**Algorithms**

**Binary Search:** Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

public int RecursiveBinarySearch(int[] arr, int startIndex, int endIndex, int val)

{

if (endIndex >= startIndex)

{

int mid = (startIndex + endIndex) / 2;

// If the element is present at the

// middle itself

if (arr[mid] == val)

{

return mid;

}

// If element is smaller than mid, then

// it can only be present in left subarray

if (arr[mid] > val)

{

return RecursiveBinarySearch(arr, startIndex, mid - 1, val);

}

// Else the element can only be present

// in right subarray

return RecursiveBinarySearch(arr, mid + 1, endIndex, val);

}

// We reach here when element is not present

// in array

return -1;

}

**Jump Search :** Like Binary Search, Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements**.**

What is the optimal block size to be skipped?

In the worst case, we have to do n/m jumps and if the last checked value is greater than the element to be searched for, we perform m-1 comparisons more for linear search. Therefore the total number of comparisons in the worst case will be ((n/m) + m-1). The value of the function ((n/m) + m-1) will be minimum when m = √n. Therefore, the best step size is m = √n.

public static int jumpSearch(int[] arr, int x)

    {

        int n = arr.Length;

        // Finding block size to be jumped

        int step = (int)Math.Floor(Math.Sqrt(n));

        // Finding the block where element is

        // present (if it is present)

        int prev = 0;

        while (arr[Math.Min(step, n)-1] < x)

        {

            prev = step;

            step += (int)Math.Floor(Math.Sqrt(n));

            if (prev >= n)

                return -1;

        }

        // Doing a linear search for x in block

        // beginning with prev.

        while (arr[prev] < x)

        {

            prev++;

            // If we reached next block or end of

            // array, element is not present.

            if (prev == Math.Min(step, n))

                return -1;

        }

        // If element is found

        if (arr[prev] == x)

            return prev;

        return -1;

    }

**Exponential Search** :

Exponential search involves two steps:

1. Find range where element is present
2. Do Binary Search in above found range.

**How to find the range where element may be present?**  
The idea is to start with subarray size 1, compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater.  
Once we find an index i (after repeated doubling of i), we know that the element must be present between i/2 and i (Why i/2? because we could not find a greater value in previous iteration)

static int exponentialSearch(int []arr,

                         int n, int x)

{

    // If x is present at

    // first location itself

    if (arr[0] == x)

        return 0;

    // Find range for binary search

    // by repeated doubling

    int i = 1;

    while (i < n && arr[i] <= x)

        i = i \* 2;

    // Call binary search for

    // the found range.

    return binarySearch(arr, i/2,

                       Math.Min(i, n), x);

}

**Interpolation Search :** Given a sorted array of n uniformly distributed values arr[], write a function to search for a particular element x in the array.

Linear Search finds the element in O(n) time, Jump Search takes O(√ n) time and Binary Search take O(Log n) time.

The Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to the middle element to check. On the other hand, interpolation search may go to different locations according to the value of the key being searched. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side.

// The idea of formula is to return higher value of **pos**

// when element to be searched is closer to **arr[hi]**. And

// smaller value when closer to **arr[lo]**

pos = lo + [ (x-arr[lo])\*(hi-lo) / (arr[hi]-arr[Lo]) ]

arr[] ==> Array where elements need to be searched

x ==> Element to be searched

lo ==> Starting index in arr[]

hi ==> Ending index in arr[]

 static int interpolationSearch(int x)

    {

        // Find indexes of

        // two corners

        int lo = 0, hi = (arr.Length - 1);

        // Since array is sorted,

        // an element present in

        // array must be in range

        // defined by corner

        while (lo <= hi &&

                x >= arr[lo] &&

                x <= arr[hi])

        {

            if (lo == hi)

            {

                if (arr[lo] == x) return lo;

                return -1;

            }

            // Probing the position

            // with keeping uniform

            // distribution in mind.

            int pos = lo + (((hi - lo) /

                             (arr[hi] - arr[lo])) \*

                                   (x - arr[lo]));

            // Condition of

            // target found

            if (arr[pos] == x)

                return pos;

            // If x is larger, x

            // is in upper part

            if (arr[pos] < x)

                lo = pos + 1;

            // If x is smaller, x

            // is in the lower part

            else

                hi = pos - 1;

        }

        return -1;

    }

**Sorting Algos**

**Selection Sort :** The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.  
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

**11** 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 **12** 25 22 64

// Find the minimum element in arr[2...4]

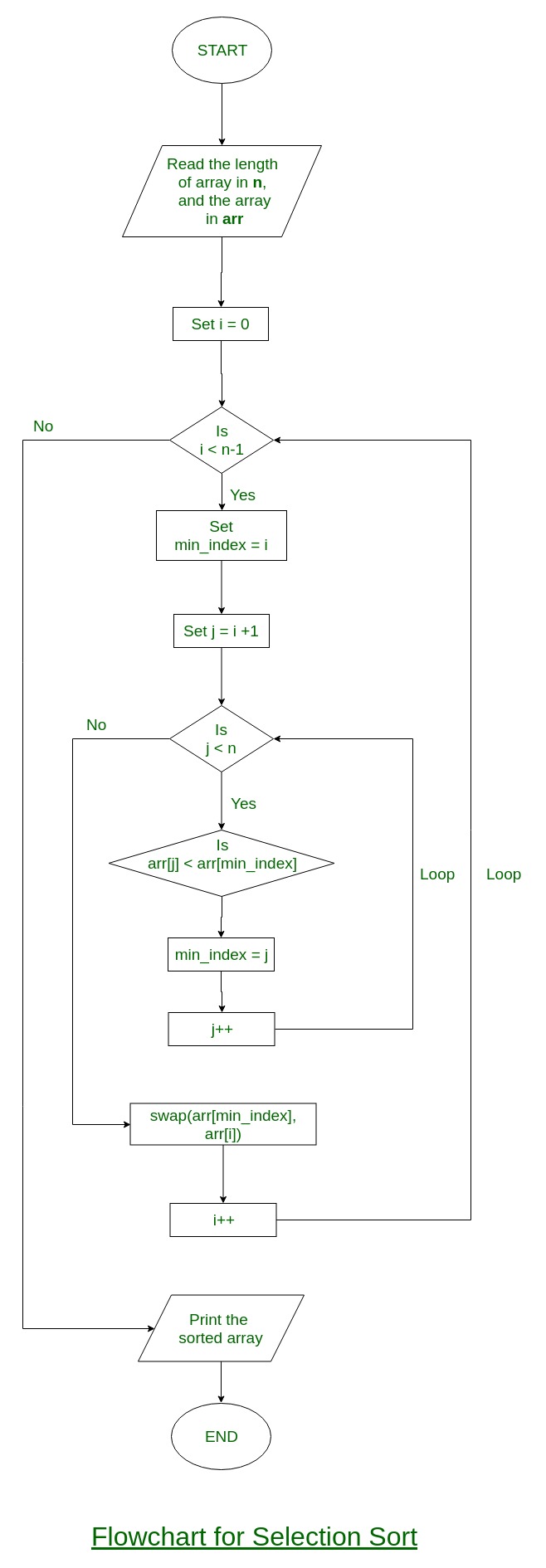
// and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 **25** 64



public int[] SelectionSortMethod(int[] arr)

{

var len = arr.Length;

for (var i = 0; i < len; i++)

{

var min = i;

for (var j = i + 1; j < len; j++)

{

if (arr[min] > arr[j])

{

min = j;

}

var temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

}

return arr;

}

**Insertion Sort**

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

 void sort(int[] arr)

    {

        int n = arr.Length;

        for (int i = 1; i < n; ++i) {

            int key = arr[i];

            int 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;

        }

    }

**Merge Sort**

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

L = start index

R = end index

M =middle index

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

// 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);

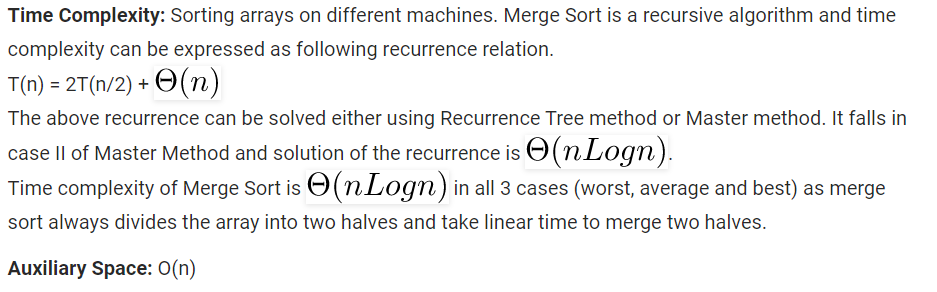
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);

    }

}

Time complexity of merge sort explained : <https://www.youtube.com/watch?v=g1AwUYauqgg>



Merge sort Vs Quick Sort

<https://www.youtube.com/watch?v=es2T6KY45cA>

**QuickSort**

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

* Always pick first element as pivot.
* Always pick last element as pivot (implemented below)
* Pick a random element as pivot.
* Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time

quickSort(arr[], low, high)

{

if (low < high)

{

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

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

partition (arr[], low, high)

{

// pivot (Element to be placed at right position)

pivot = arr[high];

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

for (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] and arr[j]

}

}

swap arr[i + 1] and arr[high])

return (i + 1)

}

**Analysis of QuickSort**  
Time taken by QuickSort in general can be written as following.

T(n) = T(k) + T(n-k-1) + (n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.  
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

***Worst Case:*** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.

When K =n-1;

T(n) = T(0) + T(n-1) + (n)

which is equivalent to

T(n) = T(n-1) + (n)

The solution of above recurrence is (n2).

***Best Case:*** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

When K = n/2-1;

T(n) = 2T(n/2) + (n)

The solution of above recurrence is (nLogn). It can be solved using case 2 of [Master Theorem](http://en.wikipedia.org/wiki/Master_theorem).

# HeapSort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

**What is**[**Binary Heap**](https://www.geeksforgeeks.org/binary-heap/)**?**  
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source [Wikipedia](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees))

A [Binary Heap](https://www.geeksforgeeks.org/binary-heap/) is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

**Heap Sort Algorithm for sorting in increasing order:**  
**1.** Build a max heap from the input data.  
**2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.  
**3.** Repeat above steps while size of heap is greater than 1.

**How to build the heap?**  
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

public void Sort(int [] arr)

{

var len = arr.Length;

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

{

Heapify(arr, len, i);

}

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

{

var temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

Heapify(arr, i, 0);

}

}

public void Heapify(int [] arr, int arrSize, int nodeIndex)

{

var largest = nodeIndex;

var left = 2 \* nodeIndex;

var right = (2 \* nodeIndex) + 1;

if(left<arrSize && arr[largest] < arr[left])

{

largest = left;

}

if (right < arrSize && arr[largest] < arr[right])

{

largest = right;

}

if(largest!= nodeIndex)

{

var temp = arr[nodeIndex];

arr[nodeIndex] = arr[largest];

arr[largest] = temp;

Heapify(arr, arrSize, largest);

}

}

Heap sort is an in-place algorithm.  
Its typical implementation is not stable, but can be made stable (See [this](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/))

**Time Complexity:**Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

**Applications of HeapSort**  
**1.** [Sort a nearly sorted (or K sorted) array](https://www.geeksforgeeks.org/nearly-sorted-algorithm/)  
**2.**[k largest(or smallest) elements in an array](https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/)

K sorted explantion

<https://www.youtube.com/watch?v=yQ84lk-EXTQ>

**Radix Sort**

<https://www.youtube.com/watch?v=JMlYkE8hGJM>

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.

Counting sort is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

# Counting Sort

[Counting sort](http://en.wikipedia.org/wiki/Counting_sort) is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

<https://www.youtube.com/watch?v=pEJiGC-ObQE&list=PLdo5W4Nhv31bEiyP4tclZMc5mP_X7OD7k&index=11&t=0s>

**Bucket Sort**

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.

Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.