**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]

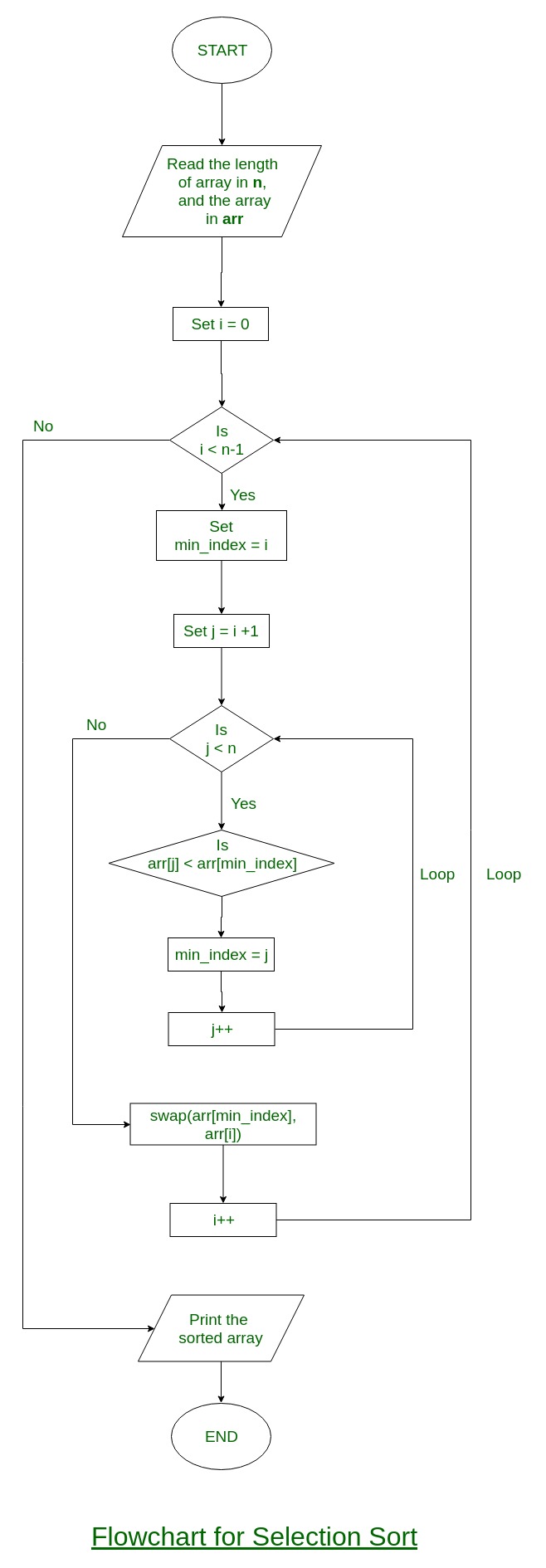
// and place it at beginning of arr[2...4]

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;

}