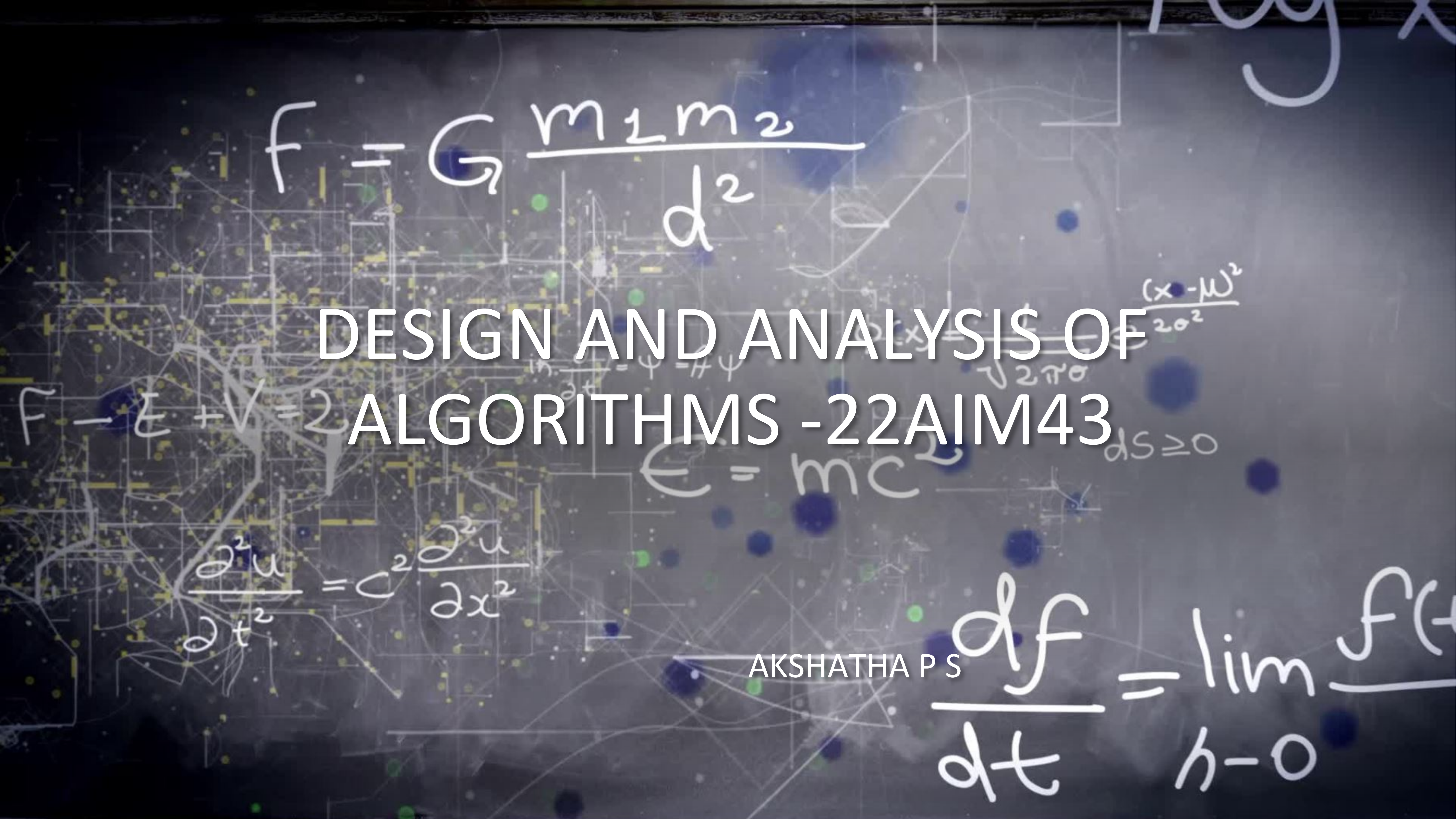


DESIGN AND ANALYSIS OF ALGORITHMS -22AIM43

AKSHATHA P S

MODULE-1	**INTRODUCTION**	**22AIM43.1**	**8 Hours**
Introduction to Algorithms, Role of algorithms in computing, Time and Space Complexity of Algorithms, Asymptotic notations, worst-case, Average-case and Best-case analysis, Analysis Framework- Empirical analysis- Mathematical analysis for Recursive and Non-recursive algorithms.			
Case Study	Illustrate real-world applications of algorithms and growth functions.		
Text Book	Text Book 1:1.1,1.2,1.3		
MODULE-2	**DIVIDE AND CONQUER**	**22AIM43.2**	**8 Hours**
Divide and Conquer Methodology: Binary search, Merge sort, Quick sort, Finding the maximum and minimum, Strassen's matrix, advantages and disadvantages of divide and conquer.			
Case Study	Compare and contrast the time complexity and suitability of the bubble sort, merge sort, and quicksort algorithms. Provide scenarios where one might be preferred over the others.		
Text Book	Text Book 1: 2.1,2.2		
MODULE-3	**GREEDY METHOD AND DYNAMIC PROGRAMMING**	**22AIM43.3**	**8 Hours**
Greeedy method: Introduction, Job scheduling problem, Minimum Spanning tree algorithms – Kruskals & Prims. Optimal Tree Problem: Huffman Trees. **DYNAMIC PROGRAMMING:** Introduction, Knapsack problems, Travelling Salesman problem. Transitive closure - Warshall's and Floyds algorithm.			

[illegible]	[illegible]		
MODULE-4	**DECREASE & CONQUER, TRANSFORM & CONQUER**	**22AIM43.4**	**8 Hours**
Decrease &conquer: Introduction – Decrease by constant, decrease by constant factor-Fake Coin Problem-Russian Peasant Multiplication, variable size decrease. **Transform & conquer**: Introduction, Balanced Search trees – AVL trees & 2-3 trees, Red Black Trees			
Text Book	Text Book 1: 5.1,5.2,5.3,5.4,5.5,5.6		
MODULE-5	**BACKTRACKING, BRANCH AND BOUND**	**22AIM43.5,22AIM43.6**	**8 Hours**
Backtracking: Introduction, N Queens problem, subset sum problem, **Branch and Bound:** Introduction, Travelling Salesman problem, Knapsack problem, Assignment problem, NP-Hard and NP-Complete problems: Basic concepts, non-deterministic algorithms.			
[illegible]	[illegible]		

Suggested Learning Resources:

Text Books:

1. Anany Levitin,"Introduction to the Design &Analysis of Algorithms",3rd Edition, PEARSON Education, 2012.

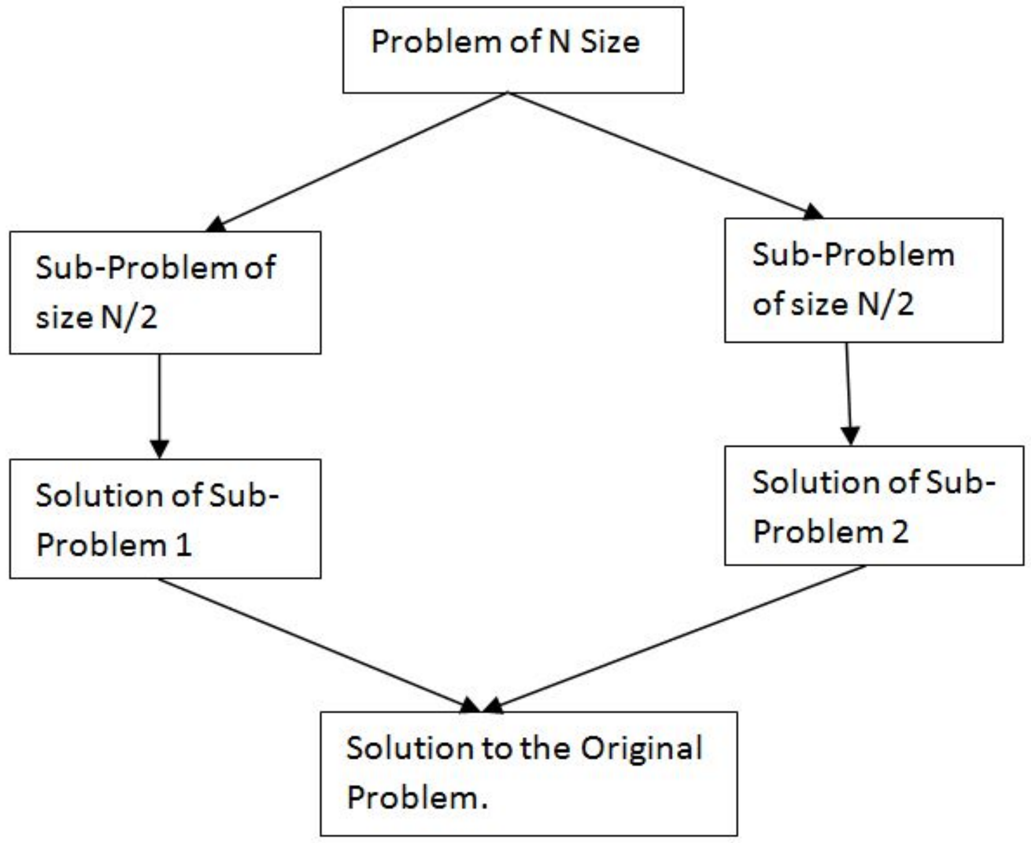
ReferenceBook:

1.Thomas H Cormen, Charles E Leiserson, Ronald R Rivest & Clifford Stein, "Introduction to Algorithms ", THIRD Edition, Eastern Economy Edition

Module 2: Divide and Conquer

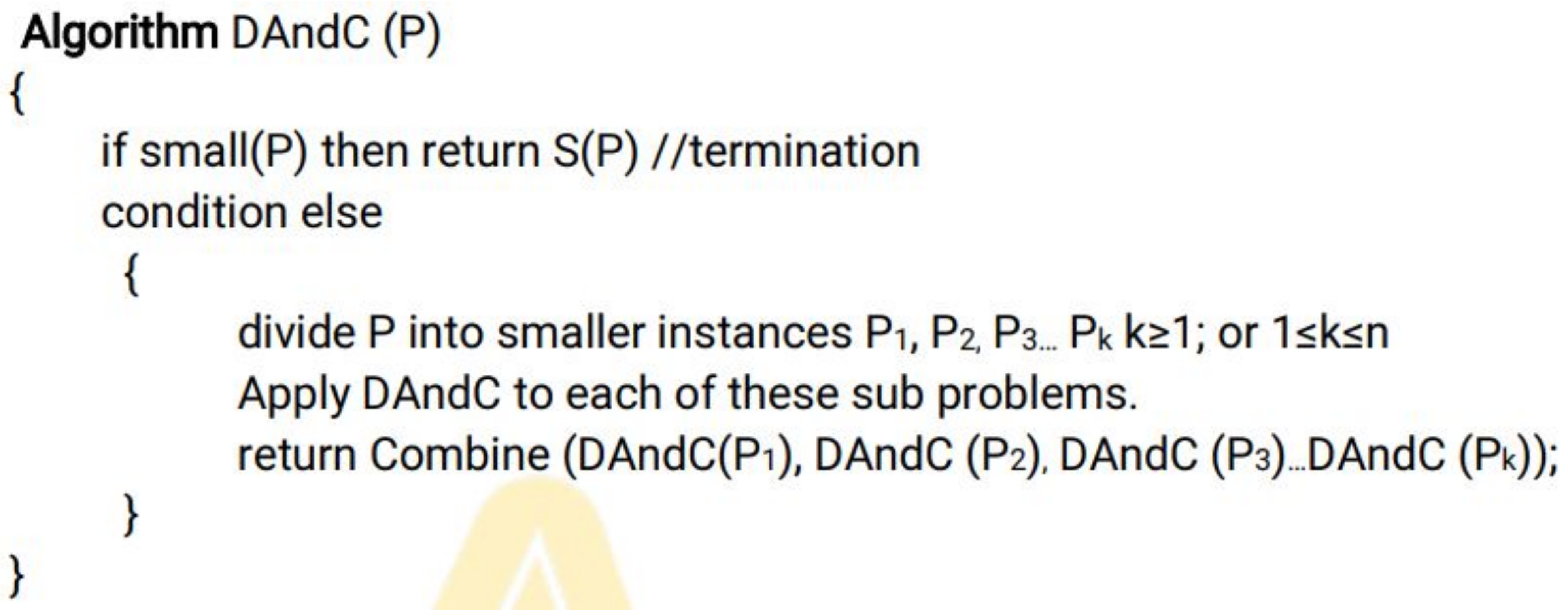
General method for divide and conquer technique

Divide and Conquer is one of the best-known general algorithm design technique.


Problem of N Size
Sub-Problem of size N/2
Sub-Problem of size N/2
Solution of Sub-Problem 1
Solution of Sub-Problem 2
Solution to the Original Problem.

A typical case with k=2

Control Abstraction for divide and conquer technique



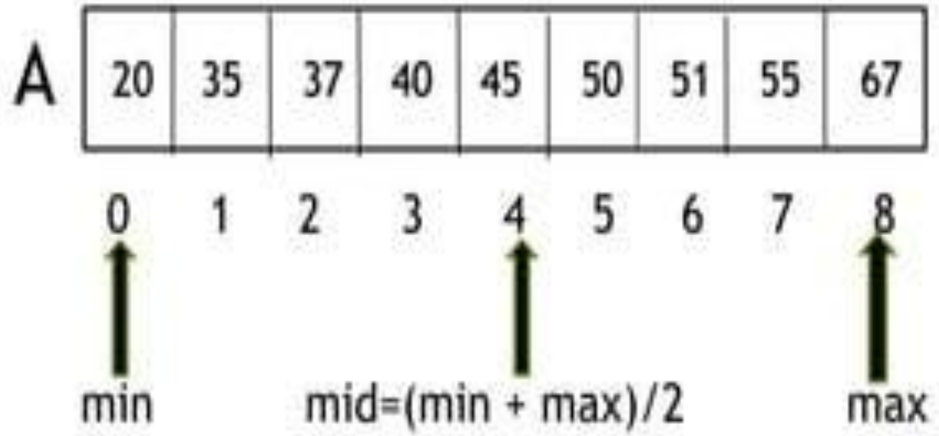
```
Algorithm DAndC (P)
{
    if small(P) then return S(P) //termination
    condition else
     {
          divide P into smaller instances $P_1$, $P_2$, $P_3$... $P_k$ $k \geq 1$; or $1 \leq k \leq n$
          Apply DAndC to each of these sub problems.
          return Combine (DAndC($P_1$), DAndC ($P_2$), DAndC ($P_3$)...DAndC ($P_k$));
     }
}
```

Introduction

- **Binary Search** is a search algorithm that finds the position of a target value within a sorted array.
- **Binary Search** is also known as **half-interval search** or **logarithmic search.**
- **Binary Search** algorithm works on the principle of **Divide And Conquer**.
- **Binary Search** can be implemented on only sorted list of elements.

Example

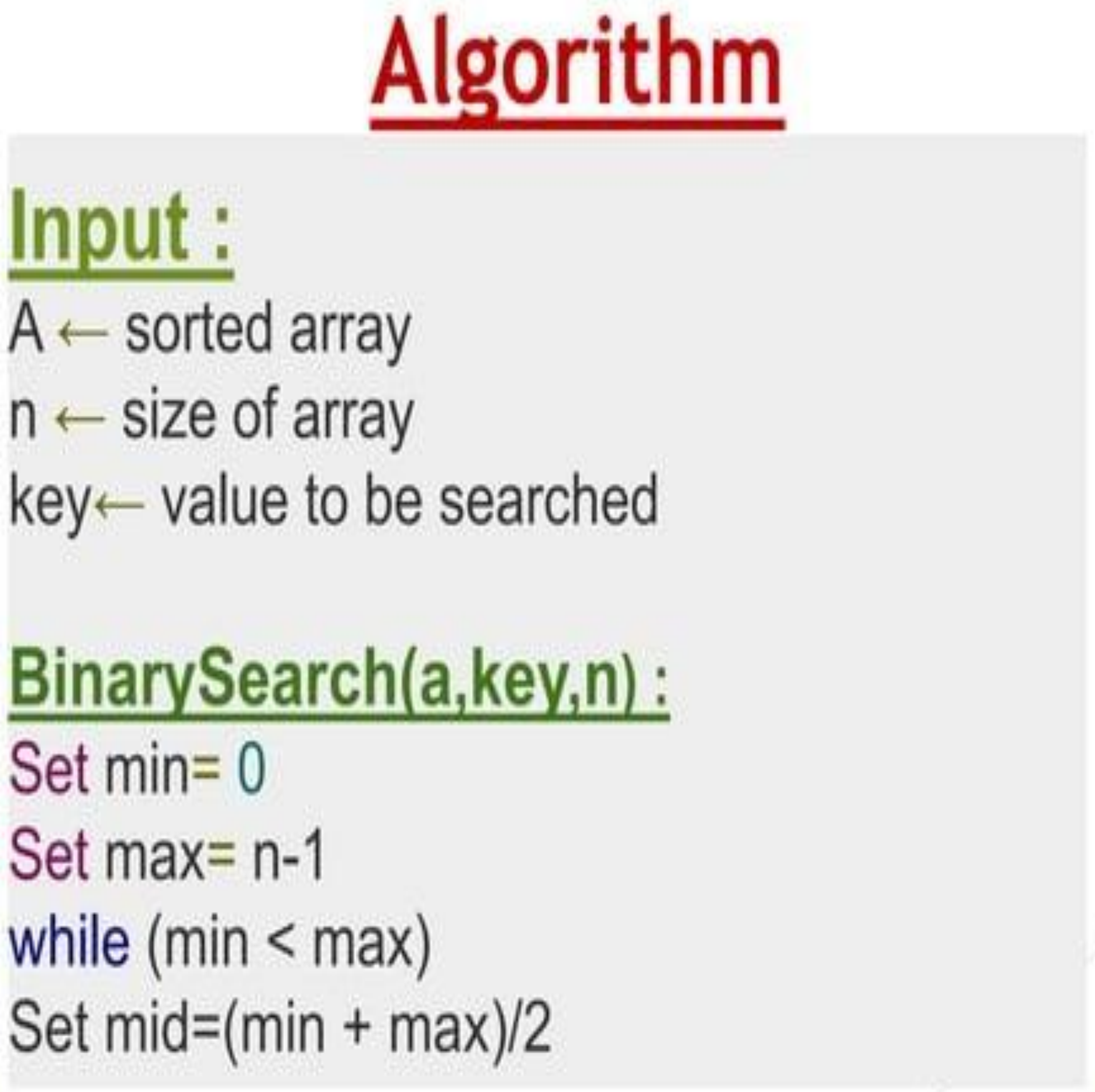
- Consider an Array:



If(37==a[mid]) $\longrightarrow$ return mid

Else if(37<a[mid]) $\longrightarrow$ max=mid-1

Else if(37>a[mid]) $\longrightarrow$ min=mid+1



Algorithm

Input :

A ← sorted array

n ← size of array

key← value to be searched

BinarySearch(a,key,n) :

```
Set min= 0
Set max= n-1
while (min < max)
Set mid=(min + max)/2
```

Algorithm

```
if (key ==A[mid])
Return mid
Else if(key<A[mid])
set max= mid - 1
Else
set min= mid + 1
End if
End while
Return (- 1)
```

Example:-

Consider the elements :

-15,-6,0,7,9,23,25,54,82,101,112,125,131,142,151.

x=151

low	high	mid
1	14	7
8	14	11
12	14	13
14	14	14
		Found

x=-14

low	high	mid
1	14	7
1	6	3
1	2	1
2	2	2
2	1	**Not found**

x=9

low	high	mid
1	14	7
1	6	3
4	6	5
		Found

Analysis of input size at each iteration of Binary Search:

At Iteration 1:

Length of array = ***n***

At Iteration 2:

Length of array = ***n/2***

At Iteration 3:

Length of array = $(n/2)/2 = n/2^2$

Therefore, after Iteration k:

Length of array = $n/2^k$

Length of array = ***$n/2^k$***

Also, we know that after **k iterations**, the **length of the array becomes 1** Therefore, the Length of the array

$n/2^k = 1$

=> **$n = 2^k$**

Applying log function on both sides:

=> ***$\log_2 n = \log_2 2^k$***

=> ***$\log_2 n = k * \log_2 2$***

As ***($\log_a (a) = 1$)*** *Therefore,* ***$k = \log_2(n)$***

Time Complexity

- The running time complexity for **Binary Search** are of three types :-
  - In Worst Case ,Time Complexity : O(log n)
  - In Average Case ,Time Complexity : O(log n)
  - In Best Case , Time Complexity : O(1)

Applications of Binary Search:

- **Searching in sorted arrays:** Binary search is used to efficiently find an element in a sorted array.
- **Database queries:** Binary search can be used to quickly locate records in a database table that is sorted by a specific key.
- **Finding the closest match:** Binary search can be used to find the closest value to a target value in a sorted list.
- **Interpolation search:** Binary search can be used as a starting point for interpolation search, which is an even faster search algorithm.

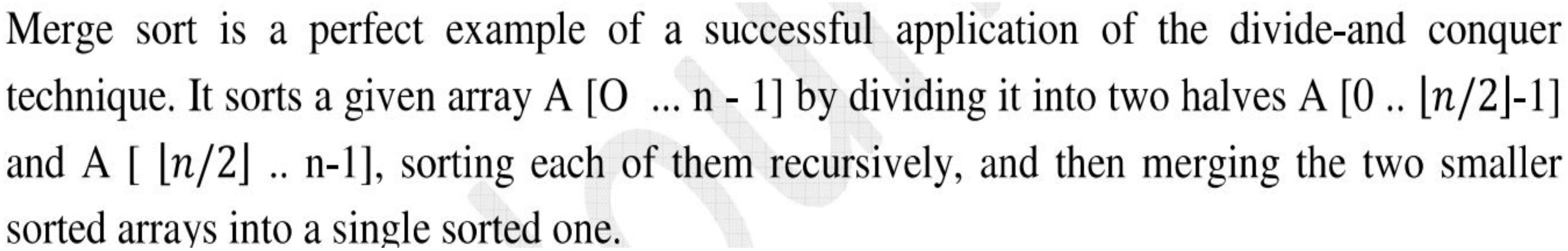
Advantages of Binary Search:

- **Efficient:** Binary search has a time complexity of **O(log n)**, which makes it very efficient for searching large sorted arrays.
- **Simple to implement:** Binary search is relatively easy to implement and understand.
- **Versatile:** Binary search can be used in a wide variety of applications.
- **Reliable:** Binary search is a reliable algorithm that will always find the target element if it exists in the array.

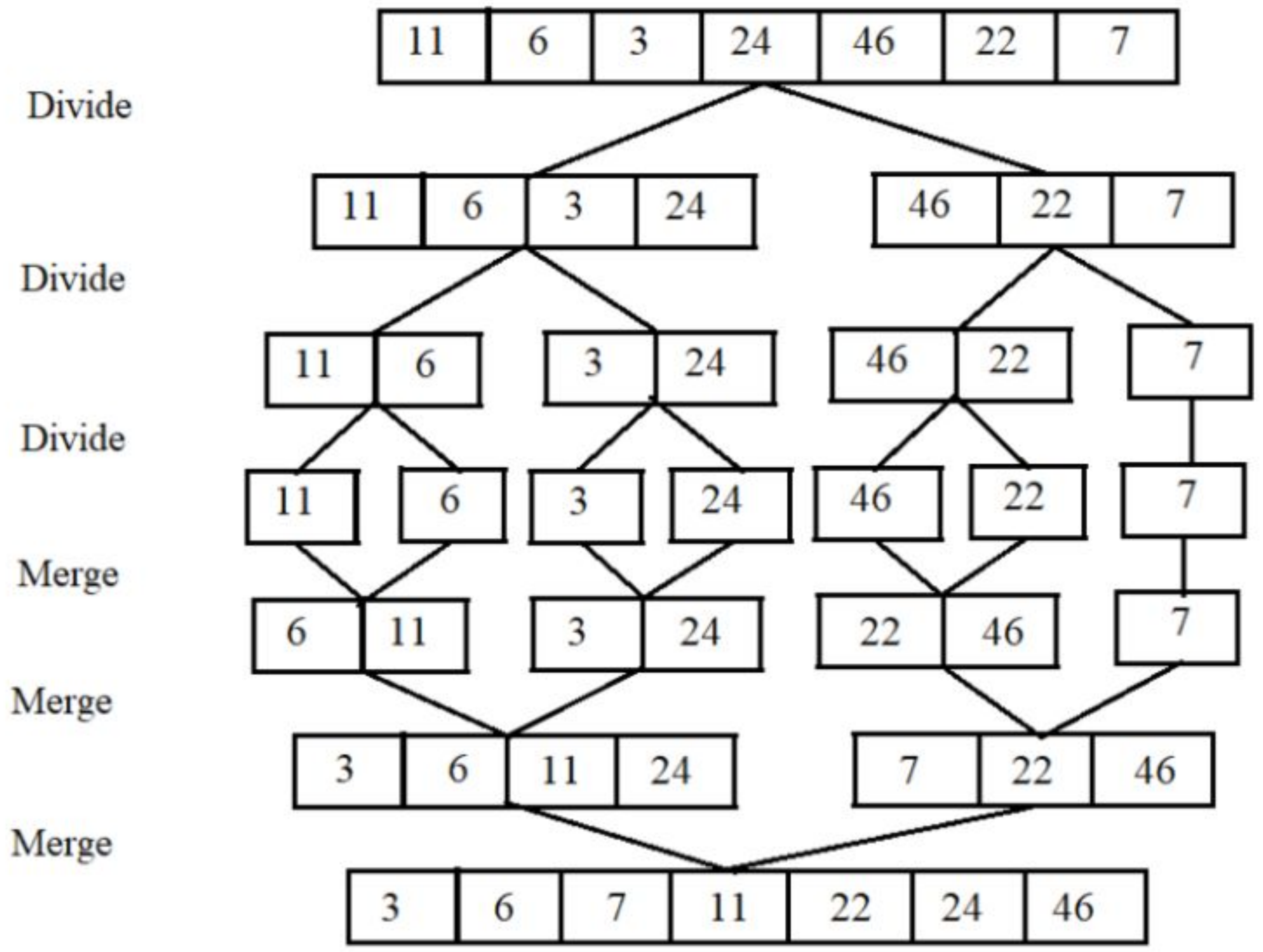
Disadvantages of Binary Search:

- **Requires a sorted array:** Binary search only works on sorted arrays. If the array is not sorted, it must be sorted before binary search can be used.
- **May not be the best choice for large arrays:** For very large arrays (e.g., billions of elements), other search algorithms such as interpolation search or hash tables may be more efficient.

MERGE SORT

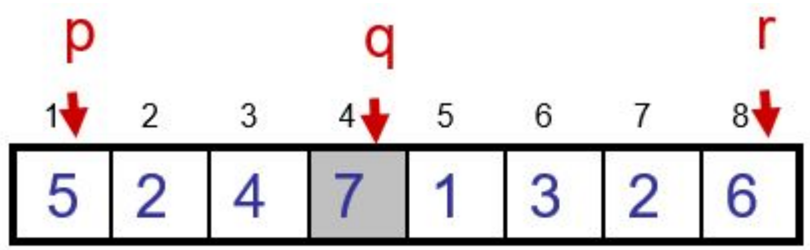

Merge sort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A [O ... n - 1] by dividing it into two halves A [0 .. $\lfloor n/2 \rfloor$-1] and A [$\lfloor n/2 \rfloor$ .. n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

- To sort an array *A*[p . . r]:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of n/2 elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences


Divide
Divide
Divide
Merge
Merge
Merge
11 6 3 24 46 22 7
11 6 3 24
46 22 7
11 6
3 24
46 22
7
11
6
3
24
46
22
7
6 11
3 24
22 46
7
3 6 11 24
7 22 46
3 6 7 11 22 24 46

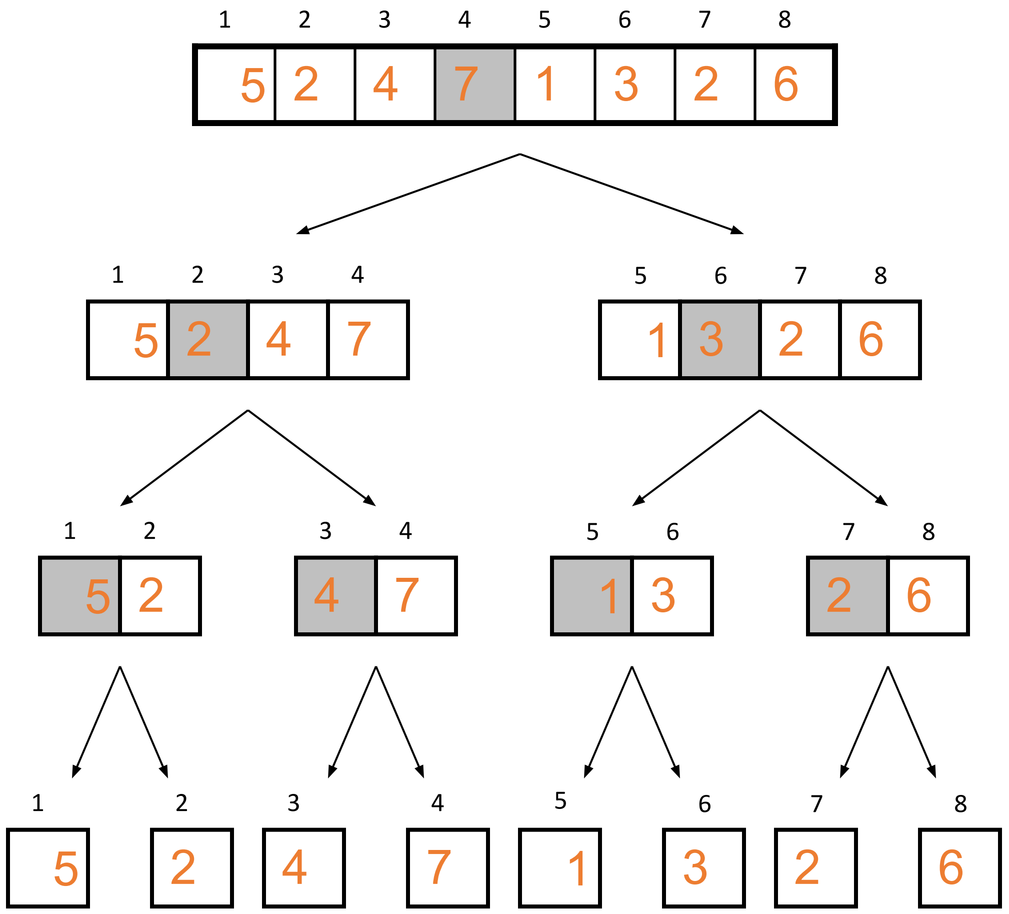
```
Alg.: MERGE-SORT(A, p, r)
    if p < r                                ▷ Check for base case
      then q ← ⌊(p + r)/2⌋                  ▷ Divide
       MERGE-SORT(A, p, q)                  ▷ Conquer
       MERGE-SORT(A, q + 1, r)              ▷ Conquer
       MERGE(A, p, q, r)                    ▷ Combine
```

- Initial call: MERGE-SORT(A, 1, n)



Example – n Power of 2

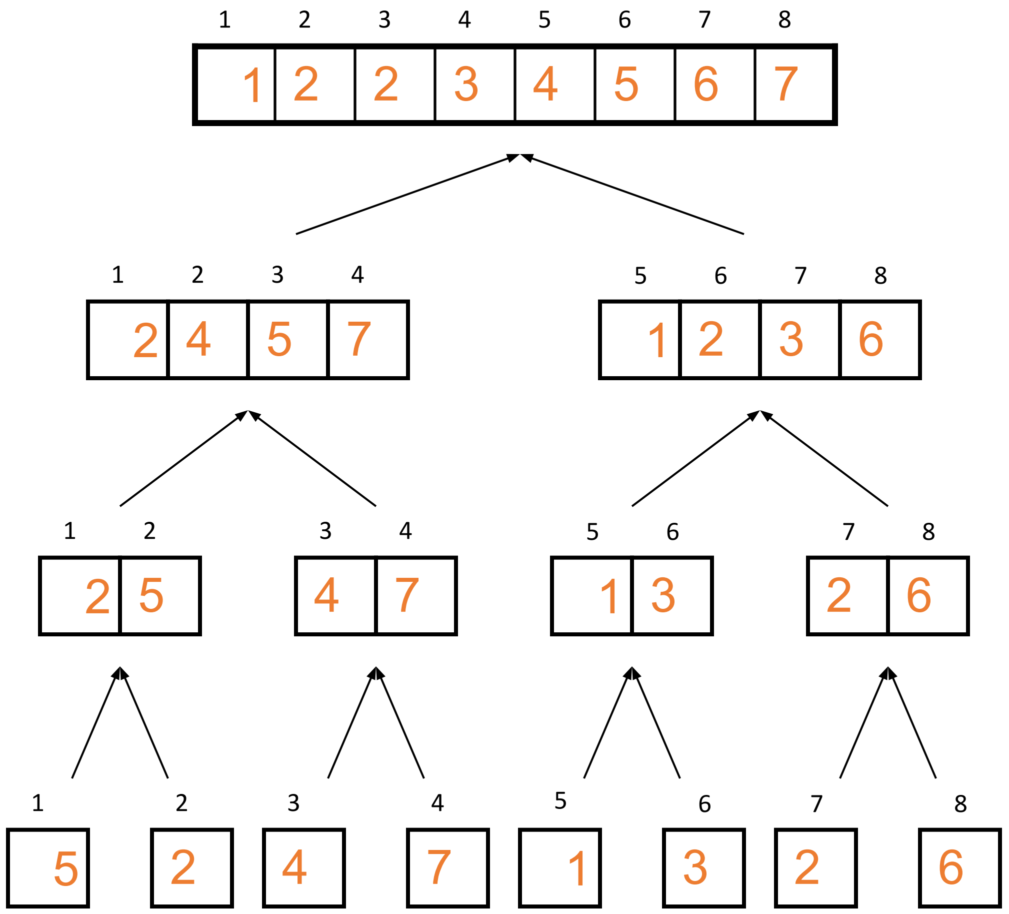
Divide



q = 4

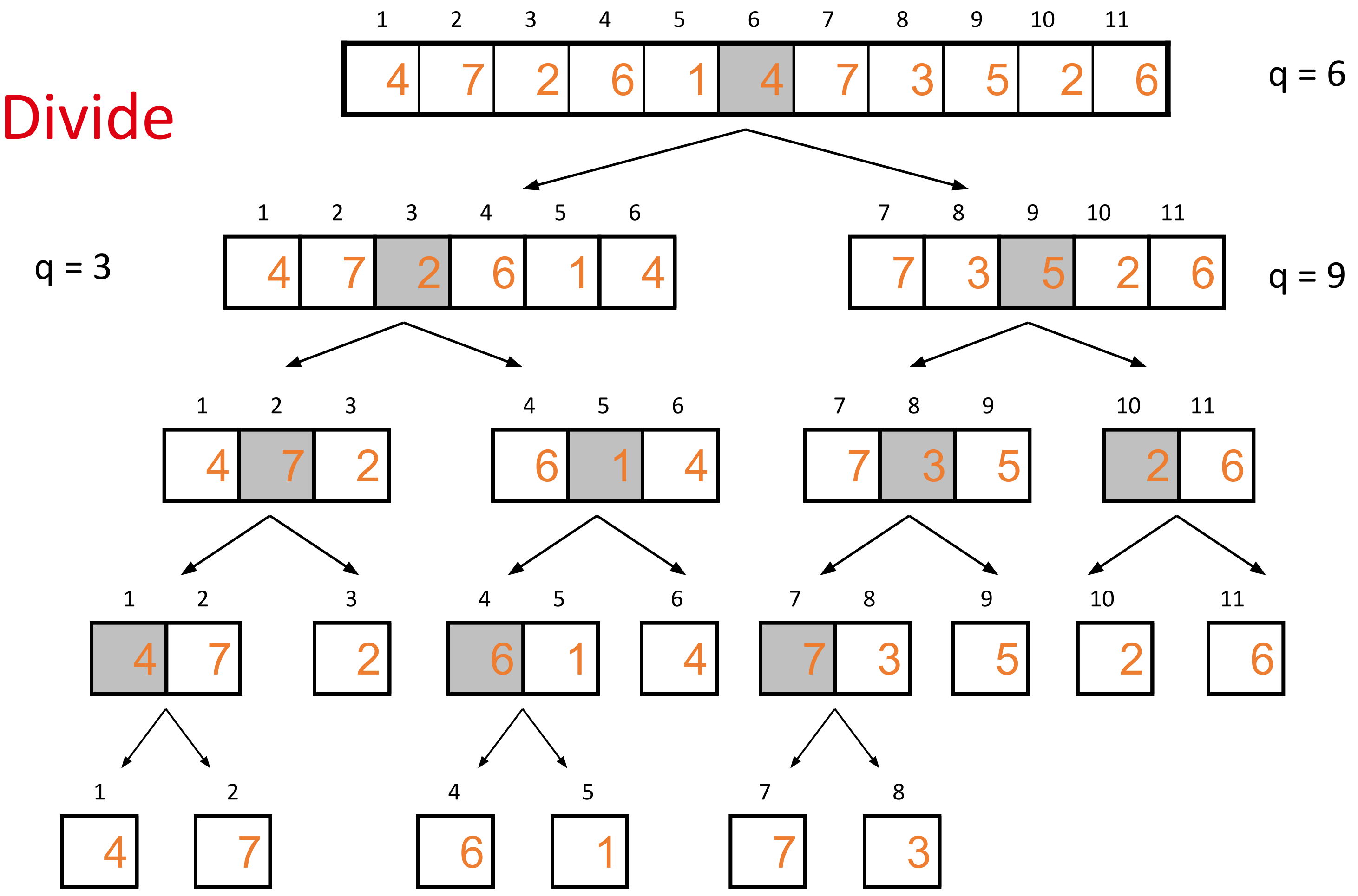
Example – n Power of 2

Conquer
and
Merge



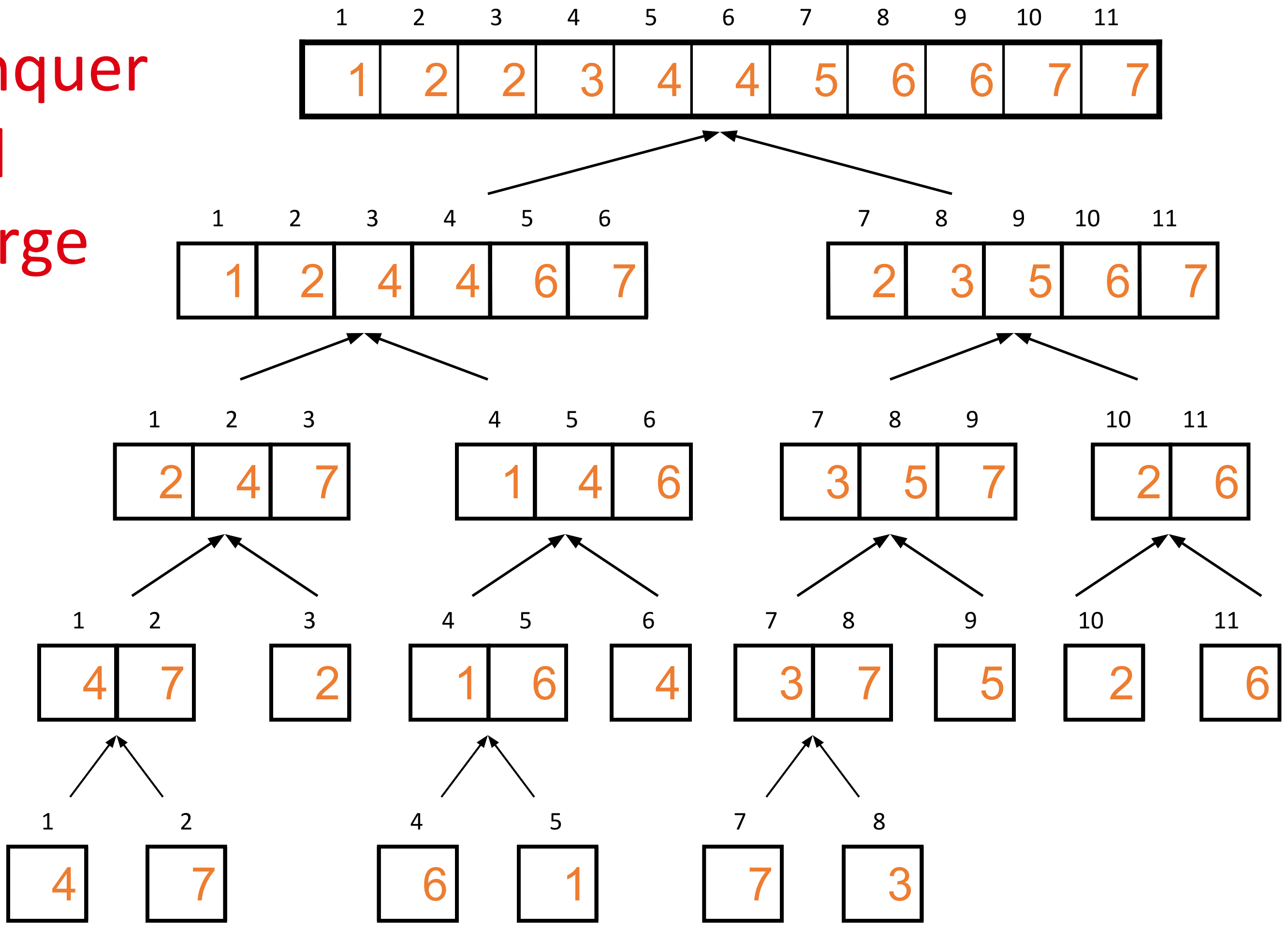
Example – n Not a Power of 2

Divide

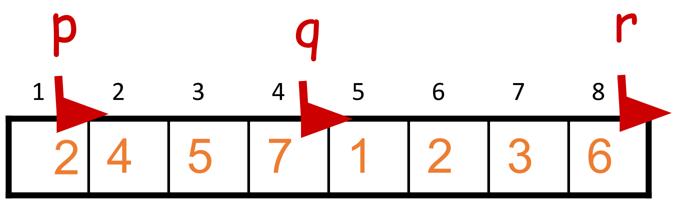


Example – *n* Not a Power of 2

Conquer and Merge



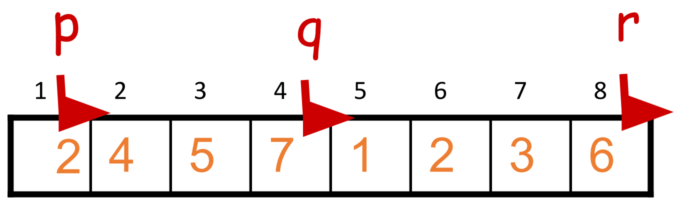
Merging

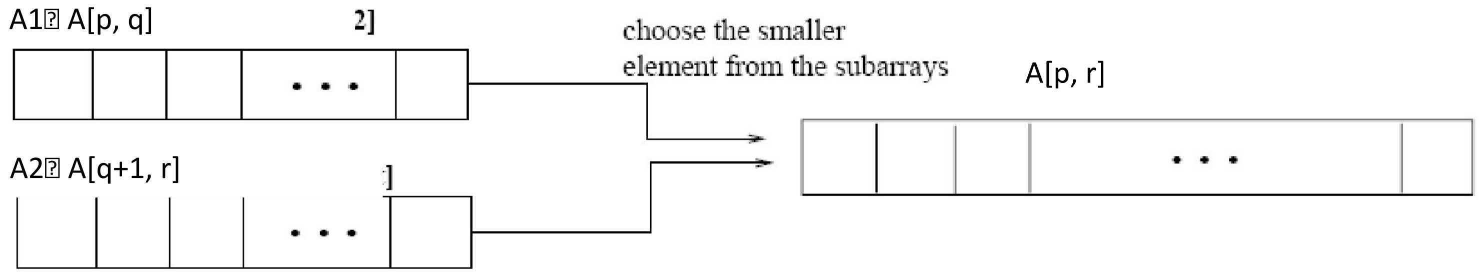


- **Input:** Array A and indices p, q, r such that $p \le q < r$
  - Subarrays $A[p\ .\ .\ q]$ and $A[q + 1\ .\ .\ r]$ are sorted
- **Output:** One single sorted subarray $A[p\ .\ .\ r]$

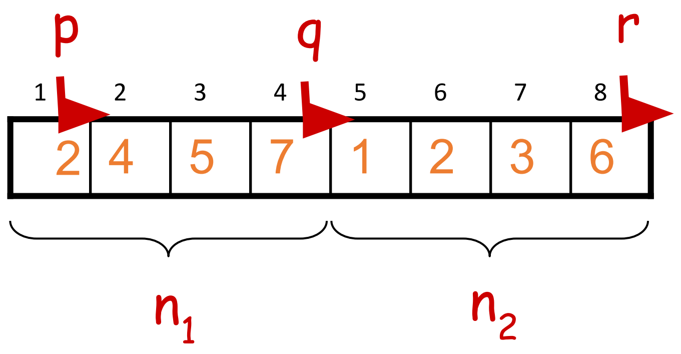
Merging

- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile

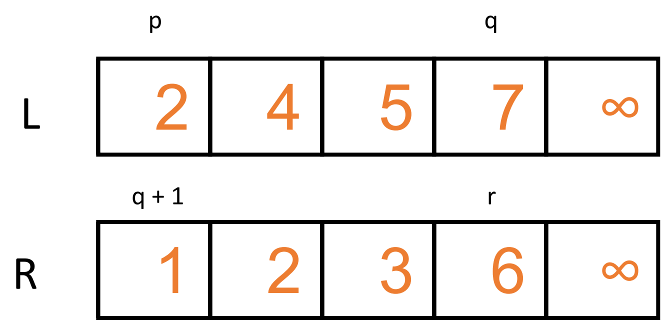




Merge - Pseudocode



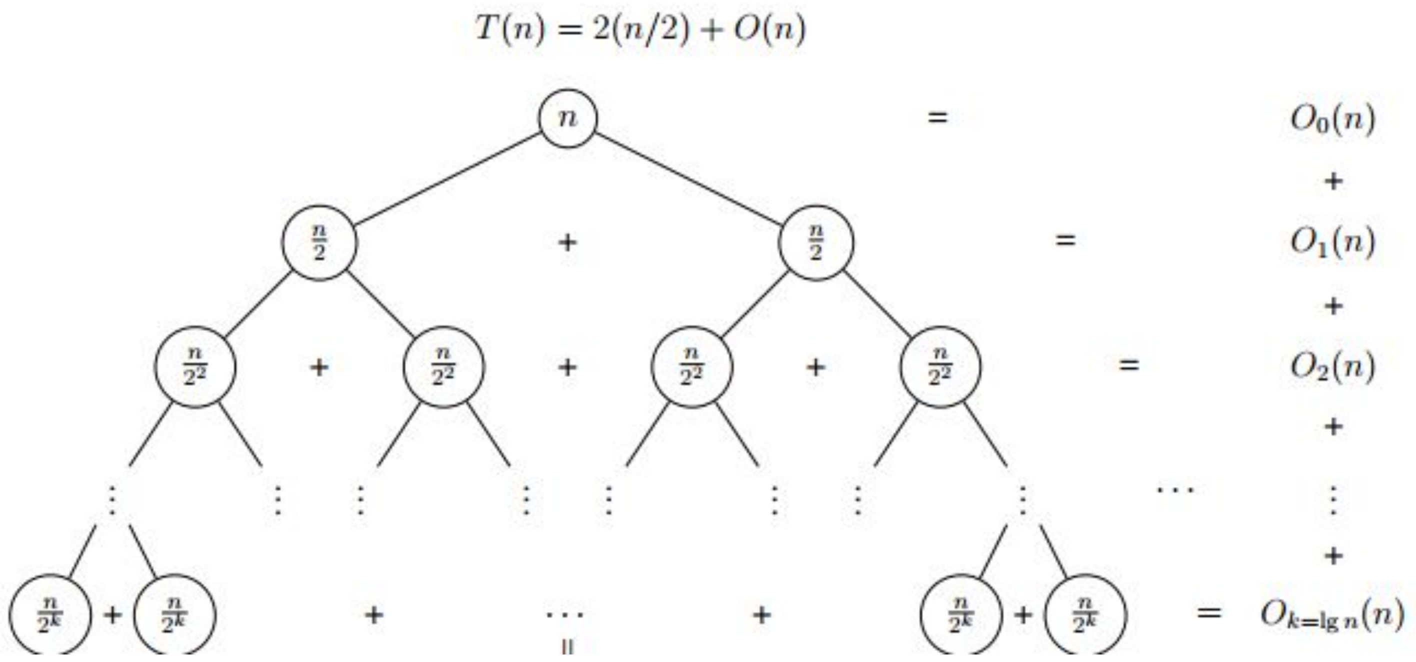
```
Alg.: MERGE(A, p, q, r)
1.  Compute $n_1$ and $n_2$
2.  Copy the first $n_1$ elements into $L[1 \,.\,.\, n_1 + 1]$ and the next $n_2$
    elements into $R[1 \,.\,.\, n_2 + 1]$
3.  $L[n_1 + 1] \leftarrow \infty$;    $R[n_2 + 1] \leftarrow \infty$
4.  $i \leftarrow 1$;    $j \leftarrow 1$
5.  for $k \leftarrow p$ to $r$
6.      do if $L[i] \le R[j]$
7.          then $A[k] \leftarrow L[i]$
8.               $i \leftarrow i + 1$
9.          else $A[k] \leftarrow R[j]$
10.              $j \leftarrow j + 1$
```



MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow$ $C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$



Best Case: O(n log n)
Average Case: O(n log n)
Worst Case: O(n log n)

Advantages

- It can be applied to files of any size.
- Reading of the input during the run –creation step is sequential.
- If heap sort is used for the in-memory part of the merge ,its operation can be overlapped.

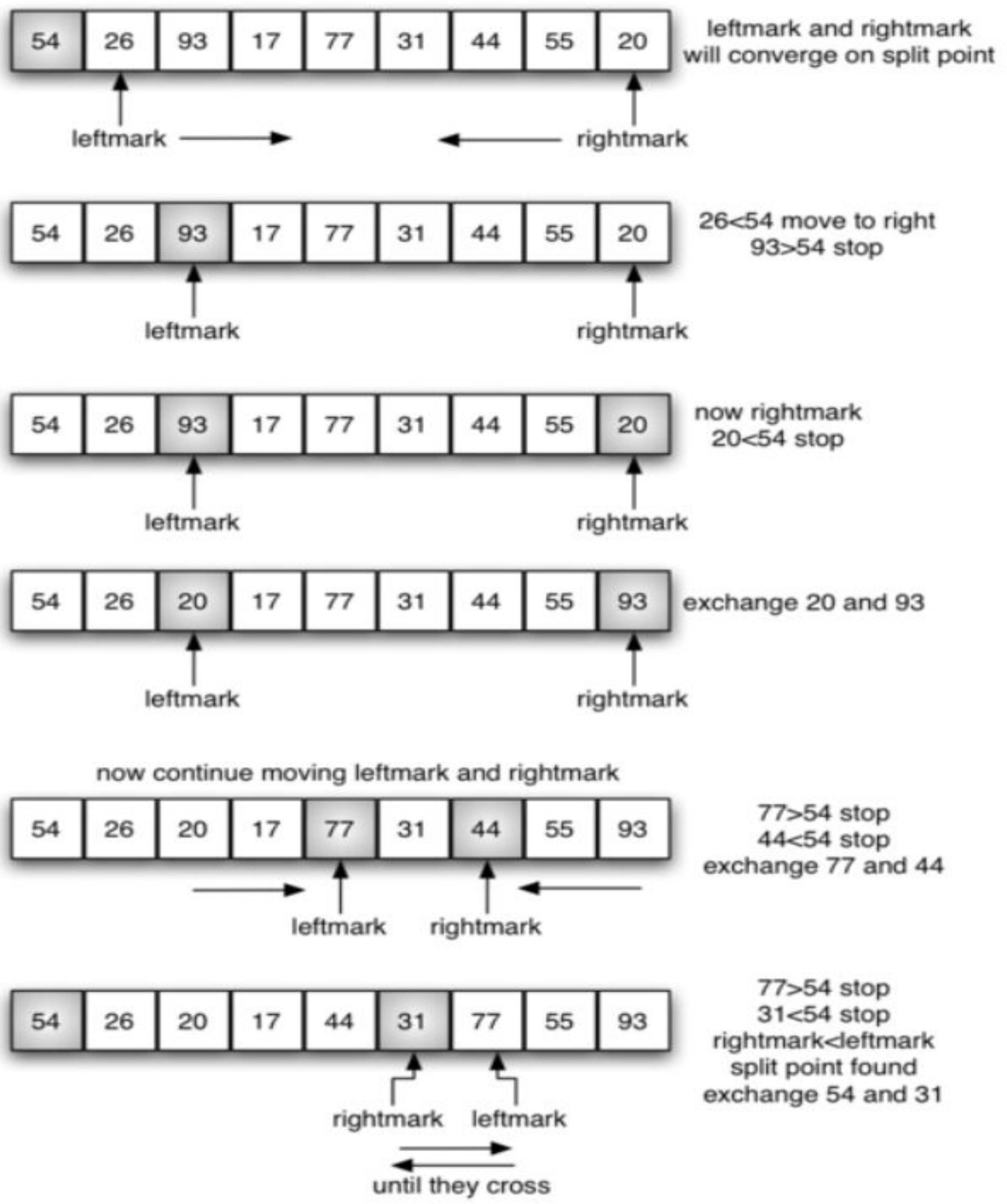
Disadvantages

- It requires extra space.
- Merge sort requires more space than other sort.

Quick Sort

The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.



54 26 93 17 77 31 44 55 20
leftmark and rightmark will converge on split point
leftmark
rightmark
54 26 93 17 77 31 44 55 20
26<54 move to right
93>54 stop
leftmark
rightmark
54 26 93 17 77 31 44 55 20
now rightmark
20<54 stop
leftmark
rightmark
54 26 20 17 77 31 44 55 93
exchange 20 and 93
leftmark
rightmark
now continue moving leftmark and rightmark
54 26 20 17 77 31 44 55 93
77>54 stop
44<54 stop
exchange 77 and 44
leftmark
rightmark
54 26 20 17 44 31 77 55 93
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
rightmark
leftmark
until they cross

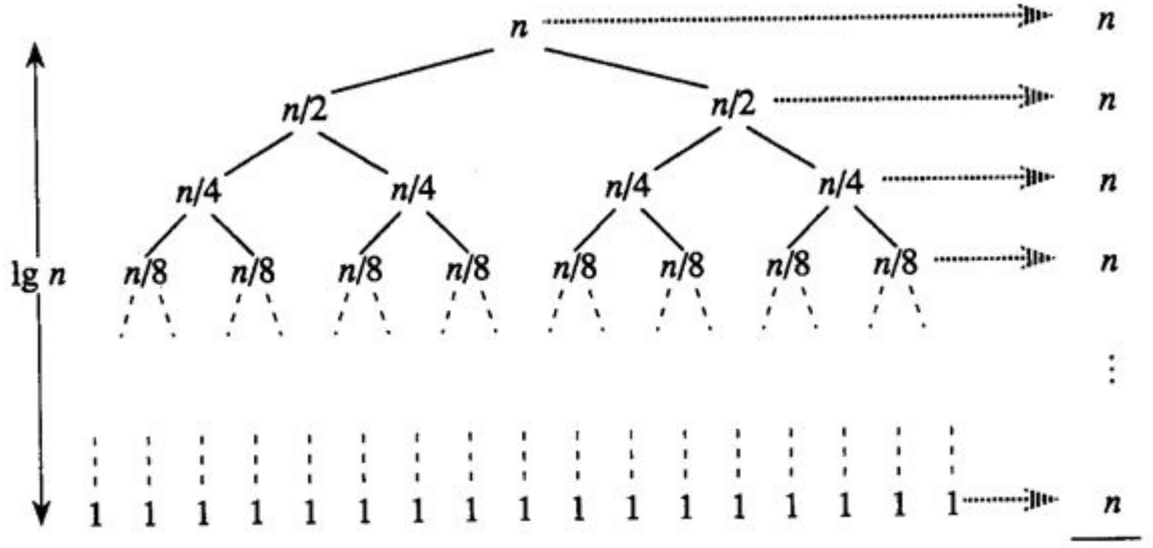
```
Alg.: QUICKSORT($A$, p, r)

  if p < r

    then q ← PARTITION($A$, p, r)

        QUICKSORT ($A$, p, q)

        QUICKSORT ($A$, q+1, r)
```

```
Alg. PARTITION (A, p, r)
1.    $x \leftarrow A[p]$
2.    $i \leftarrow p - 1$
3.    $j \leftarrow r + 1$
4.    while TRUE
5.          do repeat $j \leftarrow j - 1$
6.                until $A[j] \le x$
7.          do  repeat $i \leftarrow i + 1$
8.                until $A[i] \ge x$
9.          if $i < j$
10.               then exchange $A[i] \leftrightarrow A[j]$
11.         else return j
```

Quick Sort Time complexity Analysis for Best and Average Cases

Best Case:

The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.



- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2

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

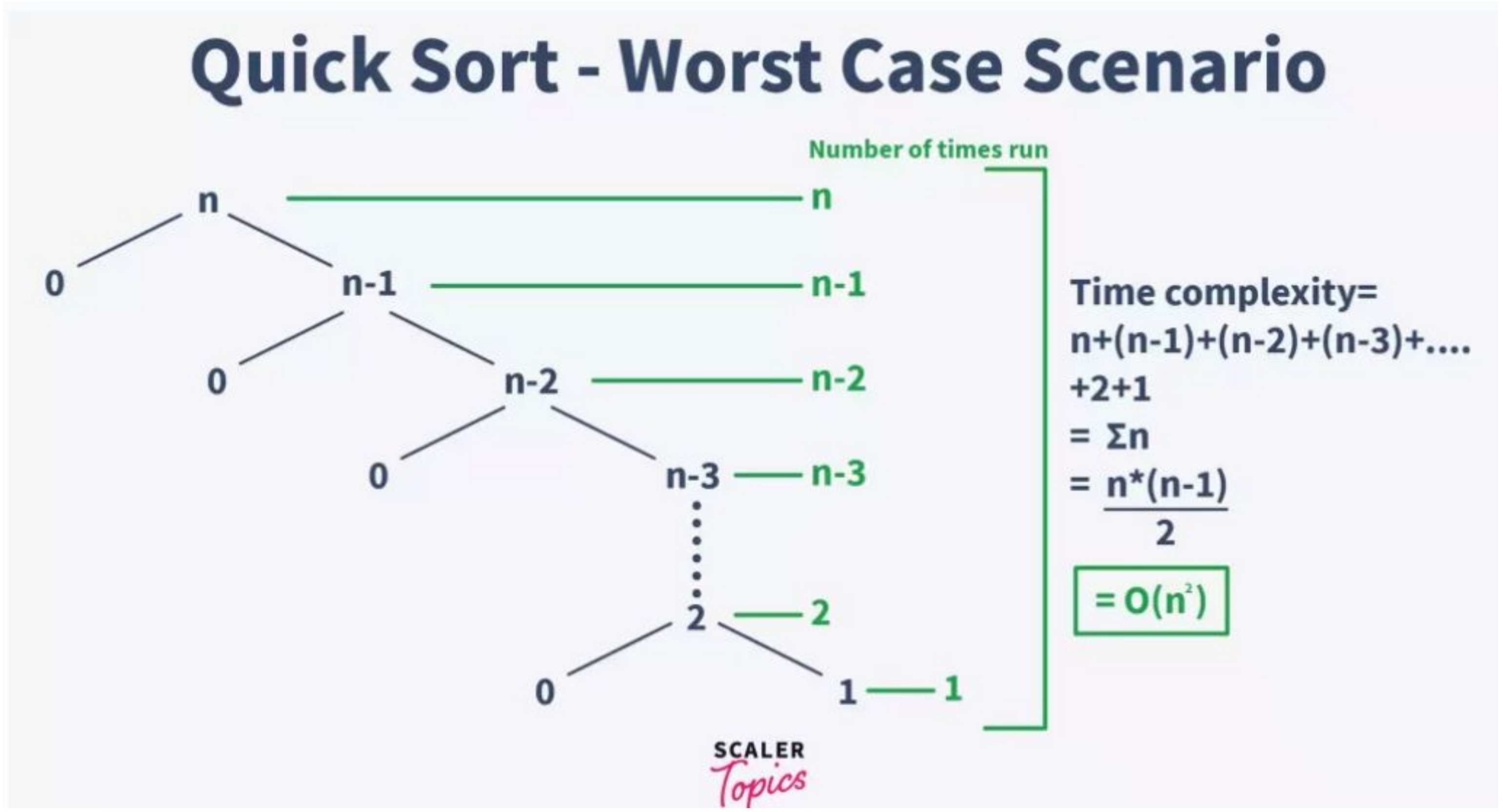
$$T(n) = O(n\lg n)$$

Best and Avg Time complexity is : O(nlogn)

Quick Sort Time complexity for Worst Case

Worst Case:

The worst case occurs when the partition process always picks the first or last element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is the recurrence for the worst case.



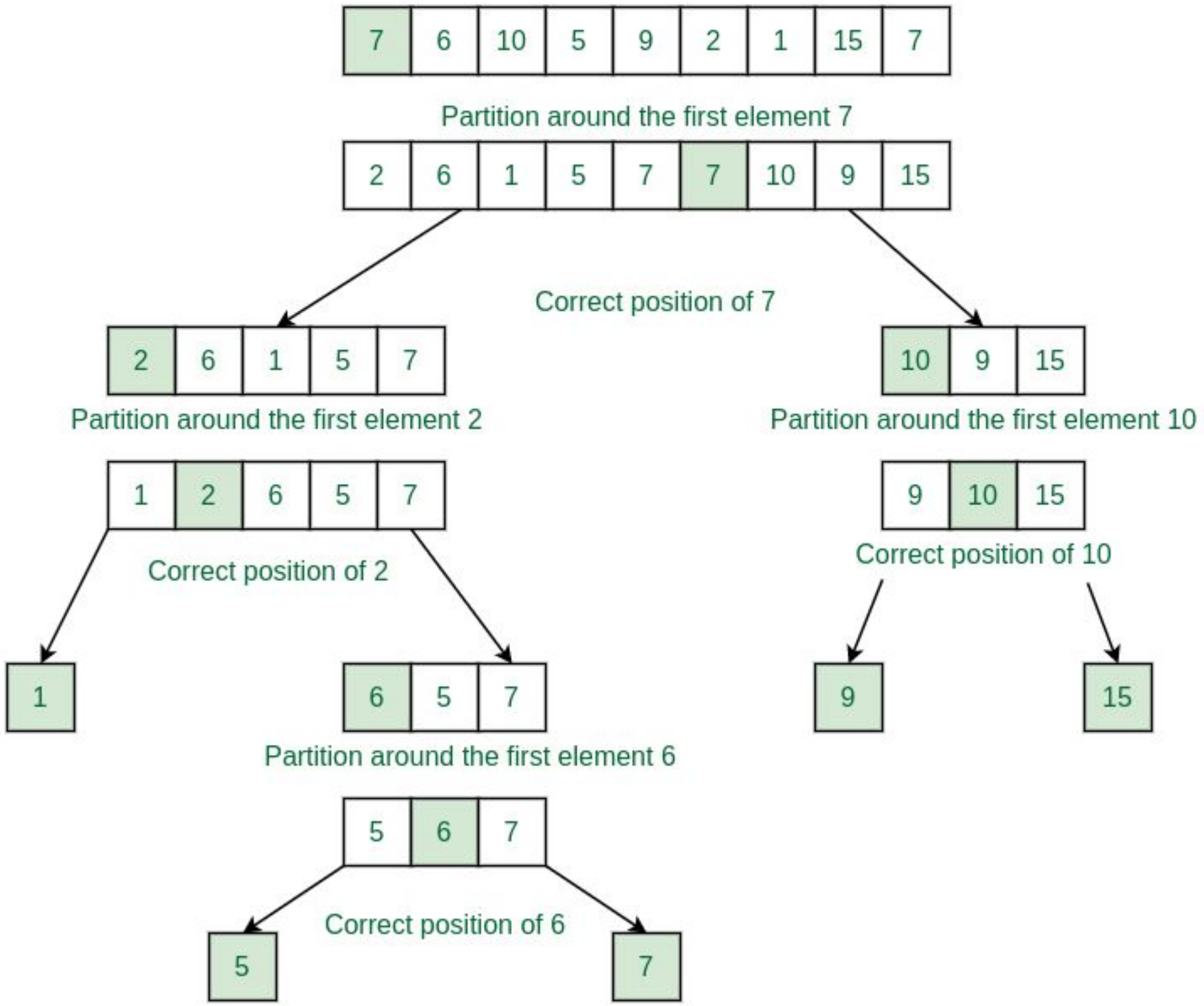
QuickSort Advantages and Disadvantages

Let’s cover a few key advantages of using quicksort:

- It works rapidly and effectively.
- It has the best time complexity when compared to other sorting algorithms.
- QuickSort has a space complexity of `O(logn)`, making it an excellent choice for situations when space is limited.

Despite being the fastest algorithm, quicksort has a few drawbacks. Let’s look at some of the drawbacks of quicksort.

- This sorting technique is considered unstable since it doesn’t maintain the key-value pairs initial order.
- It isn’t as effective when the pivot element is the largest or smallest, or when all of the components have the same size. The performance of the quicksort is significantly impacted by these worst-case scenarios.
- It’s difficult to implement since it’s a recursive process, especially if recursion isn’t available.


7 6 10 5 9 2 1 15 7
Partition around the first element 7
2 6 1 5 7 7 10 9 15
Correct position of 7
2 6 1 5 7
Partition around the first element 2
10 9 15
Partition around the first element 10
1 2 6 5 7
Correct position of 2
9 10 15
Correct position of 10
1
6 5 7
Partition around the first element 6
9
15
5 6 7
Correct position of 6
5
7

0	1	2	3	4	5	6
E	X	A	M	P	L	E

0	1	2	3	4	5	6
$\mathbf{E}$	$\overset{i}{X}$	A	M	P	L	$\overset{j}{E}$
$\mathbf{E}$	E	$\overset{j}{A}$	$\overset{i}{M}$	P	L	X
A	E	$\mathbf{E}$	M	P	L	X
$\mathbf{A}$	$\overset{ij}{E}$					
$\overset{j}{\mathbf{A}}$	$\overset{i}{E}$					
$\mathbf{A}$	E					
	E					
			$\mathbf{M}$	$\overset{i}{P}$	L	$\overset{j}{X}$
			$\mathbf{M}$	$\overset{i}{P}$	$\overset{j}{L}$	X
			$\mathbf{M}$	$\overset{i}{L}$	$\overset{j}{P}$	X
			$\mathbf{M}$	$\overset{j}{L}$	$\overset{i}{P}$	X
			L	$\mathbf{M}$	P	X
			L			
					$\mathbf{P}$	$\overset{ij}{X}$
					$\overset{j}{\mathbf{P}}$	$\overset{i}{X}$
					$\mathbf{P}$	X
						X