Algorithms Analysis and Design

Chapter 4

Divide and Conquer
Part 1

Introduction

- Brute force
- Divide and conquer 
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs

- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

It is well-known that there is no universal technique that can be the best-performing for all problems

Divide and Conquer Strategy

Divide and Conquer

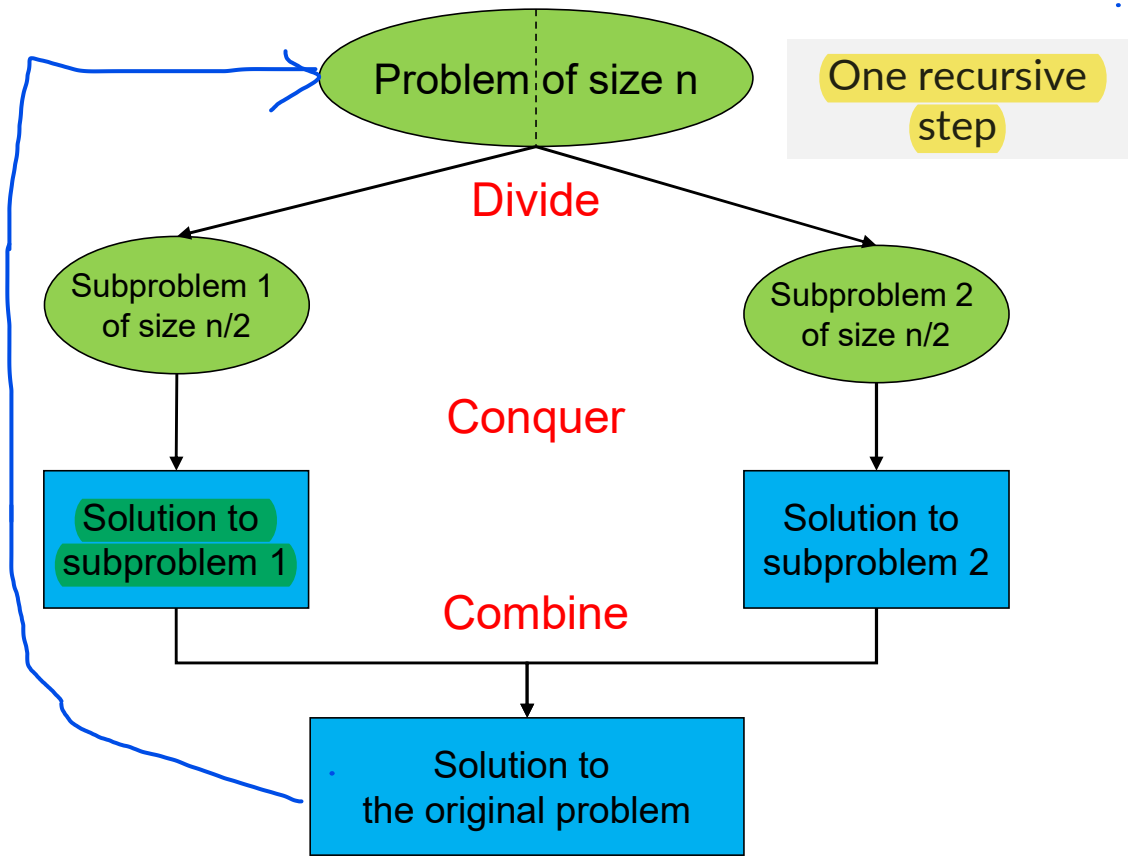
- Divide and Conquer is probably the best-known general algorithm design technique.
- ***Divide and conquer*** is a common algorithm design technique based on recursion. This paradigm, divide-and-conquer, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem

- **Divide-and-conquer** algorithms **work according to the following general plan**:
  - **Divide**: A problem is **divided** into several subproblems of **the same type**, ideally of **about equal size**.
  - **Conquer**: The subproblems are **solved recursively** (successively and independently)
  - **Combine**: If necessary, the solutions to the subproblems are **combined** to get a solution to the original problem.

Recursion itself lends to a general problem-solving strategy called divide and conquer

Divide and Conquer

- Divide-and-conquer technique (**typical case**).
- The case of dividing a problem into two smaller subproblems, by far the most widely occurring case (at least for divide-and-conquer algorithms designed to be executed on a single-processor computer).

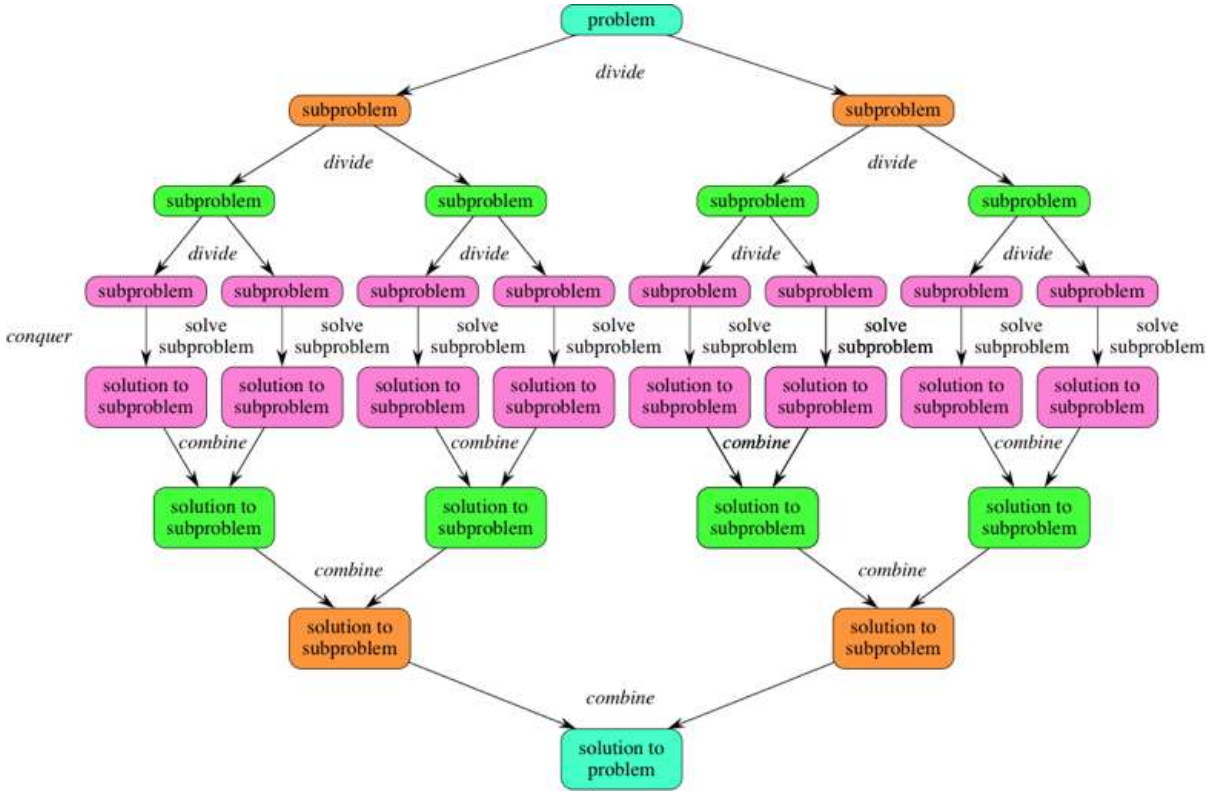


You can easily remember the steps of a divide-and-conquer algorithm as:

divide, conquer, combine

Divide and Conquer

- If we expand out two more recursive steps, it looks like this:



Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm **makes multiple recursive calls**

Apply the strategy recursively as long as the subproblem is large

General method for DAC

- A **divide and conquer** algorithm:

- If the problem is **small**, it is **solved directly**.
- If the problem is **large**, the problem is divided into two or more parts called sub-problems
- The divide-and-conquer algorithm is also used to solve each sub-problem → **Recursion**
- The solutions to the sub-problems are **combined** into a solution to the original problem

Because divide-and-conquer solves subproblems recursively, each subproblem must be **smaller than the original problem**, and **there must be a base case for subproblems (The problem instance that can be solved directly)**

General method for DAC

سيدوكود لـ DAC بشكل عام

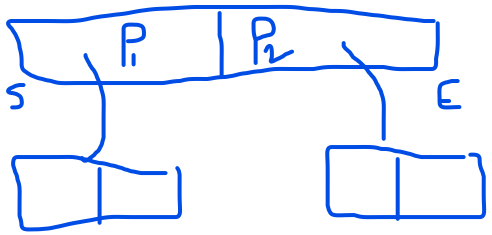
```
DAC ( Problem P )
{
        if (size of input is small enough) then
            solve directly
            return
        endif
        Divide P into two or more subproblems P1, P2, ….
        Call DAC (P1) , DAC (P2) …. to get subsolutions S1, S2 , ……
        Combine subsolutions S1, S2, ….
}
```

Base Case

Divide

Conquer

Combine



Example : Compute sum of n numbers

- let us consider the problem of computing the sum of n numbers $a_0, \ldots, an\_1$.
  - if $n = 1$, we simply return a_0 as the answer.
  - If $n > 1$, we can divide the problem into two instances of the same problem:
    - to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers.
  - Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}).$$

Is DAC an efficient way to compute the sum of n numbers?

Is it be more efficient than the brute-force summation?

Not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution

pseudocode for the divide and conquer algorithm for finding the sum of n numbers

```
ALGORITHM divide_and_conquer_sum(arr, start, end)
    if start == end:
        return arr[start]
    else:
        mid = (start + end) / 2
        left_sum = divide_and_conquer_sum(arr, start, mid)
        right_sum = divide_and_conquer_sum(arr, mid + 1, end)
        return left_sum + right_sum
```

Let's Discuss

Not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution

It is worth keeping in mind that the divide-and-conquer technique is ideally **suited for parallel computations**, in which each subproblem can be solved simultaneously by its own processor

Divide-and-Conquer Examples

- **Sorting**:
  - merge sort — $n \log n$
  - quicksort
- **Mathematics**
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Exponentiation
- **Computational geometry**
  - Closest-pair
  - convex-hull algorithms
- **Searching**:
  - Binary search: decrease-by-half (or degenerate divide&conq.)
- **Binary tree traversals**

- In fact, the divide-and-conquer approach yields some of the most important and efficient algorithms in computer science.
- We will discuss a few classic examples of such algorithms in this chapter (only sequential algorithms).

General Divide and Conquer Recurrence

In the most **typical case** of divide-and-conquer a problem's instance of size n is divided into two instances of size $n/2$

More generally,
An instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.)

We get the following recurrence for the running time $T(n)$:

$$\mathbf{T}(n) = \begin{cases} \theta(1), & n: small \\ a\,T\left(\frac{n}{b}\right) + D(n) + C(n), & n: large \end{cases}$$

f(n)

master method

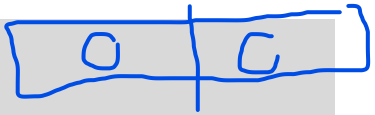
General Divide and Conquer Recurrence

General divide and conquer recurrence:

$$T(n) = a\,T\left(\frac{n}{b}\right) + D(n) + C(n), \qquad n: large$$

- n : Size of input
- a: Number of subproblems need to be solved.
- b: Number of subproblems
- n/b: Size of each subproblem (All subproblems are assumed to have the same size)
- $D(n)$: The cost of dividing the problem an instance of size n into instances of size n/b.
- $C(n)$: The cost of combining the solutions.

مجموع n من العناصر



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

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

بدون كتابة كود وبطبقها على ال mastermethod

$O(n)$

من حيث $T(n)$ ما استفدت ممكن اني استفدت من ال Parallel computation

ال brute force افضل من ال DAC في هاي المسألة

General Divide and Conquer Recurrence

$$\boldsymbol{T(n) = a\,T\left(\frac{n}{b}\right) + D(n) + C(n), \qquad n: large}$$

The order of growth of its solution T (n) depends on:

- The values of the constants a and b
- The order of growth of the $D(n)$ and $C(n)$

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the **master theorem**

Master Method

- Master method **provides bound** for recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

$\boldsymbol{D(n) + C(n)}$

$$T(n) = a\,T\left(\frac{n}{b}\right) + c.n^{d}$$

Where $\boldsymbol{a \geq 1}$, $\boldsymbol{b > 1}$ $\boldsymbol{and\ f(n)\ is\ a\ given\ function\ which\ is\ asymptotically\ positive.}$

- This recurrence characterizes a divide-and-conquer algorithm that divides a problem of size *n* into subproblems, each of size $\frac{n}{b}$, and solves them recursively.

Master Theorem

$$T(n) = a\,T(\frac{n}{b}) + c.n^d$$

- **Master Theorem** : If $f(n) \in O(n^d)$ where $d \geq 0$ Then:
- if $a = b^d$ $\quad T(n) \in O(n^d \log_b n)$
- if $a > b^d$ $\quad T(n) \in O(n^{\log_b a})$
- if $a < b^d$ $\quad T(n) \in O(n^d)$

Analogous results hold for the θ and Ω notations, too.

Note that the master theorem is useful to find the **solution's efficiency class** (order of growth) without going through solving the recurrence

Example : Divide-and-conquer sum-computation algorithm

- The problem is divided into 2 subproblems each of size $n/2 \rightarrow \boldsymbol{b = 2}$
- 2 subproblems need to be solved $\rightarrow \boldsymbol{a = 2}$
- The cost of dividing the problem and combining the solutions is constant θ (1)

$$\boldsymbol{T(n) = 2\,T(n/2) + 1} \qquad \boldsymbol{n > 1}$$

a = 2 , b = 2 , d = 0 ; hence, since a > b^d

$T(n) \in O(n^{\log_b a})$

$T(n) \in O(n^{\log_2 2})$ $\rightarrow$ $T(n) \in O(n)$

How does this algorithm compare with the brute-force algorithm for this problem?????

Example

$T(n) = a\,T(n/b) + D(n) + C(n)$

Consider a divide and conquer algorithm XYZ with the following properties:

- each time, a problem is divided into 2 subproblems where size of each subproblem is half of the input. $b = 2$
- Two subproblems need to be solved. $a = 2$
- The time taken to divide the problem and merging the solutions is O(n)

1) Set up the recurrence relation of the running time T(n).

   $T(n) = 2\,T(n/2) + O(n)$

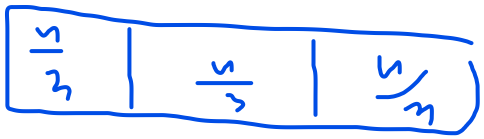
1) Use the master theorem to derive the order of growth of XYZ algorithm.

   a = 2 , b = 2 , d = 1

   Since $a = b^d$ → $T(n) \in O(n^d \log_b n)$

   $T(n) \in O(n \log n)$

Exercise



To solve problem P of size n, it is divided into 3 equal subproblems. 2 out of the 3 subproblems need to be solved to solve P. The solutions of the 2 solved subproblems are merged to obtain the global solution of P with a cost of O(n).

$b=3$

$a=2$

1. Set up the running time function T(n) of the algorithm that solves P.
2. What is the order of growth this algorithm.

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

Exercise

Find the order of growth for solutions of the following recurrences.

Hint: Use master theorem

1. $T(n) = 4\,T(n/2) + n\,,$ $\quad$ T(1) = 1
2. $T(n) = 4\,T(n/2) + n^2\,,$ $\quad$ T(1) = 1
3. $T(n) = 4\,T(n/2) + n^3\,,$ $\quad$ T(1) = 1

Pair Activity

Finding the Maximum Element in an Array using Divide and Conquer

Activity: Finding the Maximum Element in an Array using Divide and Conquer

- How a brute force approach would solve this problem and what its time complexity would be
- Design a divide-and-conquer algorithm for finding the position (index) of the largest element in an array of n numbers.

divide the array in two halves, recursively find the maximum element in each half, and compare the two maximums to find the overall maximum.

- Work in small groups to implement the algorithm in pseudocode
- Discuss with your students the time complexity of the algorithm and how it compares to the brute force approach.

Divide-and-Conquer Examples

- **Sorting**:
  - merge sort
  - quicksort
- **Binary tree traversals**
- **Mathematics**
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Exponentiation
- **Computational geometry**
  - Closest-pair
  - convex-hull algorithms
- **Searching**:
  - Binary search: decrease-by-half (or degenerate divide&conq.)

Binary Search

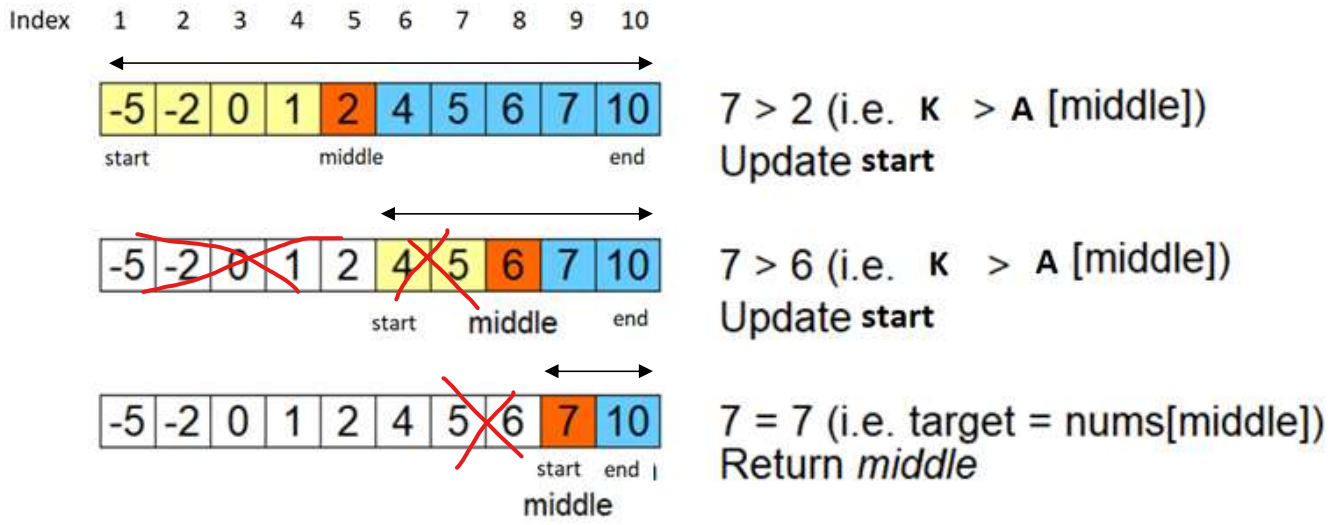
- **Binary search** is a remarkably efficient algorithm for **searching for the value K in a sorted array**. If K is found, the algorithm returns an index of the array element equals K. If K is not found, the algorithm returns -1, which is assumed not to be a valid index.
- **Binary search method:**
  - Compare a search key K with the array's middle element $A[m]$. m = ⌊(start + end)/2⌋ (rounded down)
  - If they match, the algorithm stops.
  - Otherwise, the same operation is repeated recursively:
    - for the first half of the array if $K < A[m]$
    - for the second half if $K > A[m]$:

$$\underbrace{A[1] \ldots A[m-1]}_{\text{search here if } K<A[m]} \quad \overset{K}{\updownarrow} A[m] \quad \underbrace{A[m+1] \ldots A[n]}_{\text{search here if } K>A[m]}$$

$$\underbrace{A[0] \ldots A[m-1]}_{\text{search here if } K<A[m]} \quad \overset{K}{\updownarrow} A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\text{search here if } K>A[m]}$$

Binary Search Example

- Apply **binary search** algorithm to searching for **K = 7** in the array



- Apply binary search algorithm to searching for **K = 2** in the array. (**Best case**)
- **Interactive animations of Binary Search**

https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search Example

For a better understanding of Binary search algorithm and visualizing its operation through animation:

https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search using DAC

```
Algorithm BinarySearch (A , start , end , key )
{
    if (start > end)
        return -1
    else
        mid = ⌊(start + end)/2⌋
        if key == A[mid]
            return mid
        else if key < A[mid]
            return BinarySearch (A , start, mid-1 , key)
        else
            return BinarySearch (A, mid+1, end, key
}
```

$$T(n) = \begin{cases} d, & n = 1 \\ T\left(\frac{n}{2}\right) + c, & n > 1 \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + c \qquad n>1$$

- Cost of solving one subproblem of size n/2
- Cost of dividing the problem

Exercise

- **Binary search can be easily implemented as a non-recursive algorithm.**

1. Write the pseudocode of this non-recursive version.
2. Analyze the non-recursive version and derive its time complexity in the worst case.

Algorithm ***NR_BinarySearch*** **(*A [1 … N] , K*)**

{

//Input: An array A[1…. n] sorted in ascending order and a search key K

//Output: An index of the array's element that is equal to *K*

// or -1 if there is no such element

}

Analysis of Binary Search

- The problem is divided into 2 subproblems each of size $n/2$
- Cost of dividing the problem: computing the middle takes $O(1)$
- solving 1 sub-problem out of 2 subproblems takes $T(n/2)$ **why???????**
- Cost of combining the solutions ??? *Not required*
- Total:

$$T(n) = T(n/2) + O(1) \qquad \text{if } n > 1$$

a = 1 , b = 2 , d = 0

Since a = b^d → $T(n) \in O(n^d \log_b n)$

$T(n) \in O(\log n)$

- The worst-case time efficiency of binary search is in $\Theta(\log n)$

Notes on Binary Search

- In fact, if $a = 1$, some people consider binary search algorithm degenerate cases of divide-and-conquer, where just one of two subproblems of half the size needs to be solved. However, others consider this paradigm decrease-by-a-constant-factor as different design paradigm.
- there are searching algorithms (see interpolation search and hashing) with a better average-case time efficiency, and one of them (hashing) **does not even require the array to be sorted**! These algorithms do require some special calculations in addition to key comparisons, however.

Search Topics

- Interpolation Search
- Hashing
- Real application of binary search

Exercise

- If we want to design a Divide and Conquer algorithm for searching a key value in unsorted array.

1. Set up the running time function T(n) of the algorithm.
2. What is the order of growth of this algorithm.

Some applications of binary search

- **Some applications of Binary Search**
  - Find the first value greater than or equal to x in a given array of sorted integers
  - Find the peak of an array which increases and then decreases
- **Some real-life applications of binary search**
  - Searching in the dictionary
  - Searching for a book in a library where books are usually arranged in alphabetical order or by some integer code.
  - Page number

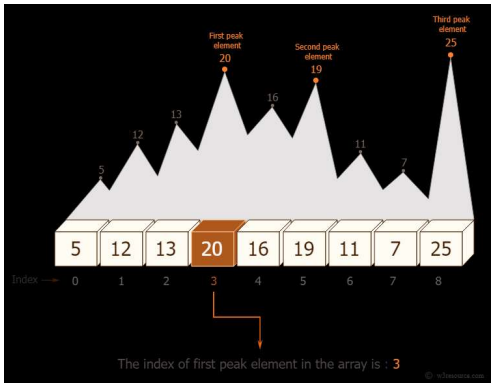
Exercise

- **Find peak element**

A peak element is an element that is strictly greater than its neighbors.
Given an integer array, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

Examples:

Input : [8, 9, 10, 2, 5, 6]
Output: The peak element is 10 (or 6)
Input : [8, 9, 10, 12, 15]
Output: The peak element is 15
Input : [10, 8, 6, 5, 3, 2]
Output: The peak element is 10



a. Design an efficient divide-and-conquer algorithm for finding the peak element.
b. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.
c. How does this algorithm compare with the brute-force algorithm for this problem?

Sorting Algorithms

- **In this part, we will learn two sorting algorithms as a successful application of the Divide and Conquer technique.**
  - Merge sort
  - Quicksort

$T(n) = aT(\frac{n}{b}) + D(n) + C(n)$

$= 2T(\frac{n}{2}) + O(1) + O(n)$

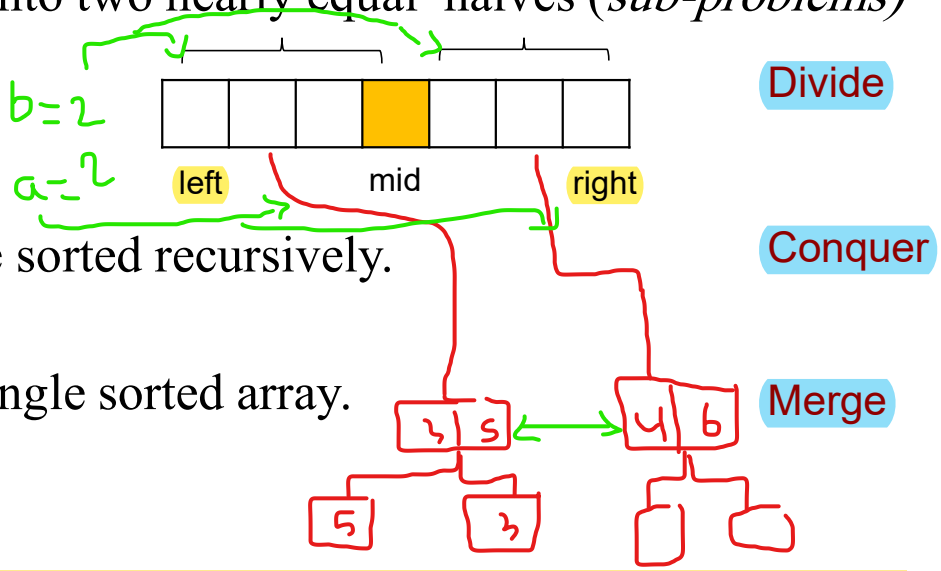
$= 2T(\frac{n}{2}) + n$

$= O(n \log n)$

Merge sort

- A *sub-problem* of one element is already sorted [Base case]
- The original array $A[\,1 \ \ldots \ n\,]$ is divided into two nearly equal halves (*sub-problems*)

  $A[\,1 \ \ldots \lfloor n/2 \rfloor\,]$ and $A[\lfloor n/2 \rfloor + 1 \ \ldots \ n\,]$.
- sub-problems of two or more elements are sorted recursively.
- The two sorted arrays are merged into a single sorted array.



The problem is **small** if there is just a single element which we don't have to sort it. Otherwise, the problem is considered large and need to bs solved recursively

Pseudocode of DAC for Merge Sort

```
ALGORITHM MergeSort ( A , left,  right )
{
        if (left = right) then
            Return
        endif
        mid  =  (left  + right)/2
        MergeSort ( A , left, mid)
        MergeSort ( A , mid + 1, right)
        Merge ( A , left, mid, right)
}
```

// No sorting needed for single-element subarray

// Divide the array into two halves
// Sort them recursively

// Merge the sorted halves

Pseudocode of DAC for Merge Sort

ALGORITHM $MergeSort\ (\ A\ , left,\ \ right\ )$

{

$\boldsymbol{if}\ (left = right)\ \boldsymbol{then}$

Return

endif

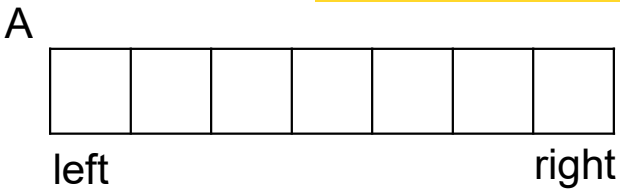
$mid\ =\ (left\ + right)/2$

$MergeSort\ (\ A\ , left, mid)$

$MergeSort\ (\ A\ , mid + 1, right)$

$Merge\ (\ A\ , left, mid, right)$

}

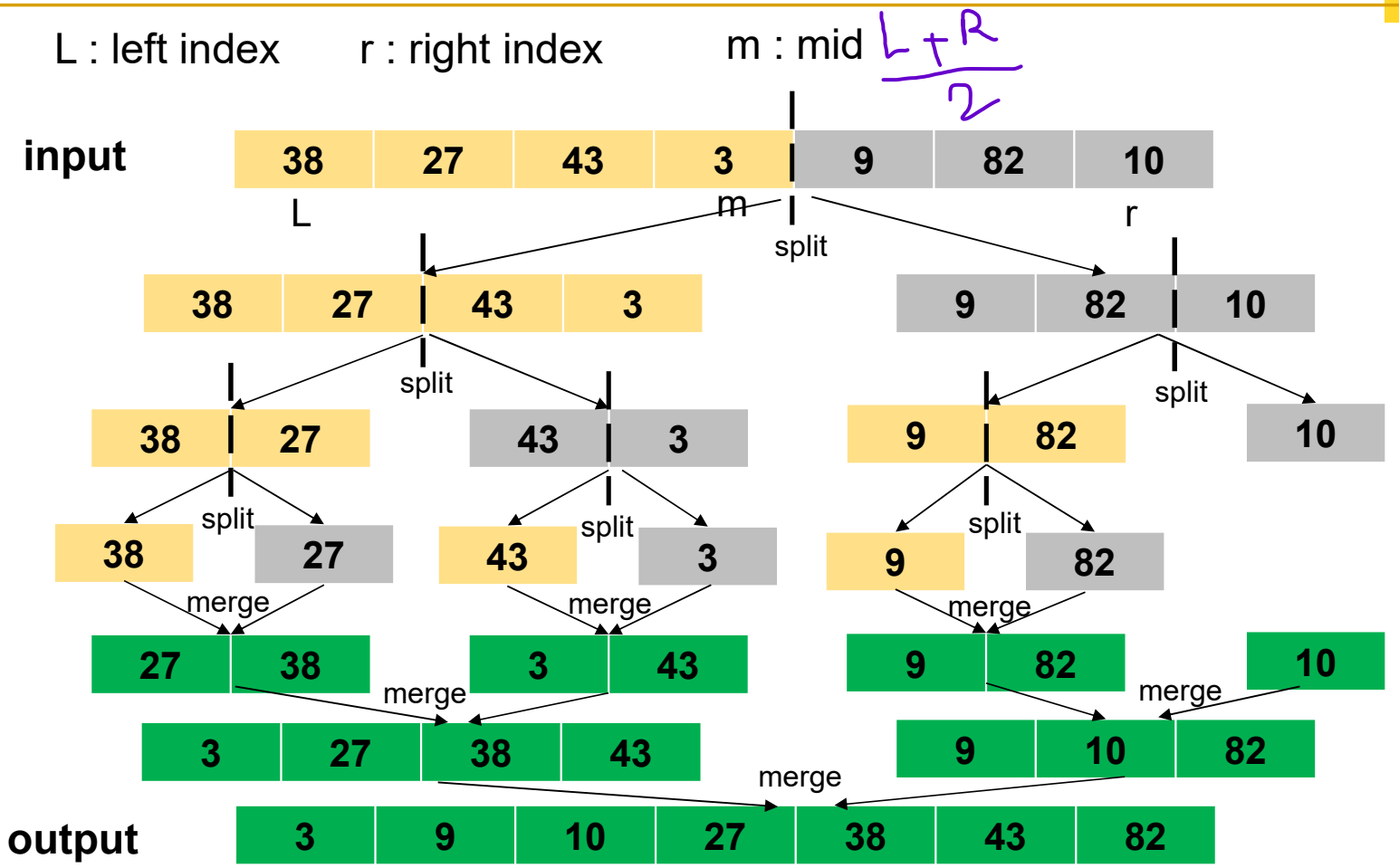


- When **left = right**, there is one element (the problem is small enough)
- **left < right** : indicates that there is at least two elements (**solved recursively**)

Merge() takes two sorted subarrays of A and merges them into a single sorted subarray of A.

It requires O(n) time

Merge sort Examples



Is l < r (yes)
m = floor ((l + r) / 2)
Divide the array into two parts

the array is recursively divided into two halves till the size becomes 1

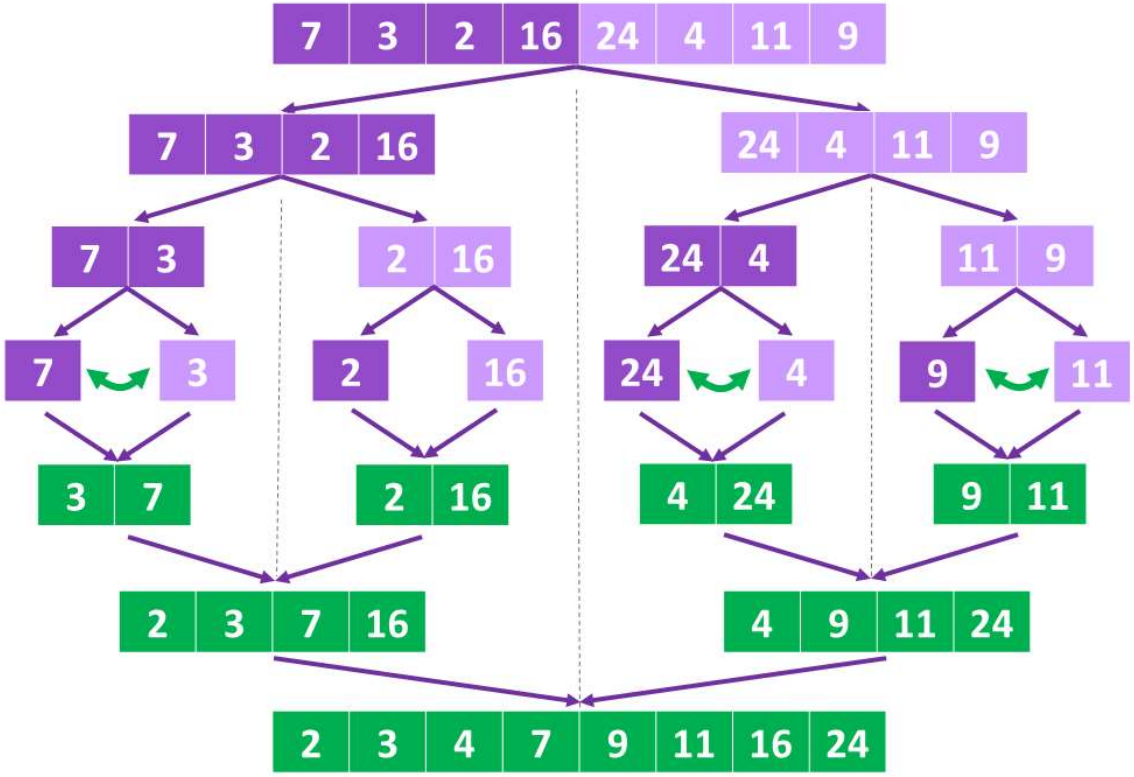
Is l < r (No)
Array of one element is already sorted

Once the size becomes 1, the merge processes come into action

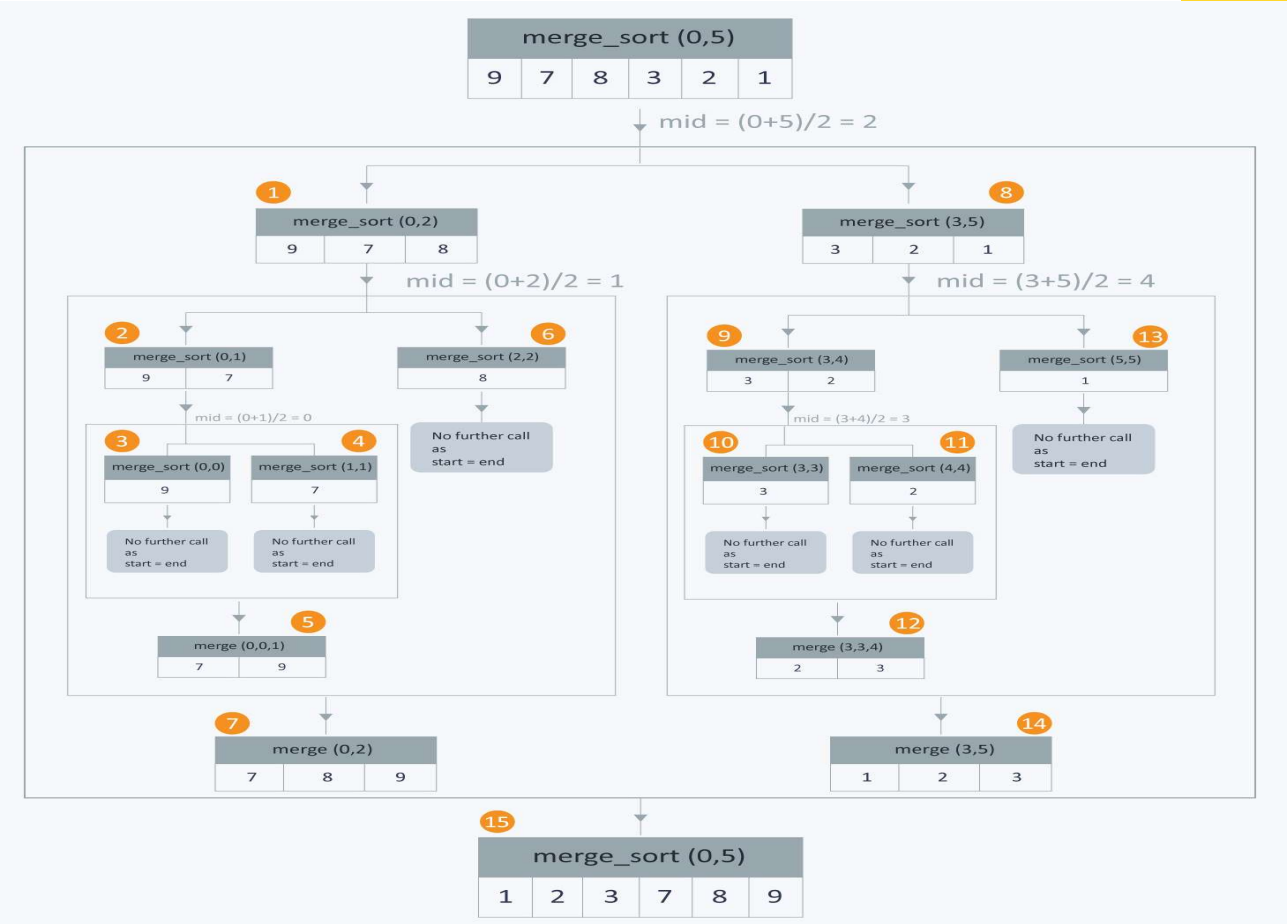
Merge sort Examples

Given the following list: 7, 3, 2, 16, 24, 4, 11, 9

Draw the tree generated by applying **Merge Sort**.



Merge sort Examples



Merge sort Examples

For a better understanding of Merge sort algorithm and visualizing its operation through animation:

https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/

https://visualgo.net/en

Analysis of Merge Sort

```
ALGORITHM MergeSort ( A , left,  right )
{
        if (left < right) then
            mid = (left + right)/2
            MergeSort ( A , left, mid)
            MergeSort ( A , mid + 1, right)
            Merge ( A , left, mid, right)
        endif
}
```

right

$O(1)$

$T(n/2)$

$T(n/2)$

$O(n)$

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

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

Analysis of Merge Sort

- **How efficient is merge sort???**
- Based on the general recurrence : $T(n) = a\,T\left(\frac{n}{b}\right) + cost\ of\ divide + cost\ of\ merge$

- The problem is divided into 2 subproblems each of size $n/2$
- Cost of dividing the problem: computing the middle takes $O(1)$
- solving 2 sub-problems each of size n/2 takes **2T(n/2)**
- Cost of combining the solutions : Merging two sorted lists takes $O(n)$
- Total:

$$T(n) = 2T(n/2) + O(1) + O(n) \qquad \text{if } n > 1$$

a = 2 , b = 2 , d = 1

Since a = b^d → $T(n) \in O(n^d \log_b n)$

$T(n) \in O(n\ logn)$

Time complexity of Merge Sort

- What is the best case of merge sort algorithm?
- What is the worst case of merge sort algorithm?

O(nlogn)

Time complexity of Merge Sort is O(n*Log n) in **all the 3 cases (worst, average and best)** as merge sort always divides the array in two halves and takes linear time to merge two halves regardless of the nature of the input data

Merge two sorted arrays

Basically

- After finishing elements from any of the sub arrays, we can add the remaining elements from the other sub array to our sorted output array as it is.
- This is because left and right sub arrays are already sorted.

Time Complexity

The mentioned merge procedure takes **Θ(n)** time.

This is because we are just filling an array of size n from left & right sub arrays by incrementing i and j at most Θ(n) times.

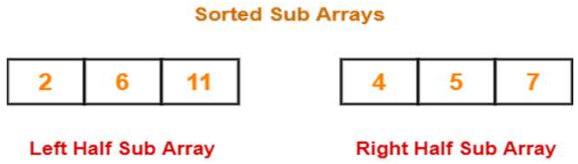
How to merge two sorted lists

The merge procedure of merge sort algorithm is **used to merge two sorted arrays into a third array in sorted order**

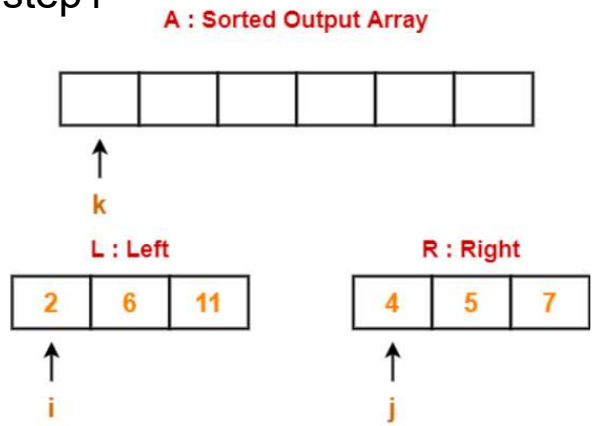
- **The merging of two sorted arrays can be done as follows:**
  - Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
  - The elements pointed to are compared.
  - The smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
  - This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

Example

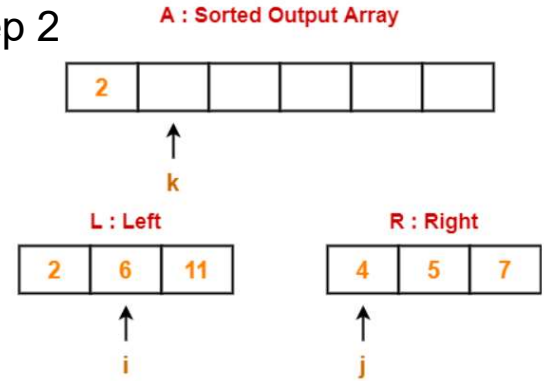
- Consider we want to merge the following two sorted sub arrays into a third array in sorted order



step1

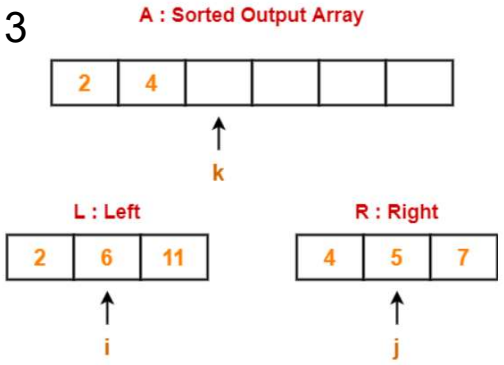


Step 2



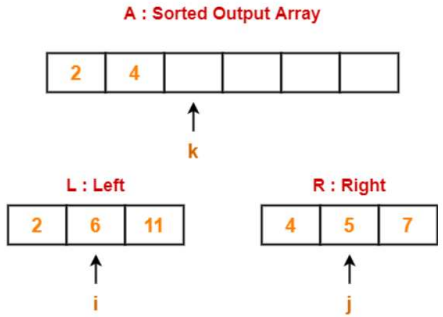
- Since L[1] < R[1], so we perform A[1] = L[1] i.e. we copy the first element from left sub array to our sorted output array.
- increment i and k by 1.

Step 3

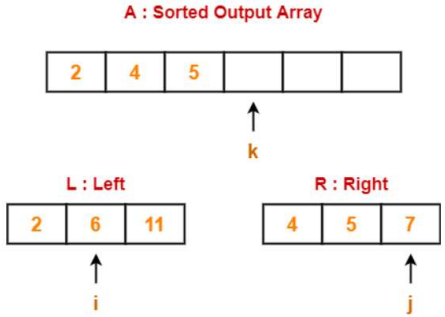


- Since L[2] > R[1], so we perform A[2] = R[1] i.e. we copy the first element from right sub array to our sorted output array.
- Increment j and k by 1.

Example [Cont]



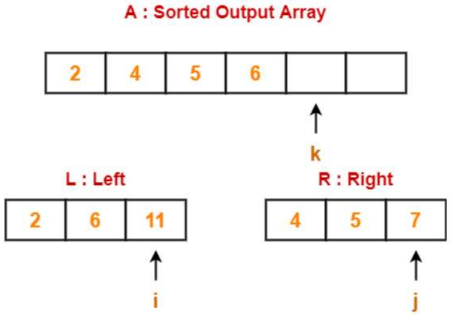
- Since L[2] > R[2], so we perform A[2] = R[1]
- increment j and k by 1.



Step 4

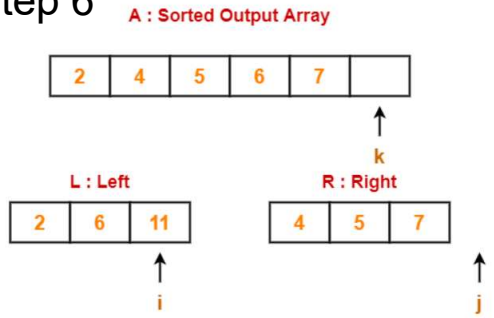
- Since L[2] < R[3], so we perform A[4] = L[2]
- increment i and k by 1.

Step 5

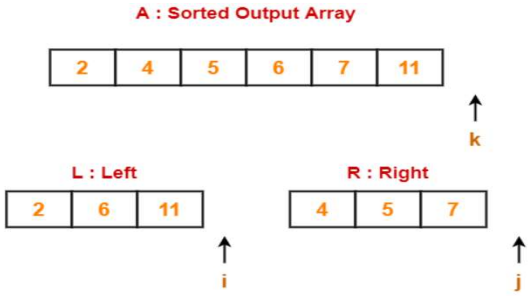


- Since L[3] > R[3], so we perform A[5] = R[3]
- increment j and k by 1.

Step 6



- all the elements from right sub array have been added to the sorted output array.
- we add remaining elements from the left sub array to the sorted output



Step 7

Pseudocode of merging two sorted lists

```
Merge (A, left, mid, right)
{
    create list L [left ... mid] , R [mid + 1 ... right]
    i = left, j = mid + 1
    while (both lists are not empty i <= mid && j <= right) do
        if (L [i] <= R [j])
            A [k] = L [i]
            i = i + 1
            k = k + 1
        else
            A [k]=R [j]
            j = j + 1
            k = k + 1
    //append the remainder of non-empty list to output list
    If (i > mid) put the rest of R in A[k ... right]
    If ( j > right ) put the rest of L in A[k ... right]
}
```

Time Complexity

The merge procedure takes **Θ(n)** time.

This is because we are just filling an array of size n from left & right sub arrays by incrementing i and j at most Θ(n) times.

- Sorting properties
  - Stable $\Rightarrow$ relative order of equal keys unchanged
    - Stable: 3, 1, 4, 3, 3, 2 $\rightarrow$ 1, 2, 3, 3, 3, 4
    - Unstable: 3, 1, 4, 3, 3, 2 $\rightarrow$ 1, 2, 3, 3, 3, 4

- When merge encounters equal keys, it copies the one with the smaller index to the output array. So, merge sort is **stable** sort

- We've seen that **selection sort** is simple and inefficient, and **merge-sort** is more efficient in time at the expense of additional space. Fortunately, we can do better than that.

- We'll talk about **quicksort**, which uses a divide and conquer approach, and **heapsort**, which uses an array to store a tree structure

Exercise

a) Apply merge sort to sot the list *E, X, A, M, P, L, E* in alphabetical order (**show your work**)

b) Apply merge sort on the array A = {41, 52, 26, 38, 57, 9, 49}

Is merge sort in-place or out of place algorithm?

The traditional implementation of Merge Sort is an out-of-place algorithm, meaning that it requires additional memory to store temporary arrays during the merging process. This is because Merge Sort works by dividing the array into smaller subarrays, sorting them recursively, and then merging them back into a sorted array. During the merging step, the algorithm creates temporary arrays to hold the divided subarrays and then merges them back into the original array. This merging process requires extra memory space proportional to the size of the input array.

- However, it is possible to modify the Merge Sort algorithm to make it an in-place algorithm, where the sorting is performed directly on the original array without requiring additional memory?????
- will affect the run-time complexity of the algorithm

Is merge sort in-place or out of place algorithm?

- However, it is possible to modify the Merge Sort algorithm to make it an in-place algorithm, where the sorting is performed directly on the original array without requiring additional memory?????
- will this affect the run-time complexity of the algorithm????

Bottom-up merge sort

- can we implement merge sort in a bottom-up manner (iterative merge sort)?

Pair Activity

By engaging students in this activity, they will have the opportunity to actively participate in

- implementing the iterative Binary Search algorithm
- compare it with the recursive approach, and gain insights into the differences in time and space complexity.
- It will also foster discussions and critical thinking about algorithm design choices.

Comparing Binary Search: Iterative vs Recursive Approach

Activity: Finding the Maximum Element in an Array using Divide and Conquer

- Divide students into pairs or small groups
- Task1: Implement Binary Search iteratively (bottom-up)
- Task2: Explain the time complexity of both approaches (O(log n))
- Task3: Discuss space complexity differences
- Task4: Discuss the advantages and disadvantages of each approach, considering factors such as code simplicity, readability, and potential limitations (e.g., stack overflow in the recursive approach for extremely large inputs).