NAME **: SHASHWAT N. SHAH**

SAP ID **: 60004220126**

BATCH **: COMPS B3**

SUBJECT **: ANALYSIS OF ALGORITHMS (AOA)**

Experiment No-1

Aim: Write a program to implement and analyze Insertion and Selection Sort

Theory:

Insertion sort is a simple sorting algorithm that works like the way we sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
It is assumed that the first element is already sorted in the card game, and then we select an unsorted element. If the selected unsorted element is greater than the first element, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted elements 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 $\mathbf{O(n^2)}$, 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. The time complexity for each case is as shown below:

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

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 time complexity of selection sort in all cases is **$O(n^2)$** as it makes all the comparisons regardless of the elements of the array.

Code (for best, average and worst cases of Insertion Sort):

```
#include <stdio.h>

#include<time.h>

int main() {

   int key, n;

   double start, end, TT;

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

   scanf("%d", &n);

   int arr[n];

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

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

   {

      //arr[i]= 13*i + 92;  // Best Case

      // OR

      //arr[i]= rand()%100000 + 72;  // Average Case

      // OR

      //arr[i]= 100000 - 7*i; // Worst Case

   }

   start=clock();
```

```
for(int i=1;i<n;i++)
    {
        key=arr[i];
        int j=i-1;
        while(key<arr[j] && j>=0)
        {
            arr[j+1]=arr[j];
            j--;
        }
        arr[j+1]=key;
    }
    end=clock();
    /* printf("The sorted array is: \n");
    for(int i=0;i<n;i++)
    {
        printf("%d\n", arr[i]);
    } */
    TT=(end-start);
    printf("Time taken is %f", TT);
    return 0;
}
```

Outputs:

Best case:

Enter the number of elements: 1000

Time taken is 8.000000

Average Case:

Enter the number of elements: 1000

Time taken is 701.000000

Worst Case:

Enter the number of elements: 1000

Time taken is 1531.000000

Code (for best, average and worst cases of Selection Sort):

```
#include <stdio.h>

#include<time.h>

int main() {

    int imin, n, temp;

    double start, end, TT;

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

    scanf("%d", &n);

    int arr[n];

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

    {
```

```
//arr[i]= 69*i + 92;  //Best case
        //arr[i]= rand()%100000 + 72;  //Average Case
        //arr[i]= 100000 - 7*i; //Worst Case
    }
    start=clock();
    for(int i=0;i<n-1;i++)
    {
        imin=i;
        for(int j=i+1;j<n;j++)
        {
            if(arr[imin]>arr[j])
            {
                imin=j;
            }
        }
        temp=arr[imin];
        arr[imin]=arr[i];
        arr[i]=temp;
    }
    end=clock();
   /* printf("The sorted array is: \n");
```

```
for(int i=0;i<n;i++)
    {
        printf("%d\n", arr[i]);
    } */
    TT=(end-start);
    printf("Time taken is %f", TT);
    return 0;
}
```

Outputs:

Best case:

Enter the number of elements: 1000

Time taken is 1516.000000

Average Case:

Enter the number of elements: 1000

Time taken is 1469.000000

Worst Case:

Enter the number of elements: 1000

Time taken is 1416.000000

Conclusion: Hence we have analyzed the performances of both these algorithms and it is evident that Insertion sort performs with improved time complexities in best and average cases while Selection sort always performs with the same $O(n^2)$ complexity for any case.

Experiment No-2

Aim: Write a program to implement and analyse Merge and Quick Sort

Theory:

Merge sort is the sorting technique that follows the divide and conquer approach. 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 are merged into the four-element lists, and so on until we get the sorted list. One of the main advantages of merge sort is that it has a time complexity of **O(n*lgn)** for all cases, which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort. Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Quicksort 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 sub problems, then solving the sub problems, 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.
Its time complexities for each case is given as follows:

Best	-	O(n*log n)…..Elements in random order
Worst		O(n^2)…Elements sorted in ascending or descending order
Average	-	O(n*log n)…..Elements in random order

Code (for best, average and worst cases of Merge Sort):

```
#include <stdio.h>
#include  <time.h>
int main()
{
    int arr[1000], n;
    double start, end, TT;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    for(int i=0;i<n;i++)
    {
        //arr[i]= 13*i + 92;  //Best
        //arr[i]= rand()%10000 + 72;  //Average
        //arr[i]= 100000 - 7*i;  //Worst
    }
    start=clock();
    mergeSort(arr, 0, n-1);
```

```
    end=clock();
 /* printf("The sorted array is: \n");
    for(int i=0;i<n;i++)
    {
        printf("%d\n", arr[i]);
    }  */
    TT=(end-start);
    printf("Time taken is %f", TT);
    return 0;
}

void merge(int arr[], int low, int mid, int high)
{
  int n1 = mid - low + 1;
  int n2 = high - mid;
  int L[n1], M[n2];
  for (int i = 0; i < n1; i++)
    L[i] = arr[low + i];
  for (int j = 0; j < n2; j++)
  M[j] = arr[mid + 1 + j];
  int i, j, k;
  i = 0;
```

```
  j = 0;
  k = low;
  while (i < n1 && j < n2)
  {
    if (L[i] <= M[j])
    {
      arr[k++] = L[i++];
    }
    else
    {
      arr[k++] = M[j++];
    }
  }
  while (i < n1)
  {
    arr[k++] = L[i++];
  }
  while (j < n2)
  {
    arr[k++] = M[j++];
  }
}
```

```
void mergeSort(int arr[], int low, int high)
{
  if (low < high)
  {
    int mid = (low+high) / 2;
    mergeSort(arr, low, mid);
    mergeSort(arr, mid + 1, high);
    merge(arr, low, mid, high);
  }
}
```

Outputs:

Best case:

Enter the number of elements: 1000

Time taken is 151.000000

Average Case:

Enter the number of elements: 1000

Time taken is 204.000000

Worst Case:

Enter the number of elements: 1000

Time taken is 131.000000

Code (for best, average and worst cases of Quick Sort):

```
#include <stdio.h>
#include  <time.h>
int main() {

    int arr[1000], high, low, n;
    double start, end, TT;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    for(int i=0;i<n;i++)
    {
        // arr[i]= 13*i + 92;  //Worst
        // arr[i]= rand()%100000 + 72;  //Average & Best
        // arr[i]= 100000 - 7*i;  //Worst
    }
    start=clock();
    quickSort(arr, 0, n-1);
    end=clock();
    printf("The sorted array is: \n");
    for(int i=0;i<n;i++)
    {
        printf("%d\n", arr[i]);
    }
    TT=(end-start);
    printf("Time taken is %f", TT);
    return 0;
}

void quickSort(int array[], int low, int high)
{
  if (low < high)
  {
    int pi = partition(array, low, high);
    quickSort(array, low, pi - 1);
    quickSort(array, pi + 1, high);
  }
}
```

```
int partition(int array[], int low, int high)
{

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

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

Outputs:

Best case:

Enter the number of elements: 1000

Time taken is 157.000000

Worst Case:

Enter the number of elements: 1000

Time taken is 2686.000000

Worst Case:

Enter the number of elements: 1000

Time taken is 4292.000000

Conclusion: Hence we have analyzed the performances of both these algorithms and it is evident that Merge sort performs with same Time complexity O(n*lgn) in all cases while Quick sort always performs with worst time complexity $O(n^2)$ when the array is sorted in ascending or descending order and average case of O(n*lgn).

Experiment No-3

Aim: Write a program to implement Min Max and Binary Search using Divide and Conquer approach

Theory:

The Min Max algorithm is a divide-and-conquer algorithm that finds the maximum and minimum values in an array. The algorithm recursively divides the array into two halves until there is only one element in the subarray. It then compares the values in the subarray to find the maximum and minimum values. Finally, it combines the maximum and minimum values from the subarrays to find the maximum and minimum values in the original array. The time complexity of the Min Max algorithm is **O(n)**, where n is the size of the array. This is because the algorithm divides the array in half at each recursive step, and it takes constant time to find the maximum and minimum values in a subarray with only one element.

The binary search algorithm is a divide-and-conquer algorithm that searches for a target value in a sorted array. The algorithm divides the array in half at each recursive step and compares the target value to the middle element of the array. If the target value is less than the middle element, the algorithm recursively searches the left half of the array. If the target value is greater than the middle element, the algorithm recursively searches the right half of the array. The algorithm repeats this process until the target value is found or the search range is empty.
The time complexity of the binary search algorithm is $\mathbf{O(\log_2 n)}$, where n is the size of the array. This is because the algorithm divides the search range in half at each recursive step, and it takes constant time to check if the target value is equal to the middle element. The search range is reduced by a factor of two at each step, so the total number of recursive steps is proportional to lg(n).

Code for MinMax:

```
#include <stdio.h>

void find_max_min(int arr[], int n, int *max, int *min)

{

   if (n == 1) {
```

```
        *max = arr[0];
        *min = arr[0];
        return;
    }
    int mid = n / 2;
    int left_max, left_min, right_max, right_min;

    find_max_min(arr, mid, &left_max, &left_min);
    find_max_min(arr + mid, n - mid, &right_max, &right_min);

    *max = (left_max > right_max) ? left_max : right_max;
    *min = (left_min < right_min) ? left_min : right_min;
}
int main()
{
    int n, i;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int arr[n];
    double start, end, TT;
    for(int i=0;i<n;i++)
    {
```

```
        arr[i]= rand()%100000 + 72;
    }
    int max, min;
    start=clock();
    find_max_min(arr, n, &max, &min);
    end=clock();
    printf("Maximum element is %d\n", max);
    printf("Minimum element is %d\n", min);
    TT=(end-start);
    printf("Time taken is %f", TT);
    return 0;
}
```

Output:

```
Enter the size of the array: 1000
Maximum element is 100004
Minimum element is 153
Time taken is 20.000000
```

Code for Binary Search:

```
#include <stdio.h>
int main() {
```

```
int n, arr[1000], high, low, mid, x, ind;
   double start, end, TT;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   for(int i=0;i<n;i++)
   {
      arr[i]=15*i + 179;  //input array in ascending order
   }
   printf("Enter the element you want to search: ");
   scanf("%d", &x);
   start=clock();
   ind=BinarySearch(arr, n-1, 0, x);
   if(ind==-1)
   {
      printf("\nElement not found");
   }
   else
   {
      printf("\nElement found at position %d", ind+1);
   }
   end=clock();
   TT=end-start;
```

```
    printf("\nTime taken is %f", TT);
    return 0;
}

int BinarySearch(int arr[], int high, int low, int x)
{
    if(high==low)
    {
        if(arr[low]==x)
        {
            return low;
        }
        else
        {
            return -1;
        }
    }
    int mid=(high+low)/2;
    if(arr[mid]==x)
    {
        return mid;
    }
```

```
    else if(arr[mid]>x)
    {
        return BinarySearch(arr, mid-1, low, x);
    }
    else
    {
        return BinarySearch(arr, high, mid+1, x);
    }
}
```

Output:

```
Enter the number of elements: 1000

Enter the element you want to search: 194

Element found at position 2

Time taken is 22.000000
```

Conclusion: Hence we have implemented both these algorithms and understood the divide and conquer approach.

Experiment No-4

Aim: Write a program to implement Single Source shortest path using Greedy Approach

Theory:

Dijkstra's algorithm is a shortest path algorithm used to find the shortest path between two vertices in a weighted graph. The algorithm starts at a source vertex and explores the graph, calculating the minimum distance from the source to every other vertex in the graph.
The algorithm maintains a set of visited vertices and a set of unvisited vertices. Initially, the distance to the source vertex is set to zero and the distance to all other vertices is set to infinity. At each iteration, the algorithm selects the unvisited vertex with the smallest distance and visits all of its neighbors, updating their distances if a shorter path is found. This process continues until all vertices have been visited or the target vertex has been reached.
Dijkstra's algorithm works for both directed and undirected graphs, as long as the graph is non-negative and connected. It is a greedy algorithm, meaning that it selects the locally optimal choice at each step, hoping to find the globally optimal solution.
The time complexity of Dijkstra's algorithm is O(E+V log V), where E is the number of edges and V is the number of vertices in the graph. This time complexity is achieved using a priority queue data structure to store and retrieve the unvisited vertices with the smallest distance.

Code:

```
#include <stdio.h>



int min_source(int done[],int distance[],int source)

{

   int i;

   int x = 1000;

   int y;
```

```
    for(i=0;i<9;i++)
    {
        if(done[i]!=1&&distance[i]<x)
        {
            x=distance[i];
            y=i;
        }
    }
    return y;
}

void Djikstra(int a[9][9],int source)
{
    int source1 = source;
    int done[100],distance[1000],parent[100],v[100];
    int distances,i;

    for(int i=0;i<9;i++)
    {
        done[i]=0;
        distance[i]=10000;
    }
```

```
distance[source]=0;


for(i=0;i<9;i++)
{
   done[source]=1;


   for(int j =0;j<9;j++)
   {
      if(a[source][j]!=0)
      {
         distances = distance[source]+a[source][j];
         if(distance[j]>distances)
         {
            distance[j]=distances;
            parent[j]=source;
         }


      }
   }
   source = min_source(done,distance,source);
}
int max_distance = 0;
```

```
v[0] = source;
printf("The max. distance from the source is:%d\n",distance[source]);
for(i=0;i<8;i++)
{
   source1 = source;
   source = parent[source];
   if(source == source1)
   {
      break;
   }
   v[i+1]=source;
}
printf("\nPath is:\n");
for(int j=i;j>=1;j--)
{
   printf("%d-->",v[j]);
}
printf("%d",v[0]);
printf("\n\n\n");
printf("Table is \n");
printf("_____\n");
printf("| Source | Distance | Parent |\n");
```

```
    printf("|_________|_____________|_________|\n");
    for(i=0;i<9;i++)
    {
        printf("|   %d    |    %d     |   %d    |\n",i,distance[i],parent[i]);
        printf("|_________|_____________|_________|\n");
    }
}

void main() {
    printf("Djikstra Algo Implementation\n");
    int source,size;
    source =0;
    int a[9][9]={
        {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},
        {0,7,0,0,9,14,0,0,0},
        {0,0,0,9,0,10,0,0,0},
        {0,4,0,14,10,0,2,0,0},
        {0,0,0,0,0,2,0,1,6},
        {8,11,0,0,0,0,1,0,7},
        {0,2,0,0,0,0,6,7,0},
```

```
    };
    Djikstra(a,source);
    return 0;
}
```

Output:

Djikstra Algo Implementation

The max. distance from the source is:21

Path is:

0-->7-->6-->5-->4

Table is

Source	Distance	Parent
0	0	0
1	4	0
2	12	1
3	19	2

4	21	5
5	11	6
6	9	7
7	8	0
8	14	2

Conclusion: Hence we have implemented Dijkstra's algorithm and understood the greedy approach to solve graph related problems.

Experiment No-5

Aim: Write a program to implement Minimum Spanning tree (Prim's and Kruskal's)

Theory:

Prim's algorithm and Kruskal's algorithm are two popular algorithms for finding the Minimum Spanning Tree (MST) of a weighted undirected graph.
Prim's algorithm starts with a vertex and then repeatedly adds the cheapest edge that connects a vertex in the MST to a vertex outside the MST, until all vertices are in the MST. It can be implemented using a priority queue or a min-heap, and has a time complexity of **O(E log V)** using a min-heap and **O(V^2)** using an adjacency matrix.
Kruskal's algorithm, on the other hand, starts with an empty MST and repeatedly adds the cheapest edge that does not create a cycle, until all vertices are in the MST. It can be implemented using a union-find data structure or a simple array, and has a time complexity of **O(E log E)** using a sorting algorithm and **O(E log V)** using a union-find data structure.

Code for Prim's:

```
#include <stdio.h>

#define INF 99999

#define V 5

int minKey(int key[], int mstSet[]) {

    int min = INF, 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[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]]);
    }
}

void primMST(int graph[V][V]) {
    int parent[V];
    int  key[V];
    int mstSet[V];
    for (int i = 0; i < V; i++) {
        key[i] = INF;
        mstSet[i] = 0;
    }
    key[0] = 0;
    parent[0] = -1;
```

```
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        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);
}
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 },
    };
    primMST(graph);
    return 0;
```

```
}
```

Output:

```
Edge    Weight

0 - 1    2

1 - 2    3

0 - 3    6

1 - 4    5
```

Code for Kruskal's:

```
#include <stdio.h>

#define MAX 10

int parent[MAX];

int find(int i) {
    while (parent[i] != i) {
        i = parent[i];
    }
    return i;
}

void union_ij(int i, int j) {
    int a = find(i);
    int b = find(j);
    parent[a] = b;
}
```

```
void kruskal(int cost[][MAX], int n) {
    int mincost = 0, ne = 1, i, j, a, b, u, v;
    for (i = 1; i <= n; i++) {
        parent[i] = i;
    }
    while (ne < n) {
        int min = 999;
        for (i = 1; i <= n; i++) {
            for (j = 1; j <= n; j++) {
                if (find(i) != find(j) && cost[i][j] < min) {
                    min = cost[i][j];
                    a = u = i;
                    b = v = j;
                }
            }
        }
        if (find(u) != find(v)) {
            printf("\nEdge %d: (%d, %d) cost: %d", ne++, a, b, min);
            mincost += min;
            union_ij(u, v);
        }
        cost[a][b] = cost[b][a] = 999;
```

```
    }
    printf("\nMinimum cost = %d\n", mincost);
}
int main() {
    int n, i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    int cost[MAX][MAX];
    printf("Enter the cost matrix:\n");
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0) {
                cost[i][j] = 999;
            }
        }
    }
    kruskal(cost, n);
    return 0;
}
```

Output:

Enter the number of vertices: 5

Enter the cost matrix:

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

Edge 1: (1, 2) cost: 2

Edge 2: (2, 3) cost: 3

Edge 3: (2, 5) cost: 5

Edge 4: (1, 4) cost: 6

Minimum cost = 16

Conclusion: Both algorithms guarantee to find the MST of the graph, and the choice of algorithm depends on the specific properties of the graph and the available data structures.

Experiment No-6

Aim: Write a program to implement Longest common subsequence

Theory:

To find the LCS between two strings, we can use dynamic programming. We can create a matrix where the rows repre sent the characters in one string and the columns represent the characters in the other string. We can then fill in the matrix by comparing each pair of characters and using the values from the previous cells to determine the value for the current cell. The algorithm for finding the LCS can be summarized as follows:

1. Create a matrix with one more row and column than the lengths of the two strings.
2. Initialize the first row and column of the matrix to 0.
3. Iterate through each cell of the matrix, starting from the second row and column.
4. If the characters in the two strings at the current row and column match, set the value of the current cell to the value of the cell in the previous row and column plus
5. If the characters in the two strings at the current row and column do not match, set the value of the current cell to the maximum of the value of the cell in the previous row or the cell in the previous column.
6. The value in the last cell of the matrix is the length of the LCS.
7. To find the LCS, start from the last cell of the matrix and backtrack through the matrix. If the value in the current cell is equal to the value in the cell above and to the left, add the character to the LCS and move diagonally to the cell in the previous row and column. If the value in the current cell is equal to the value in the cell above, move up to the cell in the previous row. If the value in the current cell is equal to the value in the cell to the left, move left to the cell in the previous column.
8. This algorithm has a time complexity of **O(mn)**, where m and n are the lengths of the two strings. Therefore, it is suitable for finding the LCS between two relatively short strings. For longer strings, more efficient algorithms may be needed

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 lcsAlgo() { m =
 strlen(S1);n =
```

```
 strlen(S2);
 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--;
 }
 printf("S1 : %s \nS2 : %s \n", S1, S2);
 printf("LCS: %s", lcsAlgo);
}

int main()
{
 printf("Enter string 1: ");
```

```
  scanf("%s",S1);
  printf("Enter string 2: ");
  scanf("%s",S2);
  lcsAlgo();
  printf("\n");
 }
```

Output:

Enter string 1: PROPERTIES

Enter string 2: PROSPERITY

S1 : PROPERTIES

S2 : PROSPERITY

LCS: PROPERI

Conclusion: This algorithm has a time complexity of **O(mn)**, where m and n are the lengths of the two strings. Therefore, it is suitable for finding the LCS between two relatively short strings. For longer strings, more efficient algorithms may be needed

Experiment No-7

Aim: Write a program to implement Single Source shortest path using Dynamic Programming

Theory:

The single source shortest path algorithm (for arbitrary weight positive or negative) is also known Bellman- Ford algorithm is used to find minimum distance from source vertex to any other vertex. The main difference between this algorithm with Dijkstra's algorithm is, in Dijkstra's algorithm we cannot handle the negative weight, but here we can handle it easily. Bellman-Ford algorithm finds the distance in bottom-up manner. At first it finds those distances which have only one edge in the path. After that increase the path length to find all possible solutions.

The time complexity for each case is given as follows:

Best Case Complexity	-	O(E)
Average Case Complexity	-	O(VE)
Worst Case Complexity	-	O(VE)

Code:

```
#include <stdio.h>

void bellman_ford(int a[9][9], int source) {

    int done[9], distance[9], parent[9];

    int i, j, k;

    for(i=0;i<9;i++) {

        done[i] = 0;

        distance[i] = 10000;

        parent[i] = -1;
```

```
}
distance[source] = 0;

for(i=0;i<8;i++) {
   for(j=0;j<9;j++) {
      for(k=0;k<9;k++) {
         if(a[j][k] != 0 && distance[j] != 10000 && distance[k] > distance[j] + a[j][k]) {
            distance[k] = distance[j] + a[j][k];
            parent[k] = j;
         }
      }
   }
}
for(j=0;j<9;j++) {
   for(k=0;k<9;k++) {
      if(a[j][k] != 0 && distance[j] != 10000 && distance[k] > distance[j] + a[j][k]) {
         printf("Negative cycle detected!");
         return;
      }
   }
}
```

```
    printf("Table is \n");
    printf("_____\n");
    printf("| Source | Distance | Parent |\n");
    printf("|________|__________|________|\n");
    for(i=0;i<9;i++) {
        printf("|   %d    |    %d     |   %d    |\n",i,distance[i],parent[i]);
        printf("|________|__________|________|\n");
    }
}
int main() {
    int source = 0;
    int a[9][9]={
        {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},
        {0,7,0,0,9,14,0,0,0},
        {0,0,0,9,0,10,0,0,0},
        {0,4,0,14,10,0,2,0,0},
        {0,0,0,0,0,2,0,1,6},
        {8,11,0,0,0,0,1,0,7},
        {0,2,0,0,0,0,6,7,0},
    };
```

```
    bellman_ford(a,source);

    return 0;

}
```

Output:

Table is

```
_____________________________

| Source | Distance | Parent |

|________|__________|________|

|  0   |   0    |  -1   |

|________|__________|________|

|  1   |   4    |  0   |

|________|__________|________|

|  2   |   12    |  1   |

|________|__________|________|

|  3   |   19    |  2   |

|________|__________|________|

|  4   |   21    |  5   |

|________|__________|________|

|  5   |   11    |  6   |

|________|__________|________|

|  6   |   9    |  7   |
```

|_____|_____|_____|

| 7 | 8 | 0 |

|_____|_____|_____|

| 8 | 14 | 2 |

|_____|_____|_____|

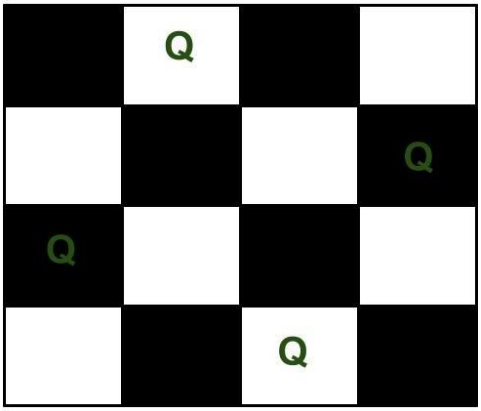
Conclusion: Hence we have implemented Single Source shortest path dynamic programming (Bellman Ford Algo) and verified its time complexity.

Experiment No-8

Aim: Write a program to implement N queens Problem

Theory:

The N-Queens problem is a classical backtracking problem that requires us to place N number of queens on an N x N chessboard (one in each row) such that no queen can intercept any other queen in any direction. One value of N can have multiple solutions for this problem. Following is one of the solutions for N=4.



The time complexity of N-Queens problem is **O(n!)** as it recursively finds solutions for each value of n, then n-1, then n-2 and so on.

Code:

```
#include <stdio.h>

#include <math.h>

#include <stdbool.h>

int n;

int x[100];

bool isSafe(int k, int i) {

    for (int j = 1; j <= k - 1; j++) {
```

```
        if (x[j] == i || abs(x[j] - i) == abs(j - k)) {
            return false;
        }
    }
    return true;
}

void nQueens(int k) {
    for (int i = 1; i <= n; i++) {
        if (isSafe(k, i))
        {
            x[k] = i;
            if (k == n)
            {
                for (int j = 1; j <= n; j++)
                {
                    printf("%d ", x[j]);
                }
                printf("\n");
            }
            else
            {
```

```
            nQueens(k + 1);
          }
        }
    }
}

int main()
{
    do
    {
        printf("N-Queens Problem\nEnter the number of queens: ");
        scanf("%d", &n);
        nQueens(1);
    }while(true);
    return 0;
}
```

Output:

```
N-Queens Problem
Enter the number of queens: 4
2 4 1 3
3 1 4 2
N-Queens Problem
```

Enter the number of queens: 5

1 3 5 2 4

1 4 2 5 3

2 4 1 3 5

2 5 3 1 4

3 1 4 2 5

3 5 2 4 1

4 1 3 5 2

4 2 5 3 1

5 2 4 1 3

5 3 1 4 2

Conclusion: Hence we have implemented and verified the method of backtracking by solving the N-Queens problem.

Experiment No-9

Aim: Write a program to implement Sum of subset.

Theory:

Sum of subset problem is the problem of finding a subset such that the sum of elements equal a given number. The backtracking approach generates all permutations in the worst case but in general, performs better than the recursive approach towards subset sum problem. We generate a hypothetical binary tree to generate all possible combinations of considering or not considering an element at each level and hence we get all the possible subsets which sum up to the required number. In code, we use a recursive approach to achieve this. The output is printed as an array of binary numbers which shows whether the element at i^{th} index is considered or not.

Since it's a binary tree kind of approach, the **time complexity is $O(2^n)$**.

Code:

```
#include <stdio.h>

int m, n, arr[100], x[100]={0};

int SumOfSubsets(int s, int k, int r)

{

   x[k]=1;

   if(s+arr[k]==m)

   {

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

      {

         x[j]=0;

      }
```

```
        printf("Answer is\n");
        for(int i=0;i<n;i++)
        {
            printf("%d  ", x[i]);
        }
        printf("\n");
    }
    else if(s+arr[k]+arr[k+1]<=m)
    {
        SumOfSubsets(s+arr[k], k+1, r-arr[k]);
    }
    if(s+r-arr[k]>=m && s+arr[k+1]<=m)
    {
        x[k]=0;
        SumOfSubsets(s,k+1,r-arr[k]);
    }
}

int main()
{
    int s=0;
    printf("Sum of Subsets\nEnter the number of elements: ");
```

```
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
        s += arr[i];
    }
    printf("Enter the sum needed: ");
    scanf("%d", &m);
    SumOfSubsets(0, 0, s);
    return 0;
}
```

Output:

```
Sum of Subsets
Enter the number of elements: 8
Enter 8 elements:
1 2 3 4 5 6 7 8
Enter the sum needed: 13
Answer is
1 1  1  0  0  0  1  0
Answer is
1 1  0  1  0  1  0  0
```

Answer is

1 0 1 1 1 0 0 0

Answer is

1 0 0 1 0 0 0 1

Answer is

1 0 0 0 1 0 1 0

Answer is

0 1 1 0 0 0 0 1

Answer is

0 1 0 1 0 0 1 0

Answer is

0 1 0 0 1 1 0 0

Answer is

0 0 1 1 0 1 0 0

Answer is

0 0 0 0 1 0 0 1

Answer is

0 0 0 0 0 1 1 0

Conclusion: Hence we have implemented the sum of subsets problem using recursive calls and backtracking. It is quite an elegant and simple solution to an interesting problem.

Experiment No-10

Aim: Write a program to implement String matching using Rabin Karp and KMP Algorithm

Theory:

The Rabin-Karp algorithm is a string-matching algorithm that uses hashing to compare the pattern with the text. The algorithm compares the hash value of the pattern with the hash value of each substring of the text of the same length as the pattern. If the hash values match, the algorithm then checks whether the pattern and substring are the same, character by character.
Time complexity: The average and best-case time complexity of the Rabin-Karp algorithm is **O(n+m)**, where n is the length of the text and m is the length of the pattern. However, the worst-case time complexity is **O(nm)**, which occurs when there are many hash collisions.
The Knuth-Morris-Pratt (KMP) algorithm is a string-matching algorithm that uses a prefix function to avoid unnecessary comparisons. The prefix function is calculated on the pattern string before the actual string matching. It uses the information from the prefix function to avoid matching the characters that have already been matched. The prefix function is calculated in linear time, and the actual matching also takes linear time.
Time complexity: The time complexity of the KMP algorithm is **O(n+m)**, where n is the length of the text and m is the length of the pattern. The calculation of the prefix function takes O(m) time, and the actual matching takes O(n) time. Therefore, the overall time complexity is O(n+m).

Code for Rabin Karp:

```
#include <stdio.h>

#include <string.h>

int d=23;

void search(char P[], char T[], int q)

{

int m = strlen(P);
```

```
int n = strlen(T);
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 + P[i]) % q;
t = (d * t + T[i]) % q;
}
for (i = 0; i <= n - m; i++) {
if (p == t) {
for (j = 0; j < m; j++) {
if (T[i + j] != P[j])
break;
}
if (j == m)
printf("Pattern found at index %d \n", i);
}
if (i < n - m) {
t = (d * (t - T[i] * h) + T[i + m]) % q;
```

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

int main()
{
    char P[200],T[200];
    printf("Rabin Karp\nEnter the text: \n");
    gets(T);
    printf("Enter the pattern: \n");
    gets(P);
    int q = 13;
search(P, T, q);
return 0;
}
```

Output:

```
Rabin Karp
Enter the text:
gfhdsvuyfgdshvcskfsdhcvsdyfv
Enter the pattern:
```

vsdyf

Pattern found at index 22

Code for KMP:

```
#include <stdio.h>
#include <string.h>
void compute_failure_function(char *pattern, int *failure)
{
int i = 1, j = 0, m = strlen(pattern);
failure[0] = 0;
while (i < m) {
if (pattern[i] == pattern[j])
{
   failure[i] = j + 1;
   i++; j++;
}
else if (j > 0) {
   j = failure[j-1];
}
else {
   failure[i]=0;
   i++;
}
```

```
}
}
int knuth_morris_pratt(char *text, char *pattern)
{
int i = 0, j = 0, n = strlen(text), m = strlen(pattern);
int failure[m];
compute_failure_function(pattern, failure);
while (i < n) {
if (text[i] == pattern[j])
{
    if (j == m-1)
    return i - j; i++;
    j++;
}
else if (j > 0) {
j = failure[j-1];
}
else {
i++;
}
}
return -1;
```

```
}

int main()

{

char text[100], pattern[100];

printf("Knuth-Morris-Pratt String Matcher\nEnter the text: ");

gets(text);

printf("Enter the pattern: ");

gets(pattern);

int pos = knuth_morris_pratt(text, pattern);

if (pos == -1)

printf("Pattern not found\n");

else

printf("Pattern found at position %d\n", pos);

return 0;

}
```

Output:

```
Knuth-Morris-Pratt String Matcher

Enter the text: sgfjsdhbvjhsgfkuyvdvfshncdbs

Enter the pattern: shnc

Pattern found at position 21
```

Conclusion: The Rabin-Karp algorithm has the advantage of being able to perform multiple pattern matching with one pre-processing step. This makes it efficient when the same pattern needs to be matched against multiple texts. In contrast, the Knuth-Morris-Pratt algorithm needs to recompute the prefix function for each text, which can be time-consuming.

On the other hand, the Knuth-Morris-Pratt algorithm has the advantage of having a worst-case time complexity of O(n+m), which is better than the worst-case time complexity of the Rabin-Karp algorithm of O(nm). This makes the Knuth-Morris-Pratt algorithm more efficient when the pattern and text are both large.

Another advantage of the Knuth-Morris-Pratt algorithm is that it does not rely on any hash function, unlike the Rabin-Karp algorithm. This makes the Knuth-Morris-Pratt algorithm more reliable when working with string data that may have collisions in their hash values.

-Shashwat Shah

60004220126