SNS COLLEGE OF ENGINEERING

Coimbatore-107

An Autonomous Institution

Accredited by AICTE and Accredited by NAAC – UGC with ‘A’ Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME: ANALYSIS OF ALGORITHMS

II YEAR/ IV SEMESTER

UNIT – II

BRUTE FORCE METHOD & DIVIDE AND CONQUER METHOD

Topic

Brute Force Method: Selection sort- Bubble Sort-Sequential Search

Divide and conquer methodology: Quick sort – Merge sort – Binary search

Brute Force Method

- It is a straightforward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.
- Many problems are solved in day-to-day life using the brute force strategy, for example, exploring all the paths to a nearby market to find the minimum shortest path.
- The brute force method is ideal for solving small and simpler problems.
- The brute force approach is inefficient and slow than other methods.

Topic 1: Selection Sort (using Brute Force Approach)

Selection Sort is a comparison-based sorting algorithm.

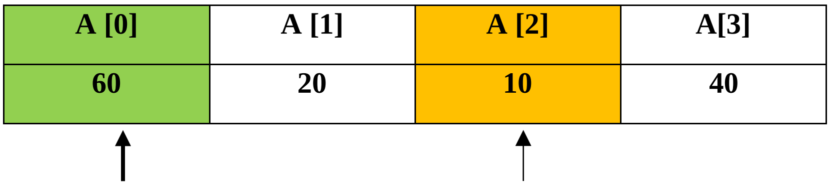
1. First, we find the smallest element and compare it with the first element in array.
2. If first element is smaller, Swapping is done. Otherwise, comparison is made with second element and smallest element. Swapping is done, if second element is smaller.
3. This is process is continued with all the elements in the array until entire list is sorted.

Example:

Step 1: When i=0;

Compare; A [0] =60 First Element and A [2] =10 Smallest Element;

First Element 60 is Greater. So, Swap 60 & 10; i++

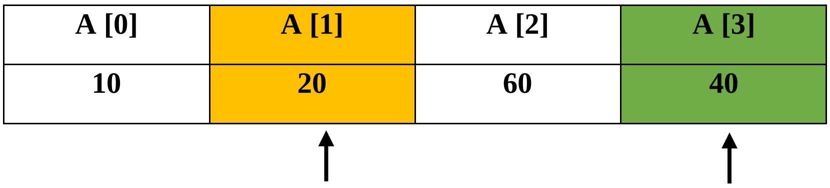


A [0]	A [1]	A [2]	A[3]
60	20	10	40

Step 2: When i=1;

Compare; A[1] =20 Next Element and A[2] =40 Smallest Element;

Next Element 20 is Smaller. No Swap; i++



A [0]	A [1]	A [2]	A [3]
10	20	60	40

Step 3: When i=2;

Compare; A [2] =60 Nextt Element and A [3] =40 Smallest Element;

Next Element 60 is Greater. So, Swap 60 & 40; i++

A[0]	A[1]	A[2]	A[3]
10	20	60	40

Final Sorted List:

A[0]	A[1]	A[2]	A[3]
10	20	40	60

Algorithm:

Algorithm SelectionSort(Arr[n])

```
{
   for ( i = 0; i < n - 2; i++) {
     // Assume the current position holds the minimum element
     min = A[i];
   // Iterate through the unsorted portion to find the actual minimum
      for ( j = i + 1; j < n-1; j++)
{ min=findmin(A[j])
         if (A[i] > A[min])
          {
          temp=A[i]
          A[i]=A[min]
          A[min]=temp
          }
}
 // Move minimum element to its correct position
      int temp = arr[i];
      arr[i] = arr[min_idx];
```

```
        arr[min_idx] = temp;
    }
}
```

Complexity Analysis:

Time Complexity: (Best , Worst & Average Case)

- compare that element with every other Array element taken as:

  $(n-1) + (n-2) + \ldots\ldots + 2 + 1 = n(n-1)/2$ By (Summation Formula):

  $= (n^2-n)/2$

  Order of Polynomial is n^2.

  Therefore **$T(n) = O(n^2)$**

Space Complexity: $S(n) = O(1)$, Since no extra memory used is for temporary variables.

Topic 2: Bubble Sort (using Brute Force Approach)

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

It compares First element with the consecutive next elements and swaps then and there with smallest elements until last element.

Again, Compares first element and smaller element until last element. Same process is continued until entire elements are sorted.

Example: **PASS1:**

Step 1: When i=0; j=0

A [0]	**A [1]**	**A [2]**	**A[3]**	**A[4]**
70	**30**	**20**	**40**	**35**

Compare; A [0] =70 & A [1] =30 ; 70 greater; So Swap 70 & 30; j++;

Step 2: When i=0; j=1

A [0]	A [1]	A [2]	A[3]	A[4]
30	70	20	40	35

Compare; A [1] =70 & A [2] =20 ; 70 greater; So Swap 70 & 20; j++;

Step 3: When i=0; j=2

A [0]	A [1]	A [2]	A[3]	A[4]
30	20	70	40	35

Compare; A [2] =70 & A [3] =40 ; 70 greater; So Swap 70 & 40; j++

Step 4: When i=0; j=3

A [0]	A [1]	A [2]	A[3]	A[4]
30	20	40	70	35

Compare; A [2] =70 & A [3] =35 ; 70 greater; So Swap 70 & 35; j++

Step 4: When i=0; j=4;condition become false.

A [0]	A [1]	A [2]	A[3]	A[4]
30	20	40	35	70

j iteration stops since j is not greater than n-1.

PASS:2

Step 1: When i=1; j=0

A [0]	A [1]	A [2]	A[3]	A[4]
30	20	40	35	70

Compare; A [0] =30 & A [1] =20; 30 greater; So Swap 30 & 20; j++;

Step 2: When i=1; j=1

A [0]	A [1]	A [2]	A[3]	A[4]
20	30	40	35	70

Compare; A [1] =30 & A [2] =40 ; 30 smaller; No Swap; j++;

Step 2: When i=1; j=2

A [0]	A [1]	A [2]	A[3]	A[4]
20	30	40	35	70

Compare; A [2] =40 & A [3] =35 ; 40 Greater; So Swap 35 & 40; j++;

Step 2: When i=1; j=3

A [0]	A [1]	A [2]	A[3]	A[4]
20	30	35	40	70

Compare; A [3] =40 & A [4] =70 ; 40 Smaller; No Swap; j++;

Step 4: When i=0; j=4;Condition become false.

j iteration stops; List is Sorted.

Algorithm:

Algorithm BubbleSort(A[0..n-1], n)

```
{
    for (i = 0; i < n - 1; i++)// move through passes
 {
     flag=0;
          for (j = 0; j < n - i - 1; j++) //Compare 1 element with next element
{
             if (A[j] > A[j + 1])
```

```
{
temp=A[j];
A[j]=A[j+1];
A[j+1]=temp
flag=1
            }
 }
}
if(flag==0)
break;
}
```

Analysis:

Time Complexity:

1.Maximum Comparisons made: = (n-1) comparisons

1+2+…...(n-1)= Sum of 'n' Natural numbers

=(n(n-1))/2

= (n2-n)/2

Degree of Polynomial=n2; Order=f(n)=O(n2)

2.Maximum Swaps made: = (n-1) comparisons

1+2+…...(n-1) = Sum of 'n' Natural numbers

=(n(n-1))/2

= (n2-n)/2

Degree of Polynomial=n2; Order=f(n)=O(n2)

Bubble Sort	Time Complexity			Space Complexity
	Best Case	Avg. Case	Worst Case	
	O(n)	O(n^2)	O(n^2)	O(1)

Space Complexity:

Bubble Sort is an in-place sorting algorithm, meaning it does not require any additional memory that grows with the size of the input list (apart from a small, constant amount of space for variables like the loop counters or temporary values for swapping). Therefore, its space complexity is O (1), which means it only uses a constant amount of space.

Topic 3-Sequential Search (or) Linear Search

(using Brute Force Approach)

This uses Brute Force Method that search the Key Element(required) by comparing each successive element with the Key element.

If found; Search is Success and index of Key element is returned. Otherwise, Key element is not present.

Example:

A [0]	A [1]	A [2]	A[3]	A[4]
30	20	40	35	70

Key=40;

1. Search Key with index 0; Element does not match key. So, increment index
2. Search Key with index 1; Element does not match key. So, increment index
3. Search Key with index 2; Element matches key. So, return index=2.

Algorithm:

Algorithm seqsearch(A[0…n-1], n,Key)

```
{
  for (int i = 0; i < n; i++)
    if (A[i] == Key)
      return i;
  return -1;
```

}

Analysis:

Time Complexity:

- **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
- **Worst Case:** In the worst case, the key might be present at the last index. So, the worst-case complexity is O(n) where N is the size of the list.
- **Average Case:** Algorithm runs half of the element (n/2). Order=Degree of Polynomial=O(n).

Space Complexity:

- ❖ O(1) as except for the variable to iterate through the list, no other variable is used. (Iterative Version)

Divide and Conquer Approach

Divide and Conquer Algorithm involves breaking a larger problem into smaller subproblems, solving them independently, and then combining their solutions to solve the original problem. The basic idea is to recursively divide the problem into smaller subproblems until they become simple enough to be solved directly. Once the solutions to the subproblems are obtained, they are then combined to produce the overall solution.

The main steps are:

Divide: Break the problem into smaller subproblems.

Conquer: Solve the subproblems recursively.

Combine: Merge or combine the solutions of the subproblems to obtain the solution to the original problem.

Topic 4: Quick Sort

(using Divide & Conquer Approach)

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

There are mainly three steps in the algorithm:

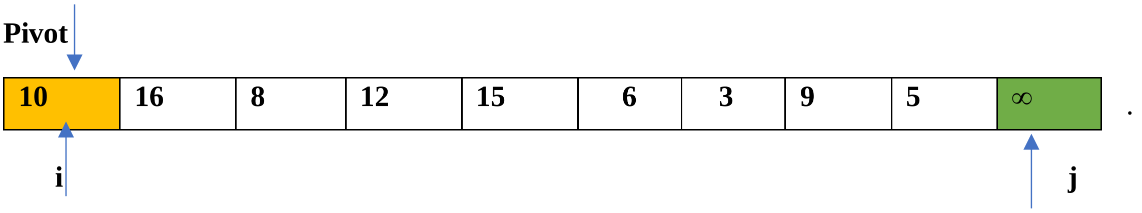
1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Example:

Step 1:

Pivot=10; i= First Element; j=Last Element;

Since Pivot=First Element; increment i and decrement j.



10	**16**	**8**	**12**	**15**	**6**	**3**	**9**	**5**	∞

Step 2:

i=16 & j=5; Check i(16)>Pivot (10) ;True ;

Check j(5) <Pivot(10); True;

Swap i&j

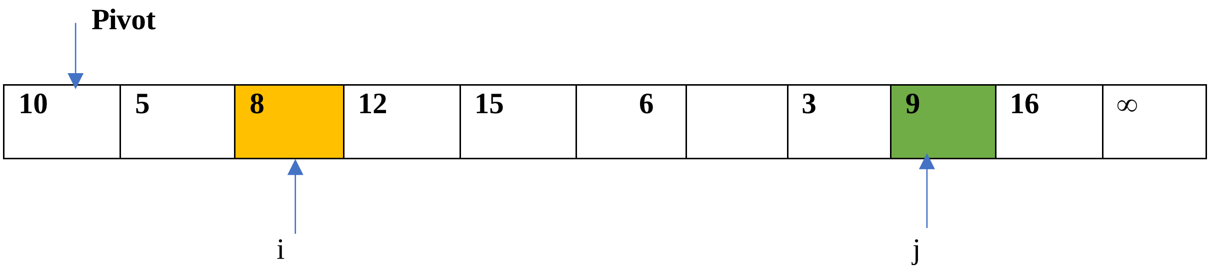
Increment i & decrement j;

Step 3:

i=8 & j=9; Check i(8)>Pivot (10) ;False;

No Swap;

Increment i;



10	**5**	**8**	**12**	**15**	**6**		**3**	**9**	**16**	∞

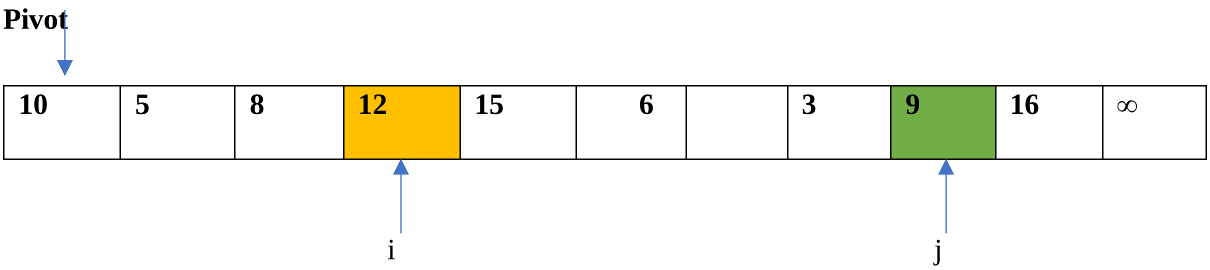
Step 4:

i=12& j=9; Check i(12)>Pivot(10) ;True ;

Check j(9)<Pivot(10); True;

Swap i&j

Increment i & decrement j;



10	**5**	**8**	**12**	**15**	**6**		**3**	**9**	**16**	∞

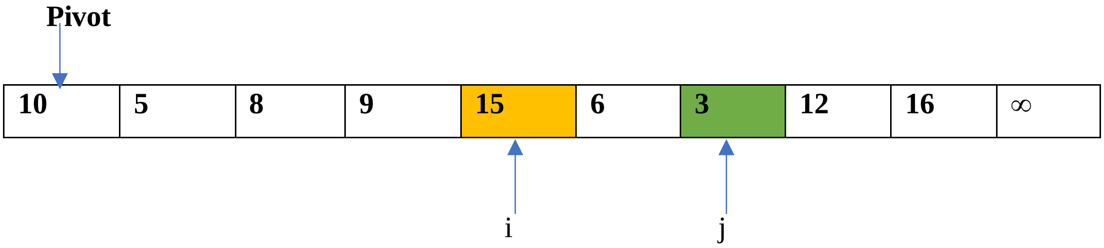
Step 5:

i=15& j=3; Check i(15)>Pivot(10) ;True ;

Check j(3)<Pivot(10); True;

Swap i&j

Increment i & decrement j;



10	**5**	**8**	**9**	**15**	**6**	**3**	**12**	**16**	∞

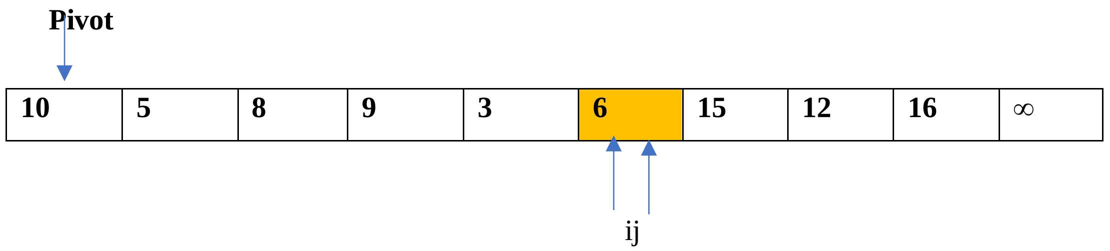
Step 5:

i=6& j=6;

Check i(6)>Pivot(10) ;False ;

Check j(6)<Pivot(10); True; j Crossed i.

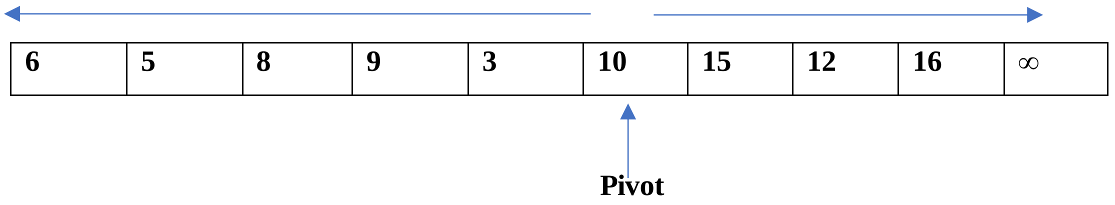
So Swap Pivot & j;



10	**5**	**8**	**9**	**3**	**6**	**15**	**12**	**16**	∞

Final List:

Elements left to Pivot are smaller **Elements right to pivot are greater**



6	**5**	**8**	**9**	**3**	**10**	**15**	**12**	**16**	∞

Recursively Apply Quick Sort:

We now apply quick sort recursively to the two subarrays:

- **Left subarray: [6,5,8,9,3]**
- **Right subarray: [15,12,16]**

Final Sorted Array:

Applying recursive calls, the array is fully sorted:

3	**5**	**6**	**8**	**9**	**10**	**12**	**15**	**16**	∞

Algorithm:

i).Algorithm Quicksort(A[0..n-1],low,high)

```
{
if(low<high)
Pivot=Partition(A[low..high])//Pivot is in mid position now
Quicksort(A[low..Pivot-1]
```

```
    Quicksort(A[Pivot +1…high])
}
```

ii).Algorithm Partition(A[low..high])

```
{
    Pivot=A[low]
    i=low
    j=high
    while(i<=j)do
    {
        while(A[i]<=Pivot)do
            i=i+1
        while(A[j]<=Pivot)do
            j=j-1
        if(i<=j)
            swap(A[i],A[j])
    }
    swap(A[low],A[j])  //When j crosses i, swap A[low] and A[j]
    return j
}
```

Analysis:

Time Complexity T(n) :

Best Case & Average Case Time Complexity: O (n log n)

- In the best case, Quick Sort performs well when the pivot divides the array into two equal parts in every step of the recursion. This ensures a balanced partitioning.
- Partitioning process takes linear time (O(n)), and there are approximately **log n** levels of recursion. So, the time complexity is **O (n log n)**.

Worst Case Time Complexity: $O(n^2)$

- This can happen if the pivot chosen is always the smallest or largest element.
- T(n)=n+(n-1) +(n-2) +…. +2+1

  By Sum of ‘n’ natural nos,We get

  $=(n(n+1))/2$

  $=(n^2+n)/2$

  Order=Higher degree of Polynomial= **O(n²)**.

By Recurrence Relation:

Time to Sort Right Sub Array

$$\boxed{T(n)= T(n/2) + T(n/2) + n} \rightarrow \textbf{Time to Partition the array}$$

Time to Sort Left Sub Array

Step 2: Consider the above Recurrence Relation;

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

$T(n)=2* T(n/2) + n$ ——— **1**

Applying Equation 1 in general recurrence relation: $T(n)=a* T(n/b) + f(n)$

Now, a=2; b=2;

f(n)=n= θ(n^d)= θ(n^1)

Therefore d=1;

Finding a=(b^d):

2=(2^1)

2=2(Both are equal)

Step 3: It can be Solved using Master Theorem or Substitution Method. Here we use Master Theorem.

If f(n) ∈ θ (n^d), then,

1.T(n)= θ (n^d), if(a<(b^d))

2. T(n)= θ((n^d) * log n), if(a=(b^d))

3. T(n)= θ ((n^ logb(a)), if(a>(b^d))

Since a=(b^d); Apply in case 2; We get,

T(n)= θ((n^d) * log n) [**since d=1]**

= θ((n^1) * log n) = θ ((n log n)

Space Complexity:

Best and Average Case

- In the best and average cases, where the array is divided into two relatively equal parts, the recursion depth will be **log n** (for the pivot and the partitioning process), so the space complexity is **O (log n)**.

Worst Case:

- In the worst case, where the array is highly unbalanced (for example, if the pivot is always the smallest or largest element), resulting in a space complexity of **O(n)**.

Additional Information

Choice of Pivot

There are many different choices for picking pivots.

- Always pick the first (or last) element as a pivot. The problem with this approach is it ends up in the worst case when array is already sorted.
- Pick a random element as a pivot. This is a preferred approach because it does not have a pattern for which the worst case happens.
- Pick the median element is pivot. This is an ideal approach in terms of time complexity as we can find median in linear time and the partition function will always divide the input array into two halves.

Partition Algorithm

- The key process in **quickSort** is a **partition ().** There are three common algorithms to partition. All these algorithms have O(n) time complexity.

  1.Naive Partition: Here we create copy of the array. First put all smaller elements and then all greater. Finally, we copy the temporary array back to original array. This requires O(n) extra space.

  2.Lomuto Partition: This is a simple algorithm; we keep track of index of smaller elements and keep swapping.

3.Hoare's Partition: This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned.

Topic 5: Merge Sort

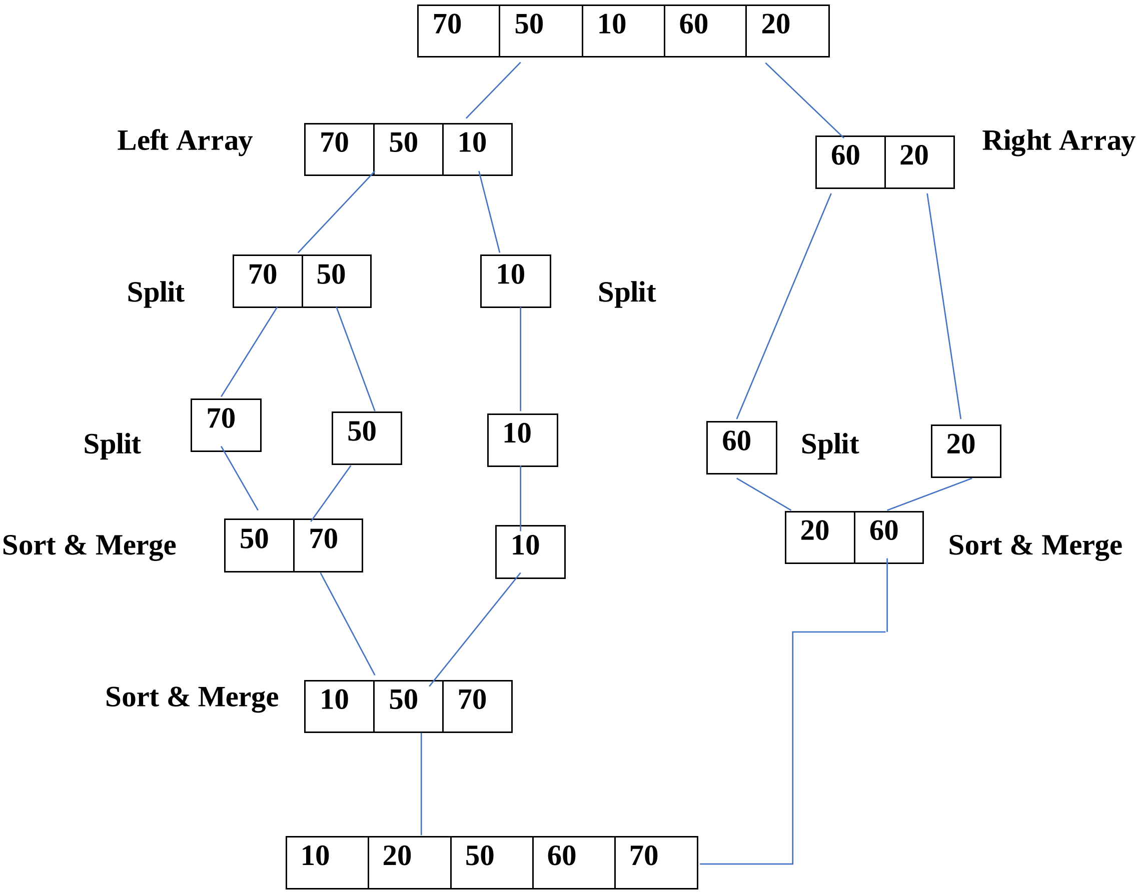
(using Divide & Conquer Approach)

Merge Sort works by recursively splitting the array into two halves, sorting each half, and then merging the sorted halves back together.

Steps:

1. **Split the Array**: Keep dividing the array into halves until each subarray has only one element.
2. **Merge**: Start merging the subarrays back together in sorted order.

Example: Initial Array: [70, 50, 10, 60, 20]; Find Mid Element=10;



Step 1: Split into Left & Right Sub Array: Left: [70, 50, 10] & Right: [60, 20] until array is splitted into individual element.

Step 2: Sort &Merge: all individual elements is sorted and merged till entire element is Sorted
Final Sorted Array: [10, 20, 50, 60, 70]

Algorithm:

i). Algorithm Mergesort(A[0…n-1],low,high)

```
{
    if(low<high)
    {
        mid=(low+high)/2
        Mergesort(A,low,mid)
        Mergesort(A, mid+1,high)
        Combine(A,low,mid,high)
    }
}
```

ii). Algorithm Combine (A [0…n-1],low,mid,high)

```
{
    k=low
    i=low
    j=mid+1
    while(i<=mid)
    {
        if(A[i]<=A[j])
        {
            Temp[k]=A[i]
            i++
            k++
        }
    } Temp[k]=A[j]
    while(j<=high)
    {
        if(A[j]>=A[i])
        {
            Temp[k]=A[j]
            j++
```

```
k++
}
} Temp[k]=A[j]
}
```

Analysis:

Time Complexity:

1.Best Case:

- Even in the best case (already sorted input), Merge Sort still divides the array and merges it back in O (n log n) time.

2.Average Case:

- On average, Merge Sort divides the array into two halves and merges them back, resulting in O (n log n) time complexity.

3.Worst Case:

- Even in the worst case (when the input is reversed), Merge Sort divides the array and merges them back in O (n log n) time.

By Recurrence Relation:

Step1: Find Recurrence Relation:

Time to Sort Right Sub Array

$$\boxed{\mathbf{T(n)= T(n/2) + T(n/2) + n}} \longrightarrow \textbf{Time to Partition the array}$$

Time to Sort Left Sub Array

Step 2: Consider the above Recurrence Relation;

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

$T(n)=2* T(n/2) + n \quad \longrightarrow \quad \mathbf{1}$

Applying Equation 1 in general recurrence relation: $T(n)=a* T(n/b) + f(n)$

Now, $a=2$; $b=2$;

$f(n)=n= \theta(n^d) = \theta(n^1)$

Therefore $d=1$;

Finding a=(b^d):

2= (2^1)

2=2(Both are equal)

Step 3: It can be Solved using Master Theorem or Substitution Method. Here we use Master Theorem.

Since a=(b^d); **Apply Case 2 in Master Theorem**; We get,

T(n)= θ((n^d) * log n) [**since d=1]**

= θ((n^1) * log n) = θ ((n log n)

Therefore T(n)= θ ((n log n)

Space Complexity:

- **O(n)** because of the additional space required to store the temporary subarrays during the merging process.

Topic 6: Binary Search

(using Divide & Conquer Approach)

- Binary search is an efficient algorithm for finding an item from a list or array. Elements in array **must be sorted** before searching is done.
- Elements to be searched is 'Key'. Low is First element. High is Last element.
- Find Mid Element= (low + high)/2. Key Element and Mid Element are Compared.
- If Key element is less than Mid Element, Search is done in Left Sublist and If Key element is greater than Mid Element, Search is done in Right Sublist. The Process is repeated until Key element is Matched.

Example:

low			mid			high
10	**20**	**30**	**40**	**50**	**60**	**70**

Key Element=60;

Step 1:

Find Middle Element:

mid= (low +high)/2

= (10+70)/2

=80/2=40; m**id=40**

Step 2:

Check A[mid]=Key; 40<60; Key is not mid element;

Step 3:

Key is Greater than mid. So, Search elements after mid (Right Sublist).

50	**60**	**70**
low	mid	high

Step 4:

Compare [mid]=Key; Key (60) is matched with mid.

Algorithm :

Algorithm BinSearch(A[0..n-1], Key)

```
{
    low=0
    high=n-1
    while(low<high)
    {
        mid=(low+high)/2
        if(key==A[mid])
            return mid
        else
        if(key<A[mid])
            BinSearch(A[], Key,low,mid-1)//Search Left Sublist
        Else
            BinSearch(A[], Key,mid+1,high)//Search Right Sublist
    }
}
```

Analysis:

Time Complexity T(n):

1.Best Case:

- O(1); When Middle Element is key.

2.Average Case and Worst Case:

- O (log n) ; When Key Element is present at one of the end or not present.

By Recurrence Relation:

Step1: Find Recurrence Relation:

$$\boxed{T(n)= \underbrace{T(n/2)}_{\text{Time to divide the sublist}} + \overbrace{O(1)}^{\text{Constant Time to Compare element}}}$$

Binary Search divide the problem into two halves and process only one half in each step.

Step 2: Apply Master Theorem:

T(n)= T(n/2) + O(1)

T(n)=1* T(n/2) +O(n^0) **[since (n^0)=1]** $\longrightarrow$ 1

Applying Equation 1 in general recurrence relation: T(n)=a* T(n/b) + f(n)

Now, a=1; b=2;

f(n)=(n^0)=(n^d)

[therefore d=0];

Finding a=(b^d);

1=(2^0) (since 2^0=1)

1=1;(Both are equal)

Step 3: It can be Solved using Master Theorem or Substitution Method. Here we use Master Theorem.

Since a=(b^d); **Apply Case 2 in Master Theorem**; We get,
T(n)= θ((n^0) * log n) [**since d=0]**
= θ((1) * log n) = θ (log n)
Therefore T(n)= θ (log n)

Space Complexity S(n):

O(1) — Binary search is an in-place algorithm that does not require additional memory beyond the input array. So It is considered Constant.

Quick Review

Analysis of Brute Force Algorithms				
Algorithm	**Time Complexity**			**Space Complexity**
	Best Case	**Average Case**	**Worst Case**	
Selection Sort	**O(n^2)**	**O(n^2)**	**O(n^2)**	**O(1)**
Bubble Sort	**O(n)**	**O(n^2)**	**O(n^2)**	**O(1)**
Linear Search	**O(1)**	**O(n)**	**O(n)**	**O(1)**

Analysis of Divide & Conquer Algorithms				
Algorithm	**Time Complexity**			**Space Complexity**
	Best Case	**Average Case**	**Worst Case**	
Merge Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**	**O(n)**
Quick Sort	**O(n log n)**	**O(n log n)**	**O(n^2)**	**O(log n)**
Binary Search	**O(log n)**	**O(log n)**	**O(log n)**	**O(1)**