Shreya Shah
60004210045
A1
Computer Engineering
AoA All Experiments+Assignments

Aim:

To write a program to analyze insertion and selection sort.

Theory:

Insertion Sort is a simple sorting algorithm that works by sorting an array one element at a time. It works by dividing the array into two parts: the sorted part and the unsorted part. The algorithm starts by assuming that the first element of the array is already sorted and then iterates through the remaining elements, inserting each element into its proper position in the sorted part of the array.

Insertion Sort has a time complexity of O(n^2), making it less efficient than other sorting algorithms such as Quick Sort or Merge Sort. However, it is often faster than these algorithms for small arrays or partially sorted arrays.

The algorithm is simple to understand and implement, making it a popular choice for simple sorting tasks. It is also an in-place sorting algorithm, meaning it does not require additional memory to sort the array, making it useful for situations where memory is limited.

Overall, Insertion Sort is a useful algorithm for sorting small or partially sorted arrays efficiently. Its simplicity and in-place nature make it a popular choice for simple sorting tasks.

Selection Sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of an array and putting it at the beginning of the sorted part of the array. The algorithm works by dividing the array into two parts: the sorted part and the unsorted part. The sorted part is initially empty, while the unsorted part contains all the elements of the array.

Selection Sort has a time complexity of O(n^2), making it less efficient than other sorting algorithms such as Quick Sort or Merge Sort. However, it is useful for small arrays or partially sorted arrays.

The algorithm is simple to understand and implement, making it a popular choice for simple sorting tasks. It is also an in-place sorting algorithm, meaning it does not require additional memory to sort the array, making it useful for situations where memory is limited.

Overall, Selection Sort is a useful algorithm for sorting small or partially sorted arrays efficiently. Its simplicity and in-place nature make it a popular choice for simple sorting tasks.

Algorithm:

Insertion Sort:

```
InsertionSort(A)
   for i = 1 to length(A) - 1
      key = A[i]
      j = i - 1
      while j >= 0 and A[j] > key
         A[j+1] = A[j]
         j = j - 1
      end while
      A[j+1] = key
   end for
end InsertionSort
```

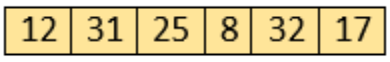
Selection Sort:

```
SelectionSort(A)
   for i = 0 to length(A) - 1
      min_idx = i
      for j = i + 1 to length(A) - 1
         if A[j] < A[min_idx]
            min_idx = j
         end if
      end for
      temp = A[i]
      A[i] = A[min_idx]
      A[min_idx] = temp
   end for
end SelectionSort
```

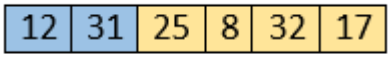
Example:

Insertion Sort:

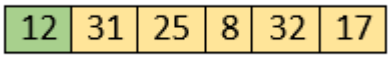
Let the elements of array are –



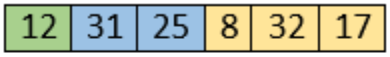
Initially, the first two elements are compared in insertion sort.

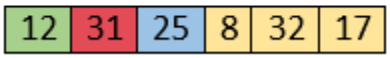


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



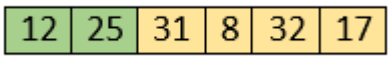
Now, move to the next two elements and compare them.



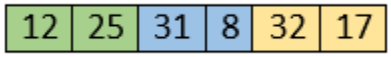


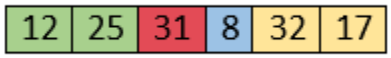
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

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.

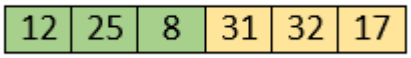


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

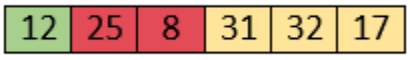




Both 31 and 8 are not sorted. So, swap them.

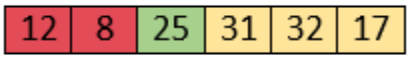


After swapping, elements 25 and 8 are unsorted.

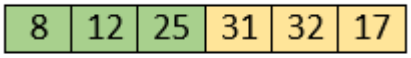


So, swap them.

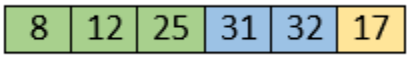
Now, elements 12 and 8 are unsorted.



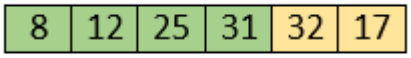
So, swap them too.



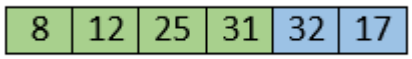
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



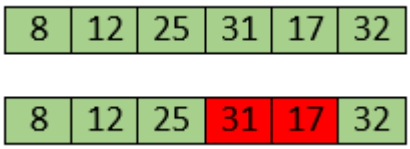
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



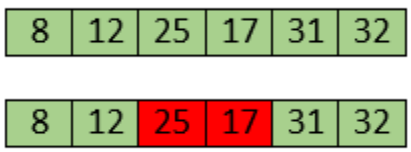
Move to the next elements that are 32 and 17.



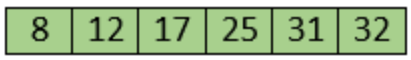
17 is smaller than 32. So, swap them.



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



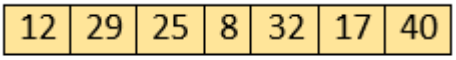
Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

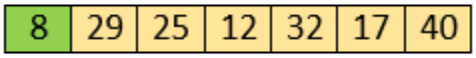
Selection Sort:

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.

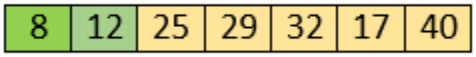
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.



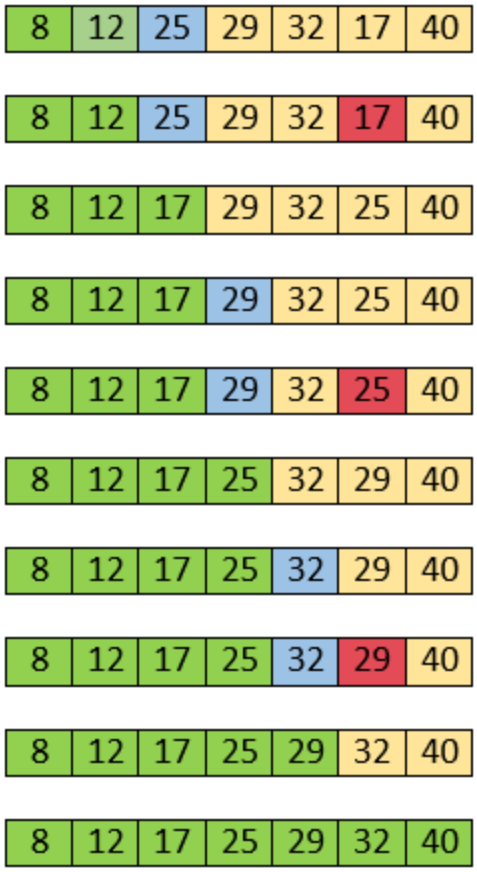
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

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.

Algorithm:

Insertion_sort(A,n)

```
For j = 2 to n
        Key = A[j]
        i = j-1
        while(i>=1 && a[i]>key)
                a[i+1] = A[i]
                i—
        A[i+1]=key
```

Selection_sort(A,n)

```
For i=1 to n-1
        Imin = i
        For j=i+1 to n
                If(A[min]>A[j])
                        Imin=j
Temp = A[imin]
A[imin] = A[i]
A[i] = Temp
```

Code:

Insertion Sort:

```
#include<stdio.h>
#include<time.h>
//insertion sort
void main ()
{
   int a[100000],i,j,temp,n;
   clock_t start, end;
    double cpu_time_used;
    start = clock();
   printf("Enter number of elements");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
     a[i]=n-i;              //worst case:- n-1,best case:- i, average case :- rand()
   }
   for(i=0;i<n;i++)
   {
      temp=a[i];
      j=i-1;
      while(j>=0 && a[j]>=temp)
      {
         a[j+1]=a[j];
         j--;
      }
      a[j+1]=temp;
   }
   // for(i=0;i<n;i++)
```

```
   // {
   //     printf("%d\t",a[i]);
   // }
   end = clock();
   cpu_time_used = ((double) (end - start));
   printf("time taken is %f",cpu_time_used);
}
```

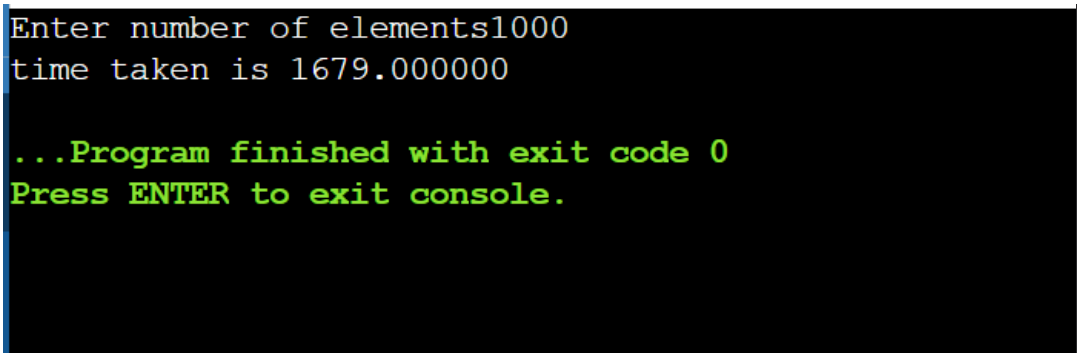
Selection Sort:

```
//selection sort
#include<stdio.h>
#include<time.h>

void main ()
{
   int a[50000],i,j,imin,temp,n;
    clock_t start, end;
    double cpu_time_used;
    start = clock();
   printf("Enter number of elements");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
      a[i]=rand();     //worst case:- n-1,best case:- i, average case :- rand()

   }
   for(i=0;i<n;i++)
   {
      imin=i;
      for(j=i+1;j<n;j++)
```

```
    {
      if(a[j]<a[imin])
      {
        a[imin]=a[j];
      }
    }
    temp=a[i];
    a[i]=a[imin];
    a[imin]=temp;
  }
  // for(i=0;i<n;i++)
  // {
  //    printf("%d\t",a[i]);
  // }
  end = clock();
  cpu_time_used = ((double) (end - start));
  printf("time taken is %f",cpu_time_used);
}
```

Output:

Insertion Sort:



Selection Sort:

```
main.c: In function 'main':
main.c:15:14: warning: implicit declaration of function 'rand' [-Wimplicit-function-declaration]
   15 |         a[i]=rand();     //worst case:- n-1,best case:- i, average case :- rand()
      |              ^~~~
Enter number of elements1000
time taken is 1247.000000

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

Conclusion:

Insertion Sort is a simple and efficient sorting algorithm that iterates over an array and places each element in its correct position in the sorted part of the array. It has a time complexity of O(n^2), making it suitable for small or partially sorted arrays.

Selection Sort is a simple and easy-to-implement sorting algorithm that iterates over an array, finds the smallest element in the unsorted part of the array, and places it at the beginning of the sorted part. It has a time complexity of O(n^2), making it suitable for small or partially sorted arrays.

AIM:

Implementing and analysing merge and quick sort in C

Theory:

Merge Sort

Merge sort is a divide-and-conquer algorithm that sorts an array by dividing it into two halves, recursively sorting each half, and then merging the two sorted halves into a single sorted array. The merge step involves comparing elements from the two sorted sub-arrays and placing them in order into a new array. This process is repeated until all elements are merged back into a single sorted array.

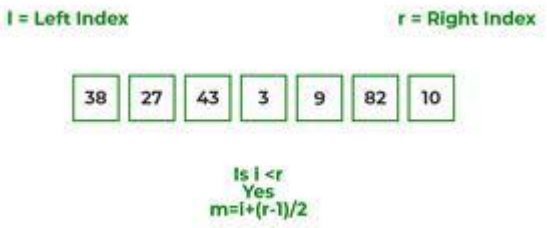
The time complexity of merge sort is O(nlogn) in the worst, average, and best cases. This is because the algorithm recursively divides the array into two halves, and then merges the sorted halves, with each merge operation taking O(n) time. While merge sort has a higher space complexity compared to other sorting algorithms, it is considered a stable sorting algorithm and is often used in external sorting algorithms where data is too large to fit in memory.

Algorithm

Example

To know the functioning of merge sort lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

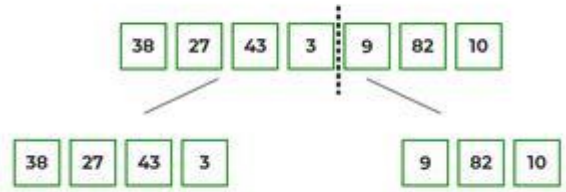
At first, check if the left index of array is less than the right index, if yes then calculate its mid point



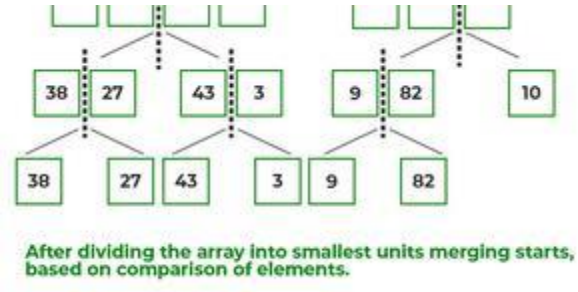
Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.

Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.

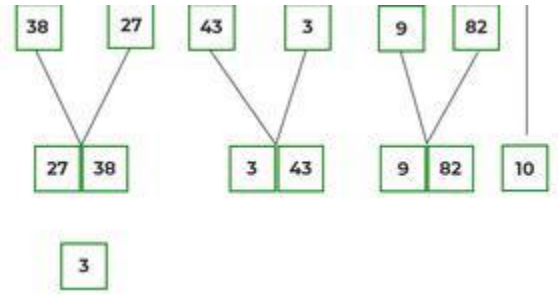


Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.

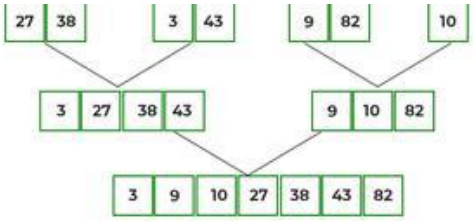


After dividing the array into smallest units, start merging the elements again based on comparison of size of elements

Firstly, compare the element for each list and then combine them into another list in a sorted manner.

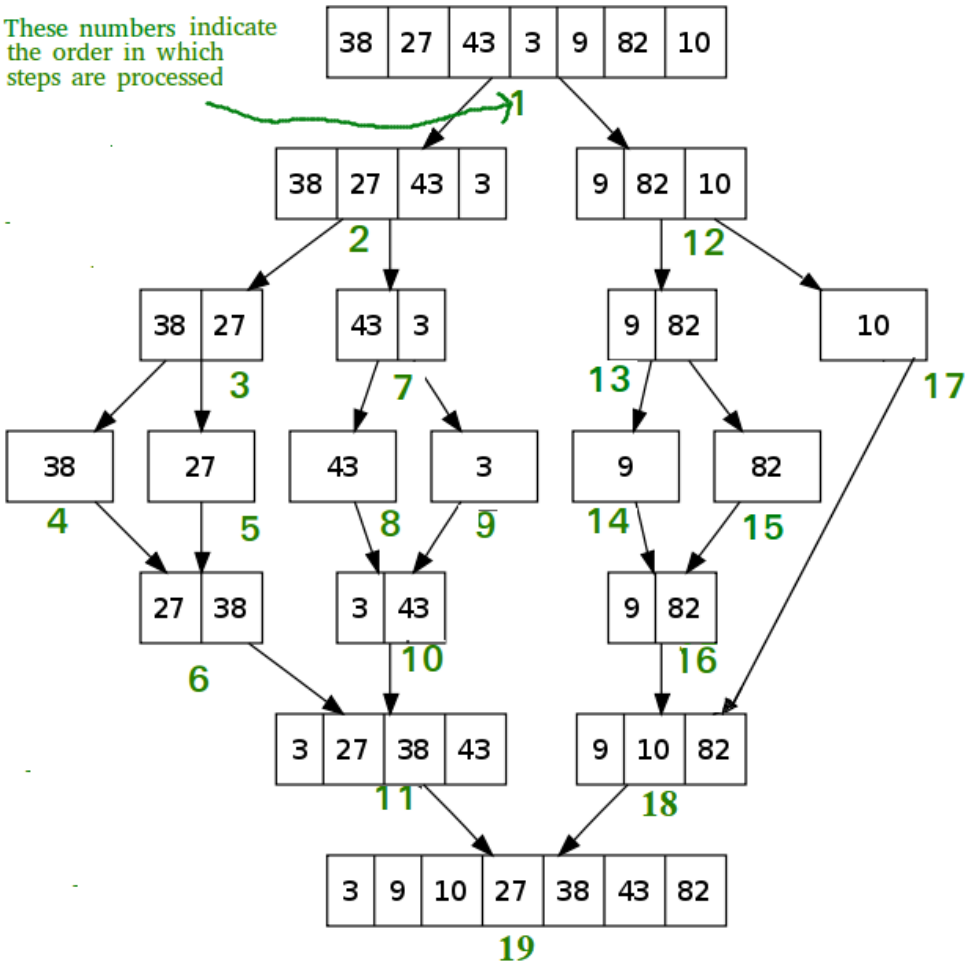


After the final merging, the list looks like this:



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Quick Sort

Quick sort is a comparison-based sorting algorithm that follows the divide-and-conquer approach. The algorithm works by partitioning the array into two parts around a pivot element, such that all elements on the left of the pivot are smaller than the pivot, and all elements on the right are larger. This is done recursively for each subarray, until the array is sorted.

The choice of the pivot element is crucial for the efficiency of the algorithm. Ideally, the pivot element should be the median of the subarray. However, finding the median is expensive and impractical. A commonly used approach is to choose the last element of the subarray as the pivot, but this can lead to worst-case performance if the input array is already sorted.

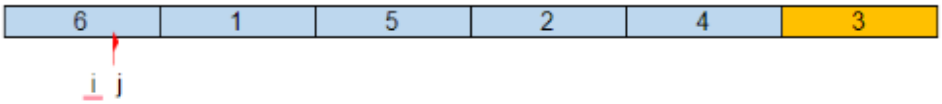
Quick sort has an average-case time complexity of O(nlogn), making it one of the fastest sorting algorithms in practice. However, the worst-case time complexity is O(n^2), which occurs when the pivot element is either the largest or smallest element of the subarray. To avoid this, several variants of quick sort have been developed, such as randomized quick sort, which randomly chooses the pivot element, and three-way quick sort, which handles arrays with many duplicate elements more efficiently.

Overall, quick sort is a versatile and efficient algorithm that is widely used in practice for sorting large datasets.

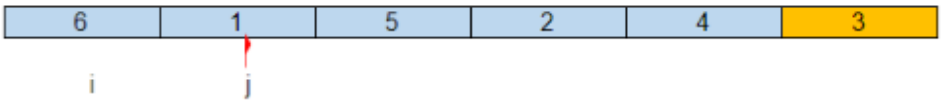
Example

Suppose the initial input array is:

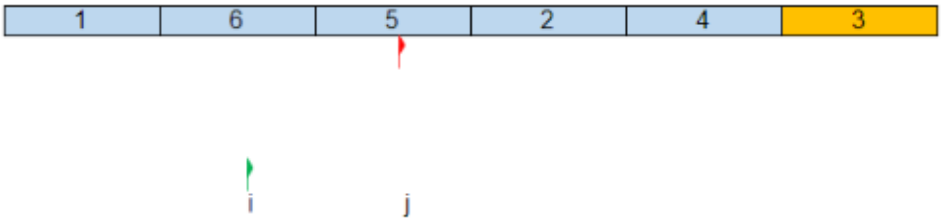
Here, 3 is chosen our pivot element and *partition()* function works as follows:



so a[j] is not <= pivot, we continue



now a[j] <= pivot so we swap a[i] and a[j], then increment i



similarly all other elements are traversed:

Now after j loop is over, we swap a[i] with the pivot

So after our first partition, the pivot element is at its correct place in the sorted array, with all its smaller elements to the left and greater to the right.

Now *QuickSort()* will be called recursively for the subarrays {1,2} and {6,4,5} and continue till the array is sorted.

Algorithm:

Merge Sort:

```
MergeSort(A,low,high)
    Mid=(low+high)/2
    MergeSort(A,low,mid)
    MergeSort(A,mid+1,high)
    Merge (A,low,mid,high)
Merge(A,low,mid,high)
    i=low
    j=mid+1
    k=low
    while(i<=mid && j<=high)
        if(A[i]<=A[j])
            B[k]=A[i]
            i++
        else
            B[k]=A[j]
            j++
        k++
    while(j<=high)
        B[k]=A[j]
        j++
        k++
    while(i<=mid)
        B[k]=A[i]
        i++
        k++
    for i=low to high
        A[i] = B[i]
```

Quick Sort:

```
QuickSort(A,low,high)
        If(low<high)
                P = Partition(A,low,high)
                Quicksort(A,low,P-1)
                Quicksort(A,P+1,high)
Partition(A,low,high)
        pivot = A[high]
        i=low
        j=low
        while(i<high)
                if(A[i]<pivot)
                        swap(A[i],A[j])
                        j++
                i++
        swap(A[j],A[high])
```

Code:

Merge Sort

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

void merge(int a[],int low,int mid,int high)
{
  int b[10000];
  int i=low;
  int j=mid+1;
  int k=low;
  while(i<=mid && j<=high)
  {
```

```
    if (a[i]<a[j])
    {
        b[k]=a[i];
        i++;
    }
    else
    {
        b[k]=a[j];
        j++;
    }
    k++;
}
if(i>mid)
{
    while(j<=high)
    {
        b[k]=a[j];
        j++;
        k++;
    }
}
if(j>high)
{
    while(i<=mid)
    {
        b[k]=a[i];
        i++;
        k++;
    }
}
```

```
  for(i=low;i<=high;i++)
  {
     a[i]=b[i];
  }
}
void merge_sort(int a[], int low,int high)
{
  if(low<high)
  {    int mid;
     mid=(low+high)/2;
     merge_sort(a,low,mid);
     merge_sort(a,mid+1,high);
     merge(a,low,mid,high);

  }

}

void main ()
{
   clock_t start,end;
   double time1;
   start=clock();
   int a[10000],low,high,n;
   printf("Enter number of elements");
   scanf("%d",&n);
   for(int i=0;i<n;i++)
   {
      // printf("Enter value");
      // scanf("%d",&a[i]);
```

```
        a[i]=rand();
    }

    low=0;
    high=n-1;
    merge_sort(a,low,high);
    // for(int i=0;i<n;i++)
    // {
    //     printf("%d\t",a[i]);
    // }
    end=clock();
    time1=(double)(end-start);
    printf("\n time taken is %f",time1);

}
```

Quick Sort

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

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
```

```
    for (int 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 quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

int main() {
    srand(time(NULL));
    int arr[1000],n;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    // printf("Unsorted array:\n");
    // for (int i = 0; i < 1000; i++) {
    //     printf("%d ", arr[i]);
```

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

    clock_t start = clock(); // Start time measurement
    quick_sort(arr, 0, n-1);
    clock_t end = clock(); // End time measurement
    double elapsed_time = ((double) (end - start)) / CLOCKS_PER_SEC;
    // printf("Sorted array:\n");
    // for (int i = 0; i < 1000; i++) {
    //     printf("%d ", arr[i]);
    // }
    printf("\n");
    printf("Time taken: %f seconds\n", elapsed_time);
    return 0;
}
```

Output:

Merge Sort

```
main.c: In function 'main':
main.c:72:14: warning: implicit declaration of function 'rand' [-Wimplicit-function-declaration]
   72 |         a[i]=rand();
      |              ^~~~
Enter number of elements1000

 time taken is 239.000000

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

Quick Sort

```
Enter number of elements: 1000

Time taken: 0.000098 seconds


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

Conclusion:

In summary, merge sort is a stable, comparison-based algorithm with time complexity O(nlogn) and is ideal for large datasets. Quick sort is faster for small to medium-sized datasets, has average time complexity O(nlogn), but is not stable and can have performance issues with certain data distributions.

Aim:

To implement Min-Max and Binary Search in an array using Divide & Conquer Approach.

Theory:

Min-Max: The divide and conquer approach is an efficient way to find the minimum and maximum values in an array. The algorithm works by dividing the array into two halves, recursively finding the minimum and maximum values in each half, and then combining the results to obtain the minimum and maximum values for the entire array. The time complexity of the divide and conquer algorithm for finding the minimum and maximum values in an array is O(n log n), which is much faster than the simple linear search algorithm that has a time complexity of O(n^2).The idea is to recursively divide the array into two equal parts and update the maximum and minimum of the whole array in recursion by passing minimum and maximum variables by reference. The base conditions for the recursion will be when the subarray is of length 1 or 2. It is an important tool for many practical applications and is a fundamental concept in computer science and algorithm design.

Divide: Divide array into two halves.

Conquer: Recursively find maximum and minimum of both halves.

Combine: Compare maximum of both halves to get overall maximum and compare minimum of both halves to get overall minimum.

Binary Search:

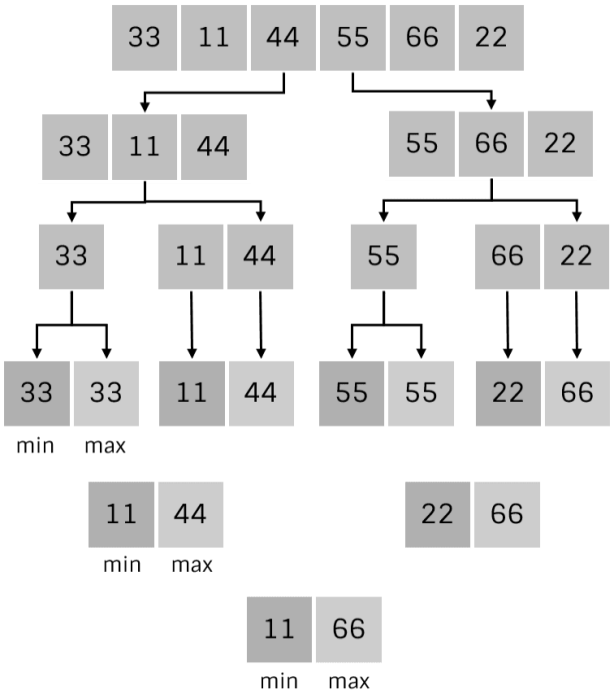
Binary search using the divide and conquer algorithm is an efficient and widely used algorithm for finding a specific element in a sorted array. The algorithm works by repeatedly dividing the search interval in half until the target element is found or it is determined that the element is not in the array. The divide and conquer approach reduces the search interval by half at each step, resulting in a time complexity of O(log n), where n is the size of the input array.

In practice, binary search is used in a wide range of applications, such as searching a database, searching for a word in a dictionary, or finding a specific item in a sorted list of items. The divide and conquer algorithm provides a simple and elegant way to perform the search, and its time complexity makes it an attractive option for large datasets.

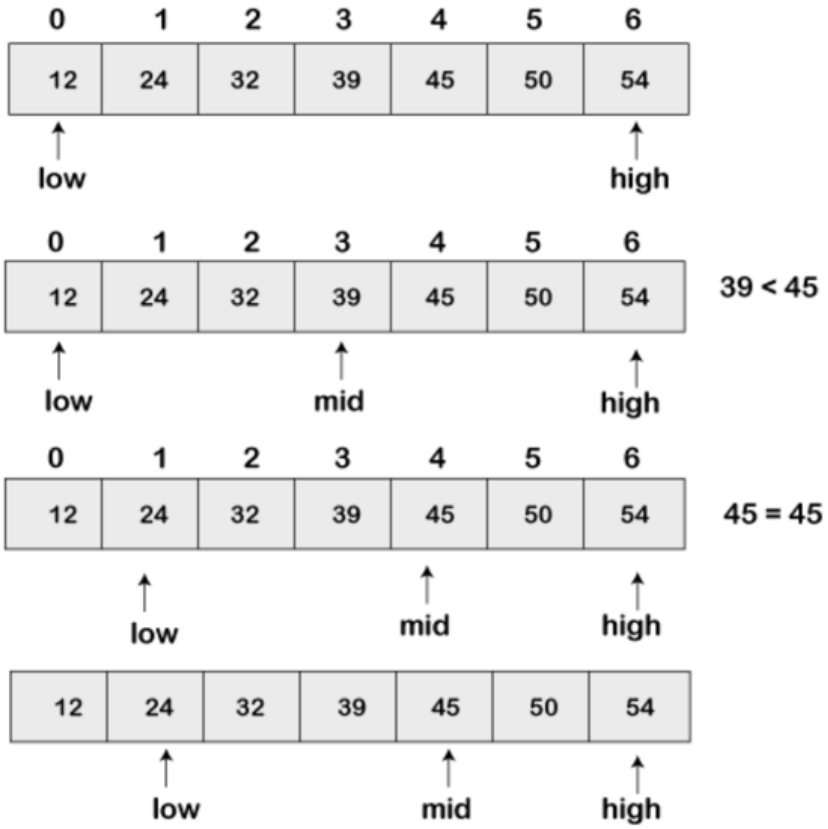
However, it is important to note that binary search only works on sorted arrays. If the array is not sorted, the algorithm will not work correctly. Additionally, if the target element appears multiple times in the array, the algorithm may return any of the indices where the element appears, not necessarily the first or last occurrence. Overall, binary search using the divide and conquer algorithm is a powerful tool for searching sorted arrays and has numerous practical applications.

Example:

Min-Max:



Binary Search:



Algorithm:

Min_Max(A,i,j,max,min)

```
If(i==j)
        max=min=A[i]
else if(i=j-1)
        if A[i]<A[j]
                min=A[i]
                max=A[j]
        else
                max=A[i]
                min=A[j]
else
        mid=(i+j)/2
        Min_Max(A,i,mid,max,min)
        Min_Max(A,mid+1,j,max1,min1)
        If max1>max
                max=max1
        If min1<min
                min=min1
```

Binary_Search(A,low,high,X)

```
If(low==high)
        If(A[low]==X)
                Return low
        Else
                Return -1
Else
        Mid=(low+high)/2
        If(A[mid]==X)
                Return mid
        Else if(A[mid]>X)
                Binary_Search(A,low,mid-1,X)
        Else
                Binary_Search(A,mid+1,high,X)
```

Code:

Min-Max:

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

int max, min;
int a[100];

void maxmin(int i, int j) {
   int max1, min1, mid;
   if(i == j) {
      max = min = a[i];
   }
   else {
      if(i == j-1) {
         if(a[i] < a[j]) {
            max = a[j];
            min = a[i];
         }
         else {
            max = a[i];
            min = a[j];
         }
      }
      else {
         mid = (i+j)/2;
         maxmin(i, mid);
         max1 = max;
         min1 = min;
```

```
            maxmin(mid+1, j);
            if(max < max1)
               max = max1;
            if(min > min1)
               min = min1;
         }
      }
   }

int main () {
   int i, num;
   clock_t start, end;
   double cpu_time_used;
   start = clock();

   printf("\nEnter the total number of numbers : ");
   scanf("%d", &num);
   printf("Randomly generated array of size %d : \n", num);
   srand(time(NULL));
   for(i = 0; i < num; i++) {
      a[i] = rand() % 100 + 1;
      // printf("%d ", a[i]);
   }

   max = a[0];
   min = a[0];
   maxmin(0, num-1);
   printf("\nMinimum element in the array : %d\n", min);
   printf("Maximum element in the array : %d\n", max);
   end = clock();
```

```
    cpu_time_used = (double)(end-start);
    printf("\ntime taken: %f",cpu_time_used);
    return 0;
}
```

Binary Search:

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

int binarySearch(int arr[], int x, int low, int high)
{int mid;
        if (low > high)
            return -1;

        else
            mid = (low + high) / 2 ;
            if (arr[mid] == x)
                return mid;

            else if (x > arr[mid])
                return binarySearch(arr, x, mid + 1, high);

            else
                return binarySearch(arr, x, low, mid - 1) ;
}
void main(){
    int a[10000], n,i,x,ans;
    clock_t start, end;
```

```
    double cpu_time_used;
    start = clock();
    printf("\nenter number of elements: ");
    scanf("%d",&n);
    //printf("\nelements: ");
    for(i=0;i<n;i++){
    // scanf("%d",&a[i]);
    a[i] = i;
    }
    printf("\nenter element to be searched: ");
    scanf("%d",&x);
    ans = binarySearch(a, x, 0, n-1);
    if(ans == -1)
        printf("\nnot found");
    else
        printf("\n%d found at %d",x,ans);
        end = clock();
        cpu_time_used = (double)(end-start);
        printf("\ntime taken: %f",cpu_time_used);
}
```

Output:

Min-Max:

```
Enter the total number of numbers : 1000
Randomly generated array of size 1000 :

Minimum element in the array : 1
Maximum element in the array : 100

time taken: 130.000000

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

Binary Search:

```
enter number of elements: 1000

enter element to be searched: 500

500 found at 500
time taken: 119.000000

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

Conclusion:

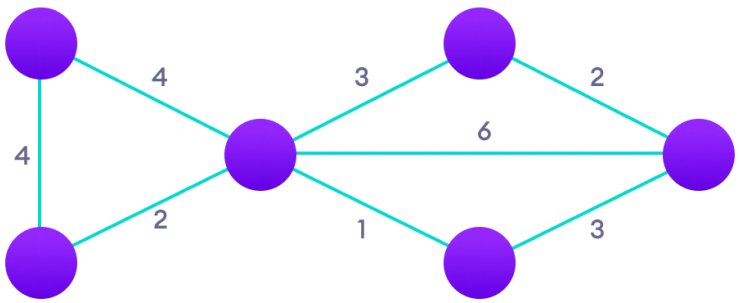
The Divide and Conquer Approach is used to easily calculate Min-Max in an array or perform Binary Search.

Aim: To implement Single Source Shortest Path using Greedy Approach (Dijkstra)

Theory: Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

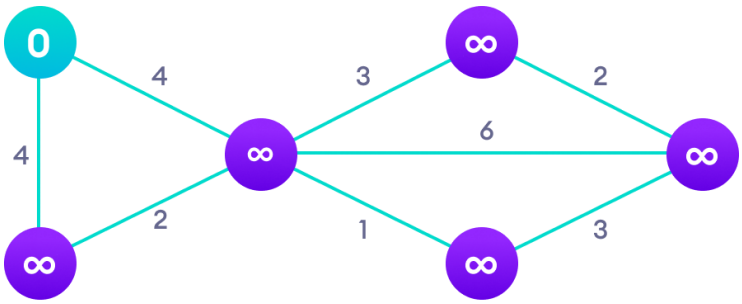
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Example: For a given graph



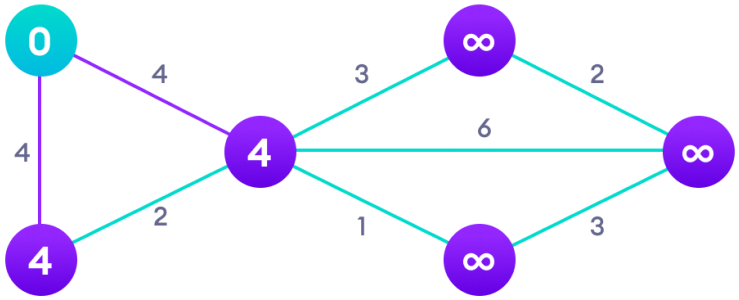
Step: 1

Start with a weighted graph



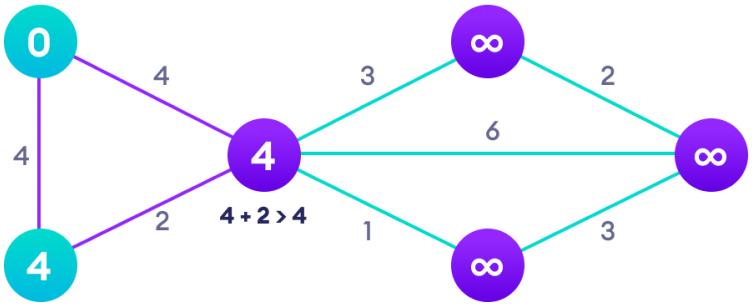
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



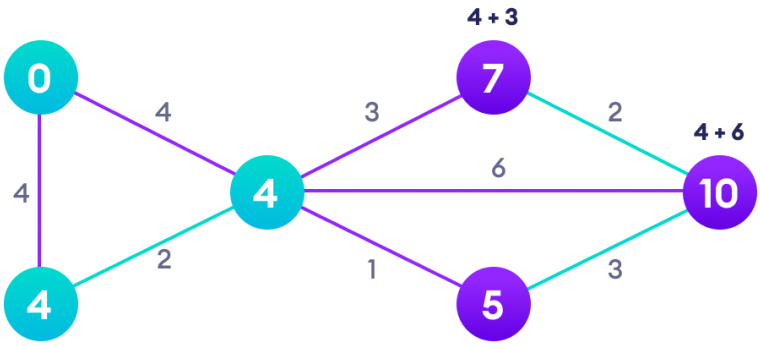
Step: 3

Go to each vertex and update its path length



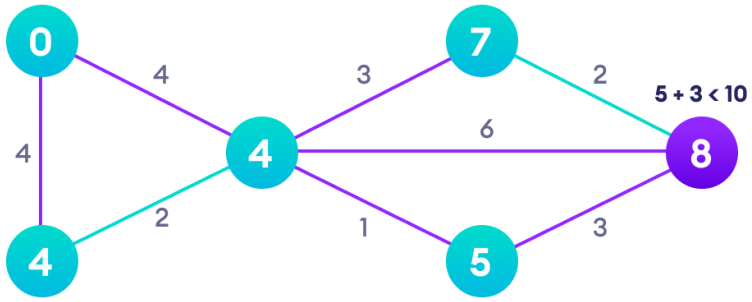
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



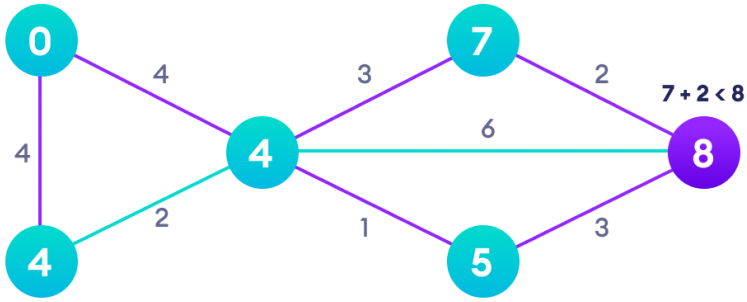
Step: 5

Avoid updating path lengths of already visited vertices



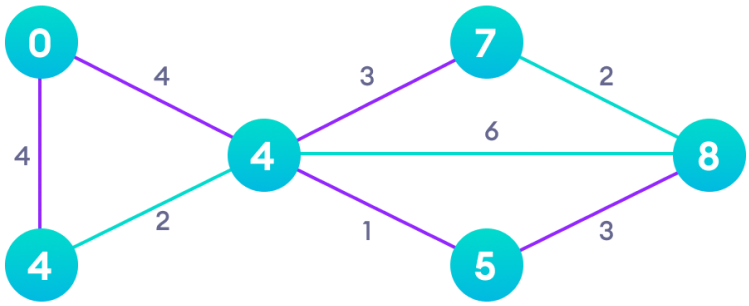
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Algorithm:

```
Dijkstra(G,w,s)
        Initialize_single_source(G,s)
        S = Φ
        Q = G.V
        While(Q!= Φ)
                u = Extract_min(Q)
                S = S U {u}
                For each vertex v of G Adj[u]
                        Relax(u,v,w)
Initialize_single_source(G,s)
        For each vertex v of G.V
                v.d = ∞
                v. π = NIL
Relax(u,v,w)
        If v.d > u.d + w(u,v)
                v.d = u.d + w(u,v)
                v. π = u
```

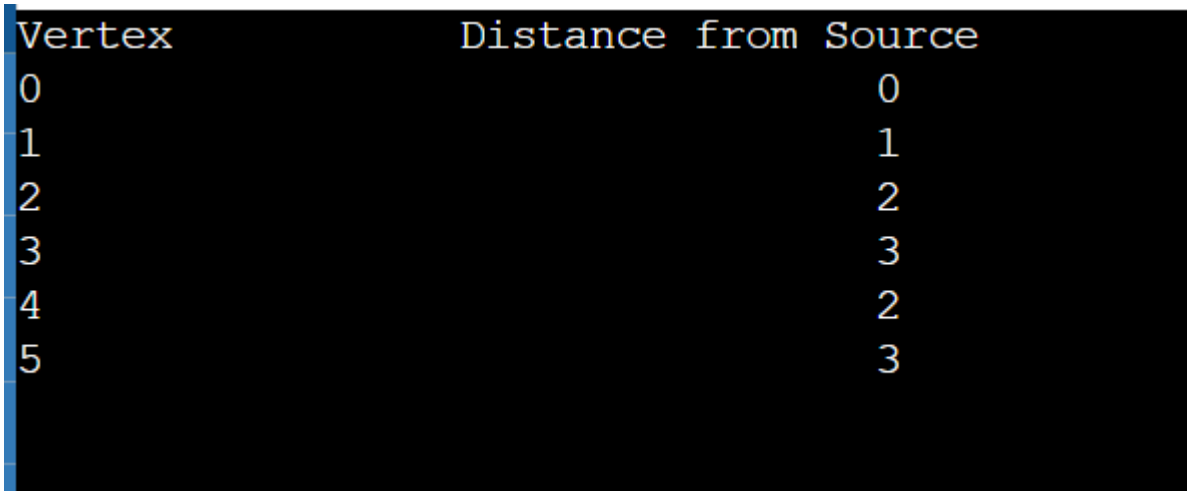
Code: //dijkstra algo shortest distance from source

```
#include <stdio.h>
#include<stdbool.h>
#include <limits.h>
int minDistance(int dist[], bool sptSet[])
{
   int min = INT_MAX, min_index;
```

```
    for (int v = 0; v < 6; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}
void dijkstra(int graph[6][6],int src){
    int dist[6];
    bool sptSet[6];
    for (int i = 0; i < 6; i++)
        {dist[i] = INT_MAX;
        sptSet[i] = false;}

    dist[src] = 0;

    for (int count = 0; count < 5; count++) {

        int u = minDistance(dist, sptSet);

        sptSet[u] = true;

        for (int v = 0; v < 6; 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];
    }

    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < 6; i++)
        printf("%d \t\t\t\t %d\n", i, dist[i]);
}



void main()
{
    int graph[6][6]={{0,1,4,0,0,0},
                {0,0,1,2,1,0},
                {0,0,0,0,5,0},
                {0,0,0,0,0,2},
                {0,0,0,0,0,1},
                {0,0,0,0,0,0}};
    dijkstra(graph,0);
}
```

Output:



Conclusion: Thus we have implemented shortest path of each node from the source using Greedy Approach (Dijkstra's Algorithm)

Aim- To write a code and run it for minimum spanning tree suing Prim's and Kruskal's Algorithm

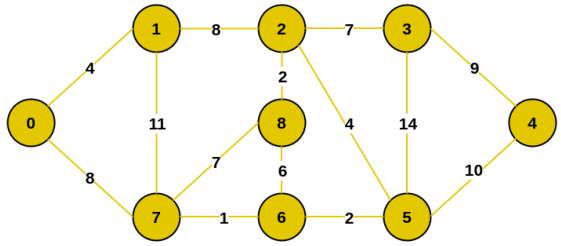
Theory- In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first at the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way. A group of edges that connects two sets of vertices in a graph is called cut in graph theory. So, at every step of Prim's algorithm, find a cut, pick the minimum weight edge from the cut, and include this vertex in MST Set (the set that contains already included vertices).

Example-

Prim's-

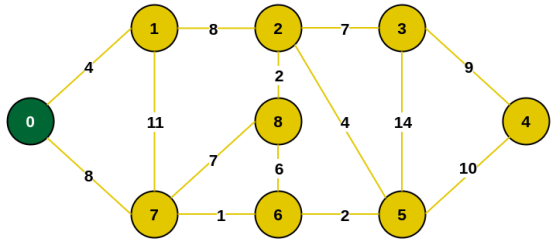
Consider the following graph as an example for which we need to find the Minimum Spanning Tree (MST).



Example of a Graph

Example of a graph

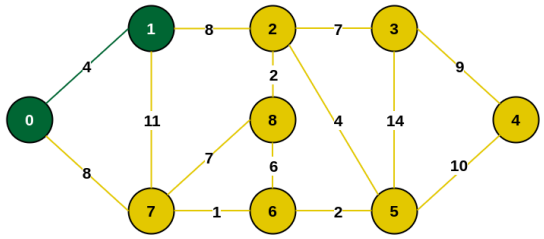
Step 1: Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex 0 as the starting vertex.



Select an arbitrary starting vertex. Here we have selected 0

0 is selected as starting vertex

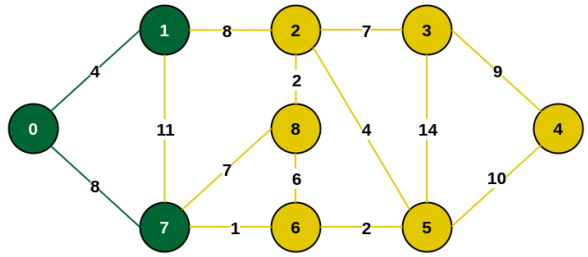
Step 2: All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.



Minimum weighted edge from MST to other vertices is 0-1 with weight 4

1 is added to the MST

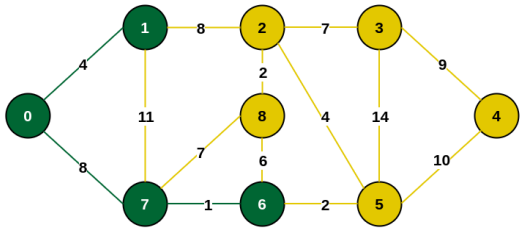
Step 3: The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].



Minimum weighted edge from MST to other vertices is 0-7 with weight 8

7 is added in the MST

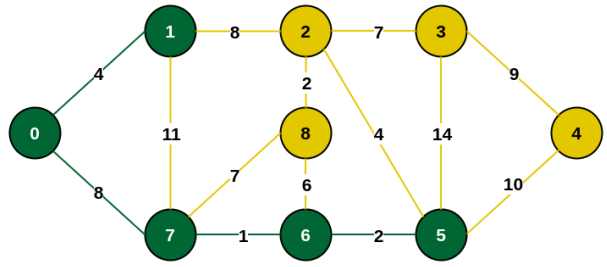
Step 4: The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).



Minimum weighted edge from MST to other vertices is 7-6 with weight 1

6 is added in the MST

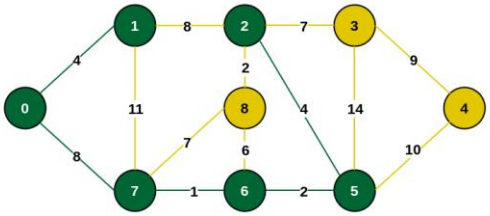
Step 5: The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.



Minimum weighted edge from MST to other vertices is 6-5 with weight 2

Include vertex 5 in the MST

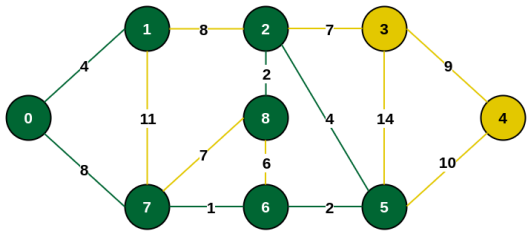
Step 6: Among the current connecting edges, the edge {5, 2} has the minimum weight. So include that edge and the vertex 2 in the MST.



Minimum weighted edge from MST to other vertices is 5-2 with weight 4

Include vertex 2 in the MST

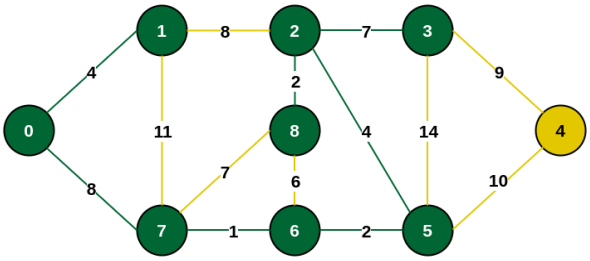
Step 7: The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.



Minimum weighted edge from MST to other vertices is 2-8 with weight 2

Add vertex 8 in the MST

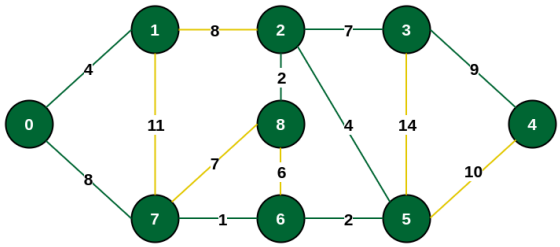
Step 8: See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.



Minimum weighted edge from MST to other vertices is 2-3 with weight 7

Include vertex 3 in MST

Step 9: Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4

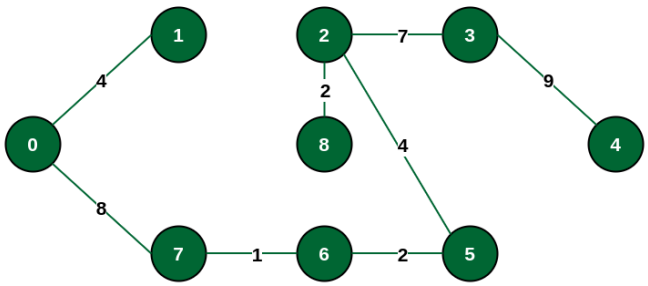


Minimum weighted edge from MST to other vertices is 3-4 with weight 9

is {3, 4}.

Include vertex 4 in the MST

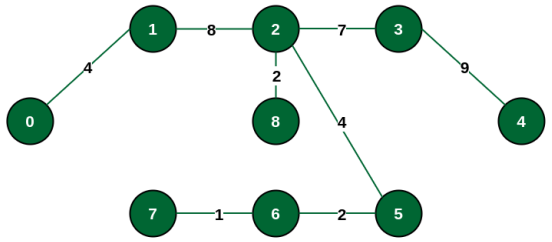
The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = 37.



The final structure of MST

The structure of the MST formed using the above method

Note: If we had selected the edge {1, 2} in the third step then the MST would look like the following.

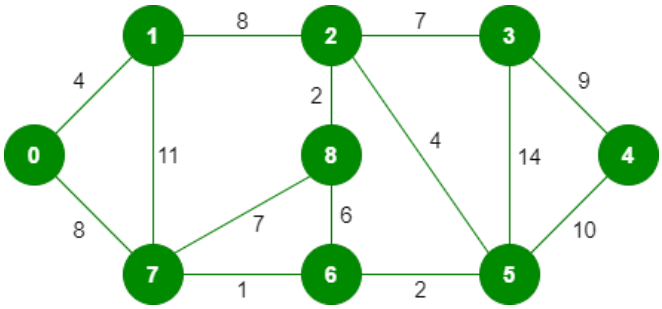


Alternative MST structure

Structure of the alternate MST if we had selected edge {1, 2} in the MST

Kruskal's-

Input Graph:



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

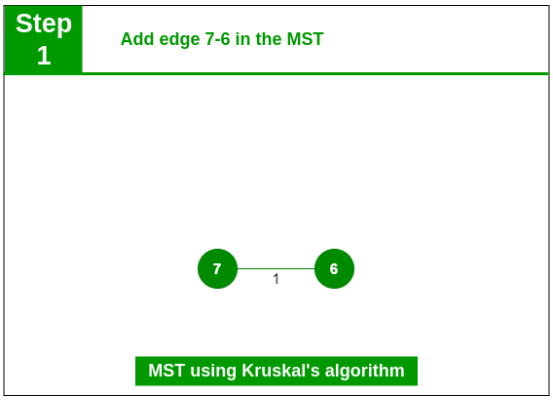
After sorting:

Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5

6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

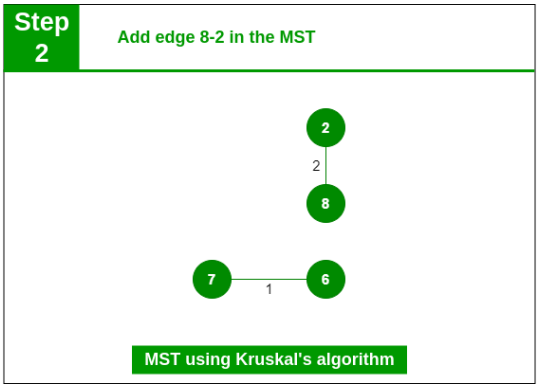
Now pick all edges one by one from the sorted list of edges

Step 1: Pick edge 7-6. No cycle is formed, include it.



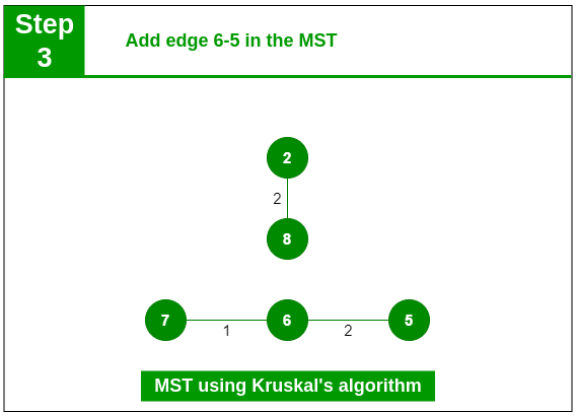
Add edge 7-6 in the MST

Step 2: Pick edge 8-2. No cycle is formed, include it.



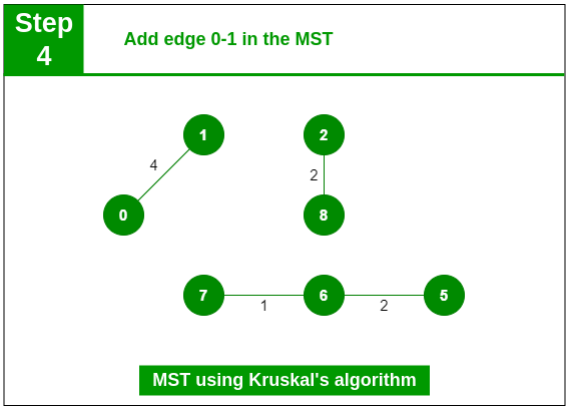
Add edge 8-2 in the MST

Step 3: Pick edge 6-5. No cycle is formed, include it.



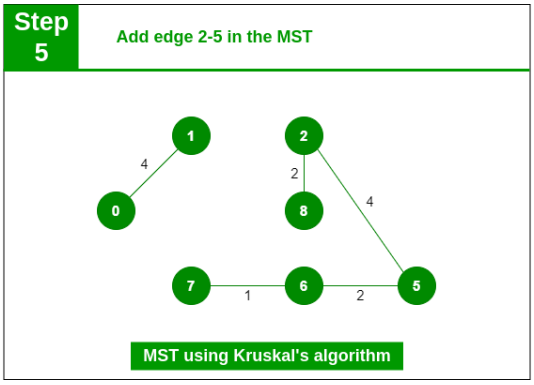
Add edge 6-5 in the MST

Step 4: Pick edge 0-1. No cycle is formed, include it



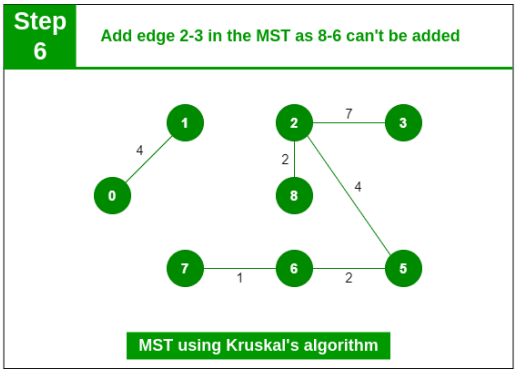
Add edge 0-1 in the MST

Step 5: Pick edge 2-5. No cycle is formed, include it.



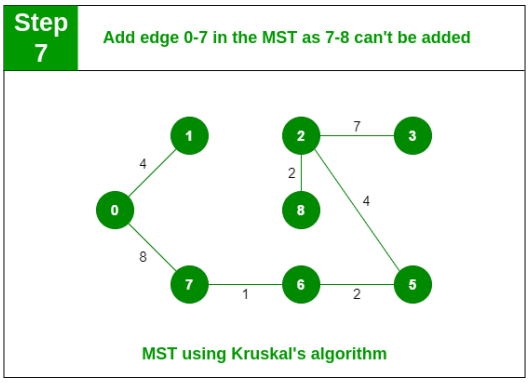
Add edge 2-5 in the MST

Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.



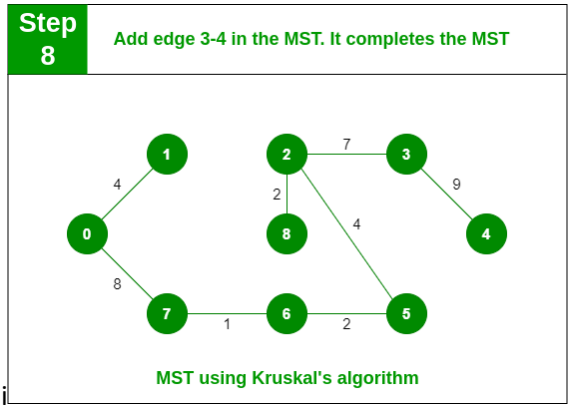
Add edge 2-3 in the MST

Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.



Add edge 0-7 in MST

Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle



i

Add edge 3-4 in the MST

Note: Since the number of edges included in the MST equals to (V – 1), so the algorithm stops here

Algorithm-

Prim's-

MST-PRIM (G, w, r)

1. for each u ∈ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. While Q? Ø
7. do u ← EXTRACT - MIN (Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u, v) < key [v]
10. then π [v] ← u
11. key [v] ← w (u, v)

Kruskal's-

MST- KRUSKAL (G, w)

1. A ← Ø
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non-decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
7. then A ← A ∪ {(u, v)}
8. UNION (u, v)
9. return A

Code-

Prim's

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

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
   // Initialize min value
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
      if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;

   return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d \t%d \n", parent[i], i,
            graph[i][parent[i]]);
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
   // Array to store constructed MST
   int parent[V];
   // Key values used to pick minimum weight edge in cut
   int key[V];
   // To represent set of vertices included in MST
   bool mstSet[V];

   // Initialize all keys as INFINITE
   for (int i = 0; i < V; i++)
      key[i] = INT_MAX, mstSet[i] = false;
```

```
    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first
    // vertex.
    key[0] = 0;

    // First node is always root of MST
    parent[0] = -1;

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {

        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
            // vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only
            // if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false
                && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

// Driver's code
int main()
{
    int graph[V][V] = { { 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 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

Kruskal's

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

// Comparator function to use in sorting
int comparator(const void* p1, const void* p2)
{
   const int(*x)[3] = p1;
   const int(*y)[3] = p2;

   return (*x)[2] - (*y)[2];
}

// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
   for (int i = 0; i < n; i++) {
      parent[i] = i;
      rank[i] = 0;
   }
}

// Function to find the parent of a node
int findParent(int parent[], int component)
{
   if (parent[component] == component)
      return component;

   return parent[component]
         = findParent(parent, parent[component]);
}

// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
   // Finding the parents
   u = findParent(parent, u);
   v = findParent(parent, v);

   if (rank[u] < rank[v]) {
      parent[u] = v;
   }
   else if (rank[u] > rank[v]) {
      parent[v] = u;
   }
   else {
      parent[v] = u;

      // Since the rank increases if the ranks of two sets are same
```

```
      rank[u]++;
   }
}

// Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
   // First we sort the edge array in ascending order
   // so that we can access minimum distances/cost
   qsort(edge, n, sizeof(edge[0]), comparator);

   int parent[n];
   int rank[n];

   // Function to initialize parent[] and rank[]
   makeSet(parent, rank, n);

   // To store the minimun cost
   int minCost = 0;

   printf("Following are the edges in the constructed MST\n");
   for (int i = 0; i < n; i++) {
      int v1 = findParent(parent, edge[i][0]);
      int v2 = findParent(parent, edge[i][1]);
      int wt = edge[i][2];

      // If the parents are different that
      // means they are in different sets so
      // union them
      if (v1 != v2) {
         unionSet(v1, v2, parent, rank, n);
         minCost += wt;
         printf("%d -- %d == %d\n", edge[i][0],
               edge[i][1], wt);
      }
   }

   printf("Minimum Cost Spanning Tree: %d\n", minCost);
}
int main()
{
   int edge[5][3] = { { 0, 1, 10 },
                  { 0, 2, 6 },
                  { 0, 3, 5 },
                  { 1, 3, 15 },
                  { 2, 3, 4 } };

   kruskalAlgo(5, edge);

   return 0;
}
```

Output-

Prim's

```
Output

/tmp/d8kpQJGQyl.o
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

Kruskal's

```
Output

/tmp/d8kpQJGQyl.o
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

Conclusion-Thus we have used an example to illustrate Minimum spanning tree using Prim's and Kruskal's algorithm

AIM:- To implement longest common subsequence.

THEORY:-

A common subsequence of two or more strings is a sequence that appears in the same order in each of the strings, but not necessarily consecutively. For example, given the strings "ABCD" and "ACDF", the common subsequence "ACD" appears in both strings. The longest common subsequence (LCS) of two or more strings is the longest subsequence that is common to all the strings. For example, given the strings "ABCD" and "ACDF", the LCS is "ACD".

To find the LCS of two strings, we can use dynamic programming. We first create a table with one row and one column for each string, and fill in the cells according to the following rules:

If the two characters being compared are the same, the value in the current cell is the value in the diagonal cell plus one.Otherwise, the value in the current cell is the maximum of the value in the cell to the left and the value in the cell above.We start at the top left corner of the table and work our way down to the bottom right corner. The value in the bottom right corner of the table is the length of the LCS.

To find the LCS itself, we can backtrack through the table from the bottom right corner to the top left corner, following the same rules as above. If the value in the current cell is equal to the value in the cell to the left or the cell above, we move in that direction.If the value in the current cell is equal to the value in the diagonal cell plus one, we add the corresponding character to the LCS and move diagonally.

The algorithm for finding the LCS of two strings using dynamic programming involves creating a table to store the length of the LCS of prefixes of the two input strings. This table is initialized with zeros, and then filled in using dynamic programming. The length of the LCS can be read off from the bottom-right corner of the table. To find the actual LCS, we can backtrack through the table starting from this corner, following a path of cells with the same value until we reach the top-left corner. Along the way, we can record the characters corresponding to the cells visited. Finally, we reverse the order of these characters to obtain the final answer. The time complexity of this algorithm is O(nm), where n and m are the lengths of the two input strings, and the space complexity is also O(nm).

Example:

*Say the strings are **S1 = "AGGTAB"** and **S2 = "GXTXAYB"**.*

First step: *Initially create a 2D matrix (say dp[][]) of size 8 x 7 whose first row and first column are filled with 0.*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0							
G	2	0							
G	3	0							
T	4	0							
A	5	0							
B	6	0							

Creating dp table and filling 0 in 0th row and column

Creating the dp table

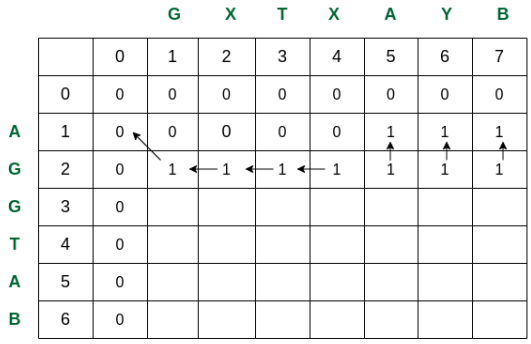
Second step: *Traverse for i = 1. When j becomes 5, S1[0] and S2[4] are equal. So the dp[][] is updated. For the other elements take the maximum of dp[i-1][j] and dp[i][j-1]. (In this case, if both values are equal, we have used arrows to the previous rows).*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0							
G	3	0							
T	4	0							
A	5	0							
B	6	0							

Updating dp table for row 1

Filling the row no 1

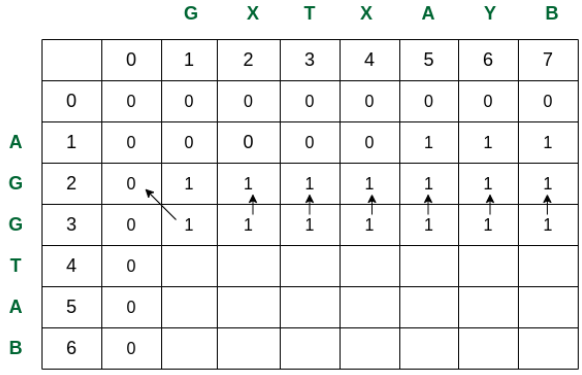
Third step: *While traversed for i = 2, S1[1] and S2[0] are the same (both are 'G'). So the dp value in that cell is updated. Rest of the elements are updated as per the conditions.*



Updating dp table for row 2

Filling the row no. 2

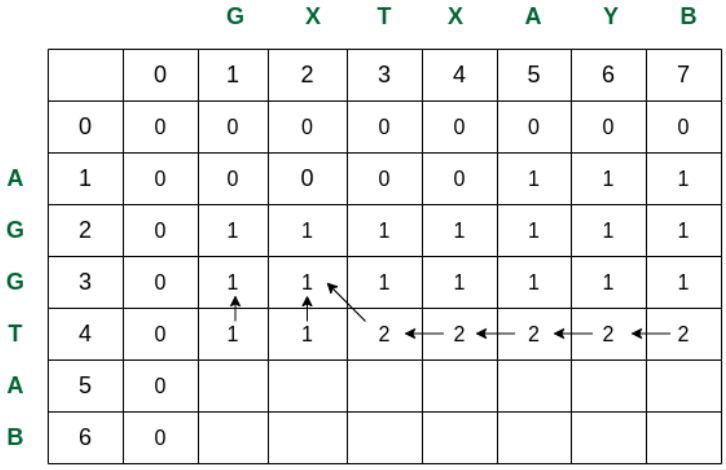
Fourth step: *For i = 3, S1[2] and S2[0] are again same. The updations are as follows.*



Updating dp table for row 3

Filling row no. 3

Fifth step: *For i = 4, we can see that S1[3] and S2[2] are same. So dp[4][3] updated as dp[3][2] + 1 = 2.*



Updating dp table for row 4

Filling row 4

Sixth step: *Here we can see that for i = 5 and j = 5 the values of S1[4] and S2[4] are same (i.e., both are 'A'). So dp[5][5] is updated accordingly and becomes 3.*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0	1	1	2	2	3	3	3
B	6	0							

Updating dp table for row 5

Filling row 5

Final step: *For i = 6, see the last characters of both strings are same (they are 'B'). Therefore the value of dp[6][7] becomes 4.*

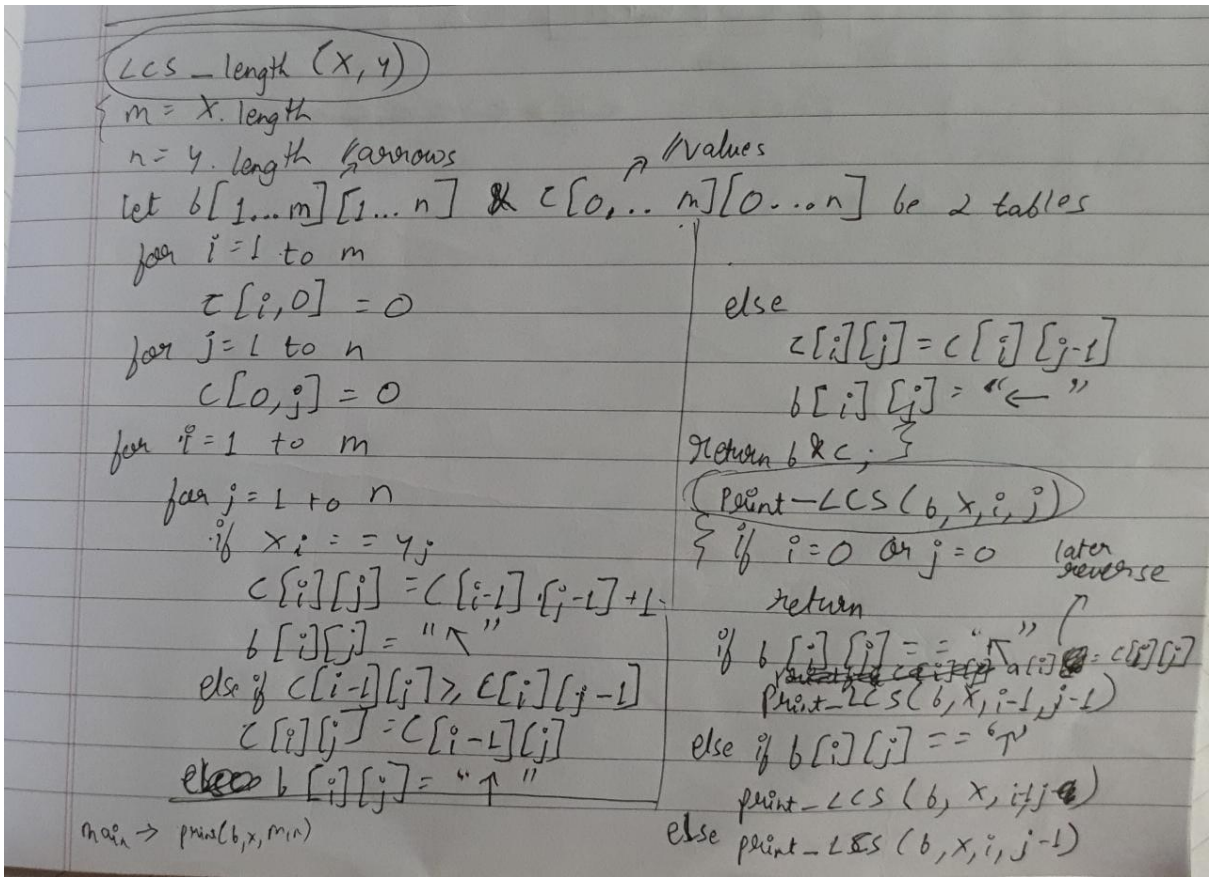
			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0	1	1	2	2	3	3	3
B	6	0	1	1	2	2	3	3	4

Updating dp table for row 6

Filling the final row

So we get the maximum length of common subsequence as ***4****.*

ALGORITHM:



CODE:-

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

int i, j, m, n, LCS_table[20][20];
char S1[20] , S2[20] , b[20][20];
void input()
{
   printf("Enter two strings");
   scanf("%s",S1);
   scanf("%s",S2);
}
void lcsAlgo() {
```

```
 m = strlen(S1);
 n = strlen(S2);

 // Filling 0's in the matrix
 for (i = 0; i <= m; i++)
  LCS_table[i][0] = 0;
 for (i = 0; i <= n; i++)
  LCS_table[0][i] = 0;

 for (i = 1; i <= m; i++)
  for (j = 1; j <= n; j++) {
   if (S1[i - 1] == S2[j - 1]) {
    LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
   } else if (LCS_table[i - 1][j] >= LCS_table[i][j - 1]) {
    LCS_table[i][j] = LCS_table[i - 1][j];
   } else {
    LCS_table[i][j] = LCS_table[i][j - 1];
   }
  }

 int index = LCS_table[m][n];
 char lcsAlgo[index + 1];
 lcsAlgo[index] = '\0';

 int i = m, j = n;
 while (i > 0 && j > 0) {
  if (S1[i - 1] == S2[j - 1]) {
   lcsAlgo[index - 1] = S1[i - 1];
   i--;
   j--;
   index--;
```

```
    }

    else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
      i--;
    else
      j--;
  }

  // Printing the sub sequences
  printf("S1 : %s \nS2 : %s \n", S1, S2);
  printf("LCS: %s", lcsAlgo);
}

int main() {
  input();
  lcsAlgo();
  printf("\n");
}
```

OUTPUT:-

```
/tmp/dK3pxAdZmx.o
Enter two stringsABCDEFGH
ABHJK
S1 : ABCDEFGH
S2 : ABHJK
LCS: ABH
```

CONCLUSION:-

The longest common subsequence (LCS) is the longest sequence of characters that appears in the same order in both input strings. The LCS problem can be solved using dynamic programming, which involves creating a table to store the length of the LCS of prefixes of the two input strings. The algorithm has a time complexity of O(nm) and a space complexity of O(nm), where n and m are the lengths of the two input strings.The LCS problem has applications in areas such as bioinformatics, text comparison, and data compression.

Aim: Bellman-Ford algorithm is to find the shortest path between a source vertex and all other vertices in a weighted directed graph, where the weights can be negative. The algorithm achieves this by iteratively relaxing the edges in the graph and updating the distances of the vertices until the shortest path is found.

Theory:

The Bellman-Ford algorithm is based on the principle of dynamic programming, which involves breaking a problem down into smaller subproblems and solving them in a bottom-up manner.

The algorithm maintains an array of distances d[] for each vertex in the graph, where d[s] is the distance from the source vertex s to itself, and d[v] is the distance from the source vertex s to the vertex v.

The algorithm then iterates over all the edges in the graph for |V|-1 times, where |V| is the number of vertices in the graph. In each iteration, the algorithm relaxes all the edges in the graph and updates the distances of the vertices if a shorter path is found.

The relaxation of an edge (u,v) involves comparing the distance of the source vertex s to u plus the weight of the edge (u,v) with the current distance of the source vertex s to v. If the former is smaller than the latter, the distance of v is updated to the former.

The algorithm terminates when all the vertices have been relaxed for |V|-1 times or when no more updates can be made to the distances of the vertices. If the algorithm terminates after |V|-1 iterations, the shortest path has been found. If the algorithm terminates with an update in the |V|-th iteration, there exists a negative cycle in the graph.

The time complexity of the Bellman-Ford algorithm is O(|V||E|), where |V| is the number of vertices in the graph, and |E| is the number of edges in the graph. This makes it slower than other shortest path algorithms such as Dijkstra's algorithm, but it is more flexible as it can handle negative edge weights.

Algorithm:

Bellmanford(G,w,s)

```
    Initialise_Single_Source(G,w,s)
    For i=1 to |G.V|-1
        For each edge(u,v) of G.E
            Relax(u,v,w)
        For each edge(u,v) of G.E
            If v.d > u.d + w(u,v)
                Return false
            Else
                Return true
Initialize_single_source(G,s)
    For each vertex v of G.V
        v.d = ∞
v. π = NIL
Relax(u,v,w)
    If v.d > u.d + w(u,v)
        v.d = u.d + w(u,v)
        v. π = u
```

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX_NODES 1000
#define MAX_EDGES 10000
int dist[MAX_NODES];
int edges[MAX_EDGES][3];

void BellmanFord(int nodes, int edges_count, int start_node) {
```

```
    for (int i = 0; i < nodes; i++) {
        if (i == start_node) {
            dist[i] = 0;
        } else {
            dist[i] = INT_MAX;
        }
    }
    for (int i = 1; i < nodes; i++) {
        for (int j = 0; j < edges_count; j++) {
            int source = edges[j][0];
            int destination = edges[j][1];
            int weight = edges[j][2];

            if (dist[source] != INT_MAX && dist[source] + weight < dist[destination]) {
                dist[destination] = dist[source] + weight;
            }
        }
    }
    for (int i = 0; i < edges_count; i++) {
        int source = edges[i][0];
        int destination = edges[i][1];
        int weight = edges[i][2];

        if (dist[source] != INT_MAX && dist[source] + weight < dist[destination]) {
            printf("Graph contains negative-weight cycle\n");
            return;
        }
    }
    for (int i = 0; i < nodes; i++) {
```

```
        printf("Shortest distance from node %d to %d is %d\n", start_node, i, dist[i]);
    }
}

int main() {
    int nodes, edges_count, start_node;
    printf("Enter number of nodes, edges and starting node: ");
    scanf("%d %d %d", &nodes, &edges_count, &start_node);
    printf("Enter the edges in the format <source> <destination> <weight>:\n");
    for (int i = 0; i < edges_count; i++) {
        scanf("%d %d %d", &edges[i][0], &edges[i][1], &edges[i][2]);
    }

    BellmanFord(nodes, edges_count, start_node);
    return 0;
}
```

Output:

```
Enter number of nodes, edges and starting node: 7 10 0
Enter the edges in the format <source> <destination> <weight>:
0 1 6
0 2 5
0 3 5
2 1 -2
3 2 -2
1 4 -1
2 4 1
3 5 -1
5 6 3
4 6 3
Shortest distance from node 0 to 0 is 0
Shortest distance from node 0 to 1 is 1
Shortest distance from node 0 to 2 is 3
Shortest distance from node 0 to 3 is 5
Shortest distance from node 0 to 4 is 0
Shortest distance from node 0 to 5 is 4
Shortest distance from node 0 to 6 is 3
```

Conclusion:

In conclusion, the Bellman-Ford algorithm is a useful algorithm for finding shortest paths in graphs with negative edge weights, and its ability to detect negative-weight cycles makes it a valuable tool in many applications. However, its time complexity of O(|V||E|) makes it less efficient than Dijkstra's algorithm for graphs with non-negative edge weights.

Experiment 8

Aim: To implement n-queens problem.

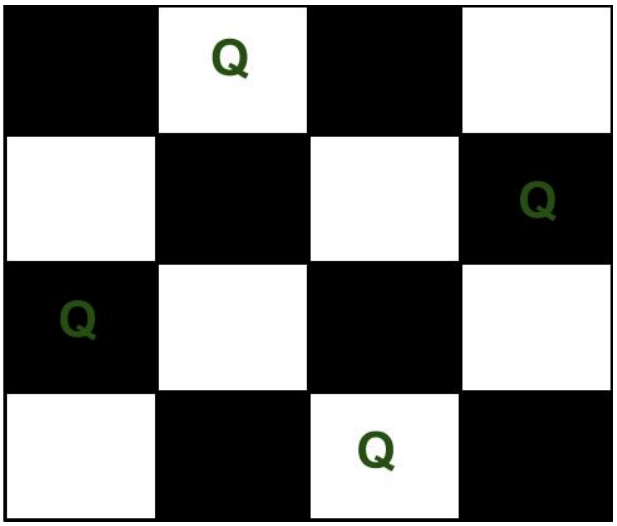
Theory: The n queens problem is a classic problem in computer science and mathematics that involves placing n chess queens on an n x n chessboard such that no two queens threaten each other. In other words, no two queens can be placed on the same row, column, or diagonal.

The problem has a simple solution for small values of n, but it becomes increasingly challenging as n grows larger. The problem can be solved using various algorithms, including brute force search, backtracking, and heuristic methods. The solution to the problem has applications in various fields, including computer science, mathematics, and operations research.

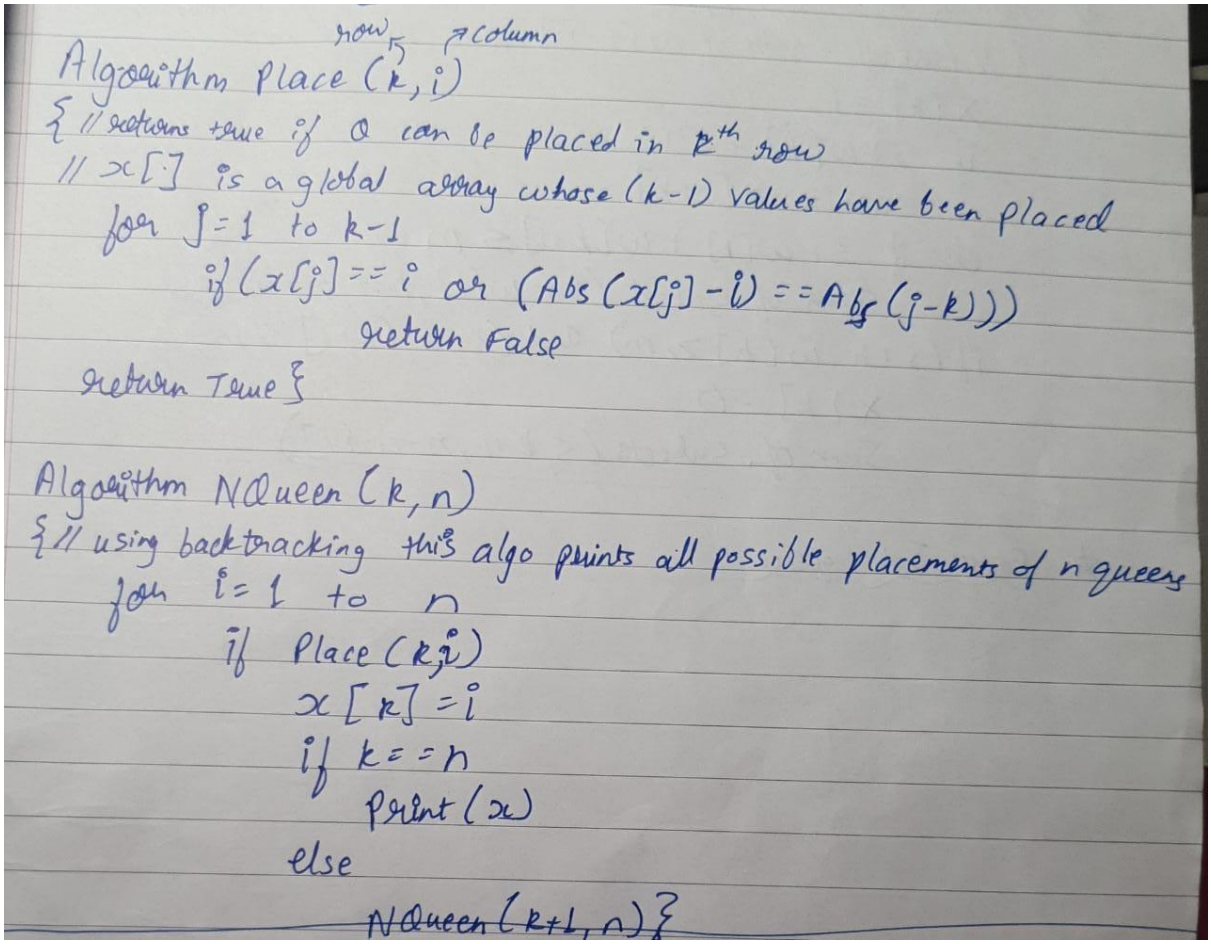
The n queens problem is a popular topic for research and study in computer science and mathematics and has many variations and extensions that continue to inspire new research and discoveries.

Example:

following is a solution for 4 Queen problem.



Algorithm:



```
                    row ↖   ↗ column
Algorithm Place (k, i)
{ // returns true if Q can be placed in kth row
// x[] is a global array whose (k-1) values have been placed
   for j = 1 to k-1
        if (x[j] == i or (Abs (x[j] - i) == Abs (j-k)))
                return False
   return True }

Algorithm NQueen (k, n)
{ // using backtracking this algo prints all possible placements of n queens
   for i = 1 to n
        if Place (k, i)
            x [k] = i
            if k == n
                print (x)
            else
                NQueen (k+1, n) }
```

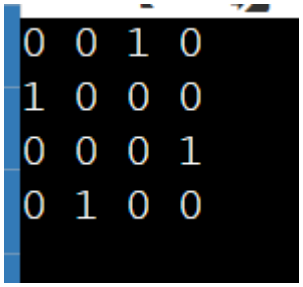
Code:

```
#define N 4
#include <stdbool.h>
#include <stdio.h>
void printSolution(int board[N][N])
{
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++)
printf("%d ", board[i][j]);
printf("\n");
}
}
bool isSafe(int board[N][N], int row, int col)
{
int i, j;
for (i = 0; i < col; i++)
```

```
if (board[row][i])
return false;
for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
if (board[i][j])
return false;
for (i = row, j = col; j >= 0 && i < N; i++, j--)
if (board[i][j])
return false;
return true;
}
bool solveNQUtil(int board[N][N], int col)
{
if (col >= N)
return true;
for (int i = 0; i < N; i++) {
if (isSafe(board, i, col)) {
board[i][col] = 1;
if (solveNQUtil(board, col + 1))
return true;
board[i][col] = 0;
}
}
return false;
}
bool solveNQ()
{
int board[N][N] = { { 0, 0, 0, 0 },
                    { 0, 0, 0, 0 },
                    { 0, 0, 0, 0 },
                    { 0, 0, 0, 0 } };
if (solveNQUtil(board, 0) == false) {
printf("Solution does not exist");
return false;
}
printSolution(board);
return true;
}
```

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

Output:



Conclusion: Hence we have successfully implemented n queens problem.

Experiment 9

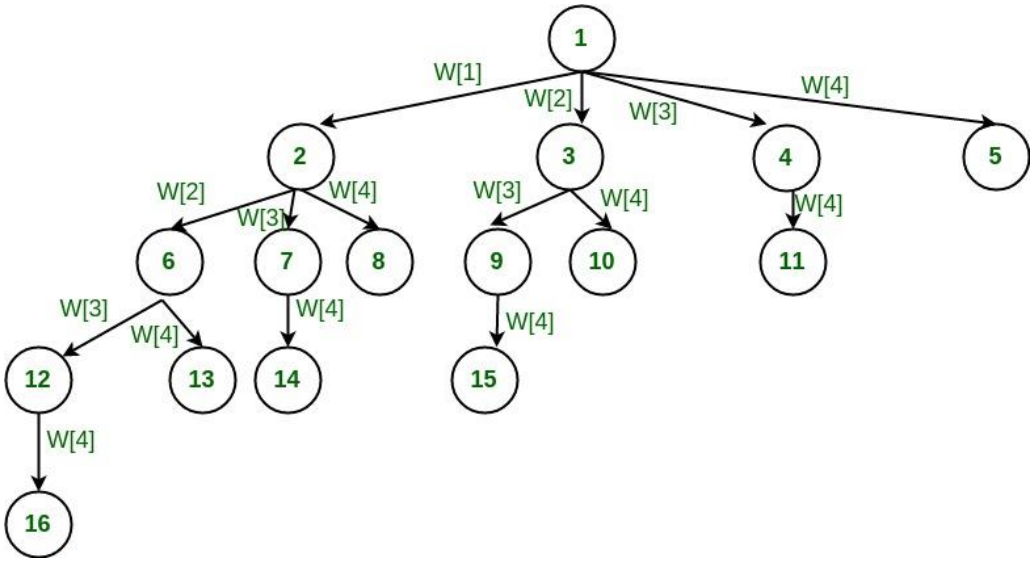
Aim: To implement sum of subsets.

Theory: The sum of subsets problem is a classic problem in computer science that involves finding all possible subsets of a set of numbers whose sum equals a given target value. Backtracking is a commonly used algorithmic technique to solve this problem efficiently.

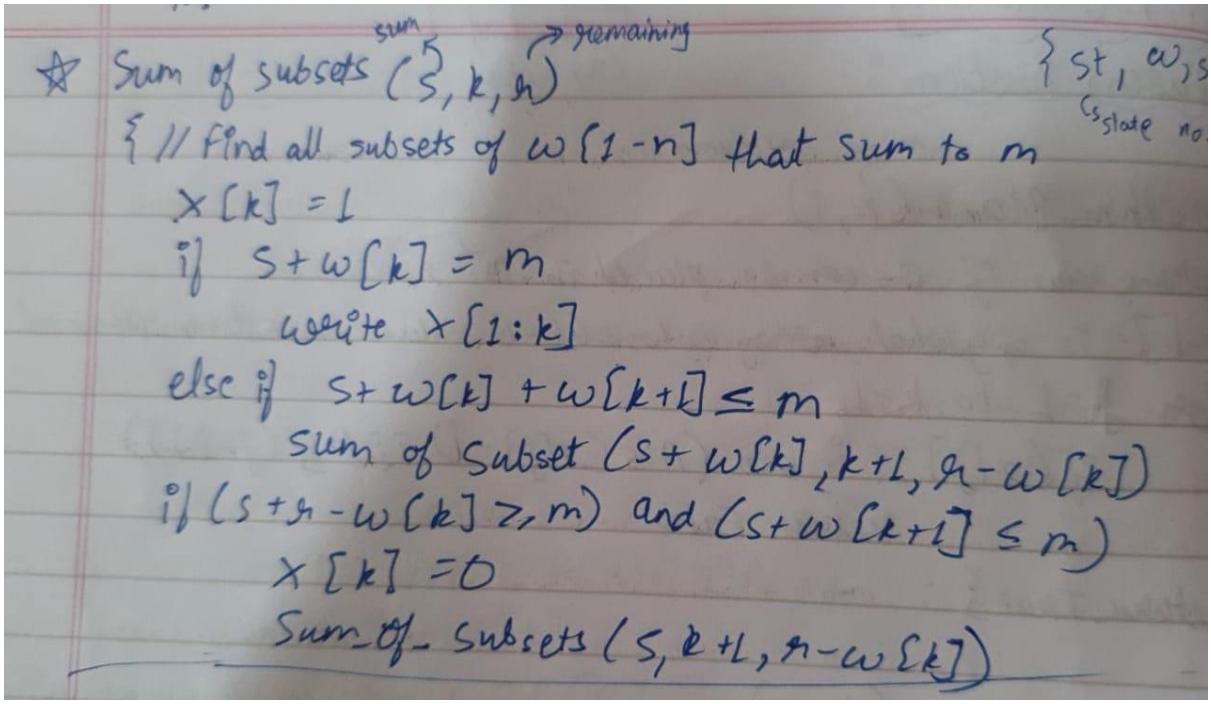
In the backtracking approach, the algorithm starts with an empty subset and adds elements to it one by one. At each step, the algorithm checks whether adding the next element to the subset will exceed the target sum. If it does, the algorithm backtracks and tries a different element.

The backtracking approach avoids considering subsets that cannot contribute to the target sum, which improves the efficiency of the algorithm. The sum of subsets problem has applications in various fields, including operations research, finance, and computer science.

Example:



Algorithm:



Code:

```
#include <stdio.h>
#include <stdlib.h>
static int total_nodes;
void printValues(int A[], int size){
for (int i = 0; i < size; i++) {
printf("%*d", 5, A[i]);
}
printf("\n");
}
void subset_sum(int s[], int t[], int s_size, int t_size, int sum, int ite, int const
target_sum){
total_nodes++;
if (target_sum == sum) {
printValues(t, t_size);
subset_sum(s, t, s_size, t_size - 1, sum - s[ite], ite + 1, target_sum);
return;
}
else {
for (int i = ite; i < s_size; i++) {
t[t_size] = s[i];
```

```
subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
}
}
}
void generateSubsets(int s[], int size, int target_sum){
int* tuplet_vector = (int*)malloc(size * sizeof(int));
subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
free(tuplet_vector);
}
int main(){
int set[] = { 7, 6, 12 , 54, 2 , 20 , 15 };
int size = sizeof(set) / sizeof(set[0]);
printf("The set is ");
printValues(set , size);
generateSubsets(set, size, 25);
printf("Total Nodes generated %d\n", total_nodes);
return 0;
}
```

Output:

```
The set is     7    6   12   54    2   20   15
    7    6   12
Total Nodes generated 121
```

Conclusion: Hence we have successfully implemented sum of subsets.

Experiment 10

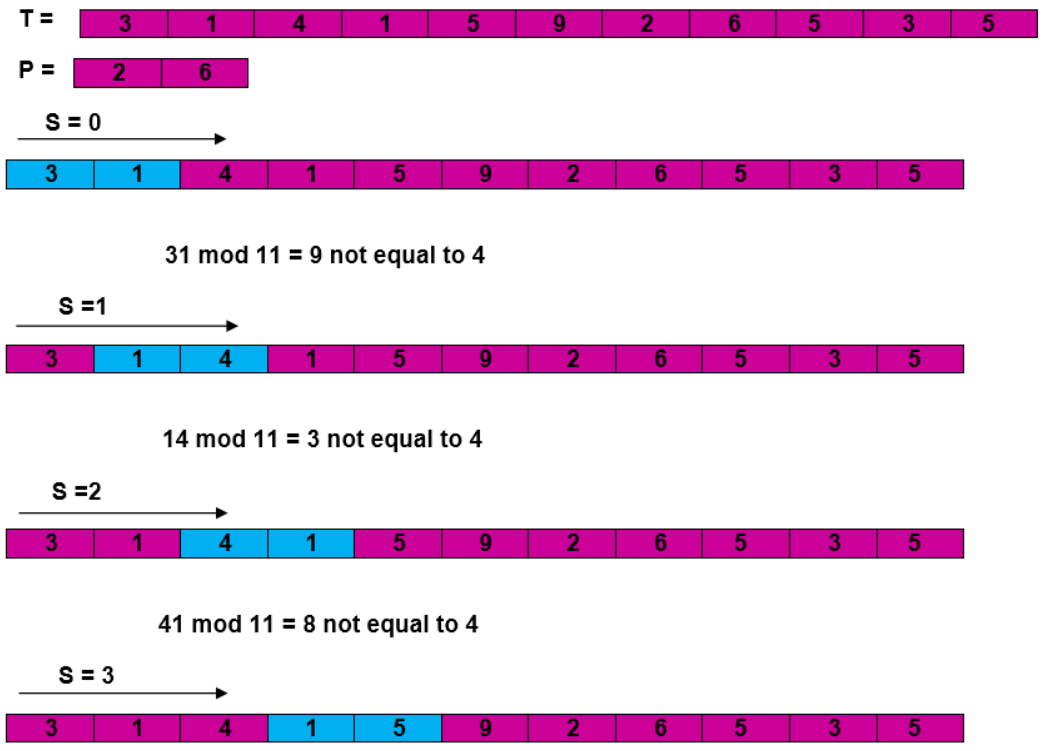
Aim: To implement string matching algorithms Rabin Karp and Knuth Morris Pratt.

Rabin Karp:

Theory: The Rabin-Karp algorithm is a string matching algorithm that searches for a given pattern in a text string by using hashing. The algorithm calculates a hash value for the pattern and for each substring of the text, and compares these hash values to determine whether the pattern matches the substring.

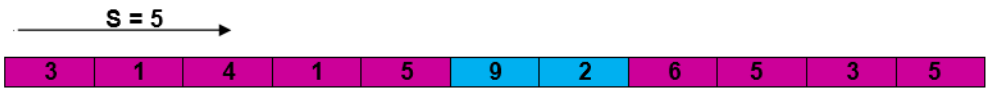
The Rabin-Karp algorithm is efficient for pattern matching because it avoids comparing each character of the pattern to each character of the text string. Instead, it compares the hash values, which are much faster to compute. However, the algorithm can have poor worst-case performance if the hash function produces many hash collisions.

Example:

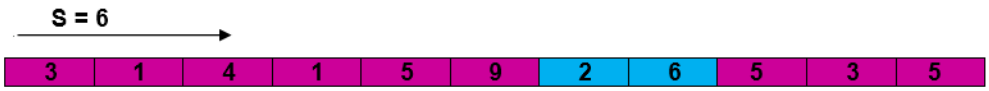


15 mod 11 = 4 equal to 4 SPURIOUS HIT

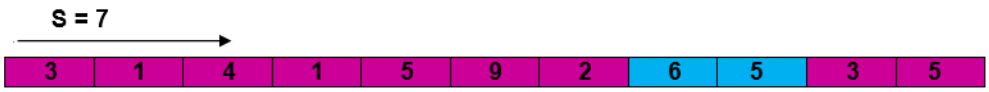
59 mod 11 = 4 equal to 4 SPURIOUS HIT



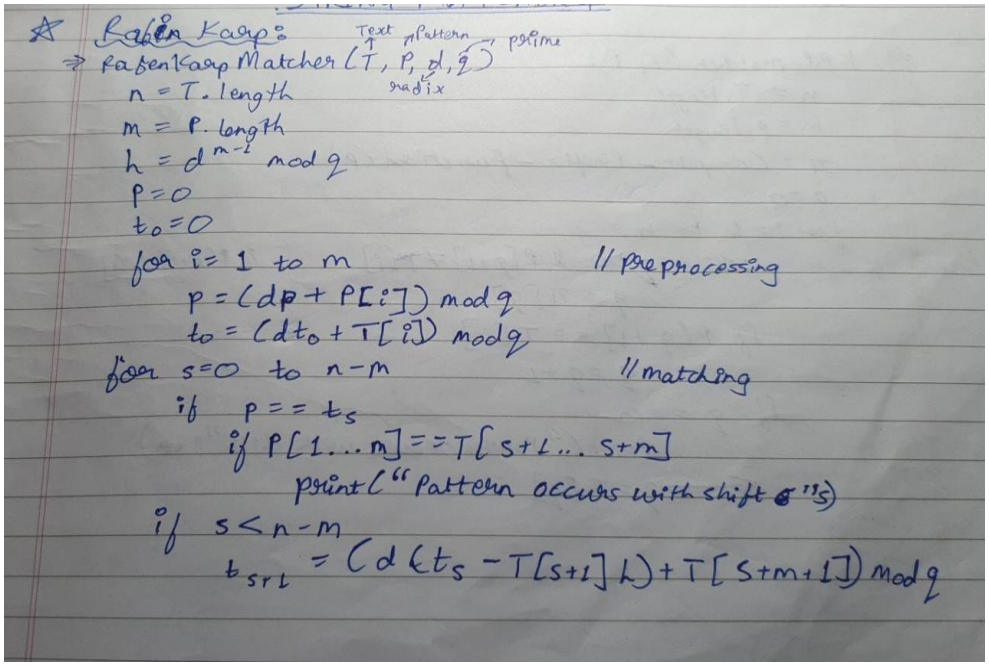
92 mod 11 = 4 equal to 4 SPURIOUS HIT



26 mod 11 = 4 EXACT MATCH



Algorithm:



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);
    }
    }
}
void main()
{char txt[] = "hello good morning";
    char pat[] = "good";
    int q = 101;
    search(pat, txt, q);
}
```

Output:

```
Pattern found at index 6
```

Knuth Morris Pratt:

Theory: The Knuth-Morris-Pratt (KMP) algorithm is a string matching algorithm that searches for a given pattern in a text string by using a failure function. The algorithm calculates the failure function, which is an array of values that indicate the positions in the pattern where a prefix matches a suffix. The algorithm then uses the failure function to avoid unnecessary comparisons while searching for the pattern in the text.

The KMP algorithm has better worst-case performance than other string matching algorithms because it avoids comparing overlapping substrings of the pattern. It is particularly efficient when the pattern contains repeated prefixes.

Example:

P :

a	b	a	b	a	c	a

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	

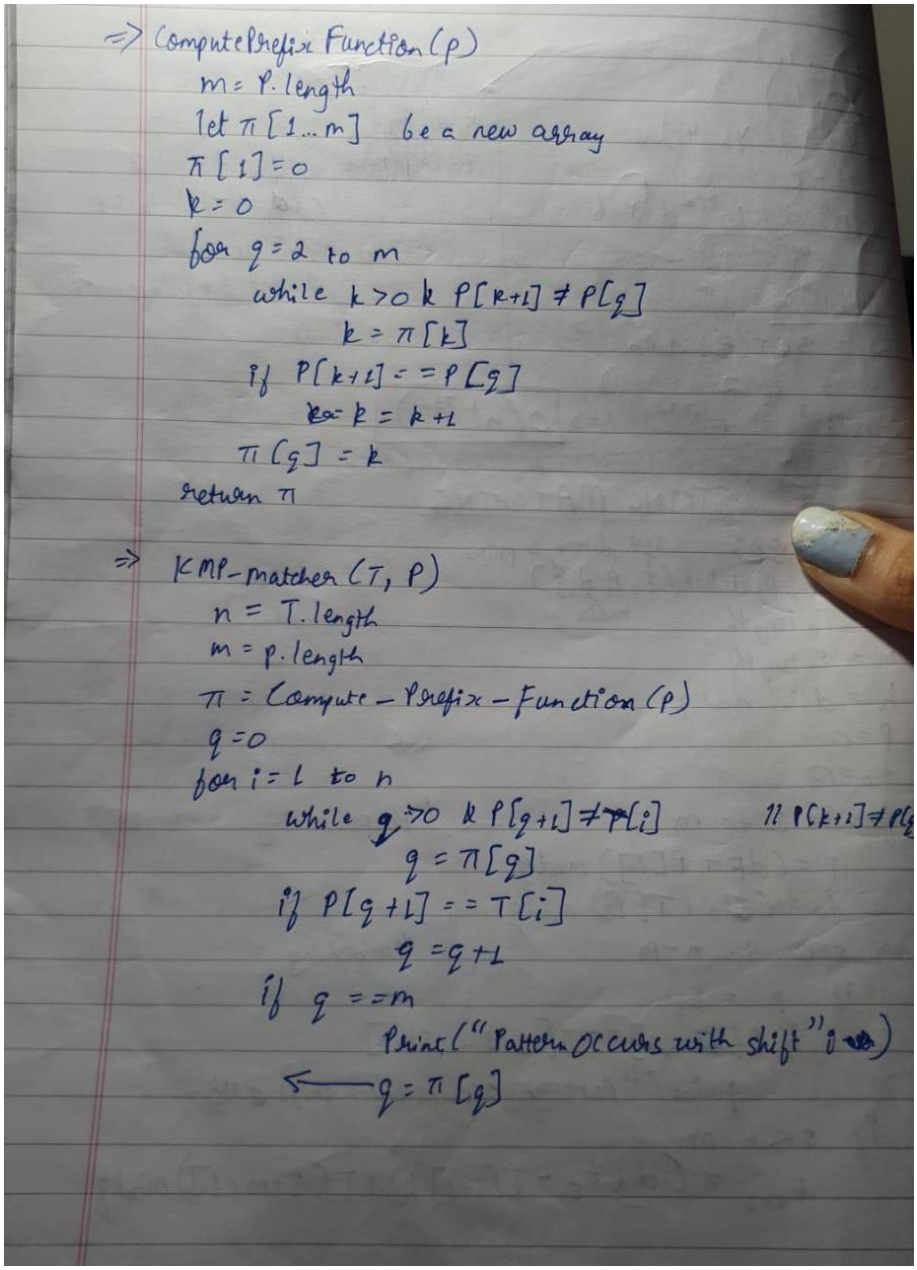
Step6: q = 7, k = 1

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

Algorithm:



Code:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.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 = (int *) malloc(sizeof(int) * M);
```

```
    int j = 0;
    computeLPSArray(pat, M, lps);
    int i = 0;
    while (i < N) {
        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;
        }
    }
    free(lps);
}
void computeLPSArray(char *pat, int M, int *lps) {
    int len = 0;
    int i;
    lps[0] = 0;
    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++;
        }
     }
  }
}
void main() {
   char *txt = "hello good morning";
   char *pat = "good";
   KMPSearch(pat, txt);
}
```

Output:

```
Found pattern at index 6
```

Conclusion: Hence we have successfully implemented string matching algorithms Rabin Karp and Knuth Morris Pratt.

Shreya Shah
60004210045
A1 Comps
AOA

Assignment - 1

Q1] → Asymptonic Notations are: a mathematical tool to find time or space complexity of an algorithm without implementing it in a programming language
It is a way of describing a major component of the cost of the entire algorithm. There are 3 notations:
(a) Big oh (O)
(b) Big Omega (Ω)
(c) Big theta (Θ)

→ The order of growth refers to rate at which the resources required by an algo increase as the size of input increases. It allows us to compare performance of different algorithms and to make predictions about how the algo will scale as size of input increases

→ Efficiency classes are a way of describing the performance or time complexity of an algo or in terms of growth rate of its resource usage. Some examples are
(a) $O(\log n)$: Logarithmic time. Its running time grows logarithmically with its input size
(b) $O(n)$: Linear Running time grows linearly with input size
(c) $O(n^2)$: quadratic Running time grows quadratically with input size.

Q2] Karatsuba algorithm is a fast multiplication algo for long integers. It is a divide & conquer algo
The naive algo has a running time of $O(n^2)$ while this has running time of $O(n^{\log_2 3}) \approx O(n^{1.585})$
Karatsuba stated that if we have to multiply 2 n-digit numbers x and y:
~~assume B is the base of m and m < n, first~~

P.T.O.

Both Numbers can be represented as

$x = a * B + b$

$y = c * D + d$

where a, b, c, d are similar and B & D are powers of 10 such that $B \leq$ no. of digits in $x/2$

& D is largest power 10 such that $D \leq$ no of digits in $y/2$

$xy = (a * B + b) * (c * D + d)$

$= a * c * B * D + (a * d + b * c) * B + b * d$

Now instead of separating:

$z_0 = b * d$;

$z_1 = (a+b) * (c+d)$; $z_2 = a * c$

Thus $x * y = z_2 + B * D + (z_1 - z_2 - z_0) * B + z_0$

eg: $x = 1234$ $y = 5678$

$x = 12 * 100 + 34$ $y = 56 * 100 + 78$

$z_0 = 34 * 78 = 2652$ $z_1 = (12+34) * (56+78) = 4992$

$z_2 = 12 * 56 = 672$

$\therefore x * y = 672 * 10000 + (4992 - 672 - 2652) * 100 + 2652$

$= 7000052$

Q3]

Greedy	Dynamic
→ Choosing best option for best profit	→ Optimizing recursive backtracking solution
→ Time complexity is polynomial	→ Time complexity is polynomial
→ More efficient	→ less efficient
→ no guarantee of optimal solution	→ Always gives optimal solution
→ eg: Fractional Knapsack	→ eg: 0/1 Knapsack

Shreya Shah
60004210048

Q4] All pair shortest path Problem involves finding shortest path between all pairs of vertices in a weighted graph. One such algo is Floyd Warshall. Algo computes shortest path between all pair in $O(v^2)$ times ~~space~~ $O(V^2)$ space complexity

Eg: $D^0 =$

	1	2	3	4
1	0	4	∞	8
2	∞	0	12	5
3	5	∞	0	∞
4	∞	∞	7	0

$D^1 =$

0	4	∞	8
∞	0	12	5
5	9	0	13
∞	∞	7	0

$D^2 =$

0	4	16	8
∞	0	12	5
5	9	0	13
∞	∞	7	0

$D^3 =$

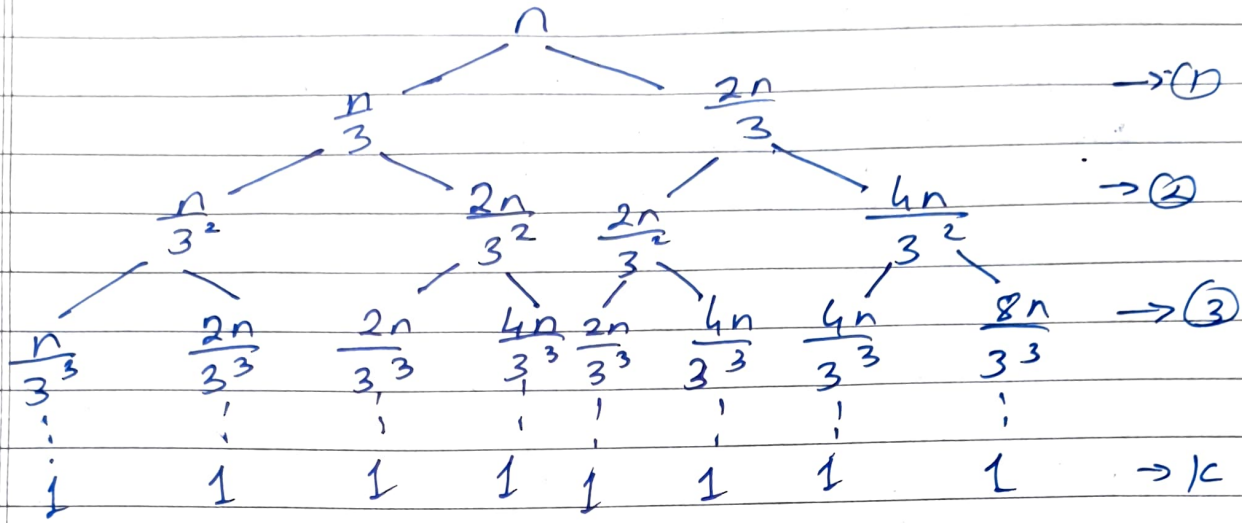
0	4	16	8
17	0	12	5
5	9	0	13
12	16	7	0

$D^4 =$

0	4	15	8
17	0	12	5
5	9	0	13
12	16	7	0

PTO

Q5] $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n^2$



Cost of each corresponding division of tree is n^2

Let height of tree = k

Value of n at level 1 = $2n/3$

2 = $4n/3^2$.

$k = \left(\frac{2}{3}\right)^{lc} n$

$\therefore \left(\frac{2}{3}\right)^k . n = 1 \quad \therefore k \log \frac{2}{3} + \log n = 0$

$\therefore k = \frac{\log n}{\log 3/2} = \log_{3/2} n$

$\therefore$ Total cost : $kn^2 = \log_{3/2} n \cdot n^2 = n^2 \log_{3/2} n$

$\therefore$ Total time complexity $T(n) = O(n^2 \log_{3/2} n)$

Shreya Shah
60004210045
AI Comps
AOA

Assignment - 2

Q1] $w = \{2, 3, 4, 5\}$ $\quad n = 5$

$p = \{3, 4, 5, 6\}$ (p → w)

$S^0 = \{(0, 0)\}$

$S_1^0 = \{(3, 2)\}$

$S^1 = \{(0, 0), (3, 2)\}$

$S_2^1 = \{(4, 3), (7, 5)\}$

$S^2 = \{(0, 0), (3, 2), (4, 3), (7, 5)\}$

$S_3^2 = \{(5, 4), (8, 6), (9, 7), (12, 9)\}$

$S^3 = \{(0, 0), (3, 2), (4, 3), (7, 5), (5, 4), (8, 8)\}$

$S_4^3 = \{(6, 5), (9, 7), (10, 8), (13, 10), (11, 9)\}$

$S^4 = \{(0, 0), (3, 2), (4, 3), (6, 5), [(7, 5)], (5, 4), (8, 8)\}$

max profit

$(7, 5) - (4, 3) = (3, 2)$ O_2

$(3, 2) - (3, 2) = 0, 0$ O_1

$[1, 1, 0, 0]$

Q2]

Dynamic :

(i) Guarantee optimal solution
(ii) Subproblems overlap
(iii) Does more work compared to both
(iv) No specialized set of feasible solutions
(v) Employ memorization

Divide and conquer :

(i) Does not aim for optimal solution
(ii) Subproblems are independent
(iii) Slower and inefficient
(iv) No memorization

Greedy :

(i) Does not guarantee optimal solution
(ii) Sub problems do not overlap
(iii) Does little work
(iv) Select choice which is locally optimum
(v) No memorization

Q3] Algorithm.

```
Sum_of_subsets (s, k, r)
{ // Find all subsets of w[1:n] that sum to m
    x[k] = 1
    if (s + w[k] == m)
        write x[1:k]
    else if (s + w[k] + w[k+1] <= m)
        sum_of_subsets( s + w[k], k+1, r - w[k])
    if (s + r - w[k] >= m) and (s + w[k+1] <= m)
        x[k] = 0
        sum_of_subsets( s, k+1, r - w[k])
```

Eg: n = 4, w = {4, 5, 8, 9} & m = 9, set s = 0

Step 1: i = 1, Adding item w_i

~~sum = sum + w[i]~~ $s = s + w_i = 0 + 4 = 4$

$s \leq m$, ∴ add to soln set

$x[i] = x[1] = 1 \Rightarrow x = \{1, 0, 0, 0\}$

Step 2: i = 2, Adding item w_i

$s = s + w_i = 4 + 5 = 9$

$s == m$

So solution found & add to sol" set

$x[2] = 1 \Rightarrow x = \{1, 1, 0, 0\}$

PTO

60004210045
Shreya Shah

Q4] Max $Z - (3x_1 + 2x_2 + 0s_1 + 0s_2 + 0s_3) = 0$

subject to $-x_1 + 2x_2 + s_1 = 4$

$3x_1 + 2x_2 + s_2 = 14$

$x_1 - x_2 + s_3 = 3$

Simplex Table:

Iteration No.	Basic Variable	coeff of x_1	x_2	s_1	s_2	s_3	RHS Soln	Ratio
0	Z	-3	-2	0	0	0	0	
	s_1	-1	2	1	0	0	4	—
	s_2	3	2	0	1	0	14	$\frac{14}{3} = 4.67$
	s_3	1	-1	0	0	1	3	$\frac{3}{1} = 3$
1	Z	0	-5	0	0	3	9	
s_3 exits	s_1	0	1	1	0	1	7	7
x_1 enters	s_2	0	5	0	1	-3	5	1
	x_1	1	-1	0	0	1	3	—
1	Z	0	0	0	1	0	14	
s_2 leaves	s_1	0	0	1	-1/5	8/5	6	
x_2 enters	x_2	0	1	0	1/5	-3/5	1	
	x_1	1	0	0	1/5	2/5	4	

$x_1 = 4, x_2 = 1, \therefore Z_{max} = 14$

Qs] (a) Rabin Karp Matcher (T, P, d, q)

```
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
to = 0
for i = 1 to m
    p = (dp + P[i]) mod q
    to = (dto + T[i]) mod q
for s = 0 to n - m
    if p == ts
        if P[1...m] == T[s+1...s+m]
            print ("Pattern occurs with shift" s)
    if s < n - m
        ts+1 = (d(ts - T[s+1]h) + T[s+m+1]) mod q
```

(b)

```
Compute Prefix Function (P)
m = P.length
let π[1...m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 ∧ P[k+1] ≠ P[q]
        k = π[k]
    if P[k+1] == P[q]
        k = k + 1
    π[q] = k
return π
```

```
KMP-matcher (T, P)
n = T.length
m = P.length
π = Compute Prefix Function(P)
q = 0
for i = 1 to n
    while q > 0 ∧ P[q+1] ≠ P[i]
        q = π[q]
    if P[q+1] == T[i]
        q = q + 1
    if q == m
        Print ("Pattern occurs with shift" i)
    q = π[q]
```