Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 1:

Aim: To study and implement Selection Sort and Insertion Sort

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]
```

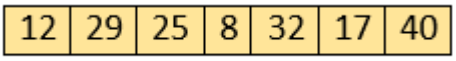
Step 4: EXIT

```
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

To understand the working of the Selection sort algorithm, let's take an unsorted array.

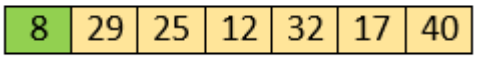
Let the elements of array are -



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

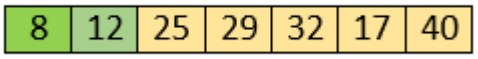
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

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

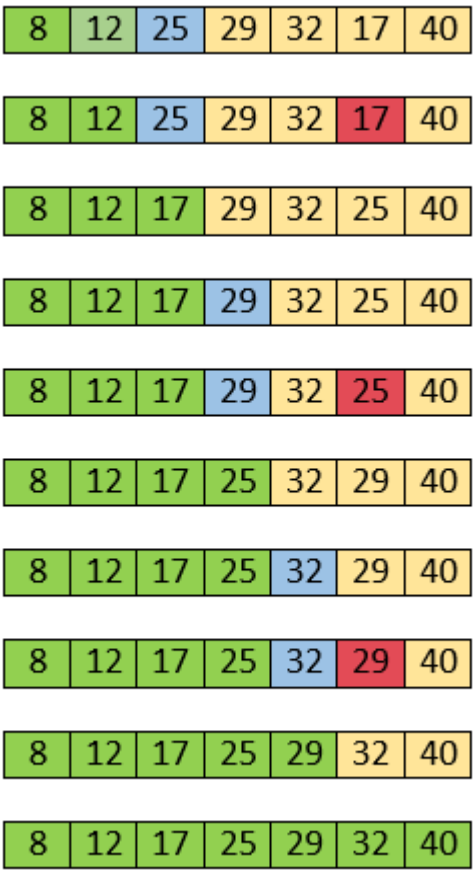


For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

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

1. Time Complexity

Best Case $O(n^2)$

Average Case	$O(n^2)$
Worst Case	$O(n^2)$

2. Space Complexity

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

Code:

```
#include <stdio.h>

#include <conio.h>


void swap(int *n1,int *n2)

{

	int temp=*n1;

	*n1=*n2;

	*n2=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]);
        }
```

```
        printf("\n");
}

void main()
{
        int arr[]={64, 25, 12, 22, 11};
        int n=sizeof(arr)/sizeof(arr[0]);
        clrscr();
        selSort(arr,n);
        printArr(arr,n);
        getche();
}
```

Output:

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.

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.

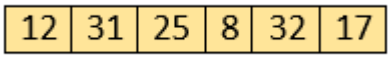
Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

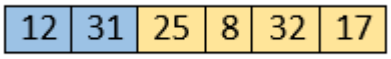
Working of Insertion sort Algorithm

To understand the working of the insertion sort algorithm, let's take an unsorted array.

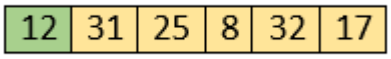
Let the elements of array are -



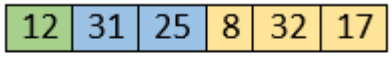
Initially, the first two elements are compared in insertion sort.

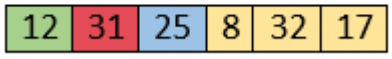


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



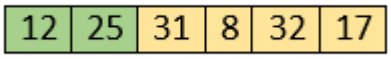
Now, move to the next two elements and compare them.



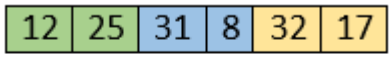


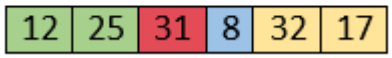
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

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.

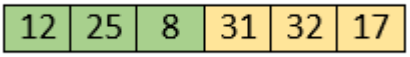


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

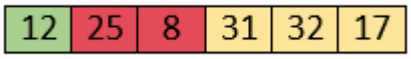




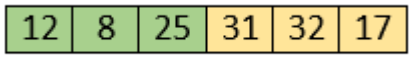
Both 31 and 8 are not sorted. So, swap them.



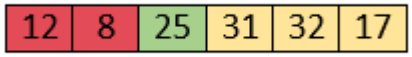
After swapping, elements 25 and 8 are unsorted.



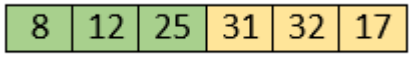
So, swap them.



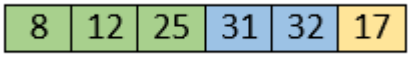
Now, elements 12 and 8 are unsorted.



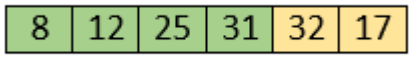
So, swap them too.



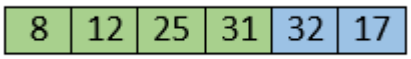
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



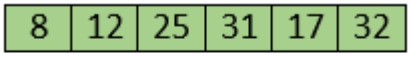
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

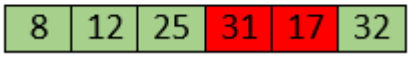


Move to the next elements that are 32 and 17.

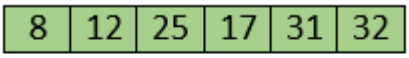


17 is smaller than 32. So, swap them.

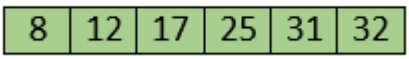




Swapping makes 31 and 17 unsorted. So, swap them too.



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.

1. Time Complexity

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

2. Space Complexity

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

Code:

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

```
#include<conio.h>

void insSort(int arr[],int n)
{
	int i,j,key;
	for(i=1;i<n;i++)
	{
		key=arr[i];
		j=i-1;
		while(j>=0 && arr[j]>key)
		{
			arr[j+1]=arr[j];
			j-=1;
		}
		arr[j+1]=key;
	}
}

void printArr(int arr[],int n)
{
	int i;
```

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

void main()
{
        int arr[]={12,11,13,5,6};
        int n=sizeof(arr)/sizeof(arr[0]);
        clrscr();
        insSort(arr,n);
        printArr(arr,n);
        getche();
}
```

Output:

```
5 6 11 12 13
```

Conclusion:

Thus we have successfully implemented Selection Sort and Insertion Sort

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 2:

Aim: To study and implement Merge Sort and Quick Sort

Quick Sort Algorithm

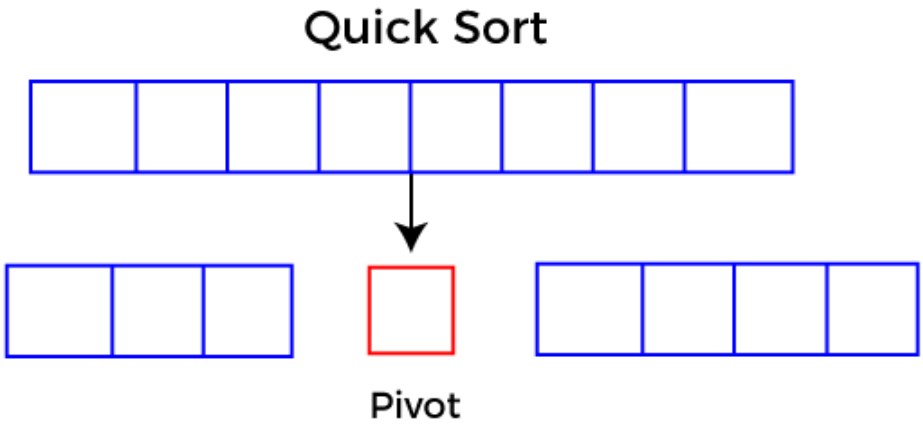
Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms 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.



Algorithm

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)
 }
}
```

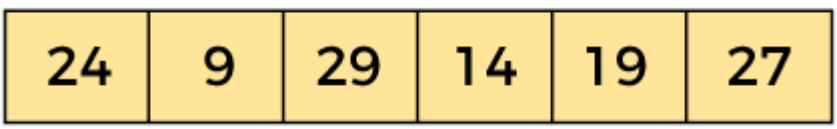
Partition Algorithm:

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
}
```

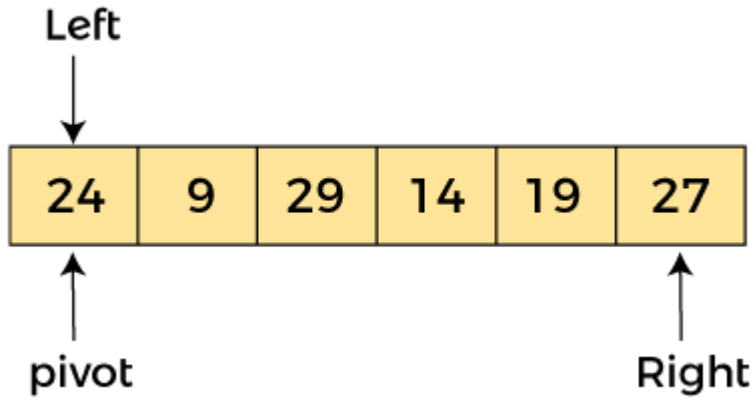
Working of Quick Sort Algorithm

Let the elements of array are -

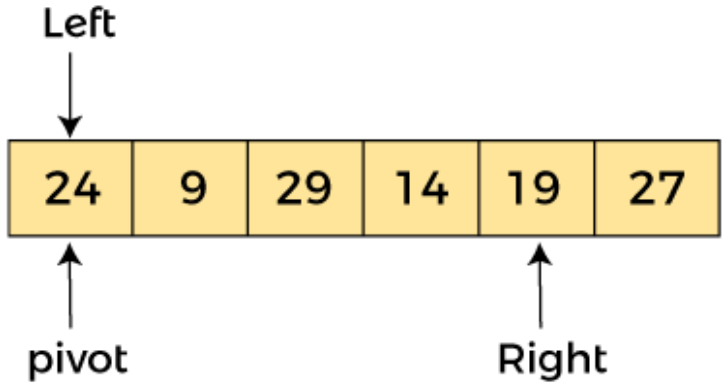


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

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

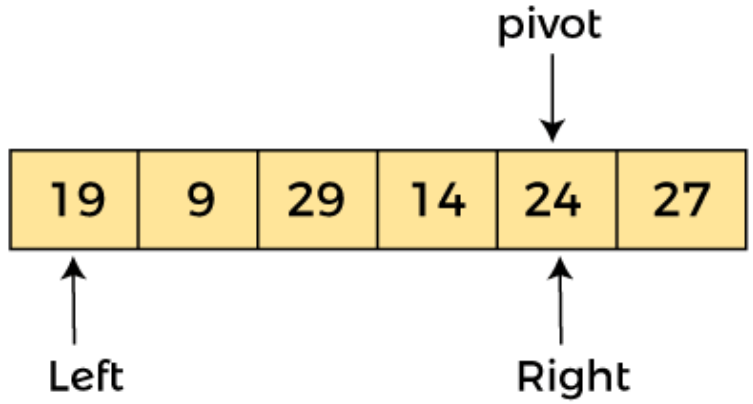


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



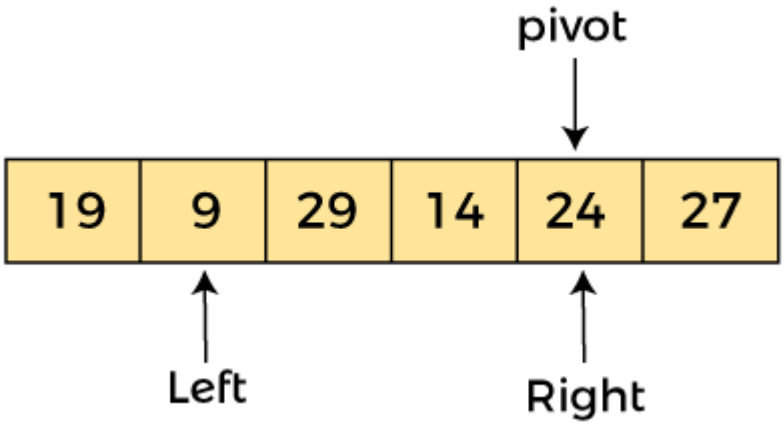
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

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

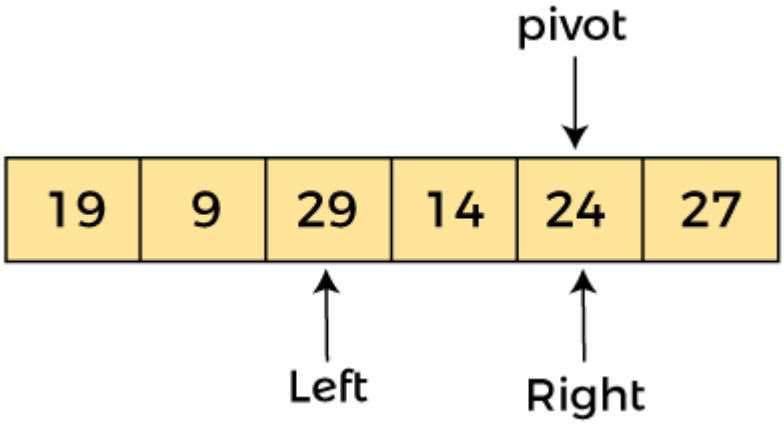


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

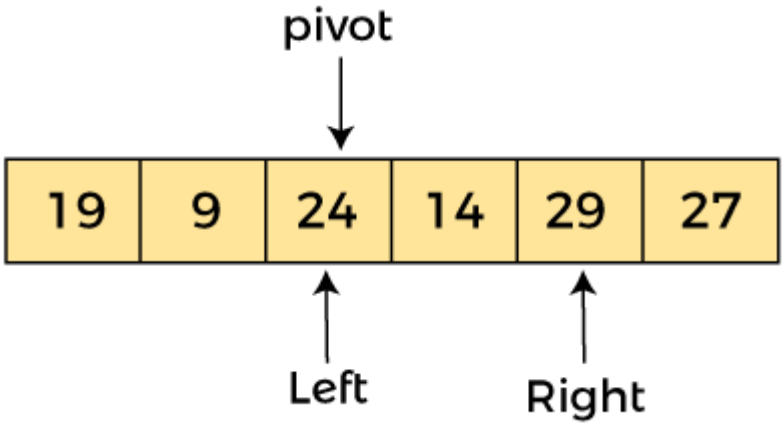
As a[pivot] > a[left], so algorithm moves one position to right as -



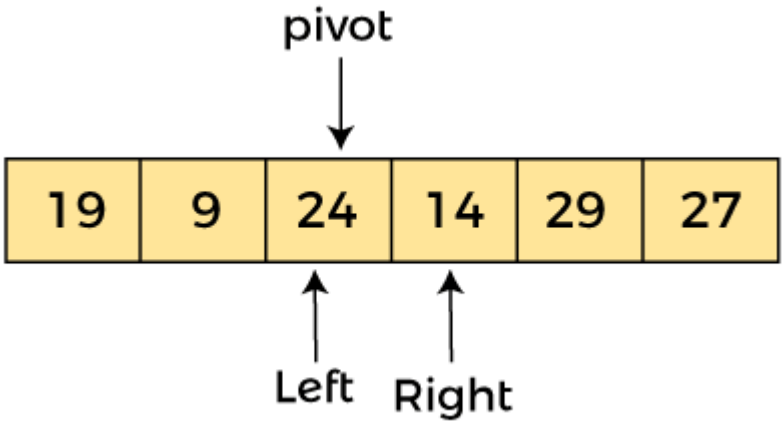
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



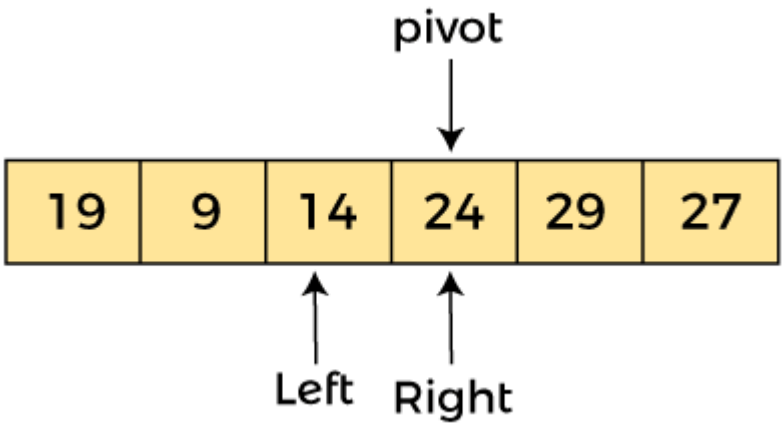
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



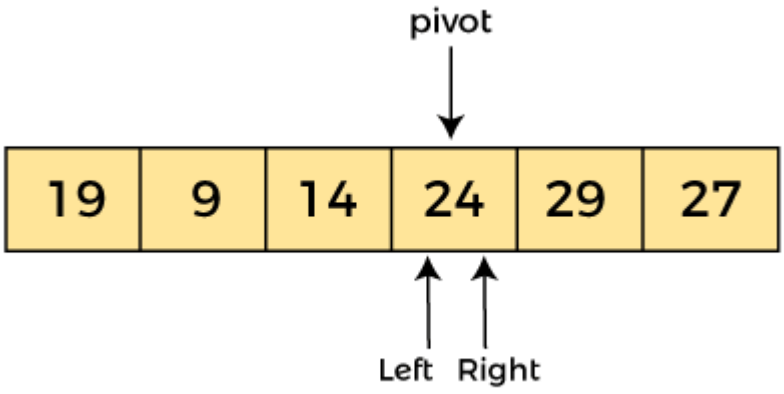
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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. -



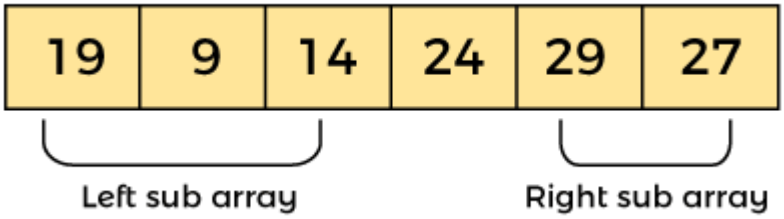
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



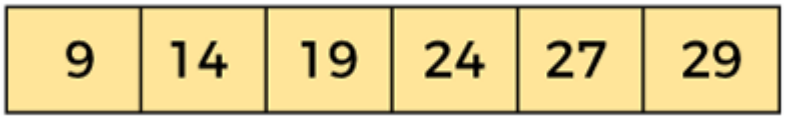
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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 **$O(n^2)$**.

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(
Stable	NO

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

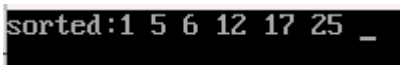
Code:

```
#include<stdio.h>

#include<conio.h>



void swap(int *n1,int *n2)

{

    int t=*n1;

    *n1=*n2;

    *n2=t;
```

```
}

int partition(int arr[],int low, int high)
{
	int j;
	int pivot=arr[high];
	int i=low-1;
	for(j=low;j<=high-1;j++)
	{
		if(arr[j]<pivot)
		{
			i++;
			swap(&arr[i],&arr[j]);
		}
	}
	swap(&arr[i+1],&arr[high]);
	return (i+1);
}

void quicksort(int arr[],int low,int high)
{
```

```
        if(low<high)
        {
                int pi=partition(arr,low,high);
                quicksort(arr,low,pi-1);
                quicksort(arr,pi+1,high);
        }
}

void main()
{
        int arr[]={12,17,6,25,1,5};
        int I,n=sizeof(arr)/sizeof(arr[0]);
        clrscr();
        quicksort(arr,0,n-1);
        printf("sorted:");
        for(i=0;i<n;i++)
                printf("%d ",arr[i]);
        getche();
}
```

Output:



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.

Algorithm

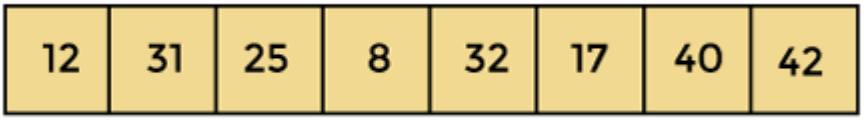
In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
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

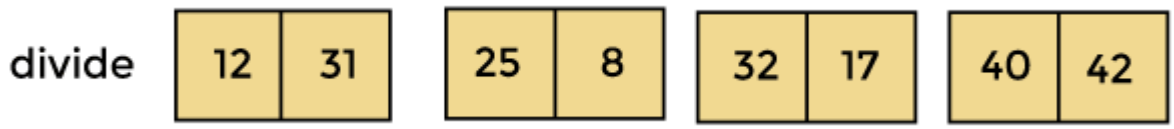
Let the elements of array are -



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.

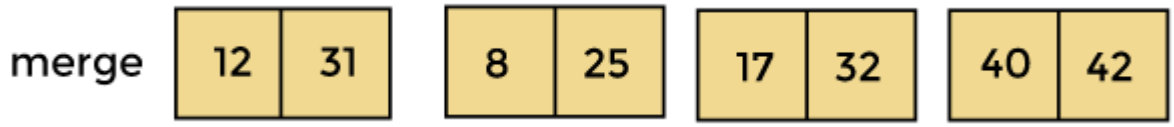


Now, again divide these arrays to get the atomic value that cannot be further divided.

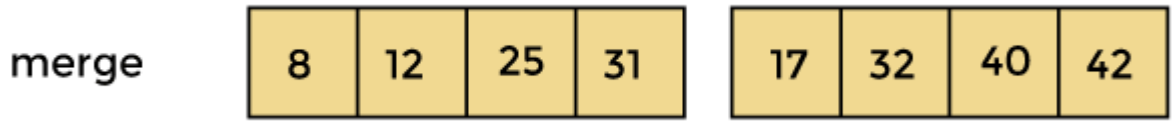
Now, combine them in the same manner they were broken.

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

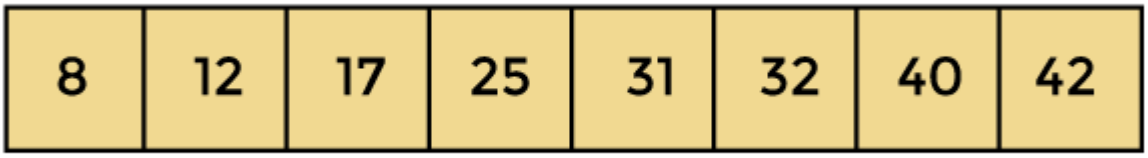
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



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.

Merge sort complexity

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)**.

2. Space Complexity

Space Complexity	**O(n)**
Stable	

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

Code:

```
#include<stdlib.h>

#include<stdio.h>


void merge(int arr[], int l, int m, int r);


void mergeSort(int arr[], int l, int r)

{

  if (l < r)

  {

    int m = l+(r-l)/2;

    mergeSort(arr, l, m);

    mergeSort(arr, m+1, r);

    merge(arr, l, m, r);

  }

}
```

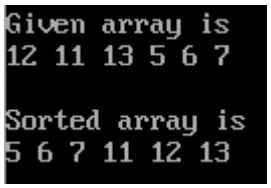
```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    int L[100], R[100];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
```

```
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}

while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
while (j < n2)
{
    arr[k] = R[j];
    j++;
```

```
        k++;
    }
}

void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);
```

```
    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Output:



Conclusion:

Thus, we have successfully implemented Merge Sort and Quick Sort

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 3:

Aim: To study and implement Binary Search Algorithm

Binary Search Algorithm

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element. Otherwise, the search is called unsuccessful.

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

NOTE: Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.

Algorithm

```
Binary_Search(a, lower_bound, upper_bound, val)
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
set pos = mid
print pos
go to step 6
else if a[mid] > val
set end = mid - 1
else
set beg = mid + 1
```

[end of if]
[end of loop]
Step 5: if pos = -1
print "value is not present in the array"
[end of if]
Step 6: exit

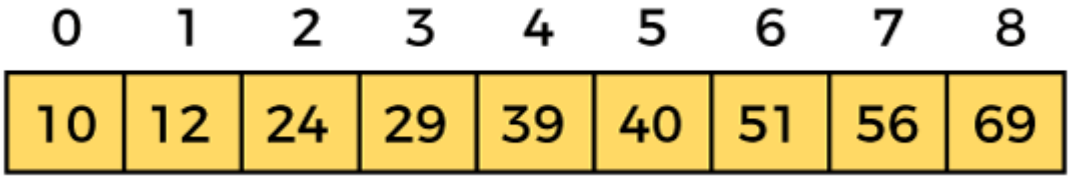
Working of Binary search

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -



Let the element to search is, K = 56

We have to use the below formula to calculate the mid of the array -

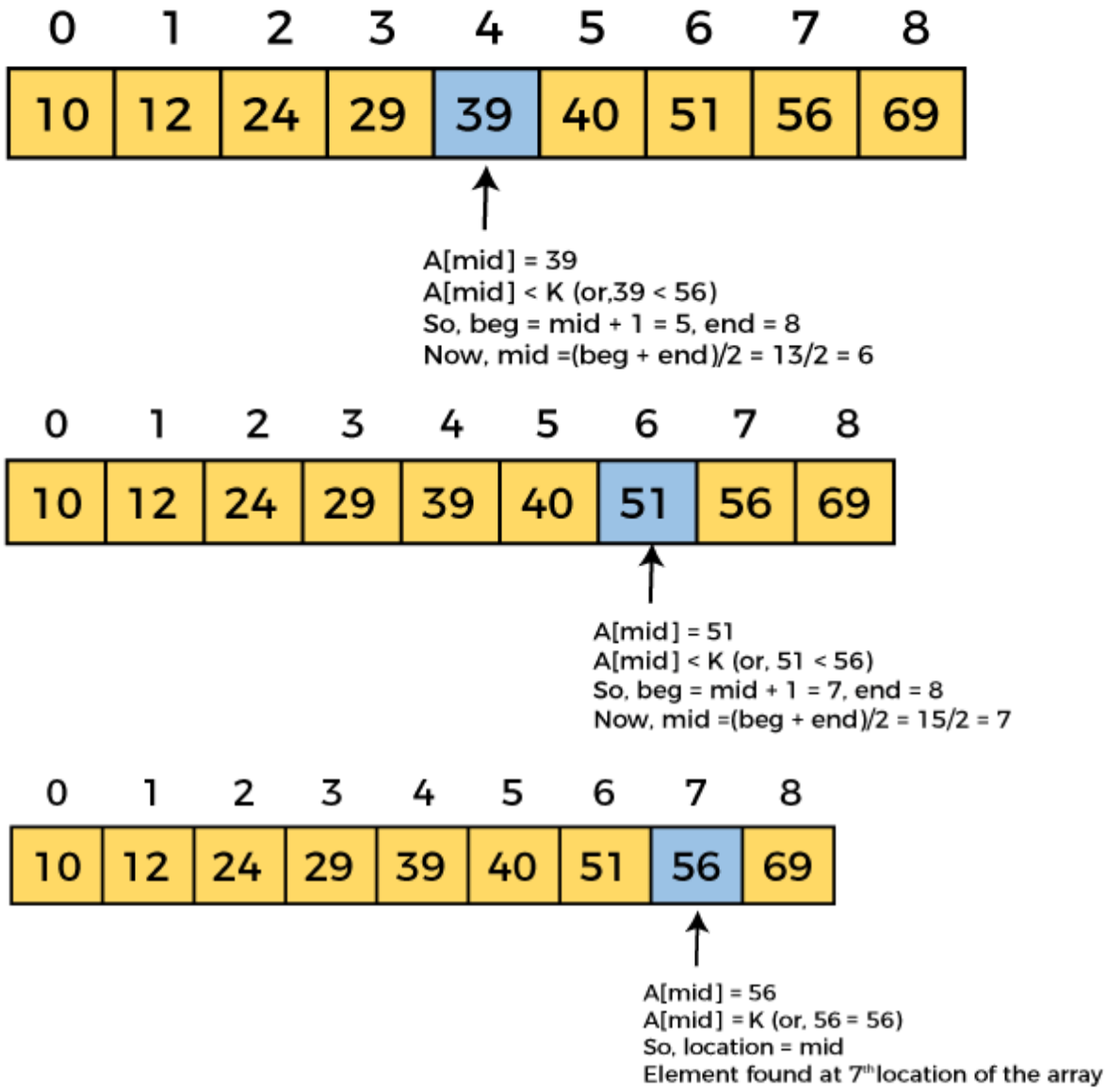
1. mid = (beg + end)/2

beg = 0

end = 8

mid = (0 + 8)/2 = 4.

So, 4 is the mid of the array.



Now, algorithm will return the index of the element matched.

Binary Search complexity

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

- Best Case Complexity - In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is O(1).
- Average Case Complexity - The average case time complexity of Binary search is O(logn).
- Worst Case Complexity - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is O(logn).

2. Space Complexity
   - The space complexity of binary search is O(1).

Code:

```
#include<stdio.h>

#include<conio.h>


int binSearch(int arr[],int low,int high,int key)

{

    if(high>=low)

    {

        int mid=(low+high)/2;

        if(arr[mid]==key)

        {

            return mid;

        }
```

```
            if(arr[mid]>key)
            {
                return binSearch(arr,low,mid-1,key);
            }
            return binSearch(arr,mid+1,high,key);
        }
        return -1;
}

void main()
{
        int arr[]={2,3,4,10,40};
        int n=sizeof(arr)/sizeof(arr[0]);
        int key=10;
        int res,i;
        clrscr();
        res=binSearch(arr,0,n-1,key);
        printf("array = ");
        for(i=0;i<n;i++)
        {
```

```
        printf("%d ",arr[i]);
    }
    (res==-1) ? printf("\nnot present") : printf("\npresent at pos %d",res+1);
    getche();
}
```

Output:

```
array = 2 3 4 10 40
present at pos 4
```

Conclusion:

Hence, we have successfully implemented binary search algorithm.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 4:

Aim: Implement the shortest path using Djikstra's algorithm

Dijkstra's Algorithm:

Dijkstra's algorithm is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956.

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

Need for Dijkstra's Algorithm (Purpose and Use-Cases)

The need for Dijkstra's algorithm arises in many applications where finding the shortest path between two points is crucial.

For example, It can be used in the routing protocols for computer networks and also used by map systems to find the shortest path between starting point and the Destination (as explained in How does Google Maps work?)

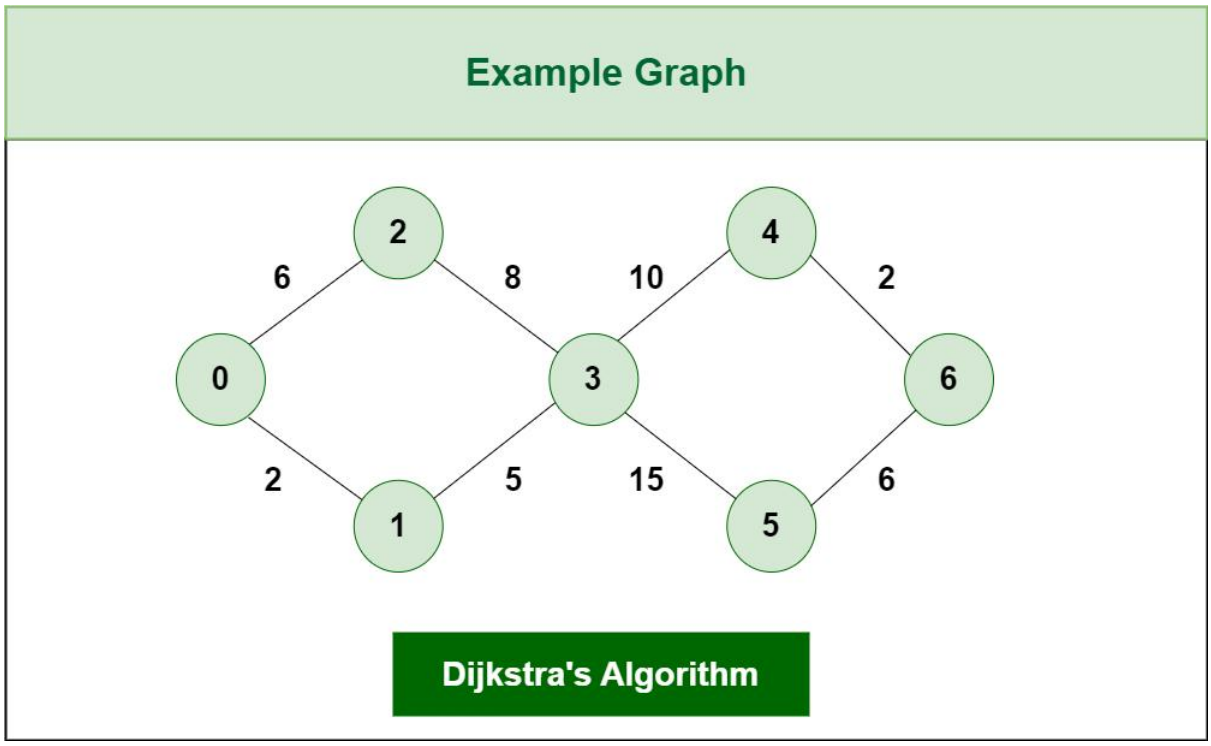
Can Dijkstra's Algorithm work on both Directed and Undirected graphs?

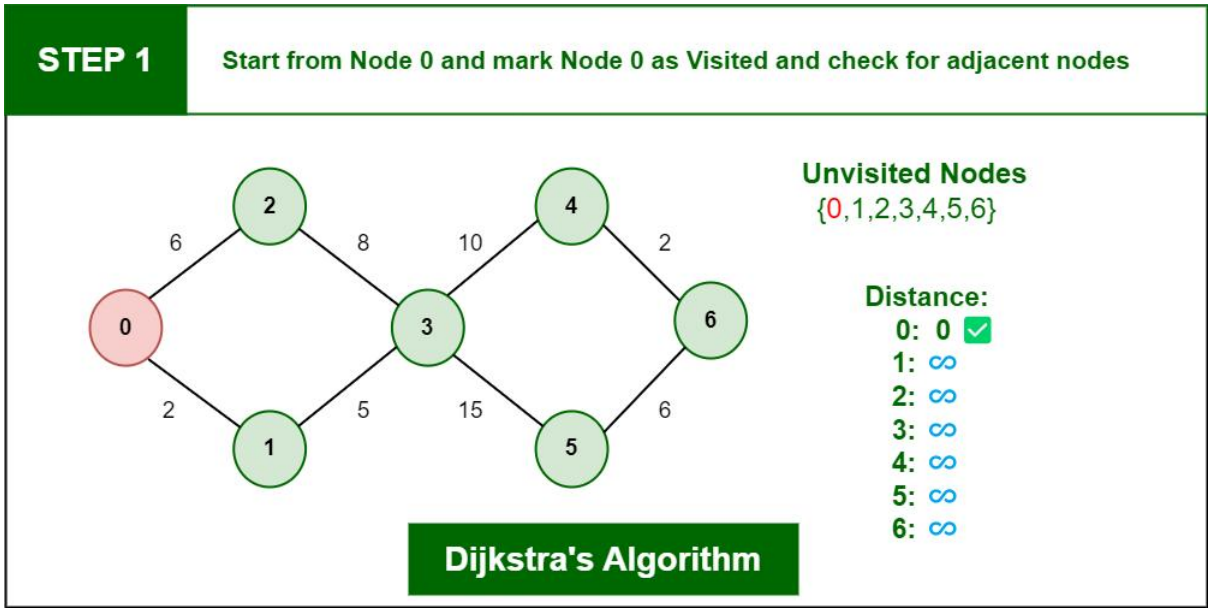
Yes, Dijkstra's algorithm can work on both directed graphs and undirected graphs as this algorithm is designed to work on any type of graph as long as it meets the requirements of having non-negative edge weights and being connected.

In a directed graph, each edge has a direction, indicating the direction of travel between the vertices connected by the edge. In this case, the algorithm follows the direction of the edges when searching for the shortest path.

In an undirected graph, the edges have no direction, and the algorithm can traverse both forward and backward along the edges when searching for the shortest path.

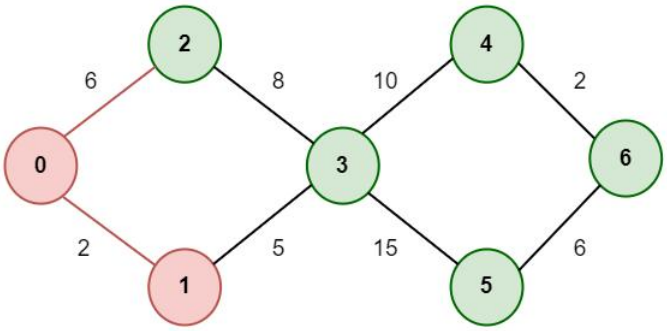
How does Dijkstra’s Algorithm works?





STEP 2

Mark Node 1 as Visited and add the Distance



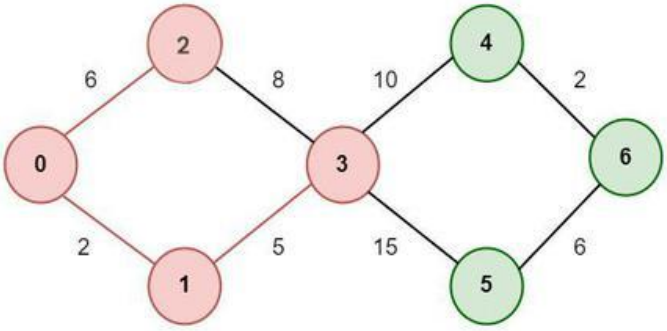
Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✅
1: 2 ✅
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

Dijkstra's Algorithm

STEP 3

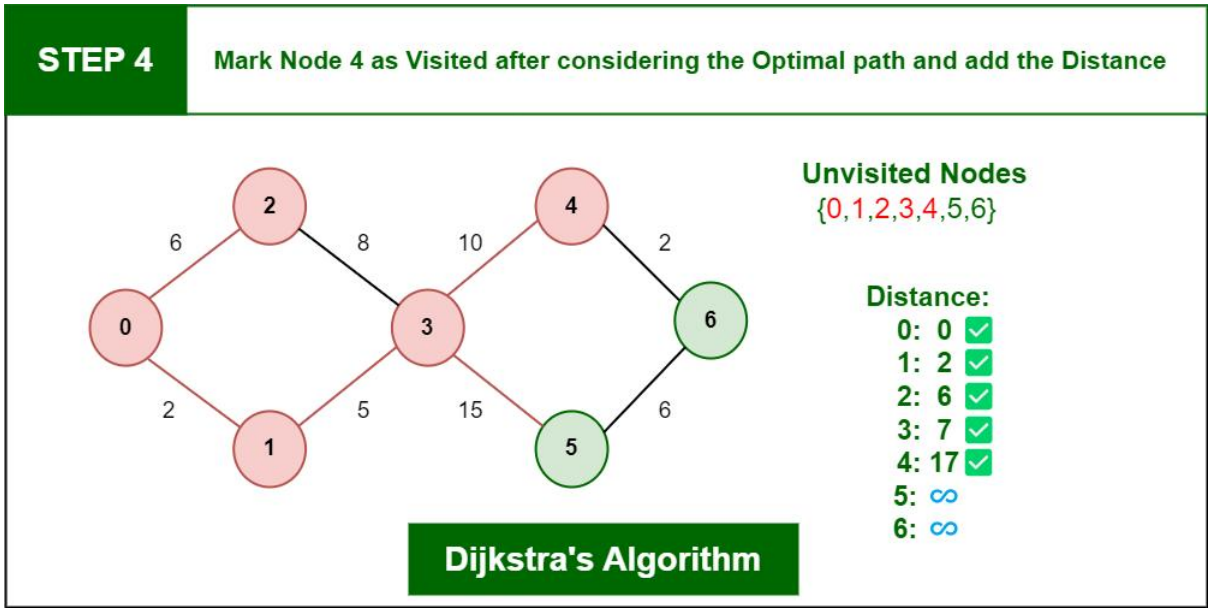
Mark Node 3 as Visited after considering the Optimal path and add the Distance

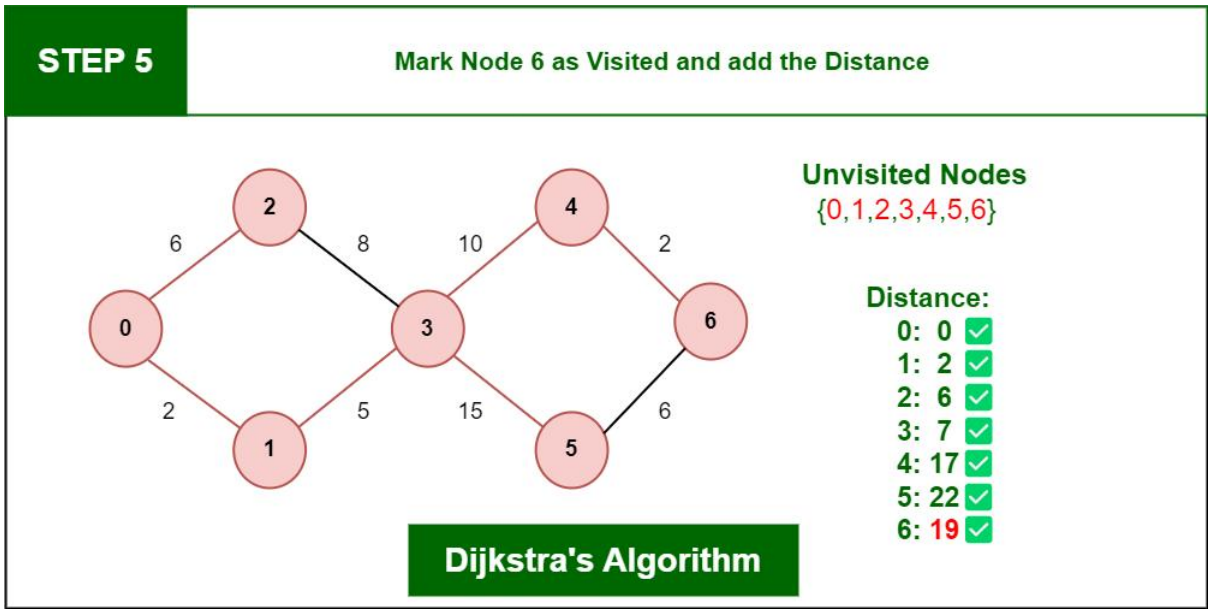


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✅
1: 2 ✅
2: 6 ✅
3: 7 ✅
4: ∞
5: ∞
6: ∞

Dijkstra's Algorithm





Pseudo Code for Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
  // Initialize distances to all nodes as infinity, except for the source node.
  distances = map infinity to all nodes
  distances = 0

  // Initialize an empty set of visited nodes and a priority queue to keep track of the nodes to visit.
```

```
  visited = empty set
  queue = new PriorityQueue()
  queue.enqueue(source, 0)

  // Loop until all nodes have been visited.
  while queue is not empty:
    // Dequeue the node with the smallest distance from the priority queue.
    current = queue.dequeue()

    // If the node has already been visited, skip it.
    if current in visited:
      continue

    // Mark the node as visited.
    visited.add(current)

    // Check all neighboring nodes to see if their distances need to be updated.
    for neighbor in Graph.neighbors(current):
      // Calculate the tentative distance to the neighbor through the current node.
      tentative_distance = distances[current] + Graph.distance(current, neighbor)

      // If the tentative distance is smaller than the current distance to the neighbor, update the distance.
      if tentative_distance < distances[neighbor]:
        distances[neighbor] = tentative_distance
```

// Enqueue the neighbor with its new distance to be considered for visitation in the future.

queue.enqueue(neighbor, distances[neighbor])

// Return the calculated distances from the source to all other nodes in the graph.

return distances

Code:

```
#include <limits.h>

#include <stdbool.h>

#include <stdio.h>


// Number of vertices in the graph

#define V 9


// A utility function to find the vertex with minimum

// distance value, from the set of vertices not yet included

// in shortest path tree

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
// array
void printSolution(int dist[]) {
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t\t\t %d\n", i, dist[i]);
}


// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src) {
    int dist[V]; // The output array. dist[i] will hold the shortest distance from src to i
    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest path tree or shortest distance from src to i is finalized
```

```
// Initialize all distances as INFINITE and sptSet[] as false
for (int i = 0; i < V; i++)
    dist[i] = INT_MAX, sptSet[i] = false;

// Distance of source vertex from itself is always 0
dist[src] = 0;

// Find shortest path for all vertices
for (int count = 0; count < V - 1; count++) {
    // Pick the minimum distance vertex from the set of
    // vertices not yet processed. u is always equal to
    // src in the first iteration.
    int u = minDistance(dist, sptSet);

    // Mark the picked vertex as processed
    sptSet[u] = true;

    // Update dist value of the adjacent vertices of the
    // picked vertex.
    for (int v = 0; v < V; v++)
        // Update dist[v] only if is not in sptSet,
```

```
            // there is an edge from u to v, and total

            // weight of path from src to v through u is

            // smaller than current value of dist[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:

```
Vertex      Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14
```

Conclusion:

We've understand that with Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can find the shortest path from a node (called the "source node") to all other nodes in the graph, producing a shortest-path tree.

This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, specially in domains that require modeling networks.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 5:

Aim: To study and implement Kruskal/Prims algorithm(using greedy approach)

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:

1. Kruskal

```
#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
```

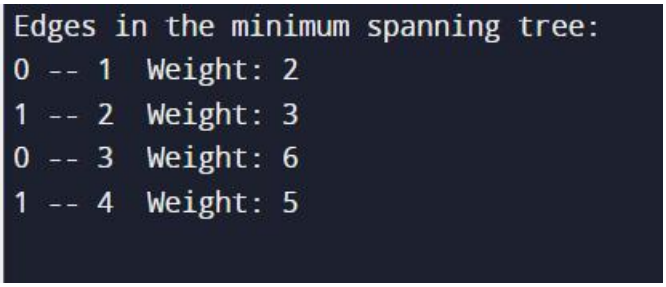
2. Prims

```
#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:



Conclusion:

Thus we have successfully implemented the Kruskal/Prims algorithm.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 6:

Aim: To implement Job sequencing with deadlines using Greedy.

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

```
Following is maximum profit sequence of jobs
c a e
```

Conclusion:

Thus we have successfully implemented Job sequencing with deadlines using Greedy.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 7:

Aim: To study and implement all pair shortest path problem using Floyd Warshall's algorithm.

Floyd Warshall Algorithm:

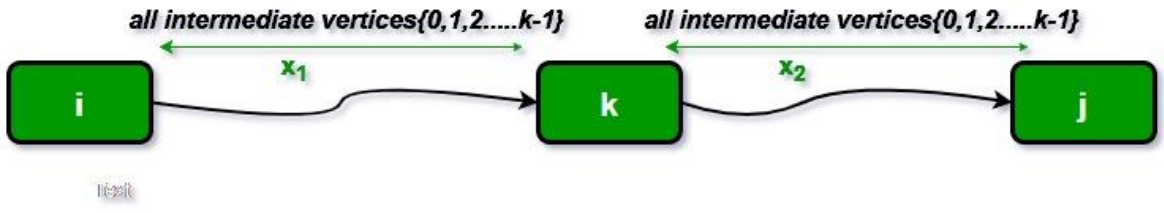
The Floyd-Warshall algorithm, named after its creators Robert Floyd and Stephen Warshall, is a fundamental algorithm in computer science and graph theory. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. 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). It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Idea Behind Floyd Warshall Algortihm:

Suppose we have a graph G[][] with V vertices from 1 to N. Now we have to evaluate a shortestPathMatrix[][] where shortestPathMatrix[i][j] represents the shortest path between vertices i and j.
Obviously the shortest path between i to j will have some k number of intermediate nodes. The idea behind floyd warshall algorithm is to treat each and every vertex from 1 to N as an intermediate node one by one.

The following figure shows the above optimal substructure property in floyd warshall algorithm:



Floyd Warshall Algorithm:

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
- For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
    - k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
    - k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j], if dist[i][j] > dist[i][k] + dist[k][j]

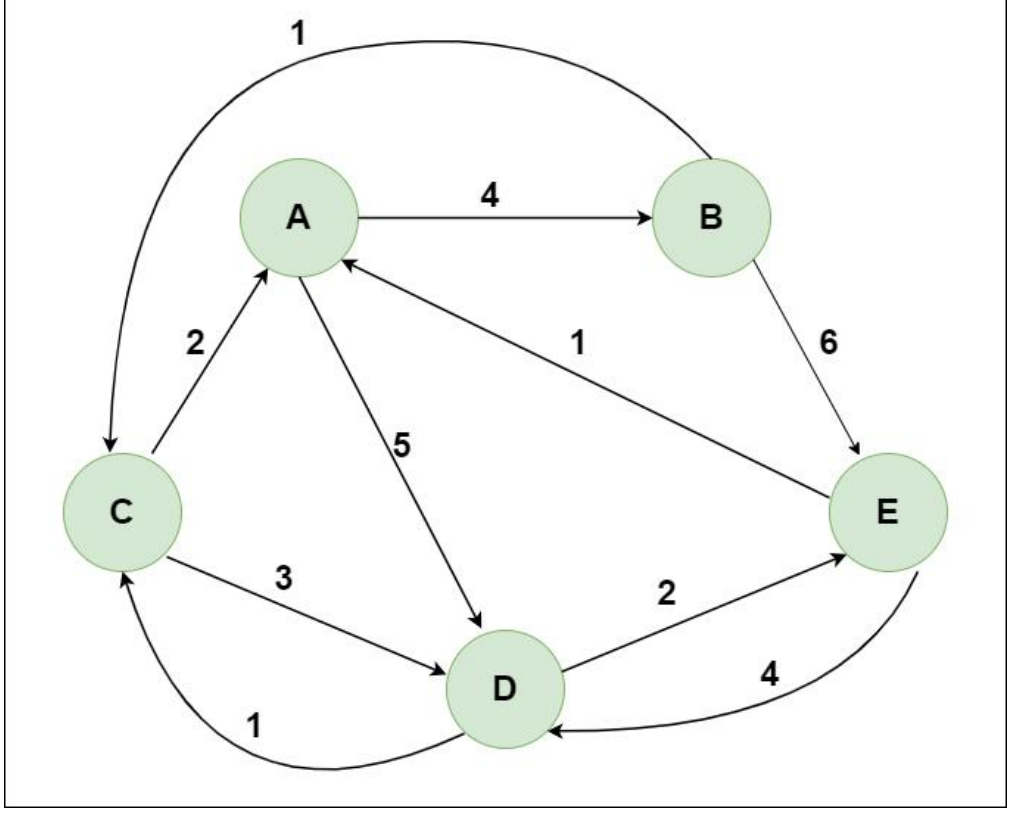
Pseudo-Code of Floyd Warshall Algorithm :

```
For k = 0 to n – 1
For i = 0 to n – 1
For j = 0 to n – 1
Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])

where i = source Node, j = Destination Node, k = Intermediate Node
```

Illustration of Floyd Warshall Algorithm :

Suppose we have a graph as shown in the image:

Example Graph



Step 1: Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from i to j, also Distance[i][j] = Infinity if there is no edge from i to j.

Step1: Initializing Distance[ ][ ] using the Input Graph

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Step 2: Treat node A as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to A) + (Distance from A to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][A] + Distance[A][j])

Step 2: Using Node A as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

→

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

Step 3: Treat node B as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to B) + (Distance from B to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][B] + Distance[B][j])

Step 3: Using Node B as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

Step 4: Treat node C as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to C) + (Distance from C to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][C] + Distance[C][j])

Step 4: Using Node C as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

Step 5: Treat node D as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to D) + (Distance from D to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][D] + Distance[D][j])

Step 5: Using Node D as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 6: Treat node E as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to E) + (Distance from E to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][E] + Distance[E][j])

Step 6: Using Node E as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 7: Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.

Step 7: Return Distance[ ][ ] matrix as the result

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

- Time Complexity: O(V3), where V is the number of vertices in the graph and we run three nested loops each of size V
- Auxiliary Space: O(V2), to create a 2-D matrix in order to store the shortest distance for each pair of nodes.

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");
        }
}

// driver's code
int main()
{
        printf("------------ Floyd Warshall's Algorithm ------------\n\n");
        int graph[V][V] = { { 0, 3, INF, 7 },
                                        { 8, 0, 2, INF },
                                        { 5, INF, 0, 1 },
                                        { 2, INF, INF, 0 } };

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

Output:

```
------------ Floyd Warshall's Algorithm ------------

The following matrix shows the shortest distances between every pair of
    vertices
      0      3      5      6
      5      0      2      3
      3      6      0      1
      2      5      7      0
```

Conclusion:

Thus we have successfully implemented Floyd Warshall's algorithm.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 8:

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

0/1 Knapsack problem

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

What is the 0/1 knapsack problem?

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.

What is the fractional knapsack problem?

The fractional knapsack problem means that we can divide the item. For example, we have an item of 3 kg then we can pick the item of 2 kg and leave the item of 1 kg. The fractional knapsack problem is solved by the Greedy approach.

Example of 0/1 knapsack problem.

Consider the problem having weights and profits are:

Weights: {3, 4, 6, 5}

Profits: {2, 3, 1, 4}

The weight of the knapsack is 8 kg

The number of items is 4

The above problem can be solved by using the following method:

x_i = {1, 0, 0, 1}

= {0, 0, 0, 1}

= {0, 1, 0, 1}

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:

2^4 = 16; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach. In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.

How this problem can be solved by using the Dynamic programming approach?

First,

we create a matrix shown as below:

	0	1	2	3	4	5
0						
1						
2						
3						
4						

In the above matrix, columns represent the weight, i.e., 8. The rows represent the profits and weights of items. Here we have not taken the weight 8 directly, problem is divided into sub-problems, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8. The solution of the sub-problems would be saved in the cells and answer to the problem would be stored in the final cell. First, we write the weights in the ascending order and profits according to their weights shown as below:

w_i = {3, 4, 5, 6}

p_i = {2, 3, 4, 1}

The first row and the first column would be 0 as there is no item for w=0

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

When i=1, W=1

w_1 = 3; Since we have only one item in the set having weight 3, but the capacity of the knapsack is 1. We cannot fill the item of 3kg in the knapsack of capacity 1 kg so add 0 at M[1][1] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

When i = 1, W = 2

w_1 = 3; Since we have only one item in the set having weight 3, but the capacity of the knapsack is 2. We cannot fill the item of 3kg in the knapsack of capacity 2 kg so add 0 at M[1][2] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0			
2	0					

3	0					
4	0					

When i=1, W=3

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is also 3; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][3] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2		
2	0					
3	0					
4	0					

When i=1, W = 4

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 4; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][4] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	
2	0					
3	0					
4	0					

When i=1, W = 5

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 5; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][5] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0					
3	0					
4	0					

When i =1, W=6

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 6; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][6] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0					
3	0					
4	0					

When i=1, W = 7

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 7; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][7] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0					

3	0					
4	0					

When i =1, W =8

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 8; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][8] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0					
3	0					
4	0					

Now the value of 'i' gets incremented, and becomes 2.

When i =2, W = 1

The weight corresponding to the value 2 is 4, i.e., w_2 = 4. Since we have only one item in the set having weight equal to 4, and the weight of the knapsack is 1. We cannot put the item of weight 4 in a knapsack, so we add 0 at M[2][1] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0				
3	0					
4	0					

When i =2, W = 2

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have only one item in the set having weight equal to 4, and the weight of the knapsack is 2. We cannot put the item of weight 4 in a knapsack, so we add 0 at M[2][2] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0			
3	0					
4	0					

When i =2, W = 3

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 3. We can put the item of weight 3 in a knapsack, so we add 2 at M[2][3] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2		
3	0					
4	0					

When i =2, W = 4

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 4. We can put item of weight 4 in a knapsack as the profit corresponding to weight 4 is more than the item having weight 3, so we add 3 at M[2][4] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0

1	0	0	0	2	2	2
2	0	0	0	2	3	
3	0					
4	0					

When i = 2, W = 5

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 5. We can put item of weight 4 in a knapsack and the profit corresponding to weight is 3, so we add 3 at M[2][5] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0					
4	0					

When i = 2, W = 6

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 6. We can put item of weight 4 in a knapsack and the profit corresponding to weight is 3, so we add 3 at M[2][6] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0					

4	0					

When i = 2, W = 7

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 7. We can put item of weight 4 and 3 in a knapsack and the profits corresponding to weights are 2 and 3; therefore, the total profit is 5, so we add 5 at M[2][7] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	0	3	3
3	0					
4	0					

When i = 2, W = 8

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 7. We can put item of weight 4 and 3 in a knapsack and the profits corresponding to weights are 2 and 3; therefore, the total profit is 5, so we add 5 at M[2][7] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0					
4	0					

Now the value of 'i' gets incremented, and becomes 3.

When i = 3, W = 1

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set having weights 3, 4, and 5, and the weight of the knapsack is 1. We cannot put neither of the items in a knapsack, so we add 0 at M[3][1] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0				
4	0					

When i = 3, W = 2

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set having weight 3, 4, and 5, and the weight of the knapsack is 1. We cannot put neither of the items in a knapsack, so we add 0 at M[3][2] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0			
4	0					

When i = 3, W = 3

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively and weight of the knapsack is 3. The item with a weight 3 can be put in the knapsack and the profit corresponding to the item is 2, so we add 2 at M[3][3] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3

3	0	0	0	2		
4	0					

When i = 3, W = 4

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 4. We can keep the item of either weight 3 or 4; the profit (3) corresponding to the weight 4 is more than the profit corresponding to the weight 3 so we add 3 at M[3][4] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	
4	0					

When i = 3, W = 5

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 5. We can keep the item of either weight 3, 4 or 5; the profit (3) corresponding to the weight 4 is more than the profits corresponding to the weight 3 and 5 so we add 3 at M[3][5] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0					

When i =3, W = 6

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 6. We can keep the item of either weight 3, 4 or 5; the profit (3) corresponding to the weight 4 is more than the profits corresponding to the weight 3 and 5 so we add 3 at M[3][6] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0					

When i =3, W = 7

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 7. In this case, we can keep both the items of weight 3 and 4, the sum of the profit would be equal to (2 + 3), i.e., 5, so we add 5 at M[3][7] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0					

When i = 3, W = 8

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and the weight of the knapsack is 8. In this case, we can keep both the items of weight 3 and 4, the sum of the profit would be equal to (2 + 3), i.e., 5, so we add 5 at M[3][8] shown as below:

	0	1	2	3	4	5

0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0					

Now the value of 'i' gets incremented and becomes 4.

When i = 4, W = 1

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 1. The weight of all the items is more than the weight of the knapsack, so we cannot add any item in the knapsack; Therefore, we add 0 at M[4][1] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0	0				

When i = 4, W = 2

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 2. The weight of all the items is more than the weight of the knapsack, so we cannot add any item in the knapsack; Therefore, we add 0 at M[4][2] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3

3	0	0	0	1	3	3
4	0	0	0			

When i = 4, W = 3

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 3. The item with a weight 3 can be put in the knapsack and the profit corresponding to the weight 4 is 2, so we will add 2 at M[4][3] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0	0	0	2		

When i = 4, W = 4

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 4. The item with a weight 4 can be put in the knapsack and the profit corresponding to the weight 4 is 3, so we will add 3 at M[4][4] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0	0	0	2	3	

When i = 4, W = 5

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 5. The

item with a weight 4 can be put in the knapsack and the profit corresponding to the weight 4 is 3, so we will add 3 at M[4][5] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0	0	0	2	3	3

When i = 4, W = 6

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 6. In this case, we can put the items in the knapsack either of weight 3, 4, 5 or 6 but the profit, i.e., 4 corresponding to the weight 6 is highest among all the items; therefore, we add 4 at M[4][6] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	1	3	3
4	0	0	0	2	3	3

When i = 4, W = 7

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 7. Here, if we add two items of weights 3 and 4 then it will produce the maximum profit, i.e., (2 + 3) equals to 5, so we add 5 at M[4][7] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0

1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	2	3	3
4	0	0	0	2	3	3

When i = 4, W = 8

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 8. Here, if we add two items of weights 3 and 4 then it will produce the maximum profit, i.e., (2 + 3) equals to 5, so we add 5 at M[4][8] shown as below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	2	3	3
4	0	0	0	2	3	3

As we can observe in the above table that 5 is the maximum profit among all the entries. The pointer points to the last row and the last column having 5 value. Now we will compare 5 value with the previous row; if the previous row, i.e., i = 3 contains the same value 5 then the pointer will shift upwards. Since the previous row contains the value 5 so the pointer will be shifted upwards as shown in the below table:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	2	3	3

4	0	0	0	2	3	3

Again, we will compare the value 5 from the above row, i.e., i = 2. Since the above row contains the value 5 so the pointer will again be shifted upwards as shown in the below table:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	2	3	3
4	0	0	0	2	3	3

Again, we will compare the value 5 from the above row, i.e., i = 1. Since the above row does not contain the same value so we will consider the row i=1, and the weight corresponding to the row is 4. Therefore, we have selected the weight 4 and we have rejected the weights 5 and 6 shown below:

x = { 1, 0, 0}

The profit corresponding to the weight is 3. Therefore, the remaining profit is (5 - 3) equals to 2. Now we will compare this value 2 with the row i = 2. Since the row (i = 1) contains the value 2; therefore, the pointer shifted upwards shown below:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	2	3	3
4	0	0	0	2	3	3

Again we compare the value 2 with a above row, i.e., i = 1. Since the row i =0 does not contain the value 2, so row i = 1 will be selected and the weight corresponding to the i = 1 is 3 shown below:

X = {1, 1, 0, 0}

The profit corresponding to the weight is 2. Therefore, the remaining profit is 0. We compare 0 value with the above row. Since the above row contains a 0 value but the profit corresponding to this row is 0. In this problem, two weights are selected, i.e., 3 and 4 to maximize the profit.

Code:

```
#include<stdio.h>

#include<conio.h>


int max(int a,int b)

{

        return((a>b)?a:b);

}


int ks(int W,int wt[], int val[],int n)

{

        int i,w;

        int K[100][100];


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

        {

                for(w=0;w<=W;w++)

                {

                        if(i==0 || w==0)
```

```
                K[i][w]=0;
            else if(wt[i-1]<=w)
                K[i][w]=max(val[i-1]+K[i-1][w-wt[i-1]],K[i-1][w]);
            else
                K[i][w]=K[i-1][w];
        }
    }
    return K[n][W];
}

void main()
{
    int profit[]={60,100,120};
    int wt[]={10,20,30};
    int W=50;
    int n=sizeof(profit)/sizeof(profit[0]);
    printf("max profit:%d",ks(W,wt,profit,n));
    getche();
}
```

Output:

```
max profit:220_
```

Conclusion:

Hence, we have successfully implemented knapsack algorithm.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 9:

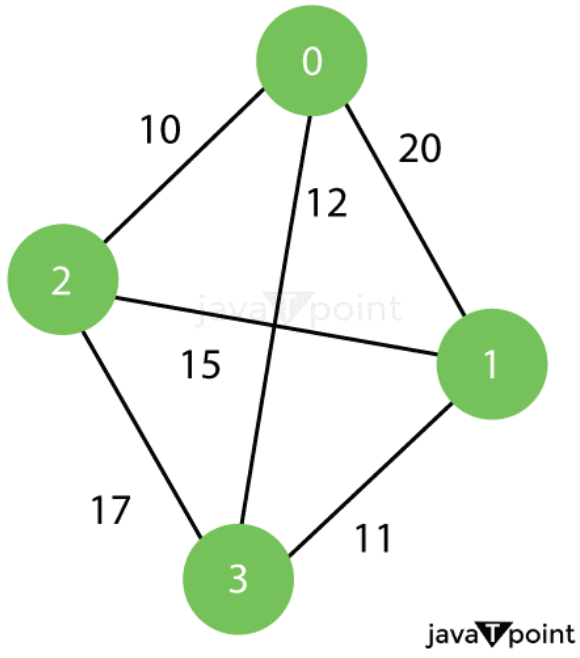
Aim: To study and implement Travelling Salesman Problem

Travelling Salesman Problem

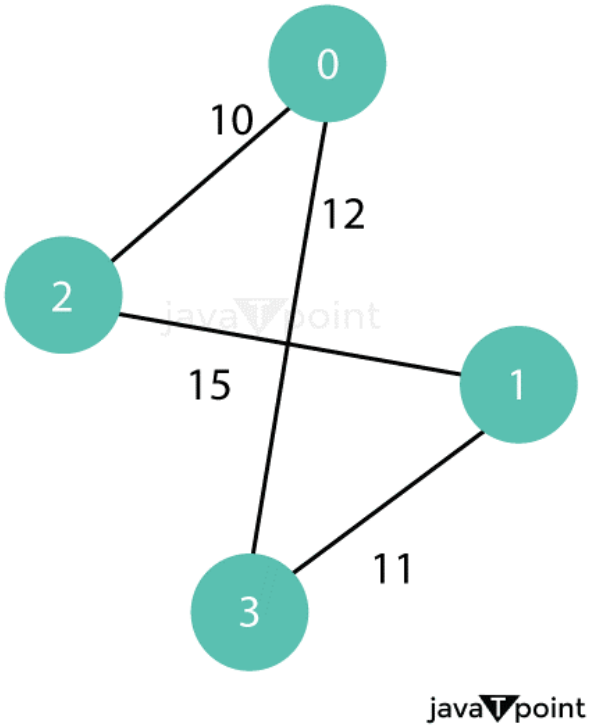
The Travelling Salesman Problem (also known as the Travelling Salesperson Problem or TSP) is an NP-hard graph computational problem where the salesman must visit all cities (denoted using vertices in a graph) given in a set just once. The distances (denoted using edges in the graph) between all these cities are known. We are requested to find the shortest possible route in which the salesman visits all the cities and returns to the origin city.

Example 1 of Travelling Salesman Problem

Input:

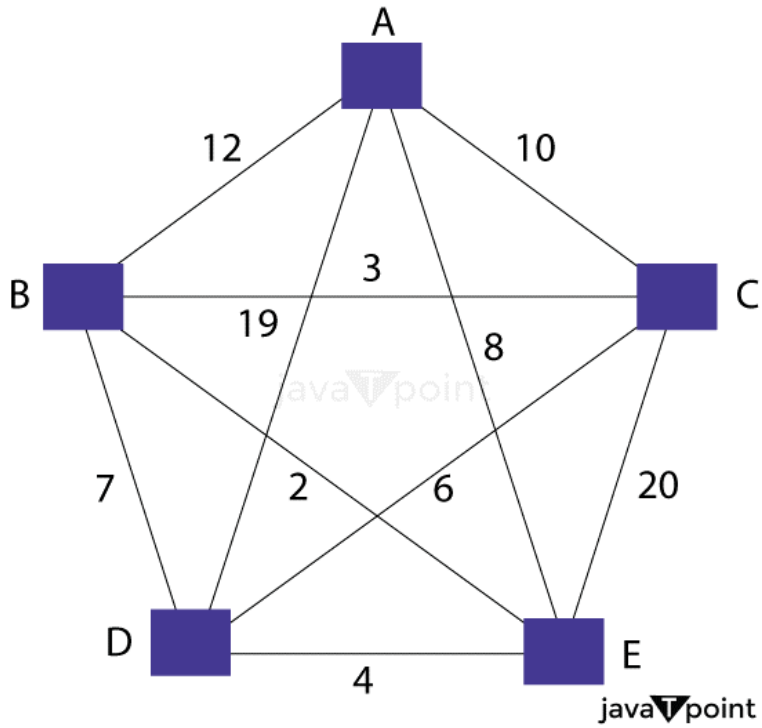


Output:



Example 2 of Travelling Salesman Problem

Input:



Output:

```
Minimum Weight Hamiltonian Cycle: EACBDE = 32
```

Solution of the Travelling Salesman Problem

Approaches:

1. Simple or Naïve Approach
2. Dynamic Programming Approach
3. Greedy Approach

Solving Travelling Salesman Problem using Simple Approach

In the following approach, we will solve the problem using the steps mentioned below:

Step 1: Firstly, we will consider City 1 as the starting and ending point. Since the route is cyclic, any point can be considered a starting point.

Step 2: As the second step, we will generate all the possible permutations of the cities, which are (n-1)!

Step 3: After that, we will find the cost of each permutation and keep a record of the minimum cost permutation.

Step 4: At last, we will return the permutation with minimum cost.

Solving Travelling Salesman Problem Using Dynamic Programming Approach

In the following approach, we will solve the problem using the steps mentioned below:

Step 1: In travelling salesman problem algorithm, we will accept a subset N of the cities that require to be visited, the distance among the cities, and starting city S as inputs. Each city is represented by a unique city id (let's say 1, 2, 3, 4, 5, ..., n).

Step 2: Let us consider 1 as the starting and ending point of the problem. For every other node V (except 1), we will calculate the minimum cost path having 1 as the starting point, V as the ending point, and all nodes appearing exactly once.

Step 3: Let the cost of this path cost (i), and the cost of the corresponding cycle would be cost(i) + distance(i, 1) where the distance(i, 1) from V to 1. Finally, we will return the minimum of all [cost(i) + distance(i, 1)] values.

Solving Travelling Salesman Problem Using Greedy Approach

In the following approach, we will solve the problem using the steps mentioned below:

Step 1: First of all, we will create two primary data holders - The first data holder will be the list to store the indices of the cities in terms of the input matrix of distances between the cities, and the second one will be the array to store the result.

Step 2: For the second step, we will traverse through the given adjacency matrix for all the cities and update the cost by checking if the cost of reaching any city from the current city is lower than the current cost.

Step 3: At last, we will generate the minimum path cycle with the help of the above step and return their minimum cost.

Code:

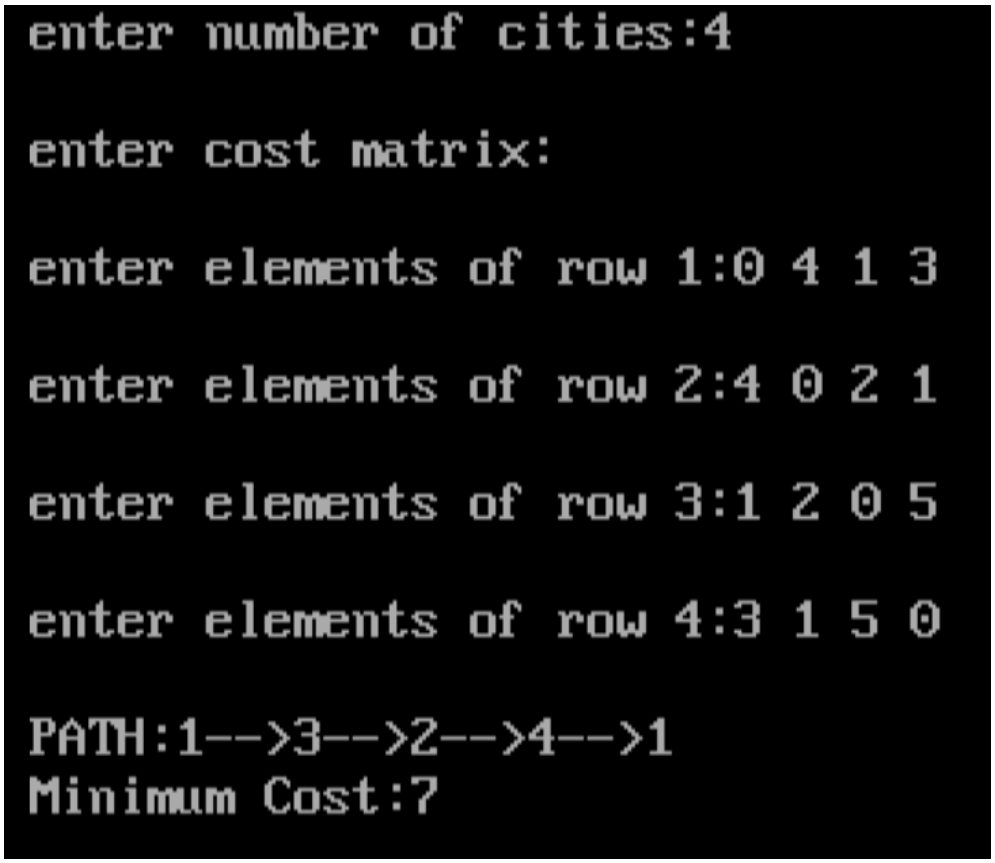
```
#include<stdio.h>

#include<conio.h>



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



void ip()

{

	int i,j;

	printf("enter number of cities:");

	scanf("%d",&n);

	printf("\nenter cost matrix:\n");
```

```
        for(i=0;i<n;i++)
        {
                printf("\nenter elements of row %d:",i+1);
                for(j=0;j<n;j++)
                {
                        scanf("%d",&arr[i][j]);
                }
                completed[i]=0;
        }
}

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+=arr[city][ncity];
            return;
        }
        mincost(ncity);
}

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

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

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

void main()
{
        clrscr();
        ip();
        printf("\nPATH:");
        mincost(0);
        printf("\nMinimum Cost:%d",cost);
        getche();
}
```

Output:



Conclusion:

Hence, we have successfully implemented Travelling Salesman Problem.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 10:

Aim: To implement the Subset Sum problem.

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:

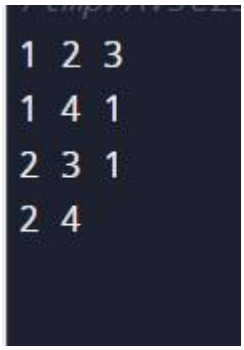
Time Complexity: O(2n) The above solution may try all subsets of the given set in the worst case. Therefore the time complexity of the above solution is exponential. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).
Auxiliary Space: O(n) where n is recursion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

Code:

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

#define MAX_SIZE 100

int ans[MAX_SIZE][MAX_SIZE];
int count = 0;

void subsetSum(int nums[], int temp[], int sum, int i, int k, int n) {
    if (sum == k) {
        for (int j = 0; j < i; j++)
            ans[count][j] = temp[j];
        count++;
        return;
    }
    if (sum > k || i >= n)
        return;

    temp[i] = nums[i];
    subsetSum(nums, temp, sum + nums[i], i + 1, k, n);
```

```
    temp[i] = 0; // Backtrack: reset temp[i] to 0 before calling subsetSum without including nums[i]
    subsetSum(nums, temp, sum, i + 1, k, n);
}

int main() {
    int nums[] = {1, 2, 3, 4, 1};
    int n = sizeof(nums) / sizeof(nums[0]);
    int temp[MAX_SIZE];

    subsetSum(nums, temp, 0, 0, 6, n);

    for (int i = 0; i < count; i++) {
        for (int j = 0; j < n; j++) {
            if (ans[i][j] != 0)
                printf("%d ", ans[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:



Conclusion:

Thus we have successfully implemented Subset Sum problem.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 11:

Aim: To implement the N-Queens Problem.

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

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:

Follow the steps mentioned below to implement the idea:

1. Start in the leftmost column

2. If all queens are placed return true
3. Try all rows in the current column. Do the following for every row.
4. If the queen can be placed safely in this row
5. Then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
6. If placing the queen in [row, column] leads to a solution then return true.
7. If placing queen doesn't lead to a solution then unmark this [row, column] then backtrack and try other rows.
8. If all rows have been tried and a valid solution is not found, return false to trigger backtracking.

Code:

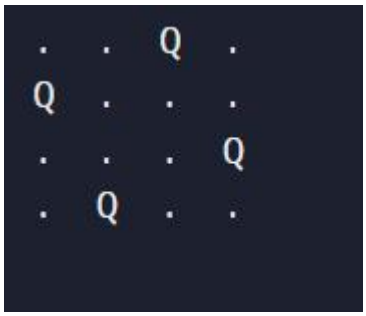
```
#include <stdio.h>

#include <stdlib.h>

#define N 4

void printSolution(char board[N][N]) {

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

        for (int j = 0; j < N; j++)

            printf(" %c ", board[i][j]);

        printf("\n");

    }

}

int isSafe(char board[N][N], int row, int col) {

    int i, j;

    for (i = 0; i < col; i++)
```

```
        if (board[row][i] == 'Q')
            return 0;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 'Q')
            return 0;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j] == 'Q')
            return 0;

    return 1;
}

int solveNQUtil(char board[N][N], int col) {
    if (col >= N)
        return 1;

    for (int i = 0; i < N; i++) {
```

```
        if (isSafe(board, i, col)) {
            board[i][col] = 'Q';

            if (solveNQUtil(board, col + 1))
                return 1;

            board[i][col] = '.';
        }
    }

    return 0;
}

void solveNQ() {
    char board[N][N];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            board[i][j] = '.';

    if (solveNQUtil(board, 0) == 0) {
        printf("Solution does not exist");
```

```
        return;
    }

    printSolution(board);
}

int main() {
    solveNQ();
    return 0;
}
```

Output:



Conclusion:

Thus we have successfully implemented N-Queens Problem.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 12:

Aim: To study and implement Knuth-Morris-Pratt algorithm

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:

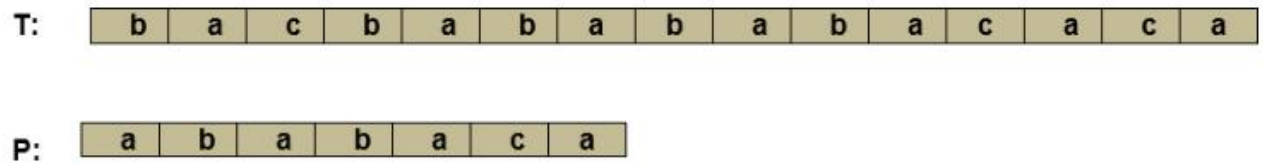
The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

```
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:



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:

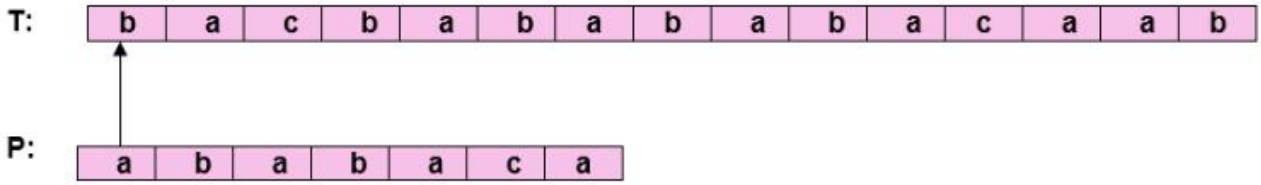
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

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

Step1: i=1, q=0

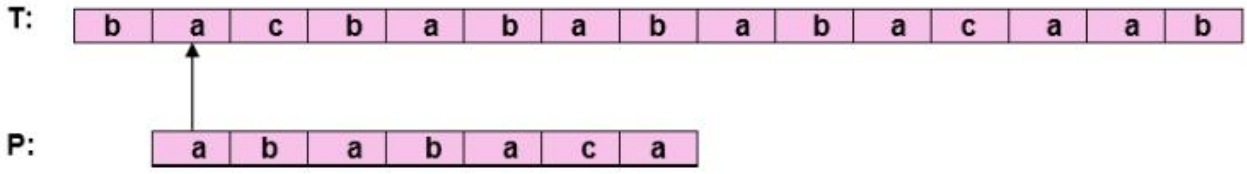
Comparing P [1] with T [1]



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

Step2: i = 2, q = 0

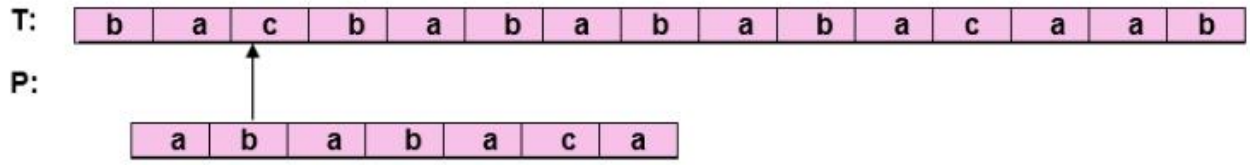
Comparing P [1] with T [2]



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

Step 3: i = 3, q = 1

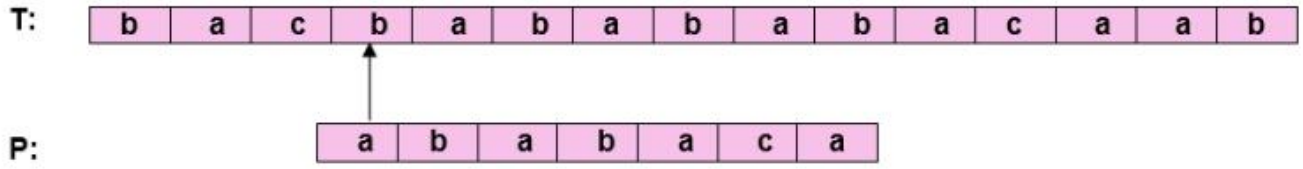
Comparing P [2] with T [3] P [2] doesn't match with T [3]



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

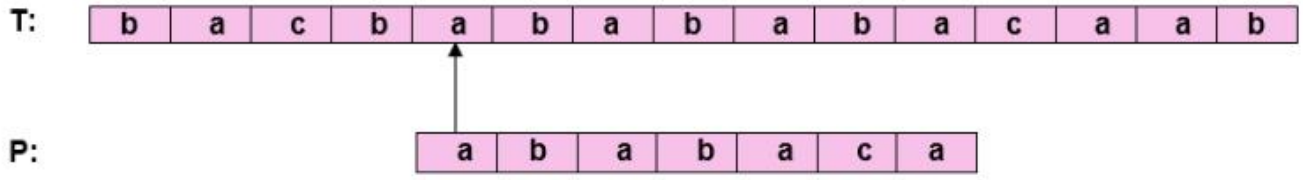
Step4: i = 4, q = 0

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



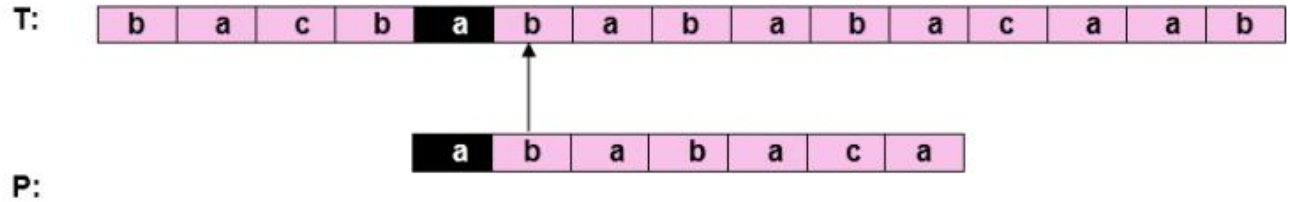
Step5: i = 5, q = 0

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



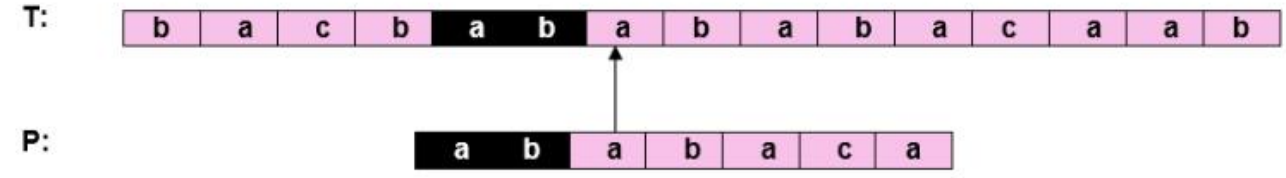
Step6: i = 6, q = 1

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



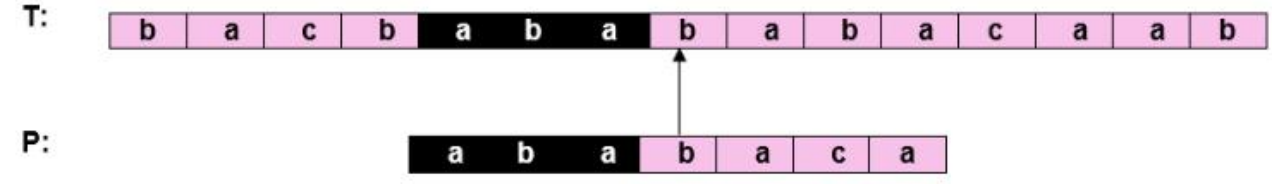
Step7: i = 7, q = 2

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



Step8: i = 8, q =3

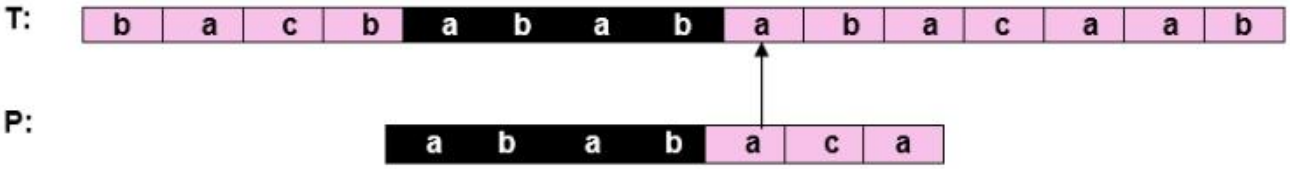
Comparing P [4] with T [8] P [4] matches with T [8]



Step9: i = 9, q = 4

Comparing P [5] with T [9]

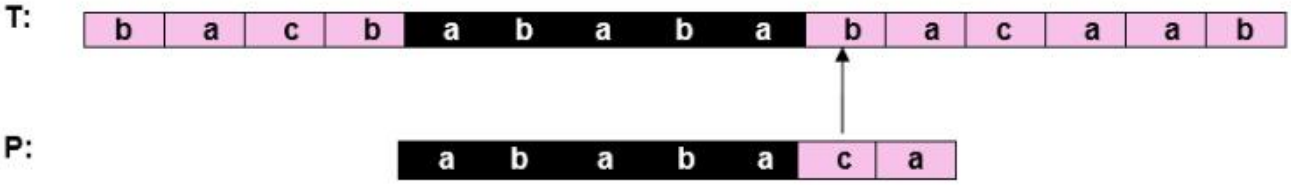
P [5] matches with T [9]



Step10: i = 10, q = 5

Comparing P [6] with T [10]

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

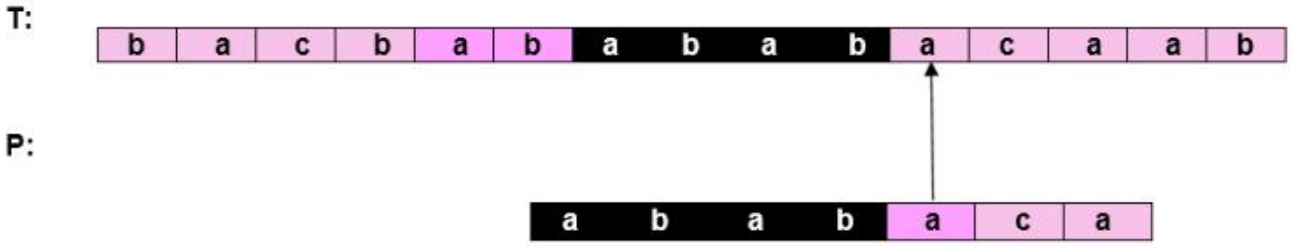


Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

Step11: i = 11, q =4

Comparing P [5] with T [11]

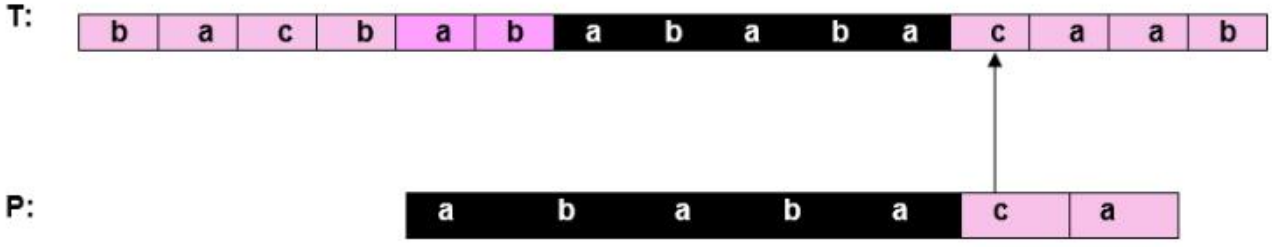
P [5] match with T [11]

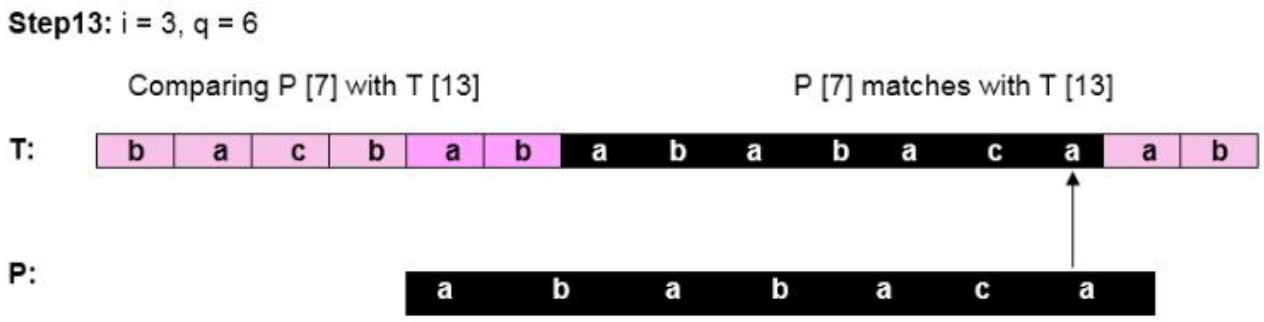


Step12: i = 12, q = 5

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.

Code:

```
#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:

```
------------ KMP ALGORITHM ---------------

Enter the text: abcdefg
Enter the pattern: cd
Found pattern at index 2


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

```
------------ KMP ALGORITHM ---------------

Enter the text: abcdabcdab
Enter the pattern: ab
Found pattern at index 0
Found pattern at index 4
Found pattern at index 8


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

Conclusion:

Thus we have successfully implemented Knuth-Morris-Pratt algorithm.

Name: Fareeha Shaikh

Roll No: S21-92

EXPERIMENT 13:

Aim: To study & implement Rabin-Karp Algorithm for Pattern Searching

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.....

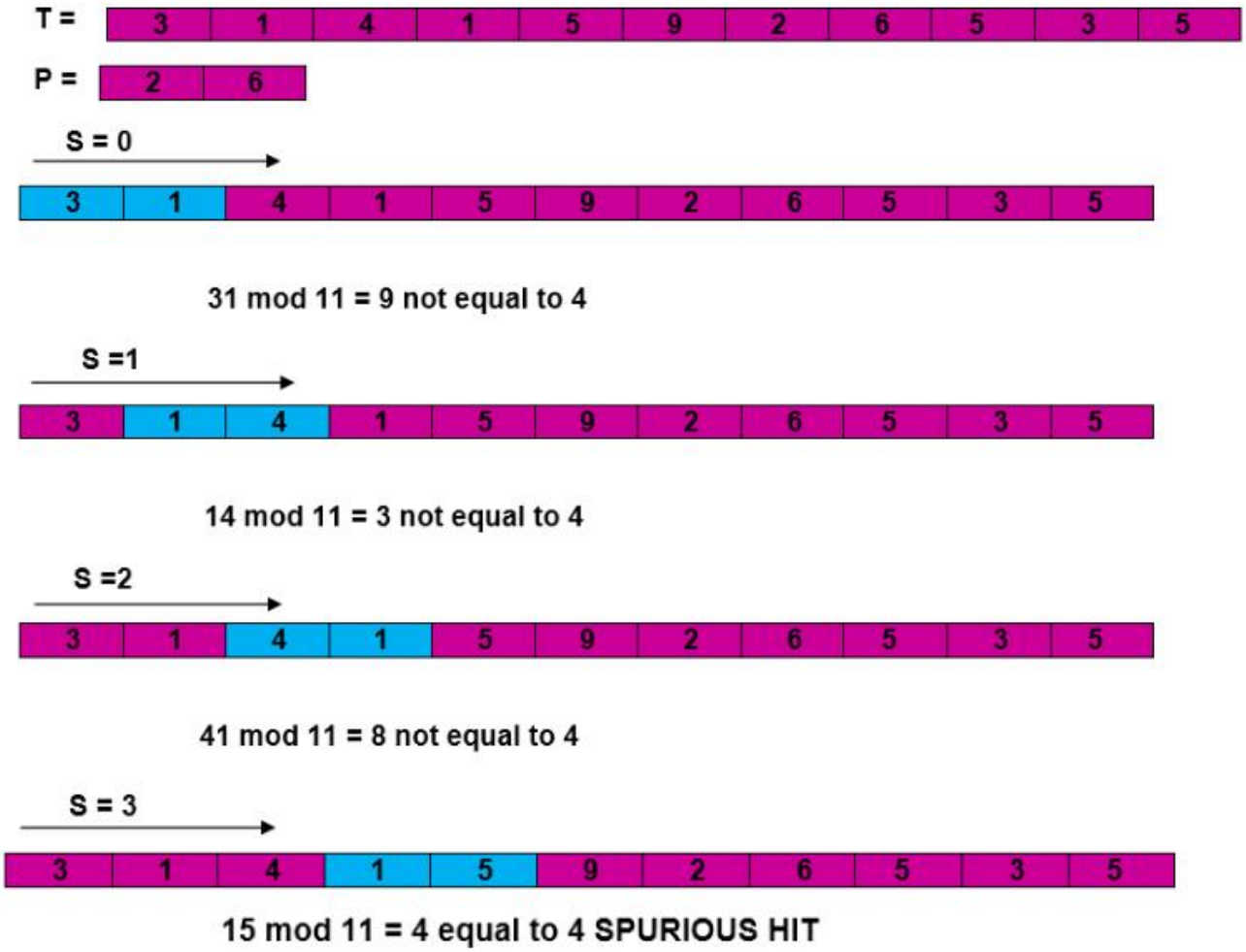
P = 26

Here T.Length =11 so Q = 11

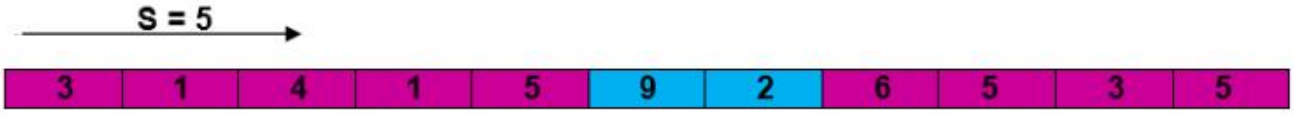
And P mod Q = 26 mod 11 = 4

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

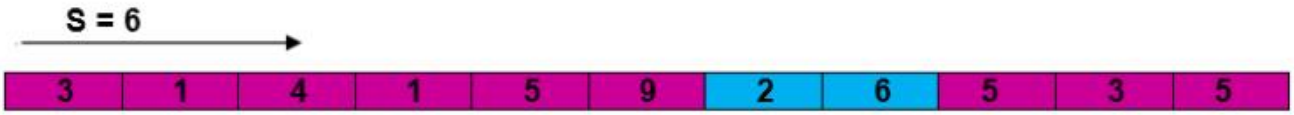
Solution:



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

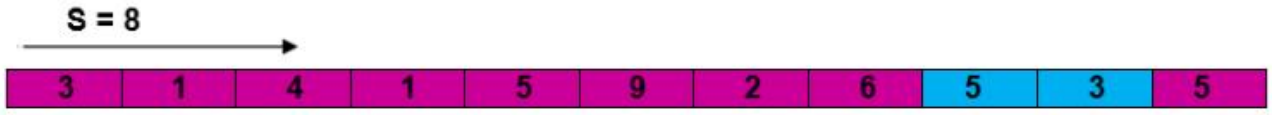


92 mod 11 = 4 equal to 4 **SPURIOUS HIT**



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.

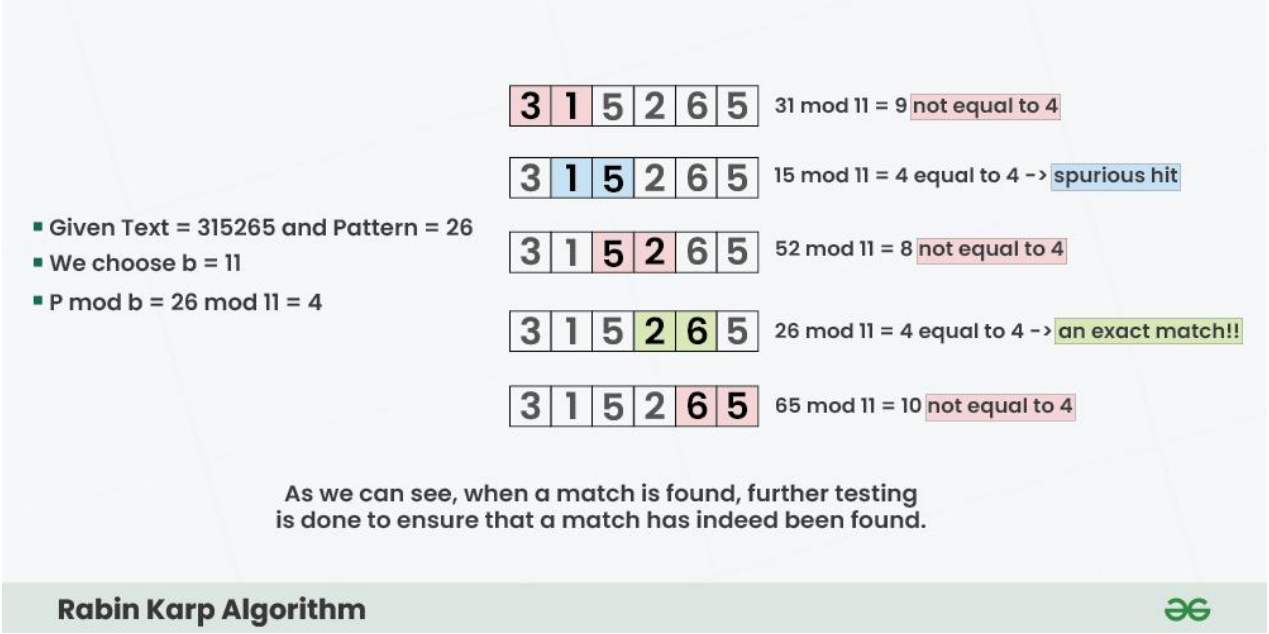
Step 4: Slide the pattern over the text:
Start by calculating the hash value for the first substring of the text that is the same length as the pattern.

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.

Code:

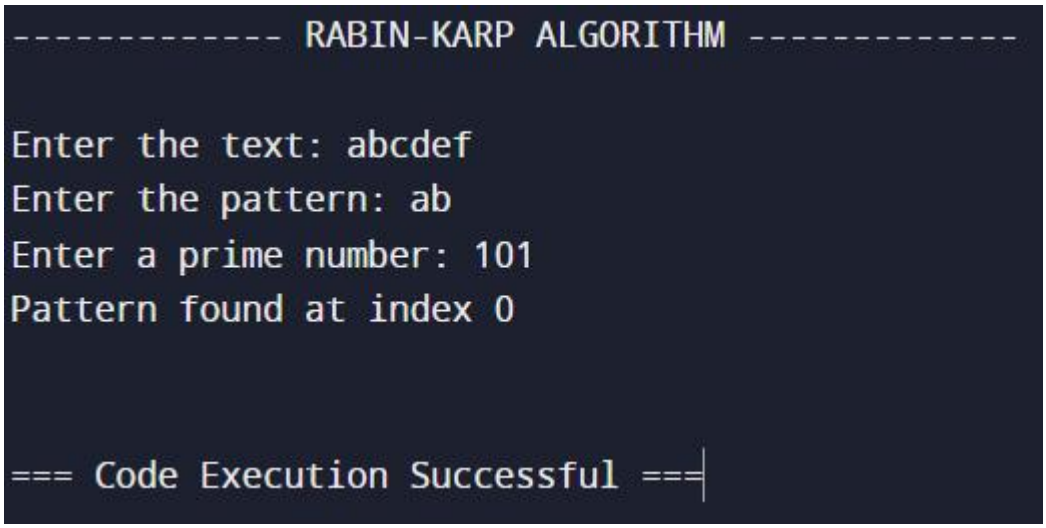
```
#include <stdio.h>

#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:

Thus we have successfully implemented Rabin-Karp Algorithm for Pattern Searching.

AOA: Assignment 1:-

1. Use master theorem to find order of following Divide and conquer recurrence rate relation (assuming T=1)

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

Here, $a=2$, $b=2$, $k=3$, $p=0$

$b^k = 2^3 = 8$

Case 3: $a < b^k$, i.e. $2 < 8$

$\because p = 0$

$\therefore T(n) = \Theta(n^3 \log^0 n) = \Theta(n^3)$

b) $T(n) = T(an/10) + n$

Here, $a=1$, $b = 10/a$, $k=1$, $p=0$

$\therefore b^k = 10/a$

$a < b^k$

$\therefore$ Case 3: $\because p \geq 0$;

$\therefore T(n) = \Theta(n^k \cdot \log^p n) = \Theta(n)$

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

Here, $a=16$, $b=4$, $k=2$, $p=0$

Case 2: $a = b^k$ i.e. $16 = 4^2$

$\therefore p > -1$, then $\Theta(n^{\log_4 k} \log^{p+1} n) = \Theta(n^2 \log n)$

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

Here, $a=4$, $b=2$, $k=3$, $p=0$

case 3: $4 < 2^3$

$p \geq 0$

$\therefore T(n) = \Theta(n^3 \log^0 n) = \Theta(n^3)$

e) $T(n) = 2T(n/2) + n+1$

Here, $a=2, b=2, k=1, p=0$

Case 2: $a = b^k$ $\because 2 = 2^1$

$\therefore p > -1$

$\therefore = O(n^{\log_2 2} \log^{0+1} n)$

$= O(n^1 \log n)$

$= O(n \log n)$

Q.2)

i) $T(n) = 5T(n/2) + 3$, $T(1) = 1$

Using master theorem,

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

$5 > 2^0$

$\therefore a > b^k$

$\therefore T(n) = O(n^{\log_b a})$

$= O(n^{\log_2 5})$

ii) $T(n) = T(n/4) + cn$ & $T(1) = 1$

Using master's,

$a = 1, b = 4, k = 1, p = 0$

Case 3: $a < b^k$ and $p \geq 0$

$\therefore T(n) = O(n^k \cdot \log^p n)$

$= O(n^1 \cdot \log^0 n)$

$= O(n)$

3. Fix a constant $c > 0$. Solve recurrence $T(n) = cT(n/2) + n$ using substitution method.

→ $T(n) = cT(n/2) + n$ ... (1)

Putting $n = n/2$ in (1)

$T(n/2) = cT(n/4) + n/2 \quad \cdots (2)$

using (2) in (1)

$T(n) = c\left[cT(n/4) + n/2\right] + n$

$T(n) = c^2 T(n/4) + cn/2 + n \quad \cdots (3)$

Putting $n = n/4$ in (1)

$T(n/4) = cT(n/8) + n/4 \quad \cdots (3a)$

using (3a) in (3)

$T(n) = c^2\left[cT(n/8) + n/4\right] + cn/2 + n$

$= c^3 T(n/8) + c^2 n/4 + cn/2 + n$

∴ After k iteration,

$T(n) = c^k T(n/2^k) + n \sum_{i=1}^{k-1} (c^{k-1}/2^{k-1})$

$= c^k T(n/2^k) + n\left[\frac{1-(c/2)^k}{1-(c/2)}\right] \quad c^k T(n/2^k) + n\left[\frac{c^k}{2^{k-1}} + \frac{c^{k-1}}{2^{k-2}} \cdots 1\right]$

Let $n = 2^k$

$\therefore k = \log_2 n$

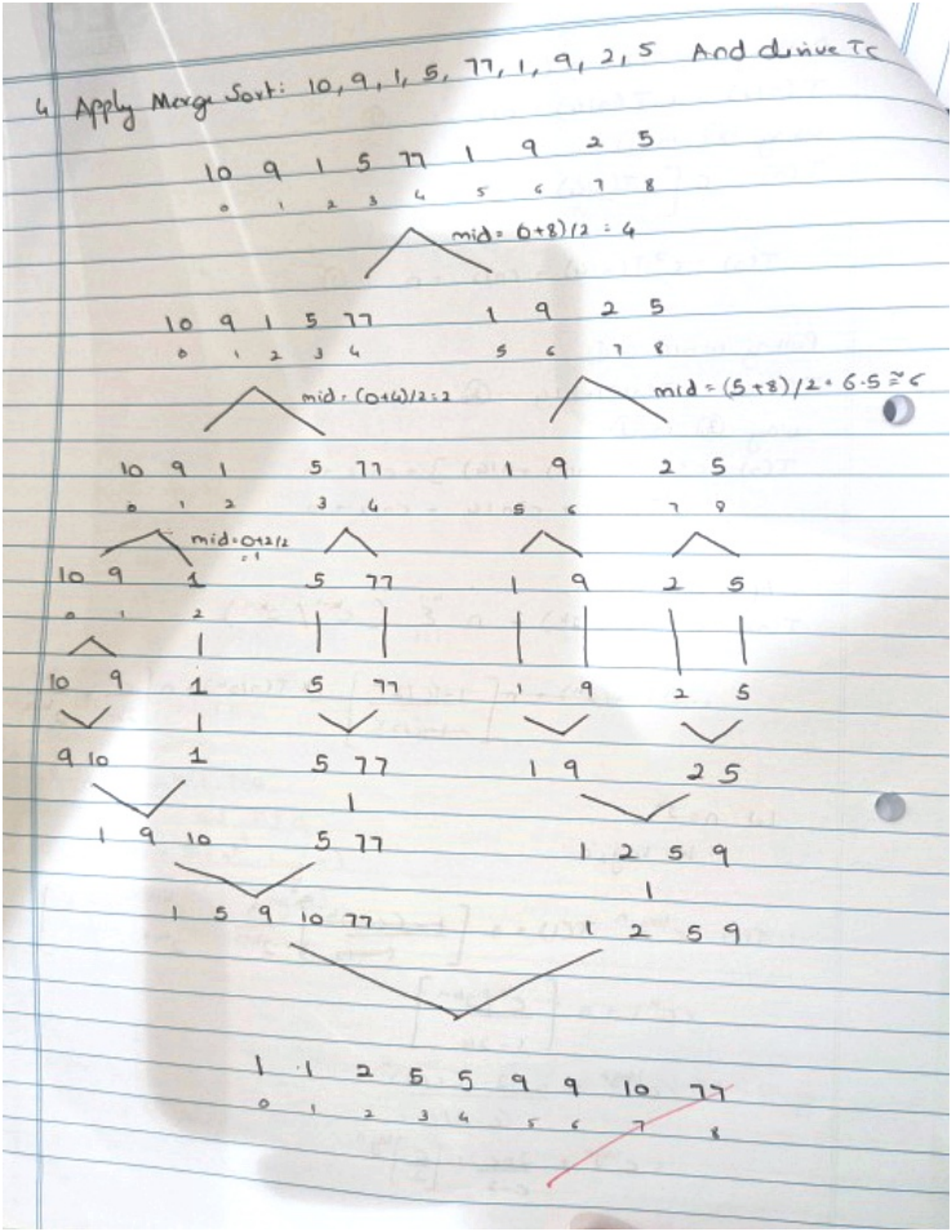
$\therefore T(n) = c^{\log_2 n} T(1) + n\left[\frac{1-(c/2)^k}{1-c/2}\right] \frac{c^k}{2^{k-1}} + \frac{c^{k-1}}{2^{k-2}} \cdots 1$

$= c^k 1 + n\left[\frac{c^k/2^{k-1}}{1-2/c}\right]$

$= c^{\log n} + \frac{n \cdot 2 \cdot (c/2)^k}{(c-2)/c}$

$= c^{\log n} + \frac{2nc}{c-2} \cdot \left[\frac{c}{2}\right]^{\log n}$

4 Apply Merge Sort: 10, 9, 1, 5, 77, 1, 9, 2, 5 And drive Tc



Time complexity:

1) Divide: To compute middle index of the array. It's done in constant time $\therefore D(n) = \Theta(1)$

2) Conquer: recursively solve 2 subproblems each of size $(n/2)$, which contributes $2T(n/2)$ to run time

3) Combine: It merges 2 sorted sublists and does n comparisons $\therefore C(n) = \Theta(n)$

$$\therefore T(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ T(n/2) + T(n/2) + \Theta(n) + cn \end{cases}$$

Using substitution method

$T(n) = 2T(n/2) + n \quad \cdots (1)$

putting $n = n/2$ in (1)

$T(n/2) = 2T(n/4) + n/2 \quad \cdots (2)$

using (2) in (1)

$T(n) = 2\left[2T(n/4) + n/2\right] + n$

$= 4T(n/4) + 2n \quad \cdots (3)$

putting $n = n/4$ in (1)

$T(n/4) = 2T(n/8) + n/4 \quad \cdots (4)$

using (4) in (3)

$T(n) = 4\left[2T(n/8) + n/4\right] + 2n$

$= 8T(n/8) + 3n$

$\therefore$ After k iterations,

$T(n) = 2^k T(n/2^k) + kn$

Let $n = 2^k \quad \therefore k = \log_2 n$

$T(n) = n\,T(1) + n\log_2 n$

$= n + n\log_2 n$

$\therefore T(n) = O(n\log_2 n)$

5. Quick sort: E, X, A, M, P, L, E. Find best, worst, average case TC

→ Let 1st element be pivot:

0	1	2	3	4	5	6	
E	X	A	M	P	L	E	∞

i → (pivot), j ←

We increment i till i ≤ pivot and element j till j ≥ pivot ~~and~~

$\therefore$

0	1	2	3	4	5	6	
E	X	A	M	P	L	E	∞

pivot, i →, j ←

$\therefore$ Swapping i & j:

0	1	2	3	4	5	6	
E	E	A	M	P	L	X	∞

pivot, i →, j ←

Again increment i till element(i) ≥ pivot and element j till element(j) ≤ pivot.

E	E	A	M	P	L	X	∞

pivot, j, i

$\therefore$ cross-over $\therefore$ swap a[pivot] and a[j]

$\therefore$

A	E	E	M	P	L	X

Sorting for right part:

M P L X
p i j

Incrementing i till a[i] > pivot and j till a[j] < pivot

M P L X
i j

- swap a[i] and a[j]

∴ M L P X
p i j

∴ M L P X
p j i

∴ swap a[p] and a[j]

∴ L M P X

∴ the sorted array is:

0	1	2	3	4	5	6
A	E	E	L	M	P	X

Best case:
when pivot is the middle element
∴ for each subarray, = n/2
time to fix pivot = n

$T(n) = 2T(n/2) + n$ ... ①

Putting $n = n/2$ in ①

$T(n/2) = 2T(n/2^2) + n/2$ ... ②

using ② in ①

$T(n) = 2[2T(n/4) + n/2] + n$

$T(n) = 4T(n/4) + 2n$ ... ③

Putting $n = n/4$ in ①

$T(n/4) = 2T(n/8) + n/4$ ... ④

using ④ in ③,

$T(n) = 4(2T(n/8) + n/4) + 2n$

$= 8T(n/8) + 3n$ →

∴ After k iterations,

$T(n) = 2^k T(n/2^k) + kn$

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

$\therefore T(n) = nT(1) + n\log_2 n$

$= n + n\log_2 n$

$= O(n\log_2 n)$

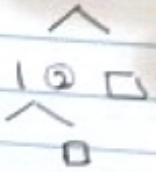
Worst Case:-

It is when list is already sorted.

∴ pivot will always be at either end.

∴ subproblem with $(n-1)$ elements and 0 elements will be created

eg: 1 2 3



: To divide the list, the algorithm scans and detects correct position for pivot ∴ n
one sublist of size 0 other of n-1

∴ $T(n) = T(n-1) + n$ ... (1)

Put $n = n-1$ in (1)

$\therefore T(n-1) = T(n-2) + (n-1)$ ... (2)

using (2) in (1)

$T(n) = T(n-2) + (n-1) + n$ ... (3)

: after k iteration,

$T(n) = T(n-k) + (n-k+1) \cdots (n-1) + n$

Let $k = n$

$\therefore T(n) = T(0) + 1 + 2 \cdots (n-1) + n$

$= n(n+1)/2$

$= \frac{n^2}{2} + \frac{n}{2}$

$= O(n^2)$ //.

Average case:
when pivot nearly divides list into two equal sublists.
$T(n) = O(n \log_2 n)$

6. Determine frequency count for all statements and calculate TC

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

Frequency count:
initialization of I=1
increment of X and I combined: 2n
loop = n

Time complexity:
Initialization: O(1)
while loop: O(n)
Increment of X and I combined: O(n)

∴ Overall: O(1) + O(n) + O(n)
= O(2n+1)
= O(n)

7. Explain recurrences and various methods to solve:

→ Recurrences are mathematical equations that describe behaviour of a function in terms of its value at smaller input. They are commonly used to analyze time complexity.

Methods:

i) Substitution method:
It involves representing recurrences of a tree guessing a solution and then solving by using mathematical induction.

ii) Recurrence tree method:
It involves representing recurrences of a tree where each node

represents cost of single subproblem

iii) Master Theorem:
It involves a straight-forward way to solve recurrence of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

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

iv) Iteration method:
This method solves by expanding the recurrence into iterations until a pattern emerges.

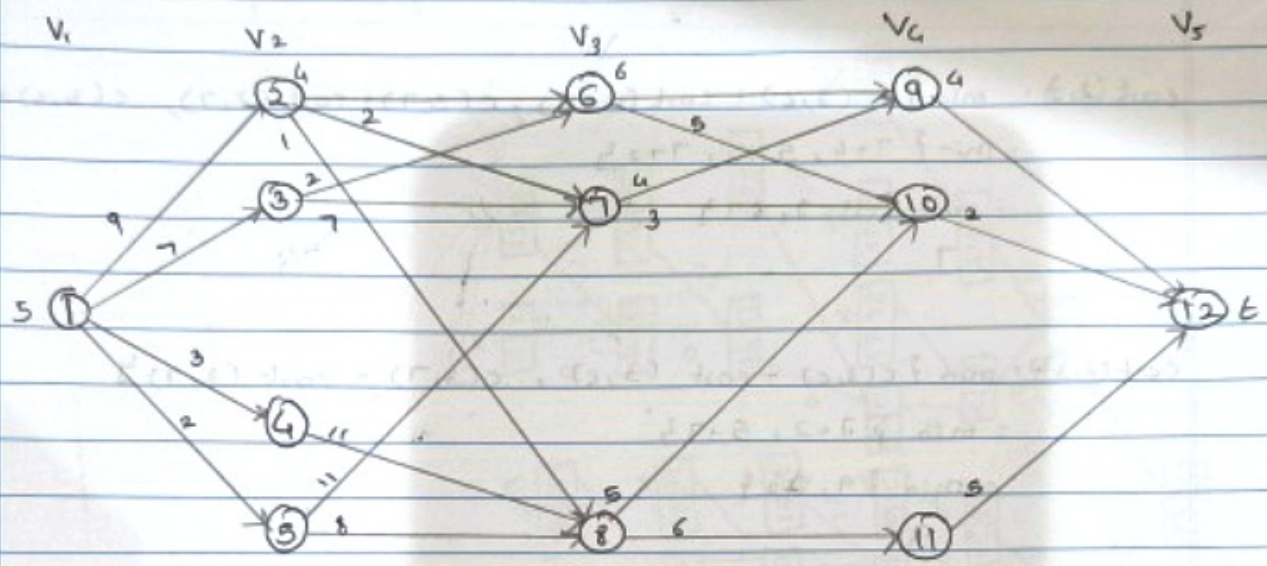
v) Generating functions:
These are used to represent sequences of number as power series. Recurrences can be translated into equations involving generating functions which can be solved using algebraic technique.

8.

P Type	NP Type
i) P problems are a set of problems that can be solved in polynomial time by deterministic algorithms.	i) NP problems are problems that can be solved in non-deterministic polynomial time
ii) P problems can be solved and verified in polynomial time.	ii) The solutions to NP problems cannot be obtained in polynomial time, but if the solution is given, it can be verified in polynomial time.
iii) P problems are subset of NP problems	iii) NP problems are a superset of P problems.
iv) All P problems are deterministic in nature.	iv) All NP problems are non-deterministic in nature.
v) Solution to P class problems is easy to find	v) Solution to NP class problem is hard to find.
vi) Examples: Selection Sort, Linear Search	vi) Examples: Travelling salesman problem, knapsack problem.
vii) Solutions are deterministic if they are computed deterministically by turing machine	viii) Solutions involve non-deterministic choices, where multiple devices are considered simultaneously, but a valid solution can be verified deterministically

AOA : Assignment 2 :-

Q.1. Solve following MCPT:-



→ 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	11	12	12	12

cost (3,6) = min { c(4,9) + cost(6,9) , c(4,10) + cost(6,10) }
= min { 4+6 , 2+5 }
= min {10, 7}
= 7

cost (3,7) = min { c(4,9) + cost(7,9) , c(4,10) + cost(7,10) }
= min { 4+4, 2+3 }
= min { 8, 5 }
= 5

cost(3,8) = min { c(4,20) + cost(8,10), c(4,11) + cost(8,11) }
= min { 2+5, 5+6 }
= min { 7, 11 }
= 7

cost(2,8) = 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+2 }
= min { 11, 7, 9 }
= 7

cost(2,3) = min { c(3,6) + cost(3,6), c(3,7) + cost(3,7) }
= min { 7+2, 5+7 }
= min { 9, 12 }
= 9

cost(2,4) = min { c(3,8) + cost(6,8) }
= 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 }
= min { 16, 15 }
= 15

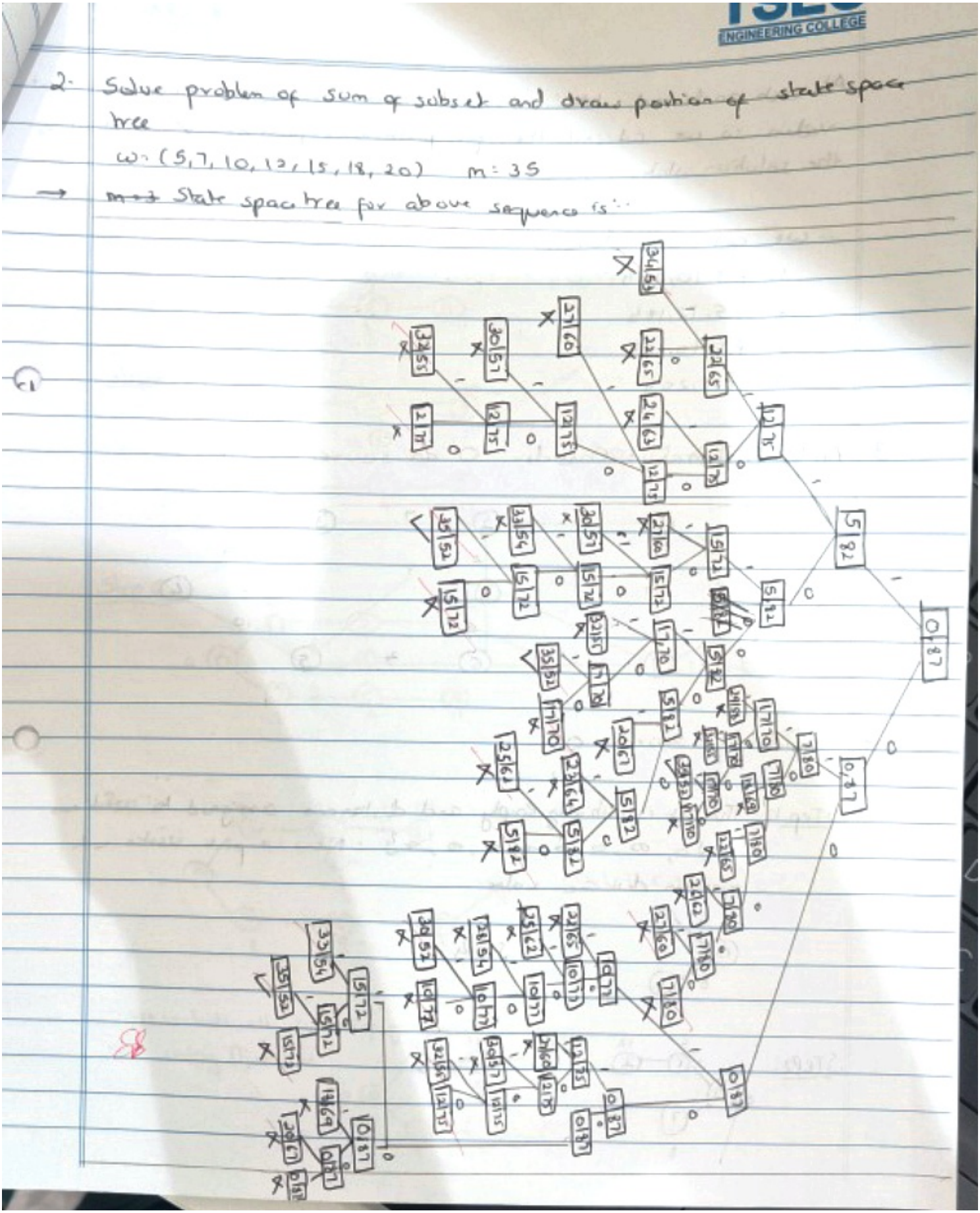
cost(1,1) = min { c(2,2) + cost(1,2), c(2,3) + cost(1,3), c(2,4) + cost(1,4),
c(2,5) + cost(1,5) }
= min { 7+9, 9+7, 18+3, 15+2 }
= min { 16, 16, 27, 17 }
= 16

∴ Shortest paths: 1-2-7-10-12 & 1-3-6-10-12

2. Solve problem of sum of subset and draw portion of state space tree

w: (5, 7, 10, 12, 15, 18, 20) m: 35

→ ~~m=~~ State space tree for above sequence is:



At each node end, '✗' means it can't accomodate any of the new values so we cut-sort them for further expansion. ✓ means it is the solution states.

∴ we get 4 solutions.

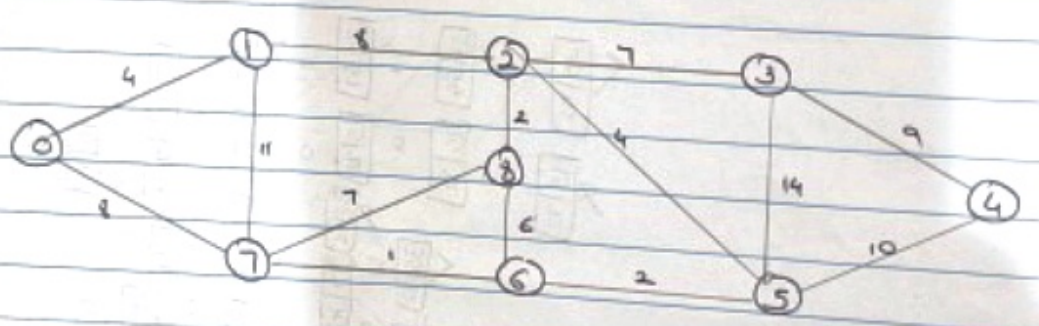
1: {5, 10, 20}

2: {5, 12, 18}

3: {7, 10, 18}

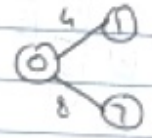
4: {15, 20}

3. Apply Djikstra's. Consider 0 as source.



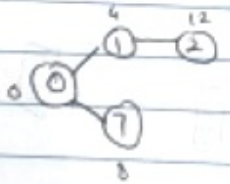
→ Considering source: 0

Step 1:- The set is initially empty and distances assigned to vertices are {0, ∞, ∞, ∞, ∞, ∞, ∞, ∞}. Now we pick vertex with minimum distance value.



{0}

Step 2:

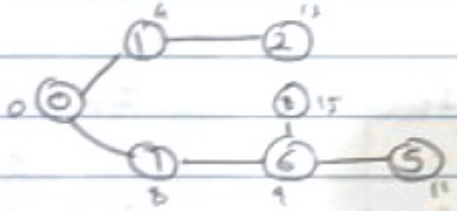


{0, 1}

Step3:

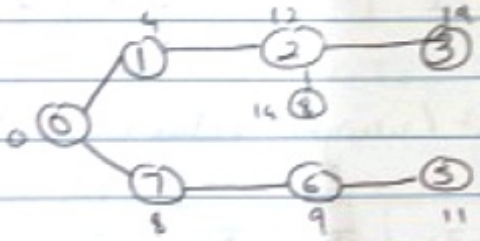
{0, 1, 7}

Step4:



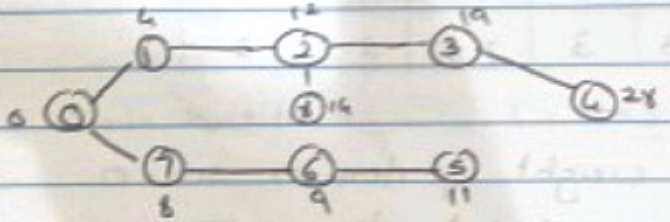
{0, 1, 7, 6}

Step5:



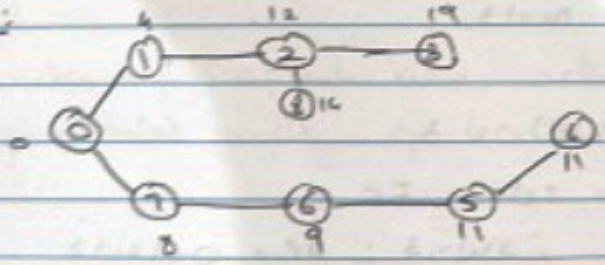
{0, 1, 7, 6, 2}

Step6:



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

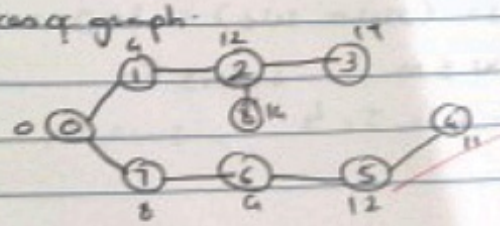
Step7:



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

∴ Set includes all vertices of graph.

SPT using Dijkstra's:



4. Solve fractional knapsack for: n=6
P: {18, 5, 9, 10, 12, 7} W: {7, 2, 3, 5, 3, 2} M=13

Arranging items in decreasing order of profit/weight value.
Assume items are labelled as $I_1, I_2 \cdots I_n$.

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

Decreasing order of Profit/Weight value:

xi	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

Initialise selected weight of items => sw = 0
profit of selected items => sp = 0
set of selected items, s = { }
knapsack capacity : M=13

Iteration 1: sw = (sw + w_5) = 3
sw ≤ M so select I_5
∴ s = {I_5} ; sw = 3 ; sp = 0+12=12

Iteration 2: sw = (sw + w_6) = 3+2 = 5
sw ≤ M ∴ select I_6
s = {I_5, I_6} ; sw = 5 ; sp = 12+7=19

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Iteration 3: $sw = (sw + w_3) = 5 + 3 = 8$

$sw \leq m$ ∴ select I_3

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

Iteration 4: $sw + w_1 > m$ ∴ we break down I_1,

remaining capacity of knapsack is 5, select 5 items of I_1

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

∴ $S = \{I_5, I_6, I_3, I_1 * 5/7\}$

$sp = sp + v_i * 5/7 = 28 + 18 \times \frac{5}{7}$

~~$= 12.857$~~

$\therefore SP = 40.85$

$sw = sw + w_1 \times \frac{5}{7} = 8 + 7 \times \frac{5}{7} = 13$

$= 13$

∴ knapsack is full.

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

5. Write an algo to solve N Queen Problem. Show its working for N=4.

→ N Queen problem demands us to place N queens on NxN chessboard so that no queen can attack another directly.

i) We create a board of NxN size that stores characters. It will store 'Q' if queen has been placed at that position else '.'

ii) We'll create a recursive function called 'solve' that takes board and column and all boards as arguments. We will pass column as 0. So we can start exploring arrangements from column 1.

iii) In solve function we'll go row by row for each column and will check if that particular cell is safe or not for the placement of the queen, will do so with issafe() function.

iv) For each possible cell where the queen is going to be placed, will first check issafe() function.

v) If the cell is safe, we put 'Q' in that row & column of the board and again call the solve function by incrementing the column by 1.

vi) Whenever we reach a position where the column becomes equal to board length, this implies that all columns and possible arrangements have been explored, so we return.

vii) Coming on to issafe(), we check if a queen is already present in that row / column / diagonal. If yes, we return false. Else we put board[row][column] = "Q" and return true.

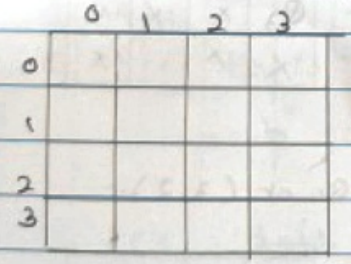
Algorithm:

```
Algo N_queen(k,n)
// Input: n: number of queen
// Output: nx1 solution table
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
```

N-Queen Problem:-

This problem demands us to put n queens on nxn chessboard in a way that one queen is present in each row and column and no queen can attack other queen directly.



nxn board.

We will put Q_1, Q_2, Q_3, Q_4 in this.

Q_1 can be placed anywhere.

placing Q_1 at (0,0):

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

Put Q_3 at (1,2)

Q_1	x	x	x
x	x	Q_2	x
x	x	x	x
x		x	x

But, this placement of Q_3 blocks all boxes of row 3. Hence, there is no way to put Q_3. We backtrack and put Q_2 at (1,3)

Q_1	x	x	x
x	x	x	Q_2
x	-	x	x
x	x		x

Now, puting Q_3 at (2,1)

Q_1	x	x	x
x	x	x	Q_2
x	Q_3	x	x
x	x	x	x

Now, there is no box to place Q_4. Backtracking all ways of placing Q_2 & Q_3 have been explored ∴ Adjusting Q_1

Put Q_1 at (0,1):

x	Q_1	x	x
x	x	x	
	x		x
	x		

Put Q_2 at (1,3)

x	Q_1	x	x
x	x	x	Q_2
	x	x	x
	x		x

Put Q_3 at (2,0):

x	Q_1	x	x
x	x	x	Q_2
Q_3	x	x	x
x	x		x

Put Q_4 at (3,2):-

	Q_1		
			Q_2
Q_3			
		Q_4	

∴ Through backtracking, we reached a solution when n queens are put in each column and row so that no queen is attacking any other on nxn board.

The two solutions for N=4 Queens: (1,3,0,2) & (2,0,3,1)

6. Find longest common subsequence for following string:
X: abacdb Y: bacadb

→ Formula: If x[i] == y[j]
c[i][j] = 1 + c[i-1, j-1]
else:
c[i][j] = Max { c[i-1, j], c[i, j-1] }

i ↓ \ j Y→	^	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)

∴ b d c a b

∴ LCS: ~~dcab~~ / bacd

TSEC
ENGINEERING COLLEGE

7.

Prim's Algorithm	Kruskal's Algorithm
i) It starts to build minimum spanning tree from any vertex in the graph.	i) It starts to build MST from the vertex carrying minimum weight in graph
ii) It traverses a node more than one time to get the minimum distance.	ii) It traverses one node only once.
iii) Prim's algorithm has a time complexity of $O(V^2)$, V being no. of vertices and can be improved to $O(E \log V)$ using Fibonacci heaps.	iii) Kruskal's time complexity is $O(E \log V)$, V being no. of vertices.
iv) It gives connected component as well as it works only on connected graph.	iv) It can generate forest at any instant as well as work on disconnected components.
v) It runs faster in dense graphs	v) It runs faster in sparse graphs
vi) Applications: TSP, network for roads and rail, etc.	vi) Applications: LAN connection, TV network, etc.
vii) It prefers list data structures	vii) It prefers heap data structure.