SORTING AND SEARCHING ALGORITHMS

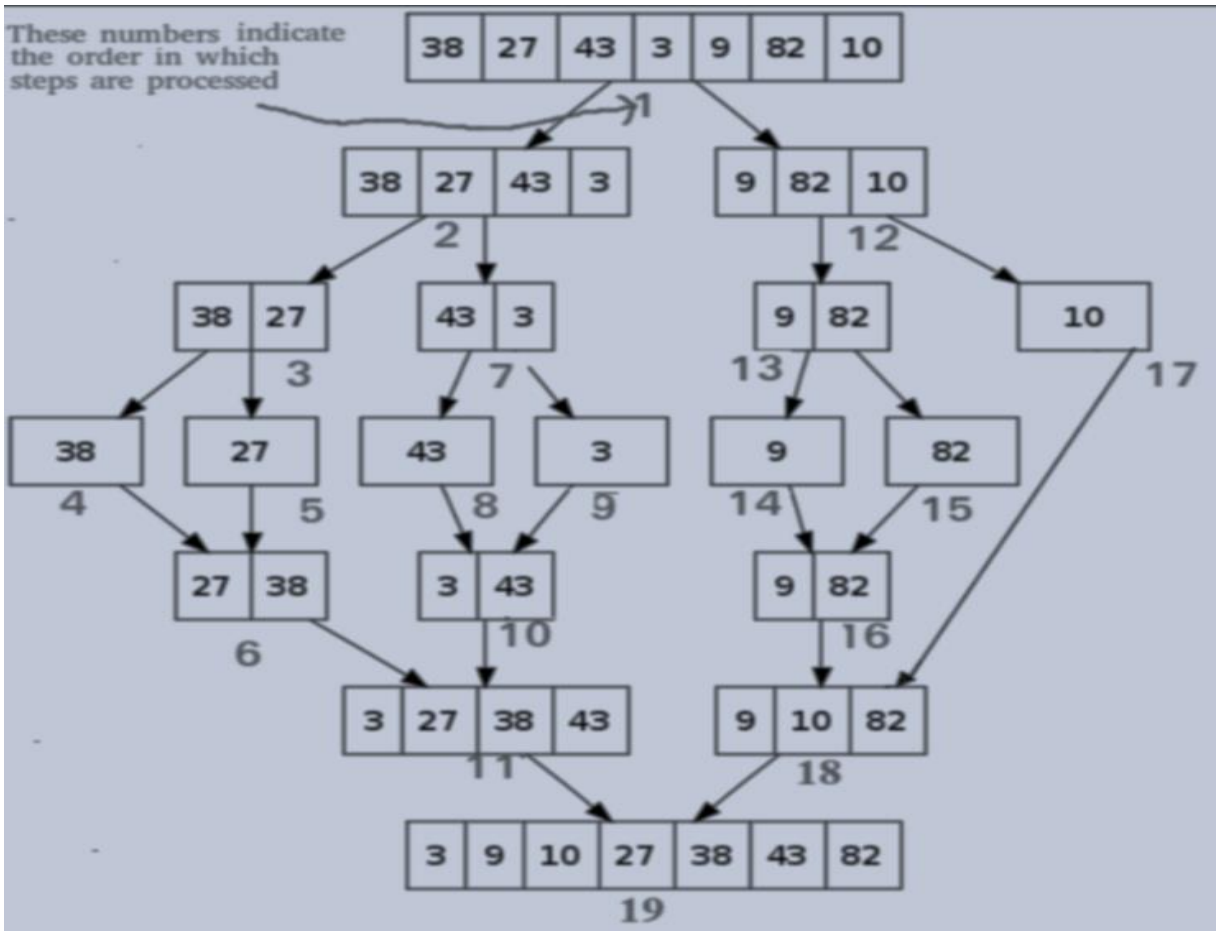
Merge Sort

The Merge Sort algorithm is a sorting algorithm that is based on the Divide and Conquer Methodology. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Process:

1. it as a recursive algorithm continuously splits the array in half until it cannot be further divided.
2. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion.
3. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves.
4. Finally, when both halves are sorted, the merge operation is applied.
5. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Example:



Algorithm:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

```
if left > right
    return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

step 4: Stop

Merge Sort merge function:

```
MergeSort(arr[], l,  r)
If r > l
Find the middle point to divide the array into two halves:
middle m = l + (r – l)/2
Call mergeSort for first half:
Call mergeSort(arr, l, m)
Call mergeSort for second half:
Call mergeSort(arr, m + 1, r)
Merge the two halves sorted in steps 2 and 3:
Call merge(arr, l, m, r)
```

Time complexity:

(N log(N)), Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \theta(n)$$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort is O(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

in merge sort the merging step requires extra space to store the elements.

Quick Sort

Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot.

There are many different versions of quick Sort that pick pivot in different ways.

This sorting technique is considered unstable since it does not maintain the key-value pairs initial order.

1. Always pick the first element as a pivot.
2. Always pick the last element as a pivot (implemented below)
3. Pick a random element as a pivot.
4. Pick median as the pivot.

Process:

Quick Sort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a 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.

Quick Sort is an efficient algorithm that performs well in practice.

Partition Algorithm:

1. we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i.
2. While traversing, if we find a smaller element, we swap the current element with arr[i].
3. Otherwise, we ignore the current element.

Pseudo Code:

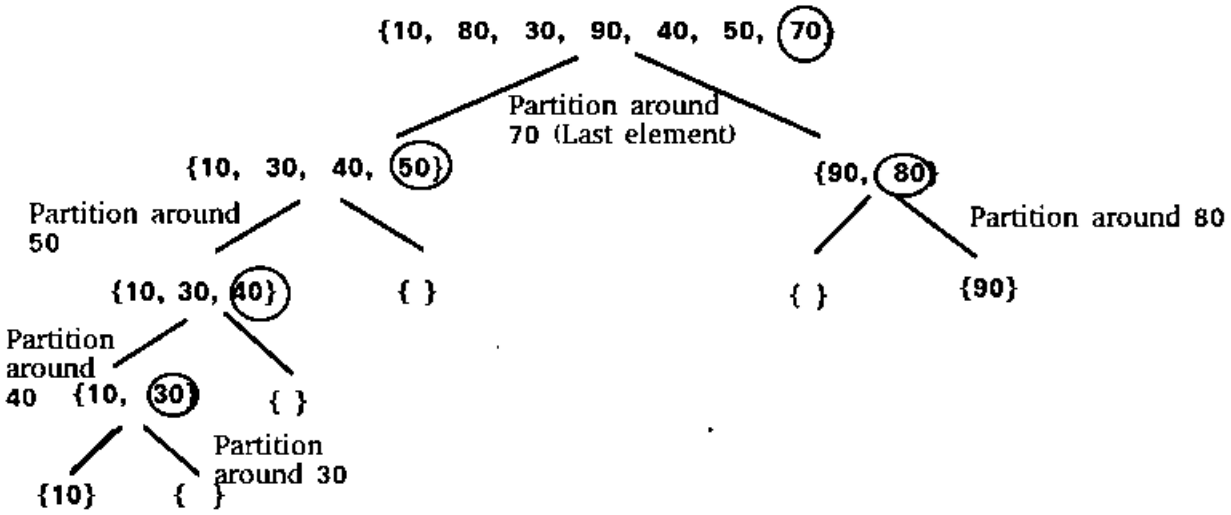
```
partition (array, start, end)
{
    // Setting rightmost Index as pivot
    pivot = arr[end];

    i = (start - 1)  // Index of smaller element and indicates the
                     // right position of pivot found so far

for (j = start; j <= end- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[end])
    return (i + 1)
}
```

```
//start -> Starting index,  end --> Ending index
Quicksort(array, start, end)
{
   if (start < end)
   {
      pIndex = Partition(A, start, end)
      Quicksort(A,start,pIndex-1)
      Quicksort(A,pIndex+1, end)
   }
}
```

EXAMPLE:



Time Complexity

Best Case Complexity:

1. Quicksort's best-case time complexity is O (n*logn).
2. T(n) = 2T(n/2) + O(n)

Average Case Complexity:

1. Quicksort's average case time complexity is O(n*logn)
2. T(n) = T(n/9) + T(9n/10) + O(n)

Worst Case Complexity:

1. The worst-case time complexity of quicksort is O (n2)
2. T(n) = T(0) + T(n-1) + O(n)

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

the bubble sort algorithm is stable.

Bubble sort performs the swapping of adjacent pairs without the use of any major data structure. Hence Bubble sort algorithm is an in-place algorithm.

Process:

1. Run a nested for loop to traverse the input array using two variables i and j, such that $0 \leq i < n-1$ and $0 \leq j < n-i-1$
2. If arr[j] is greater than arr[j+1] then swap these adjacent elements, else move on
3. Print the sorted array

Example

		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i=1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i=2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i=3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	4				
	1	1	2	3					
i=6	0	1	2	3					
		1	2						

Time Complexity

Worst and Average Case Time Complexity: O(N2). The worst case occurs when an array is reverse sorted.Best Case Time Complexity: O(N). The best case occurs when an array is already sorted.

Auxiliary Space: O(1)

Binary Search

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

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

Binary Search Algorithm:

Begin with the mid element of the whole array as a search key.

If the value of the search key is equal to the item then return an index of the search key.

Or 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 from the second point until the value is found or the interval is empty.

Pseudo Code:

```
int binarySearch(int X[], int l, int r, int key)
{
    if (l > r)
        return -1
    else
    {
        int mid = l + (r - l) / 2
        if (X[mid] == key)
            return mid
        if (X[mid] > key)
            return binarySearch(X, l, mid - 1, key)
        else
            return binarySearch(X, mid + 1, r, key)
    }
}
```

Time Complexity

Binary Search time complexity analysis is done below-

In each iteration or in each recursive call, the search gets reduced to half of the array.

So for n elements in the array, there are log2n iterations or recursive calls.

Time Complexity of Binary Search Algorithm is O(log2n).

Here, n is the number of elements in the sorted linear array.

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

The subarray which already sorted.

The remaining subarray was unsorted.

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

Pesudo Algorithm

```
procedure selection sort
arr : array of integers
n : size of the array

for i =1 to n-1
// set current element as minIndex
minIndex=i
// check for all other element(right side of current element)
for j =i+1 to n
      if(arr[j] < arr[minIndex])
         minIndex=j
      end if
end for

// swap  the current element with the minimum element to the right side
swap(arr[minIndex], arr[i])
end for
End procedure
```

Example

Unsorted list	Least element	Sorted list
{18,10,7,20,2}	2	{}
{18,10,7,20}	7	{2}
{18,10,20}	10	{2,7}
{18,20}	18	{2,7,10)
{20}	20	{2,7,10,18}
{}		{2,7,10,18,20}

Time Complexity

The time complexity of Selection Sort is O(N2) as there are two nested loops:

One loop to select an element of Array one by one = O(N)

Another loop to compare that element with every other Array element = O(N)

Therefore, overall complexity = O(N) * O(N) = O(N*N) = O(N2)

Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array.

The selection sort never makes more than O(N) swaps and can be useful when memory write is a costly operation.

Selection Sort Complexity

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)
Stability	No

TIME COMPLEXITY CHART

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)