarySearch(int arr[], int l, int r, int x)
{
   if(r >= l)
   {
      int mid = l + (r - l) / 2;

      if (arr[mid] == x)
         return mid;

      if(arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);

      return binarySearch(arr, mid + 1, r, x);
   }
   return -1;
}
int main()
{
   int x;
   int arr[] = {2, 3, 17, 21, 42, 69, 71, 90, 97};
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Enter the element you want to find: ");
   scanf("%d",&x);
   int result = binarySearch(arr, 0, n - 1, x);
   (result == -1) ? printf("Element is not present in the array") : printf("The element is present at index
%d", result);
   return 0;
}
```

OUTPUT:



CONCLUSION:

We can conclude that we have successfully implemented Binary Search algorithm.

NAME: Meet Raut

DIV: S21

ROLL.NO: 2201084

Experiment 4:

- **AIM: To study and implement Single source shortest path using Dijkstra's Algorithm.**

- **THEORY:**

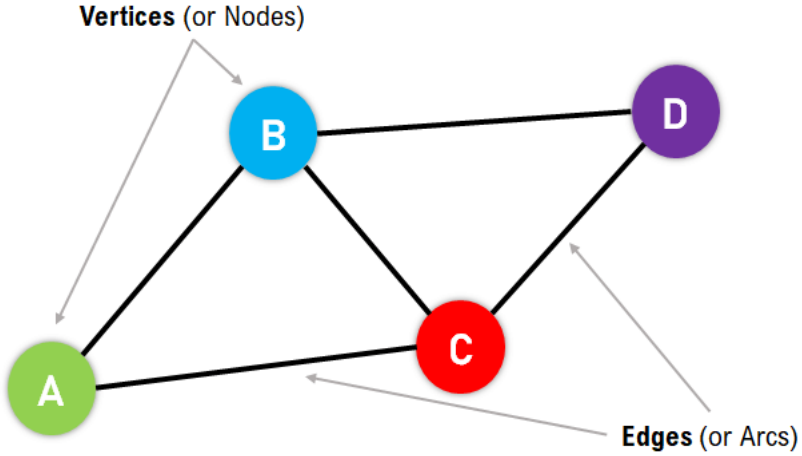
A Brief Introduction to Graphs:

Graphs are non-linear data structures representing the "connections" between the elements. These elements are known as the **Vertices**, and the lines or arcs that connect any two vertices in the graph are known as the **Edges**. More formally, a Graph comprises **a set of Vertices (V)** and **a set of Edges (E)**. The Graph is denoted by **G(V, E)**.

Components of a Graph:

1. **Vertices:**Vertices are the basic units of the graph used to represent real-life objects, persons, or entities. Sometimes, vertices are also known as Nodes.
2. **Edges:**Edges are drawn or used to connect two vertices of the graph. Sometimes, edges are also known as Arcs.

The following figure shows a graphical representation of a Graph:

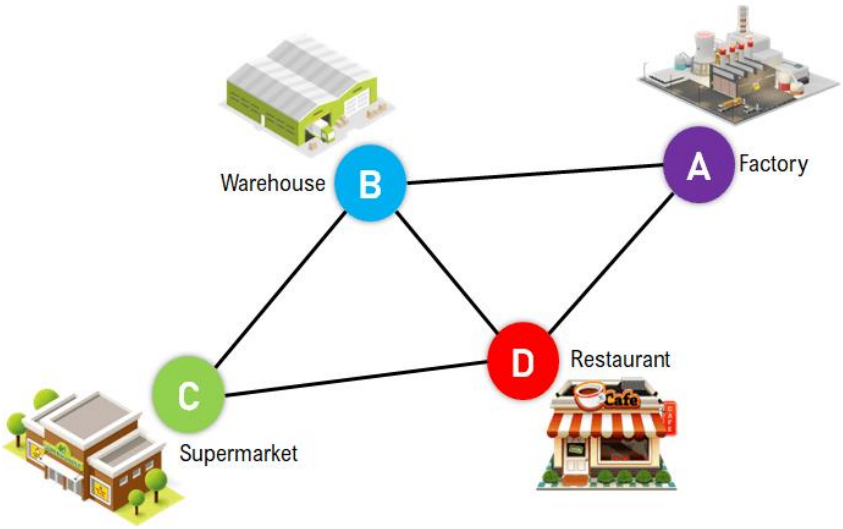


In the above figure, the Vertices/Nodes are denoted with Colored Circles, and the Edges are denoted with the lines connecting the nodes.

Applications of the Graphs:

Graphs are used to solve many real-life problems. Graphs are utilized to represent the networks. These networks may include telephone or circuit networks or paths in a city.

For example, we could use Graphs to design a transportation network model where the vertices display the facilities that send or receive the products, and the edges represent roads or paths connecting them. The following is a pictorial representation of the same:



Graphs are also utilized in different Social Media Platforms like LinkedIn, Facebook, Twitter, and more. For example, Platforms like Facebook use Graphs to store the data of their users where every person is indicated with a vertex, and each of them is a structure containing information like Person ID, Name, Gender, Address, etc.

Types of Graphs:

The Graphs can be categorized into two types:

1. Undirected Graph
2. Directed Graph

Undirected Graph: A Graph with edges that do not have a direction is termed an Undirected Graph. The edges of this graph imply a two-way relationship in which each edge can be traversed in both directions. The following figure displays a simple undirected graph with four nodes and five edges.

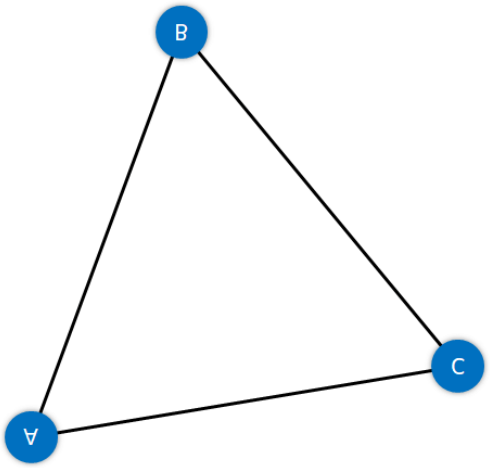


Figure 3: A Simple Undirected Graph

Directed Graph: A Graph with edges with direction is termed a Directed Graph. The edges of this graph imply a one-way relationship in which each edge can only be traversed in a single direction. The following figure displays a simple directed graph with four nodes and five edges.

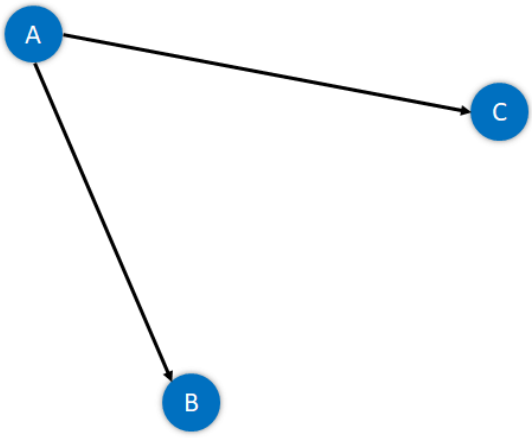


Figure 4: A Simple Directed Graph

The absolute length, position, or orientation of the edges in a graph illustration characteristically does not have meaning. In other words, we can visualize the same graph in different ways by rearranging the vertices or distorting the edges if the underlying structure of the graph does not alter.

What are Weighted Graphs?

A Graph is said to be Weighted if each edge is assigned a 'weight'. The weight of an edge can denote distance, time, or anything that models the 'connection' between the pair of vertices it connects.

For instance, we can observe a blue number next to each edge in the following figure of the Weighted Graph. This number is utilized to signify the weight of the corresponding edge.

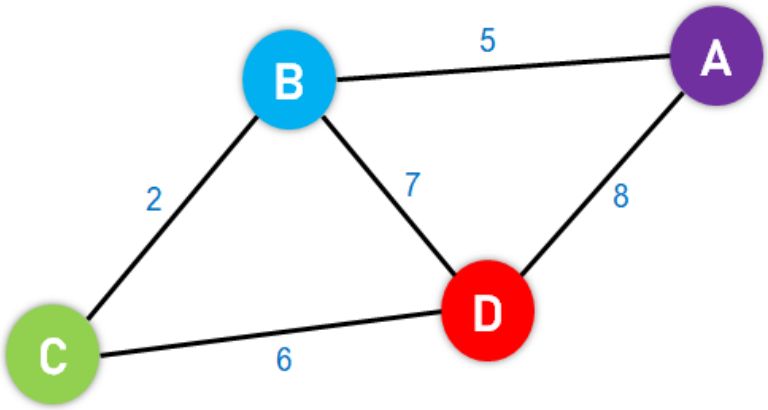


Figure 5: An Example of a Weighted Graph

➢ Dijkstra's Algorithm:

Dijkstra's Algorithm is a Graph algorithm **that finds the shortest path** from a source vertex to all other vertices in the Graph (single source shortest path). It is a type of Greedy Algorithm that only works on Weighted Graphs having positive weights. The time complexity of Dijkstra's Algorithm is $\mathbf{O(V^2)}$ with the help of the adjacency matrix representation of the graph. This time complexity can be reduced to **O((V + E) log V)** with the help of an adjacency list representation of the graph, where **V** is the number of vertices and **E** is the number of edges in the graph.

History of Dijkstra's Algorithm:

Dijkstra's Algorithm was designed and published by **Dr. Edsger W. Dijkstra**, a Dutch Computer Scientist, Software Engineer, Programmer, Science Essayist, and Systems Scientist.

During an Interview with Philip L. Frana for the Communications of the ACM journal in the year 2001, Dr. Edsger W. Dijkstra revealed:

"What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city? It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame."

Dijkstra thought about the shortest path problem while working as a programmer at the Mathematical Centre in Amsterdam in 1956 to illustrate the capabilities of a new computer known as ARMAC. His goal was to select both a problem and a solution (produced by the computer) that people with no computer background could comprehend. He developed the shortest path algorithm and later executed it for ARMAC for a vaguely shortened transportation map of 64 cities in the Netherlands (64 cities, so 6 bits would be sufficient to encode the city

number). A year later, he came across another issue from hardware engineers operating the next computer of the institute: Minimize the amount of wire required to connect the pins on the machine's back panel. As a solution, he re-discovered the algorithm called Prim's minimal spanning tree algorithm and published it in the year 1959.

Fundamentals of Dijkstra's Algorithm:

The following are the basic concepts of Dijkstra's Algorithm:

1. Dijkstra's Algorithm begins at the node we select (the source node), and it examines the graph to find the shortest path between that node and all the other nodes in the graph.
2. The Algorithm keeps records of the presently acknowledged shortest distance from each node to the source node, and it updates these values if it finds any shorter path.
3. Once the Algorithm has retrieved the shortest path between the source and another node, that node is marked as 'visited' and included in the path.
4. The procedure continues until all the nodes in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

Understanding the Working of Dijkstra's Algorithm:

A **graph** and **source vertex** are requirements for Dijkstra's Algorithm. This Algorithm is established on Greedy Approach and thus finds the locally optimal choice (local minima in this case) at each step of the Algorithm.

Each Vertex in this Algorithm will have two properties defined for it:

1. Visited Property
2. Path Property

Let us understand these properties in brief.

Visited Property:

1. The 'visited' property signifies whether or not the node has been visited.
2. We are using this property so that we do not revisit any node.
3. A node is marked visited only when the shortest path has been found.

Path Property:

1. The 'path' property stores the value of the current minimum path to the node.
2. The current minimum path implies the shortest way we have reached this node till now.
3. This property is revised when any neighbor of the node is visited.

4. This property is significant because it will store the final answer for each node.

Initially, we mark all the vertices, or nodes, unvisited as they have yet to be visited. The path to all the nodes is also set to infinity apart from the source node. Moreover, the path to the source node is set to zero (0).

We then select the source node and mark it as visited. After that, we access all the neighboring nodes of the source node and perform relaxation on every node. Relaxation is the process of lowering the cost of reaching a node with the help of another node.

In the process of relaxation, the path of each node is revised to the minimum value amongst the node's current path, the sum of the path to the previous node, and the path from the previous node to the current node.

Let us suppose that p[n] is the value of the current path for node n, p[m] is the value of the path up to the previously visited node m, and w is the weight of the edge between the current node and previously visited one (edge weight between n and m).

In the mathematical sense, relaxation can be exemplified as:

p[n] = minimum(p[n], p[m] + w)

We then mark an unvisited node with the least path as visited in every subsequent step and update its neighbor's paths.

We repeat this procedure until all the nodes in the graph are marked visited.

Whenever we add a node to the visited set, the path to all its neighboring nodes also changes accordingly.

If any node is left unreachable (disconnected component), its path remains 'infinity'. In case the source itself is a separate component, then the path to all other nodes remains 'infinity'.

Understanding Dijkstra's Algorithm with an Example

The following is the step that we will follow to implement Dijkstra's Algorithm:

Step 1: First, we will mark the source node with a current distance of 0 and set the rest of the nodes to INFINITY.

Step 2: We will then set the unvisited node with the smallest current distance as the current node, suppose X.

Step 3: For each neighbor N of the current node X: We will then add the current distance of X with the weight of the edge joining X-N. If it is smaller than the current distance of N, set it as the new current distance of N.

Step 4: We will then mark the current node X as visited.

Step 5: We will repeat the process from **'Step 2'** if there is any node unvisited left in the graph.

Let us now understand the implementation of the algorithm with the help of an example:

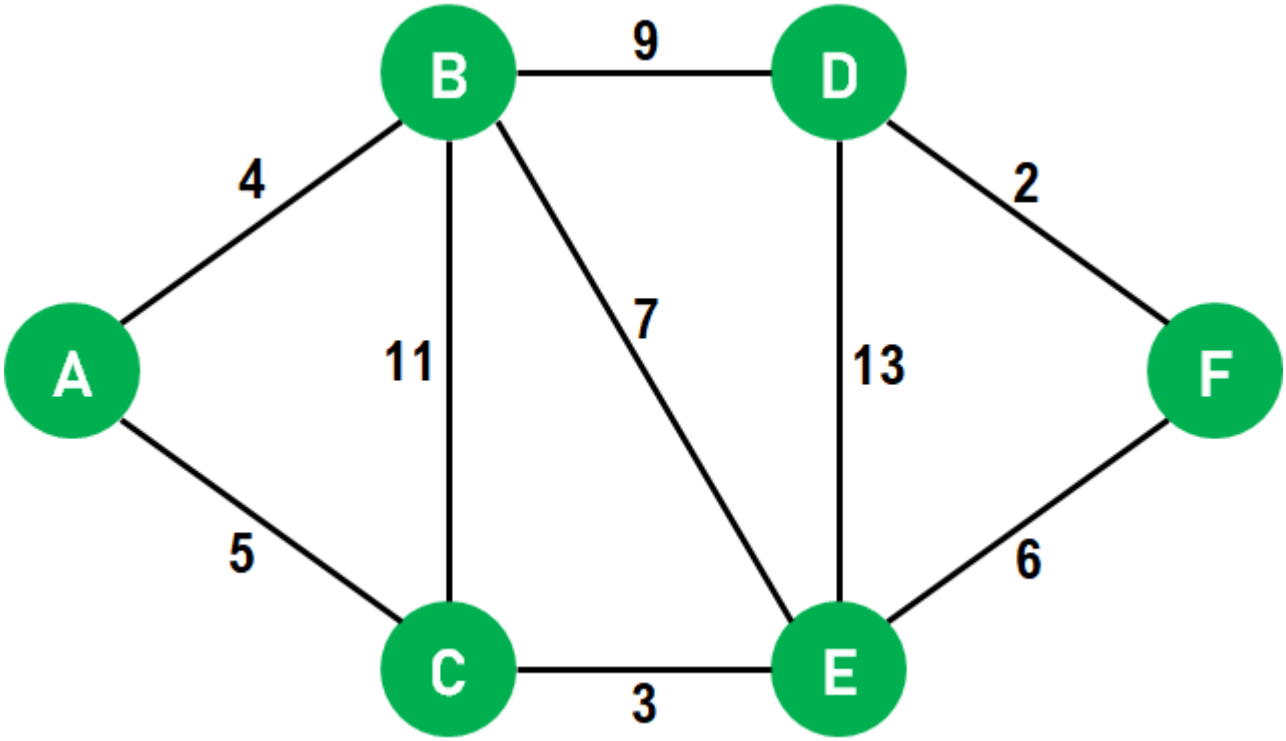


Figure 6: The Given Graph

1. We will use the above graph as the input, with node **A** as the source.
2. First, we will mark all the nodes as unvisited.
3. We will set the path to **0** at node **A** and **INFINITY** for all the other nodes.
4. We will now mark source node **A** as visited and access its neighboring nodes. **Note:** We have only accessed the neighboring nodes, not visited them.
5. We will now update the path to node **B** by **4** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **B** is **4**, and the **minimum((0 + 4), INFINITY)** is **4**.
6. We will also update the path to node **C** by **5** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **C** is **5**, and the **minimum((0 + 5), INFINITY)** is **5**. Both the neighbors of node **A** are now relaxed; therefore, we can move ahead.
7. We will now select the next unvisited node with the least path and visit it. Hence, we will visit node **B** and perform relaxation on its unvisited neighbors. After performing

relaxation, the path to node **C** will remain **5**, whereas the path to node **E** will become **11**, and the path to node **D** will become **13**.

8. We will now visit node **E** and perform relaxation on its neighboring nodes **B, D**, and **F**. Since only node **F** is unvisited, it will be relaxed. Thus, the path to node **B** will remain as it is, i.e., **4**, the path to node **D** will also remain **13**, and the path to node **F** will become **14 (8 + 6)**.
9. Now we will visit node **D**, and only node **F** will be relaxed. However, the path to node **F** will remain unchanged, i.e., **14**.
10. Since only node **F** is remaining, we will visit it but not perform any relaxation as all its neighboring nodes are already visited.
11. Once all the nodes of the graphs are visited, the program will end.

Hence, the final paths we concluded are:

```
A = 0
B = 4 (A -> B)
C = 5 (A -> C)
D = 4 + 9 = 13 (A -> B -> D)
E = 5 + 3 = 8 (A -> C -> E)
F = 5 + 3 + 6 = 14 (A -> C -> E -> F)
```

Pseudocode for Dijkstra's Algorithm:

We will now understand a pseudocode for Dijkstra's Algorithm.

- We have to maintain a record of the path distance of every node. Therefore, we can store the path distance of each node in an array of size n, where n is the total number of nodes.
- Moreover, we want to retrieve the shortest path along with the length of that path. To overcome this problem, we will map each node to the node that last updated its path length.
- Once the algorithm is complete, we can backtrack the destination node to the source node to retrieve the path.
- We can use a minimum Priority Queue to retrieve the node with the least path distance in an efficient way.

Let us now implement a pseudocode of the above illustration:

Pseudocode:

```
function Dijkstra_Algorithm(Graph, source_node)
   // iterating through the nodes in Graph and set their distances to INFINITY
   for each node N in Graph:
      distance[N] = INFINITY
      previous[N] = NULL
      If N != source_node, add N to Priority Queue G
   // setting the distance of the source node of the Graph to 0
   distance[source_node] = 0

   // iterating until the Priority Queue G is not empty
   while G is NOT empty:
      // selecting a node Q having the least distance and marking it as visited
      Q = node in G with the least distance[]
      mark Q visited

      // iterating through the unvisited neighboring nodes of the node Q and perform
ing relaxation accordingly
      for each unvisited neighbor node N of Q:
         temporary_distance = distance[Q] + distance_between(Q, N)

         // if the temporary distance is less than the given distance of the path to the
Node, updating the resultant distance with the minimum value
         if temporary_distance < distance[N]
            distance[N] := temporary_distance
            previous[N] := Q

   // returning the final list of distance
   return distance[], previous[]
```

Explanation:

In the above pseudocode, we have defined a function that accepts multiple parameters - the Graph consisting of the nodes and the source node. Inside this function, we have iterated through each node in the Graph, set their initial distance to **INFINITY**, and set the previous

node value to **NULL**. We have also checked whether any selected node is not a source node and added the same into the Priority Queue. Moreover, we have set the distance of the source node to **0**. We then iterated through the nodes in the priority queue, selected the node with the least distance, and marked it as visited. We then iterated through the unvisited neighboring nodes of the selected node and performed relaxation accordingly. At last, we have compared both the distances (original and temporary distance) between the source node and the destination node, updated the resultant distance with the minimum value and previous node information, and returned the final list of distances with their previous node information.

Dijkstra's Algorithm using Adjacency Matrix:

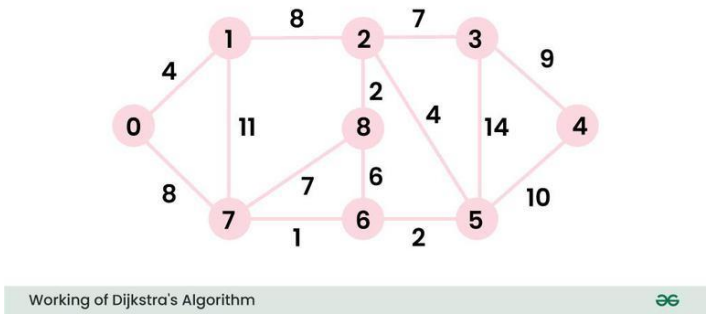
The idea is to generate a SPT (shortest path tree) with a given source as a root. Maintain an Adjacency Matrix with two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Algorithm:

- Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.
- While sptSet doesn't include all vertices
  - Pick a vertex u that is not there in sptSet and has a minimum distance value.
  - Include u to sptSet.
  - Then update the distance value of all adjacent vertices of u.
  - To update the distance values, iterate through all adjacent vertices.
  - For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Note: We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.

Examples:
Input: *src = 0, the graph is shown below.*

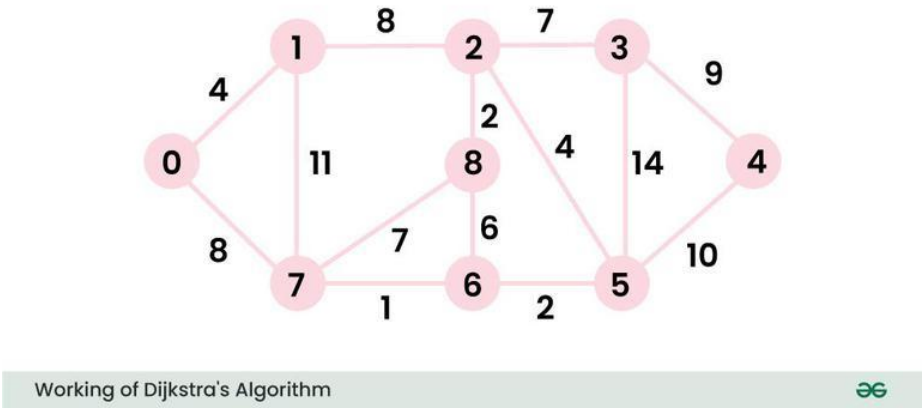


Working of Dijkstra's Algorithm

Output: 0 4 12 19 21 11 9 8 14
Explanation: The distance from 0 to 1 = 4.
The minimum distance from 0 to 2 = 12. 0->1->2
The minimum distance from 0 to 3 = 19. 0->1->2->3
The minimum distance from 0 to 4 = 21. 0->7->6->5->4
The minimum distance from 0 to 5 = 11. 0->7->6->5
The minimum distance from 0 to 6 = 9. 0->7->6
The minimum distance from 0 to 7 = 8. 0->7
The minimum distance from 0 to 8 = 14. 0->1->2->8

Illustration of Dijkstra Algorithm:

To understand the Dijkstra's Algorithm lets take a graph and find the shortest path from source to all nodes.
Consider below graph and **src = 0**

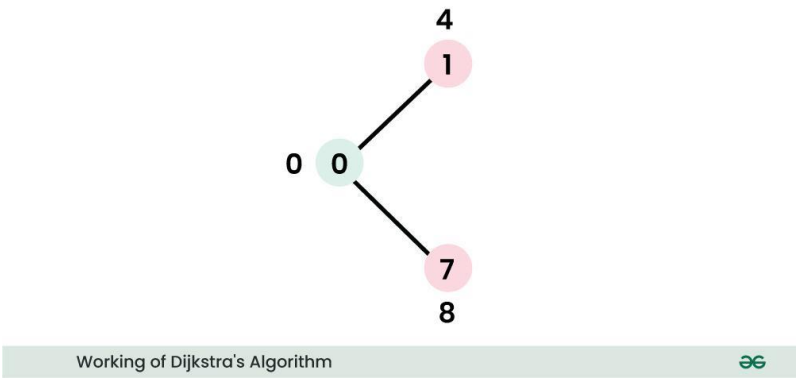


Working of Dijkstra's Algorithm

Step 1:

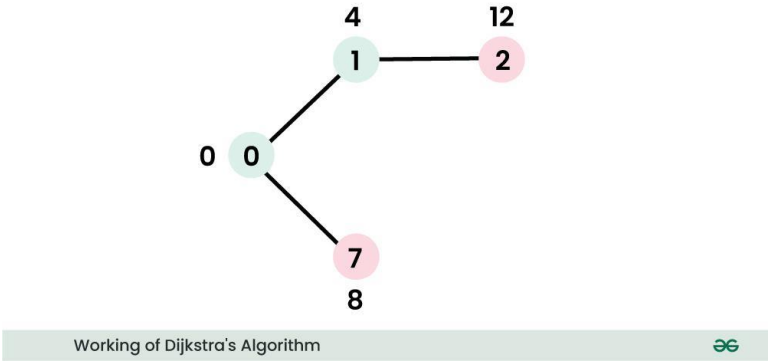
- The set **sptSet** is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where **INF** indicates infinite.
- Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in **sptSet**. So **sptSet** becomes {0}. After including 0 to **sptSet**, update distance values of its adjacent vertices.
- Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in **SPT** are shown in **green** colour.



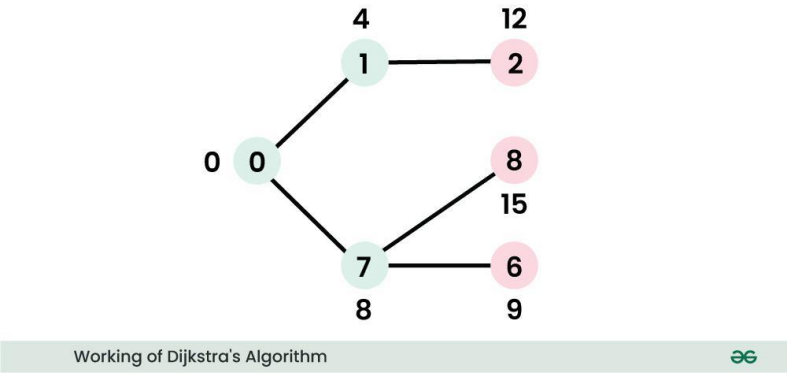
Step 2:

- Pick the vertex with minimum distance value and not already included in **SPT** (not in **sptSET**). The vertex 1 is picked and added to **sptSet**.
- So **sptSet** now becomes {0, 1}. Update the distance values of adjacent vertices of 1.
- The distance value of vertex 2 becomes **12**.



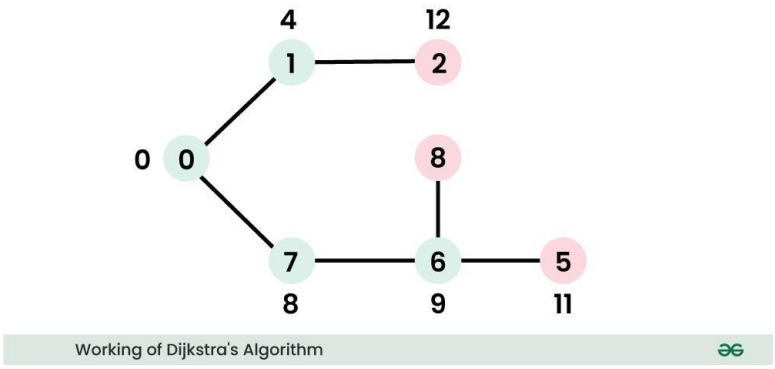
Step 3:

- Pick the vertex with minimum distance value and not already included in **SPT** (not in **sptSET**). Vertex 7 is picked. So **sptSet** now becomes **{0, 1, 7}.**
- Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (**15 and 9** respectively).

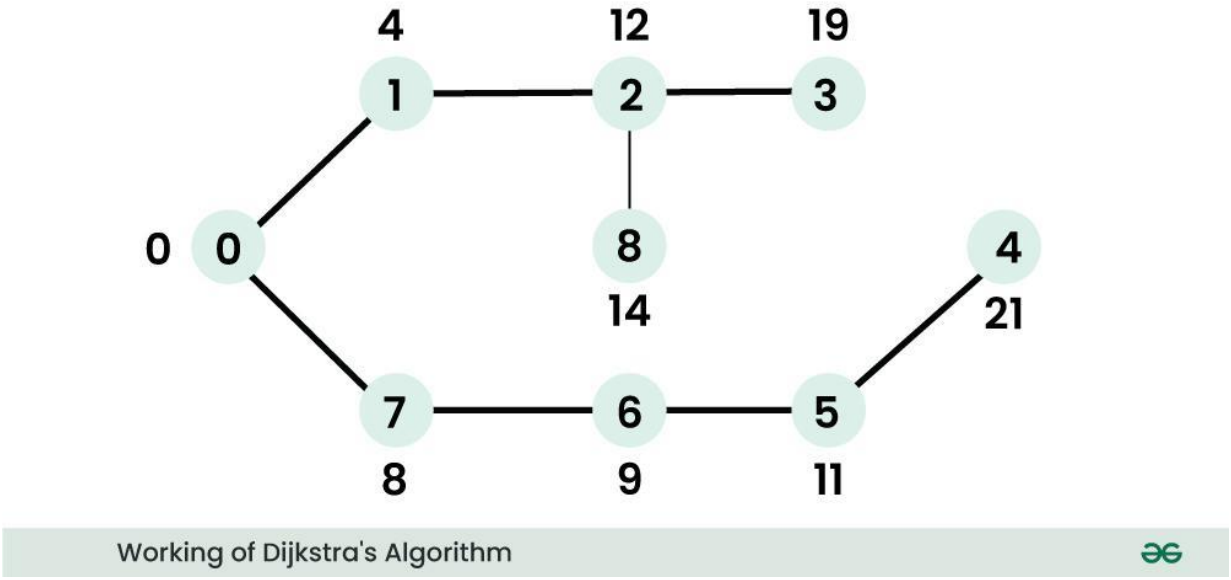


Step 4:

- Pick the vertex with minimum distance value and not already included in **SPT** (not in **sptSET**). Vertex 6 is picked. So **sptSet** now becomes **{0, 1, 7, 6}**.
- Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until **sptSet** includes all vertices of the given graph. Finally, we get the following S**hortest Path Tree (SPT).**



- **C PROGRAM:**

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
// Number of vertices in the graph
#define V 9
int minDistance(int dist[], bool sptSet[])
{
// Initialize min value
int min = INT_MAX, min_index;
for (int v = 0; v < V; v++)
if (sptSet[v] == false && dist[v] <= min)
min = dist[v], min_index = v;
return min_index;
}
// A utility function to print the constructed distance arr
void printSolution(int dist[])
{
   printf("Meet Raut\nS21 \n84\n");
   printf("+-----------------------------+-----------------------------+\n");
   printf("|            Vertex           |    Distance from Source     |\n");
   printf("+-----------------------------+-----------------------------+\n");
   for (int i = 0; i < V; i++)
   {
      printf("|\t\t\t\t%d\t\t\t\t|\t\t\t\t%d\t\t\t\t|\n", i, dist[i]);
      printf("+-----------------------------+-----------------------------+\n");
   }
}

void dijkstra(int graph[V][V], int src)
   {
   int dist[V];
   bool sptSet[V];
   for (int i = 0; i < V; i++)
   dist[i] = INT_MAX, sptSet[i] = false;
   dist[src] = 0;
   for (int count = 0; count < V - 1; count++) {
   int u = minDistance(dist, sptSet);
   sptSet[u] = true;
```

```
    for (int v = 0; v < V; v++)
    if (!sptSet[v] && graph[u][v]
    && dist[u] != INT_MAX
    && dist[u] + graph[u][v] < dist[v])
    dist[v] = dist[u] + graph[u][v];
    }
    // print the constructed distance array
    printSolution(dist);
}
// driver's code
int main()
{
/* Let us create the example graph discussed above */
int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },
{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },
{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },
{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },
{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },
{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },
{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },
{ 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
// Function call
dijkstra(graph, 0);
return 0;
}
```

- OUTPUT:

```
Meet Raut
S21
84
+-------------------------------+-------------------------------+
|             Vertex            |     Distance from Source      |
+-------------------------------+-------------------------------+
|               0               |               0               |
+-------------------------------+-------------------------------+
|               1               |               4               |
+-------------------------------+-------------------------------+
|               2               |               12              |
+-------------------------------+-------------------------------+
|               3               |               19              |
+-------------------------------+-------------------------------+
|               4               |               21              |
+-------------------------------+-------------------------------+
|               5               |               11              |
+-------------------------------+-------------------------------+
|               6               |               9               |
+-------------------------------+-------------------------------+
|               7               |               8               |
+-------------------------------+-------------------------------+
|               8               |               14              |
+-------------------------------+-------------------------------+
```

- **<u>CONCLUSION:</u> Hence, we have successfully implemented Single source shortest path using Dijkstra's Algorithm.**

Experiment 5: Kruskal/Prims Algorithm

AIM: To study & implement the Kruskal/Prims Algorithm (using Greedy approach)

THEORY:

The Kruskal and Prim's algorithms are two fundamental approaches in graph theory used for finding the minimum spanning tree (MST) of a connected, undirected graph. Both algorithms employ a greedy strategy, making locally optimal choices at each step with the goal of finding the globally optimal solution, i.e., the minimum spanning tree.

Kruskal's Algorithm:

Kruskal's algorithm starts by sorting all the edges of the graph in non-decreasing order of their weights. It then iterates through these sorted edges, adding each edge to the MST if it does not create a cycle. It maintains a forest of trees initially, where each tree is a single vertex. As it adds edges to the MST, it merges smaller trees into larger ones until all vertices are connected.

The key idea behind Kruskal's algorithm is the disjoint-set data structure, which efficiently keeps track of the connected components of the graph. This data structure allows the algorithm to determine whether adding an edge creates a cycle in near-constant time.

Prim's Algorithm:

Prim's algorithm, on the other hand, starts with an arbitrary vertex as the initial MST and then grows the MST one vertex at a time. At each step, it selects the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST. This process continues until all vertices are included in the MST.

Prim's algorithm relies on the concept of priority queues, where vertices are prioritized based on their minimum edge weight connecting them to the MST. It maintains a set of vertices in the MST and updates the priorities of vertices adjacent to the MST as edges are added.

Comparing the Algorithms:

Both Kruskal's and Prim's algorithms guarantee the construction of a minimum spanning tree, but they differ in their implementations and efficiencies.

Kruskal's algorithm is generally preferred for sparse graphs, where the number of edges is much less than the number of vertices. Its time complexity is O(E log E), where E is the number of edges.

Prim's algorithm, on the other hand, is more efficient for dense graphs, where the number of edges is close to the number of vertices. Its time complexity is O(V^2) with a simple array-based implementation and can be improved to O(E + V log V) using more advanced data structures like Fibonacci heaps.

CODE:

Kruskal algorithm:

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


#define MAX_VERTICES 100
#define MAX_EDGES 100


typedef struct {
   int u;
   int v;
   int weight;
} Edge;
typedef struct {
   int parent;
   int rank;
```

```
} Subset;


int find(Subset subsets[], int i) {
   if (subsets[i].parent != i)
      subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;
}


void Union(Subset subsets[], int x, int y) {
   int xroot = find(subsets, x);
   int yroot = find(subsets, y);
   if (subsets[xroot].rank < subsets[yroot].rank)
      subsets[xroot].parent = yroot;
   else if (subsets[xroot].rank > subsets[yroot].rank)
      subsets[yroot].parent = xroot;
   else {
      subsets[yroot].parent = xroot;
      subsets[xroot].rank++;
   }
}


int comparator(const void* a, const void* b) {
   return ((Edge*)a)->weight - ((Edge*)b)->weight;
}


void kruskalMST(Edge edges[], int V, int E) {
   Edge result[V];
   Subset subsets[V];
   int e = 0, i = 0;
```

```
   for (i = 0; i < V; ++i) {
      subsets[i].parent = i;
      subsets[i].rank = 0;
   }


   qsort(edges, E, sizeof(Edge), comparator);


   i = 0;
   while (e < V - 1 && i < E) {
      Edge next_edge = edges[i++];
      int x = find(subsets, next_edge.u);
      int y = find(subsets, next_edge.v);
      if (x != y) {
         result[e++] = next_edge;
         Union(subsets, x, y);
      }
   }


  printf("Edges in the minimum spanning tree:\n");
   for (i = 0; i < e; ++i)
      printf("%d -- %d\tWeight: %d\n", result[i].u, result[i].v,
result[i].weight);
}


int main() {
   int V = 4; // Number of vertices
   int E = 5; // Number of edges
   Edge edges[MAX_EDGES] = {{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2,
3, 4}};
   kruskalMST(edges, V, E);
   return 0;
```

}

Output

```
Edges in the minimum spanning tree:
2 -- 3  Weight: 4
0 -- 3  Weight: 5
0 -- 1  Weight: 10

=== Code Execution Successful ===
```

Prim's Algorithm:

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


#define MAX_VERTICES 100
int minKey(int key[], int mstSet[], int V) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (mstSet[v] == 0 && key[v] < min)
         min = key[v], min_index = v;
   return min_index;
}


void printMST(int parent[], int
graph[MAX_VERTICES][MAX_VERTICES], int V) {
   printf("Edges in the minimum spanning tree:\n");
   for (int i = 1; i < V; i++)
      printf("%d -- %d\tWeight: %d\n", parent[i], i,
graph[i][parent[i]]);
}
```

```
void primMST(int
graph[MAX_VERTICES][MAX_VERTICES], int V) {
   int parent[V];
   int key[V];
   int mstSet[V];


   for (int i = 0; i < V; i++)
      key[i] = INT_MAX, mstSet[i] = 0;


   key[0] = 0;
   parent[0] = -1;


   for (int count = 0; count < V - 1; count++) {
      int u = minKey(key, mstSet, V);
      mstSet[u] = 1;


      for (int v = 0; v < V; v++)
         if (graph[u][v] && mstSet[v] == 0 && graph[u][v] <
key[v])
            parent[v] = u, key[v] = graph[u][v];
   }


   printMST(parent, graph, V);
}


int main() {
   int V = 5; // Number of vertices
   int graph[MAX_VERTICES][MAX_VERTICES] = {
      {0, 2, 0, 6, 0},
```

```
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };
    primMST(graph, V);
    return 0;
}
```

OUTPUT:

```
/tmp/gs2ce6DayA.o
Edges in the minimum spanning tree:
0 -- 1  Weight: 2
1 -- 2  Weight: 3
0 -- 3  Weight: 6
1 -- 4  Weight: 5


=== Code Execution Successful ===
```

CONCLUSION:

In summary, both Kruskal's and Prim's algorithms offer efficient solutions to the minimum spanning tree problem. Kruskal's algorithm focuses on sorting edges by weight and then greedily adding them to the MST, while Prim's algorithm starts with a single vertex and grows the MST incrementally. The choice between the two algorithms often depends on the characteristics of the graph being analyzed, with Kruskal's algorithm preferred for sparse graphs and Prim's algorithm for dense graphs.

NAME: Meet Raut

BATCH: S21

ROLL NO.: 2201084

Experiment 6 : Job Sequencing With Deadlines

AIM

To implement Job sequencing with deadlines using Greedy

THEORY

Job scheduling algorithm is applied to schedule the jobs on a single processor to maximise the profits.

Here-

- You are given a set of jobs.
- Each job has a defined deadline and some profit associated with it.
- The profit of a job is given only when that job is completed within its deadline.
- Only one processor is available for processing all the jobs.
- Processor takes one unit of time to complete a job.

The greedy approach of the job scheduling algorithm states that, “Given ‘n’ number of jobs with a starting time and ending time, they need to be scheduled in such a way that maximum profit is received within the maximum deadline”.

Approach to Solution

A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.
Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.
An optimal solution of the problem would be a feasible solution which gives the maximum profit.

Job Scheduling Algorithm

Set of jobs with deadlines and profits are taken as an input with the job scheduling algorithm and scheduled subset of jobs with maximum profit are obtained as the final output.

Algorithm

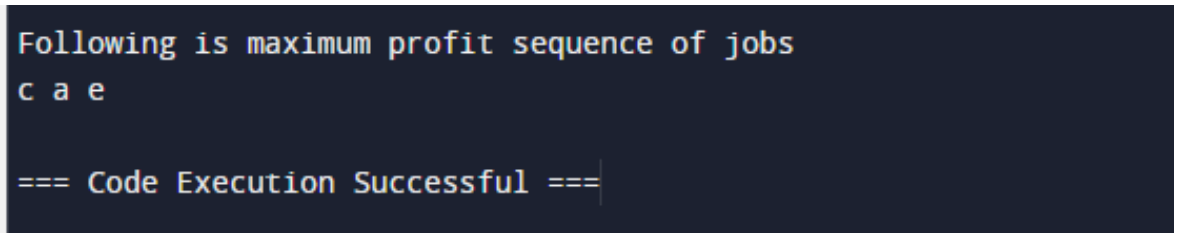
1. Step 1 − Find the maximum deadline value from the input set of jobs.
2. Step 2 − Once, the deadline is decided, arrange the jobs in descending order of their profits.
3. Step 3 − Selects the jobs with highest profits, their time periods not exceeding the maximum deadline.
4. Step 4 − The selected set of jobs are the output.

CODE

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

typedef struct Job
{

char id; int dead; int profit;

} Job;

int compare(const void *a, const void *b)
{
Job *temp1 = (Job *)a; Job *temp2 = (Job *)b;
return (temp2->profit - temp1->profit);
}

int min(int num1, int num2)
{
return (num1 > num2) ? num2 : num1;
}

void printJobScheduling(Job arr[], int n)
{
qsort(arr, n, sizeof(Job), compare); int result[n];
bool slot[n];

for (int i = 0; i < n; i++) slot[i] = false;

for (int i = 0; i < n; i++)
{

for (int j = min(n, arr[i].dead) - 1; j >= 0; j--)
{

if (slot[j] == false)
{
result[j] = i; slot[j] = true; break;
}
}
}
```

```
for (int i = 0; i < n; i++) if (slot[i])
printf("%c ", arr[result[i]].id);
}

int main()
{Job arr[] = {{'a', 2, 100},
{'b', 1, 19},
{'c', 2, 27},
{'d', 1, 25},
{'e', 3, 15}};
int n = sizeof(arr) / sizeof(arr[0]); printf(
"Following is maximum profit sequence of jobs \n");

printJobScheduling(arr, n); return 0;
}
```

OUTPUT



Conclusion

Hence we have successfully studied and implemented job sequencing with deadlines using greedy algorithm.

Experiment 7: Floyd-Warshall Algorithm

AIM: To study & implement the Floyd-Warshall Algorithm

THEORY:

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

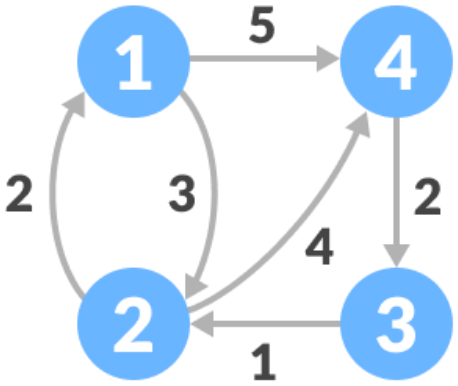
A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

This algorithm follows the dynamic programming approach to find the shortest paths.

How Floyd-Warshall Algorithm Works?

Let the given graph be:



Initial graph

Follow the steps below to find the shortest path between all the pairs of vertices.

1. **Create a matrix `A0` of dimension `n*n` where n is the number of vertices. The row and the column are indexed as `i` and `j` respectively. `i` and `j` are the vertices of the graph.**

   Each cell `A[i][j]` is filled with the distance from the `ith` vertex to the `jth` vertex. If there is no path from `ith` vertex to `jth` vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Fill each cell with the distance between ith and jth vertex

2. **Now, create a matrix `A1` using matrix `A0`. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.**

Let `k` be the intermediate vertex in the shortest path from source to destination. In this step, `k` is the first vertex. `A[i][j]` is filled with `(A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j])`.

That is, if the direct distance from the source to the destination is greater than the path through the vertex `k`, then the cell is filled with `A[i][k] + A[k][j]`.

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex k

For example: For $A_1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since $4 < 7$, $A_0[2, 4]$ is filled with 4.

3. Similarly, A_2 is created using A_1. The elements in the second column and the second row are left as they are.

In this step, `k` is the second vertex (i.e. vertex 2). The remaining

steps are the same as in step 2.

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 2

4. Similarly, A_3 and A_4 is also created.

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 3

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 4

5. A_4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

n = no of vertices

A = matrix of dimension n*n

for k = 1 to n

** for i = 1 to n**

**   for j = 1 to n**

**   Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])**

return A

CODE:

```
#include <stdio.h>

#define V 4
#define INF 99999

void printSolution(int dist[][V]);

void floydWarshall(int dist[][V]) {
   int i, j, k;

   for (k = 0; k < V; k++) {
      for (i = 0; i < V; i++) {
         for (j = 0; j < V; j++) {
```

```
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}

    printSolution(dist);
}

void printSolution(int dist[][V]) {
    printf("The following matrix shows the shortest distances between
every pair of vertices\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int graph[V][V] = {{0, 5, INF, 10},
```

```
        {INF, 0, 3, INF},
        {INF, INF, 0, 1},
        {INF, INF, INF, 0}};

    floydWarshall(graph);
    return 0;
}
```

Output:

```
/tmp/UaQahY7KYf.o
The following matrix shows the shortest distances between every pair of vertices
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0


=== Code Execution Successful ===
```

CONCLUSION:

Successfully Implemented Floyd-Warshall Algorithm

AOA-Exp 8

Name: Meet Raut

Roll No: 84
Batch: S21

Aim:
To Study and implement 0/1 knapsack problem using dynamic programming approach

Theory:
Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, i.e. the bag can hold at most W weight in it. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.
The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Algorithm:
Follow the below steps to solve the problem:
The maximum value obtained from 'N' items is the max of the following two values.
- Case 1 (include the Nth item): Value of the Nth item plus maximum value obtained by remaining N-1 items and remaining weight i.e. (W-weight of the Nth item).
- Case 2 (exclude the Nth item): Maximum value obtained by N-1 items and W weight.
- If the weight of the 'Nth'item is greater than 'W', then the Nth item cannot be included and Case 2 is the only possibility.

Example

Let us consider that the capacity of the knapsack is W = 8 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	2	4	7	10
Weight	1	3	5	7

AOA-Exp 8

Step 1

Construct an adjacency table with maximum weight of knapsack as rows and items with respective weights and profits as columns.

Values to be stored in the table are cumulative profits of the items whose weights do not exceed the maximum weight of the knapsack (designated values of each row)

So we add zeroes to the 0^{th} row and 0^{th} column because if the weight of item is 0, then it weighs nothing; if the maximum weight of knapsack is 0, then no item can be added into the knapsack.

Maximum Weights →

Items with Weights and Profits ↓

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0								
2 (3,4)	0								
3 (5,7)	0								
4 (7,10)	0								

The remaining values are filled with the maximum profit achievable with respect to the items and weight per column that can be stored in the knapsack.

The formula to store the profit values is –

$c[i,w]=\max\{c[i-1,w-w[i]]+P[i]\}$

By computing all the values using the formula, the table obtained would be −

AOA-Exp 8

Maximum Weights →

Items with Weights and Profits ↓

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	12

To find the items to be added in the knapsack, recognize the maximum profit from the table and identify the items that make up the profit, in this example, its {1, 7}.

Maximum Weights →

Items with Weights and Profits ↓

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	(12)

The optimal solution is {1, 7} with the maximum profit is 12.

AOA-Exp 8

Code:

```c
#include <stdio.h>


int max(int a, int b) { return (a > b) ? a : b; }


int knapSack(int W, int wt[], int val[], int n)
{

    if (n == 0 || W == 0)
        return 0;

    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);


    else
        return max(
            val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}


int main()
{
    int n, W;
    printf("Enter the number of items: ");
    scanf("%d", &n);

    int profit[n], weight[n];
    printf("Enter the profits of the items:\n");
    for (int i = 0; i < n; ++i) {
        scanf("%d", &profit[i]);
    }

    printf("Enter the weights of the items:\n");
    for (int i = 0; i < n; ++i) {
        scanf("%d", &weight[i]);
    }

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &W);

    printf("Maximum value in knapsack: %d", knapSack(W, weight, profit, n));
    return 0;
}
```

AOA-Exp 8

Output:

```
Enter the number of items: 5
Enter the values of the items:
3
4
2
1
5
Enter the weights of the items:
6
4
3
5
8
Enter the capacity of the knapsack: 10
Maximum value in knapsack: 7
```

NAME: Meet Raut

DIV: S2-1

ROLL.NO: 2201084

Experiment 9:

- **AIM: To study and implement Traveling Salesman Problem.**

- **THEORY:**

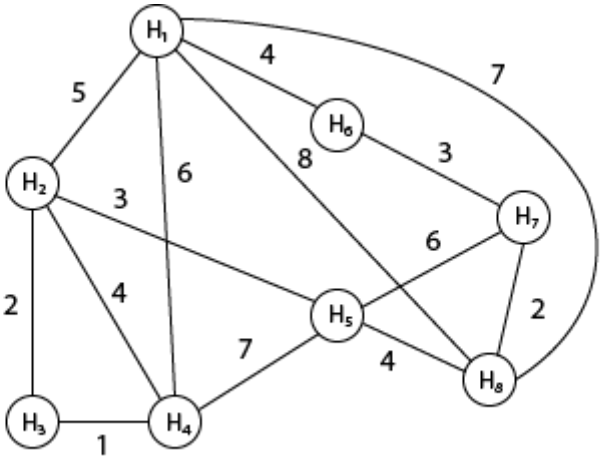
The traveling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman needs to minimize the total length of the trip.

Suppose the cities are x_1 x_2..... x_n where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesperson problem is to find a route starting and ending at x_1 that will take in all cities with the minimum cost.

In Traveling Salesman Problem, we see a complete undirected graph and a distance between each village i.e.

Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

The area assigned to the agent where he has to drop the newspaper is shown in fig:



Solution: The cost- adjacency matrix of graph G is as follows:

$cost_{ij}$ =

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(1) H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 and then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(1) H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(2) H_6	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(**H_1**)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(**H_6**)	4	0	0	0	0	0	(3)	0
(**H_7**)	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	4	0	2	0

Mark area H_8 because it is the minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(**H_1**)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(**H_6**)	4	0	0	0	0	0	(3)	0
(**H_7**)	0	0	0	0	6	3	0	(2)
(**H_8**)	7	0	0	0	(4)	0	2	0

Mark area H_5 because it is the minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
(H5)	0	(3)	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H_2 because it is the minimum cost area reachable from H_2

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
(H2)	5	0	(2)	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
(H5)	0	(3)	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H_3 because it is the minimum cost area reachable from H_3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_4 and then select the minimum cost area reachable from H_4 it is H_1.So, using the greedy strategy, we get the following.

$$H_1 \xrightarrow{4} H_6 \xrightarrow{3} H_7 \xrightarrow{2} H_8 \xrightarrow{4} H_5 \xrightarrow{3} H_2 \xrightarrow{2} H_3 \xrightarrow{1} H_4 \xrightarrow{6} H_1.$$

Thus, the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

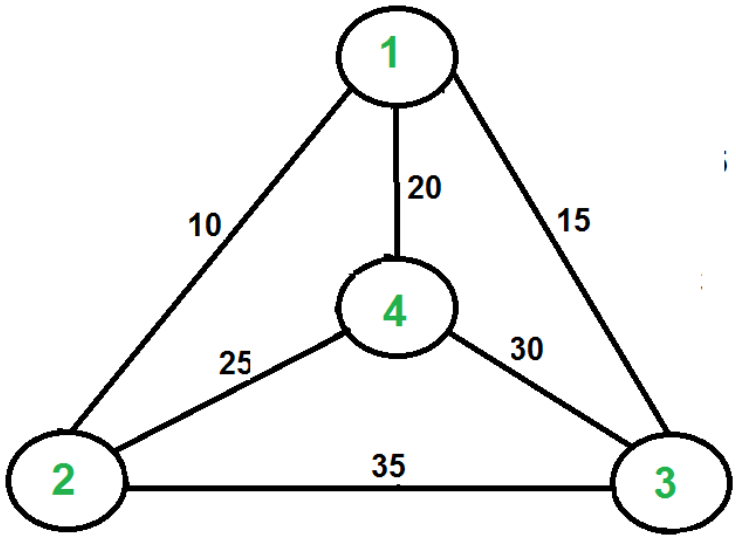
Matroids:

A matroid is an ordered pair M(S, I) satisfying the following conditions:

1. S is a finite set.
2. I is a nonempty family of subsets of S, called the independent subsets of S, such that if $B \in I$ and $A \in I$. We say that I is hereditary if it satisfies this property. Note that the empty set $\emptyset$ is necessarily a member of I.
3. If $A \in I$, $B \in I$ and $|A| < |B|$, then there is some element $x \in B - A$ such that $A \cup \{x\} \in I$. We say that M satisfies the exchange property.

We say that a matroid M (S, I) is weighted if there is an associated weight function w that assigns a strictly positive weight w (x) to each element $x \in S$. The weight function w extends to a subset of S by summation:

$$w(A) = \sum_{x \in A} w(x)$$



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP-hard problem. There is no polynomial-time know solution for this problem. The following are different solutions for the traveling salesman problem.

Naive Solution:

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! Permutations of cities.

3) Calculate the cost of every permutation and keep track of the minimum cost permutation.

4) Return the permutation with minimum cost.

Time Complexity: (n!)

Dynamic Programming:

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far.

Now the question is how to get cost(i)? To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Let us define a term C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i. We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},
 C(S, i) = dist(1, i)
Else if size of S is greater than 2.
 C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.
```

Below is the dynamic programming solution for the problem using top down recursive+memoized approach:-

For maintaining the subsets we can use the bitmasks to represent the remaining nodes in our subset. Since bits are faster to operate and there are only few nodes in graph, bitmasks is better to use.

For example: –

10100 represents node 2 and node 4 are left in set to be processed

010010 represents node 1 and 4 are left in subset.

NOTE:- ignore the 0th bit since our graph is 1-based

Three popular Travelling Salesman Problem Algorithms

Here are some of the most popular solutions to the Travelling Salesman Problem:

1. The brute-force approach

The Brute Force approach, also known as the Naive Approach, calculates and compares all possible permutations of routes or paths to determine the shortest unique solution. To solve the TSP using the Brute-Force approach, you must calculate the total number of routes and then draw and list all the possible routes. Calculate the distance of each route and then choose the shortest one—this is the optimal solution.

This is only feasible for small problems, rarely useful beyond theoretical computer science tutorials.

2. The branch and bound method

The branch and bound algorithm starts by creating an initial route, typically from the starting point to the first node in a set of cities. Then, it systematically explores different permutations to extend the route one node at a time. Each time a new node is added, the algorithm calculates the current path's length and compares it to the optimal route found so far. If the current path is already longer than the optimal route, it "bounds" or prunes that branch of the exploration, as it would not lead to a more optimal solution.

This pruning is the key to making the algorithm efficient. By discarding unpromising paths, the search space is narrowed down, and the algorithm can focus on exploring only the most promising paths. The process continues until all possible routes are explored, and the shortest one is identified as the optimal solution to the traveling salesman problem. Branch and bound is an effective greedy approach for tackling NP-hard optimization problems like the travelling salesman problem.

3. The nearest neighbor method

To implement the Nearest Neighbor algorithm, we begin at a randomly selected starting point. From there, we find the closest unvisited node and add it to the sequencing. Then, we move to the next node and repeat the process of finding the nearest unvisited node until all nodes are included in the tour. Finally, we return to the starting city to complete the cycle.

While the Nearest Neighbor approach is relatively easy to understand and quick to execute, it rarely finds the optimal solution for the traveling salesperson problem. It can be significantly longer than the optimal route, especially for large and complex instances. Nonetheless, the Nearest Neighbor algorithm serves as a good starting point for tackling the travelling salesman problem and can be useful when a quick and reasonably good solution is needed.

This greedy algorithm can be used effectively as a way to generate an initial feasible solution quickly, to then feed into a more sophisticated local search algorithm, which then tweaks the solution until a given stopping condition.

- **C PROGRAM:**

```
#include<stdio.h>

int ary[10][10], completed[10],  n, cost=0;

void takeInput()

{

int i, j;

printf("Enter the number of villages: ");

scanf("%d",&n);



printf("\nEnter the Cost Matrix\n");



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

{

printf("\nEnter Elements of Row: %d\n",i+1);
```

```
for( j=0;j < n;j++)
scanf("%d",&ary[i][j]);
completed[i]=0;
}

printf("\n\nThe cost list is:");

for( i=0;i < n;i++)
{
printf("\n");
for(j=0;j < n;j++)
printf("\t%d",ary[i][j]);
}
}

void mincost(int city)
{
int i,ncity;
completed[city]=1;

printf("%d--->",city+1);
```

```
ncity=least(city);


if(ncity==999)
{
ncity=0;
printf("%d",ncity+1);
cost+=ary[city][ncity];
return;
}
mincost(ncity);
}


int least(int c)
{
int i,nc=999;
int min=999,kmin;


for(i=0;i < n;i++)
{
if((ary[c][i]!=0)&&(completed[i]==0))
if(ary[c][i]+ary[i][c] < min)
{
```

```
min=ary[i][0]+ary[c][i];
kmin=ary[c][i];
nc=i;
}
}

if(min!=999)
cost=cost + kmin;
return nc;
}

int main()
{
takeInput();
printf("\n\nThe Path is:\n");
mincost(0); //passing 0 because starting vertex
printf("\n\nMinimum cost is %d\n ",cost);
return 0;
}
```

OUTPUT:

```
--------------- TRAVELING SALESMAN PROBLEM (TSP) ---------------

Enter the number of villages: 4

Enter the Cost Matrix

Enter Elements of Row: 1
0 4 1 3

Enter Elements of Row: 2
4 0 2 1

Enter Elements of Row: 3
1 2 0 5

Enter Elements of Row: 4
3 1 5 0


The cost list is:
    0   4   1   3
    4   0   2   1
    1   2   0   5
    3   1   5   0

The Path is:
1--->3--->2--->4--->1

Minimum cost is 7
```

- **<u>CONCLUSION:</u> Hence, we have successfully implemented Traveling Salesman Problem (TSP); LO 1, LO 2.**

Name : Meet Raut
Batch : S21 Roll No. : 84

Experiment 10 : Sum of Subsets

Aim

To implement the Subset Sum problem.

Theory

Given a set of non-negative integers and a value sum, the task is to check if there is a subset of the given set whose sum is equal to the given sum.

Subset Sum Problem using Recursion:
For the recursive approach, there will be two cases.

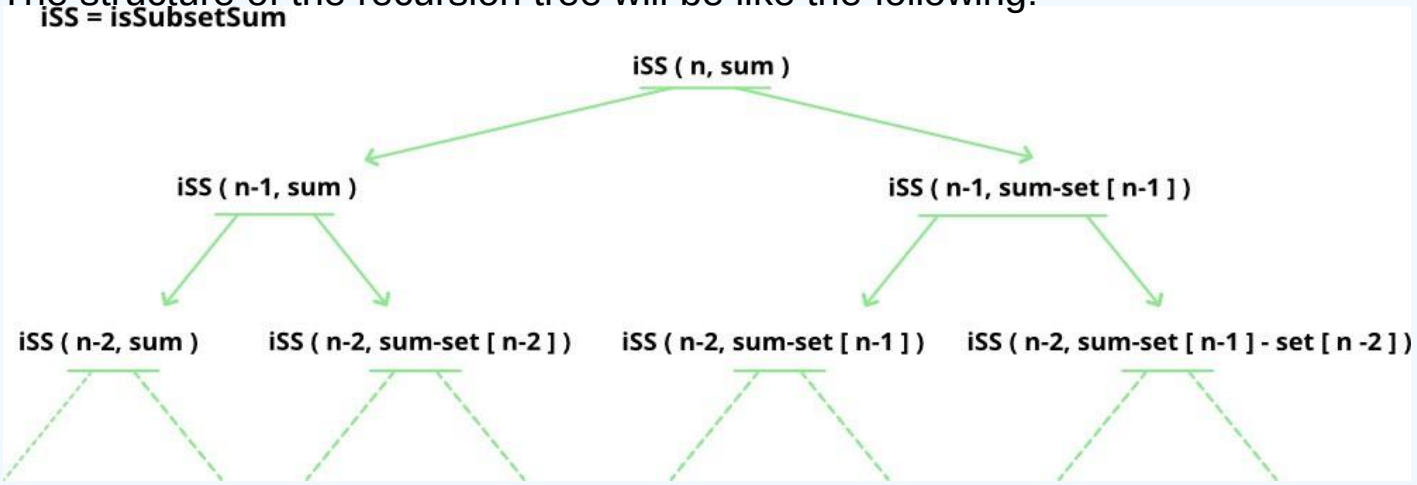
Consider the ‘last’ element to be a part of the subset. Now the new required sum = required sum – value of ‘last’ element.
Don’t include the ‘last’ element in the subset. Then the new required sum = old required sum.
In both cases, the number of available elements decreases by 1.

Mathematically the recurrence relation will look like the following:

isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) | isSubsetSum(set, n-1, sum-set[n-1])

Base Cases:
isSubsetSum(set, n, sum) = false, if sum > 0 and n = 0
isSubsetSum(set, n, sum) = true, if sum = 0

The structure of the recursion tree will be like the following:



Follow the below steps to implement the recursion:

- Build a recursive function and pass the index to be considered (here gradually moving from the last end) and the remaining sum amount.
- For each index check the base cases and utilise the above recursive call.
- If the answer is true for any recursion call, then there exists such a subset. Otherwise, no such subset exists.

Complexity Analysis:

- Time Complexity: O(2n) The above solution may try all subsets of the given set in the worst case. Therefore the time complexity of the above solution is exponential. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).
- Auxiliary Space: O(n) where n is recursion stack space.

Code

```
// Online C++ compiler to run C++ program online
#include <iostream>
#include <vector>
using namespace std;
vector<vector<int>> ans;

void subsetSum(vector<int> &nums, vector<int> &temp, int sum, int i, int k)
{
    if(sum == k)
    {
        ans.push_back(temp);
        return;
    }
    if(sum > k) return;
    if(i >= nums.size()) return;

    temp.push_back(nums[i]);
    subsetSum(nums, temp, sum+nums[i], i+1, k);
    temp.pop_back();
    subsetSum(nums, temp, sum, i+1, k);
}

int main() {
    // Write C++ code here
    vector<int> nums = {1,2,3,4,1};
    vector<int> temp;
     subsetSum(nums, temp, 0, 0, 6);
```

```
    for(auto i : ans)
    {
        for(auto j : i)
        {
            cout<<j<<" ";
        }
        cout<<endl;
    }

    return 0;
}
```

Output

```
1 2 3
1 4 1
2 3 1
2 4
```

Conclusion

Hence we have successfully studied and implemented the Subset Sum Problem.

NAME: Meet Raut

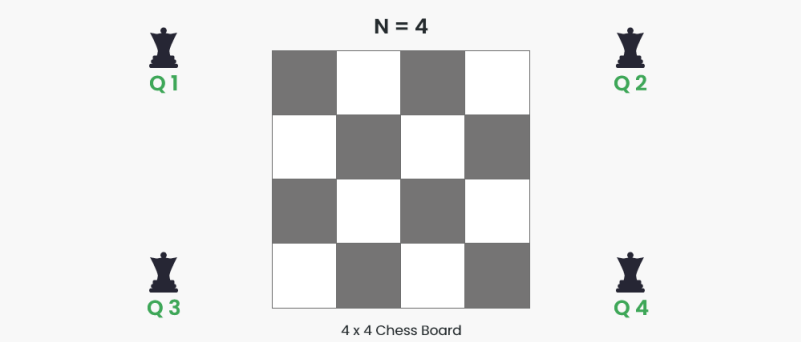
DIV: S2-1

ROLL.NO: 2201084

Experiment 11:

- **AIM:** **To study and implement N Queen Problem using backtracking approach.**

- **THEORY:**



The **N** Queen is the problem of placing **N** chess queens on an **N×N** chessboard so that no two queens attack each other.
For example, the following is a solution for the 4 Queen problem.

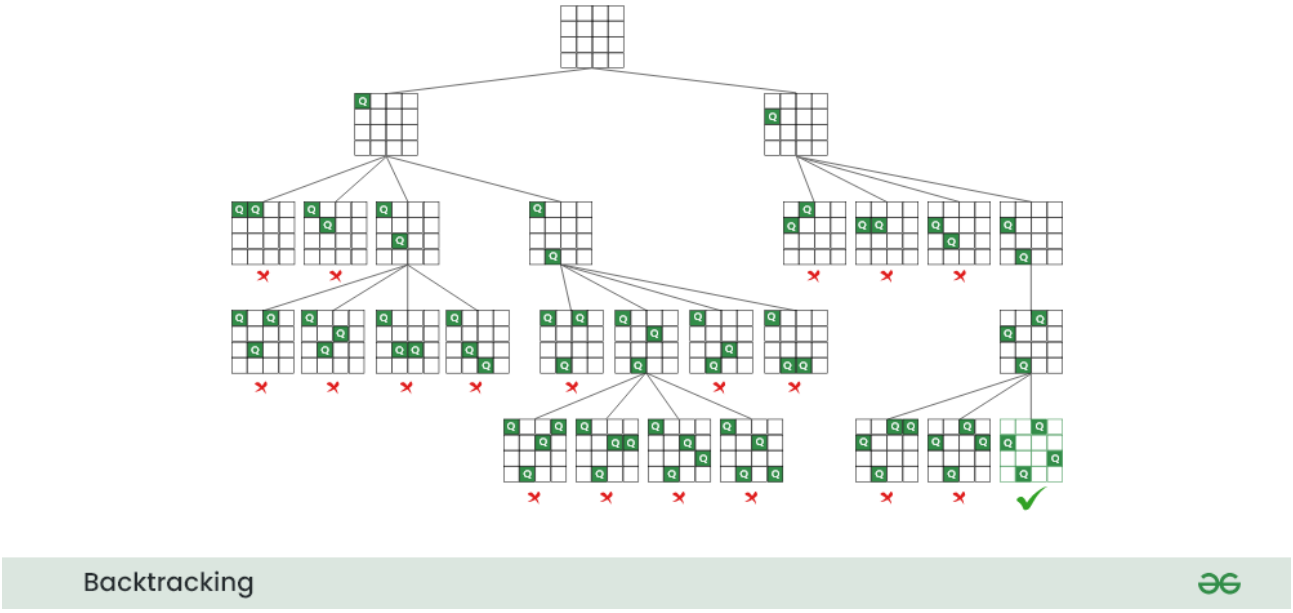
The expected output is in the form of a matrix that has '**Q**'s for the blocks where queens are placed and the empty spaces are represented by **'.'** . For example, the following is the output matrix for the above 4-Queen solution.

```
. Q . .
. . . Q
Q . . .
. . Q .
```

N Queen Problem using Backtracking:

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**.

Below is the recursive tree of the above approach:



Backtracking

Follow the steps mentioned below to implement the idea:

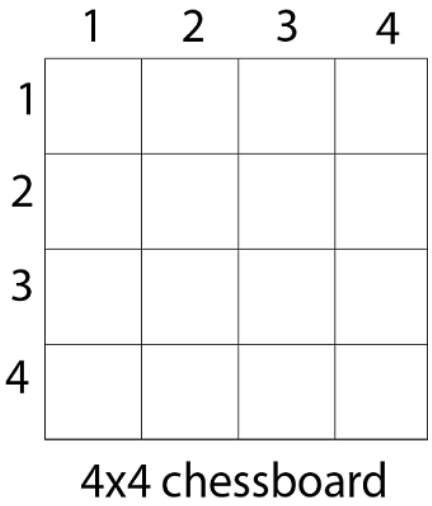
- Start in the leftmost column
- If all queens are placed return true
- Try all rows in the current column. Do the following for every row.
    - If the queen can be placed safely in this row
        - Then mark this **[row, column]** as part of the solution and recursively check if placing queen here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

        - If placing the queen in **[row, column]** leads to a solution then return **true**.
        - If placing queen doesn't lead to a solution then unmark this **[row, column]** then backtrack and try other rows.
    - If all rows have been tried and valid solution is not found return **false** to trigger backtracking.

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

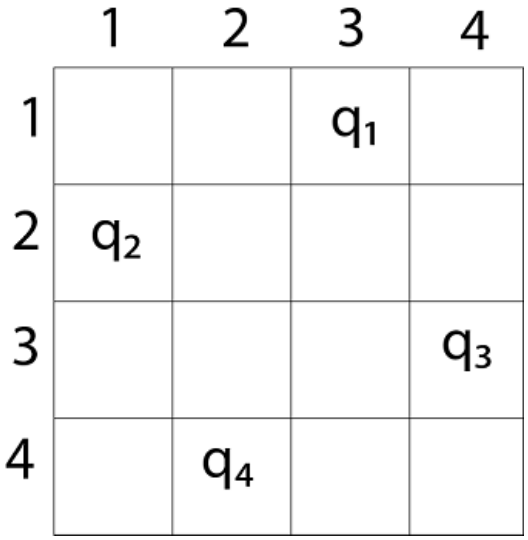
It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.



The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

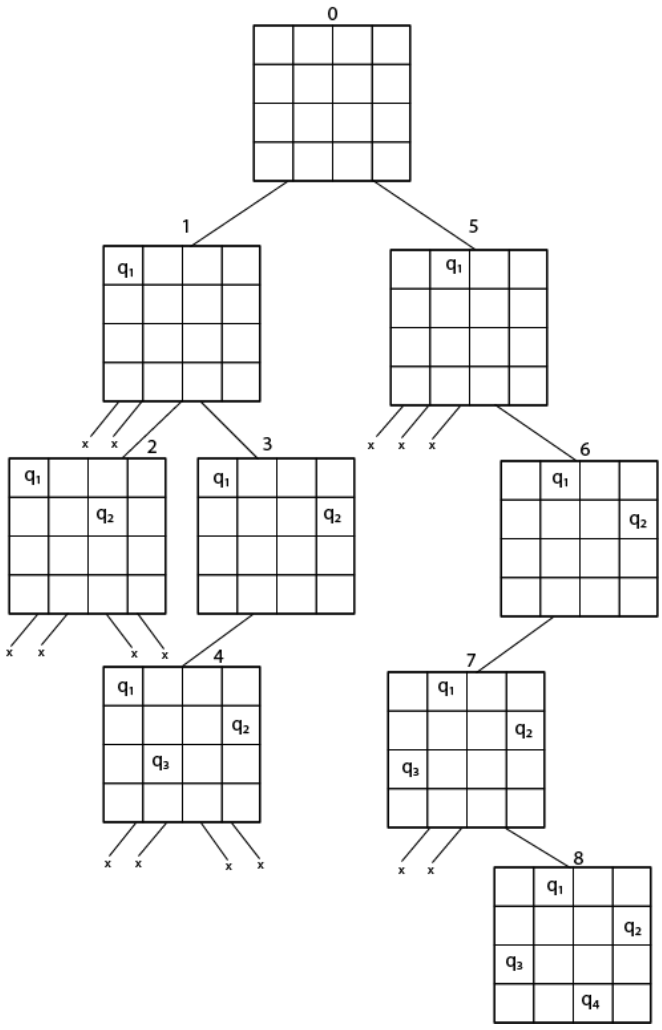
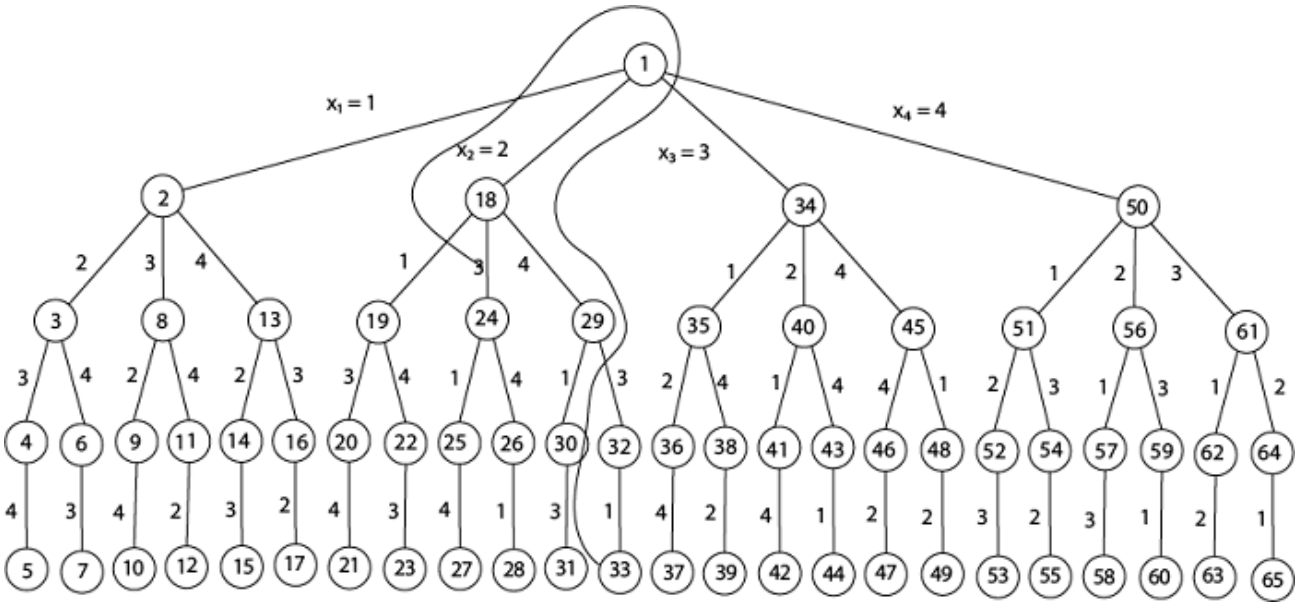


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:

	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5							q_5	
6	q_6							
7			q_7					
8					q_8			

Thus, the solution for 8 -queen problem for (4, 6, 8, 2, 7, 1, 3, 5).

If two queens are placed at position (i, j) and (k, l).

Then they are on same diagonal only if (i - j) = k - l or i + j = k + l.

The first equation implies that j - l = i - k.

The second equation implies that j - l = k - i.

Therefore, two queens lie on the duplicate diagonal if and only if |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

```
Place (k, i)
  {
    For j ← 1 to k - 1
     do if (x [j] = i)
      or (Abs x [j]) - i) = (Abs (j - k))
   then return false;
    return true;
}
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
N - Queens (k, n)
{
  For i ← 1 to n
      do if Place (k, i) then
  {
    x [k] ← i;
    if (k ==n) then
      write (x [1....n));
    else
    N - Queens (k + 1, n);
  }
}
```

N Queen Problem Algorithm

1. We create a board of N x N size that stores characters. It will store 'Q' if the queen has been placed at that position else '.'
2. We will create a recursive function called "solve" that takes board and column and all Boards (that stores all the possible arrangements) as arguments. We will pass the column as 0 so that we can start exploring the arrangements from column 1.
3. In solve function we will go row by row for each column and will check if that particular cell is safe or not for the placement of the queen, we will do so with the help of isSafe() function.
4. For each possible cell where the queen is going to be placed, we will first check isSafe() function.
5. If the cell is safe, we put 'Q' in that row and column of the board and again call the solve function by incrementing the column by 1.
6. Whenever we reach a position where the column becomes equal to board length, this implies that all the columns and possible arrangements have been explored, and so we return.
7. Coming on to the boolean isSafe() function, we check if a queen is already present in that row/ column/upper left diagonal/lower left diagonal/upper right diagonal /lower right diagonal. If the queen is present in any of the directions, we return false. Else we put board[row][col] = 'Q' and return true.

Time & Space Complexity

As we got to know in the algorithm for each cell, to check if the queen can be placed there or not, we are iterating for N times. So the recurrence relation comes out to be:

T(N) = N * T(N-1) + N.

T(N-1) = N * T(N-2) + N

T(1) = 1

This totals T(N) = N* N!. therefore, the time complexity comes out to be O(N * N!).

And as we have used an extra board of characters of N x N Size, The space complexity comes out to be O(N *N).

- **C PROGRAM:**

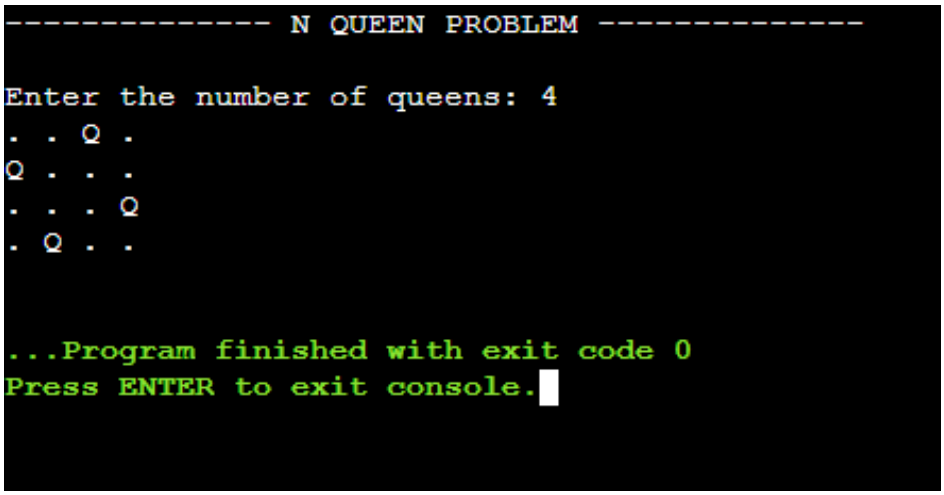
```
#include <stdio.h>
#include <stdlib.h>

int isSafe(int row, int col, char **board, int n) {
   int i, j;

   for (j = 0; j < col; j++) {
      if (board[row][j] == 'Q') {
         return 0;
      }
   }

   for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
      if (board[i][j] == 'Q') {
         return 0;
      }
```

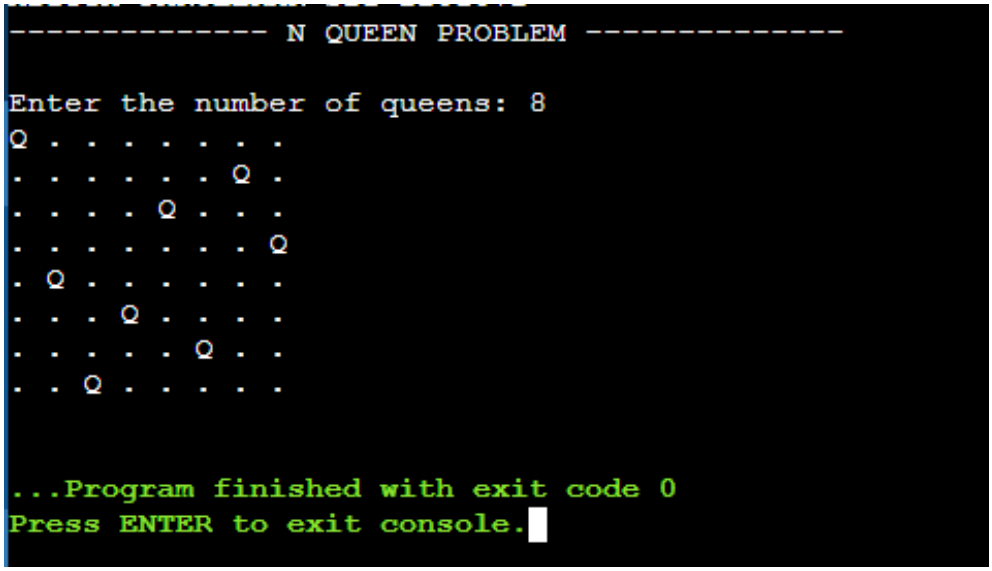
```
    }

    for (i = row, j = col; j >= 0 && i < n; i++, j--) {
        if (board[i][j] == 'Q') {
            return 0;
        }
    }

    return 1;
}

int solveNQueens(char **board, int col, int n) {
    if (col >= n) {
        return 1;
    }

    for (int i = 0; i < n; i++) {
        if (isSafe(i, col, board, n)) {
            board[i][col] = 'Q';

            if (solveNQueens(board, col + 1, n)) {
                return 1;
            }
            board[i][col] = '.';
        }
    }
    return 0;
}
```

```
void printBoard(char **board, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%c ", board[i][j]);
        }
        printf("\n");
    }
}

void solution(int n) {
    char **board = (char **)malloc(n * sizeof(char *));
    for (int i = 0; i < n; i++) {
        board[i] = (char *)malloc(n * sizeof(char));
        for (int j = 0; j < n; j++) {
            board[i][j] = '.';
        }
    }

    if (solveNQueens(board, 0, n) == 0) {
        printf("Solution does not exist");
        return;
    }

    printBoard(board, n);

    for (int i = 0; i < n; i++) {
        free(board[i]);
    }
    free(board);
```

```
}

int main() {
    int n;
    printf("---------- N QUEEN PROBLEM ----------\n\n");
    printf("Enter the number of queens: ");
    scanf("%d", &n);

    solution(n);
    return 0;
}
```

- **OUTPUT:**





- **<u>CONCLUSION:</u> Hence, we have successfully implemented N Queen Problem using backtracking approach; LO 1, LO 2.**

NAME: Meet Raut

DIV: S21

ROLL.NO: 2201084

Experiment 12:

- **AIM:** To study and implement Knuth-Morris-Pratt algorithm.

- **THEORY:**

The Knuth-Morris-Pratt (KMP)Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

The Prefix Function (Π)

Following pseudo code compute the prefix function, Π:

```
COMPUTE- PREFIX- FUNCTION (P)
1. m ←length [P]                //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π
```

Running Time Analysis:

In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

Example: Compute Π for the pattern 'p' below:

P :

a	b	a	b	a	c	a

Solution:

```
Initially: m = length [p] = 7
           Π [1] = 0
           k = 0
```

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

```
KMP-MATCHER (T, P)
 1. n ← length [T]
 2. m ← length [P]
 3. Π← COMPUTE-PREFIX-FUNCTION (P)
 4. q ← 0                // numbers of characters matched
 5. for i ← 1 to n       // scan S from left to right
 6. do while q > 0 and P [q + 1] ≠ T [i]
 7. do q ← Π [q]                 // next character does not match
 8. If P [q + 1] = T [i]
 9. then q ← q + 1               // next character matches
 10. If q = m                              // is all of p matched?
 11. then print "Pattern occurs with shift" i - m
 12. q ← Π [q]                           // look for the next match
```

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

Example: Given a string 'T' and pattern 'P' as follows:

T:	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:	a	b	a	b	a	c	a

Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

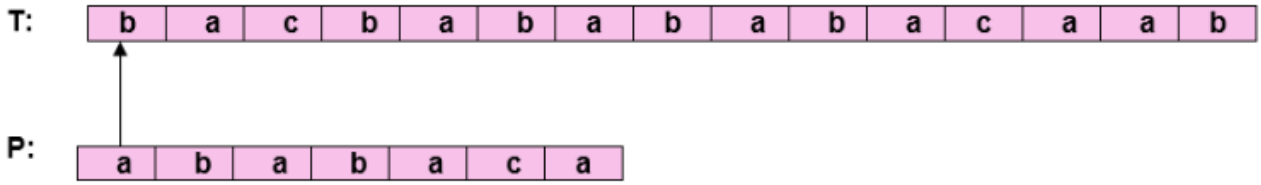
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

```
Initially: n = size of T = 15
m = size of P = 7
```

Step1: i=1, q=0

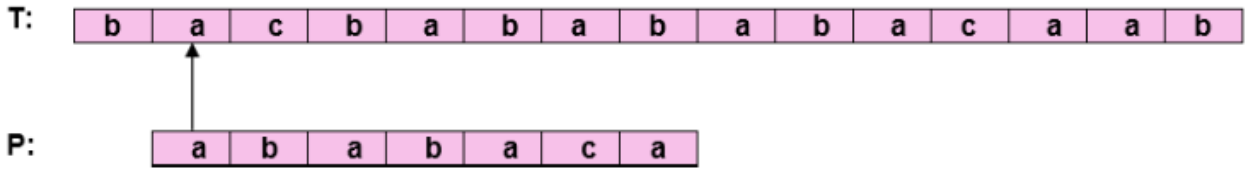
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

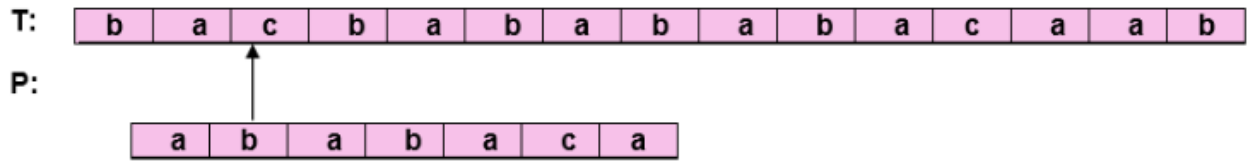
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

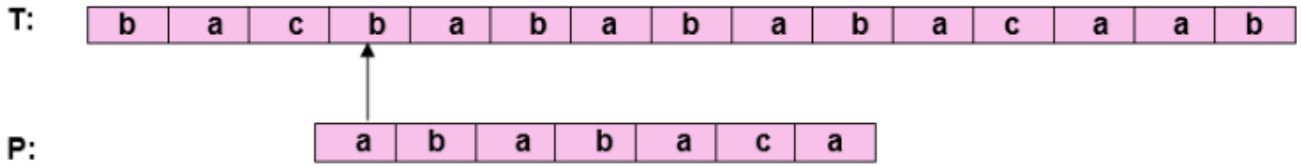
Comparing P [2] with T [3] P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

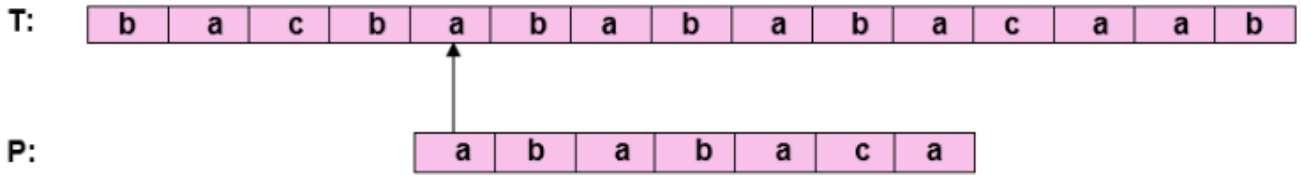
Step4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



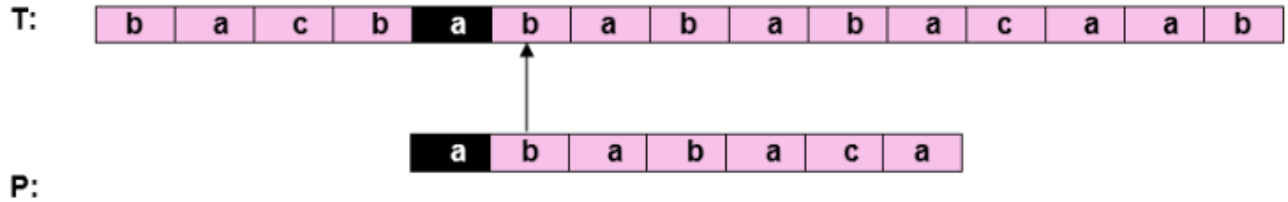
Step5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



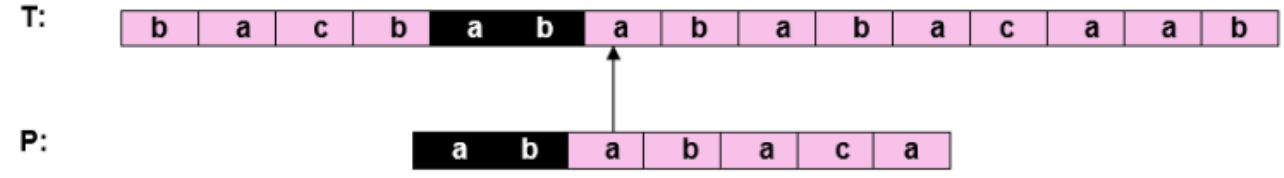
Step6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



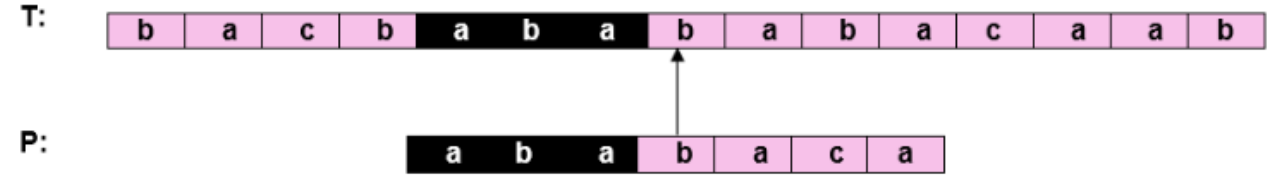
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



Step8: i = 8, q =3

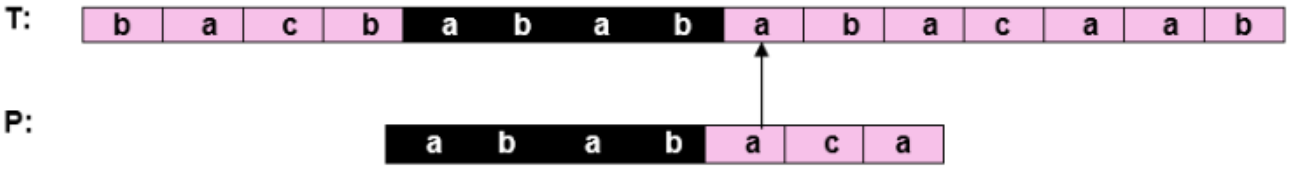
Comparing P [4] with T [8] P [4] matches with T [8]



Step9: i = 9, q = 4

Comparing P [5] with T [9]

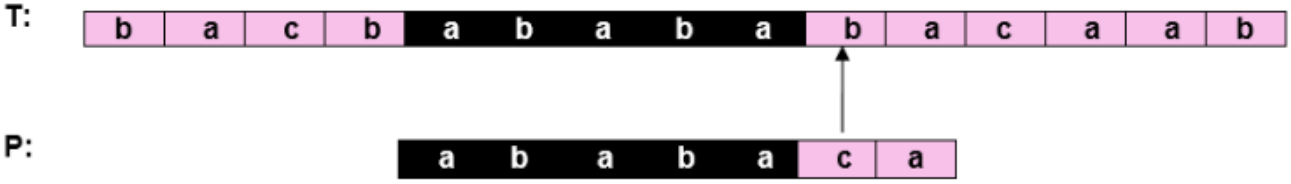
P [5] matches with T [9]



Step10: i = 10, q = 5

Comparing P [6] with T [10]

P [6] doesn't match with T [10]

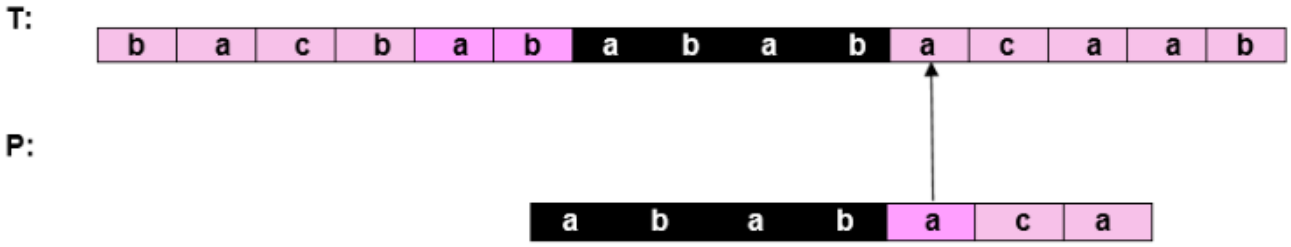


Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

Step11: i = 11, q =4

Comparing P [5] with T [11]

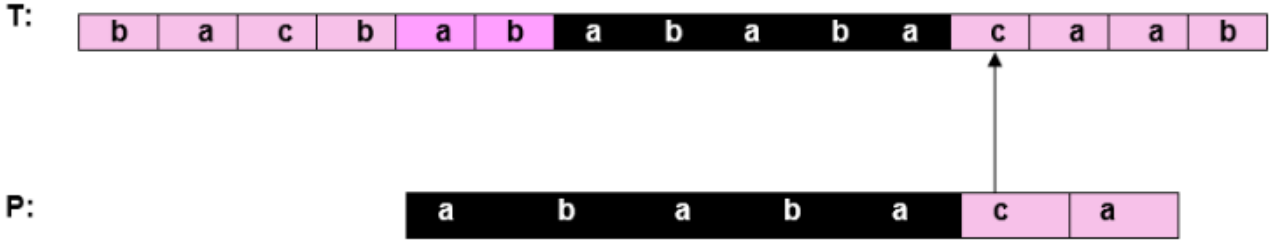
P [5] match with T [11]

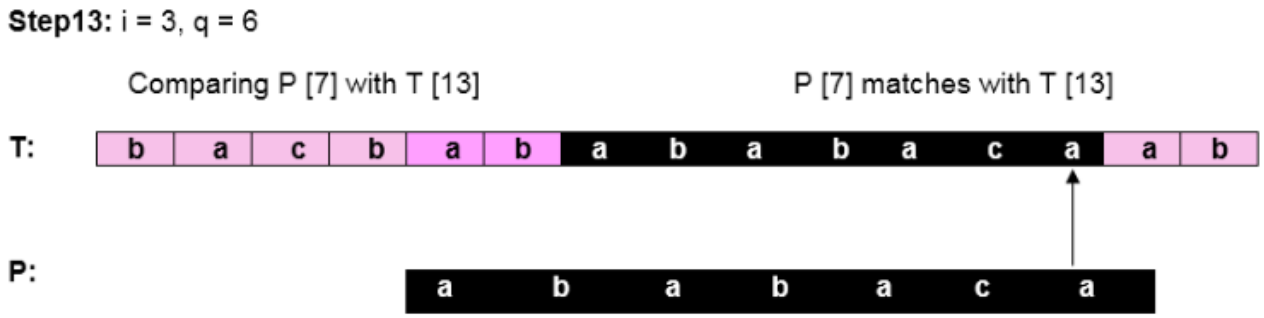


Step12: i = 12, q = 5

Comparing P [6] with T [12]

P [6] matches with T [12]





Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

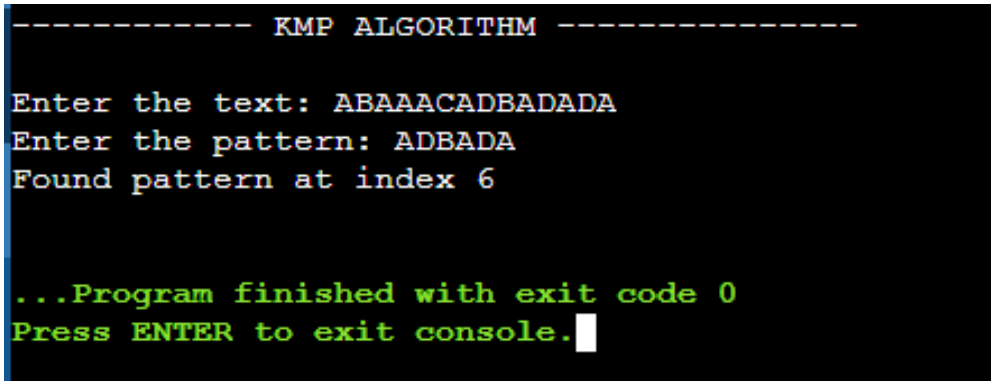
- **C PROGRAM:**

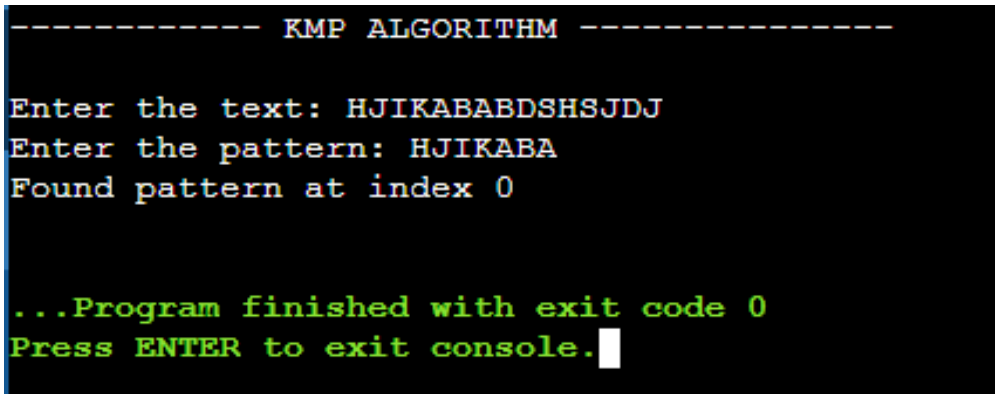
```
#include <stdio.h>
#include <string.h>

void computeLPSArray(char* pat, int M, int* lps);

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

    int lps[M];

    computeLPSArray(pat, M, lps);

    int i = 0;
    int j = 0;
    while ((N - i) >= (M - j)) {
```

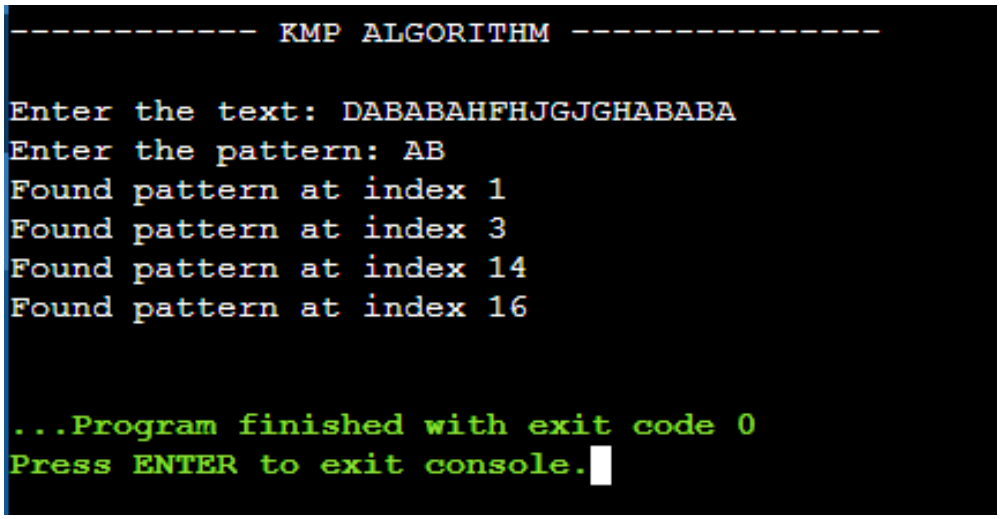
```
            if (pat[j] == txt[i]) {
                j++;
                i++;
            }

            if (j == M) {
                printf("Found pattern at index %d\n", i - j);
                j = lps[j - 1];
            }
            else if (i < N && pat[j] != txt[i]) {
                if (j != 0)
                    j = lps[j - 1];
                else
                    i = i + 1;
            }
        }
}

void computeLPSArray(char* pat, int M, int* lps)
{
        int len = 0;
        lps[0] = 0;

        int i = 1;
        while (i < M) {
            if (pat[i] == pat[len]) {
                len++;
                lps[i] = len;
                i++;
```

```
            }
            else {
                if (len != 0) {
                    len = lps[len - 1];
                }
                else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
}

int main()
{
    char txt[100], pat[100];

    printf("------------ KMP ALGORITHM ----------\n\n");

    printf("Enter the text: ");
    scanf("%s", txt);

    printf("Enter the pattern: ");
    scanf("%s", pat);

    KMPSearch(pat, txt);
    return 0;
}
```

- **OUTPUT:**







- **CONCLUSION: Hence, we have successfully implemented Knuth-Morris-Pratt algorithm; LO 1, LO 2.**

NAME: Meet Raut

DIV: S21

ROLL.NO: 2201084

Experiment 13:

- **AIM: To study and implement Rabin-Karp Algorithm for Pattern Searching.**

- **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.

```
RABIN-KARP-MATCHER (T, P, d, q)
 1. n ← length [T]
 2. m  ← length [P]
 3. h  ←  d^(m-1) mod q
 4. p ←  0
 5. t_0 ←  0
 6. for i ← 1 to m
 7. do p ←  (dp + P[i]) mod q
 8. t_0 ← (dt_0+T [i]) mod q
 9. for s  ←  0 to n-m
10. do if p = t_s
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then t_s+1 ←  (d (t_s-T [s+1]h)+T [s+m+1])mod q
```

Example: For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.....

T = 31415926535.....

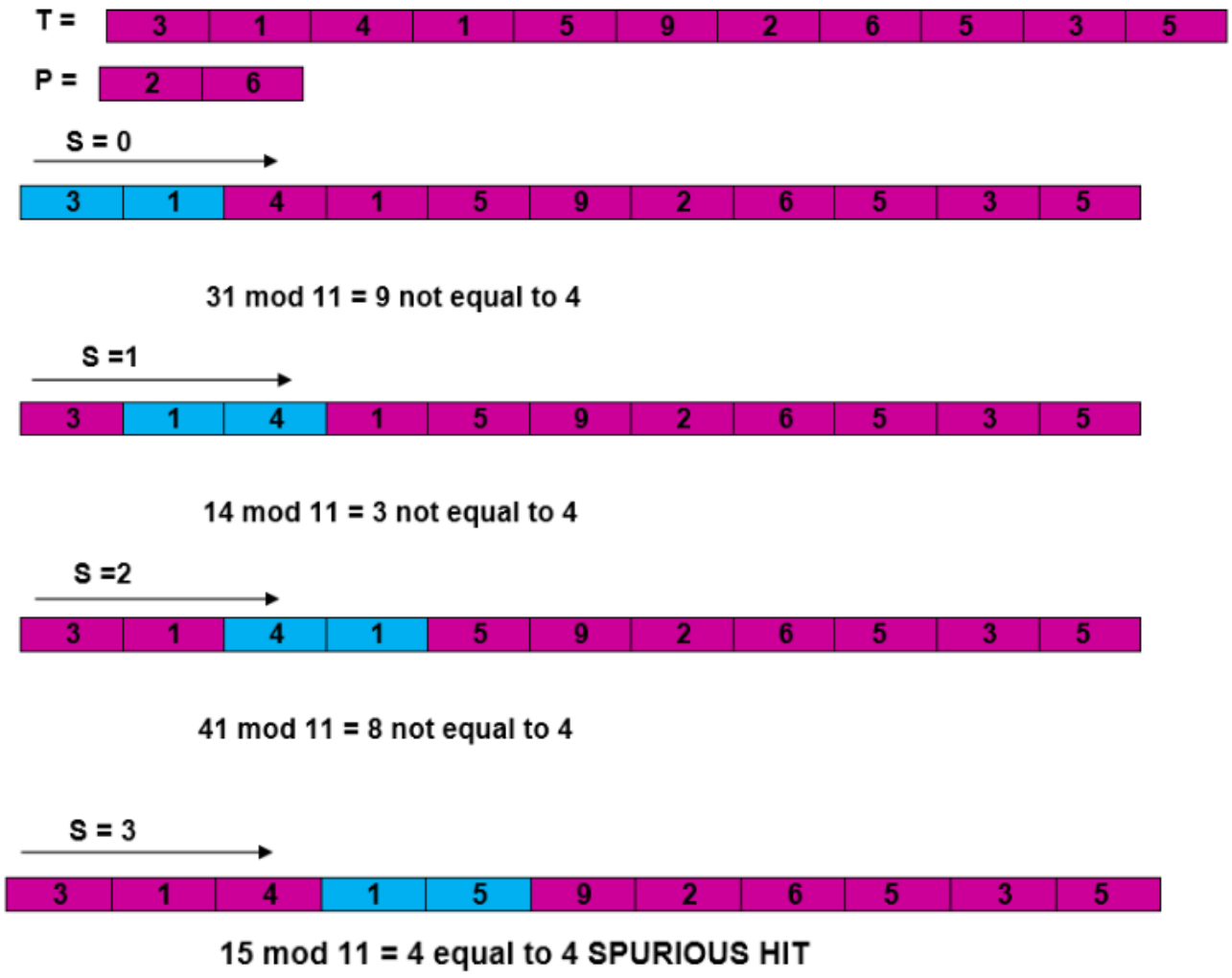
P = 26

Here T.Length =11 so Q = 11

And P mod Q = 26 mod 11 = 4

Now find the exact match of P mod Q...

Solution:

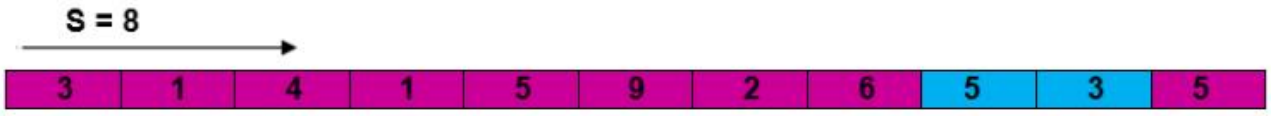


59 mod 11 = 4 equal to 4 **SPURIOUS HIT**

92 mod 11 = 4 equal to 4 **SPURIOUS HIT**

26 mod 11 = 4 **EXACT MATCH**

65 mod 11 = 10 not equal to 4



53 mod 11 = 9 not equal to 4

35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

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.

Given a text T**[0. . .n-1]** and a pattern P**[0. . .m-1]**, write a function search(char P[], char T[]) that prints all occurrences of P[] present in T[] using Rabin Karp algorithm. You may assume that **n > m**.

Examples:

Input: *T[] = "THIS IS A TEST TEXT", P[] = "TEST"*
Output: *Pattern found at index 10*

Input: *T[] = "AABAACAADAABAABA", P[] = "AABA"*
Output: *Pattern found at index 0*
Pattern found at index 9
Pattern found at index 12

How is Hash Value calculated in Rabin-Karp?

Hash value is used to efficiently check for potential matches between a **pattern** and substrings of a larger **text**. The hash value is calculated using a **rolling hash function**, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the **text** and calculate the hash value for each substring without recalculating the entire hash from scratch.
Here's how the hash value is typically calculated in Rabin-Karp:
Step 1: Choose a suitable **base** and a **modulus**:

- Select a prime number '**p**' as the modulus. This choice helps avoid overflow issues and ensures a good distribution of hash values.
- Choose a base '**b**' (usually a prime number as well), which is often the size of the character set (e.g., 256 for ASCII characters).

Step 2: Initialize the hash value:

- Set an initial hash value '**hash**' to **0**.

Step 3: Calculate the initial hash value for the **pattern**:

- Iterate over each character in the **pattern** from **left** to **right**.
- For each character **'c'** at position **'i'**, calculate its contribution to the hash value as **'c * (bpattern_length – i – 1) % p'** and add it to '**hash**'.
- This gives you the hash value for the entire **pattern**.

Step 4: Slide the pattern over the **text**:

- Start by calculating the hash value for the first substring of the **text** that is the same length as the **pattern**.

Step 5: Update the hash value for each subsequent substring:

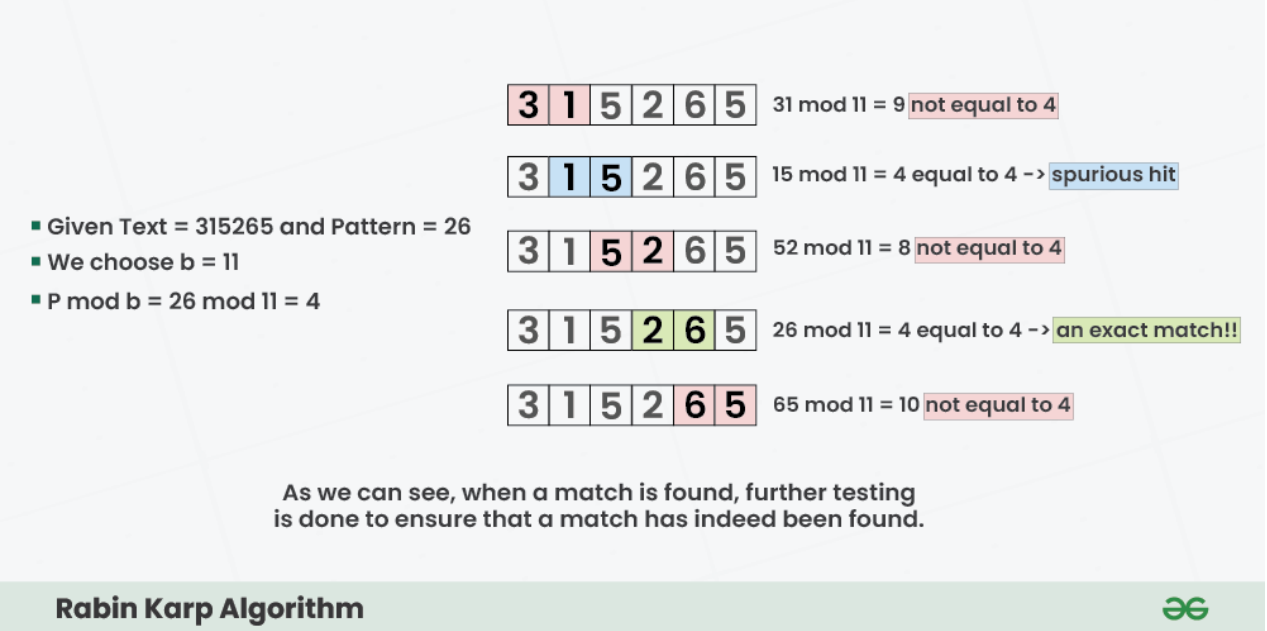
- To slide the **pattern** one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
- The formula for updating the hash value when moving from position **'i'** to **'i+1'** is:

```
hash = (hash - (text[i - pattern_length] * (b^pattern_length - 1)) % p) * b + text[i]
```

Step 6: Compare hash values:

- When the hash value of a substring in the **text** matches the hash value of the **pattern**, it's a **potential match**.
- If the hash values match, we should perform a character-by-character comparison to confirm the match, as hash collisions can occur.

Below is the Illustration of above algorithm:



Step-by-step approach:

- Initially calculate the hash value of the pattern.
- Start iterating from the starting of the string:
    - Calculate the hash value of the current substring having length **m**.
    - If the hash value of the current substring and the pattern are same check if the substring is same as the pattern.
    - If they are same, store the starting index as a valid answer. Otherwise, continue for the next substrings.
- Return the starting indices as the required answer.

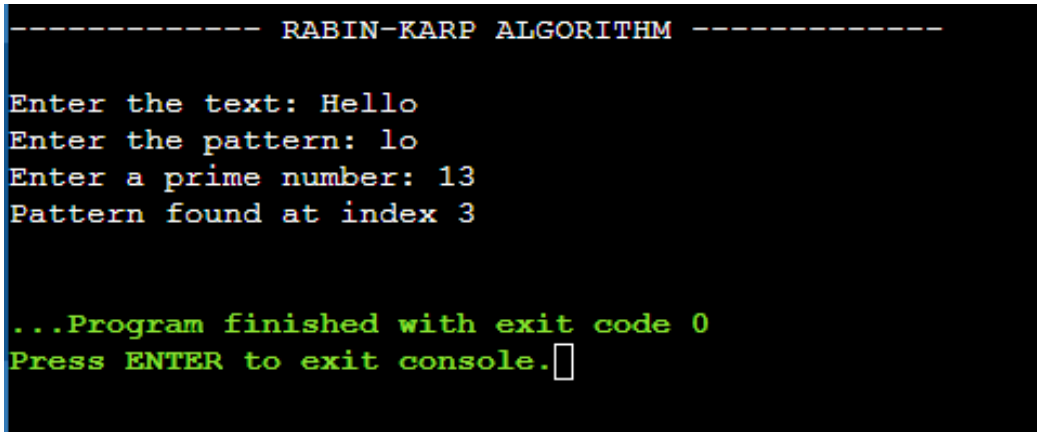
➢ C PROGRAM:

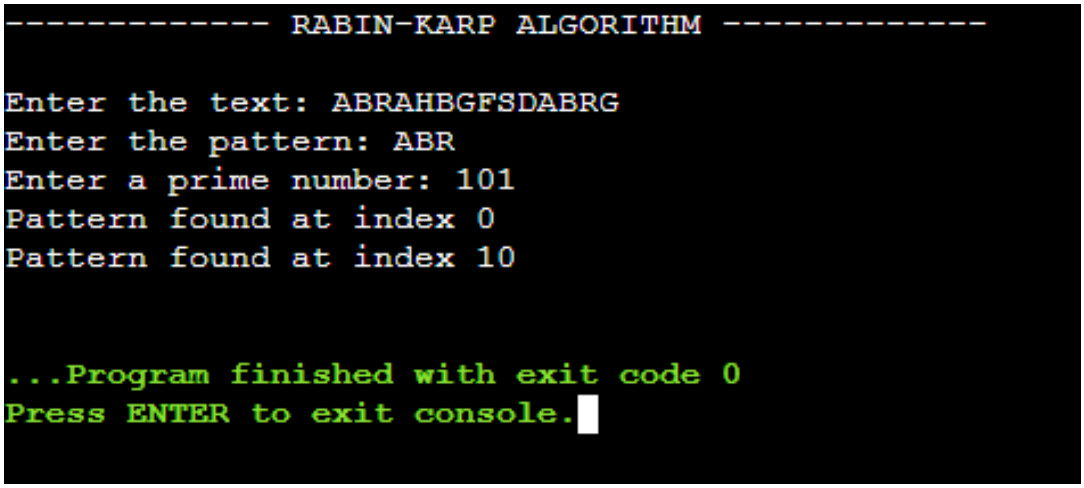
```
#include <stdio.h>
#include <string.h>

#define d 256

void search(char pat[], char txt[], int q)
{
   int M = strlen(pat);
   int N = strlen(txt);
```

```
int i, j;
int p = 0;
int t = 0;
int h = 1;

for (i = 0; i < M - 1; i++)
    h = (h * d) % q;

for (i = 0; i < M; i++) {
    p = (d * p + pat[i]) % q;
    t = (d * t + txt[i]) % q;
}

for (i = 0; i <= N - M; i++) {
    if (p == t) {
        for (j = 0; j < M; j++) {
            if (txt[i + j] != pat[j])
                break;
        }

        if (j == M)
            printf("Pattern found at index %d\n", i);
    }

    if (i < N - M) {
        t = (d * (t - txt[i] * h) + txt[i + M]) % q;

        if (t < 0)
            t = (t + q);
```

```
        }
    }
}

int main()
{
    char txt[100];
    char pat[100];
    int q;

    printf("---------- RABIN-KARP ALGORITHM ----------\n\n");

    printf("Enter the text: ");
    scanf("%s", txt);

    printf("Enter the pattern: ");
    scanf("%s", pat);

    printf("Enter a prime number: ");
    scanf("%d", &q);

    search(pat, txt, q);
    return 0;
}
```

- **OUTPUT:**





- **CONCLUSION: Hence, we have successfully implemented Rabin-Karp Algorithm for Pattern Searching; LO 1, LO 2.**

Meet Raut
S21 84

Assignment 1

Q.1)

a)

Ans $T(n) = 2T(n/2) + n^3$

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

$a = 2, \quad b = 2, \quad f(n) = n^3$

$n^{\log_b a} = n^{\log_2 2} = n \quad < f(n) \ (n^3)$

According to master theorem.

case 3:

If $f(n) = \Omega(n^{\log_b a + \epsilon})$ $(\epsilon > 0)$ & if $af(n/b) \leq kf(n)$

Then $T(n) = \theta(f(n))$

$af(n/b) = 2f(n/2)^3 = n^3/4 \leq kn^3$

This is satisfied for $k = 1/2$

so by master theorem $\theta(n^3) = T(n)$

b)

Ans $T(n) = T(9n/10) + n$

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

$a = 1, \quad b = 10/9, \quad f(n) = n$

$n^{\log_b a} = n^{\log_{10/9} 1} = 1 \quad < f(n) \ (i.e\ n^1)$

If falls in case 3 of master theorem

check if $af(n/b) \leq kf(n)$, where $k < 1$

$1 \times (9/10) \leq k \cdot n \Rightarrow k \geq 9/10$

It satisfies for $k = 19/20$

So by masters theorem $T(n) = \theta(n)$

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

c) $T(n) = 16T(n/4) + n^2$

Ans by Comparing,

$a=16, \quad b=4, \quad f(n) = n^2$

$n^{\log_b a} = n^{\log_4 16} = n^2 = f(n) \quad (n^2)$

According to second case of master theorem

$T(n) = \Theta(n^2 \log n)$

d) $T(n) = 4T(n/2) + n^3 + 4n + 5$

Ans. $a = 4, \quad b = 2, \quad f(n) = n^3 + 4n + 5$

$n^{\log_b a} = n^{\log_2 4} = n^2 < f(n) \quad (\text{i.e. } n^3 + 4n + 5)$

So by case 3 of master theorem

If $af(n/b) \leq kf(n), \quad k < 1$

$4((n/2)^3 + 4(n/2) + 5)$

$\Rightarrow 4(n^3/8 + 2n + 5)$

$\Rightarrow n^3/2 + n/2 + 5/4 \leq k(n^3 + 4n + 5)$

This holds true for $k > 1/8$

So by master theorem, $T(n) = \Theta(n^3 + 4n + 5)$

e)

Ans. $a = 2, \quad b = 2, \quad f(n) = n+1$

$n^{\log_b a} = n^{\log_2 2} = n < f(n) \quad (n+1)$

This falls in case 3

If $af(n/b) \leq k.f(n), \quad k < 1 \; ; \; 2(n/2+1) \leq k(n+1) \Rightarrow n+2 \leq k(n+1)$

This holds for $k = 1/2$

So, by master theorem $T(n) = \Theta(n+1)$

Incorrect correct it

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

Q. 2)

a)

Ans. Using master's th^m

$a = 5, \quad b = 2, \quad k = 0, \quad p = 0$

$n^{\log_b a} = n^{\log_2 5} > f(n)$

This fall in case 1 of master's th^m

$f(n) = O(n^{\log_2 5 - \epsilon}) \quad (\epsilon > 0)$

$\therefore T(n) = \theta(n^{\log_2 5})$

b)

Ans. $a = 1, \quad b = 4, \quad f(n) = n$

$n^{\log_b a} = n^{\log_4 1} = n^0 = 1 < n$ (i.e. $f(n)$

we fall in case 3

$f(n) = O(n^{\log_b a + \epsilon}) \quad (\epsilon > 0)$

if $a f(n/b) < k \cdot f(n) \quad k < 1$

$1 \cdot (n/4) < k(n)$

This holds true for $k = 1/2$

$\therefore T(n) = \theta(f(n)) = \theta(n)$

Q. 3)

Ans. Assume $T(n) = O(n^k)$ for constant k

$T(n) = cT(n/2) + n$

$T(n) = cn^k + n = c(n/2)^k + n = \frac{cn^k}{2^k} + n$

we want to prove, $T(n) = O(n^k)$, let's try to bound

$T(n) \leq \frac{cn^k}{2^k} + n$

let $c/2^k$ be constant c

$T(n) \leq cn^k + n \leq c'n^k$ (say)

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

$k=1 \Longrightarrow T(n) \leq cn + n \leq c'n$

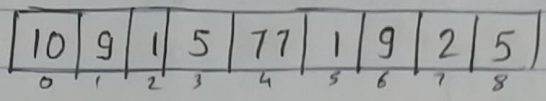
It holds true for $n \geq 1$ if $c' \geq 2c$

Hence, by induction,

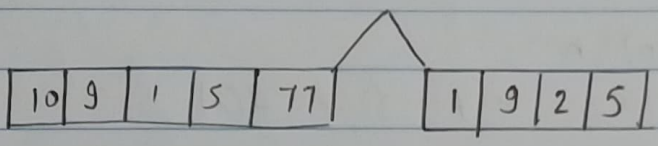
$T(n) = O(n)$

Q.4)

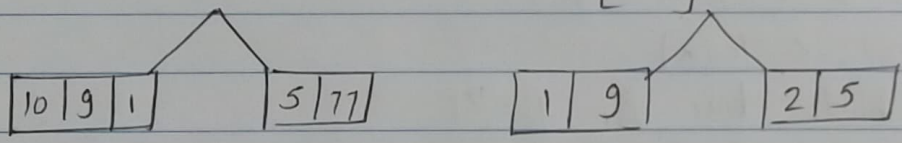
Ans.

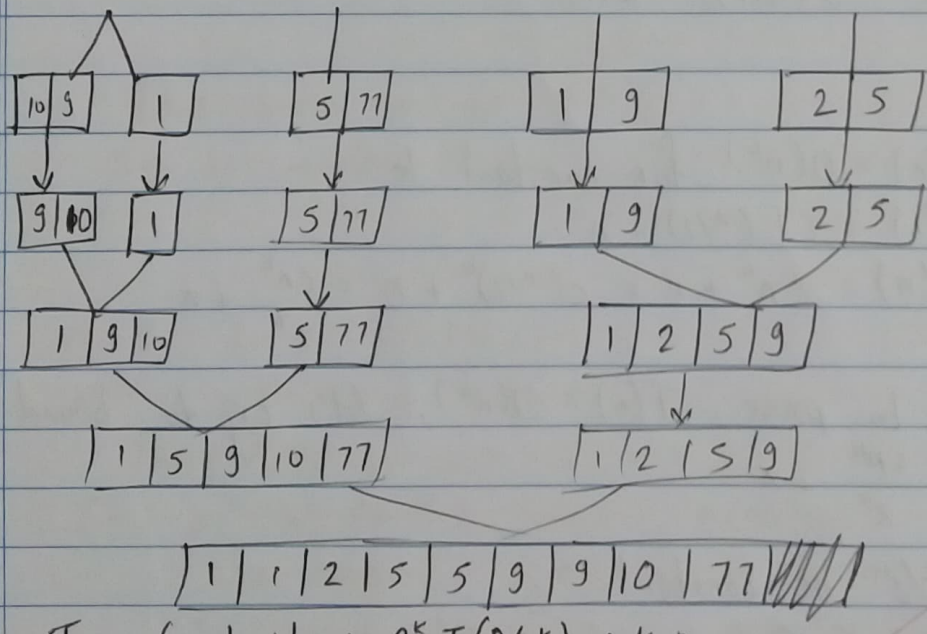


$m = \frac{l+n}{2} = \frac{0+8}{2} = 4$



$m = \frac{0+4}{2} = 2$ $\qquad m = \left\lceil \frac{0+3}{2} \right\rceil = 1$





Time Complexity : $2^k T(n/2^k) + k \cdot n$

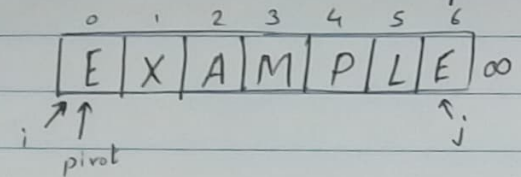
Let $2^k = n \Rightarrow k = \log_2 n$

$n T(n/n) + n \log_2 n \Rightarrow O(n \log_2 n)$

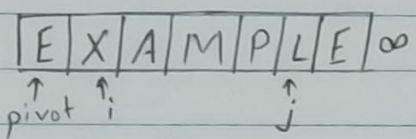
THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

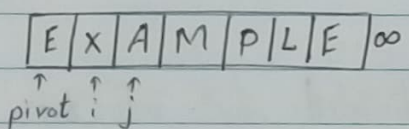
(Q.5)

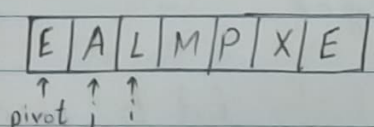
Ans. Let first element be the pivot



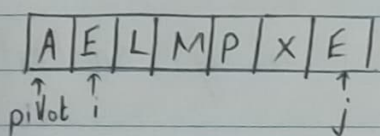
Well increment i till i < pivot and
decrement j till j > pivot and swap i & j.



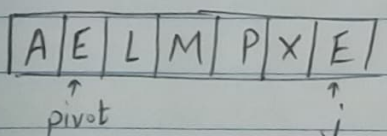




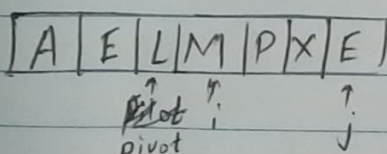
Since, index[j] < index[i], pivot = j



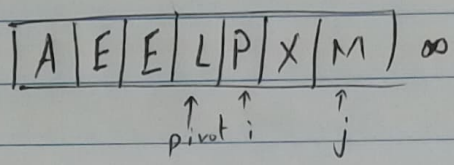
This won't sort, since A is in sorted position



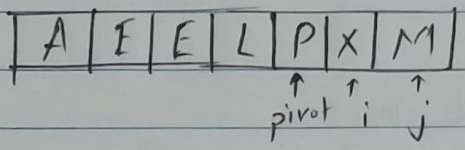
This won't sort, as well since E is sorted too

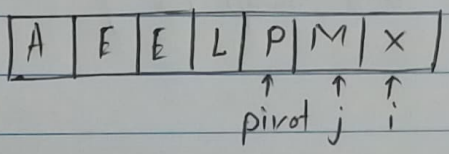


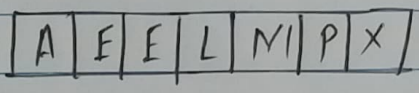
THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE



This won't sort







Best case : Occurs when pivot divides list into two equal sublist of equal sizes. $O(n \log n)$

Avg. case : On avg. pivot nearly divides list to nearly two equal parts $O(n \log n)$

Worst case : This occurs when pivot is always smallest or largest element in list $O(n^2)$

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

Q.6)
Ans.

```
I = 1 ;
While ( I <= n) {
      X = X + I;
      I = I + 1;
}
```

Frequency Count :
Initialisation for I = 1
Increment for X and I combined = 2n
Loop Run = n

Time complexity :
Initialization = O(1)
While loop = O(n)
Increment for X & I combined = O(n)
Overall = O(1) + O(n) + O(n) = O(2n+1) = O(n)

Q.7)
Ans. i) Substitution method :
It involves guessing a solution and then solving by using mathematical induction.

ii) Recurrence tree method :
It involes representing recurrence of a tree where each node represents cost of single subproblem

iii) Master theorem :
It involes a straight forward way to solve recurrence of the form $T(n) = aT(n/b) + f(n)$

It's a powerful and widely used method to solve recurrence problem

iv) Iteration method:

This method involves expanding the recurrence into iterations until a pattern emerges.

v) Generating functions:

This are used to represents sequence of number as power series. Recurrences can be translated into equations involving generation functions, which can be solved using algebric techniques

Q.8)

Ans.

P - Type	NP - Type
→ Set of decision problems that can be solved using turing machine	→ Set of decision problems for which a solution can be guessed efficiently.
→ Running Time of an algo is bounded by polynomial function size	→ If someone claims to be solution to NP problem, it can be verified quickly
→ Effectively soluable as running time grosses polynomially with input.	→ A solution in NP can be guessed efficiently but maybe difficult for traditional
→ Ex. Sorting, searching, basic arithmetic operations	→ Ex. Travelling salesman problem, knapsack problem

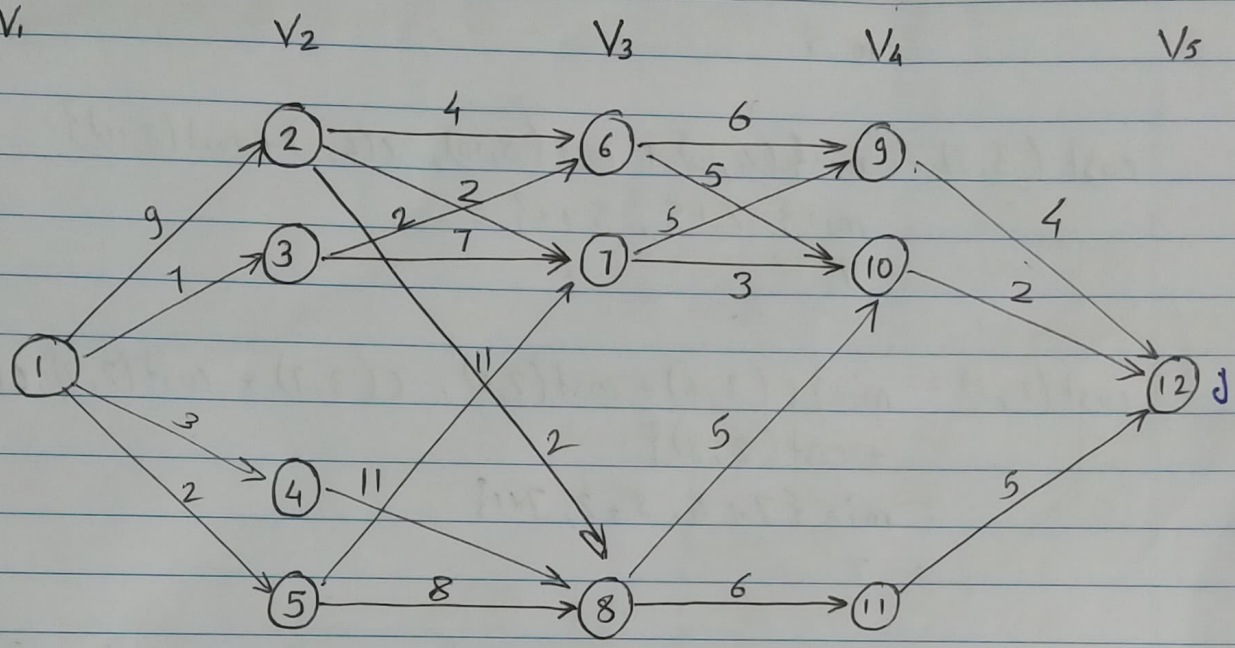
Meet Raut.
S21 84

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Assignment 2

Q.1)

Ans.



Ans. Formula : $cost(i, j) = \min\{c(j, l) + cost(i+1, l)\}$

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	12

$cost(3,6) = \min\{(4,9) + cost(6,9), c(4,10) + cost(6,10)\}$

$= \min\{4+6, 2+5\}$

$= \min\{10, 7\}$

$= 7$

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

$cost(3,7) = \min\{c(4,9) + cost(7,9), c(4,10) + cost(7,10)\}$
$= \min\{4+4, 2+3\}$
$= 5$

$cost(3,8) = \min\{c(4,10) + cost(8,10), c(4,11) + cost(8,11)\}$
$= \min\{2+5, 5+6\}$
$= 7$

$cost(2,2) = \min\{c(3,6) + cost(2,6), c(3,7) + cost(2,7), c(3,8) + cost(2,8)\}$
$= \min\{7+4, 5+2, 7+1\}$
$= 7$

$cost(2,3) = \min\{c(3,6) + cost(3,6), c(3,7) + cost(3,7)\}$
$= \min\{7+2, 5+1\}$
$= 9$

$cost(2,4) = \min\{c(3,8) + cost(4,8)\}$
$= \min\{7+11\}$
$= 18$

$cost(2,5) = \min\{c(3,7) + cost(5,7), c(3,8) + cost(5,8)\}$
$= \min\{5+11, 7+8\}$
$= 15$

$cost(1,1) = \min\{c(2,2) + cost(2,2), c(2,3) + cost(3,3), c(2,4) + cost(4,4), c(2,4) + cost(4,4), c(2,5) + cost(5,5)\}$
$= \min\{7+9, 9+7, 18+3, 15+2\}$
$= 16$

Shortest Paths: $1 \to 2 \to 7 \to 10 \to 12$ & $1 \to 3 \to 6 \to 10 \to 12$

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

Q.2)

Ans. $M = 35$

$W = (5, 7, 10, 12, 15, 18, 20)$

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

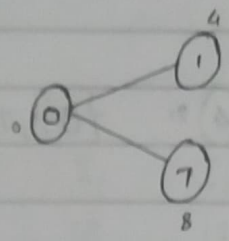
At each node end, 'x' means it cannot accommodate any of next value, so we cut-sort them for further expansion and 'v' means it is the solution states.

We get four solutions

1 ⟶ {5, 10, 20}
2 → {5, 12, 18}
3 → {7, 10, 18}
4 → {15, 20}

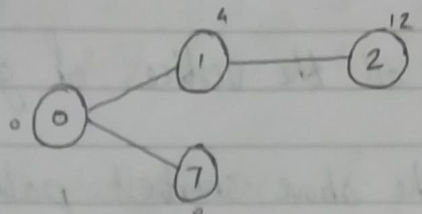
Q.3)

Ans. Step 1:



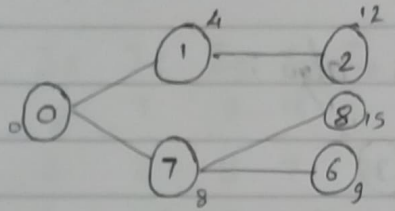
{0}

Step 2:



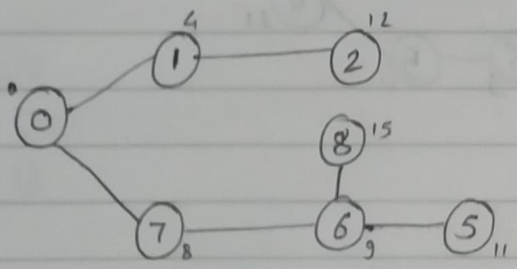
{0, 1}

Step 3:



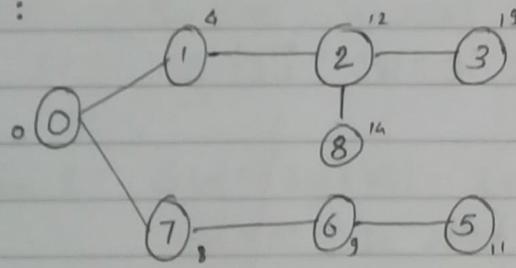
{0, 1, 7}

Step 4:



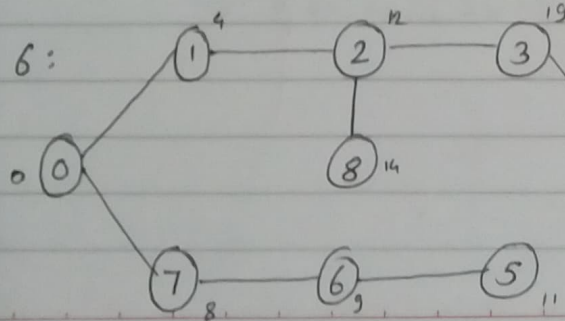
{0, 1, 7, 6}

Step 5:



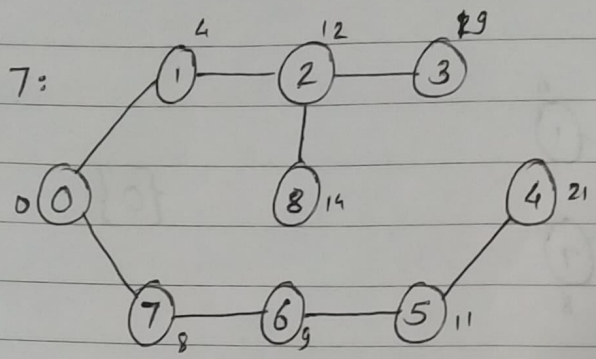
{0, 1, 7, 6, 2}

Step 6:



{0, 1, 7, 6, 2, 8, 3}

Step 7:

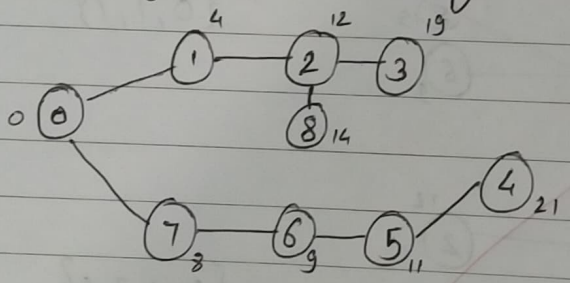


{0, 1, 7, 6, 8, 3, 5, 4}

Spt set includes all the vertices of given graph.

∴ Finally we get the above shortest path tree.

SPT using Djikstra's algo:



Q.4)

Ans. Initially :

Item (x_i)	I_1	I_5	I_4	I_3	I_6	I_2
Profit (v_i)	18	12	10	9	7	5
Weight (w_i)	7	3	5	3	2	2
$P_i = v_i/w_i$	2.57	4	2	3	3.5	2.5

$\therefore$ decreasing order of profit/Weight value table.

x_i	I_5	I_6	I_3	I_1	I_2	I_4
v_i	12	7	9	18	5	10
w_i	3	2	3	7	2	5
P_i	4	3.5	3	2.57	2.5	2

$SW = 10$

$SP = 0$

$M = 13$

Iteration 1 : $SW = (SW + W_5) = 0 + 3 = 3$

$SW \leq M$, So select I_5

$\therefore S = \{I_5\}$; $SW = 3$; $SP = 0 + 12 = 19$

Iteration 2 : $SW = (SW + W_6) = 3 + 2 = 5$

$SW \leq M$; So select I_6

$S = \{I_5, I_6\}$; $SW = 5$; $SP = 12 + 7 = 19$

Iteration 3 : $SW = (SW + W_3) : 5 + 3 = 8$

$SW \leq M$, so selet I_3

$S = \{I_5, I_6, I_3\}$; $SW = 8$; $SP = 19 + 9 = 28$

Iteration 4 : $SW + W_1 > M$, so break down item I_1.

The remaining capacity of knapsack is 5, so select only 5 item of I_1.

$$frac = ((M - SW)/W[i]) : \frac{13 - 8}{7} = \frac{5}{7}$$

$$S = \{I_5, I_6, I_3, I_1 * 5/7\}$$

$$SP = SP + V_i * 5/7 = 28 + (18 * 5/7) = 28 + 12.857$$

$$SP = 40.85$$

$$SW = SW + W_i * 5/7 = 8 + (7 * 5/7) = 8 + 5 = 13$$

∴ knapsack is full.

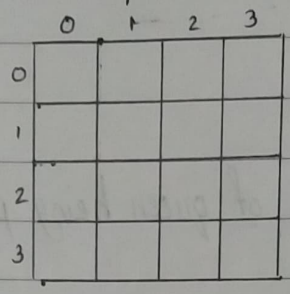
∴ Fractional knapsack select items $\{I_5, I_6, I_3, I_1 * 5/7\}$ and it gives total profit of 40.85 units.

Q.5)

Ans. Algorithm :

```
Algorith N-Queen(k, n)
// Input : n = no. of queen, k = No. of queen being processed currently
// Output : n x 1 solution tuple.

  for i ← 1 to n do
     if PLACE(k, i) then
        x[k] ← i
        if k == n then
           print x[1...N]
        else
           N-Queen(k+1, n)
        end
     end
  end

function Place(k, i)
   for j ← 1 to k-1 do
      if x[j] == i OR ((abs(x[j] - i) == abs(j-k)) then
         return false
      end
   end
return true
```

4-Queen problem:



4×4 chessboard

putting Q_1: It can be placed anywhere.

	0	1	2	3
0	Q_1	x	x	x
1	x	x		
2	x		x	
3	x			x

putting Q_2: Placing at the non-attacked places.

	0	1	2	3
0	Q_1	x	x	x
1	x	x	x	Q_2
2	x		x	x
3	x	x		x

putting Q_3: placing at the non-attacked places.

	0	1	2	3
0	Q_1	x	x	x
1	x	x	x	Q_2
2	x	Q_3	x	x
3	x	x	x	x

There is no place for Q_4.

By making some adjustment we get,

	0	1	2	3
0	X	Q_1	X	X
1	X	X	X	Q_2
2	Q_3	X	X	X
3	X	X	Q_4	X

Therefore, through backtracking we reached a solution where 4 queens are put in each row & column so that no queen is attacking any other on a 4×4 chessboard.

The two solution for n=4 queens: (1,3,0,2) & (2,0,3,1)

Q.6)

Ans. X = ababcde

Y = bacadb

If $x[i] == y[j]$

$c[i][j] = 1 + [i-1, j-1]$

else

$c[i][j] = Max\{c[i-1, j], c[i, j-1]\}$

Y →

X ↓	-	∧	b	a	c	a	d	b
	∧	0	0	0	0	0	0	0
	a	0	← 0	↖ 1	← 1	↖ 1	← 1	← 1
	b	0	↖ 1	← 1	← 1	← 1	← 1	↖ 2
	a	0	↑ 1	↖ 2	← 2	↖ 2	← 2	← 2
	b	0	↖ 1	↑ 2	← 2	← 2	← 2	↖ 3
	c	0	↑ 1	↑ 2	↖ 3	← 3	← 3	← 3
	d	0	↑ 1	↑ 2	↑ 3	← 3	↖ 4	← 4
	e	0	↑ 1	↑ 2	↑ 3	← 3	↑ 4	← 4

LCS = dcab

Q.7)
Ans.

Prim's algo	Kruskal's algo
1. Vertex based algorithm	1. Edge-based algorithm.
2. Time complexity is $O(V^2)$	2. Time complexity is $O(E+V\log V)$
3. It tends to perform better on dense graphs	3. It tends to perform better on sparse graphs.
4. It commonly uses a priority queue data structure to effectively select the next edge to add.	4. It typically uses disjoint set data structures to efficiently check for cycles and maintain connected components.