CHAPTER TWO

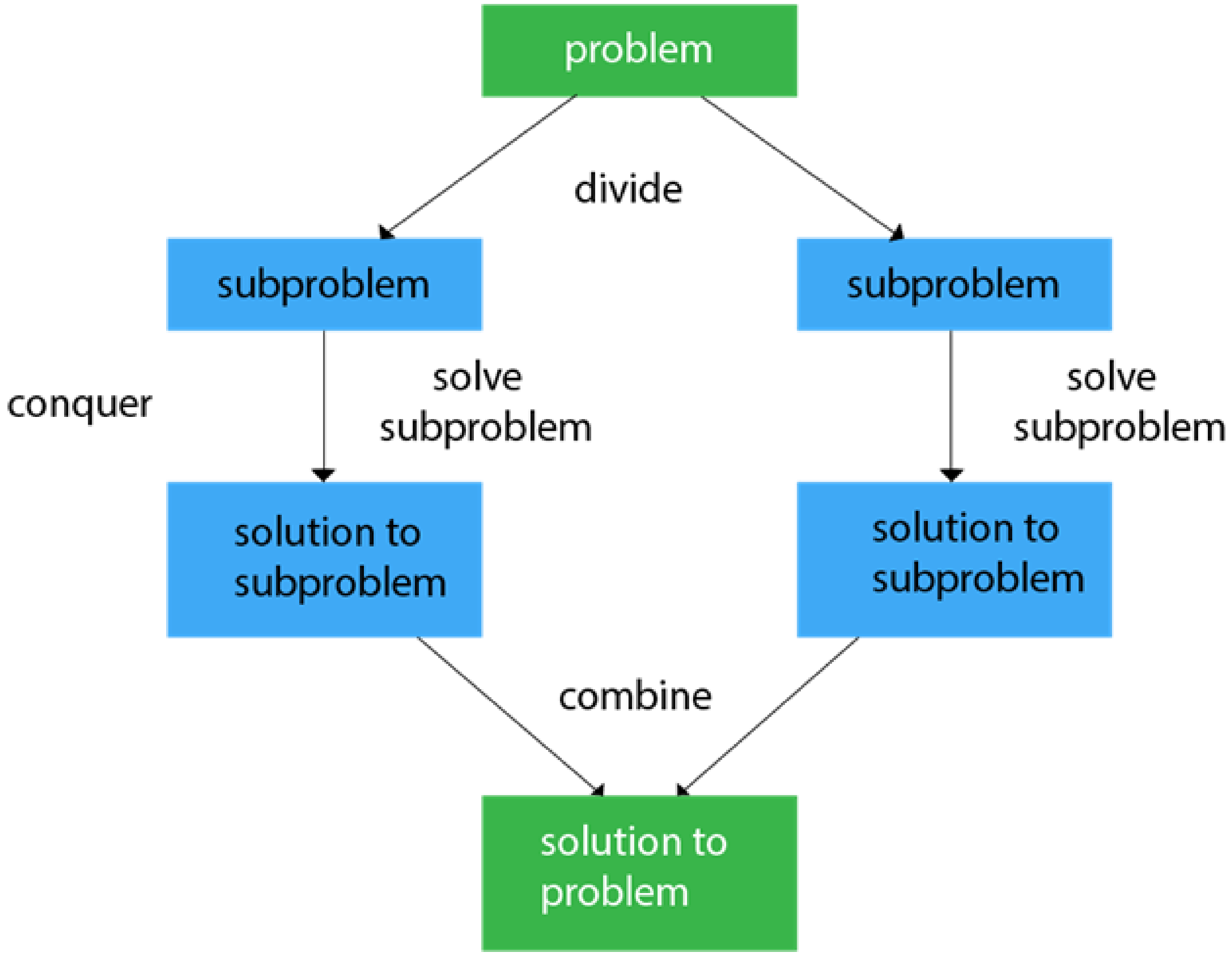
Divide and Conquer Algorithms

Divide and Conquer Algorithms

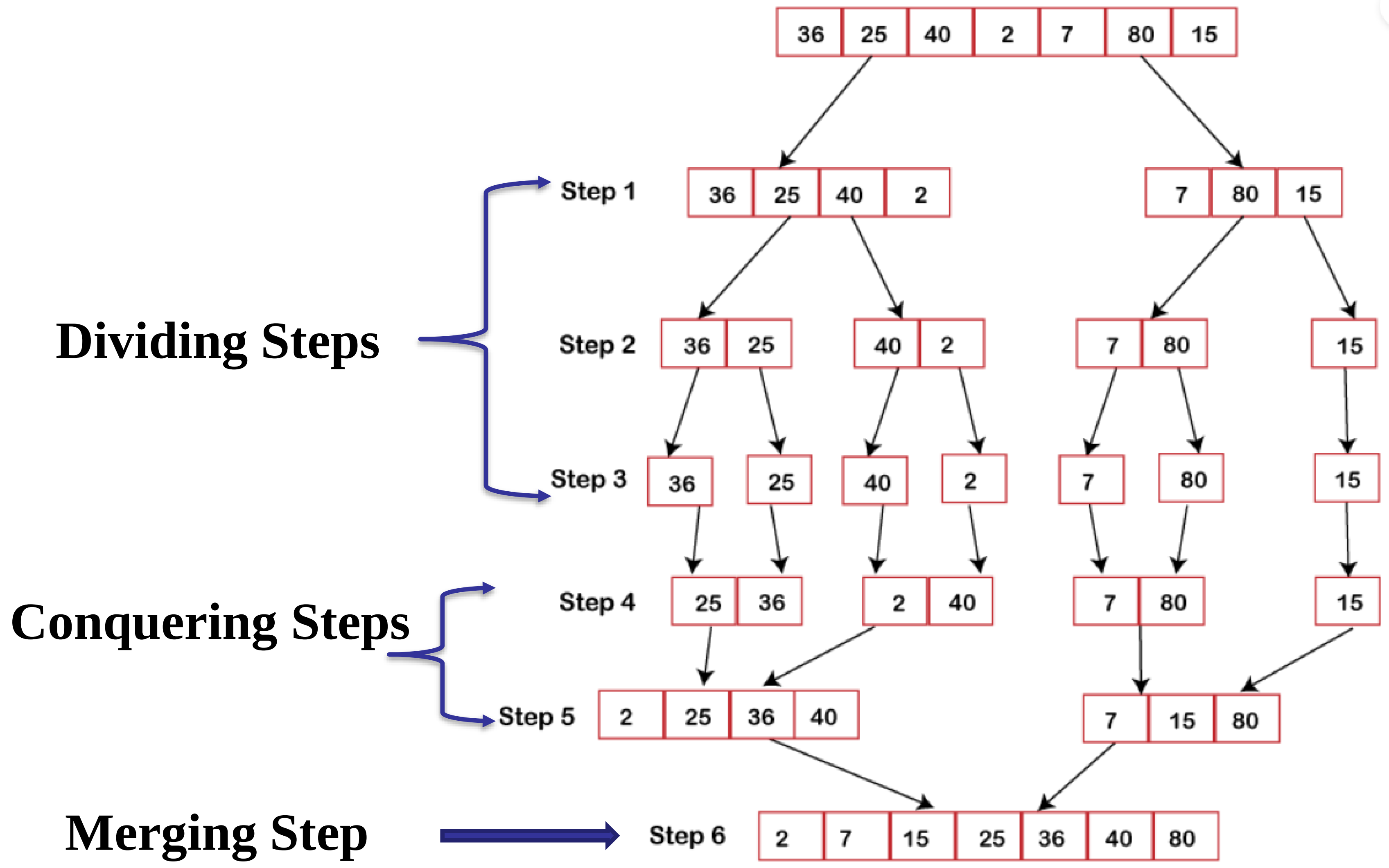
- **Divide and Conquer is an algorithmic paradigm in which the problem is solved using the Divide, Conquer, and Combine strategy.**
  - **i.e., a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.**

- **Divide and Conquer algorithm solves a problem using following three steps:**
  - **Divide:** **This involves dividing the problem into smaller sub-problems.**
  - **Conquer:** **Solve sub-problems by calling recursively until solved.**
  - **Combine:** **Combine the sub-problems to get the final solution of the whole problem.**

Divide and Conquer Algorithms



Divide and Conquer: Example-Merge Sort



Divide and Conquer: Pros and cons

- **Pros:**
  - It supports parallelism as sub-problems are independent.
    - It is suitable for multiprocessing systems.
  - It efficiently uses cache memory
    - It solves simple subproblems within the cache memory.

- **Cons:**
  - Memory management is very high.
    - Because, most of the algorithms are designed using recursion.
  - An explicit stack may overuse the space.
    - It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

Divide and Conquer: Application

- **Following are some problems, which are solved using divide and conquer approach.**
  - **Binary Search**
  - **Max-Min Problem**
  - **Merge Sort**
  - **Quick Sort**

Binary Search: Definition

- **Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.**
- **The item to be searched is compared with the middle element of the list.**
  - **If the match is found then, the location of the middle element is returned.**
  - **Otherwise, we search into either of the halves depending upon the result produced through the match.**

Binary Search: Problem

- In this approach, the index of an element x is determined if the element belongs to the list of elements.
- Therefore, for the given a sorted array arr[] of n elements, write a function to search a given element x in arr[] and return the index of x in the array.
- For example:
  - Input: arr[] = {10, 20, 30, 50, 60, 80, 110, 130, 140, 170}, x = 110
  - Output: 6 (Element x is present at index 6)
  - Input: arr[] = {10, 20, 30, 40, 60, 110, 120, 130, 170}, x = 175
  - Output: -1 (Element x is not present in arr[])

Binary Search: Solution

- Binary Search Algorithm can be implemented in two ways: Iterative Method and Recursive Method.
- The basic steps to perform Binary Search for both methods are:
  1. Firstly, we take the whole array as an interval.
  2. Set the low index to the first element and the high index to the last element of the array.
  3. Set the middle index to the average of the low and high indices.
  4. If the element at the middle index is the item to be searched, return the middle index.
  5. If the item to be searched is less than the item in the middle of the interval, we discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element (mid - 1).
  6. If the item to be searched is greater than the item in the middle of the interval, we discard the first half of the list and work recursively on the second half by calculating the new beginning (mid + 1) and middle element.
  7. Repeat steps 3-6 until the value is found or interval is empty.

Binary Search: Algorithm

For Iterative Method:

```
binarySearch(arr, x, low, high)
    repeat till low = high
        mid = (low + high)/2
          if (x == arr[mid])       // x is on the middle of list
          return mid
          else if (x > arr[mid])  // x is on the right side of list
            low = mid + 1
          else                    // x is on the left side of list
            high = mid - 1
```

Binary Search: Algorithm

For Recursive Method:

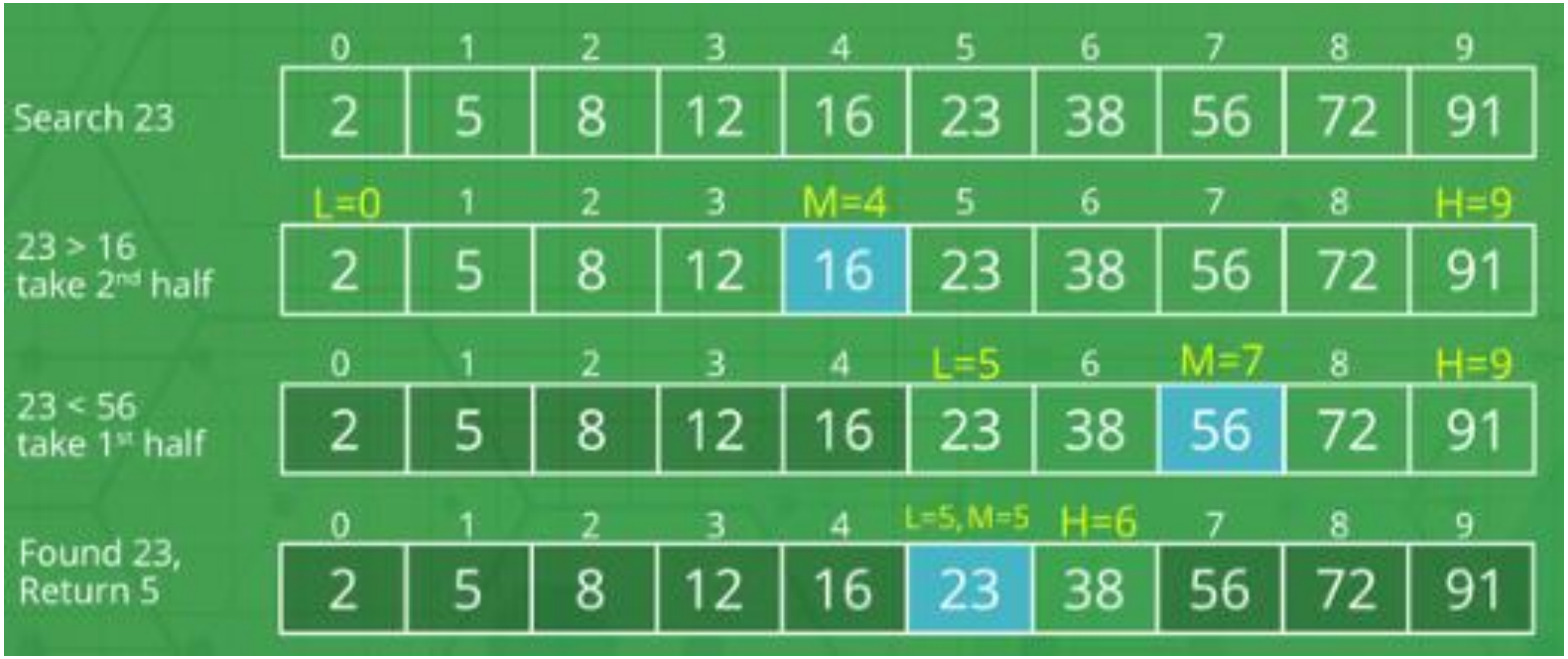
- The recursive method follows the divide and conquer approach.

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid] // x is on the middle of list
            return mid
        else if x > arr[mid]        // x is on the right side  of list
            return binarySearch(arr, x, mid + 1, high)
        else                                // x is on the left side of list
            return binarySearch(arr, x, low, mid - 1)
```

Binary Searching: Implementation

```
int binarySearch(int a[], int beg, int end, int val)   {
    int mid;
    if(end >= beg) {
        mid = (beg + end)/2;
/* if the item to be searched is present at middle */
        if(val == a[mid] ) {
            return mid;
        }
 /* if the item to be searched is greater than middle, then it can only be in right subarray */
else if(val > a[mid] ) {
            return binarySearch(a, mid+1, end, val);
        }
/* if the item to be searched is smaller than middle, then it can only be in left subarray */
    else {
            return binarySearch(a, beg, mid-1, val);
        }
    }
    return -1;
}
```

Binary Search: Example



Binary Search: Analysis

- **Input**:
  - an array A of size n, already sorted in the ascending or descending order.
- **Output**:
  - analyse to search an element item in the sorted array of size n.
- **Logic:**
  - Let T (n) = number of comparisons of an item with n elements in a sorted array.

- Set BEG = 1 and END = n

Binary Search: Analysis

- Find mid = $\text{int}\left(\frac{beg + end}{2}\right)$
- Compare the search item with the mid item.
- Case 1: item = A[mid], then LOC = mid, but it the best case and T (n) = 1
- Case 2: item ≠A [mid], then we will split the array into two equal parts of size $\frac{n}{2}$
- And again find the midpoint of the half-sorted array and compare with search element.
- Repeat the same process until a search element is found.

$T\left(\frac{n}{2}\right)+1$ ...... *(Equation 1)*

Binary Search: Analysis

Time to compare the search element with mid element, then with half of the selected half part of array

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: $T(n) = \left(T\left(\frac{n}{2^2}\right) + 1\right) + 1$..........By putting $T\,\frac{n}{2}$ in (1) equation

$T(n) = T\left(\frac{n}{2^2}\right) + 2$...................... (Equation 2)

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$.................... Putting $\frac{n}{2}$ in place of n in eq 1.

$T(n) = T\left(\frac{n}{2^3}\right) + 1 + 2$

$T(n) = T\left(\frac{n}{2^3}\right) + 3$................................ (Equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$.................. Putting $\frac{n}{3}$ in place of n in eq1

Put $T\left(\frac{n}{2^3}\right)$ in eq (3)

$T(n) = T\left(\frac{n}{2^4}\right) + 4$

Repeat the same process ith times

$T(n) = T\left(\frac{n}{2^i}\right) + i$ .....

Stopping Condition: T (1) =1

- **At least there will be only one term left that's why that term will compare out, and only one comparison be done that's why**

$$\begin{tikzcd} & T(1) = 1 \arrow[dl, dash] \arrow[dr, dash] & \\ \text{Item} & & \text{Comparison} \end{tikzcd}$$

Binary Search: Analysis

$\frac{n}{2^i} = 1$

$\frac{n}{2^i}$ Is the last term of the equation and it will be equal to 1

n= 2^i

After i iterations, the size of the array becomes 1 (narrowed down to the first element or last element only).

Applying log both sides

$\log n = \log_2 i$

Log n = i log 2

$\frac{\log n}{\log 2} = 1$

$\log_2 n = i$

$T(n) = T\left(\frac{n}{2^i}\right) + i$

$\frac{n}{2^i}$ = 1 as in eq 5

= T (1) + i

= 1 + i................................... T (1) =1 by stopping condition

= 1+ $\log_2 n$

= $\log_2 n$ .... (1 is a constant that's why ignore it)

Therefore, binary search is of order o ($\log_2 n$)

Binary Search: Analysis

- **Let T(n) be the number of comparisons in worst-case in an array of n elements. Hence,**

$$T(n) = \begin{cases} 0 & if\, n = 1 \\ T(\frac{n}{2}) + 1 & otherwise \end{cases}$$

- **Using this recurrence relation T(n)=logn**
- **Therefore, binary search uses O(logn) time which is faster than a Linear search (O(n)) and O(logn) space complexity.**

Max-Min Problem

- **Problem:**
  - **The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.**

- **Solution:**
  - **To find the maximum and minimum numbers in a given array arr[] of size n, the following algorithm can be used.**
    1. **Naïve method (general approach)**
    2. **Divide and conquer approach.**

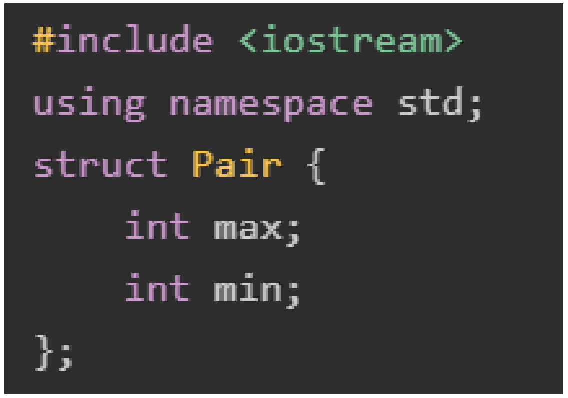
Max-Min Problem: Algorithm

Naïve Method

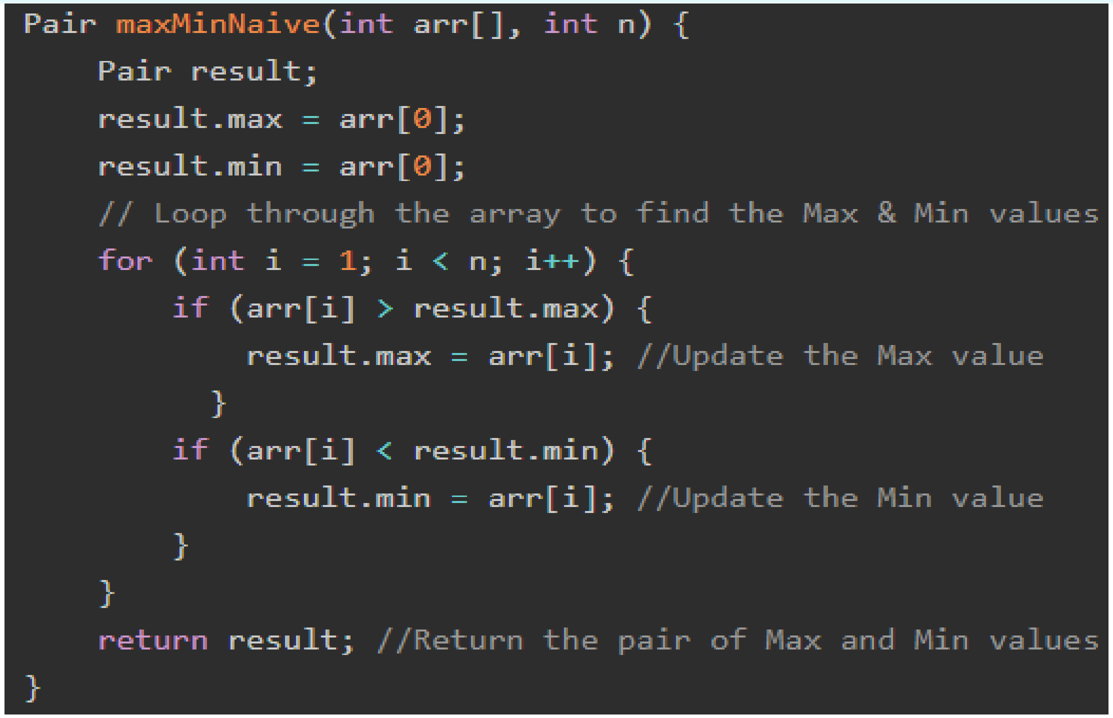
- Naïve method is a basic method to solve any problem.
- In this method, the maximum and minimum number can be found separately.

```
Max-Min-Element (arr[])
    max = arr[0]
    min = arr[0]
for i = 1 to n do
   if arr[i] > max then
      max = arr[i]
   if arr[i] < min then
      min = arr[i]
return (max, min)
```

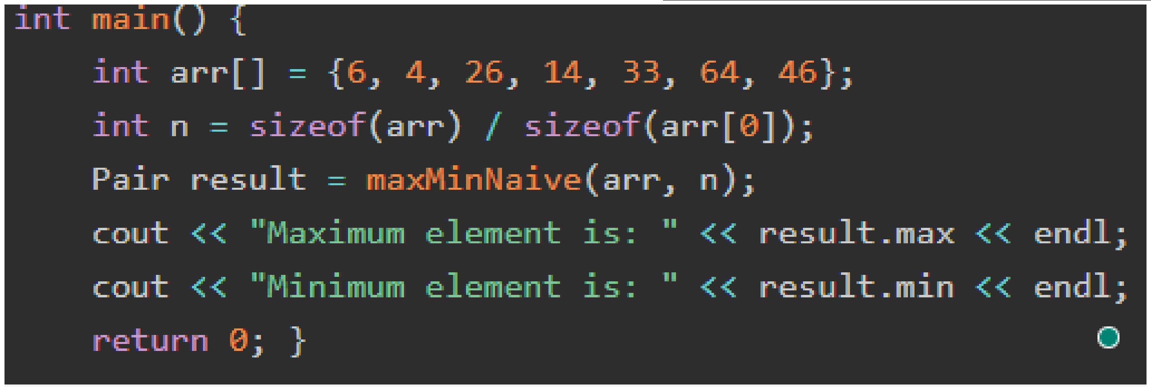
Max-Min Problem: Example



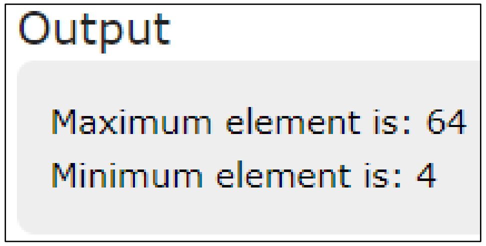
```cpp
#include <iostream>
using namespace std;
struct Pair {
    int max;
    int min;
};
```



```cpp
Pair maxMinNaive(int arr[], int n) {
    Pair result;
    result.max = arr[0];
    result.min = arr[0];
    // Loop through the array to find the Max & Min values
    for (int i = 1; i < n; i++) {
        if (arr[i] > result.max) {
            result.max = arr[i]; //Update the Max value
          }
        if (arr[i] < result.min) {
            result.min = arr[i]; //Update the Min value
        }
    }
    return result; //Return the pair of Max and Min values
}
```



```cpp
int main() {
    int arr[] = {6, 4, 26, 14, 33, 64, 46};
    int n = sizeof(arr) / sizeof(arr[0]);
    Pair result = maxMinNaive(arr, n);
    cout << "Maximum element is: " << result.max << endl;
    cout << "Minimum element is: " << result.min << endl;
    return 0; }
```



Output

```
Maximum element is: 64
Minimum element is: 4
```

Max-Min Problem: Algorithm

Divide and Conquer Approach

- **In this approach, the array is divided into two halves.**
- **Then using recursive approach maximum and minimum numbers in each halves are found.**
- **Later, return the maximum of two maxima of each half and the minimum of two minima of each half.**
- **In this given problem, the number of elements in an array is y - x + 1 (i.e., n = y - x + 1), where y is greater than or equal to x.**

Max-Min Problem: Algorithm

Divide and Conquer Approach

- **Divide:** **Divide the array into two halves.**
- **Conquer: Recursively find maximum and minimum of both halves.**
- **Combine: Compare maximum of both halves to get overall maximum and compare minimum of both halves to get overall minimum.**

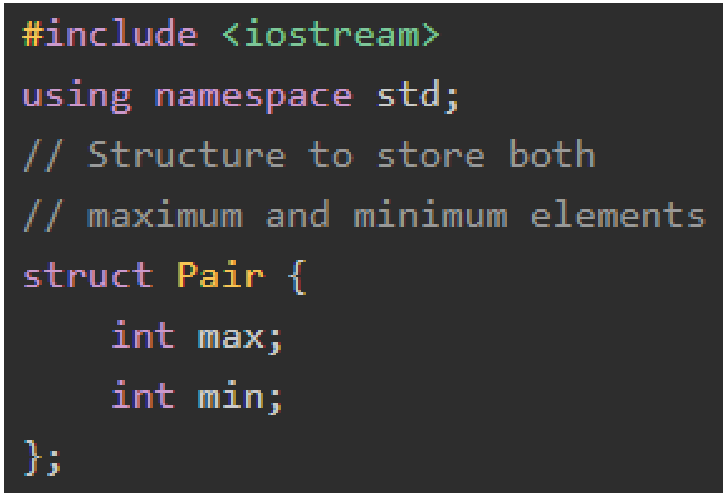
Max-Min Problem: Algorithm

Divide and Conquer Approach

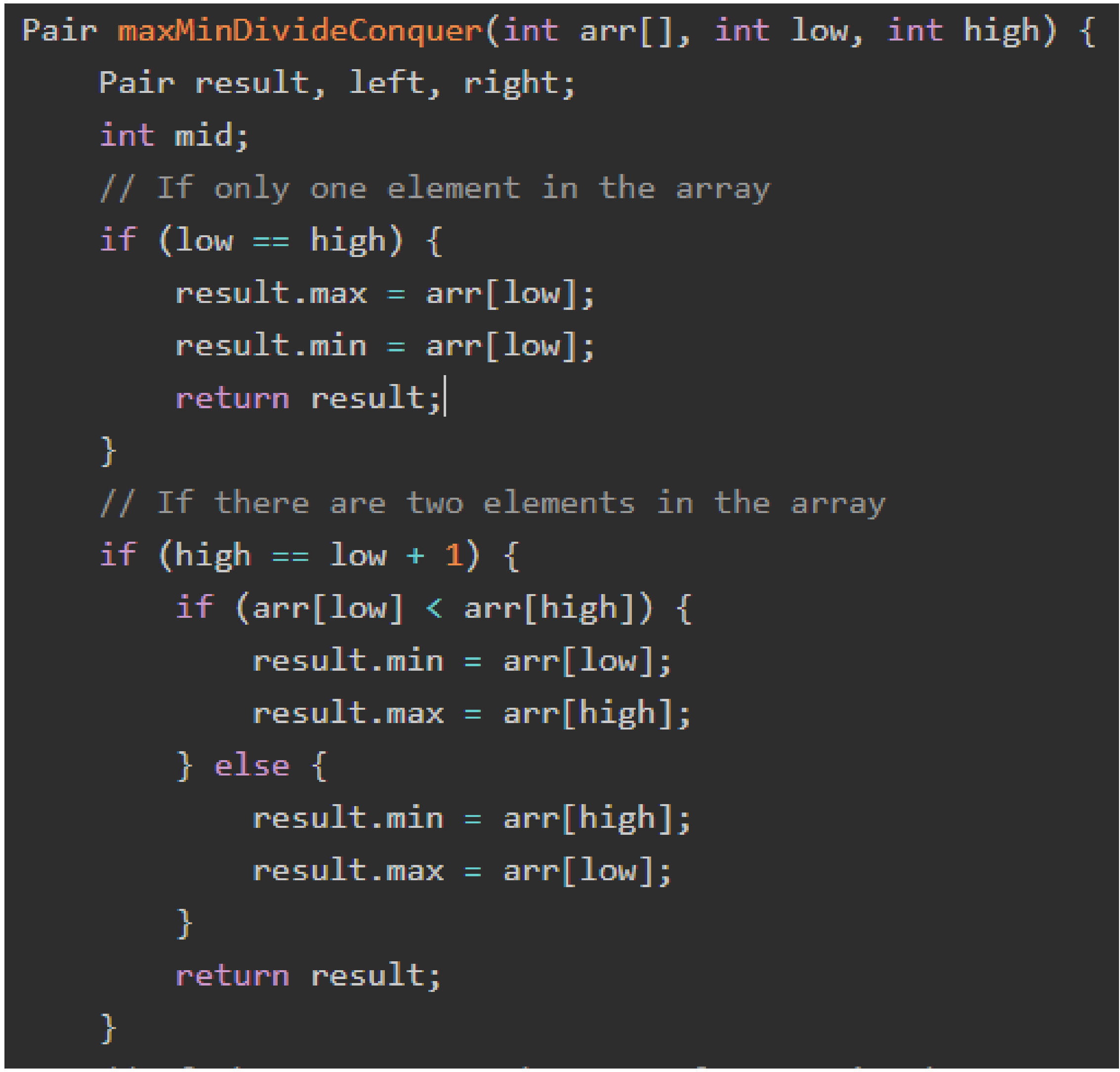
```
Max - Min(x, y)
if $y - x \le 1$ then
   return (max(numbers[x], numbers[y]), min((numbers[x],  numbers[y]))
else
   (max1, min1) = maxmin(x, $\lfloor((x + y)/2)\rfloor$)
   (max2, min2) = maxmin($\lfloor((x + y)/2) + 1)\rfloor$, y)

return (max(max1, max2), min(min1, min2))
```

Max-Min Problem: Example

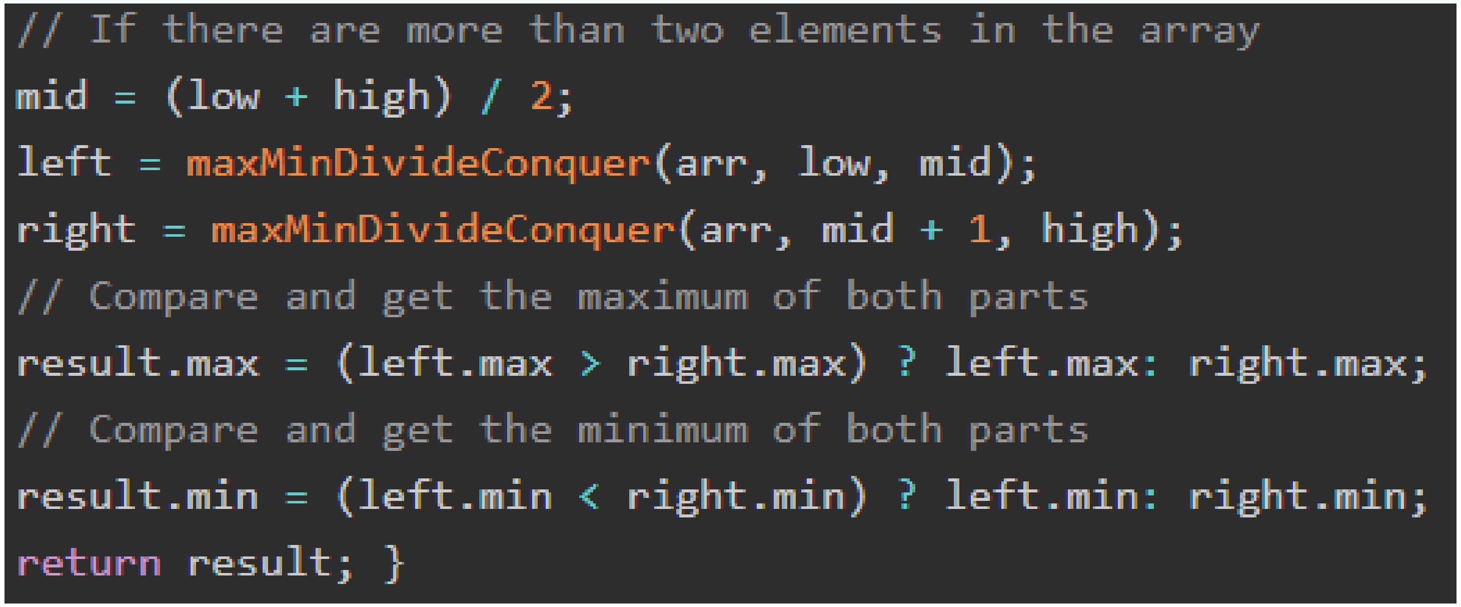


```cpp
#include <iostream>
using namespace std;
// Structure to store both
// maximum and minimum elements
struct Pair {
    int max;
    int min;
};
```

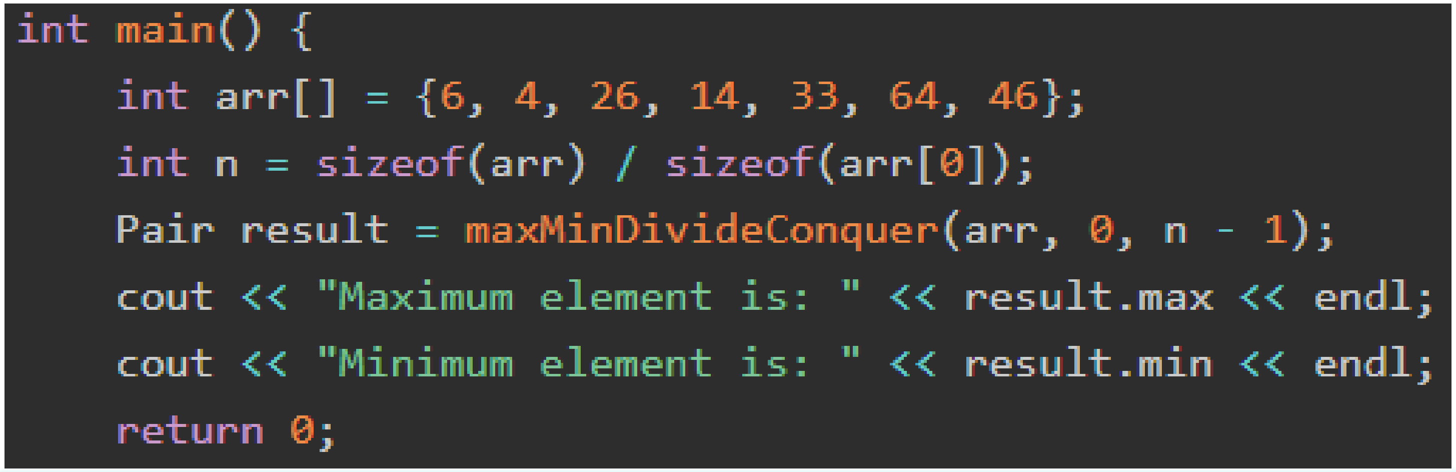


```cpp
Pair maxMinDivideConquer(int arr[], int low, int high) {
    Pair result, left, right;
    int mid;
    // If only one element in the array
    if (low == high) {
        result.max = arr[low];
        result.min = arr[low];
        return result;
    }
    // If there are two elements in the array
    if (high == low + 1) {
        if (arr[low] < arr[high]) {
            result.min = arr[low];
            result.max = arr[high];
        } else {
            result.min = arr[high];
            result.max = arr[low];
        }
        return result;
    }
```

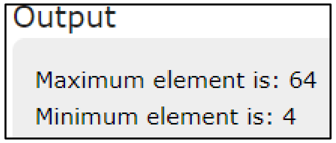
Max-Min Problem: Example



```cpp
// If there are more than two elements in the array
mid = (low + high) / 2;
left = maxMinDivideConquer(arr, low, mid);
right = maxMinDivideConquer(arr, mid + 1, high);
// Compare and get the maximum of both parts
result.max = (left.max > right.max) ? left.max: right.max;
// Compare and get the minimum of both parts
result.min = (left.min < right.min) ? left.min: right.min;
return result; }
```



```cpp
int main() {
    int arr[] = {6, 4, 26, 14, 33, 64, 46};
    int n = sizeof(arr) / sizeof(arr[0]);
    Pair result = maxMinDivideConquer(arr, 0, n - 1);
    cout << "Maximum element is: " << result.max << endl;
    cout << "Minimum element is: " << result.min << endl;
    return 0;
```



Output

```
Maximum element is: 64
Minimum element is: 4
```

Max-Min Problem: Analysis

- Method 1: if we apply the general approach to the array of size n, the number of comparisons required are: (n-1) + (n-1) = 2n-2.
- Therefore, time complexity for this approach is O(n).
- We are using constant extra space, so space complexity = O(1)

Max-Min Problem: Analysis

- **Method-2: In another approach, we will divide the problem into sub-problems and find the max and min of each group.**
  - **Of each group will compare with the only max of another group and min with min.**

  - **Let n = is the size of items in an array**
  - **Let T (n) = time required to apply the algorithm on an array of size n.**
    - **Here we divide the terms as T(n/2).**
    - **2 here tends to the comparison of the minimum with minimum and maximum with maximum.**

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2$$

Max-Min Problem: Analysis

- **If T(n) represents the numbers, then the recurrence relation can be represented as**

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\, n > 2 \\ 1 & for\, n = 2 \\ 0 & for\, n = 1 \end{cases}$$

- **Let us assume that *n* is in the form of power of 2. Hence, n = 2^k where k is height of the recursion tree.**

So,

$$T(n) = 2.T(\frac{n}{2}) + 2 = 2.\left(2.T(\frac{n}{4}) + 2\right) + 2 \ldots\ldots = \frac{3n}{2} - 2$$

Max-Min Problem: Analysis

- Compared to Naïve method, in divide and conquer approach, the number of comparisons is less.

Example:

- Analysis: suppose we have the array of size 8 elements.
- Method1: requires (2n-2), (2x8)-2=14 comparisons
- Method2: requires $\frac{3\times 8}{2}-2$ = 10 comparisons
- However, using the asymptotic notation the time complexity for both of the approaches is represented by O(n).
- Space complexity is equal to the size of the recursion call stack, which is equal to the height of the recursion tree i.e. O(logn)

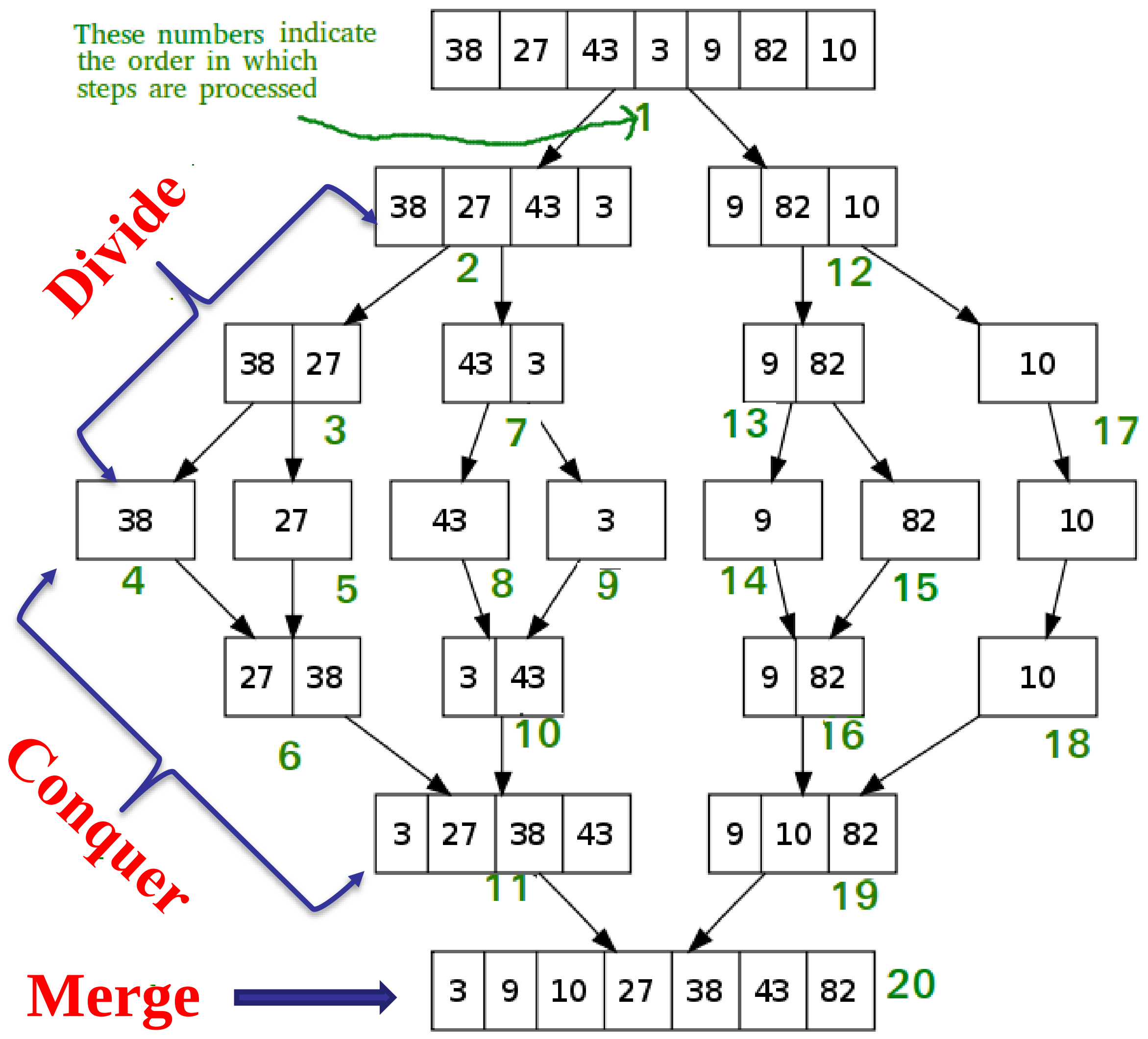
Merge Sort

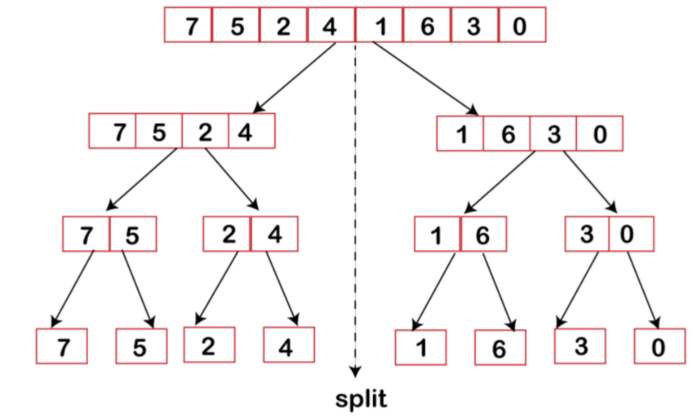
- Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
- Merge sort keeps on dividing the list into equal halves until it can no more be divided and then combines them in a sorted manner.

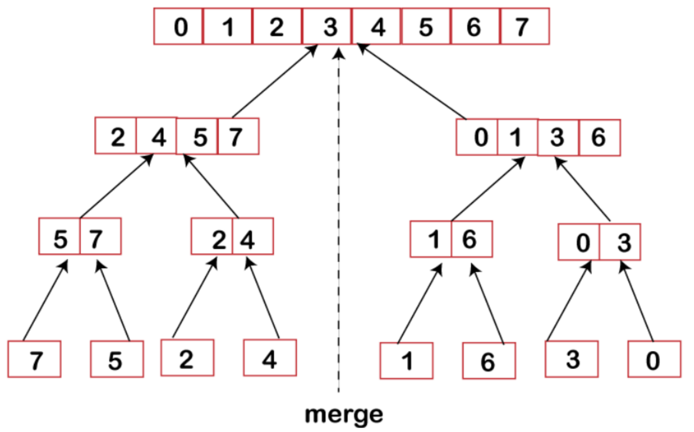
Marge Sort: Algorithm

```
step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
    if right > left
    mid= (left+right)/2 // Find the middle point to divide the array into two halves
    mergesort(array, left, mid) // Call mergeSort for first half
    mergesort(array, mid+1, right) // Call mergeSort for second half
    merge(array, left, mid, right)//Merge the two halves sorted in the previous steps.
step 4: Stop
```

Merge Sort: Example







Marge Sort: Analysis

- **Let us consider, the running time of Merge-Sort as T(n).**
- **Sorting two halves will take at the most $2T\frac{n}{2}$ Merge Sort time.**

$$\mathrm{T}(\mathrm{n}) = \begin{cases} c & if\, n \leq 1 \\ 2\,xT\left(\frac{n}{2}\right) + dxn & otherwise \end{cases} \quad where\; c\; and\; d\; are\; constants$$

- **Therefore, using this recurrence relation,**

$$T(n) = 2^i\,T\left(n/2^i\right) + i \cdot d \cdot n$$

$$As,\;\; i = log\,n,\; T(n) = 2^{log\,n} T\left(n/2^{log\,n}\right) + log\,n \cdot d \cdot n$$

$$= c \cdot n + d \cdot n \cdot log\,n$$

$$Therefore,\;\; T(n) = O(n\,log\,n).$$

The time complexity of Merge Sort is O(N*logN) 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.

Marge Sort: Advantage

- It has a time complexity of O(n*logn), which means it can sort large arrays relatively quickly.
- It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.
- It is a popular choice for sorting large datasets because it is relatively efficient and easy to implement.
- It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Quick Sort

- **Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.**
- **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.**
  - **Always pick the first element as a pivot.**
  - **Always pick the last element as a pivot**
  - **Pick a random element as a pivot.**
  - **Pick median as the pivot.**

Quick Sort

- A large array is partitioned into two arrays one of which holds values smaller than pivot and another array holds values greater than the pivot value.
- The left and right subarrays are also partitioned using the same approach and this process continues until each subarray contains a single element.
- Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

Quick Sort: Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex = partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex = leftmostIndex - 1
  for i = leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Quick Sort: Example

- **Given Array:** `44 33 11 55 77 90 40 60 99 22 88`
- **Let 44 (the left most element) be the Pivot element.**
- **Comparing 44 to the <u>right-side</u> elements, and if right-side elements are smaller than 44, then swap it. As 22 is smaller than 44 so swap them.**

22	33	11	55	77	90	40	60	99	**44**	88

- **Now comparing 44 to the <u>left side</u> element and the element must be greater than 44 then swap them. As 55 are greater than 44 so swap them.**

22	33	11	**44**	77	90	40	60	99	**55**	88

Quick Sort: Example

- **Recursively, repeating steps 1 and steps 2 until we get two lists one left from pivot element 44 and one right from pivot element.**

22	33	11	**40**	77	90	**44**	60	99	55	88

- **Swap with 77:**

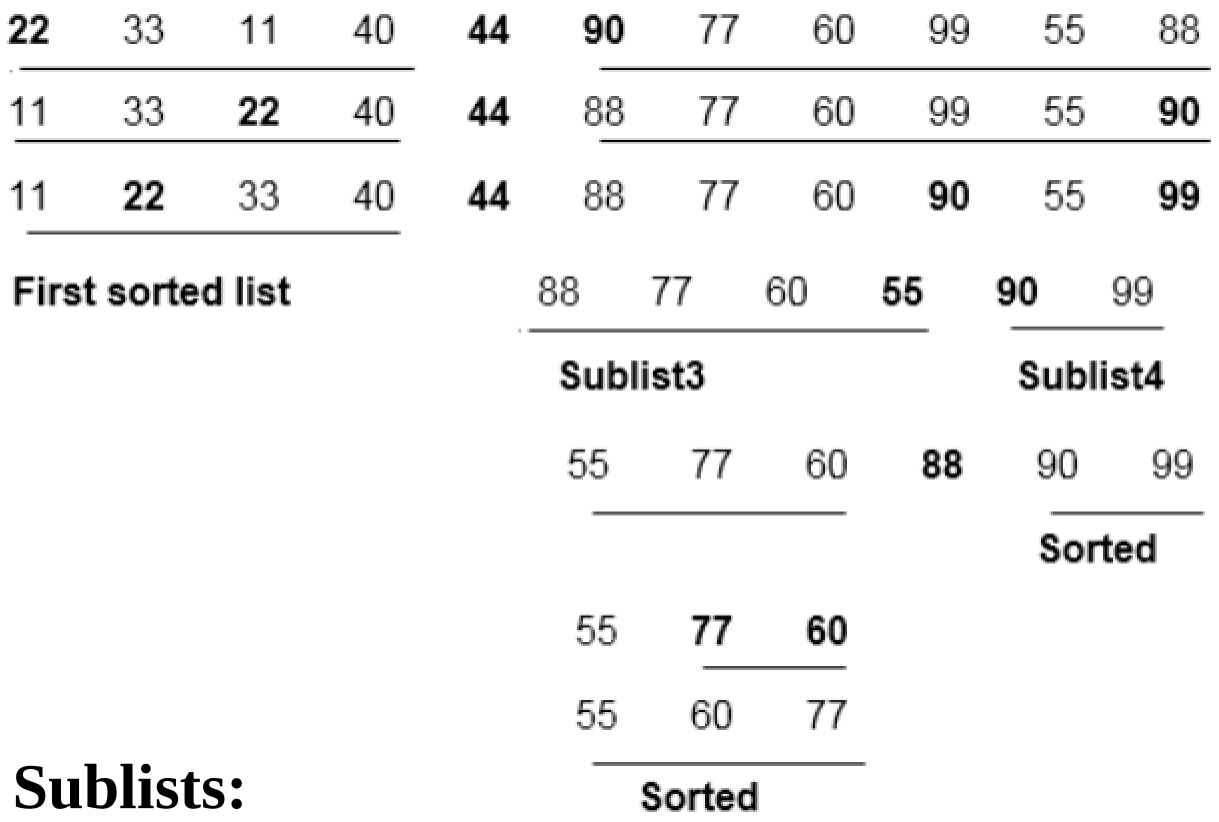
22	33	11	40	**44**	90	**77**	60	99	55	88

- **Now, the element on the right side and left side are greater than and smaller than 44 respectively.**
- **Now we get two sorted lists:**

$$\underbrace{22 \quad 33 \quad 11 \quad 40}_{\textbf{Sublist1}} \quad \mathbf{44} \quad \underbrace{90 \quad 77 \quad 66 \quad 99 \quad 55 \quad 88}_{\textbf{Sublist2}}$$

Quick Sort: Example

- And these sublists are sorted under the same process as above done.



```
22  33  11  40  44  90  77  60  99  55  88
11  33  22  40  44  88  77  60  99  55  90
11  22  33  40  44  88  77  60  90  55  99

First sorted list      88  77  60  55  90  99
                       Sublist3        Sublist4

                       55  77  60  88  90  99
                                       Sorted

                       55  77  60
                       55  60  77
                       Sorted
```

- Merging Sublists:

```
11  22  33  40  44  55  60  77  88  90  99
```

Quick Sort: Analysis

- Worst case complexity occurs when the pivot element picked is either the greatest or the smallest element.
- It is the case when items are already in sorted form and we try to sort them again.
- This will takes lots of time and space.

$$T(n) = T(1) + T(n-1) + n = O(n^2)$$

  - T (1) is time taken by pivot element.
  - T (n-1) is time taken by remaining element except for pivot element.
  - N: the number of comparisons required to identify the exact position of itself (every element)

Quick Sort: Analysis

- However, the quick sort algorithm has better performance for scattered pivots.
- Average Case Complexity occurs when the pivot element is always the middle element or near to the middle element.

$$T(n) = n + 1 + \frac{1}{n} = O(n*\log n)$$

- Quick Sort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data.
- Quick Sort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data.

Quick Sort: Analysis

- However, merge sort is generally considered better when data is huge and stored in external storage.
- Quick sort is preferred over merge sort for sorting Arrays.
  - Because quick sort is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage.
- Merge sort is also preferred over quick sort for linked lists.
  - Because merge sort can be implemented without extra space for linked lists.

Next:

Chapter 3:Greedy Algorithms