DESIGN AND ANALYSIS OF ALGORITHMS

ITC17

DEPARTMENT OF INFORMATION TECHNOLOGY

NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

SHIVANI GUPTA

2018UIT2586

IT-2

INDEX

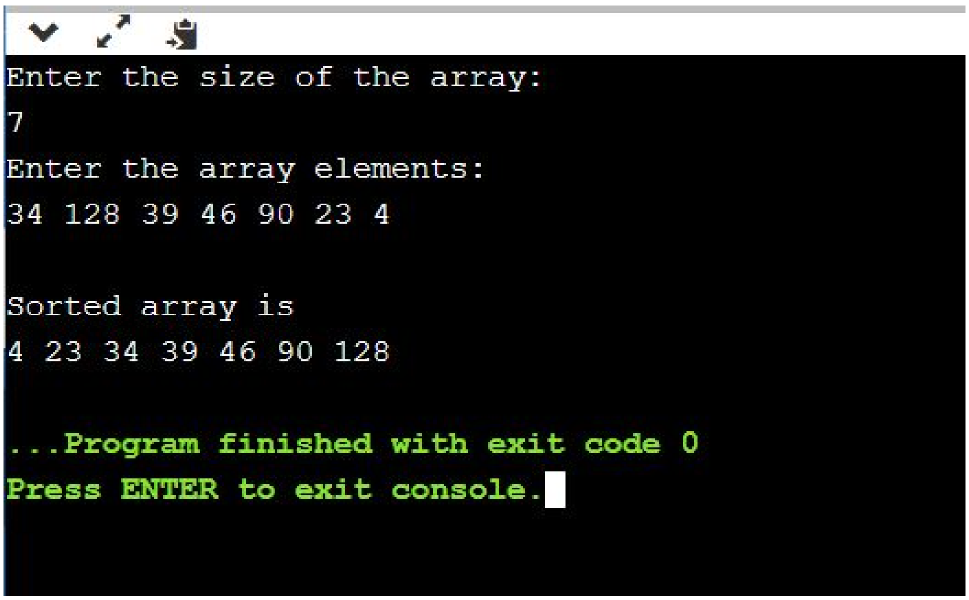
S.No.	Program Name	Pg.No.	Date	Teachers Signature
1	Sorting Algorithms 1. Merge Sort 2. Quick Sort 3. Bubble Sort 4. Bucket Sort 5. Radix Sort 6. Shell Sort 7. Selection Sort 8. Heap Sort	4-24	11/08/2020	
2	Searching Algorithms 1. Linear Search 2. Binary Search	25-28	20/08/2020	
3	Matrix Chain Multiplication	29-30	26/08/2020	
4	Strassen's Matrix Multiplication	31-33	26/08/2020	
5	Longest Common Subsequence	34-35	28/08/2020	
6	Optimal Binary Search Tree	36-37	04/09/2020	
7	Huffman Coding	38-40	17/09/2020	
8	Dijkstra's Algorithm	41-43	23/09/2020	
9	Bellman Ford Algorithm	44-45	23/09/2020	
10	String Matching Algorithms 1. Naive String Matching Algorithm 2. Rabin Karp Algorithm 3. Knuth Morris Pratt	46-50	29/09/2020	

MERGE SORT

```
#include<bits/stdc++.h>
using namespace std;
//function to merge two sorted arrays into one array
void merge(int array[], int low, int middle, int high)
{
    //indices for two halves
   int n1 = middle - low + 1;
   int n2 = high - middle;
   int Left[n1], Right[n2];

   for(int i = 0; i < n1; i++)
        Left[i] = array[low + i];

   for(int j = 0; j < n2; j++)
        Right[j] = array[middle + 1 + j];
   // Initial index of first subarray
   int i = 0;

   // Initial index of second subarray
   int j = 0;

   // Initial index of merged subarray
   int k = low;

   while (i < n1 && j < n2)
   {
        if (Left[i] <= Right[j])
        {
             array[k] = Left[i];
             i++;
        }
        else
        {
             array[k] = Right[j];
             j++;
        }
        k++;
   }
 //input the elements left in left array
   while (i < n1)
   {
```

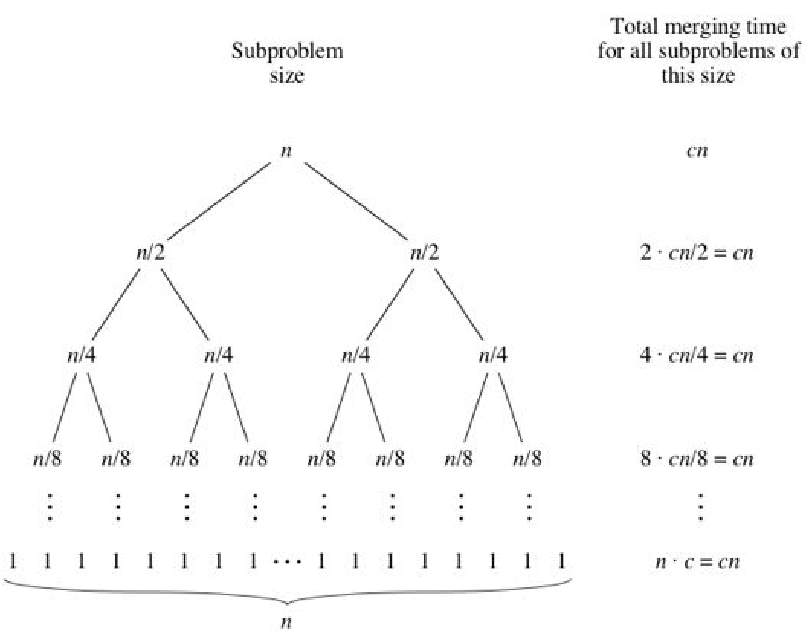
```
            array[k] = Left[i];
            i++;
            k++;
   }
//input the elements left in right array
   while (j < n2)
   {
            array[k] = Right[j];
            j++;
            k++;
   }
}

void mergeSort(int array[], int low, int high)
{
   if (low < high)
   {
            int middle = low + (high - low) / 2;
        //sort left half of array
            mergeSort(array, low, middle);
            //sort right half of array
            mergeSort(array, middle + 1, high);
        //merge function merges the sorted subarrays
            merge(array, low, middle, high);
   }
}
//function to print the array elements
void printArray(int A[], int size)
{
   for(int i = 0; i < size; i++)
            cout << A[i] << " ";
}

int main()
{
    //take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size;
   cin>>size;
   //declaration of array
   int array[size];
   //input the elements of the array
   cout << "Enter the array elements: \n";
   for(int i=0;i<size;i++) cin>>array[i];
```

```
    //function call
    mergeSort(array, 0, size - 1);

    cout << "\nSorted array is \n";
    //output the sorted array
    printArray(array, size);
    return 0;
}
```


Enter the size of the array:
7
Enter the array elements:
34 128 39 46 90 23 4

Sorted array is
4 23 34 39 46 90 128

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

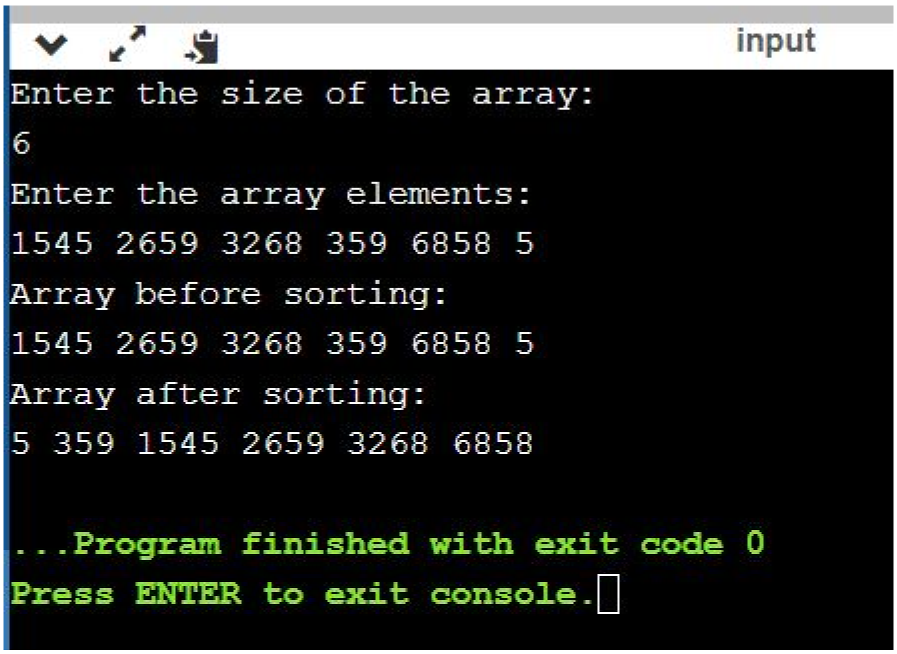


TIME COMPLEXITY - O(NlogN)
SPACE COMPLEXITY - O(N)

SHELL SORT

```
#include <iostream>
using namespace std;

int shellSort(int arr[], int n)
{
   // Start with a big gap, then reduce the gap
   for (int gap = n/2; gap > 0; gap /= 2)
   {
            for (int i = gap; i < n; i += 1)
            {
                        // add a[i] to the elements that have been gap sorted
                        // save a[i] in temp and make a hole at position i
                        int temp = arr[i];

                        // shift earlier gap-sorted elements up until the correct
                        // location for a[i] is found
                        int j;
                        for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                                    arr[j] = arr[j - gap];

                        // put temp (the original a[i]) in its correct location
                        arr[j] = temp;
            }
   }
   return 0;
}
void printArray(int arr[], int n)
{
   for (int i=0; i<n; i++)
            cout << arr[i] << " ";
}
int main()
{
    //take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size;
   cin>>size;
   //declaration of array
   int array[size];
   //input the elements of the array
   cout << "Enter the array elements: \n";
   for(int i=0;i<size;i++) cin>>array[i];
```

```
    cout << "Array before sorting: \n";
    printArray(array, size);

    shellSort(array, size);

    cout << "\nArray after sorting: \n";
    printArray(array, size);
    return 0;
}
```



```
input
Enter the size of the array:
6
Enter the array elements:
1545 2659 3268 359 6858 5
Array before sorting:
1545 2659 3268 359 6858 5
Array after sorting:
5 359 1545 2659 3268 6858

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

TIME COMPLEXITY - Worst Case Complexity: less than or equal to **O(n^2)**

Best Case Complexity: **O(n*log n)**

When the array is already sorted, the total number of comparisons for each interval (or increment) is equal to the size of the array.

Average Case Complexity: **O(n*log n)**

It is around O(n1.25)

SPACE COMPLEXITY - O(1)

QUICK SORT

```
#include <bits/stdc++.h>
using namespace std;

//function to swap two elements
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}

int partition (int array[], int low, int high)
{
   int pivot = array[high]; // pivot
   int i = (low - 1); // Index of smaller element

   for (int j = low; j <= high - 1; j++)
   {
        // If current element is smaller than the pivot
        if (array[j] < pivot)
        {
             i++; // increment index of smaller element
             swap(&array[i], &array[j]);
        }
   }
   swap(&array[i + 1], &array[high]);
   return (i + 1);
}

void quickSort(int array[], int low, int high)
{
   if (low < high)
   {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(array, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(array, low, pi - 1);
        quickSort(array, pi + 1, high);
   }
}
```

```
/* Function to print an array */
void printArray(int array[], int size)
{
   int i;
   for (i = 0; i < size; i++)
              cout << array[i] << " ";
   cout << endl;
}

int main()
{
   //take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size;
   cin>>size;
   //declaration of array
   int array[size];
   //input the elements of the array
   cout << "Enter the array elements: \n";
   for(int i=0;i<size;i++) cin>>array[i];

   quickSort(array, 0, size - 1);
   cout << "Sorted array: \n";
   printArray(array, size);
   return 0;
}
```

```
Enter the size of the array:
5
Enter the array elements:
2387 23 213 98 3
Sorted array:
3 23 98 213 2387


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

TIME COMPLEXITY OF QUICK SORT

Lemma 2.14 (Textbook): The worst-case time complexity of quicksort is $\Omega(n^2)$.

Proof. The partitioning step: at least, $n-1$ comparisons.

- At each next step for $n \geq 1$, the number of comparisons is one less, so that $T(n) = T(n-1) + (n-1)$; $T(1) = 0$.
- "Telescoping" $T(n) - T(n-1) = n - 1$:

$$\begin{aligned} & T(n)+T(n-1)+T(n-2)+\ldots+T(3)+T(2) \\ & \quad -T(n-1)-T(n-2)-\ldots-T(3)-T(2)-T(1) \\ & \quad = (n-1) + (n-2) + \ldots + 2 + 1 - 0 \\ & T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{(n-1)n}{2} \end{aligned}$$

This yields that $T(n) \in \Omega(n^2)$.

Analysing Quicksort: The Average Case $T(n) \in \Theta(n \log n)$

For any pivot position i; $i \in \{0, \ldots, n-1\}$:

- Time for partitioning an array : cn
- The head and tail subarrays contain i and $n-1-i$ items, respectively: $T(n) = cn + T(i) + T(n-1-i)$

Average running time for sorting (a more complex recurrence):

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} \left(T(i) + T(n-1-i) + cn\right)$$

$$= \frac{2}{n} \left(T(0) + T(1) + \ldots + T(n-2) + T(n-1)\right) + cn, \text{ or}$$

$$nT(n) = 2\left(T(0) + T(1) + \ldots + T(n-2) + T(n-1)\right) + cn^2$$

$$(n-1)T(n-1) = 2\left(T(0) + T(1) + \ldots + T(n-2)\right) + c(n-1)^2$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c \approx 2T(n-1) + 2cn$$

Thus, $nT(n) \approx (n+1)T(n-1) + 2cn$, or $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$

Analysing Quicksort: The Average Case $T(n) \in \Theta(n \log n)$

"Telescoping" $\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2c}{n+1}$ to get the explicit form:

$$
\begin{aligned}
&\frac{T(n)}{n+1} + \frac{T(n-1)}{n} + \frac{T(n-2)}{n-1} + \ldots + \frac{T(2)}{3} + \frac{T(1)}{2} \\
&\quad - \frac{T(n-1)}{n} - \frac{T(n-2)}{n-1} - \ldots - \frac{T(2)}{3} - \frac{T(1)}{2} - \frac{T(0)}{1} \\
&\quad = \frac{2c}{n+1} + \frac{2c}{n} + \ldots + \frac{2c}{3} + \frac{2c}{2}, \text{ or}
\end{aligned}
$$

$$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} + \frac{1}{n+1}\right) \approx 2c(H_{n+1} - 1) \approx c' \log n$$

($H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n + 0.577$ is the n^{th} harmonic number).

Therefore, $T(n) \approx c'(n+1) \log n \in \Theta(n \log n)$.

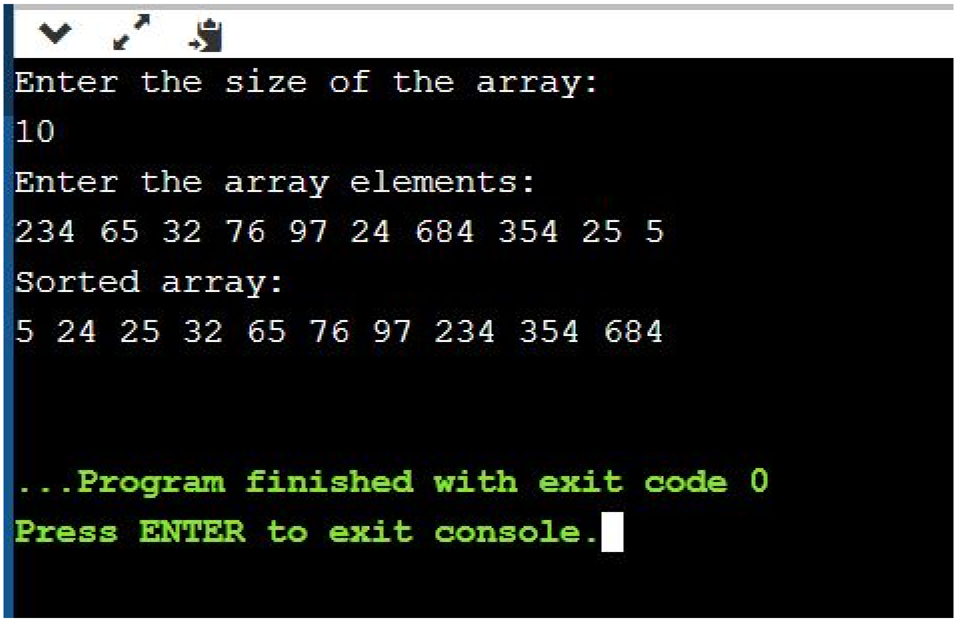
Quicksort is our first example of dramatically different worst-case and average-case performances!

SPACE TIME COMPLEXITY: **O(NlogN)**

BUBBLE SORT

```
#include <bits/stdc++.h>
using namespace std;
//Function to swap two elements
void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

void bubbleSort(int array[], int size)
{
   int i, j;
   for (i = 0; i < size-1; i++)

   // Last i elements are already in place
   for (j = 0; j < size-i-1; j++)
           if (array[j] > array[j+1])
                   swap(&array[j], &array[j+1]);
}

/* Function to print an array */
void printArray(int array[], int size)
{
   int i;
   for (i = 0; i < size; i++)
           cout << array[i] << " ";
   cout << endl;
}

int main()
{
   //take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size;
   cin>>size;
   //declaration of array
   int array[size];
   //input the elements of the array
   cout << "Enter the array elements: \n";
   for(int i=0;i<size;i++) cin>>array[i];

   bubbleSort(array, size);
```

```
   cout<<"Sorted array: \n";
   printArray(array, size);
   return 0;
}
```



COMPLEXITY ANALYSIS OF BUBBLE SORT

TIME COMPLEXITY : In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

```
(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1
Sum = n(n-1)/2
i.e O(n2)
```

Hence the **time complexity** of Bubble Sort is **O(n^2)**

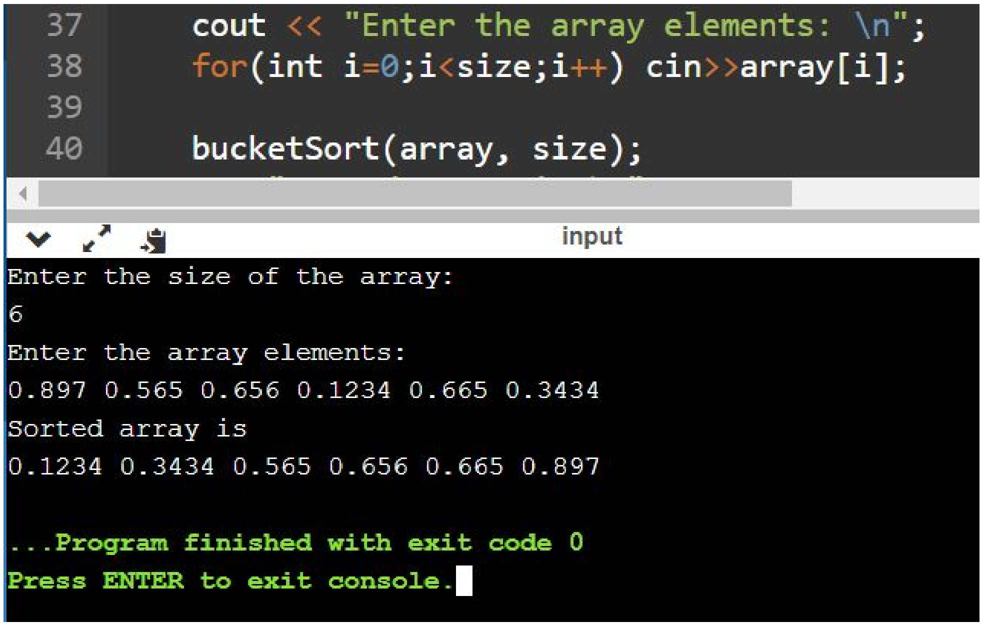
Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: O(n^2)
- Best Case Time Complexity [Big-omega]: O(n)
- Average Time Complexity [Big-theta]: O(n^2)
- Space Complexity: O(1)

BUCKET SORT

```
// C++ program to sort an array using bucket sort
#include<bits/stdc++.h>
using namespace std;

// Function to sort arr[] of size n using bucket sort
void bucketSort(float arr[], int n)
{
  // 1) Create n empty buckets
  vector<float> b[n];

  // 2) Put array elements in different buckets
  for (int i = 0; i < n; i++) {
          int bi = n * arr[i]; // Index in bucket
          b[bi].push_back(arr[i]);
  }

  // 3) Sort individual buckets
  for (int i = 0; i < n; i++)
          sort(b[i].begin(), b[i].end());

  // 4) Concatenate all buckets into arr[]
  int index = 0;
  for (int i = 0; i < n; i++)
          for (int j = 0; j < b[i].size(); j++)
                  arr[index++] = b[i][j];
}

int main()
{
//take input of size of array
  cout<<"Enter the size of the array:"<<endl;
  int size;
  cin>>size;
  //declaration of array
  float array[size];
  //input the elements of the array
  cout << "Enter the array elements: \n";
  for(int i=0;i<size;i++) cin>>array[i];

  bucketSort(array, size);
cout << "Sorted array is \n";
   for (int i = 0; i < size; i++)
      cout << array[i] << " ";
```

```
    return 0;

}
```



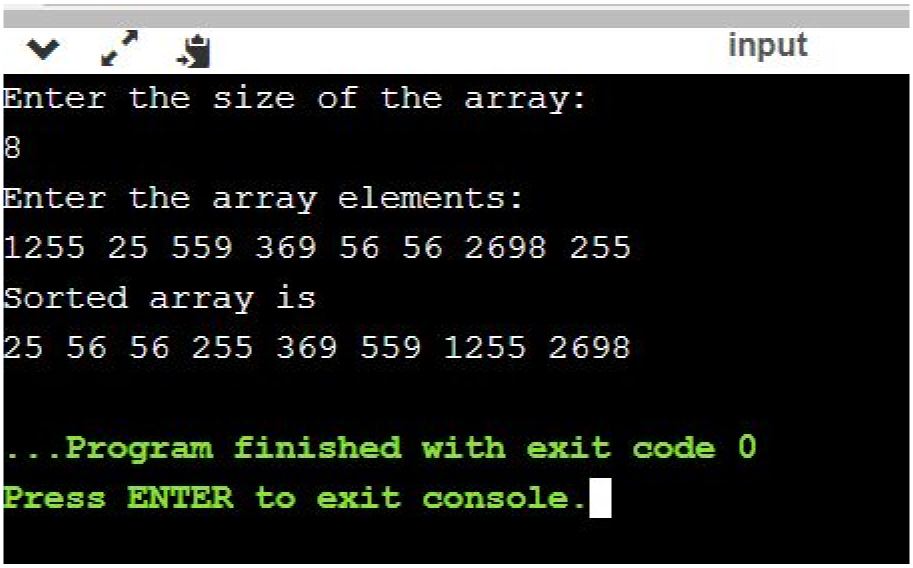
COMPLEXITY ANALYSIS OF BUCKET SORT

- **Time Complexity: O(n + k)** for best case and average case and O(n^2) for the worst case.
- **Space Complexity: O(nk)** for worst case

RADIX SORT

```
// C++ implementation of Radix Sort
#include <iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
   int mx = arr[0];
   for (int i = 1; i < n; i++)
           if (arr[i] > mx)
                   mx = arr[i];
   return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
   int output[n]; // output array
   int i, count[10] = { 0 };

   // Store count of occurrences in count[]
   for (i = 0; i < n; i++)
           count[(arr[i] / exp) % 10]++;

   // Change count[i] so that count[i] now contains actual
   // position of this digit in output[]
   for (i = 1; i < 10; i++)
           count[i] += count[i - 1];

   // Build the output array
   for (i = n - 1; i >= 0; i--) {
           output[count[(arr[i] / exp) % 10] - 1] = arr[i];
           count[(arr[i] / exp) % 10]--;
   }

   // Copy the output array to arr[], so that arr[] now
   // contains sorted numbers according to current digit
   for (i = 0; i < n; i++)
           arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
```

```
// Radix Sort
void radixsort(int arr[], int n)
{
   // Find the maximum number to know number of digits
   int m = getMax(arr, n);

   // Do counting sort for every digit. Note that instead
   // of passing digit number, exp is passed. exp is 10^i
   // where i is current digit number
   for (int exp = 1; m / exp > 0; exp *= 10)
           countSort(arr, n, exp);
}

// A utility function to print an array
void print(int arr[], int n)
{
   for (int i = 0; i < n; i++)
           cout << arr[i] << " ";
}

// Driver Code
int main()
{
//take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size;
   cin>>size;
   //declaration of array
   reinterpret_cast array[size];
   //input the elements of the array
   cout << "Enter the array elements: \n";
   for(int i=0;i<size;i++) cin>>array[i];

   radixsort(array, size);
cout << "Sorted array is \n";
    for (int i = 0; i < size; i++)
       cout << array[i] << " ";
   return 0;

}
```



```
input
Enter the size of the array:
8
Enter the array elements:
1255 25 559 369 56 56 2698 255
Sorted array is
25 56 56 255 369 559 1255 2698

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

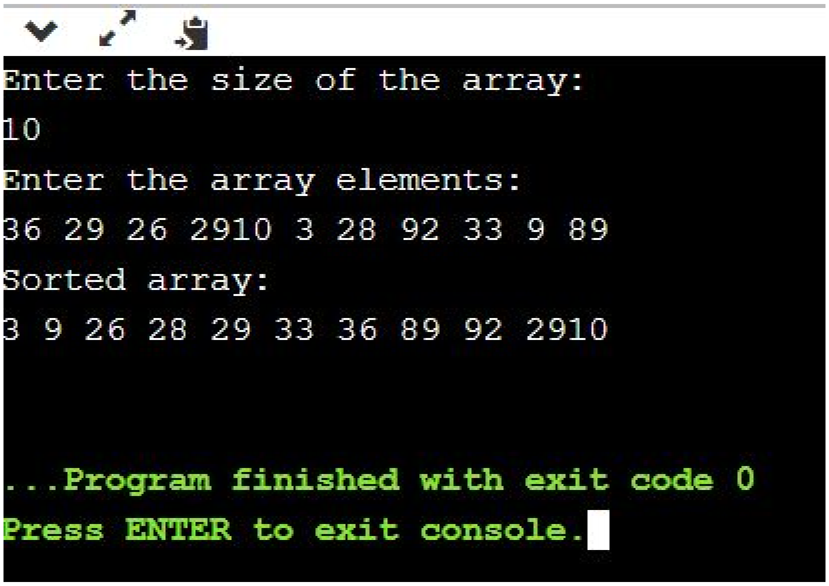
COMPLEXITY ANALYSIS OF RADIX SORT :

For the radix sort that uses counting sort as an intermediate stable sort, the **time complexity is `O(d(n+k))`.**
Here, `d` is the number cycle and `O(n+k)` is the time complexity of counting sort.

SELECTION SORT

```
#include <bits/stdc++.h>
using namespace std;
//function to swap two elements
void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

void selectionSort(int array[], int size)
{
   int i, j, min_index;

   // One by one move boundary of unsorted subarray
   for (i = 0; i < size-1; i++)
   {
          // Find the minimum element in unsorted array
          min_index = i;
          for (j = i+1; j < size; j++)
          if (array[j] < array[min_index])
                 min_index = j;

          // Swap the found minimum element with the first element
          swap(&array[min_index], &array[i]);
   }
}

/* Function to print an array */
void printArray(int array[], int size)
{
   int i;
   for (i=0; i < size; i++)
          cout << array[i] << " ";
   cout << endl;
}

int main()
{
    //take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size;
   cin>>size;
```

```
    //declaration of array
    int array[size];
    //input the elements of the array
    cout << "Enter the array elements: \n";
    for(int i=0;i<size;i++) cin>>array[i];

    selectionSort(array, size);
    cout << "Sorted array: \n";
    printArray(array, size);
    return 0;
}
```



COMPLEXITY ANALYSIS OF SELECTION SORT

Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **$O(n^2)$**

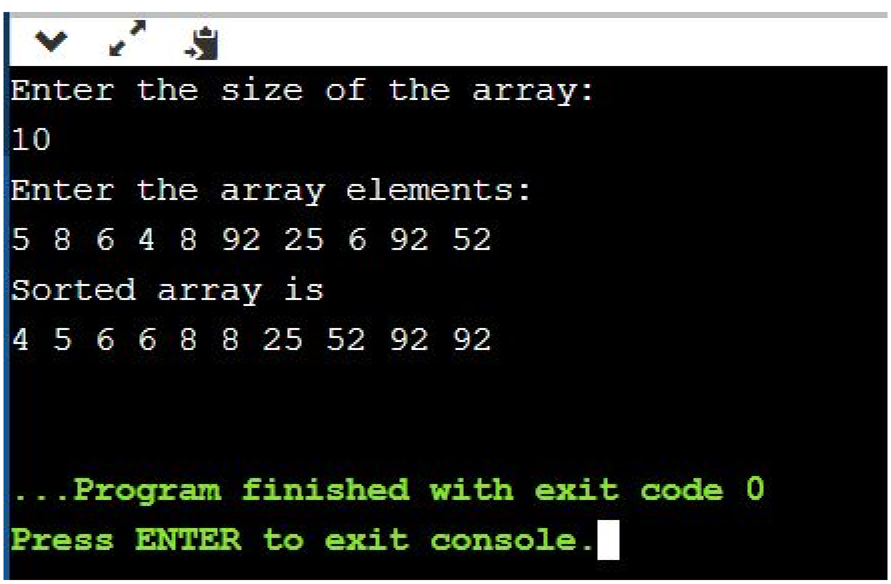
Average Time Complexity [Big-theta]: **$O(n^2)$**

Space Complexity: **O(1)**

HEAP SORT

```
#include <iostream>
using namespace std;
void heapify(int array[], int size, int i)
{
   int largest = i; // Initialize largest as root
   int left = 2*i + 1; // left = 2*i + 1
   int right = 2*i + 2; // right = 2*i + 2

   // If left child is larger than root
   if (left < size && array[left] > array[largest])
           largest = left;

   // If right child is larger than largest so far
   if (right < size && array[right] > array[largest])
           largest = right;

   // If largest is not root
   if (largest != i)
   {
           swap(array[i], array[largest]);

           // Recursively heapify the affected sub-tree
           heapify(array, size, largest);
   }
}

// main function to do heap sort
void heapSort(int array[], int size)
{
   // Build heap (rearrange array)
   for (int i = size / 2 - 1; i >= 0; i--)
           heapify(array, size, i);

   // One by one extract an element from heap
   for (int i=size-1; i>0; i--)
   {
           // Move current root to end
           swap(array[0], array[i]);

           // call max heapify on the reduced heap
           heapify(array, i, 0);
   }
}
```

```
/* function to print array */
void printArray(int array[], int size)
{
   for (int i=0; i<size; ++i)
              cout << array[i] << " ";
   cout << "\n";
}

int main()
{
    //take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size;
   cin>>size;
   //declaration of array
   int array[size];
   //input the elements of the array
   cout << "Enter the array elements: \n";
   for(int i=0;i<size;i++) cin>>array[i];

   heapSort(array, size);

   cout << "Sorted array is \n";
   printArray(array, size);
}
```



COMPLEXITY ANALYSIS OF HEAP SORT

Worst Case Time Complexity: O(n*log n)

Best Case Time Complexity: O(n*log n)

Average Time Complexity: O(n*log n)

Space Complexity : O(1)

LINEAR SEARCH

```
#include <iostream>
using namespace std;

int linearSearch(int array[], int size, int element)
{
   int i;
   for (i = 0; i < size; i++)
            if (array[i] == element)
                     return i;
   return -1;
}

int main()
{
    //take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size, element;
   cin>>size;
   //declaration of array
   int array[size];
   //input the elements of the array
   cout << "Enter the array elements: \n";
   for(int i=0;i<size;i++) cin>>array[i];
    cout<<"Enter the element to be searched in the array:"<<endl;
    cin>>element;
   // Function call
   int result = linearSearch(array, size, element);
   (result == -1)
            ? cout << "Element is not present in array"
            : cout << "Element is present at index " << result;
   return 0;
}
```

```
Enter the size of the array:
10
Enter the array elements:
5 8 6 4 8 92 25 6 92 52
Sorted array is
4 5 6 6 8 8 25 52 92 92


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

```
Enter the size of the array:
6
Enter the array elements:
65 891 91 26 91 6
Enter the element to be searched in the array:
91
Element is present at index 2

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

COMPLEXITY ANALYSIS OF LINEAR SEARCH

in best case, linear search algorithm takes O(1) operations.
in worst case, linear search algorithm takes O(n) operations.

Space complexity : O(1)

BINARY SEARCH

```
#include <bits/stdc++.h>
using namespace std;

int binarySearch(int array[], int low, int high, int element)
{
   if (high >= low) {
        int middle = low + (high - low) / 2;

        // element is present at the middle
        if (array[middle] == element)
              return middle;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (array[middle] > element)
              return binarySearch(array, low, middle - 1, element);

        // Else the element can only be present
        // in right subarray
        return binarySearch(array, middle + 1, high, element);
   }
   //not present
   return -1;
}

int main(void)
{
   //take input of size of array
   cout<<"Enter the size of the array:"<<endl;
   int size, element;
   cin>>size;
   //declaration of array
   int array[size];
   //input the elements of the array
   cout << "Enter the array elements: \n";
   for(int i=0;i<size;i++) cin>>array[i];
   cout<<"Enter the element to be searched in the array:"<<endl;
   cin>>element;
   //function call
   int result = binarySearch(array, 0, size - 1, element);
   (result == -1) ? cout << "Element is not present in array"
                  : cout << "Element is present at index " << result;
```

```
    return 0;
}
```

```
Enter the size of the array:
6
Enter the array elements:
46 9 272 4 82 03
Enter the element to be searched in the array:
3
Element is not present in array

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

```
Enter the size of the array:
6
Enter the array elements:
45 9 5 7 5 6
Enter the element to be searched in the array:
5
Element is present at index 2

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

COMPLEXITY ANALYSIS OF BINARY SEARCH

Time Complexity of Binary Search Algorithm is O($\log_2 n$).

Space Complexity of Binary Search Algorithm is O(n).

MATRIX CHAIN MULTIPLICATION

```
#include <bits/stdc++.h>
using namespace std;

// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one
    extra row and one extra column are
    allocated in m[][]. 0th row and 0th
    column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i, j] = Minimum number of scalar
    multiplications needed to compute the
    matrix A[i]A[i+1]...A[j] = A[i..j] where
    dimension of A[i] is p[i-1] x p[i] */

    // cost is zero when multiplying
    // one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L = 2; L < n; L++)
    {
        for (i = 1; i < n - L + 1; i++)
        {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k + 1][j]
                    + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
```

```
                                        m[i][j] = q;
                        }
                }
        }
        return m[1][n - 1];
}
int main()
{
        int size;
        cout<<"Enter the size of array:"<<endl;
        cin>>size;
        cout<<"Enter the elements of array:"<<endl;
        int arr[size];
        for(int i=0;i<size;i++) cin>>arr[i];

        cout << "Minimum number of multiplications is "
                << MatrixChainOrder(arr, size);
        getchar();
        return 0;
}
```

```
Enter the size of array:
4
Enter the elements of array:
1 2 3 4
Minimum number of multiplications is 18

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

COMPLEXITY ANALYSIS OF MATRIX CHAIN MULTIPLICATION

Time Complexity: $O(n^3)$
Auxiliary Space: $O(n^2)$

Strassen’s Matrix Multiplication

```
#include<iostream>
using namespace std;
//class definition
class Strassen
{
    int i,j,a[2][2],b[2][2],c[2][2],p1,p2,p3,p4,p5,p6,p7;
    public:
    //constructor
        Strassen()
        {
            p1=0,p2=0,p3=0,p4=0,p5=0,p6=0,p7=0;
            for(i=0;i<2;i++)
            {
                for(j=0;j<2;j++)
                {
                    a[i][j] = 0;
                    b[i][j] = 0;
                    c[i][j] = 0;
                }
            }
        }
        //function to take user input of the two arrays to be multiplied
        void read()
        {
            cout<<"Matrix 1 : "<<endl;
            for(i=0;i<2;i++)
            {
                for(j=0;j<2;j++)
                {
                    cin>>a[i][j];
                }
            }
            cout<<"Matrix 2 : "<<endl;
            for(i=0;i<2;i++)
            {
                for(j=0;j<2;j++)
                {
                    cin>>b[i][j];
                }
            }
        }
        //function to calculate the Resultant array
        void cal()
        {
            p1 = (a[0][0] + a[1][1])*(b[0][0] + b[1][1]);
            p2 = (a[1][0] + a[1][1])*b[0][0];
            p3 = a[0][0]*(b[0][1] - b[1][1]);
            p4 = a[1][1]*(b[1][0] - b[0][0]);
            p5 = (a[0][0] + a[0][1])*b[1][1];
```

```
            p6 = (a[1][0] - a[0][0])*(b[0][0] + b[0][1]);
            p7 = (a[0][1] - a[1][1])*(b[1][0] + b[1][1]);
            c[0][0] = p1 + p4 + - p5 + p7;
            c[0][1] = p3 + p5;
            c[1][0] = p2 + p4;
            c[1][1] = p1 + p3 - p2 + p6;
        }
        //function to print the array
        void print()
        {
            cout<<"Resultant Matrix : "<<endl;
            for(i=0;i<2;i++)
            {
                for(j=0;j<2;j++)
                {
                    cout<<c[i][j]<<"\t";
                }
                cout<<endl;
            }
        }
};
int main()
{
    //object creation of class
    Strassen s;
    cout<<"Enter the Matrix"<<endl;
    //input the matrices from user
    s.read();
    //function to calculate
    s.cal();
    //function call to print the Resultant array
    s.print();
}
```

```
Enter the Matrix
Matrix 1 :
2 3 5 4
Matrix 2 :
6 7 8 9
Resultant Matrix :
36      41
62      71


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

COMPLEXITY ANALYSIS OF STRASSEN'S MATRIX MULTIPLICATION

Analysis

$$T(n) = \begin{cases} c & if\, n = 1 \\ 7\, x\, T(\frac{n}{2}) + d\, x\, n^2 & otherwise \end{cases}$$ where *c* and *d* are constants

Using this recurrence relation, we get $T(n) = O(n^{log7})$

Hence, the complexity of Strassen's matrix multiplication algorithm is $O(n^{log7})$.

- **Worst case time complexity:** `Θ(n^2.8074)`
- **Best case time complexity:** `Θ(1)`
- **Space complexity:** `Θ(logn)`

Longest Common Subsequence

```
#include <iostream>
#include <string.h>
using namespace std;
void lcs( string X, string Y, int m, int n )
{
int L[m+1][n+1];
for (int i=0; i<=m; i++)
{
for (int j=0; j<=n; j++)
{
if (i == 0 || j == 0)
L[i][j] = 0;
else if (X[i-1] == Y[j-1])
L[i][j] = L[i-1][j-1] + 1;
else
L[i][j] = max(L[i-1][j], L[i][j-1]);
}
}
int index = L[m][n];
char lcs[index+1];
lcs[index] = '\0';
int i = m, j = n;
while (i > 0 && j > 0)
{
if (X[i-1] == Y[j-1])
{
lcs[index-1] = X[i-1];
i--; j--; index--;
}
else if (L[i-1][j] > L[i][j-1])
i--;
else
j--;
}
cout << "LCS of " << X << " and " << Y << " is " << lcs<< " of length "<<strlen(lcs);
}
int main()
{
cout<<"LONGEST COMMON SUBSEQUENCE\n";
string a,b;
cout<<"ENTER FIRST STRING\n";
cin>>a;
cout<<"ENTER SECOND STRING\n";
cin>>b;
int m=a.length();
int n = b.length();
lcs(a,b,m,n);
return 0;
}
```

```
input
LONGEST COMMON SUBSEQUENCE
ENTER FIRST STRING
shivanigupta
ENTER SECOND STRING
shivanalysisalgorithms
LCS of shivanigupta and shivanalysisalgorithms is shivanigt of length 9

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

COMPLEXITY ANALYSIS OF LONGEST COMMON SUBSEQUENCE

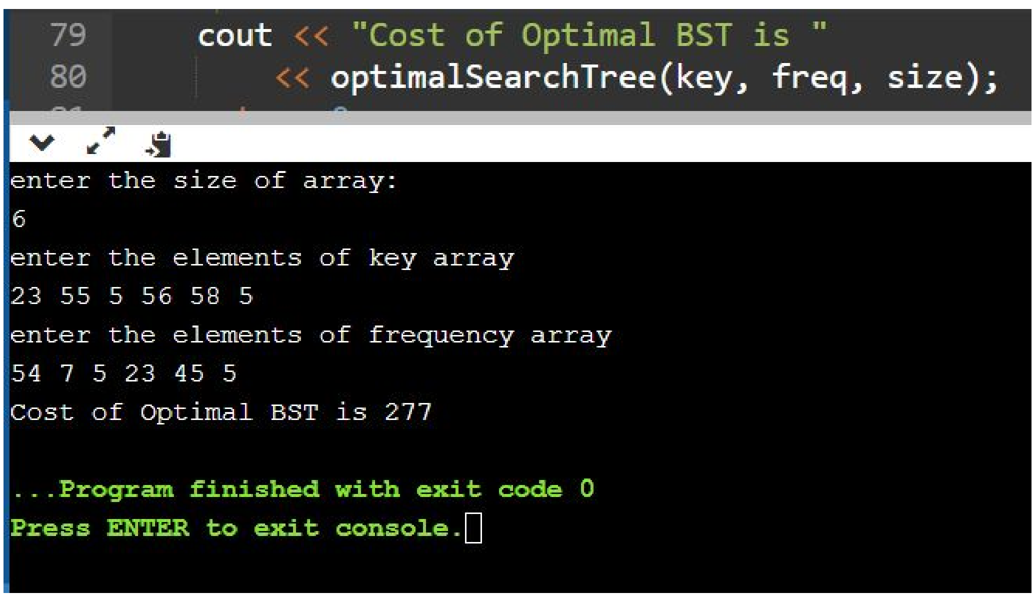
Time Complexity of the above implementation is O(mn) which is much better than the worst-case time complexity of Naive Recursive implementation.

Space Complexity-**O(mn)**

OPTIMAL BINARY SEARCH TREE

```
#include <bits/stdc++.h>
using namespace std;

int sum(int freq[], int i, int j);

int optCost(int freq[], int i, int j)
{
	// Base cases
	if (j < i) // no elements in this subarray
		return 0;
	if (j == i) // one element in this subarray
		return freq[i];

	int fsum = sum(freq, i, j);

	// Initialize minimum value
	int min = INT_MAX;

	// One by one consider all elements
	// as root and recursively find cost
	// of the BST, compare the cost with
	// min and update min if needed
	for (int r = i; r <= j; ++r)
	{
		int cost = optCost(freq, i, r - 1) +
				optCost(freq, r + 1, j);
		if (cost < min)
			min = cost;
	}

	// Return minimum value
	return min + fsum;
}

// The main function that calculates
// minimum cost of a Binary Search Tree.
// It mainly uses optCost() to find
// the optimal cost.
int optimalSearchTree(int keys[],
					int freq[], int n)
{
	// Here array keys[] is assumed to be
	// sorted in increasing order. If keys[]
	// is not sorted, then add code to sort
	// keys, and rearrange freq[] accordingly.
	return optCost(freq, 0, n - 1);
}
```

```
// A utility function to get sum of
// array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
        int s = 0;
        for (int k = i; k <= j; k++)
        s += freq[k];
        return s;
}

// Driver Code
int main()
{
    int size;
        cout<<"enter the size of array:"<<endl;
        cin>>size;
        cout<<"enter the elements of key array"<<endl;
        int key[size];
        for(int i=0;i<size;i++) cin>>key[i];
        cout<<"enter the elements of frequency array"<<endl;
        int freq[size];
        for(int i=0;i<size;i++) cin>>freq[i];

        cout << "Cost of Optimal BST is "
                << optimalSearchTree(key, freq, size);
        return 0;
}
```



The algorithm requires **O (n^3)** time, since three nested for loops are used. Each of these loops takes on at most n values.

Space Complexity:

HUFFMAN CODING

```
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct node{
char ch;
int freq;
struct node *left;
struct node *right;
}node;
node * heap[100];
int heapSize=0;
void Insert(node * element){
heapSize++;
heap[heapSize] = element;
int now = heapSize;
while(heap[now/2] -> freq > element -> freq){
heap[now] = heap[now/2];
now /= 2;
}
heap[now] = element;
}
node * DeleteMin(){
node * minElement,*lastElement;
int child,now;
minElement = heap[1];
lastElement = heap[heapSize--];
for(now = 1; now*2 <= heapSize ;now = child){
child = now*2;
if(child != heapSize && heap[child+1]->freq < heap[child] ->
freq ){
child++;
}
if(lastElement -> freq > heap[child] -> freq){
heap[now] = heap[child];
}
else{
break;
}
}
heap[now] = lastElement;
return minElement;
}
void print(node *temp,char *code){
if(temp->left==NULL && temp->right==NULL){
printf("char %c code %s\n",temp->ch,code);
```

```
return;
}
int length = strlen(code);
char leftcode[10],rightcode[10];
strcpy(leftcode,code);
strcpy(rightcode,code);
leftcode[length] = '0';
leftcode[length+1] = '\0';
rightcode[length] = '1';
rightcode[length+1] = '\0';
print(temp->left,leftcode);
print(temp->right,rightcode);
}
int main(){
heap[0] = (node *)malloc(sizeof(node));
heap[0]->freq = 0;
int n ;
printf("Enter the no of characters: ");
scanf("%d",&n);
printf("Enter the characters and their frequencies:\n");
char ch;
int freq,i;
for(i=0;i<n;i++){
scanf(" %c",&ch);
scanf("%d",&freq);
node * temp = (node *) malloc(sizeof(node));
temp -> ch = ch;
temp -> freq = freq;
temp -> left = temp -> right = NULL;Insert(temp);
}
if(n==1){
printf("char %c code 0\n",ch);
return 0;
}
for(i=0;i<n-1 ;i++){
node * left = DeleteMin();
node * right = DeleteMin();
node * temp = (node *) malloc(sizeof(node));
temp -> ch = 0;
temp -> left = left;
temp -> right = right;
temp -> freq = left->freq + right -> freq;
Insert(temp);
}
node *tree = DeleteMin();
char code[10];
```

```
code[0] = '\0';
print(tree,code);
}
```

```
Enter the no of characters: 4
Enter the characters and their frequencies:
A 4
B 5
C 2
D 7
char D code 0
char B code 10
char C code 110
char A code 111


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

COMPLEXITY ANALYSIS OF HUFFMAN CODING

Assume an encoded text string of length n and an alphabet of k symbols.
For every encoded symbol you have to traverse the tree in order to decode that symbol.

The tree contains k nodes and, *on average*, it takes O(log k) node visits to decode a symbol.

So the time complexity would be **O(n log k)**.
Space complexity is **O(k)** for the tree and **O(n)** for the decoded text.

DIJKSTRA ALGORITHM

```
#include <bits/stdc++.h>
using namespace std;
#define inf 0x3f3f3f3f
#define ll long long
#define endl "\n"
ll v, e;
vector< pair<ll, ll> >* graph = NULL;
ll* dist = NULL;
bool* visited = NULL;
ll getMinDistVertex() {
ll minIndex, minDist = inf+1;
for(ll i=0; i<v; i++) {
if(!visited[i] && dist[i] < minDist) {
minDist = dist[i];
minIndex = i;
}
}
return minIndex;
}
void Dijkstra(ll src) {
if(src >= v) {
cout<<"Invalid Input!!\n";
return ;
}
dist = new ll[v];
visited = new bool[v];
for(ll i=0; i<v; i++) {
dist[i] = inf;
visited[i] = false;
}
dist[src] = 0;
for(ll i=0; i<v-1; i++) {
ll vertex = getMinDistVertex();
visited[vertex] = true;
vector< pair<ll, ll> > neighbours = graph[vertex];
for(auto j:neighbours) {
if(!visited[j.first])
dist[j.first] = min(dist[j.first],
dist[vertex]+j.second);
}
}
cout<<"Shortest paths of each vertex from "<<src<<" are:\n";
for(ll i=0; i<v; i++) {
cout<<i<<": "<<dist[i]<<endl;
}
```

```
delete [] dist;
delete [] visited;
}
int main() {
cout<<"Enter the number of edges and vertices: ";
cin>>e>>v;
graph = new vector< pair<ll, ll> >[v];
cout<<"Enter edges input: (in the form: v1 v2 weight)\n";
for(ll i=0; i<e; i++) {
ll v1, v2, weight;
cin>>v1>>v2>>weight;
graph[v1].push_back({v2, weight});
graph[v2].push_back({v1, weight});
}
cout<<"Enter source: ";
ll src;
cin>>src;
Dijkstra(src);
delete [] graph;
return 0;
}
```

```
Enter the number of edges and vertices: 10 9
Enter edges input: (in the form: v1 v2 weight)
0 1 4
0 7 8
1 2 8
1 7 11
2 3 7
2 5 4
2 8 2
3 5 14
3 4 9
4 5 10
Enter source: 0
Shortest paths of each vertex from 0 are:
0: 0
1: 4
2: 12
3: 19
4: 26
5: 16
6: 1061109567
7: 8
8: 14


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

COMPLEXITY ANALYSIS OF DIJKSTRA ALGORITHM

Time Complexity: O(V^2), where V is the number of vertices

Space Complexity: O(V)

BELLMAN FORD ALGORITHM

```
#include <bits/stdc++.h>
using namespace std;
void BellmanFord(vector<pair<int,int>>* graph, int v){
int src;
cout<<"Enter source: ";
cin>>src;
int dist[v];
for(int i=0;i<v;i++){
dist[i] = INT_MAX;
}
dist[src] = 0;
for(int i=0;i<v;i++){
if(i == v-1){
for(int j=0;j<v;j++){
for(int k=0; k<graph[j].size(); k++){
if(dist[j] + graph[j][k].second <
dist[graph[j][k].first]){
cout<<"Negative Cycle found in the
graph"<<endl;
return;
}
}
}
}else{
for(int j=0;j<v;j++){
for(int k=0; k<graph[j].size(); k++){
if(dist[j] + graph[j][k].second <
dist[graph[j][k].first]){
dist[graph[j][k].first] = dist[j] +
graph[j][k].second;
}
}
}
}
}
cout<<"Shortest paths of each vertex from "<<src<<" are:\n";
for(int i=0;i<v;i++){
cout<<i<<": "<<dist[i]<<endl;
}
}
int main(){
int v, e;
cout<<"Enter the number of edges and vertices: ";
cin>>e>>v;
vector<pair<int,int>> graph[v];
```

```
cout<<"Enter edges input: (in the form: v1 v2 weight)\n";
int st, end, w;
for(int i=0; i<e; i++){
cin>>st>>end>>w;
graph[st].push_back(make_pair(end,w));
}
BellmanFord(graph, v);
return 0;
}
```

```
Enter the number of edges and vertices: 8 6
Enter edges input: (in the form: v1 v2 weight)
0 1 2
0 2 4
1 2 1
1 3 7
2 4 3
4 3 2
4 5 5
3 5 1
Enter source: 0
Shortest paths of each vertex from 0 are:
0: 0
1: 2
2: 3
3: 8
4: 6
5: 9


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

COMPLEXITY ANALYSIS OF BELLMAN FORD ALGORITHM

Time Complexity: O(VE), where V is the number of vertices and E is the number of edges

Space Complexity: If we assume that the graph is given, the extra space complexity is O(V) (for an array of distances).
If we assume that the graph also counts, it can be O(V^2) for an adjacency matrix and O(V+E) for an adjacency list.

STRING MATCHING ALGORITHMS
a. NAIVE APPROACH

```
#include <bits/stdc++.h>
using namespace std;
int main() {
cout<<"Enter text: ";
string s;
cin>>s;
cout<<"Enter patter to be searched: ";
string pattern;
cin>>pattern;
int n = s.length();
int m = pattern.length();
for(int i=0, j; i<=n-m; i++) {
for(j=0; j<m; j++) {
if(s[i+j] != pattern[j]) {
break;
}
}
if(j == m) {
cout<<"Pattern found at index "<<i<<endl;
return 0;
}
}
cout<<"Pattern not found!!\n";
}
```

```
Enter text: DSCJDSGYUDDDCDNBF
Enter patter to be searched: SGY
Pattern found at index 5


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

COMPLEXITY ANALYSIS OF NAIVE PATTERN MATCHING

Time Complexity:
Best case: O(n)
Worst case: O(m*(n-m+1)), where m is pattern length and n is text length.

b.KNUTH MORRIS PRATT ALGORITHM

```
#include <bits/stdc++.h>
using namespace std;
int main() {
cout<<"Enter text: ";
string s;
cin>>s;
cout<<"Enter pattern to be searched for: ";
string pattern;
cin>>pattern;
int n = s.length();
int m = pattern.length();
int lps[m] = {0};
for(int i=1, j=0; i<m; ) {
if(pattern[i] == pattern[j]) {
lps[i++] = (j++) + 1;
continue;
}
else if(!j) {
lps[i++] = 0;
continue;
}
j = lps[j-1];
}
int i, j;
for(i=0, j=0; i<n && j<m; ) {
if(s[i] == pattern[j]) {
i++, j++;
continue;
}
else if(!j) {
i++;
continue;
}
j = lps[j-1];
}
if(j == m) {
cout<<"Pattern found at "<<i-m<<endl;
}
else {
cout<<"Pattern not found!!\n";
}
}
```

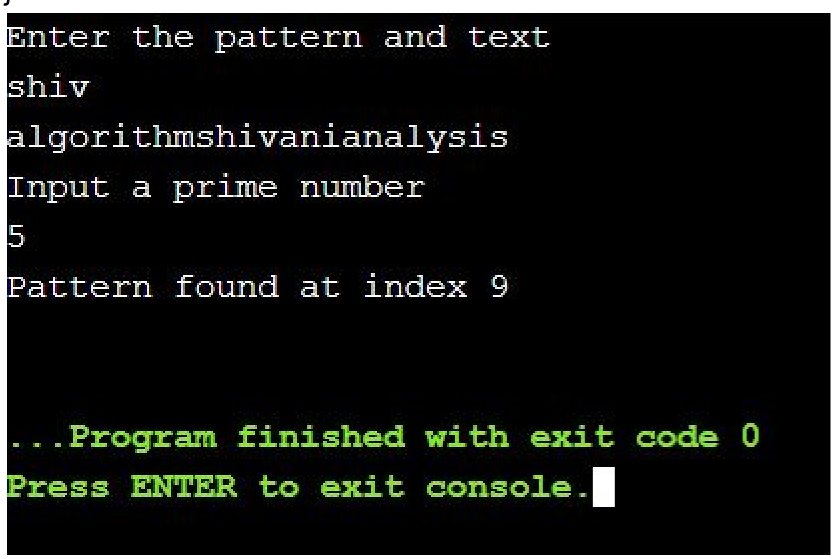
```
Enter text: shivaniguptahello
Enter pattern to be searched for: shivani
Pattern found at 0


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

Time Complexity: O(n)

RABIN KARP ALGORITHM

```
#include <bits/stdc++.h>
using namespace std;

// d is the number of characters in the input alphabet
#define d 256

/* pat -> pattern
	txt -> text
	q -> A prime number
*/
void search(string pat, string txt, int q)
{
	int M = pat.length();
	int N = txt.length();
	int i, j;
	int p = 0; // hash value for pattern
	int t = 0; // hash value for txt
	int h = 1;

	// The value of h would be "pow(d, M-1)%q"
	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;
	}

	// Slide the pattern over text one by one
	for (i = 0; i <= N - M; i++)
	{

		if ( p == t )
		{
			/* Check for characters one by one */
			for (j = 0; j < M; j++)
			{
				if (txt[i+j] != pat[j])
					break;
			}
			if (j == M)
				cout<<"Pattern found at index "<< i<<endl;
		}
		if ( i < N-M )
```

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

                        // We might get negative value of t, converting it
                        // to positive
                        if (t < 0)
                        t = (t + q);
                }
        }
}
int main()
{
        string pat, txt;
        cout<<"Enter the pattern and text"<<endl;
                cin>>pat>>txt;
        // A prime number
        int q;
        cout<<"Input a prime number"<<endl;
        cin>>q;
        // Function Call
        search(pat, txt, q);
        return 0;
}
```



```
Enter the pattern and text
shiv
algorithmshivanianalysis
Input a prime number
5
Pattern found at index 9


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

COMPLEXITY ANALYSIS OF RABINKARP ALGORITHM

TIME COMPLEXITY: The average and best-case running time of the Rabin-Karp algorithm is **O(n+m)**, but its worst-case time is **O(nm)**

SPACE COMPLEXITY: **O(|P| + |S|)**

GRAPH COLORING PROBLEM

```
#include <bits/stdc++.h>
using namespace std;
int V, E;
void print(int* color) {
cout<<"Following are the assigned colors \n";
for (int i = 0; i < V; i++) {
cout<<i<<" "<<color[i]<<endl;
}
}
bool isSafe(int** graph, int* color) {
for (int i = 0; i < V; i++) {
for (int j = i + 1; j < V; j++) {
if (graph[i][j] && color[j] == color[i]) {
return false;
}
}
}
return true;
}
bool graphColoring(int** graph, int m, int i, int* color) {
if (i == V) {
if (isSafe(graph, color)) {
print(color);
return true;
}
return false;
}
for (int j = 1; j <= m; j++) {
color[i] = j;
if (graphColoring(graph, m, i + 1, color)) {
return true;
}
color[i] = 0;
}
return false;
}
int main() {
 cout<<"Enter the number of vertices and edges: ";
 cin >> V >> E;
 int** edges = new int*[V];
 for(int i=0 ; i<V ; i++) {
 edges[i] = new int[V];
 for(int j=0 ; j<V ; j++)
 edges[i][j] = 0;
```

```
 }
 cout<<"Enter edges input: (in the form: v1 v2)\n";
 for(int i=0 ; i<E ; i++) {
 int v1, v2;
 cin>>v1>>v2;
 edges[v1][v2] = 1;
 edges[v2][v1] = 1;
 }
 cout<<"Enter the number of colours: ";
int m;
cin>>m;
int color[V];
for (int i = 0; i < V; i++) {
color[i] = 0;
}
if (!graphColoring(edges, m, 0, color)) {
cout<<"Solution does not exist\n";
}
 for(int i=0 ; i<V ; i++) {
 delete [] edges[i];
 }
 delete [] edges;
return 0;
}
```

```
Enter the number of vertices and edges: 4 5
Enter edges input: (in the form: v1 v2)
0 1
0 2
0 3
1 2
2 3
Enter the number of colours: 3
Following are the assigned colors
0 1
1 2
2 3
3 2


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

Time Complexity: O(m^V), m and V are number of colors and vertices
Space Complexity:

N QUEEN PROBLEM

```
#include<bits/stdc++.h>
using namespace std;
void placeNQueens(int** chessboard, int n, int row) {
//cout<<"call for row "<<row<<endl;
if(row >= n) {
for(int i=0 ; i<n ; i++) {
for(int j=0 ; j<n ; j++) {
cout<<chessboard[i][j]<<" ";
}
cout<<endl;
}
cout<<endl;
return;
}
for(int j=0 ; j<n ; j++) {
bool colpresent = false, leftdiagonal = false, rightdiagonal = false;
int refrow = row-1, refcol = j;
while(refrow>=0) {
if(chessboard[refrow][refcol] == 1) {
colpresent = true;
break;
}
refrow--;
}
//cout<<"col check success for ("<<row<<","<<j<<")"<<endl;
if(colpresent)
continue;
refrow = row-1, refcol = j-1;
while(refrow>=0 && refcol>=0) {
if(chessboard[refrow][refcol] == 1) {
leftdiagonal = true;
break;
}
refrow--;
refcol--;
}
//cout<<"left diagonal check success for ("<<row<<","<<j<<")"<<endl;
if(leftdiagonal)
continue;
refrow = row-1, refcol = j+1;
while(refcol<n && refrow>=0) {
if(chessboard[refrow][refcol] == 1) {
rightdiagonal = true;
break;
```

```
}
refrow--;
refcol++;
}
//cout<<"right diagonal check success for("<<row<<","<<j<<")"<<endl;
if(rightdiagonal)
continue;
chessboard[row][j] = 1;
placeNQueens(chessboard, n, row+1);
chessboard[row][j] = 0;
}
}
void placeNQueens(int n){
int** chessboard = new int*[n];
for(int i=0 ; i<n ; i++) {
chessboard[i] = new int[n];
}
for(int i=0 ; i<n ; i++) {
for(int j=0 ; j<n ; j++)
chessboard[i][j] = 0;
}
placeNQueens(chessboard, n, 0);
for(int i=0 ; i<n ; i++)
delete [] chessboard[i];
delete [] chessboard;
}
int main(){
cout<<"Enter the value of n: ";
int n;
cin >> n ;
placeNQueens(n);
return 0;
}
```

```
Enter the value of n: 4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0



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

COMPLEXITY ANALYSIS OF N QUEEN PROBLEM

the best, average and worst case complexity remains O(N!)
Space complexity: O(N)

0/1 Knapsack Problem

```
#include <bits/stdc++.h>
using namespace std;
int knapsack(int wt[], int val[], int n,int W){
 int dp[W+1];
 memset(dp, 0, sizeof(dp));
 for(int i=0; i < n; i++)
 for(int j=W; j>=wt[i]; j--)
 dp[j] = max(dp[j] , val[i] + dp[j-wt[i]]);
 return dp[W];
}
int main(){
 int n;
 cout<<"Enter number of items: ";
 cin >> n;
 int* weights = new int[n];
 int* values = new int[n];
 cout<<"Enter weights of items seperated by space:\n";
 for(int i = 0; i < n; i++){
 cin >> weights[i];
 }
 cout<<"Enter values of items seperated by space:\n";
 for(int i = 0; i < n; i++){
 cin >> values[i];
 }
 int maxWeight;
 cout<<"Enter maximum weight of knapsack: ";
 cin >> maxWeight;
 cout << "Maximum value is " << knapsack(weights, values, n,
maxWeight) << endl;
}
```

```
Enter number of items: 6
Enter weights of items seperated by space:
5 2 55 8 22 5
Enter values of items seperated by space:
3 55 8 1 57 5
Enter maximum weight of knapsack: 1000
Maximum value is 129


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

COMPLEXITY ANALYSIS OF 0/1 KNAPSACK PROBLEM

This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.

- Each entry of the table requires constant time θ(1) for its computation.
- It takes θ(nw) time to fill (n+1)(w+1) table entries.
- It takes θ(n) time for tracing the solution since tracing process traces the n rows.
- Thus, overall θ(nw) time is taken to solve 0/1 knapsack problem using dynamic programming.

Space Complexity - O(w)

FLOYD WARSHALL ALGORITHM

```
#include <bits/stdc++.h>
#define int_m 1000000
using namespace std;
void Warshall(vector<pair<int,int>>* graph, int v){
int dist[v][v];
for(int i=0;i<v;i++){
for(int j=0;j<v;j++){
if(i == j)
dist[i][j] = 0;
else
dist[i][j] = int_m;
}
}
for(int i=0; i<v; i++){
for(auto j:graph[i]){
dist[i][j.first] = j.second;
}
}
for(int i=0; i<v; i++){
for(int j=0; j<v; j++){
for(int k=0; k<v; k++){
if(dist[j][k] > dist[j][i] + dist[i][k]){
dist[j][k] = dist[j][i] + dist[i][k];
}
}
}
}
cout<<"Distance Matrix: ";
for(int i=0; i<v; i++){
cout<<endl;
for(int j=0; j<v; j++){
if(dist[i][j] == int_m)
cout<<"I ";
else
cout<<dist[i][j]<<" ";
}
}
cout<<endl;
}
int main(){
int v, e;
cout<<"Enter the number of edges and vertices: ";
cin>>e>>v;
vector<pair<int,int>> graph[v];
cout<<"Enter edges input: (in the form: v1 v2 weight)\n";
```

```
int st, end, w;
for(int i=0; i<e; i++){
cin>>st>>end>>w;
graph[st].push_back(make_pair(end,w));
}
Warshall(graph, v);
return 0;
}
```

```
Enter the number of edges and vertices: 8 6
Enter edges input: (in the form: v1 v2 weight)
0 1 2
0 2 4
1 2 1
1 3 7
2 3 4
4 3 2
4 5 5
3 5 1
Distance Matrix:
0 2 3 7 I 8
I 0 1 5 I 6
I I 0 4 I 5
I I I 0 I 1
I I I 2 0 3
I I I I I 0


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

COMPLEXITY ANALYSIS OF FLOYD WARSHALL ALGORITHM

Time Complexity: O(V^3)
Space Complexity: $O(|V|^2)$

BREADTH FIRST TRAVERSAL OF GRAPH

```
#include <bits/stdc++.h>
using namespace std;
void print(int** edges, int n, bool* visited) {
 queue<int> pendingnodes;
 pendingnodes.push(0);
 visited[0] = 1;
 while(!pendingnodes.empty()) {
 int front = pendingnodes.front();
 pendingnodes.pop();
 cout<<front<<" ";
 for(int i=0 ; i<n ; i++) {
 if(i == front)continue;
 if(visited[i] == 0 && edges[front][i] == 1) {
 pendingnodes.push(i);
 visited[i] = 1;
 }
 }
 }
}
int main() {
 int V, E;
 cout<<"Enter the number of vertices and edges: ";
 cin >> V >> E;
 int** edges = new int*[V];
 for(int i=0 ; i<V ; i++) {
 edges[i] = new int[V];
 for(int j=0 ; j<V ; j++)
 edges[i][j] = 0;
 }
 cout<<"Enter edges input: (in the form: v1 v2)\n";
 for(int i=0 ; i<E ; i++) {
 int v1, v2;
 cin>>v1>>v2;
 edges[v1][v2] = 1;
 edges[v2][v1] = 1;
 }
 bool* visited = new bool[V];
 for(int i=0 ; i<V ; i++) {
 visited[i] = false;
 }
 cout<<"BFS traversal:\n";
 print(edges, V, visited);
 cout<<endl;
 delete [] visited;
 for(int i=0 ; i<V ; i++) {
```

```
delete [] edges[i];
}
delete [] edges;
return 0;
}
```

```
Enter the number of vertices and edges: 6 8
Enter edges input: (in the form: v1 v2)
1 2
0 2
3 4
2 5
5 1
5 4
3 2
1 5
BFS traversal:
0 2 1 3 5 4


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

COMPLEXITY ANALYSIS OF BREADTH FIRST SEARCH

Time complexity is as follows:

```
V * (O(1) + O(Eaj) + O(1))
V + V * Eaj + V
2V + E(total number of edges in graph)
V + E
```

Space Complexity: O(|V|) = O(b^d)

DEPTH FIRST TRAVERSAL OF GRAPH

```
#include <bits/stdc++.h>
using namespace std;
void dfs(int** edges, int n, bool* visited, int cur) {
 cout<<cur<<" ";
 visited[cur] = true;
 for(int i=0; i<n; i++) {
 if(!visited[i] && edges[cur][i]) {
 dfs(edges, n, visited, i);
 }
 }
}
void print(int** edges, int n, bool* visited) {
 dfs(edges, n, visited, 0);
}
int main() {
 int V, E;
 cout<<"Enter the number of vertices and edges: ";
 cin >> V >> E;
 int** edges = new int*[V];
 for(int i=0 ; i<V ; i++) {
 edges[i] = new int[V];
 for(int j=0 ; j<V ; j++)
 edges[i][j] = 0;
 }
 cout<<"Enter edges input: (in the form: v1 v2)\n";
 for(int i=0 ; i<E ; i++) {
 int v1, v2;
 cin>>v1>>v2;
 edges[v1][v2] = 1;
 edges[v2][v1] = 1;
 }
 bool* visited = new bool[V];
 for(int i=0 ; i<V ; i++) {
 visited[i] = false;
 }
 cout<<"DFS traversal:\n";
 print(edges, V, visited);
 cout<<endl;
 delete [] visited;
 for(int i=0 ; i<V ; i++) {
 delete [] edges[i];
 }
 delete [] edges;
 return 0;
}
```

```
Enter the number of vertices and edges: 6 8
Enter edges input: (in the form: v1 v2)
1 2
0 2
3 4
2 5
5 1
5 4
3 2
1 5
DFS traversal:
0 2 1 5 4 3


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

COMPLEXITY ANALYSIS OF DEPTH FIRST TRAVERSAL

- **Time complexity:** O(V + E), where V is the number of vertices and E is the number of edges in the graph.
- **Space Complexity:** O(V).
  Since, an extra visited array is needed of size V.

PRIM'S ALGORITHM

```
#include<bits/stdc++.h>
using namespace std ;
int V,E;
int graph[1000][1000];
int parent[1000], weight[1000], vis[1000];
int findmin(){
int minVertex = -1 ;
for(int i = 0 ; i <V;i++){
if(!vis[i] && ( minVertex == -1 || weight[minVertex] > weight[i]) ){
minVertex = i;
}
}
return minVertex;
}
void prim(int X)
{
for(int i = 0 ; i <V;i++){
vis[i] = false;
weight[i] = INT_MAX;
}
parent[0] = -1;
weight[0] = 0;
for(int i = 0 ; i <V;i++){
int minVertex = findmin();
vis[minVertex] = true;
for(int j = 0 ; j<V;j++){
if(graph[minVertex][j] != 0 && !vis[j]){
if(graph[minVertex][j] < weight[j] ) {
weight[j] = graph[minVertex][j];
parent[j] = minVertex ;
}
}
}
}
cout<<"PRIM'S MST\n";
for(int i = 1;i<V;i++){
if(parent[i] < i){
cout<<parent[i] <<" " << i <<" " <<weight[i]<<endl;
59
}
else{
cout<<i <<" " << parent[i] <<" " <<weight[i]<<endl;
}
}
}
```

```
int main(){
cout<<"ENTER NO OF VERTICES\n";
cin>>V;
cout<<"ENTER NO OF EDGES\n";
cin>>E;
int e = 1;
for(int i = 0 ; i <V;i++){
for(int j = 0 ; j <V;j++){
graph[i][j] = 0;
}
}
while(E--){
cout<<"ENTER SRC,DEST and WEIGHT FOR EDGE"<<e++<<"\n";
int x,y,wt;
cin>>x>>y>>wt;
graph[x][y] = wt;
graph[y][x] = wt;
}
prim(0);
}
```

```
ENTER NO OF VERTICES
5
ENTER NO OF EDGES
7
ENTER SRC,DEST and WEIGHT FOR EDGE1
0 1 4
ENTER SRC,DEST and WEIGHT FOR EDGE2
0 2 8
ENTER SRC,DEST and WEIGHT FOR EDGE3
1 3 6
ENTER SRC,DEST and WEIGHT FOR EDGE4
1 2 2
ENTER SRC,DEST and WEIGHT FOR EDGE5
2 3 3
ENTER SRC,DEST and WEIGHT FOR EDGE6
2 4 9
ENTER SRC,DEST and WEIGHT FOR EDGE7
3 4 5
PRIM'S MST
0 1 4
1 2 2
2 3 3
3 4 5


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

```
}
```

COMPLEXITY ANALYSIS OF PRIM'S ALGORITHM

Time Complexity of the above program is **O(V^2).** If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to **O(E log V)** with the help of binary heap.

Space Complexity: O(V)

KRUSKAL’S ALGORITHM

```
#include <bits/stdc++.h>
using namespace std;

class Edge
{
public:
   int v = 0;
   int w = 0;

   Edge(int v, int w)
   {
      this->v = v;
      this->w = w;
   }
};

class pair_
{
public:
   int src;
   int par;
   int w;

   pair_(int src, int par, int w)
   {
      this->src = src;
      this->par = par;
      this->w = w;
   }
};

void display(vector<vector<Edge>> &gp)
{

   for (int i = 0; i < gp.size(); i++)
   {
      cout << i << " -> ";
      for (Edge e : gp[i])
      {
         cout << "(" << e.v << ", " << e.w << "), ";
      }
      cout << endl;
   }

   cout << endl;
```

```
}

void addEdge(vector<vector<Edge>> &gp, int u, int v, int w)
{
    gp[u].push_back(Edge(v, w));
    gp[v].push_back(Edge(u, w));
}

vector<int> par;
vector<int> setSize;

int findPar(int vtx)
{
    if (par[vtx] == vtx)
        return vtx;
    return par[vtx] = findPar(par[vtx]);
}

void mergeSet(int p1, int p2)
{
    if (setSize[p1] < setSize[p2])
    {
        par[p1] = p2;
        setSize[p2] += setSize[p1];
    }
    else
    {
        par[p2] = p1;
        setSize[p1] += setSize[p2];
    }
}

void kruskalAlgo(vector<vector<int>> &arr)
{
    vector<vector<Edge>> KruskalGraph(arr.size());
    sort(arr.begin(), arr.end(), [](vector<int> &a, vector<int> &b) {
        return a[2] < b[2];
    });

    for (vector<int> &ar : arr)
    {
        int u = ar[0];
        int v = ar[1];
        int p1 = findPar(u);
        int p2 = findPar(v);
        if (p1 != p2)
```

```
        {
            mergeSet(p1, p2);
            addEdge(KruskalGraph, u, v, ar[2]);
        }
    }

    display(KruskalGraph);
}
int main()
{
 int n;
 cout<<"Enter the number of vertices"<<endl;
 cin>>n;
 cout<<"Enter the number of edges"<<endl;
 int e;
 cin>>e;

 vector<vector<int>> arr;

 vector<int> temp(3,0);

 for(int i=0;i<e;i++)
 {
    int a;
    int b;
    int c;
    cout<<"Enter "<<i<<" edge :"<<endl;
    cout<<"Enter the source"<<endl;
    cin>>a;
    cout<<"Enter the destination"<<endl;
    cin>>b;
    cout<<"Enter the weight"<<endl;
    cin>>c;
    temp[0]=a;temp[1]=b;temp[2]=c;
    arr.push_back(temp);
 }

 for(int i=0;i<n;i++)
 {
  par.push_back(i);
  setSize.push_back(1);
 }

 kruskalAlgo(arr);
}
```

```
ENTER THE NO OF VERTICES
6
ENTER THE NO OF EDGES
9
ENTER SRC,DEST and WEIGHT FOR EDGE1
0 1 2
ENTER SRC,DEST and WEIGHT FOR EDGE2
0 3 1
ENTER SRC,DEST and WEIGHT FOR EDGE3
0 4 4
ENTER SRC,DEST and WEIGHT FOR EDGE4
1 2 3
ENTER SRC,DEST and WEIGHT FOR EDGE5
1 3 3
ENTER SRC,DEST and WEIGHT FOR EDGE6
1 5 7
ENTER SRC,DEST and WEIGHT FOR EDGE7
2 3 5
ENTER SRC,DEST and WEIGHT FOR EDGE8
2 5 8
ENTER SRC,DEST and WEIGHT FOR EDGE9
3 4 9
ORIGINAL GRAPH:
EDGE FROM 0 TO 1 OF WEIGHT 2
EDGE FROM 0 TO 3 OF WEIGHT 1
EDGE FROM 0 TO 4 OF WEIGHT 4
EDGE FROM 1 TO 2 OF WEIGHT 3
EDGE FROM 1 TO 3 OF WEIGHT 3
EDGE FROM 1 TO 5 OF WEIGHT 7
EDGE FROM 2 TO 3 OF WEIGHT 5
EDGE FROM 2 TO 5 OF WEIGHT 8
EDGE FROM 3 TO 4 OF WEIGHT 9
```

```
KRUSKAL'S MST
EDGE FROM 0 TO 3 OF WEIGHT 1
EDGE FROM 0 TO 1 OF WEIGHT 2
EDGE FROM 1 TO 2 OF WEIGHT 3
EDGE FROM 0 TO 4 OF WEIGHT 4
EDGE FROM 1 TO 5 OF WEIGHT 7


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

COMPLEXITY ANALYSIS OF KRUSKAL'S ALGORITHM

Time Complexity: O(ElogV)

Space Complexity: O(|V|+|E|)