Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

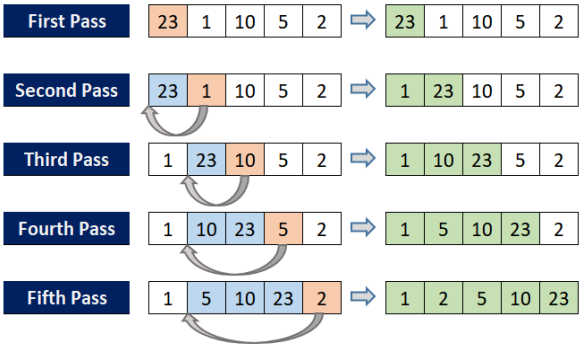
Academic Year: 2022-2023

Name	**Prerna Sunil Jadhav**
Sap Id	**60004220127**
Class	**S. Y. B.Tech (Computer Engineering)**
Course	**Analysis of Algorithm Laboratory**
Course Code	**DJ19CEL404**
Experiment No.	**01-07** (except Exp-06)

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	01

AIM: **IMPLEMENT AND ANALYSE INSERTION AND SELECTION SORT**

THEORY:



- **Insertion sort**
  - Insertion sort is a simple sorting algorithm that works similar to the way you 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.
  - Basically, Insertion sort is efficient for small data value.
  - Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.
  - Pseudocode:

```
procedure insertionSort(A: list of sortable items)
   n = length(A)
   for i = 1 to n - 1 do
       j = i
       while j > 0 and A[j-1] > A[j] do
           swap(A[j], A[j-1])
           j = j - 1
       end while
   end for
end procedure
```

  - Time Complexity:
    - The **worst-case (and average-case) complexity of the insertion sort algorithm is O(n^2).** Meaning that, in the worst case, the time taken to sort a list is proportional to the square of the number of elements in the list.
    - **The best-case time complexity of insertion sort algorithm is O(n)** time complexity. Meaning that the time taken to sort a list is proportional to the number of elements in the list; this is the case when the list is already in the correct order. There's only one iteration in this case since the inner loop operation is trivial when the list is already in order.
  - Space Complexity
    - The insertion sort encompasses a **space complexity of O(1)** due to the usage of an extra variable key.

Code:

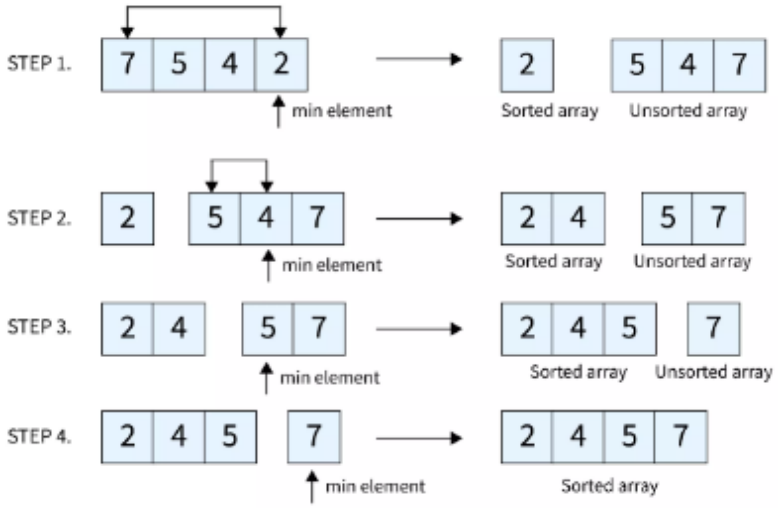
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int arr[10000];

void main()
{
   clock_t start, stop;
   clock_t start_b, stop_b;
   clock_t start_w, stop_w;

   for (int i = 0; i < 10000; i++)
   {
      arr[i] = rand();
   }

   int key, j, n = 10000;
   start = clock();

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

   stop = clock();
   float res = stop - start;
   printf("\nAvg case CPU time =%f units", res);

   start_b = clock();

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

```
    {
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = key;
  }
  stop_b = clock();
  float x = stop_b - start_b;
  printf("\nBest case CPU time =%f units", x);

  start_w = clock();

  for (int i = 1; i < n; i++)
  {
    key = arr[i];
    j = i - 1;
    while (j >= 0 && arr[j] < key)
    {
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = key;
  }
  stop_w = clock();
  x = stop_w - start_w;
  printf("\nworst case CPU time =%f units", x);
}
```

Output:

```
swljx2.5ad' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'

Avg case CPU time =158.000000 units
Best case CPU time =0.000000 units
worst case CPU time =330.000000 units
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

Selection Sort



- Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
- The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted portion.
- This process is repeated for the remaining unsorted portion of the list until the entire list is sorted.
- Pseudocode:

```
function selection sort
   for i = 1 to size - 1
      minimum = i // set current element as minimum
      for j = i+1 to n // check the element to be minimum
         if array[j] < array[minimum] then
            minimum = j;
         end if
      end for
      if indexofMinimum != i then //swap the minimum element with the current
element
         swap array[minimum] and array[i]
      end if
   end for
end function
```

- Time Complexity:
  - Worst Case Complexity: O(n2)
    - ✓ If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
  - Best Case Complexity: O(n2)
    - ✓ It occurs when the array is already sorted
  - Average Case Complexity: O(n2)
    - ✓ It occurs when the elements of the array are in jumbled order (neither ascending nor descending).
- Space Complexity:
  - Space complexity is O(1) because an extra variable is used.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    clock_t start, stop;
    clock_t start_b, stop_b;
    clock_t start_w, stop_w;

    int n = 10000, i, j, position, swap;
    int a[n];
    for (i = 0; i < n; i++)
    {
        a[i] = rand();
    }

    // sorts a jumbled array to give average case time complexity
    start = clock();

    for (i = 0; i < n - 1; i++)
    {
        position = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[position] > a[j])
                position = j;
        }
        if (position != i)
        {
            swap = a[i];
            a[i] = a[position];
            a[position] = swap;
        }
    }
    stop = clock();
    float res = stop - start;
    printf("\nAverage case CPU time =%f units", res);

    // sorts sorted array to give the best case time complexity
    start_b = clock();
    for (i = 0; i < n - 1; i++)
    {
        position = i;
```

```
        for (j = i + 1; j < n; j++)
        {
            if (a[position] > a[j])
                position = j;
        }
        if (position != i)
        {
            swap = a[i];
            a[i] = a[position];
            a[position] = swap;
        }
    }
    stop_b = clock();
    float x = stop_b - start_b;
    printf("\nBest case CPU time =%f units", x);
    // sorts the array in descending order to give worst case time complexity
    start_w = clock();
    for (i = 0; i < n - 1; i++)
    {
        position = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[position] < a[j])
                position = j;
        }
        if (position != i)
        {
            swap = a[i];
            a[i] = a[position];
            a[position] = swap;
        }
    }
    stop_w = clock();
    x = stop_w - start_w;
    printf("\nworst case CPU time =%f units", x);
    return 0;
}
```

OUTPUT:

```
n\gdb.exe' '--interpreter=mi'

Average case CPU time =239.000000 units
Best case CPU time =315.000000 units
worst case CPU time =389.000000 units
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

Sr. No.	Insertion Sort	Selection Sort
1.	The number of comparison operations performed in this sorting algorithm is less than the swapping performed.	The number of comparison operations performed in this sorting algorithm is more than the swapping performed.
2.	It is more efficient than the Selection sort.	It is less efficient than the Insertion sort.
3.	The insertion sort is used when: • The array is has a small number of elements • There are only a few elements left to be sorted	The selection sort is used when • A small list is to be sorted • The cost of swapping does not matter • Checking of all the elements is compulsory • Cost of writing to memory matters like in flash memory (number of Swaps is O(n) as compared to O(n2) of bubble sort)
4.	The insertion sort is Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is **O(kn)** when each element in the input is no more than **k** places away from its sorted position	Selection sort is an in-place comparison sorting algorithm

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	02

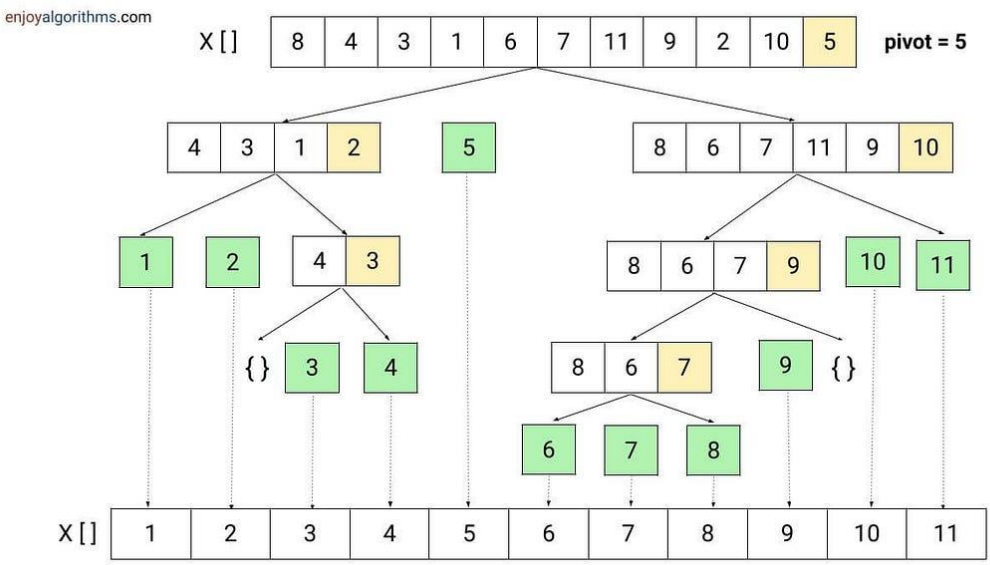
AIM: **IMPLEMENT AND ANALYSE MERGE AND QUICK SORT**

THEORY:

Quick sort

Quicksort is a sorting algorithm based on the divide and conquer approach where

- An array is divided into subarrays by selecting a pivot element (element selected from the array).
- While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
- The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.



- At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

- Algorithm:

```
QUICKSORT(array A, start, end)
{
    if (start < end)
    {
        p = partition(A, start, end)
        QUICKSORT(A, start, p - 1)
        QUICKSORT(A, p + 1, end)
    }
}
```

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

```
PARTITION(array A, start, end)
{
  pivot = A[end]
  i = start-1
  for j = start to end -1
  {
      if (A[j] < pivot) then i = i + 1 swap A[i] with A[j]
  }
  swap A[i + 1] with A[end]
  return i + 1
}
```

- Time Complexities
  - **Worst Case Complexity [Big-O]**: $O(n^2)$
    - ✓ It occurs when the pivot element picked is either the greatest or the smallest element.
    - ✓ This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty, and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.
    - ✓ However, the quicksort algorithm has better performance for scattered pivots.
  - **Best Case Complexity [Big-omega]**: O(n*log n)
    - ✓ It occurs when the pivot element is always the middle element or near to the middle element.
  - **Average Case Complexity [Big-theta]**: O(n*log n)
    - ✓ It occurs when the above conditions do not occur.
- **Space Complexity**
  - The space complexity for quicksort is O(log n).

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Function to swap two elements
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Partition the array using the last element as the pivot
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);
}

// Function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

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

// Driver program
```

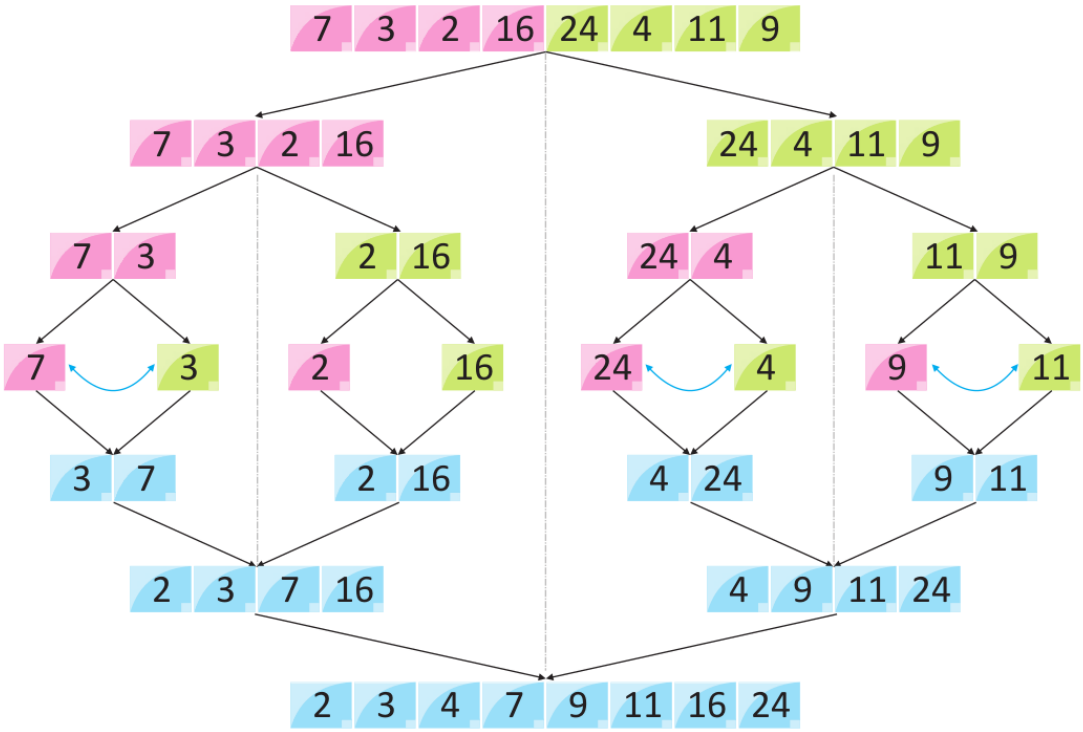
```
int main() {
   clock_t start, stop;
   clock_t start_w, stop_w;

    int n = 5000;
    int arr[n];
    for (int i = 0; i<n; i++){
        arr[i] = rand();
    }

   start = clock();

    quickSort(arr, 0, n - 1);
   stop = clock();
   float res = stop - start;
   printf("\nAverage/best case CPU time =%f units", res);

   start_w = clock();
    //worst case
    quickSort(arr, 0, n - 1);
   stop_w = clock();
   float x = stop_w - start_w;
   printf("\nworst case CPU time =%f units\n", x);

    return 0;
}
```

- Output:

```
jh' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'

Average/best case CPU time =2.000000 units
worst case CPU time =70.000000 units
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

Merge Sort

- Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
- Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided.



- This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion.
- If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves.
- Finally, when both halves are sorted, the merge operation is applied.
- Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.
- Algorithm:

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

```
//for merge function
Have we reached the end of any of the arrays?
    No:
        Compare current elements of both arrays
        Copy smaller element into sorted array
        Move pointer of element containing smaller element
    Yes:
        Copy all remaining elements of non-empty array
```

- Time Complexity:
  - The time complexity of Merge Sort is **θ(n log(n))** in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
- Space Complexity:
  - Space complexity is O(1) because an extra variable is used.

CODE:

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

void mergeDesc(int arr[], int p, int q, int r)
{

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2)
```

```
  {
    if (L[i] >= M[j])
    {
      arr[k] = L[i];
      i++;
    }
    else
    {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1)
  {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2)
  {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Divide the array into two subarrays, sort them and merge them
void mergeSortDesc(int arr[], int l, int r)
{
  if (l < r)
  {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSortDesc(arr, l, m);
    mergeSortDesc(arr, m + 1, r);

    // Merge the sorted subarrays
```

```
    mergeDesc(arr, l, m, r);
  }
}
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r)
{

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2)
  {
    if (L[i] <= M[j])
    {
      arr[k] = L[i];
      i++;
    }
    else
    {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1)
```

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

  while (j < n2)
  {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r)
{
  if (l < r)
  {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}

// Driver program
int main()
{
  clock_t start, stop;
  clock_t start_b, stop_b;
  clock_t start_w, stop_w;

  int size = 10000;

  int arr[size];

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

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

  start = clock();
  // sorting jumbled array
  mergeSort(arr, 0, size - 1);
  stop = clock();
  float res = stop - start;
  printf("\nAvg case CPU time =%f units", res);

  // sorting sorted array
  start_b = clock();
  mergeSort(arr, 0, size - 1);
  stop_b = clock();
  float x = stop_b - start_b;
  printf("\nBest case CPU time =%f units", x);

  // sorting sorted array in descending order
  start_w = clock();
  mergeSortDesc(arr, 0, size - 1);
  stop_w = clock();
  x = stop_w - start_w;
  printf("\nworst case CPU time =%f units", x);
}
```

OUTPUT:

```
ui1ar.1gp' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'

Avg case CPU time =0.000000 units
Best case CPU time =0.000000 units
worst case CPU time =1.000000 units
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

Basis for comparison	**Quick Sort**	**Merge Sort**
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	In the merge sort, the array is parted into just 2 halves (i.e. n/2).

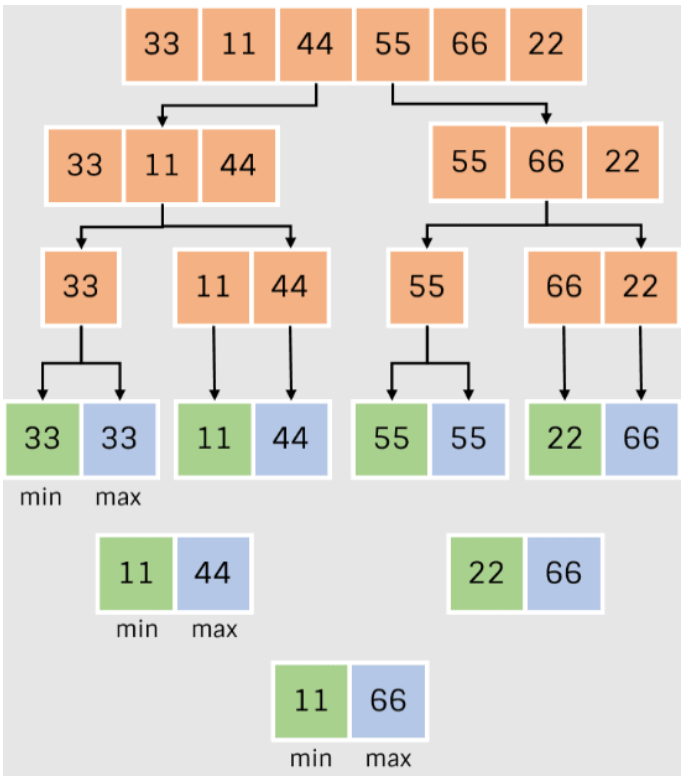
Basis for comparison	Quick Sort	Merge Sort
Worst case complexity	O(n^2)	O(n log n)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Method	Quick sort is in- place sorting method.	Merge sort is not in – place sorting method.
Space	Quicksort does not require additional array space.	For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements.

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	03

AIM: **IMPLEMENT MIN MAX AND BINARY SEARCH USING DIVIDE AND CONQUER APPROACH**

THEORY:

- **MIN-MAX using DIVIDE & CONQUER APPROACH**
  - 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.
  - Algorithm steps:
    Suppose function call minMax (X[], l, r) return maximum and minimum of the array, where l and r are the left and right end.
    - Divide array by calculating mid index i.e. mid = l + (r — l)/2
    - Recursively find the maximum and minimum of left part by calling the same function i.e. leftMinMax[2] = minMax(X, l, mid)
    - Recursively find the maximum and minimum for right part by calling the same function i.e. rightMinMax[2] = minMax(X, mid + 1, r)
    - Finally, get the overall maximum and minimum by comparing the min and max of both halves.
    - Store max and min in output[2] and return it.




```
if (leftMinMax[0] > rightMinMax[0])
    max = lminMax[0]
else
    max = rightMinMax[0]
if (leftMinMax[1] < rightMinMax[1])
    min = leftMinMax[1]
else
    min = rightMinMax[1]
```

- Base case 1: If the array size gets reduced to 1 during recursion, return that single element as both max and min.
- Base case 2: If the array size gets reduced to 2 during recursion, compare both elements and return maximum and minimum.
- Time and Space Complexities
  - The time complexity of the above solution is **O(n),** where n is the size of the input.
  - The auxiliary space required by the program is **O(n)** for recursion (call stack).

Code:

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

#define n 10

int i, a[n];
int max, min;

void maxmin(int a[], 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(a, i, mid);
            max1 = max;
```

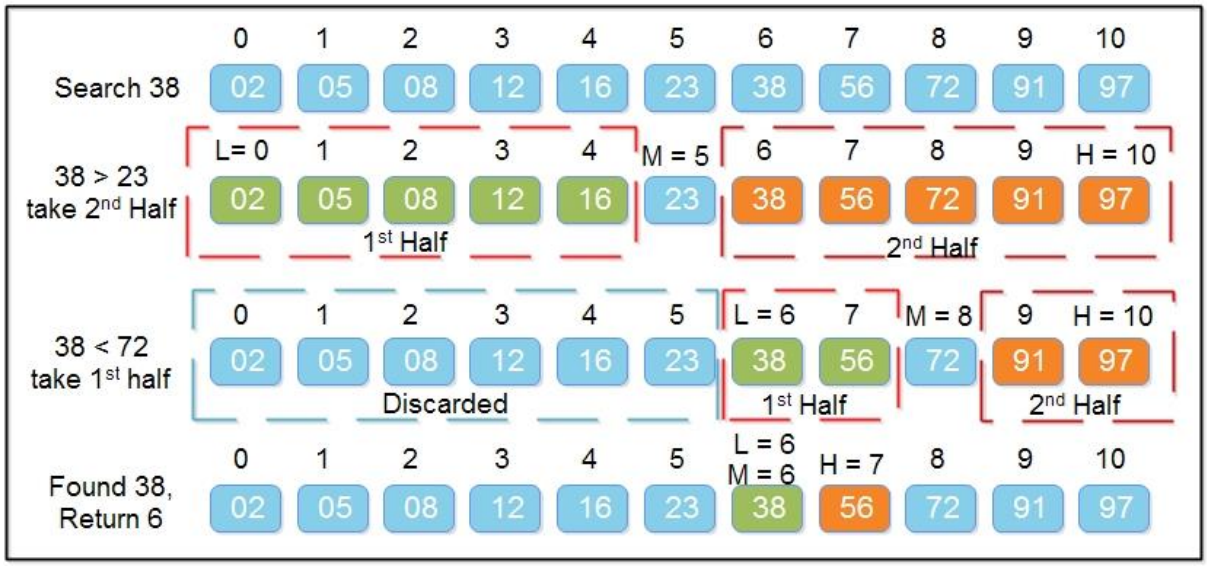
```
            min1 = min;
            maxmin(a, mid + 1, j);
            if (max < max1)
                max = max1;
            if (min > min1)
                min = min1;
        }
    }
}

int main()
{
    for (int i = 0; i < n; i++)
    {
        a[i] = rand();
    }

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

    max = a[0];
    min = a[0];

    maxmin(a, 0, n);
    printf("\nMinimum element in an array : %d\n", min);
    printf("Maximum element in an array : %d\n", max);

    return 0;
}
```

- Output:

```
n\gdb.exe' '--interpreter=mi'
41, 18467, 6334, 26500, 19169, 15724, 11478, 29358, 26962, 24464,
Minimum element in an array : 41
Maximum element in an array : 29358
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

BINARY SEARCH

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



- Algorithm:

```
Sort the array in ascending order.
Set the low index to the first element of the array and the high index to the last element.
Set the middle index to the average of the low and high indices.
If the element at the middle index is the target element, return the middle index.
If the target element is less than the element at the middle index, set the high index to the middle index - 1.
If the target element is greater than the element at the middle index, set the low index to the middle index + 1.
Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.
```

- Time Complexity:
  - The time complexity of the binary search algorithm is O(log n).
  - The best-case time complexity would be O(1) when the central index would directly match the desired value.
  - Binary search worst case differs from that. The worst-case scenario could be the values at either extremity of the list or values not in the list.

CODE:

```c
#include <stdio.h>
int binarySearch(int a[], int low, int high, int key){
    if (high >= low){
        int mid = low + (high-low)/2;
        if (key == a[mid]){
            return mid;
        }
        else if(a[mid] > key){
            binarySearch(a, low, mid-1, key);
        }
        else{
            binarySearch(a, mid+1, high, key);
        }
    }
    else{
        return -1;
    }
}
int main(){
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    int low = 0;
    int high = 9;
    printf("Enter any element: ");
    int num=0;
    scanf("%d", &num);
    int flag = binarySearch(a, low, high, num);
    if (flag == -1){
        printf("Element not found");
    }
    else{
        printf("Element found at %d index",flag);
    }
    return 0;
}
```

OUTPUT:

```
Enter any element: 4
Element found at 3 index
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>  & 'c:\Users\Jadhav\.vsc
```

```
Enter any element: 32
Element not found
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

A binary search algorithm has many benefits:

- Using each comparison eliminates half of the list that is not needed for further searches.
- This indicates whether the element to be searched is located before or after the current position within the list.
- This information can be used to limit your search.
- It works much better than linear searches for large data sets.

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	04

AIM: **IMPLEMENT SINGLE SOURCE SHORTEST PATH USING GREEDY APPROACH**

THEORY:

- Dijkstra Algorithm is a very famous greedy algorithm.
- It is used for solving the single source shortest path problem.
- It computes the shortest path from one particular source node to all other remaining nodes of the graph.
- Conditions:
  - Dijkstra algorithm works only for connected graphs.
  - Dijkstra algorithm works only for those graphs that do not contain any negative weight edge.
  - It only provides the value or cost of the shortest paths.
  - By making minor modifications in the actual algorithm, the shortest paths can be easily obtained.
  - Dijkstra algorithm works for directed as well as undirected graphs.
- **Algorithm:**

```
Algorithm: Dijkstra's-Algorithm (G, w, s)
for each vertex v ∈ G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
   u := Extract-Min (Q)
   S := S U {u}
   for each vertex v ∈ G.adj[u]
      if v.d > u.d + w(u, v)
         v.d := u.d + w(u, v)
         v.∏ := u
```

- First for loop does initialization in O(|V|) time. As there are |V| nodes in the graph, size of queue Q would be V, and hence while loop iterates |V| times in worst case. For loop inside while loop run maximum |V| time because a node can have maximum |V| – 1 neighbour. The worst case upper bound running time of this algorithm is described as O(|V2|).

CODE:

```
// Dijkstra's Algorithm in C

#include <stdio.h>
#define INFINITY 9999
#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {
  int cost[MAX][MAX], distance[MAX], pred[MAX];
  int visited[MAX], count, mindistance, nextnode, i, j;

  // Creating cost matrix
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (Graph[i][j] == 0)
        cost[i][j] = INFINITY;
      else
        cost[i][j] = Graph[i][j];

  for (i = 0; i < n; i++) {
    distance[i] = cost[start][i];
    pred[i] = start;
    visited[i] = 0;
  }

  distance[start] = 0;
  visited[start] = 1;
  count = 1;

  while (count < n - 1) {
    mindistance = INFINITY;

    for (i = 0; i < n; i++)
      if (distance[i] < mindistance && !visited[i]) {
        mindistance = distance[i];
        nextnode = i;
      }

    visited[nextnode] = 1;
    for (i = 0; i < n; i++)
      if (!visited[i])
        if (mindistance + cost[nextnode][i] < distance[i]) {
```

```
          distance[i] = mindistance + cost[nextnode][i];
          pred[i] = nextnode;
        }
    count++;
  }

  // Printing the distance
  for (i = 0; i < n; i++)
    if (i != start) {
      printf("\nDistance from source to %d: %d", i, distance[i]);
    }
}
int main() {
  int Graph[MAX][MAX], i, j, n, u;
  n = 7;

  Graph[0][0] = 0;
  Graph[0][1] = 0;
  Graph[0][2] = 1;
  Graph[0][3] = 2;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 0;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
  Graph[1][5] = 3;
  Graph[1][6] = 0;

  Graph[2][0] = 1;
  Graph[2][1] = 2;
  Graph[2][2] = 0;
  Graph[2][3] = 1;
  Graph[2][4] = 3;
  Graph[2][5] = 0;
  Graph[2][6] = 0;

  Graph[3][0] = 2;
  Graph[3][1] = 0;
  Graph[3][2] = 1;
  Graph[3][3] = 0;
```

```
  Graph[3][4] = 0;
  Graph[3][5] = 0;
  Graph[3][6] = 1;

  Graph[4][0] = 0;
  Graph[4][1] = 0;
  Graph[4][2] = 3;
  Graph[4][3] = 0;
  Graph[4][4] = 0;
  Graph[4][5] = 2;
  Graph[4][6] = 0;

  Graph[5][0] = 0;
  Graph[5][1] = 3;
  Graph[5][2] = 0;
  Graph[5][3] = 0;
  Graph[5][4] = 2;
  Graph[5][5] = 0;
  Graph[5][6] = 1;

  Graph[6][0] = 0;
  Graph[6][1] = 0;
  Graph[6][2] = 0;
  Graph[6][3] = 1;
  Graph[6][4] = 0;
  Graph[6][5] = 1;
  Graph[6][6] = 0;

  u = 0;
  Dijkstra(Graph, n, u);

  return 0;
}
```

OUTPUT:

```
2mmgck.uqi' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'

Distance from source to 1: 3
Distance from source to 2: 1
Distance from source to 3: 2
Distance from source to 4: 4
Distance from source to 5: 4
Distance from source to 6: 3
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

Dijkstra's Algorithm Applications

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	05

AIM: **IMPLEMENT MINIMUM SPANNING TREE (PRIM'S AND KRUSKAL)**

THEORY:

PRIM'S ALGORITHM

- Prim's algorithm is 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.
- The algorithm starts with an empty spanning tree.
- The idea is to maintain two sets of vertices.
- The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges.
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.
- **Pseudocode:**

```
T = ∅;
U = { 1 };
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
    T = T ∪ {(u, v)}
    U = U ∪ {v}
```

CODE:

```
// Prim's Algorithm in C

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

#define INF 9999999

// number of vertices in graph
#define V 5

// create a 2d array of size 5x5
//for adjacency matrix to represent graph
int G[V][V] = {
```

```
  {0, 9, 75, 0, 0},
  {9, 0, 95, 19, 42},
  {75, 95, 0, 51, 66},
  {0, 19, 51, 0, 31},
  {0, 42, 66, 31, 0}};

int main() {
  int no_edge;  // number of edge

  // create a array to track selected vertex
  // selected will become true otherwise false
  int selected[V];

  // set selected false initially
  memset(selected, false, sizeof(selected));

  // set number of edge to 0
  no_edge = 0;

  // the number of egde in minimum spanning tree will be
  // always less than (V -1), where V is number of vertices in
  //graph

  // choose 0th vertex and make it true
  selected[0] = true;

  int x;  //  row number
  int y;  //  col number

  // print for edge and weight
  printf("Edge : Weight\n");

  while (no_edge < V - 1) {
    //For every vertex in the set S, find the all adjacent vertices
    // , calculate the distance from the vertex selected at step 1.
    // if the vertex is already in the set S, discard it otherwise
    //choose another vertex nearest to selected vertex  at step 1.

    int min = INF;
    x = 0;
    y = 0;

    for (int i = 0; i < V; i++) {
      if (selected[i]) {
```

```
        for (int j = 0; j < V; j++) {
          if (!selected[j] && G[i][j]) {  // not in selected and there is an
edge
            if (min > G[i][j]) {
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
    printf("%d - %d : %d\n", x, y, G[x][y]);
    selected[y] = true;
    no_edge++;
  }
  return 0;
}
```

OUTPUT:

```
swnujq.shi' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'
Edge : Weight
0 - 1 : 9
1 - 3 : 19
3 - 4 : 31
3 - 2 : 51
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

KRUSHKAL'S ALGORITHM

- Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which
  - form a tree that includes every vertex
  - has the minimum sum of weights among all the trees that can be formed from the graph
- Kruskal's algorithm working
  - It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.
  - We start from the edges with the lowest weight and keep adding edges until we reach our goal.
  - The steps for implementing Kruskal's algorithm are as follows:
    - Sort all the edges from low weight to high
    - Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
    - Keep adding edges until we reach all vertices.

- Pseudocode

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
return A
```

CODE:

```
// Kruskal's algorithm in C

#include <stdio.h>

#define MAX 30

typedef struct edge {
  int u, v, w;
} edge;

typedef struct edge_list {
  edge data[MAX];
  int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();

// Applying Krushkal Algo
void kruskalAlgo() {
  int belongs[MAX], i, j, cno1, cno2;
  elist.n = 0;

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

```
    for (j = 0; j < i; j++) {
      if (Graph[i][j] != 0) {
        elist.data[elist.n].u = i;
        elist.data[elist.n].v = j;
        elist.data[elist.n].w = Graph[i][j];
        elist.n++;
      }
    }

  sort();

  for (i = 0; i < n; i++)
    belongs[i] = i;

  spanlist.n = 0;

  for (i = 0; i < elist.n; i++) {
    cno1 = find(belongs, elist.data[i].u);
    cno2 = find(belongs, elist.data[i].v);

    if (cno1 != cno2) {
      spanlist.data[spanlist.n] = elist.data[i];
      spanlist.n = spanlist.n + 1;
      applyUnion(belongs, cno1, cno2);
    }
  }
}

int find(int belongs[], int vertexno) {
  return (belongs[vertexno]);
}

void applyUnion(int belongs[], int c1, int c2) {
  int i;

  for (i = 0; i < n; i++)
    if (belongs[i] == c2)
      belongs[i] = c1;
}

// Sorting algo
void sort() {
  int i, j;
  edge temp;
```

```
  for (i = 1; i < elist.n; i++)
    for (j = 0; j < elist.n - 1; j++)
      if (elist.data[j].w > elist.data[j + 1].w) {
        temp = elist.data[j];
        elist.data[j] = elist.data[j + 1];
        elist.data[j + 1] = temp;
      }
}

// Printing the result
void print() {
  int i, cost = 0;

  for (i = 0; i < spanlist.n; i++) {
    printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v,
spanlist.data[i].w);
    cost = cost + spanlist.data[i].w;
  }

  printf("\nSpanning tree cost: %d", cost);
}

int main() {
  int i, j, total_cost;

  n = 6;

  Graph[0][0] = 0;
  Graph[0][1] = 4;
  Graph[0][2] = 4;
  Graph[0][3] = 0;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 4;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
  Graph[1][5] = 0;
  Graph[1][6] = 0;
```

```
  Graph[2][0] = 4;
  Graph[2][1] = 2;
  Graph[2][2] = 0;
  Graph[2][3] = 3;
  Graph[2][4] = 4;
  Graph[2][5] = 0;
  Graph[2][6] = 0;

  Graph[3][0] = 0;
  Graph[3][1] = 0;
  Graph[3][2] = 3;
  Graph[3][3] = 0;
  Graph[3][4] = 3;
  Graph[3][5] = 0;
  Graph[3][6] = 0;

  Graph[4][0] = 0;
  Graph[4][1] = 0;
  Graph[4][2] = 4;
  Graph[4][3] = 3;
  Graph[4][4] = 0;
  Graph[4][5] = 0;
  Graph[4][6] = 0;

  Graph[5][0] = 0;
  Graph[5][1] = 0;
  Graph[5][2] = 2;
  Graph[5][3] = 0;
  Graph[5][4] = 3;
  Graph[5][5] = 0;
  Graph[5][6] = 0;

  kruskalAlgo();
  print();
}
```

OUTPUT:

```
2 - 1 : 2
5 - 2 : 2
3 - 2 : 3
4 - 3 : 3
1 - 0 : 4
Spanning tree cost: 14
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

- Kruskal's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph.
- Instead of starting from a vertex, Kruskal's algorithm sorts all the edges from low weight to high and keeps adding the lowest edges, ignoring those edges that create a cycle.
- The time complexity of Prim's algorithm is O(E log V).

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	07

AIM: **IMPLEMENT SINGLE SOURCE SHORTEST PATH USING DYNAMIC PROGRAMMING**

THEORY:

BELLMAN-FORD ALGORITHM

- 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.
- **Pseudocode:**

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0

  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U

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

  return distance[], previous[]
```

CODE:

```
// Bellman Ford Algorithm in C

#include <stdio.h>
```

```c
#include <stdlib.h>

#define INFINITY 99999

//struct for the edges of the graph
struct Edge {
  int u;  //start vertex of the edge
  int v;  //end vertex of the edge
  int w;  //weight of the edge (u,v)
};

//Graph - it consists of edges
struct Graph {
  int V;        //total number of vertices in the graph
  int E;        //total number of edges in the graph
  struct Edge *edge;  //array of edges
};

void bellmanford(struct Graph *g, int source);
void display(int arr[], int size);

int main(void) {
  //create graph
  struct Graph *g = (struct Graph *)malloc(sizeof(struct Graph));
  g->V = 4;  //total vertices
  g->E = 5;  //total edges

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

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

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

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

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

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

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

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

  bellmanford(g, 0);  //0 is the source vertex

  return 0;
}

void bellmanford(struct Graph *g, int source) {
  //variables
  int i, j, u, v, w;

  //total vertex in the graph g
  int tV = g->V;

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

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

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

  //step 1: fill the distance array and predecessor array
```

```
  for (i = 0; i < tV; i++) {
    d[i] = INFINITY;
    p[i] = 0;
  }

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

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

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

  //step 3: detect negative cycle
  //if value changes then we have a negative cycle in the graph
  //and we cannot find the shortest distances
  for (i = 0; i < tE; i++) {
    u = g->edge[i].u;
    v = g->edge[i].v;
    w = g->edge[i].w;
    if (d[u] != INFINITY && d[v] > d[u] + w) {
      printf("Negative weight cycle detected!\n");
      return;
    }
  }

  //No negative weight cycle found!
  //print the distance and predecessor array
  printf("Distance array: ");
  display(d, tV);
  printf("Predecessor array: ");
  display(p, tV);
}
```

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

OUTPUT:

```
uq3qjk.vwj' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'
Distance array: 0 5 4 8
Predecessor array: 0 0 0 1
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

- **Time Complexity**

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

- **Space Complexity**
  The space complexity is O(V).

- Bellman Ford's Algorithm Applications
  - For calculating shortest paths in routing algorithms
  - For finding the shortest path