# AOA Assignment

**Submitted to:**

* Sir Atif Warraich

**Submitted by:**

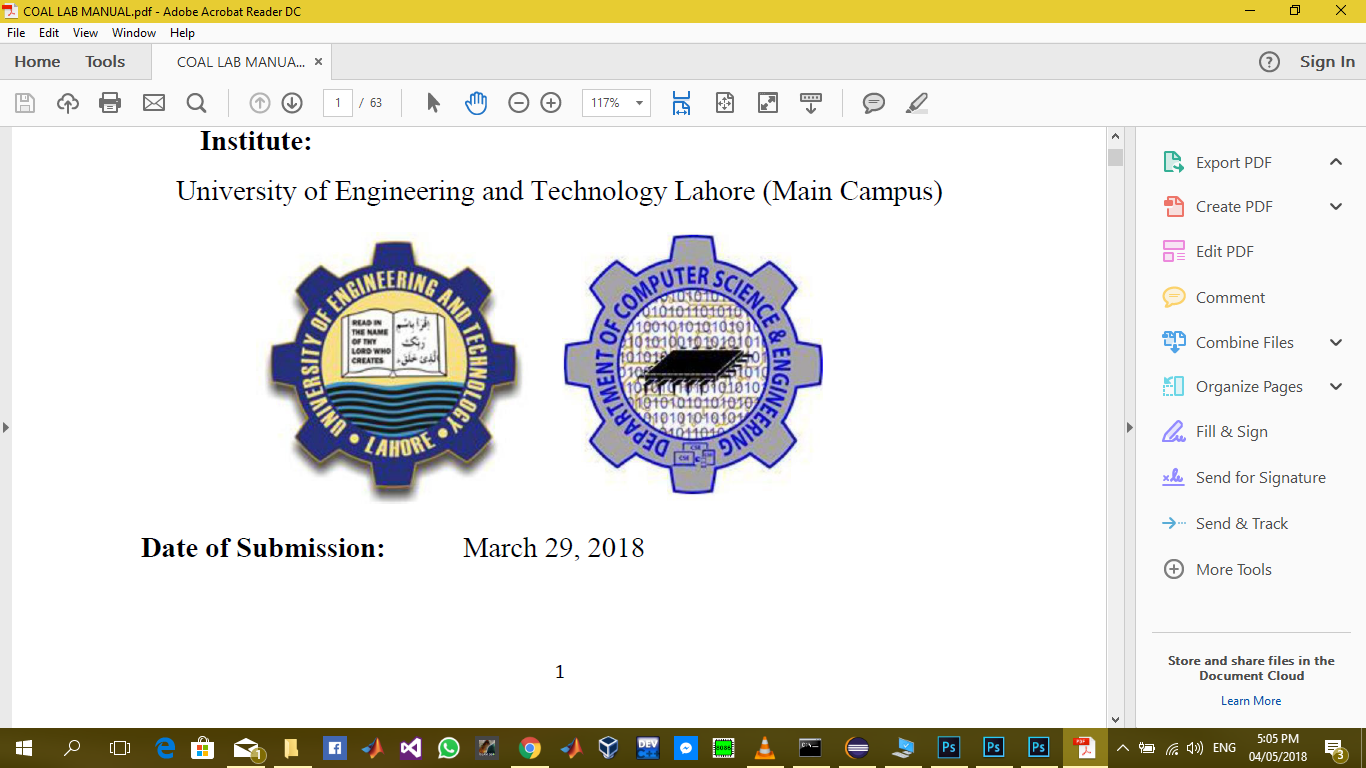
* 2016-CS-178

**Department of Computer Science and Engineering**

**UET Lahore**

**Date:** 08-Oct-2018





# 

Contents

[AOA Assignment 1](#_Toc527321055)

[Algorithms 4](#_Toc527321056)

[1. Selection Sort 4](#_Toc527321057)

[2. BubbleSort 4](#_Toc527321058)

[3. Recursive BubbleSort 5](#_Toc527321059)

[4. Insertion Sort 6](#_Toc527321060)

[5. Recursive Insertion Sort 6](#_Toc527321061)

[6. Merge Sort 7](#_Toc527321062)

[7. Iterative Merge Sort 9](#_Toc527321063)

[8. Quick Sort 12](#_Toc527321064)

[9. Iterative QuickSort 13](#_Toc527321065)

[10. Heap Sort 15](#_Toc527321066)

[11. Breadth First Search Traversal 16](#_Toc527321067)

[12. Depth First Search Traversal 18](#_Toc527321068)

[13. Dijkstra’s Shortest Path Algorithm 20](#_Toc527321069)

[14. Constant time — O(1). 22](#_Toc527321070)

[15. Logarithmic time — O(log n) 22](#_Toc527321071)

[16. Linear time — O(n) 23](#_Toc527321072)

[17. Quadratic time — O(n^2) 23](#_Toc527321073)

[18. Linear time — O(n + m) 23](#_Toc527321074)

[19. Slow Solution \_\_Time complexity O(n2) 24](#_Toc527321075)

[20. Fast solution — time complexity O(n) 24](#_Toc527321076)

[21. Model solution — time complexity O(1) 24](#_Toc527321077)

[22. Fibonacci algorithm 25](#_Toc527321078)

[23. Prime Number Check 25](#_Toc527321079)

[24. Linear Search Algo 25](#_Toc527321080)

[25. General Algorithm 26](#_Toc527321081)

[26. Alternate Sort of Array 26](#_Toc527321082)

[27. General Algorithm 27](#_Toc527321083)

[28. General Algorithm 28](#_Toc527321084)

[29. General Algorithm 29](#_Toc527321085)

[30. General Algorithm 30](#_Toc527321086)

[31. General Algorithm 31](#_Toc527321087)

[32. General Algorithm 32](#_Toc527321088)

[33. General Algorithm 33](#_Toc527321089)

[34. General Algorithm 33](#_Toc527321090)

[35. General Algorithm 35](#_Toc527321091)

[36. General Algorithm 36](#_Toc527321092)

[37. General Algorithm 37](#_Toc527321093)

[38. General Algorithm 38](#_Toc527321094)

[39. General Algorithm 38](#_Toc527321095)

[40. General Algorithm 38](#_Toc527321096)

[41. General Algorithm 39](#_Toc527321097)

[42. General Algorithm 39](#_Toc527321098)

[43. General Algorithm 40](#_Toc527321099)

[44. General Algorithm 41](#_Toc527321100)

[45. General Algorithm 42](#_Toc527321101)

[46. General Algorithm 42](#_Toc527321102)

[47. General Algorithm 42](#_Toc527321103)

[48. General Algorithm 43](#_Toc527321104)

[49. General Algorithm 44](#_Toc527321105)

[50. General Algorithm 45](#_Toc527321106)

[51. General Algorithm 45](#_Toc527321107)

[52. General Algorithm 46](#_Toc527321108)

[53. General Algorithm 47](#_Toc527321109)

# Algorithms

## Selection Sort

// C program for implementation of selection sort

#include <stdio.h>

void swap(int \*xp, int \*yp)

{

    int temp = \*xp;

    \*xp = \*yp;

    \*yp = temp;

}

void selectionSort(int arr[], int n)

{

    int i, j, min\_idx;

    // One by one move boundary of unsorted subarray

    for (i = 0; i < n-1; i++)

    {

        // Find the minimum element in unsorted array

        min\_idx = i;

        for (j = i+1; j < n; j++)

          if (arr[j] < arr[min\_idx])

            min\_idx = j;

        // Swap the found minimum element with the first element

        swap(&arr[min\_idx], &arr[i]);

    }

}

## BubbleSort

// C program for implementation of Bubble sort

#include <stdio.h>

void swap(int \*xp, int \*yp)

{

    int temp = \*xp;

    \*xp = \*yp;

    \*yp = temp;

}

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

   int i, j;

   for (i = 0; i < n-1; i++)

       // Last i elements are already in place

       for (j = 0; j < n-i-1; j++)

           if (arr[j] > arr[j+1])

              swap(&arr[j], &arr[j+1]);

}

## Recursive BubbleSort

#include <bits/stdc++.h>

using namespace std;

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

    // Base case

    if (n == 1)

        return;

    // One pass of bubble sort. After

    // this pass, the largest element

    // is moved (or bubbled) to end.

    for (int i=0; i<n-1; i++)

        if (arr[i] > arr[i+1])

            swap(arr[i], arr[i+1]);

    // Largest element is fixed,

    // recur for remaining array

    bubbleSort(arr, n-1);

}

## Insertion Sort

// C program for insertion sort

#include <stdio.h>

#include <math.h>

/\* Function to sort an array using insertion sort\*/

void insertionSort(int arr[], int n)

{

   int i, key, j;

   for (i = 1; i < n; i++)

   {

       key = arr[i];

       j = i-1;

       /\* Move elements of arr[0..i-1], that are

          greater than key, to one position ahead

          of their current position \*/

       while (j >= 0 && arr[j] > key)

       {

           arr[j+1] = arr[j];

           j = j-1;

       }

       arr[j+1] = key;

   }

}

## Recursive Insertion Sort

// Recursive C++ program for insertion sort

#include <iostream>

using namespace std;

// Recursive function to sort an array using

// insertion sort

void insertionSortRecursive(int arr[], int n)

{

    // Base case

    if (n <= 1)

        return;

    // Sort first n-1 elements

    insertionSortRecursive( arr, n-1 );

    // Insert last element at its correct position

    // in sorted array.

    int last = arr[n-1];

    int j = n-2;

    /\* Move elements of arr[0..i-1], that are

      greater than key, to one position ahead

      of their current position \*/

    while (j >= 0 && arr[j] > last)

    {

        arr[j+1] = arr[j];

        j--;

    }

    arr[j+1] = last;

}

## Merge Sort

/\* C program for Merge Sort \*/

#include<stdlib.h>

#include<stdio.h>

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

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

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 =  r - m;

    /\* create temp arrays \*/

    int L[n1], R[n2];

    /\* Copy data to temp arrays L[] and R[] \*/

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1+ j];

    /\* Merge the temp arrays back into arr[l..r]\*/

    i = 0; // Initial index of first subarray

    j = 0; // Initial index of second subarray

    k = l; // Initial index of merged subarray

    while (i < n1 && j < n2)

    {

        if (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

        else

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    /\* Copy the remaining elements of L[], if there

       are any \*/

    while (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    /\* Copy the remaining elements of R[], if there

       are any \*/

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

/\* l is for left index and r is right index of the

   sub-array of arr to be sorted \*/

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

{

    if (l < r)

    {

        // Same as (l+r)/2, but avoids overflow for

        // large l and h

        int m = l+(r-l)/2;

        // Sort first and second halves

        mergeSort(arr, l, m);

        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);

    }

}

## Iterative Merge Sort

/\* Iterative C program for merge sort \*/

#include<stdlib.h>

#include<stdio.h>

/\* Function to merge the two haves arr[l..m] and arr[m+1..r] of array arr[] \*/

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

// Utility function to find minimum of two integers

int min(int x, int y) { return (x<y)? x :y; }

/\* Iterative mergesort function to sort arr[0...n-1] \*/

void mergeSort(int arr[], int n)

{

   int curr\_size;  // For current size of subarrays to be merged

                   // curr\_size varies from 1 to n/2

   int left\_start; // For picking starting index of left subarray

                   // to be merged

   // Merge subarrays in bottom up manner.  First merge subarrays of

   // size 1 to create sorted subarrays of size 2, then merge subarrays

   // of size 2 to create sorted subarrays of size 4, and so on.

   for (curr\_size=1; curr\_size<=n-1; curr\_size = 2\*curr\_size)

   {

       // Pick starting point of different subarrays of current size

       for (left\_start=0; left\_start<n-1; left\_start += 2\*curr\_size)

       {

           // Find ending point of left subarray. mid+1 is starting

           // point of right

           int mid = left\_start + curr\_size - 1;

           int right\_end = min(left\_start + 2\*curr\_size - 1, n-1);

           // Merge Subarrays arr[left\_start...mid] & arr[mid+1...right\_end]

           merge(arr, left\_start, mid, right\_end);

       }

   }

}

/\* Function to merge the two haves arr[l..m] and arr[m+1..r] of array arr[] \*/

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

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 =  r - m;

    /\* create temp arrays \*/

    int L[n1], R[n2];

    /\* Copy data to temp arrays L[] and R[] \*/

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1+ j];

    /\* Merge the temp arrays back into arr[l..r]\*/

    i = 0;

    j = 0;

    k = l;

    while (i < n1 && j < n2)

    {

        if (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

        else

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    /\* Copy the remaining elements of L[], if there are any \*/

    while (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    /\* Copy the remaining elements of R[], if there are any \*/

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

## Quick Sort

/\* C implementation QuickSort \*/

#include<stdio.h>

// A utility function to swap two elements

void swap(int\* a, int\* b)

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

/\* This function takes last element as pivot, places

   the pivot element at its correct position in sorted

    array, and places all smaller (smaller than pivot)

   to left of pivot and all greater elements to right

   of pivot \*/

int partition (int arr[], int low, int high)

{

    int pivot = arr[high];    // pivot

    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)

    {

        // If current element is smaller than or

        // equal to pivot

        if (arr[j] <= pivot)

        {

            i++;    // increment index of smaller element

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

/\* The main function that implements QuickSort

 arr[] --> Array to be sorted,

  low  --> Starting index,

  high  --> Ending index \*/

void quickSort(int arr[], int low, int high)

{

    if (low < high)

    {

        /\* pi is partitioning index, arr[p] is now

           at right place \*/

        int pi = partition(arr, low, high);

        // Separately sort elements before

        // partition and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

## Iterative QuickSort

// An iterative implementation of quick sort

#include <stdio.h>

// A utility function to swap two elements

void swap ( int\* a, int\* b )

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

/\* This function is same in both iterative and recursive\*/

int partition (int arr[], int l, int h)

{

    int x = arr[h];

    int i = (l - 1);

    for (int j = l; j <= h- 1; j++)

    {

        if (arr[j] <= x)

        {

            i++;

            swap (&arr[i], &arr[j]);

        }

    }

    swap (&arr[i + 1], &arr[h]);

    return (i + 1);

}

/\* A[] --> Array to be sorted,

   l  --> Starting index,

   h  --> Ending index \*/

void quickSortIterative (int arr[], int l, int h)

{

    // Create an auxiliary stack

    int stack[ h - l + 1 ];

    // initialize top of stack

    int top = -1;

    // push initial values of l and h to stack

    stack[ ++top ] = l;

    stack[ ++top ] = h;

    // Keep popping from stack while is not empty

    while ( top >= 0 )

    {

        // Pop h and l

        h = stack[ top-- ];

        l = stack[ top-- ];

        // Set pivot element at its correct position

        // in sorted array

        int p = partition( arr, l, h );

        // If there are elements on left side of pivot,

        // then push left side to stack

        if ( p-1 > l )

        {

            stack[ ++top ] = l;

            stack[ ++top ] = p - 1;

        }

        // If there are elements on right side of pivot,

        // then push right side to stack

        if ( p+1 < h )

        {

            stack[ ++top ] = p + 1;

            stack[ ++top ] = h;

        }

    }

}

## Heap Sort

// C++ program for implementation of Heap Sort

#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

void heapify(int arr[], int n, int i)

{

    int largest = i; // Initialize largest as root

    int l = 2\*i + 1; // left = 2\*i + 1

    int r = 2\*i + 2; // right = 2\*i + 2

    // If left child is larger than root

    if (l < n && arr[l] > arr[largest])

        largest = l;

    // If right child is larger than largest so far

    if (r < n && arr[r] > arr[largest])

        largest = r;

    // If largest is not root

    if (largest != i)

    {

        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree

        heapify(arr, n, largest);

    }

}

// main function to do heap sort

void heapSort(int arr[], int n)

{

    // Build heap (rearrange array)

    for (int i = n / 2 - 1; i >= 0; i--)

        heapify(arr, n, i);

    // One by one extract an element from heap

    for (int i=n-1; i>=0; i--)

    {

        // Move current root to end

        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap

        heapify(arr, i, 0);

    }

}

## Breadth First Search Traversal

// Program to print BFS traversal from a given

// source vertex. BFS(int s) traverses vertices

// reachable from s.

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph

{

    int V;    // No. of vertices

    // Pointer to an array containing adjacency

    // lists

    list<int> \*adj;

public:

    Graph(int V);  // Constructor

    // function to add an edge to graph

    void addEdge(int v, int w);

    // prints BFS traversal from a given source s

    void BFS(int s);

};

Graph::Graph(int V)

{

    this->V = V;

    adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

    adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::BFS(int s)

{

    // Mark all the vertices as not visited

    bool \*visited = new bool[V];

    for(int i = 0; i < V; i++)

        visited[i] = false;

    // Create a queue for BFS

    list<int> queue;

    // Mark the current node as visited and enqueue it

    visited[s] = true;

    queue.push\_back(s);

    // 'i' will be used to get all adjacent

    // vertices of a vertex

    list<int>::iterator i;

    while(!queue.empty())

    {

        // Dequeue a vertex from queue and print it

        s = queue.front();

        cout << s << " ";

        queue.pop\_front();

        // Get all adjacent vertices of the dequeued

        // vertex s. If a adjacent has not been visited,

        // then mark it visited and enqueue it

        for (i = adj[s].begin(); i != adj[s].end(); ++i)

        {

            if (!visited[\*i])

            {

                visited[\*i] = true;

                queue.push\_back(\*i);

            }

        }

    }

}

## Depth First Search Traversal

// C++ program to print DFS traversal from

// a given vertex in a  given graph

#include<iostream>

#include<list>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph

{

    int V;    // No. of vertices

    // Pointer to an array containing

    // adjacency lists

    list<int> \*adj;

    // A recursive function used by DFS

    void DFSUtil(int v, bool visited[]);

public:

    Graph(int V);   // Constructor

    // function to add an edge to graph

    void addEdge(int v, int w);

    // DFS traversal of the vertices

    // reachable from v

    void DFS(int v);

};

Graph::Graph(int V)

{

    this->V = V;

    adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

    adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::DFSUtil(int v, bool visited[])

{

    // Mark the current node as visited and

    // print it

    visited[v] = true;

    cout << v << " ";

    // Recur for all the vertices adjacent

    // to this vertex

    list<int>::iterator i;

    for (i = adj[v].begin(); i != adj[v].end(); ++i)

        if (!visited[\*i])

            DFSUtil(\*i, visited);

}

// DFS traversal of the vertices reachable from v.

// It uses recursive DFSUtil()

void Graph::DFS(int v)

{

    // Mark all the vertices as not visited

    bool \*visited = new bool[V];

    for (int i = 0; i < V; i++)

        visited[i] = false;

    // Call the recursive helper function

    // to print DFS traversal

    DFSUtil(v, visited);

}

## Dijkstra’s Shortest Path Algorithm

// A C++ program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

#include <stdio.h>

#include <limits.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum distance value, from

// the set of vertices not yet included in shortest path tree

int minDistance(int dist[], bool sptSet[])

{

   // Initialize min value

   int min = INT\_MAX, min\_index;

   for (int v = 0; v < V; v++)

     if (sptSet[v] == false && dist[v] <= min)

         min = dist[v], min\_index = v;

   return min\_index;

}

// A utility function to print the constructed distance array

int printSolution(int dist[], int n)

{

   printf("Vertex   Distance from Source\n");

   for (int i = 0; i < V; i++)

      printf("%d tt %d\n", i, dist[i]);

}

// Function that implements Dijkstra's single source shortest path algorithm

// for a graph represented using adjacency matrix representation

void dijkstra(int graph[V][V], int src)

{

     int dist[V];     // The output array.  dist[i] will hold the shortest

                      // distance from src to i

     bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest

                     // path tree or shortest distance from src to i is finalized

     // Initialize all distances as INFINITE and stpSet[] as false

     for (int i = 0; i < V; i++)

        dist[i] = INT\_MAX, sptSet[i] = false;

     // Distance of source vertex from itself is always 0

     dist[src] = 0;

     // Find shortest path for all vertices

     for (int count = 0; count < V-1; count++)

     {

       // Pick the minimum distance vertex from the set of vertices not

       // yet processed. u is always equal to src in the first iteration.

       int u = minDistance(dist, sptSet);

       // Mark the picked vertex as processed

       sptSet[u] = true;

       // Update dist value of the adjacent vertices of the picked vertex.

       for (int v = 0; v < V; v++)

         // Update dist[v] only if is not in sptSet, there is an edge from

         // u to v, and total weight of path from src to  v through u is

         // smaller than current value of dist[v]

         if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

                                       && dist[u]+graph[u][v] < dist[v])

            dist[v] = dist[u] + graph[u][v];

     }

     // print the constructed distance array

     printSolution(dist, V);

}

## Constant time — O(1).

def constant(n)

result = n \* n;

return result;

## Logarithmic time — O(log n)

1 def logarithmic(n):

2 result = 0

3 while n > 1:

4 n //= 2

5 result += 1

6 return result

## Linear time — O(n)

1 def linear(n, A):

2 for i in xrange(n):

3 if A[i] == 0:

4 return 0

5 return 1

## Quadratic time — O(n^2)

1 def quadratic(n):

2 result = 0

3 for i in xrange(n):

4 for j in xrange(i, n):

5 result += 1

6 return result

## Linear time — O(n + m)

1 def linear2(n, m):

2 result = 0

3 for i in xrange(n):

4 result += i

5 for j in xrange(m):

6 result += j

7 return result

## Slow Solution \_\_Time complexity O(n2)

1 def slow\_solution(n):

2 result = 0

3 for i in xrange(n):

4 for j in xrange(i + 1):

5 result += 1

6 return result

## Fast solution — time complexity O(n)

1 def fast\_solution(n):

2 result = 0

3 for i in xrange(n):

4 result += (i + 1)

5 return result

## Model solution — time complexity O(1)

1 def model\_solution(n):

2 result = n \* (n + 1) // 2

3 return result

## Fibonacci algorithm

int fib( int n )

{ int [] f = alloc\_

array(int,n+2);

f[0]=0;

f[1]=1;

for( int k = 2; k<=n; k++)

{f[k] = f[k-1] +f[k-2];}

return f[n];

}

## Prime Number Check

bool prime(int n)

{ int k = 2;

while( k <= sqrt(n) )

{

if( n%k == 0 )

return false;

k++;

}

return true;

}

## Linear Search Algo

find(arr, len, key)

{

i = 0

while(i < len)

{

if(arr[i] == key)

{return i }

i++

}

return -1

}

## General Algorithm

void fun(int n, int k)

{

    for (int i=1; i<=n; i++)

    {

      int p = pow(i, k);

      for (int j=1; j<=p; j++)

      {

          // Some O(1) work

      }

    }

}

## Alternate Sort of Array

// C++ program to print an array in alternate

// sorted manner.

#include <bits/stdc++.h>

using namespace std;

// Function to print alternate sorted values

void alternateSort(int arr[], int n)

{

    // Sorting the array

    sort(arr, arr+n);

    // Printing the last element of array

    // first and then first element and then

    // second last element and then second

    // element and so on.

    int i = 0, j = n-1;

    while (i < j) {

        cout << arr[j--] << " ";

        cout << arr[i++] << " ";

    }

    // If the total element in array is odd

    // then print the last middle element.

    if (n % 2 != 0)

        cout << arr[i];

}

## General Algorithm

// A STL based C++ program to sort a nearly sorted array.

#include <bits/stdc++.h>

using namespace std;

// Given an array of size n, where every element

// is k away from its target position, sorts the

// array in O(nLogk) time.

int sortK(int arr[], int n, int k)

{

    // Insert first k+1 items in a priority queue (or min heap)

    //(A O(k) operation). We assume, k < n.

    priority\_queue<int, vector<int>, greater<int> > pq(arr, arr + k + 1);

    // i is index for remaining elements in arr[] and index

    // is target index of for current minimum element in

    // Min Heapm 'hp'.

    int index = 0;

    for (int i = k + 1; i < n; i++) {

        arr[index++] = pq.top();

        pq.pop();

        pq.push(arr[i]);

    }

    while (pq.empty() == false) {

        arr[index++] = pq.top();

        pq.pop();

    }

}

## General Algorithm

TIME COMPLEXITY O(nlog(n))

// C++ program to sort an array according absolute

// difference with x.

#include<bits/stdc++.h>

using namespace std;

// Function to sort an array according absolute

// difference with x.

void rearrange(int arr[], int n, int x)

{

    multimap<int, int> m;

    multimap<int ,int >:: iterator it;

    // Store values in a map with the difference

    // with X as key

    for (int i = 0 ; i < n; i++)

        m.insert(make\_pair(abs(x-arr[i]),arr[i]));

    // Update the values of array

    int i = 0;

    for (it = m.begin(); it != m.end(); it++)

        arr[i++] = (\*it).second ;

}

## General Algorithm

// A O(n) program to sort an input array in wave form

#include<iostream>

using namespace std;

// A utility method to swap two numbers.

void swap(int \*x, int \*y)

{

    int temp = \*x;

    \*x = \*y;

    \*y = temp;

}

// This function sorts arr[0..n-1] in wave form, i.e., arr[0] >=

// arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5] ....

void sortInWave(int arr[], int n)

{

    // Traverse all even elements

    for (int i = 0; i < n; i+=2)

    {

        // If current even element is smaller than previous

        if (i>0 && arr[i-1] > arr[i] )

            swap(&arr[i], &arr[i-1]);

        // If current even element is smaller than next

        if (i<n-1 && arr[i] < arr[i+1] )

            swap(&arr[i], &arr[i + 1]);

    }

}

## General Algorithm

TIME COMPLEXITY O(m + n)

// C++ program to Merge an array of

// size n into another array of size m + n

#include <bits/stdc++.h>

using namespace std;

/\* Assuming -1 is filled for the places

   where element is not available \*/

#define NA -1

/\* Function to move m elements at

   the end of array mPlusN[] \*/

void moveToEnd(int mPlusN[], int size)

{

   int j = size - 1;

   for (int i = size - 1; i >= 0; i--)

     if (mPlusN[i] != NA)

     {

        mPlusN[j] = mPlusN[i];

        j--;

     }

}

/\* Merges array N[] of size n into

   array mPlusN[] of size m+n\*/

int merge(int mPlusN[], int N[], int m, int n)

{

   int i = n; /\* Current index of i/p part of mPlusN[]\*/

   int j = 0; /\* Current index of N[]\*/

   int k = 0; /\* Current index of of output mPlusN[]\*/

   while (k < (m + n))

   {

    /\* Take an element from mPlusN[] if

    a) value of the picked element is smaller

       and we have not reached end of it

    b) We have reached end of N[] \*/

    if ((i < (m + n) && mPlusN[i] <= N[j]) || (j == n))

    {

        mPlusN[k] = mPlusN[i];

        k++;

        i++;

    }

    else // Otherwise take element from N[]

    {

       mPlusN[k] = N[j];

       k++;

       j++;

    }

   }

}

## General Algorithm

TIME COMPLXITY O(n)

// Efficient C++ program to sort an array of

// numbers in range from 1 to n.

#include <iostream>

using namespace std;

// function for sort array

void sortit(int arr[], int n)

{

    for (int i = 0; i < n; i++) {

      arr[i]=i+1;

    }

}

## General Algorithm

TIME COMPLEXITY O(n)

// CPP program to test whether array

// can be sorted by swapping adjacent

// elements using boolean array

#include <bits/stdc++.h>

using namespace std;

// Return true if array can be

// sorted otherwise false

bool sortedAfterSwap(int A[], bool B[], int n)

{

    int i, j;

    // Check bool array B and sorts

    // elements for continuos sequnce of 1

    for (i = 0; i < n - 1; i++) {

        if (B[i]) {

            j = i;

            while (B[j])

                j++;

            // Sort array A from i to j

            sort(A + i, A + 1 + j);

            i = j;

        }

    }

    // Check if array is sorted or not

    for (i = 0; i < n; i++) {

        if (A[i] != i + 1)

            return false;

    }

    return true;

}

## General Algorithm

// CPP program to sort an array with two types

// of values in one traversal.

#include <bits/stdc++.h>

using namespace std;

/\* Method for segregation 0 and 1 given

   input array \*/

void segregate0and1(int arr[], int n)

{

    int type0 = 0;

    int type1 = n - 1;

    while (type0 < type1) {

        if (arr[type0] == 1) {

            swap(arr[type0], arr[type1]);

            type1--;

        }

        else {

            type0++;

        }

    }

}

## General Algorithm

// Sort elements by frequency. If two elements have same

// count, then put the elements that appears first

#include<bits/stdc++.h>

using namespace std;

// Used for sorting

struct ele

{

    int count, index, val;

};

// Used for sorting by value

bool mycomp(struct ele a, struct ele b) {

    return (a.val < b.val);

}

// Used for sorting by frequency. And if frequency is same,

// then by appearance

bool mycomp2(struct ele a, struct ele b) {

    if (a.count != b.count) return (a.count < b.count);

    else return a.index > b.index;

}

void sortByFrequency(int arr[], int n)

{

    struct ele element[n];

    for (int i = 0; i < n; i++)

    {

        element[i].index = i;    /\* Fill Indexes \*/

        element[i].count = 0;    /\* Initialize counts as 0 \*/

        element[i].val = arr[i]; /\* Fill values in structure

                                     elements \*/

    }

    /\* Sort the structure elements according to value,

       we used stable sort so relative order is maintained. \*/

    stable\_sort(element, element+n, mycomp);

    /\* initialize count of first element as 1 \*/

    element[0].count = 1;

    /\* Count occurrences of remaining elements \*/

    for (int i = 1; i < n; i++)

    {

        if (element[i].val == element[i-1].val)

        {

            element[i].count += element[i-1].count+1;

            /\* Set count of previous element as -1 , we are

               doing this because we'll again sort on the

               basis of counts (if counts are equal than on

               the basis of index)\*/

            element[i-1].count = -1;

            /\* Retain the first index (Remember first index

               is always present in the first duplicate we

               used stable sort. \*/

            element[i].index = element[i-1].index;

        }

        /\* Else If previous element is not equal to current

          so set the count to 1 \*/

        else element[i].count = 1;

    }

    /\* Now we have counts and first index for each element so now

       sort on the basis of count and in case of tie use index

       to sort.\*/

    stable\_sort(element, element+n, mycomp2);

    for (int i = n-1, index=0; i >= 0; i--)

        if (element[i].count != -1)

           for (int j=0; j<element[i].count; j++)

                arr[index++] = element[i].val;

}

## General Algorithm

numbers = array (14, 82, 4, 0, 24, 28);

Foreach (numbers as number)

{

Echo number;

}

So the time complexity for this algorithm is O (n).

## General Algorithm

numbers = array (14, 82, 4, 0, 24, 28);

foreach (numbers as number1)

{

foreach (numbers as number2)

{

If (number1 >= number2)

{

Echo number1. “Is greater than or equal to " . number2;

}

else

{

Echo number1. “Is smaller than “. number2;

}

}

}

So the time complexity of this algorithm is O (n^2).

## General Algorithm

numbers = array (14,82,4,0,24,28);

foreach ($numbers as $number1)

{

foreach($numbers as $number2)

{

if($number1 >= $number2)

{

echo $number1 . " is greater than or equal to " . $number2;

}

else

{

echo $number1 . " is smaller than " . $number2;

}

}

}

foreach($numbers as $number)

{

echo $number;

}

So the complexity of this function is O (n^2 + n).

## General Algorithm

function trivial2 (list input):

foreach element of input:

sleep for logN seconds

return True

So the complexity of this function is nlogn.

## General Algorithm

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

Sequence of statements

}

So the complexity of this algorithm is O (n).

## General Algorithm

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

for (j = 0; j < M; j++) {

sequence of statements

}

}

So the complexity of this algorithm is O (n^2).

## General Algorithm

int FindMaxElement(int[] array)

{

int max = int.MinValue;

for (int i = 1; i < array.Length; i++)

{

if (array[i] > max)

{

max = array[i];

}

}

return max;

}

So complexity of this function is O(n^2).

## General Algorithm

int FindInversions(int[] array)

{

int inversions = 0;

for (int i = 0; i < array.Length - 1; i++)

{

for (int j = i + 1; j < array.Length; j++)

{

if (array[i] > array[j])

{

inversions++;

}

}

}

return inversions;

}

 complexity of this function is O(n^2).

## General Algorithm

// Quadratic complexity: O(N\*M)

// Innermost loop executes N\*min(N,M); thus, this algorithm is not of cubic complexity.

long SumMN(int n, int m)

{

long sum = 0;

for (int x = 1; x <= n; x++)

{

for (int y = 1; y <= m; y++)

{

if (x == y)

{

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

{

sum += i \* x \* y;

}

}

}

}

return sum;

}

So complexity of this function is O(n^2).

## General Algorithm

// Exponential complexity: O(2^n)

long Fibonacci(int n)

{

if (n == 0)

{

return 1;

}

else if (n == 1)

{

return 1;

}

else

{

return Fibonacci(n - 1) + Fibonacci(n - 2);

}

}

So complexity of this function is O(2^n).

## General Algorithm

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

{

statement;

}

So, complexity of this function is O(n^2).

## General Algorithm

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

{

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

{

statement;

}

}

So complexity of this function is O(n^2).

## General Algorithm

TIME COMPLEXITY O(n^2)

# include <stdio.h>

# include <stdlib.h> /\* for abs() \*/

# include <math.h>

void minAbsSumPair(int arr[], int arr\_size)

{

  int inv\_count = 0;

  int l, r, min\_sum, sum, min\_l, min\_r;

  /\* Array should have at least two elements\*/

  if(arr\_size < 2)

  {

    printf("Invalid Input");

    return;

  }

  /\* Initialization of values \*/

  min\_l = 0;

  min\_r = 1;

  min\_sum = arr[0] + arr[1];

  for(l = 0; l < arr\_size - 1; l++)

  {

    for(r = l+1; r < arr\_size; r++)

    {

      sum = arr[l] + arr[r];

      if(abs(min\_sum) > abs(sum))

      {

        min\_sum = sum;

        min\_l = l;

        min\_r = r;

      }

    }

  }

## General Algorithm

// A C++ program to find a peak element element using divide and conquer

#include <stdio.h>

// A binary search based function that returns index of a peak element

int findPeakUtil(int arr[], int low, int high, int n)

{

    // Find index of middle element

    int mid = low + (high - low)/2;  /\* (low + high)/2 \*/

    // Compare middle element with its neighbours (if neighbours exist)

    if ((mid == 0 || arr[mid-1] <= arr[mid]) &&

            (mid == n-1 || arr[mid+1] <= arr[mid]))

        return mid;

    // If middle element is not peak and its left neighbour is greater

    // than it, then left half must have a peak element

    else if (mid > 0 && arr[mid-1] > arr[mid])

        return findPeakUtil(arr, low, (mid -1), n);

    // If middle element is not peak and its right neighbour is greater

    // than it, then right half must have a peak element

    else return findPeakUtil(arr, (mid + 1), high, n);

}

// A wrapper over recursive function findPeakUtil()

int findPeak(int arr[], int n)

{

    return findPeakUtil(arr, 0, n-1, n);

}

## General Algorithm

TIME COMPLEXITY O(n\*n)

#include<stdio.h>

#include<stdlib.h>

void printRepeating(int arr[], int size)

{

  int i, j;

  printf(" Repeating elements are ");

  for(i = 0; i < size; i++)

    for(j = i+1; j < size; j++)

      if(arr[i] == arr[j])

        printf(" %d ", arr[i]);

}

## General Algorithm

TIME COMPLEXITY O(n)

// CPP program to find the only repeating

// element in an array where elements are

// from 1 to n-1.

#include <bits/stdc++.h>

using namespace std;

int findRepeating(int arr[], int n)

{

   // Find array sum and subtract sum

   // first n-1 natural numbers from it

   // to find the result.

   return accumulate(arr , arr+n , 0) -

                   ((n - 1) \* n/2);

}

## General Algorithm

// C++ program to print common elements in three arrays

#include <iostream>

using namespace std;

// This function prints common elements in ar1

void findCommon(int ar1[], int ar2[], int ar3[], int n1, int n2, int n3)

{

    // Initialize starting indexes for ar1[], ar2[] and ar3[]

    int i = 0, j = 0, k = 0;

    // Iterate through three arrays while all arrays have elements

    while (i < n1 && j < n2 && k < n3)

    {

         // If x = y and y = z, print any of them and move ahead

         // in all arrays

         if (ar1[i] == ar2[j] && ar2[j] == ar3[k])

         {   cout << ar1[i] << " ";   i++; j++; k++; }

         // x < y

         else if (ar1[i] < ar2[j])

             i++;

         // y < z

         else if (ar2[j] < ar3[k])

             j++;

         // We reach here when x > y and z < y, i.e., z is smallest

         else

             k++;

    }

}

## General Algorithm

TIME COMPLEXITY O(n\*n)

#include<iostream>

using namespace std;

/\*C++ Function to print leaders in an array \*/

void printLeaders(int arr[], int size)

{

    for (int i = 0; i < size; i++)

    {

        int j;

        for (j = i+1; j < size; j++)

        {

            if (arr[i] <= arr[j])

                break;

        }

        if (j == size) // the loop didn't break

            cout << arr[i] << " ";

  }

}

## General Algorithm

#include<stdio.h>

/\* Function to get index of ceiling of x in arr[low..high] \*/

int ceilSearch(int arr[], int low, int high, int x)

{

  int i;

  /\* If x is smaller than or equal to first element,

    then return the first element \*/

  if(x <= arr[low])

    return low;

  /\* Otherwise, linearly search for ceil value \*/

  for(i = low; i < high; i++)

  {

    if(arr[i] == x)

      return i;

    /\* if x lies between arr[i] and arr[i+1] including

       arr[i+1], then return arr[i+1] \*/

    if(arr[i] < x && arr[i+1] >= x)

       return i+1;

  }

  /\* If we reach here then x is greater than the last element

    of the array,  return -1 in this case \*/

  return -1;

}