University of Global Village (UGV), Barishal

Content of the Theory Course

University Student (UGV)

Course Title: Algorithm Design

Course Code: CSE-2102

Program: Bachelor of Science in Computer Science & Engineering (CSE)

Course Code: CSE-2102
Name of Course Title: Algorithm Design
Course Type: Core Course
Level: 3rd Semester

Name(s) of Academic Course teacher(s):

Sohag Mollik
Lecturer,
CSE at UGV
Mobile: 01304142395
E-mail: sohag.cse.just@gmail.com

ALGORITHM DESIGN

Course Code: CSE-2102	**Credits:** 2
Exam hours: 3	**CIE Marks:** 90
	SEE Marks: 60

Course Learning Outcome (CLOs): After Completing this course successfully, the student will be able to…

CLO1	Describe various algorithms and data structures and analyse their time and space complexity.
CLO2	Understand the principles of algorithm design, such as divide-and-conquer, dynamic programming, and greedy algorithms.
CLO3	Create efficient algorithms to solve problems of varying complexity and design data structures to support them.
CLO4	Apply algorithmic techniques to real-world problems and evaluate their effectiveness in practice.
CLO5	Identify NP-complete problems and explain their significance in the theory of algorithms, as well as explore possible approaches to solving them.

SUMMARY OF COURSE CONTENT:

Sl. No.	COURSE CONTENT	HRs	CLOs
1	Introduction to algorithms: Definition of algorithms, algorithm analysis, asymptotic notation, worst-case and average-case analysis, algorithm design techniques, and problem-solving strategies.	6	CLO1 & CLO2
2	Sorting algorithms: Insertion sort, selection sort, bubble sort, quicksort, merge sort, heap sort, radix sort, and comparison of sorting algorithms in terms of time and space complexity.	6	CLO3
3	Data structures and algorithms: Trees, binary search trees, AVL trees, B-trees, hash tables, priority queues, heaps, graph representations, graph algorithms such as depth-first search, breadth-first search, shortest path algorithms, and minimum spanning tree algorithms.	6	CLO4
4	Dynamic programming: The principle of optimality, the Bellman-Ford algorithm, the Floyd-Warshall algorithm, the Knapsack problem, and the Longest Common Subsequence problem.	6	CLO4
5	Greedy algorithms: The activity selection problem, the Huffman coding problem, the minimum spanning tree problem, and the shortest path problem.	6	CLO4
6	Divide-and-conquer algorithms: The maximum subarray problem, the closest pair of points problems, and the fast Fourier transform algorithm.	6	CLO5
7	NP-completeness: The P versus NP problem, polynomial-time reductions, Cook's theorem, and examples of NP-complete problems such as the traveling salesman problem, the satisfiability problem, and the graph coloring problem.	6	CLO5

Recommended Books:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. "Algorithm Design" by Jon Kleinberg and Éva Tardos
3. "The Design and Analysis of Algorithms" by Dexter C. Kozen

ASSESSMENT PATTERN

CIE- Continuous Internal Evaluation (90 Marks)

Bloom's Category Marks (out of 90)	Tests (45)	Assignments (15)	Quizzes (15)	Attendance (15)
Remember	5	03		
Understand	5	04	05	
Apply	15	05	05	
Analyze	10			
Evaluate	5	03	05	
Create	5			

SEE- Semester End Examination (60 Marks)

Bloom's Category	Test
Remember	7
Understand	7
Apply	20
Analyze	15
Evaluate	6
Create	5

COURSE PLAN

Week No	Topics	Teaching Learning Strategy(s)	Assessment Strategy(s)	Alignment to CLO
1	Introduction to algorithms: Describe various algorithms & data structure, asymptotic notation.	Lecture, PowerPoint presentation, Q&A, in-class discussions.	Feedback, Q&A, assessment	CLO1
2	Discuss worst, best & average case, time & space complexity, problem solving strategies.	Lecture, PowerPoint presentation, Q&A, in-class discussions.	Quizzes, Assignments	CLO2
3	Sorting algorithms: Insertion sort, selection sort.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment	CLO3
4	Sorting algorithms: Bubble sort, quick sort.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment	CLO3
5	Sorting algorithms: Merge sort, Radix sort.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task.	CLO3
6	Data structures and algorithms: Trees, Tree Traversal, Heap Sort, BST.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task, Assignment.	CLO4
7	Data structures and algorithms: Graph representation & types, BFS.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task, Assignment.	CLO4

COURSE PLAN

Week No	Topics	Teaching Learning Strategy(s)	Assessment Strategy(s)	Alignment to CLO
8	Data structures and algorithms: DFS & Shortest path algorithm.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task.	CLO4
9	Data structures and algorithms: MST & AVL tree	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Quizzes, Feedback, Q&A, assessment, Problem solving task. Assignments.	CLO4
10	Dynamic programming: Recursion, Introduction to DP.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task. Assignments.	CLO4
11	Dynamic programming: Tabulation vs Memorization DP, Bellman ford algorithm.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task. Assignments.	CLO4
12	Dynamic programming: Floyd Warshall algorithm, knapsack 0/1.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task. Assignments.	CLO4
13	Dynamic programming: Knapsack fractional, LCS.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Quizzes, Feedback, Q&A, assessment, Problem solving task, Assignment.	CLO4
14	Greedy algorithms: The activity selection problem, the Huffman coding problem.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task, Assignment.	CLO4

COURSE PLAN

Week No	Topics	Teaching Learning Strategy(s)	Assessment Strategy(s)	Alignment to CLO
15	Divide-and-conquer algorithms: Maximum subarray problem, find minimum element.	Lecture, PowerPoint presentation, Live coding(C&C++), Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task.	CLO5
16	Find minimum element using divided & conquer algorithm, NP hard problem.	Lecture, PowerPoint presentation, Q&A, in-class discussions.	Quizzes, Feedback, Q&A, assessment, Problem solving task. Assignments.	CLO5
17	NP-completeness: Cook's theorem, graph coloring problem.	Lecture, PowerPoint presentation, Q&A, in-class discussions.	Feedback, Q&A, assessment, Problem solving task. Assignments.	CLO5

WEEK 1

INTRODUCTION TO ALGORITHMS

Page
11-18

DEFINITION OF ALGORITHM

An algorithm is a finite sequence of unambiguous instruction for solving a particular problem. It must satisfy the specific characteristics.

Characteristics of an algorithm

All algorithm satisfy the following criteria.

Input: Zero/more quantities are externally supplied.
Output: At least one quantity is produced.
Definiteness: Each instruction is clear and unambiguous.
Finiteness: If the instruction of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.
Efficiency: Every instruction must be very basic and runs in short time with effective results better than human computations.

ASYMPTOTIC NOTATION

Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.

There are mainly three asymptotic notations:

1. Big-O Notation (O-notation)
2. Omega Notation (Ω-notation)
3. Theta Notation (Θ-notation)

- **Big-Oh** is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function f(n)).
- Let **f(n)** and **g(n)** be functions that map positive integers to positive real numbers. We say that **f(n)** is **O(g(n))** or $\mathbf{f(n)} \in \mathbf{O(g(n))}$, if there exists a real constant $\mathbf{c} > \mathbf{0}$ and there exists an integer constant $\mathbf{n_0} \geq \mathbf{1}$ such that $\mathbf{f(n)} \leq \mathbf{cg(n)}$ for every integer $\mathbf{n} \geq \mathbf{n_0}$.
- In other words $\mathbf{O(g(n))} = \{\mathbf{f(n)}$: there exist positive constants **c** and $\mathbf{n_0}$ such that $\mathbf{0} \leq \mathbf{f(n)} \leq \mathbf{cg(n)}$ for all $\mathbf{n} \geq \mathbf{n_0}\}$

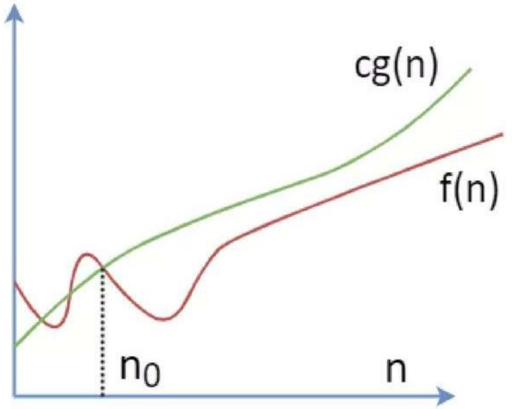


Figure 2.2: $f(n) \in O(g(n))$

O (Big-Oh) notation

Question 1: Consider the function f(n) $= 6n + 135$. Clearly. f(n) is non-negative for all integers $n \geq 0$. We wish to show that $f(n)=O(n^2)$. According to the Big-oh definition, in order to show this we need to find an integer n_0, and a constant $c > 0$ such that for all integers, $n \geq n_0$, $f(n) = c(n^2)$

Answer: Suppose we choose $c = 1$, and $f(n) = cn^2$.
$\Rightarrow 6n+135 = cn^2 = n^2$ [Since $c = 1$] $n^2-6n-135 = 0$
$\Rightarrow (n-15)(n+9) = 0$
Since $(n+9) > 0$ for all values $n \geq 0$, we conclude that $(n-15) = 0$
$\Rightarrow n_0 = 15$ for $c = 1$
For $c = 2$, $n_0 = (6 + \sqrt{1116})/4 \approx 9.9$
For $c = 4$, $n_0 = (6 + \sqrt{2196})/8 \approx 6.7$

- **Big-Omega (Ω)** is the tight lower bound notation.
- Let **f(n)** and **g(n)** be functions that map positive integers to positive real numbers. We say that **f(n)** is $\mathbf{\Omega(g(n))}$ or $\mathbf{f(n) \in \Omega(g(n))}$ if there exists a real constant $\mathbf{c > 0}$ and there exists an integer constant $\mathbf{n_0 \geq 1}$ such that $\mathbf{f(n) \geq cg(n)}$ for every integer $\mathbf{n \geq n_0}$.
- In other words $\Omega(\mathbf{g(n)}) = \{\mathbf{f(n)}$: there exist positive constants **c** and $\mathbf{n_0}$ such that $\mathbf{0 \leq cg(n) \leq f(n)}$ for all $\mathbf{n \geq n_0}\}$.

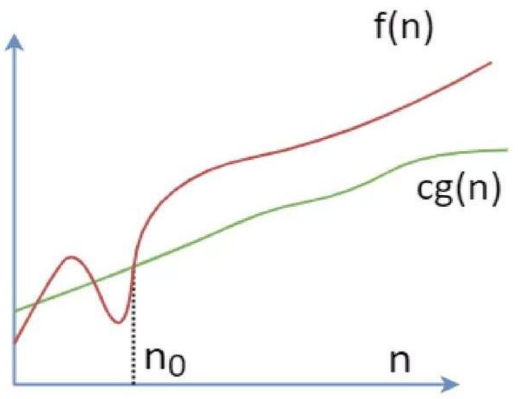


Figure 2.3: $f(n) \in \Omega(g(n))$

Ω (Big-Omega) notation

Question 2: Consider the function f(n)= $3n^2$-24n+72. Clearly f(n) is non-negative for all integers $n \geq 0$. We wish to show that $f(n) = \Omega(n^2)$. According to the big-omega definition, in order to show this we need to find an integer n_0,and a constant $c > 0$ such that for all integers $n = n_0$, $f(n) = cn^2$.

Answer: Suppose we choosc $c = 1$, Then $f(n) = cn^2$
$\Rightarrow 3n^2\text{-}24n+72 = n^2$
$\Rightarrow 2n^2\text{-}24n+72 = 0$
$\Rightarrow 2(n\text{-}6)^2 = 0$
Since $(n\text{-}6)^2 = 0$, we conclude that $n_0 = 6$.
So we have that for $c = 1$ and $n \geq 6$, $f(n) = cn^2$. Hence $f(n) = \Omega(n^2)$.

- Let **f(n)** and **g(n)** be functions that map positive integers to positive real numbers. We say that **f(n)** is $\theta(\mathbf{g(n)})$ or $\mathbf{f(n)} \in \theta(\mathbf{g(n)})$ if and only if $\mathbf{f(n)} \in \mathbf{O(g(n))}$ and $\mathbf{f(n)} \in \Omega(\mathbf{g(n)})$
- $\theta(\mathbf{g(n)}) = \{\mathbf{f(n)}$: there exist positive constants **c1**, **c2** and $\mathbf{n_0}$ such that $\mathbf{0} \leq \mathbf{c1g(n)} \leq \mathbf{f(n)} \leq \mathbf{c2g(n)}$ for all $\mathbf{n} \geq \mathbf{n_0}\}$

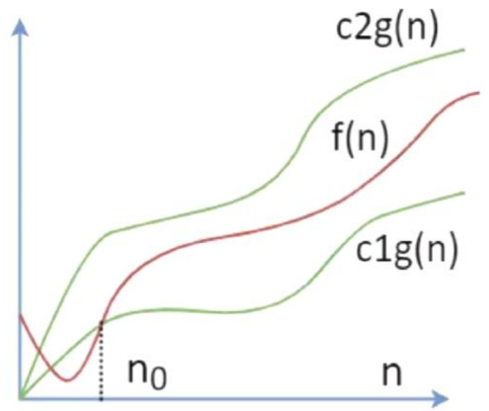


Figure 2.6: $f(n) \in \theta(g(n))$

Self Study

Exercise Practice Problem

Week 2

Worst, best & average case, time & space complexity, problem solving strategies

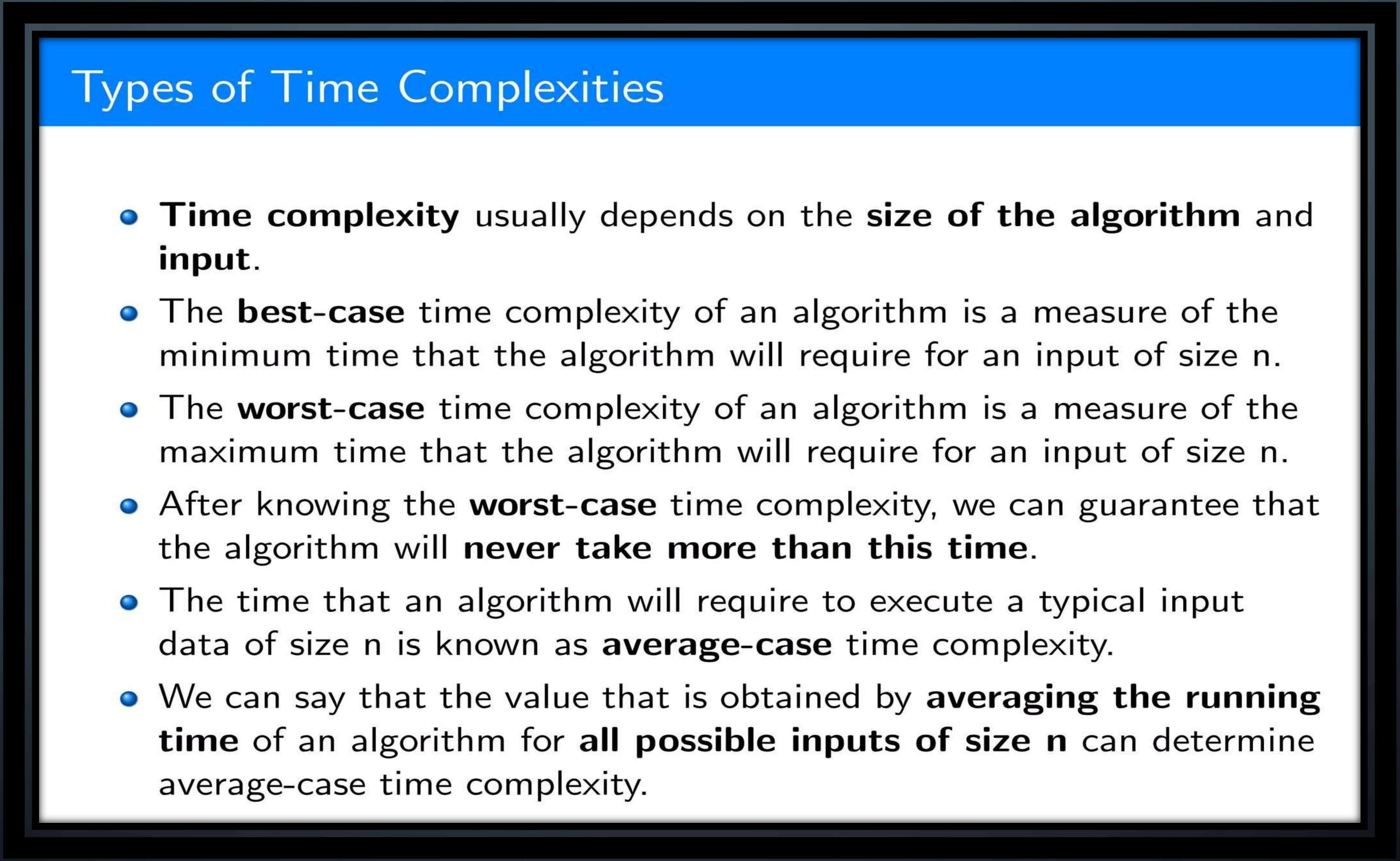
Page
20-29

TIME COMPLEXITY

Time complexity of an algorithm is a quantitative measure of the amount of computational time an algorithm takes to complete as a function of the size of its input. It expresses the relationship between the input size (n) and the number of basic operations (e.g., comparisons, additions) the algorithm performs.

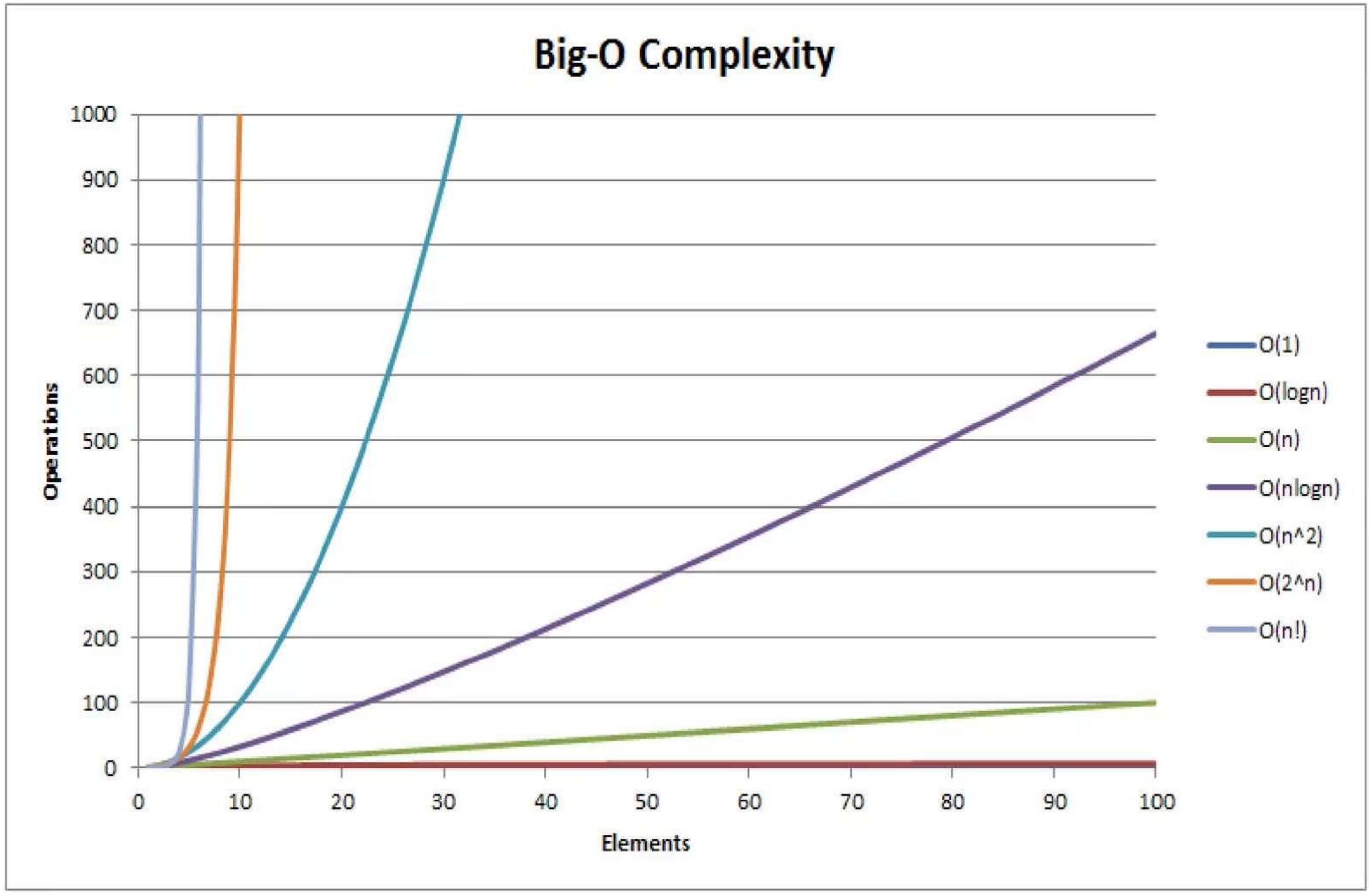
Key Aspects:

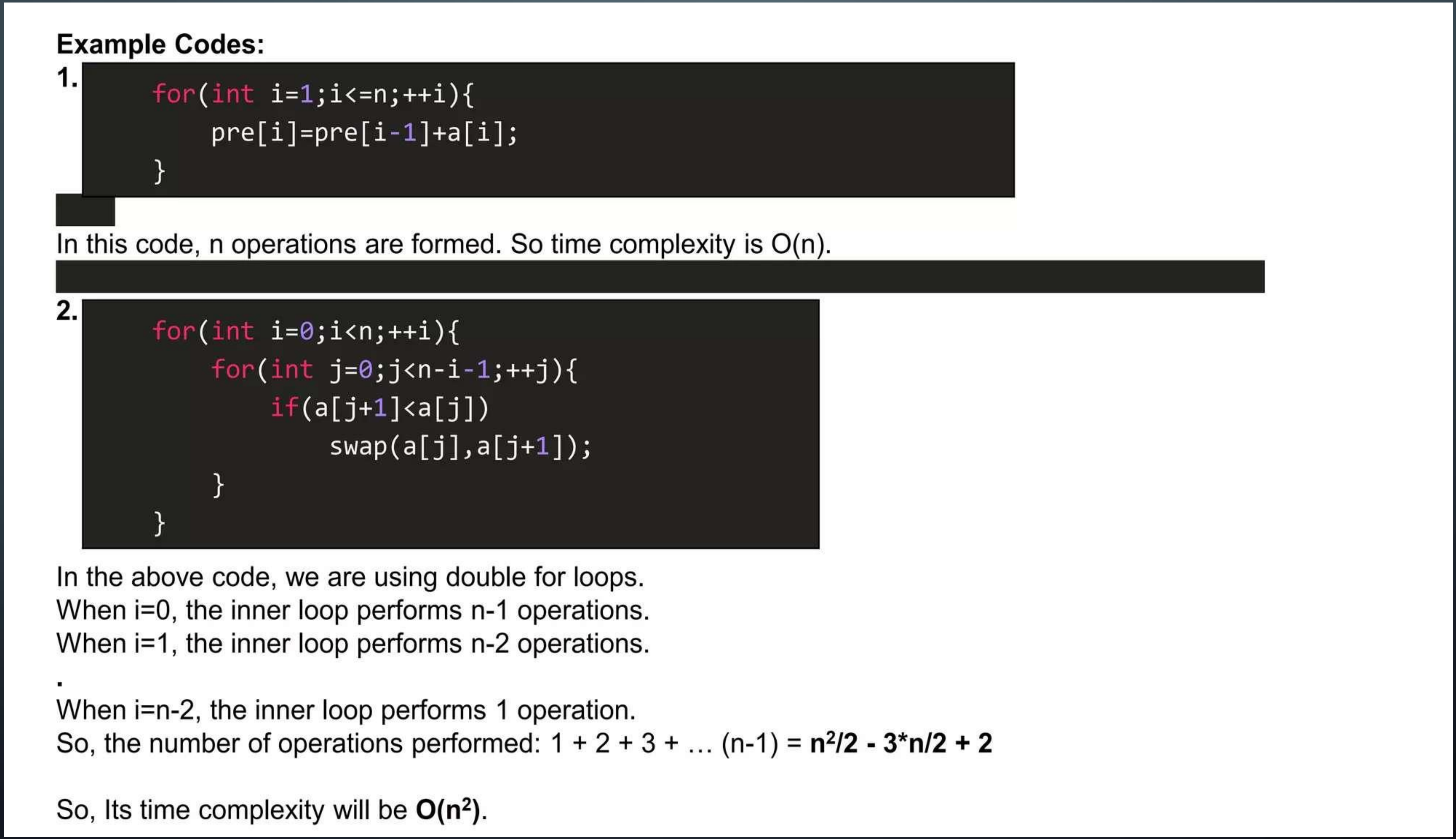
- **Input Size (n):** The size of the input data affects how the runtime grows. Input size could be the number of elements in an array, the number of nodes in a graph, or the length of a string.
- **Basic Operations:** The number of fundamental operations (like comparisons, assignments, arithmetic operations) required to solve a problem.
- **Asymptotic Behavior:** Time complexity is concerned with how the runtime grows as the input size increases, focusing on large input sizes ($n \to \infty$).
- **Expressed Using Big-O Notation:** Time complexity is typically expressed in Big-O notation, which provides an upper bound on the growth rate of the runtime.



Types of Time Complexities

- **Time complexity** usually depends on the **size of the algorithm** and **input**.
- The **best-case** time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size n.
- The **worst-case** time complexity of an algorithm is a measure of the maximum time that the algorithm will require for an input of size n.
- After knowing the **worst-case** time complexity, we can guarantee that the algorithm will **never take more than this time**.
- The time that an algorithm will require to execute a typical input data of size n is known as **average-case** time complexity.
- We can say that the value that is obtained by **averaging the running time** of an algorithm for **all possible inputs of size n** can determine average-case time complexity.


Big-O Complexity
Operations
Elements
0
100
200
300
400
500
600
700
800
900
1000
0
10
20
30
40
50
60
70
80
90
100
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)



Example Codes:

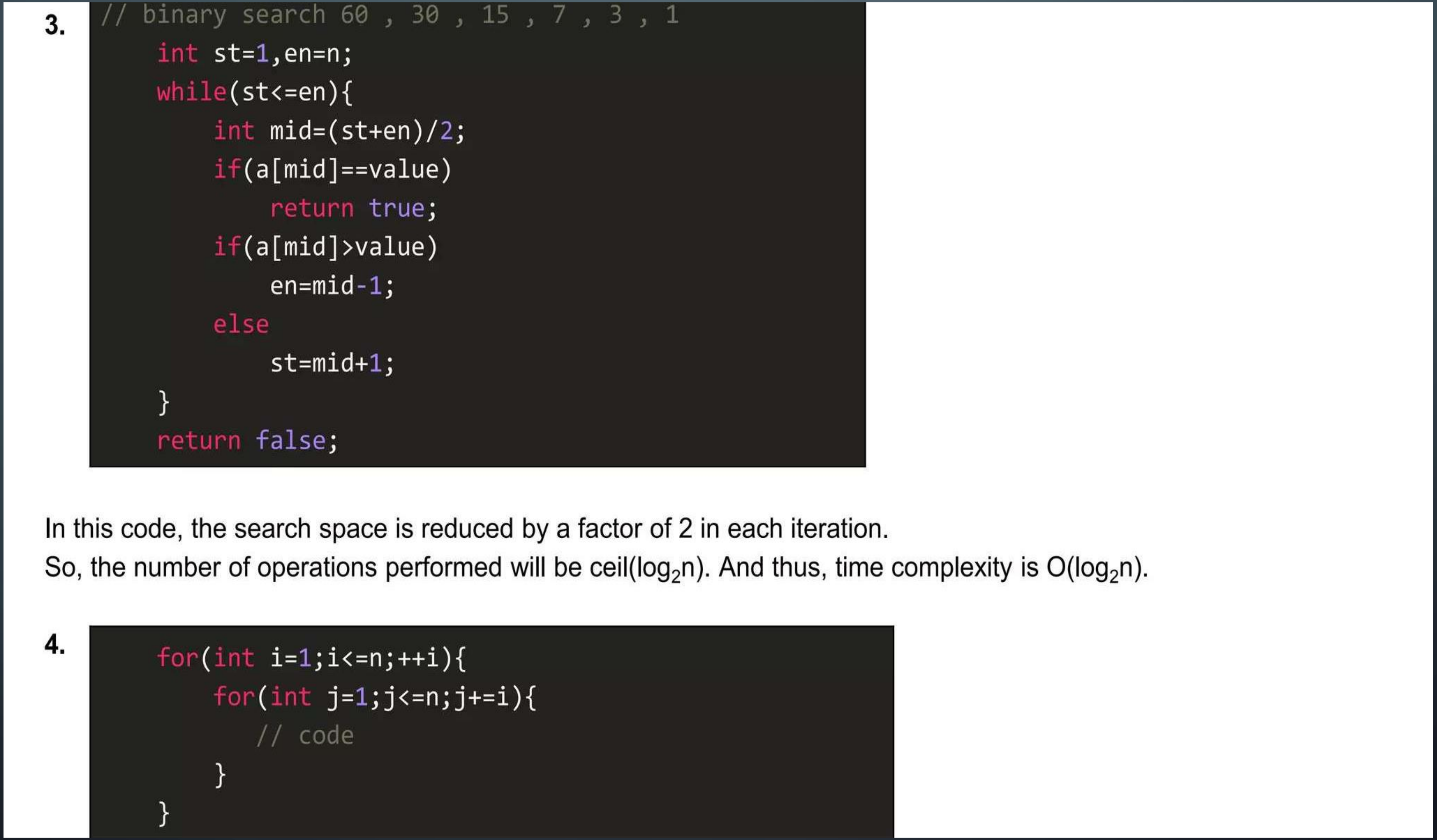
1.
```
    for(int i=1;i<=n;++i){
        pre[i]=pre[i-1]+a[i];
    }
```

In this code, n operations are formed. So time complexity is O(n).

2.
```
    for(int i=0;i<n;++i){
        for(int j=0;j<n-i-1;++j){
            if(a[j+1]<a[j])
                swap(a[j],a[j+1]);
        }
    }
```

In the above code, we are using double for loops.
When i=0, the inner loop performs n-1 operations.
When i=1, the inner loop performs n-2 operations.
.
When i=n-2, the inner loop performs 1 operation.
So, the number of operations performed: 1 + 2 + 3 + … (n-1) = **$n^2/2 - 3*n/2 + 2$**

So, Its time complexity will be **$O(n^2)$**.



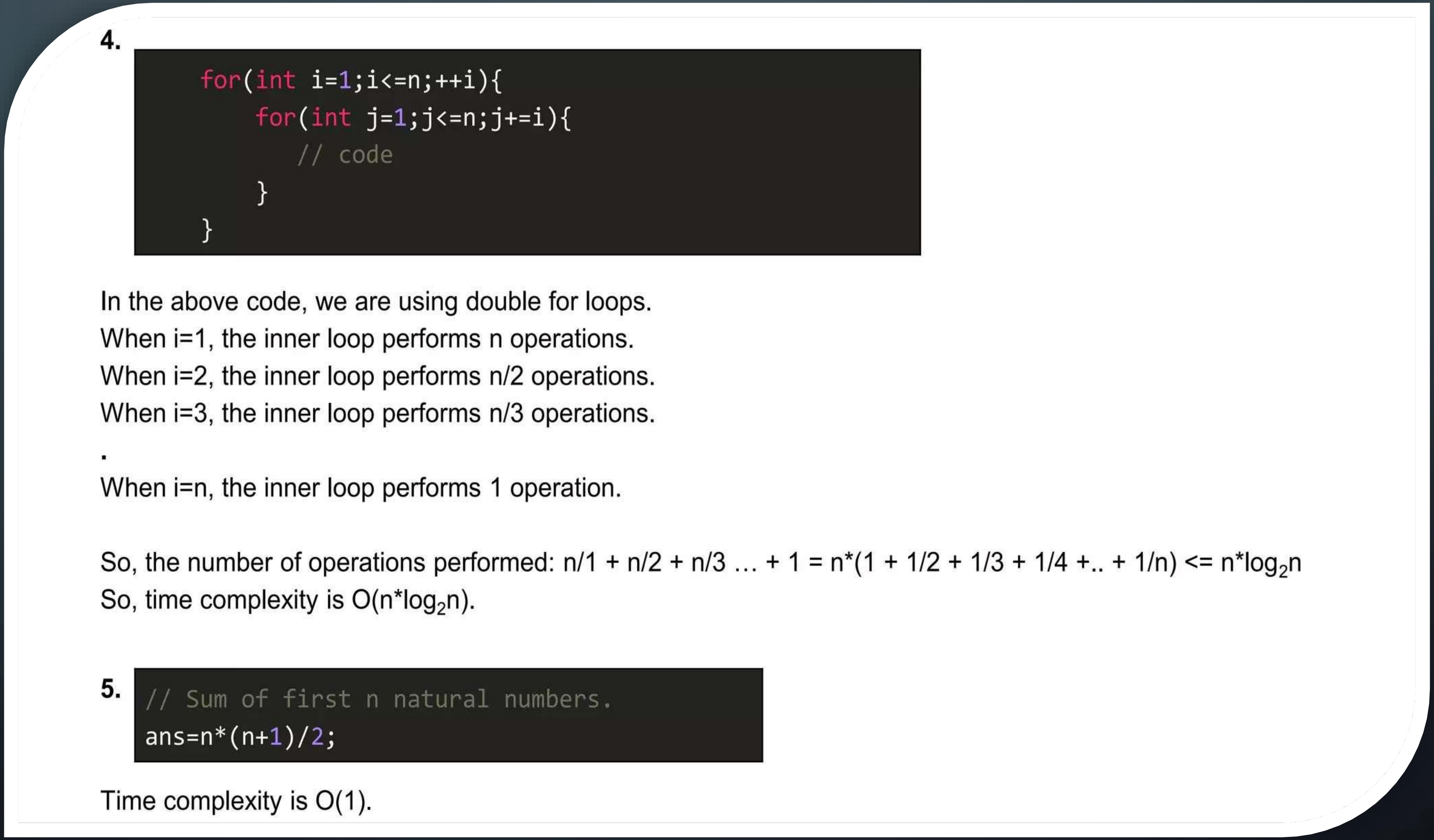
3.

```
// binary search 60 , 30 , 15 , 7 , 3 , 1
    int st=1,en=n;
    while(st<=en){
        int mid=(st+en)/2;
        if(a[mid]==value)
            return true;
        if(a[mid]>value)
            en=mid-1;
        else
            st=mid+1;
    }
    return false;
```

In this code, the search space is reduced by a factor of 2 in each iteration.
So, the number of operations performed will be ceil($\log_2 n$). And thus, time complexity is $O(\log_2 n)$.

4.

```
    for(int i=1;i<=n;++i){
        for(int j=1;j<=n;j+=i){
            // code
        }
    }
```



4.

```
    for(int i=1;i<=n;++i){
        for(int j=1;j<=n;j+=i){
           // code
        }
    }
```

In the above code, we are using double for loops.
When i=1, the inner loop performs n operations.
When i=2, the inner loop performs n/2 operations.
When i=3, the inner loop performs n/3 operations.
.
When i=n, the inner loop performs 1 operation.

So, the number of operations performed: $n/1 + n/2 + n/3 \ldots + 1 = n*(1 + 1/2 + 1/3 + 1/4 +.. + 1/n) <= n*\log_2 n$
So, time complexity is $O(n*\log_2 n)$.

5.

```
// Sum of first n natural numbers.
ans=n*(n+1)/2;
```

Time complexity is O(1).

Estimating the intended time complexity of a problem by looking at its constraints

Most computer processors and online judges can process 10^8 operations per second.

So, we can determine the complexity of the algorithm to be used by observing constraints on input size.

For $n=10^{18}$, $O(1)$ or $O(\log_2 n)$ algorithms may be used.

For $n=10^{10}$, $O(\sqrt{n})$ algorithms may be used.

For $n=10^7$, $O(n)$ algorithms may be used.

For $n=10^6$, $O(n)$ or $O(n*\log_2 n)$ algorithms may be used.

For $n=10^5$, $O(n)$, $O(n*\sqrt{n})$, $O(n*\log_2 n)$ or $O(n*\log_2 n*\log_2 n)$ algorithms may be used.

For $n=5000$, $O(n^2)$ algorithms may be used.

For $n=2000$, $O(n^2)$ or $O(n^2 * \log_2 n)$ algorithms may be used.

For $n=500$, $O(n^3)$ algorithms may be used.

For $n=100$, $O(n^4)$ algorithms may be used.

For $n=40$, $O(n*2^{n/2})$ algorithms may be used.(using meet-in-middle)

For $n=24$, $O(2^n)$ algorithms may be used.

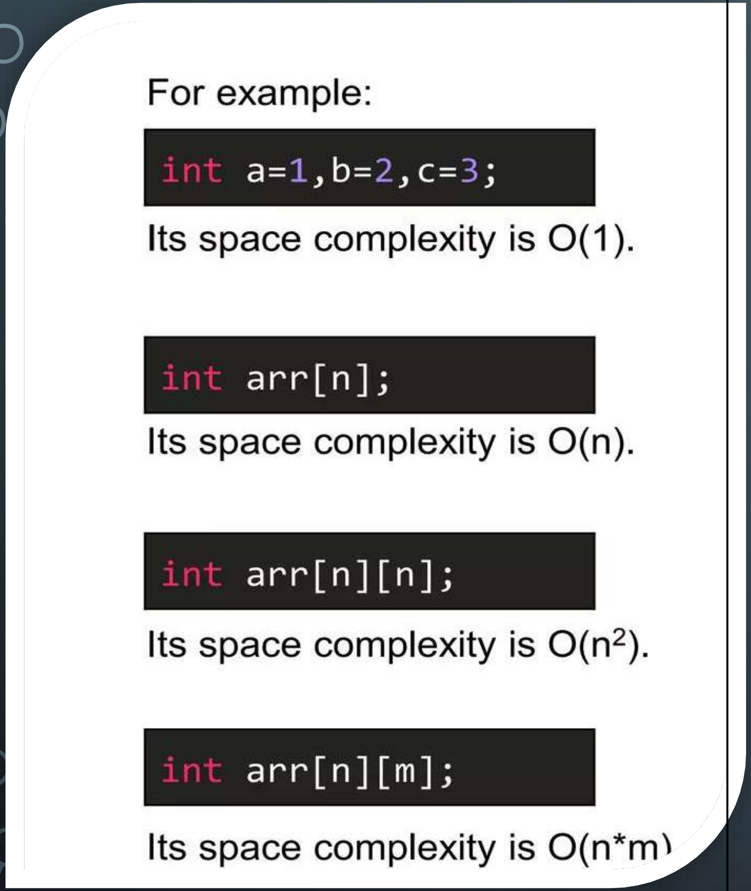
For $n=18$, $O(n*2^n)$ algorithms may be used.

SPACE COMPLEXITY

Space complexity of an algorithm is a measure of the total amount of memory required by the algorithm to run to completion as a function of the size of its input. It includes both the memory used for the input, temporary storage, and auxiliary data structures.

- **Key Components:**

1. **Fixed Part:**
   Memory required by the algorithm that does not depend on the size of the input. This includes:
   1. Constants
   2. Instruction space (code)
   3. Static variables
2. **Variable Part:**
   Memory required by the algorithm that depends on the size of the input. This includes:
   1. Input data
   2. Temporary variables
   3. Recursion stack or additional dynamic data structures (like arrays, linked lists, etc.).



For example:

```
int a=1,b=2,c=3;
```

Its space complexity is O(1).

```
int arr[n];
```

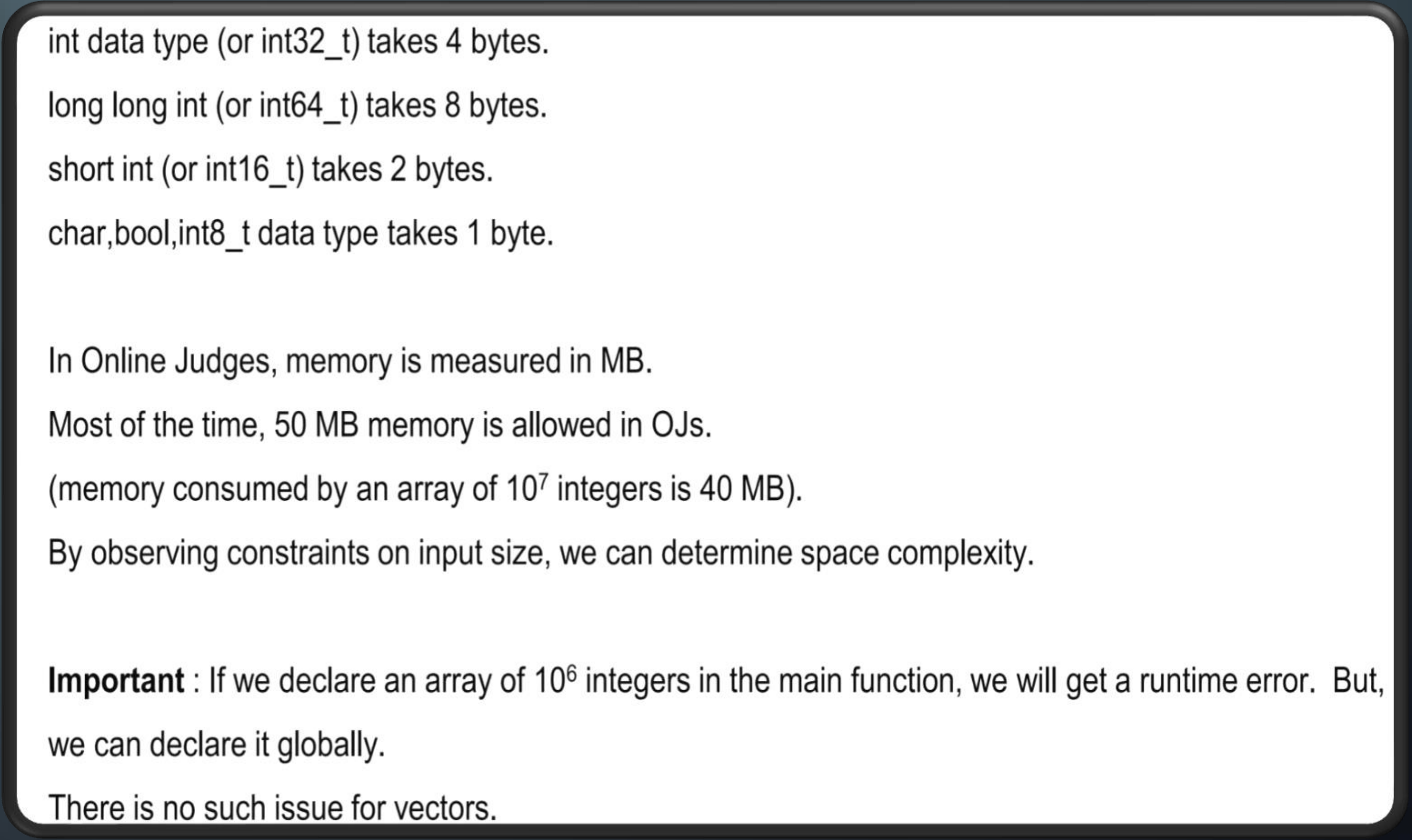
Its space complexity is O(n).

```
int arr[n][n];
```

Its space complexity is O(n^2).

```
int arr[n][m];
```

Its space complexity is O(n*m)



int data type (or int32_t) takes 4 bytes.

long long int (or int64_t) takes 8 bytes.

short int (or int16_t) takes 2 bytes.

char,bool,int8_t data type takes 1 byte.

In Online Judges, memory is measured in MB.

Most of the time, 50 MB memory is allowed in OJs.

(memory consumed by an array of 10^7 integers is 40 MB).

By observing constraints on input size, we can determine space complexity.

Important : If we declare an array of 10^6 integers in the main function, we will get a runtime error. But, we can declare it globally.

There is no such issue for vectors.

Self Study

Practice coding & difference of them

Week 3

Sorting algorithms: Insertion sort, selection sort

Page
31-39

INSERTION SORT

Insertion Sort is a simple, intuitive sorting algorithm that works similarly to how we sort playing cards in our hands. It builds the sorted array one element at a time by repeatedly picking the next element and inserting it into its correct position relative to the already sorted part of the array.

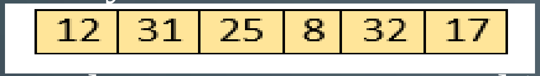
Working Procedure:

- **Step 1 -** If the element is the first element, assume that it is already sorted. Return 1.
- **Step 2 -** Pick the next element, and store it separately in a **key.**
- **Step 3 -** Now, compare the **key** with all elements in the sorted array.
- **Step 4 -** If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
- **Step 5 -** Insert the value.
- **Step 6 -** Repeat until the array is sorted.

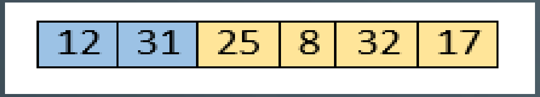
WORKING PROCEDURE WITH DEMONSTRATION

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

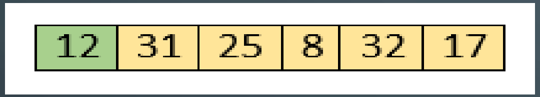
Let the elements of array are -



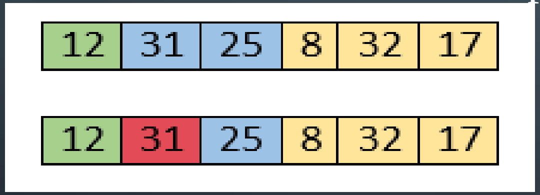
Initially, the first two elements are compared in insertion sort.



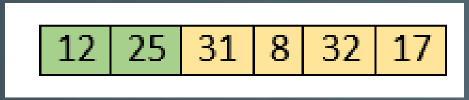
Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



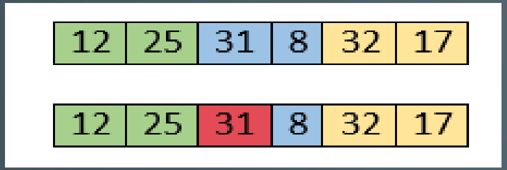
Now, move to the next two elements and compare them.



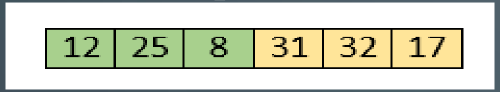
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.
For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.



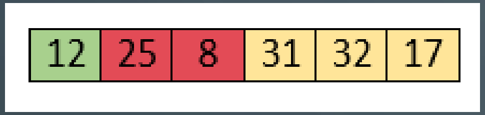
Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



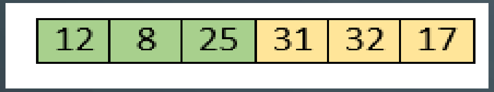
Both 31 and 8 are not sorted. So, swap them.



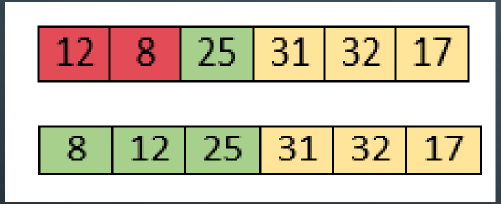
After swapping, elements 25 and 8 are unsorted.



So, swap them.

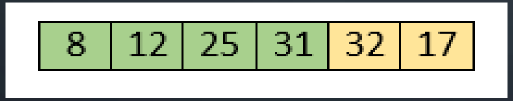


Now, elements 12 and 8 are unsorted.

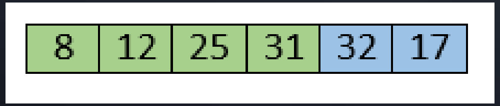


So, swap them too.

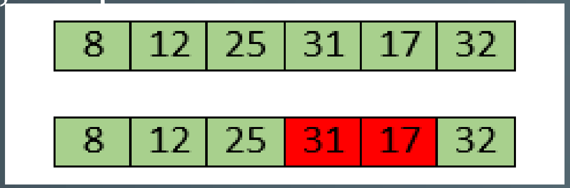
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



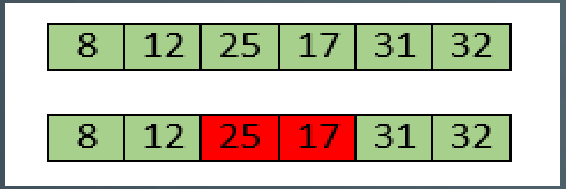
Move to the next elements that are 32 and 17.



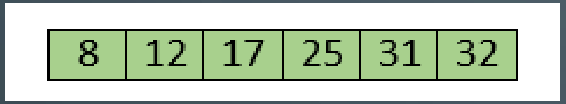
17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

Time Complexity:

Best case: O(n)

Worst case: O(n^2)

Average case: O(n^2)

Space Complexity:

O(1), Because directly modifies the input array & it sorts the input array without using additional memory structures (like arrays, lists, or stacks).

SELECTION SORT

Selection Sort is a simple comparison-based sorting algorithm. It works by dividing the input array into two parts: a sorted subarray and an unsorted subarray. The algorithm repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the sorted part.

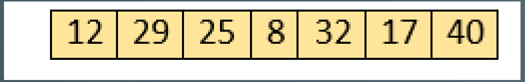
Working Procedure:

- Start with the first element as the sorted subarray (initially empty).
- Find the smallest element in the unsorted part of the array.
- Swap this smallest element with the first element of the unsorted part.
- Expand the sorted subarray by including the newly sorted element.
- Repeat steps 2–4 for the remaining unsorted portion until the entire array is sorted.

WORKING PROCEDURE WITH DEMONSTRATION

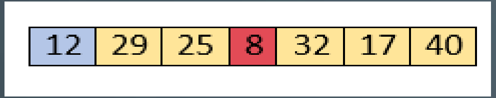
To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are –



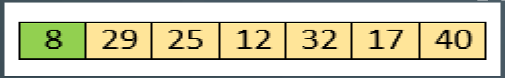
12	29	25	8	32	17	40

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.



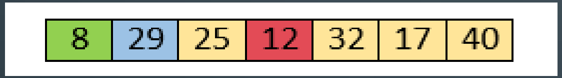
12	29	25	8	32	17	40

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.



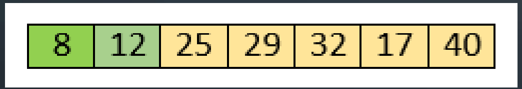
8	29	25	12	32	17	40

For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.



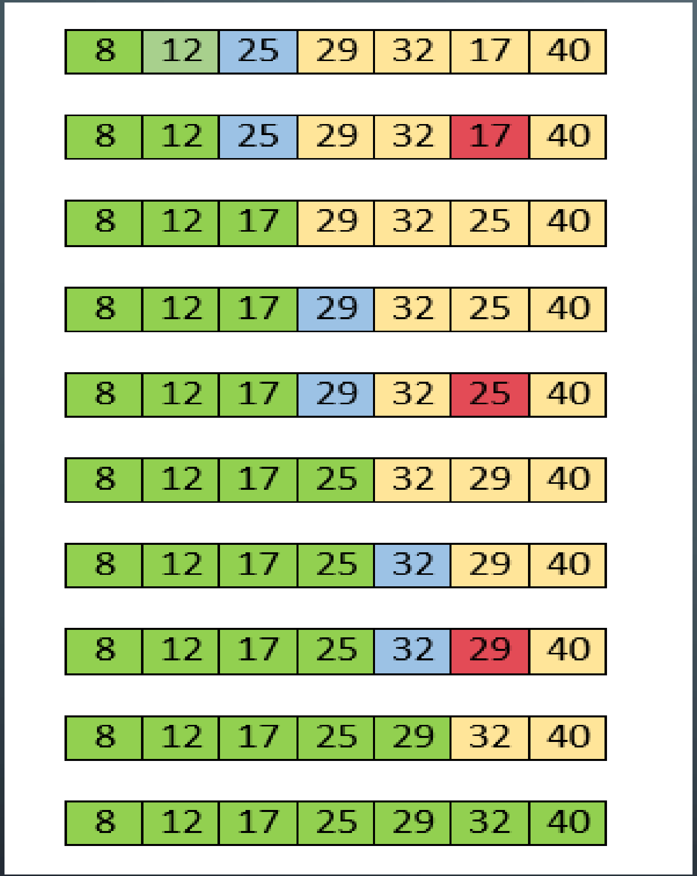
8	29	25	12	32	17	40

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



8	12	25	29	32	17	40

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



8	12	25	29	32	17	40
8	12	25	29	32	17	40
8	12	17	29	32	25	40
8	12	17	29	32	25	40
8	12	17	29	32	25	40
8	12	17	25	32	29	40
8	12	17	25	32	29	40
8	12	17	25	32	29	40
8	12	17	25	29	32	40
8	12	17	25	29	32	40

Now, the array is completely sorted.

Time Complexity:

Best case: $O(n^2)$

Worst case: $O(n^2)$

Average case: $O(n^2)$

Space Complexity:

O(1), Because directly modifies the input array & it sorts the input array without using additional memory structures (like arrays, lists, or stacks).

Self Study

Practice coding & difference of them

Week 4

Bubble Sort & Quick Sort

Page
41-54

BUBBLE SORT

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

Working Procedure:

Step 1 – Check if the first element in the input array is greater than the next element in the array.

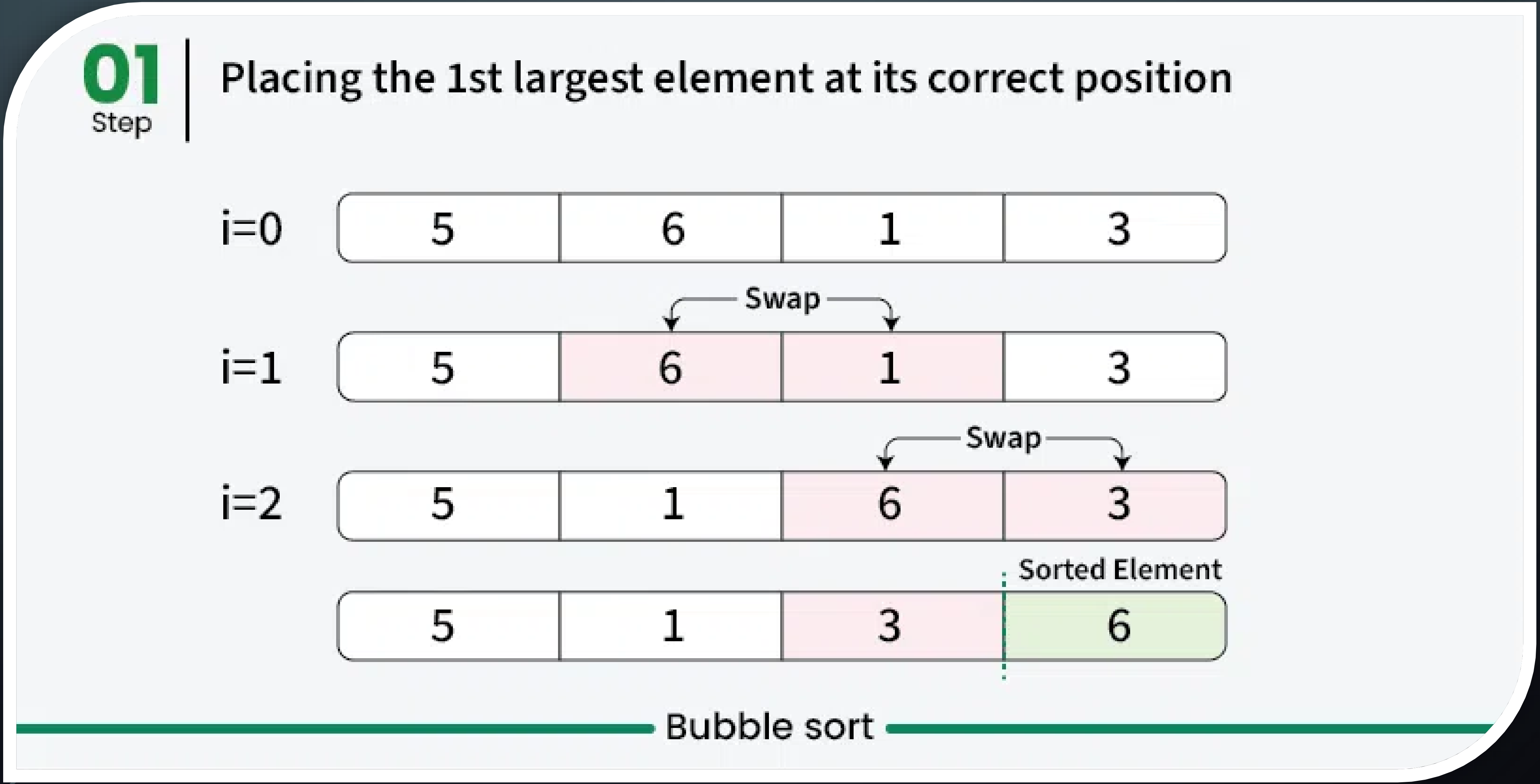
Step 2 – If it is greater, swap the two elements; otherwise move the pointer forward in the array.

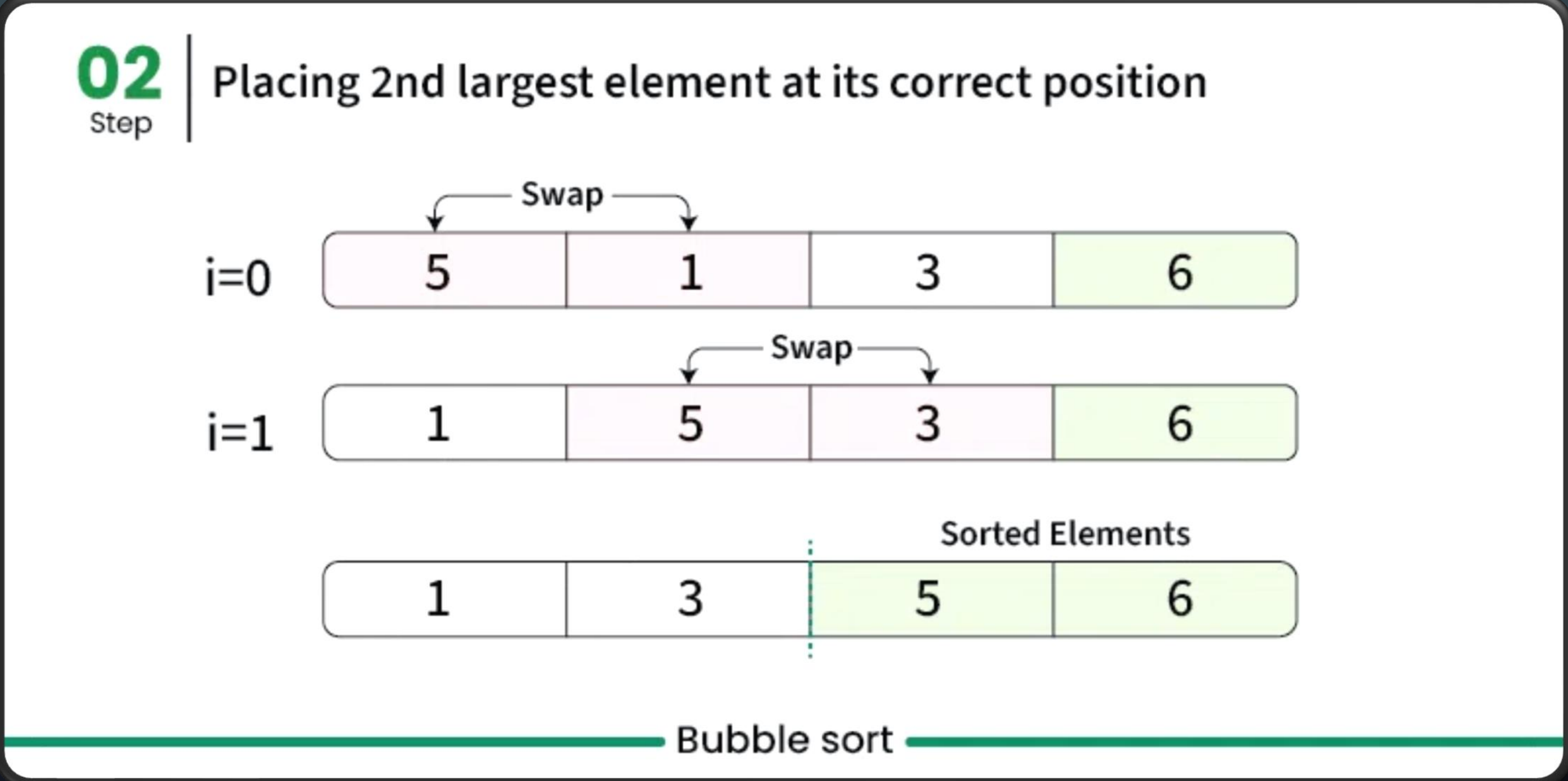
Step 3 – Repeat Step 2 until we reach the end of the array.

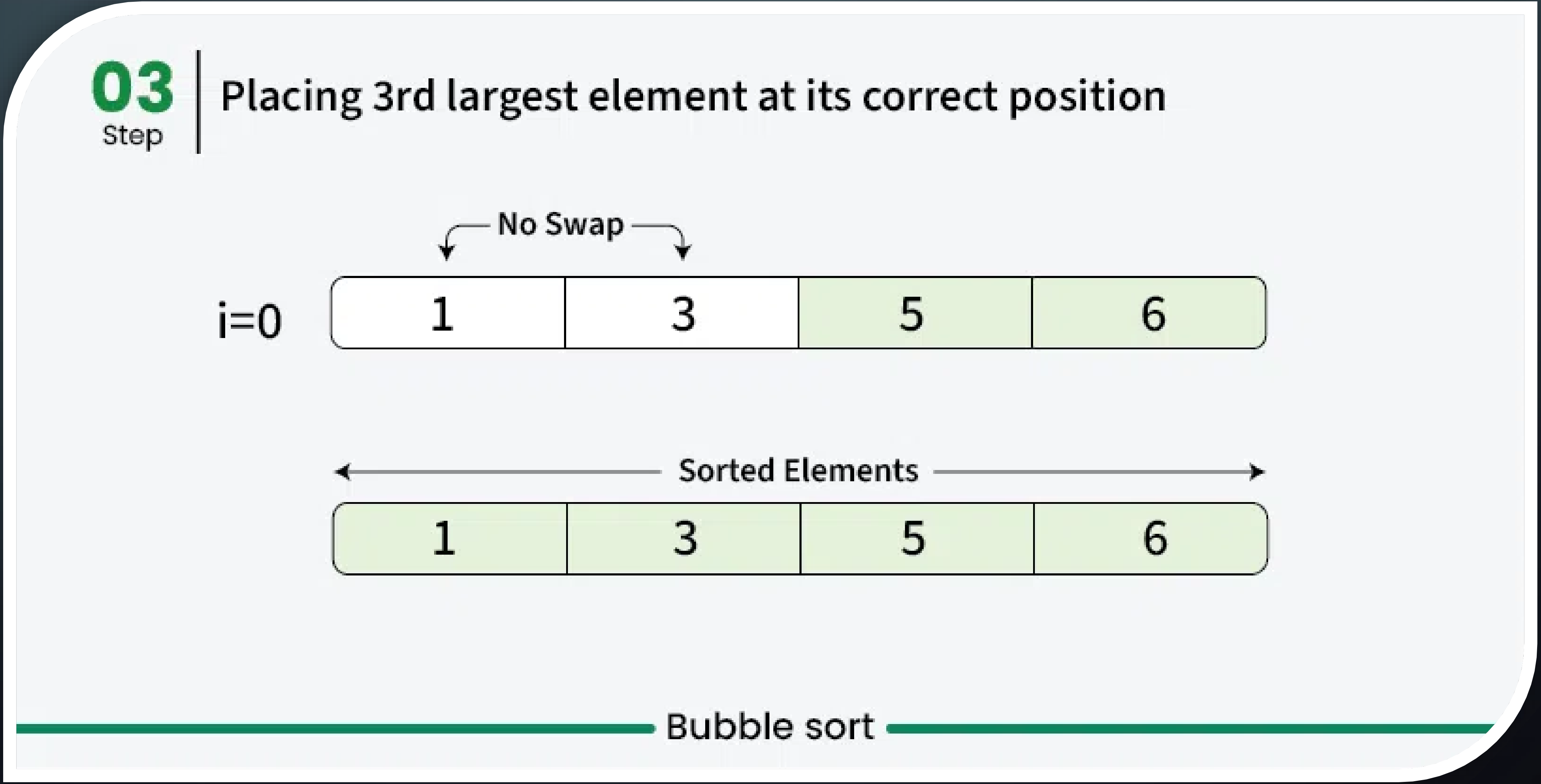
Step 4 – Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

Step 5 – The final output achieved is the sorted array.

WORKING PROCEDURE WITH DEMONSTRATION




02
Step
Placing 2nd largest element at its correct position
Swap
i=0
5
1
3
6
Swap
i=1
1
5
3
6
Sorted Elements
1
3
5
6
Bubble sort



03 Step

Placing 3rd largest element at its correct position

Bubble sort

TIME & SPACE COMPLEXITY

Time Complexity

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **O(n^2).**
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **O(n^2).**

Space Complexity: O(1)

QUICK SORT

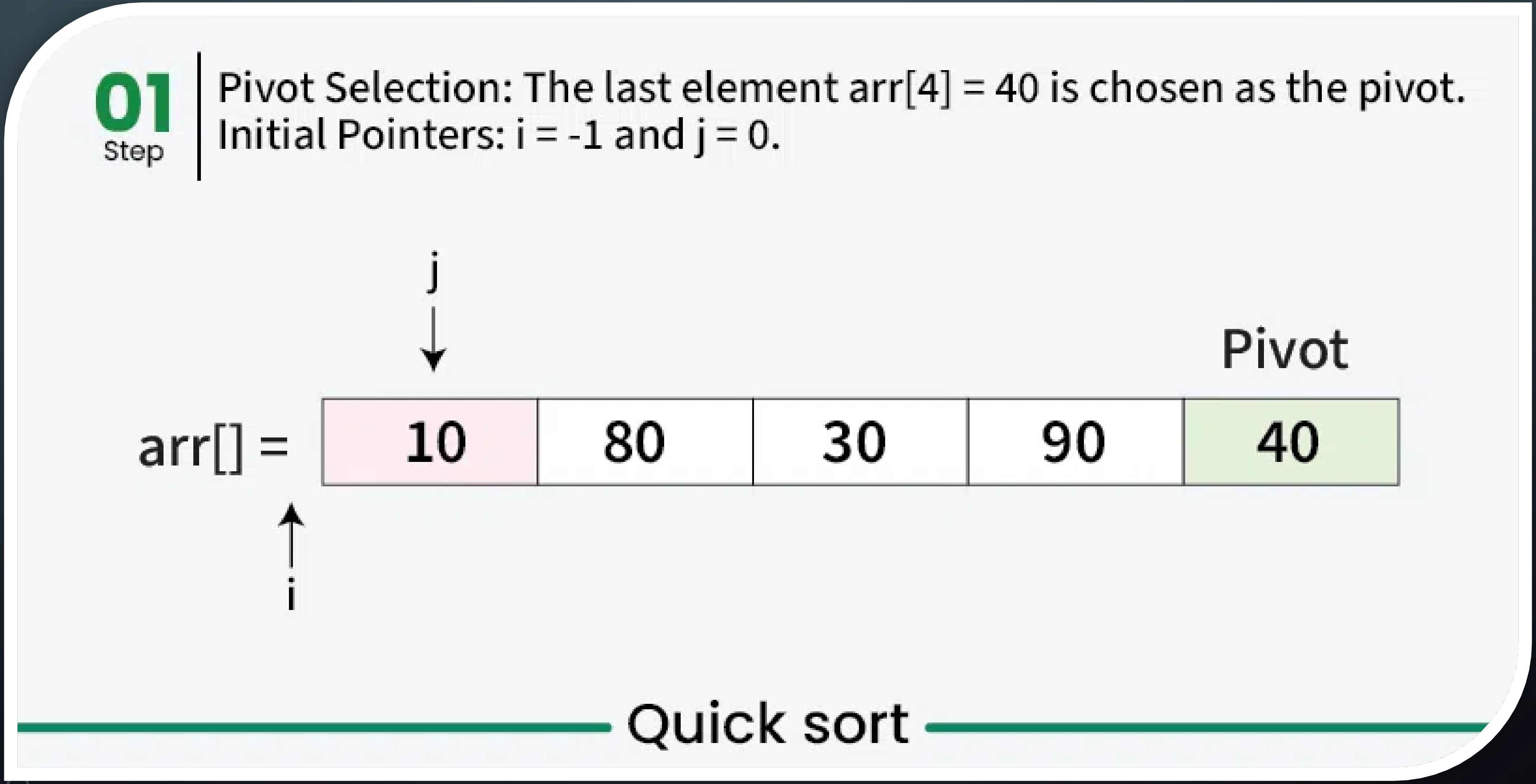
QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Working Procedure:

There are mainly three steps in the algorithm:

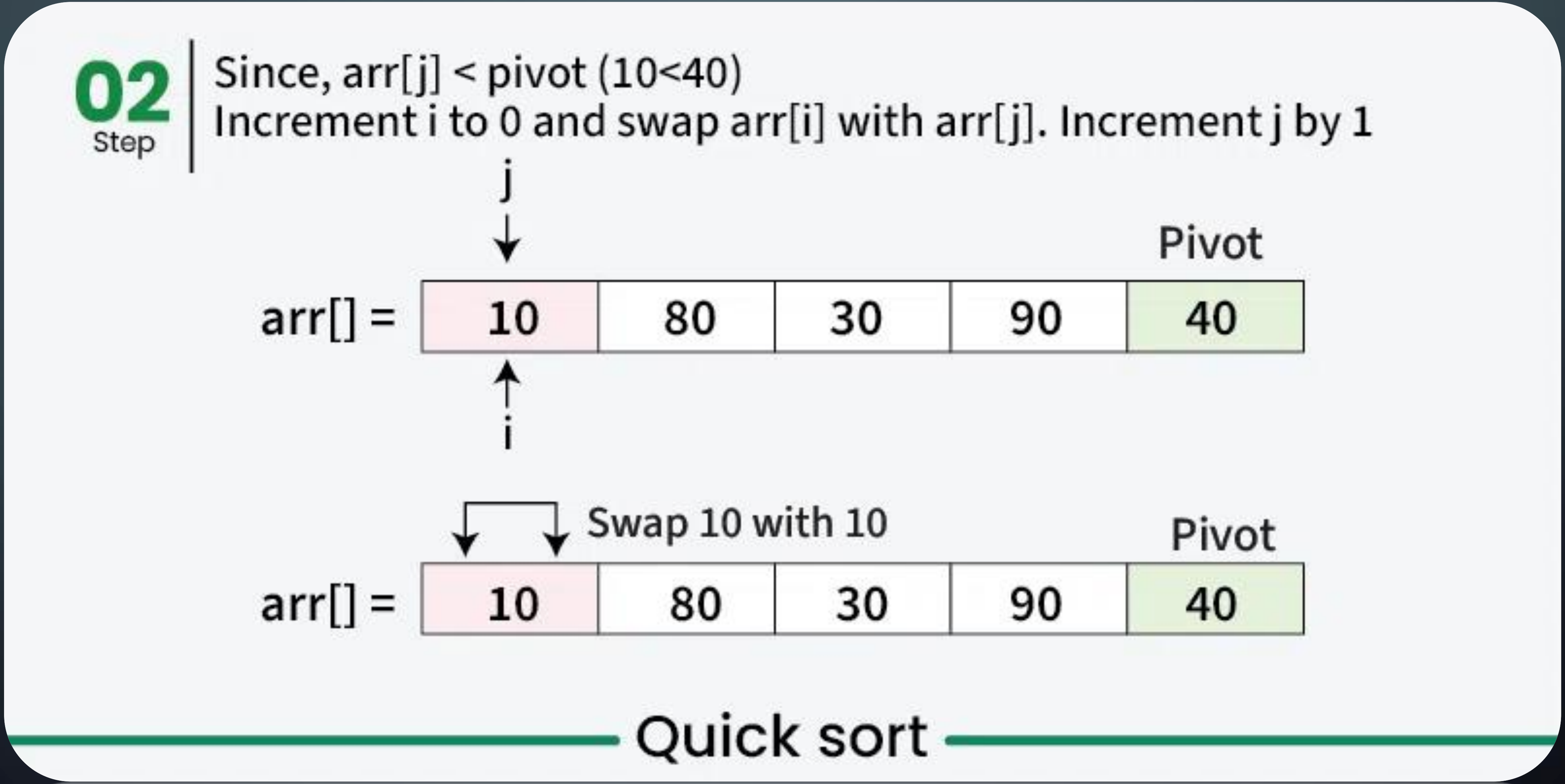
1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, 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.

WORKING PROCEDURE WITH EXAMPLE



01 Step

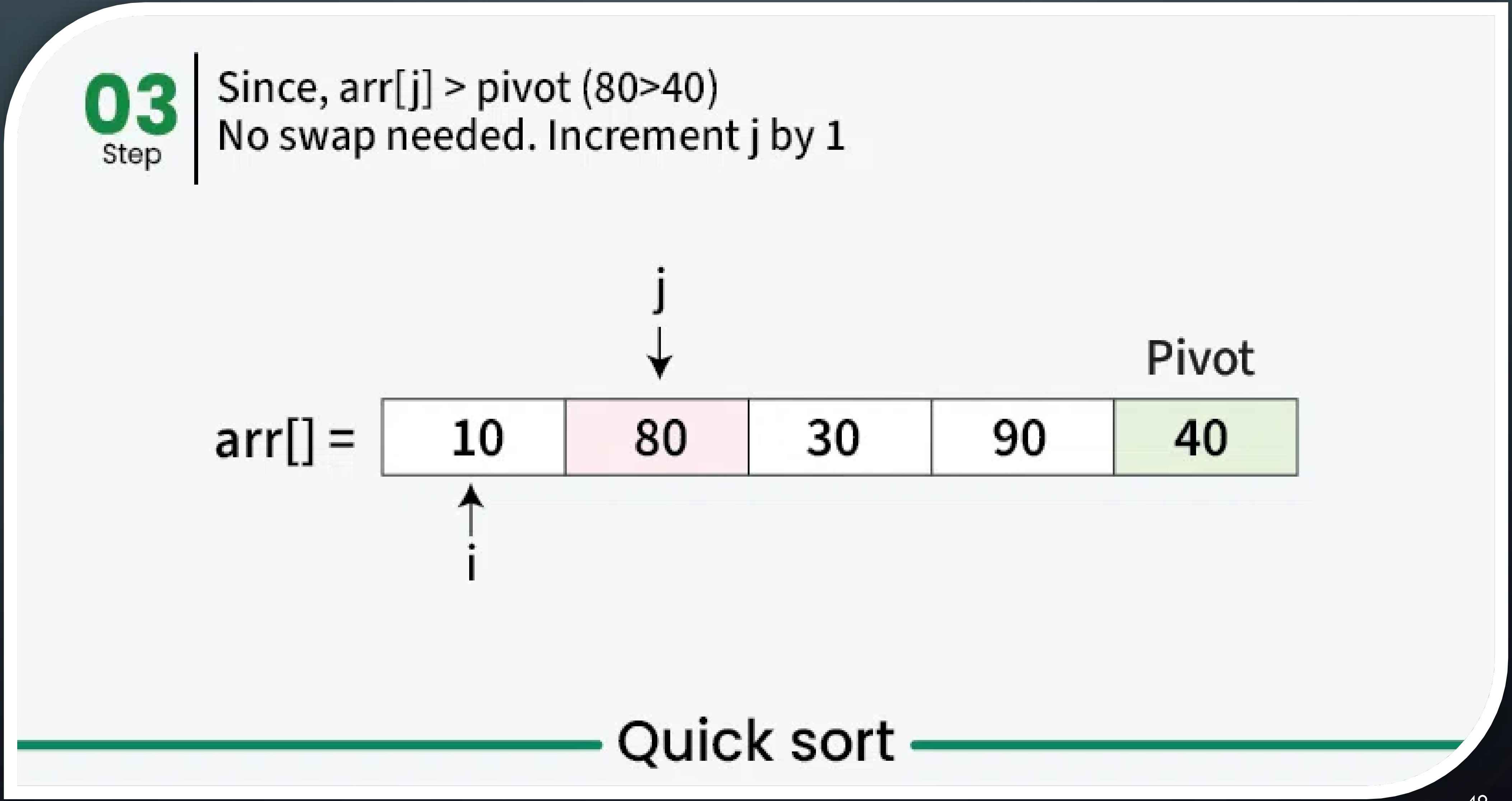
Pivot Selection: The last element arr[4] = 40 is chosen as the pivot.
Initial Pointers: i = -1 and j = 0.



02 Step

Since, arr[j] < pivot (10<40)
Increment i to 0 and swap arr[i] with arr[j]. Increment j by 1

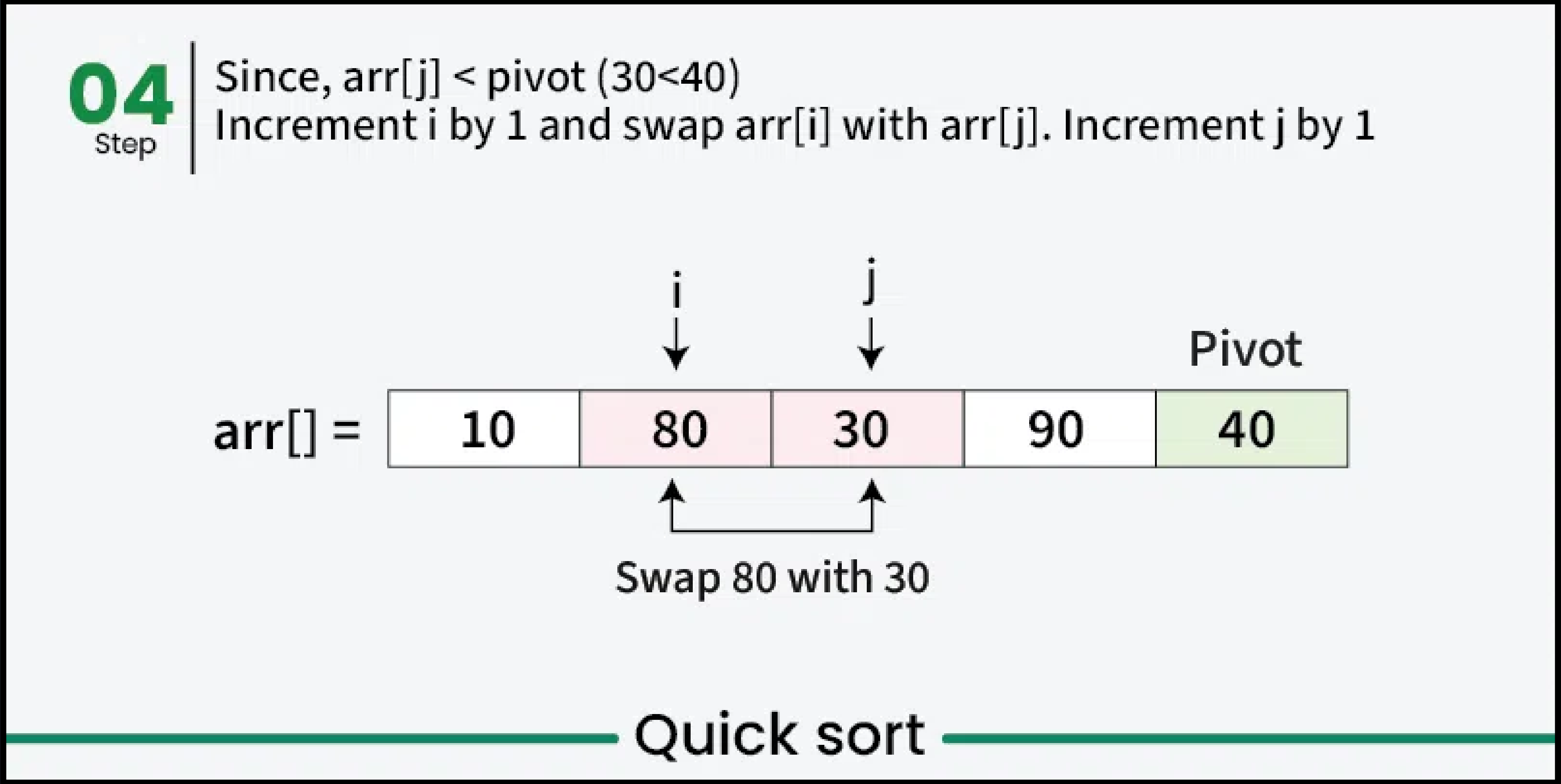
Quick sort



03 Step

Since, arr[j] > pivot (80>40)
No swap needed. Increment j by 1

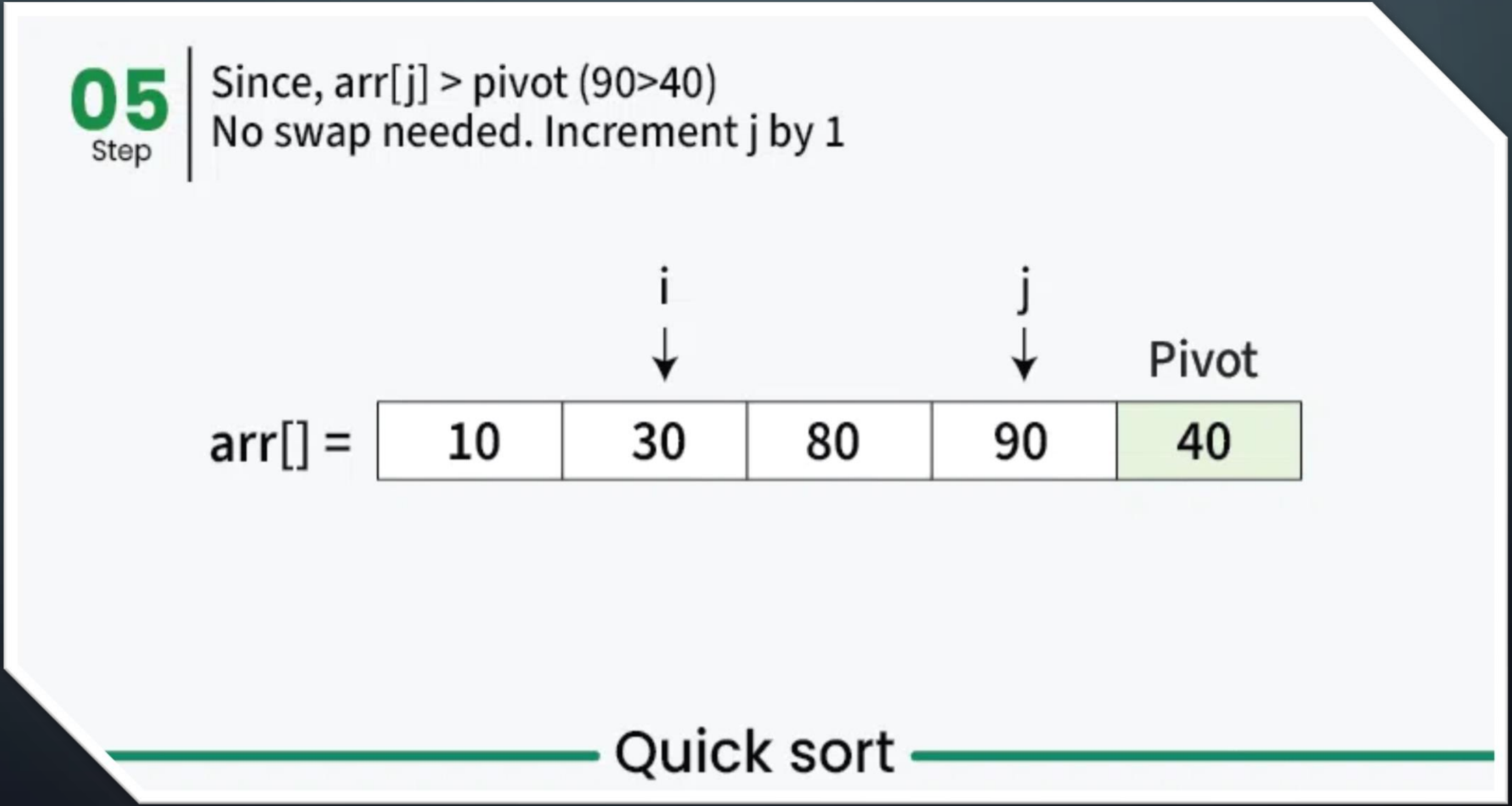
Quick sort



04 Step

Since, arr[j] < pivot (30<40)
Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1

Quick sort

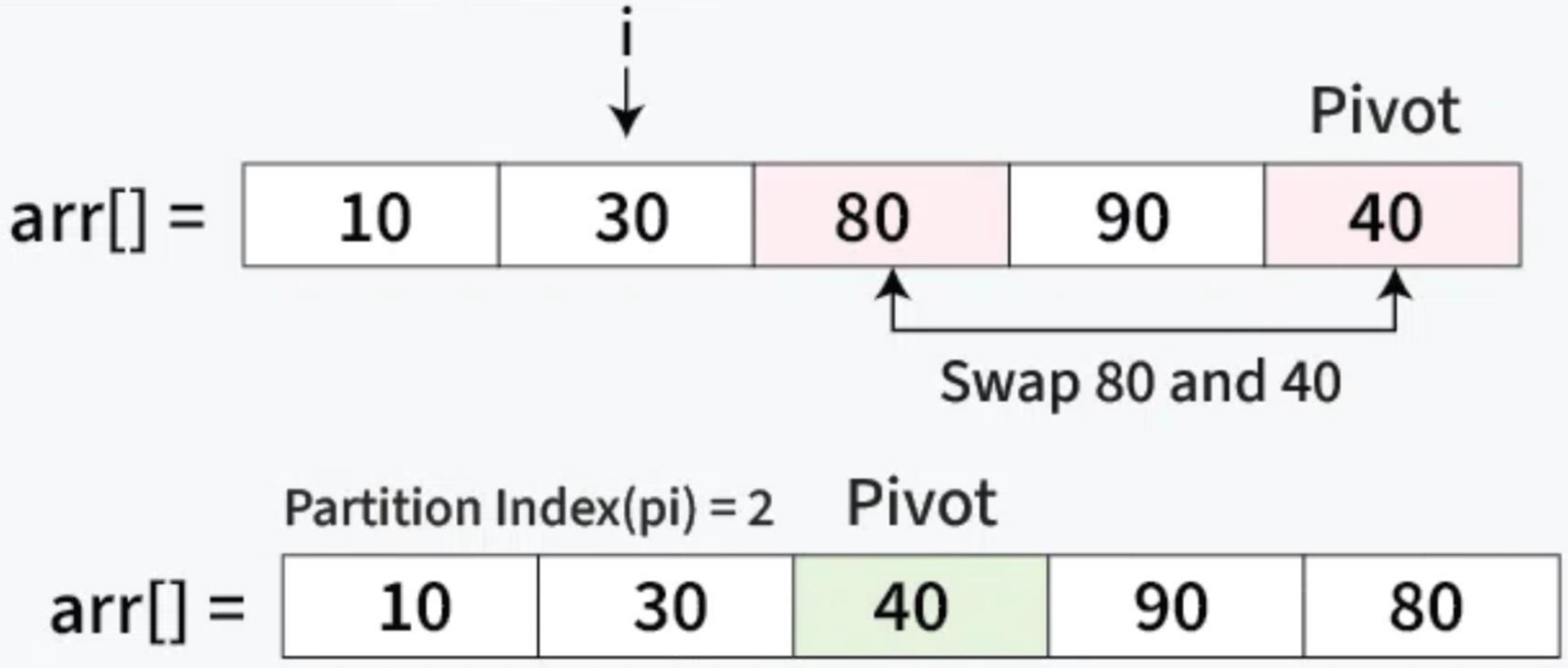

05 Step

Since, arr[j] > pivot (90>40)
No swap needed. Increment j by 1

Quick sort

06 Step

Since traversal of j has ended. Now move pivot to its correct position, Swap arr[i + 1] = arr[2] with arr[4] = 40.



Quick sort

TIME & SPACE COMPLEXITY

- **Time Complexity:**
- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case:** ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).
- **Auxiliary Space:** $O(n)$, due to recursive call stack

Self Study

Practice coding & difference of them

Week 5

Merge Sort & Radix Sort

MERGE SORT

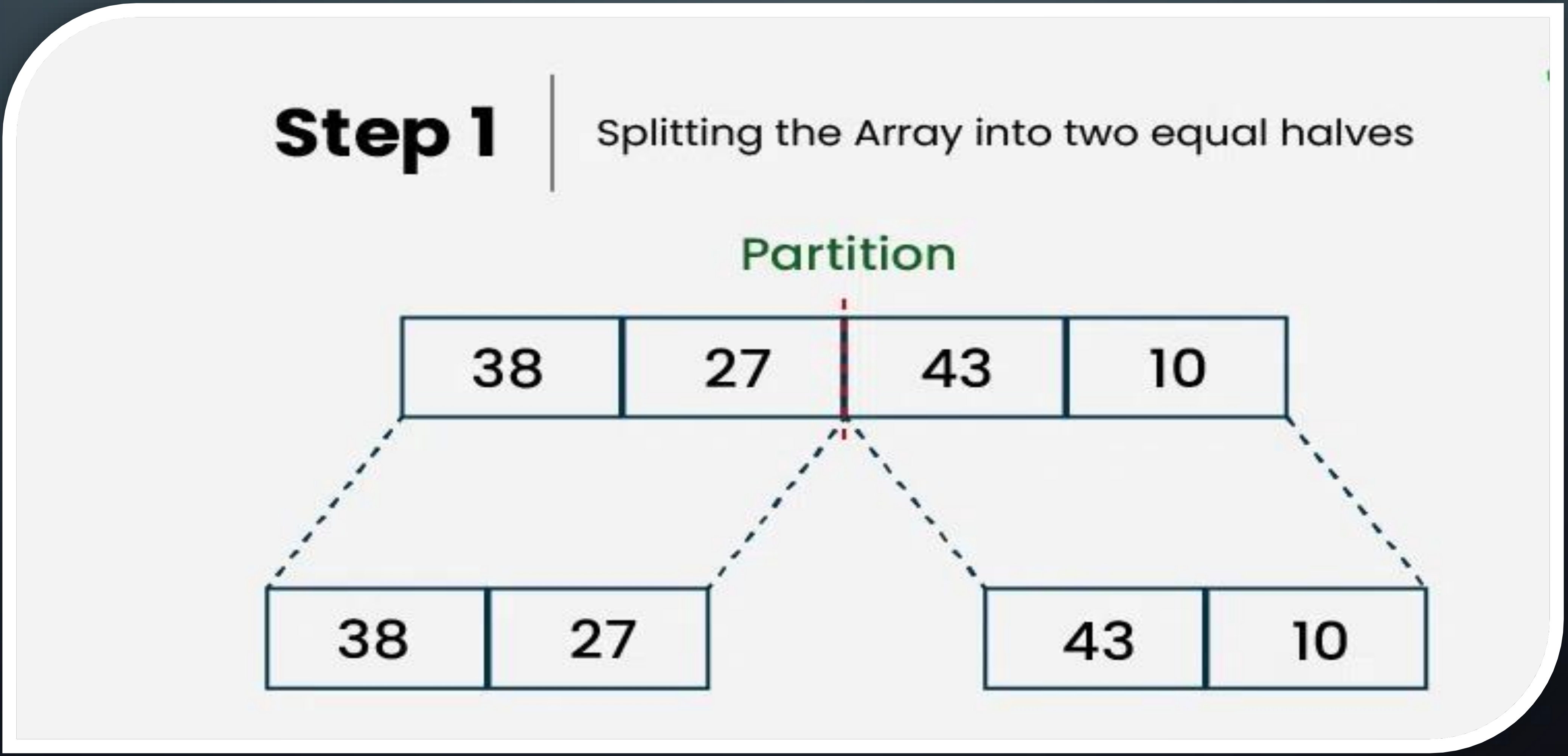
Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

Working Procedure:

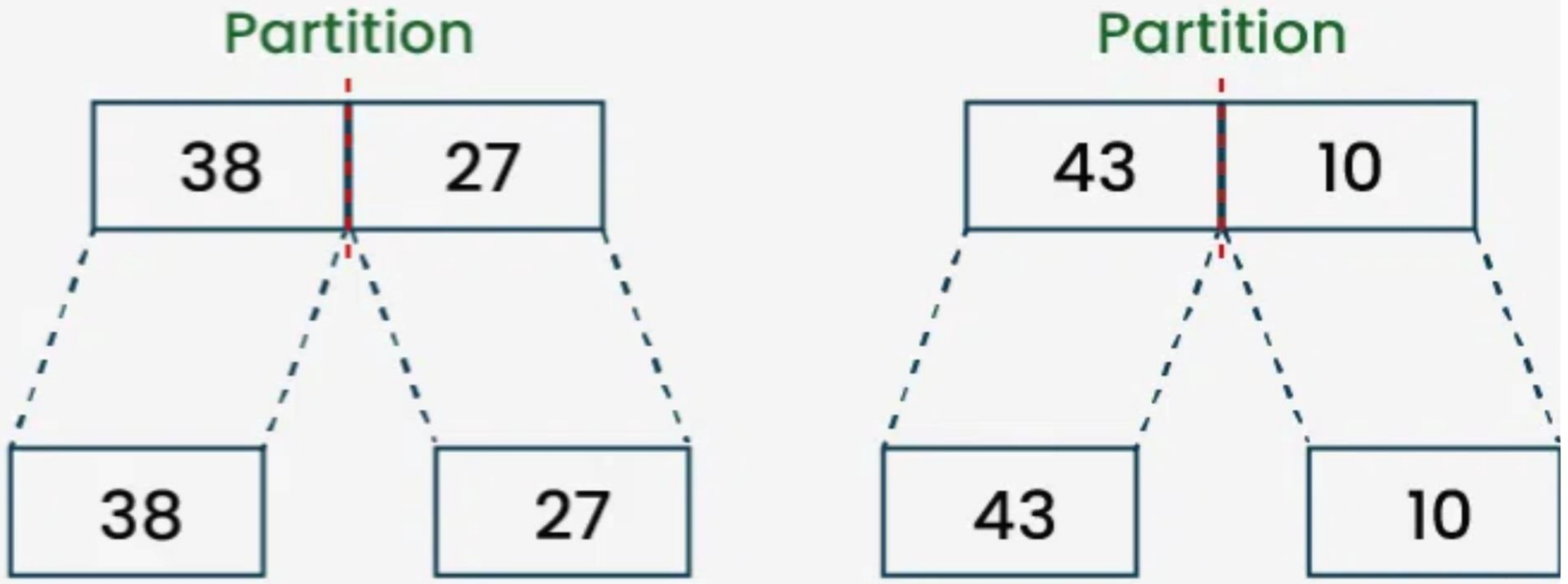
1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

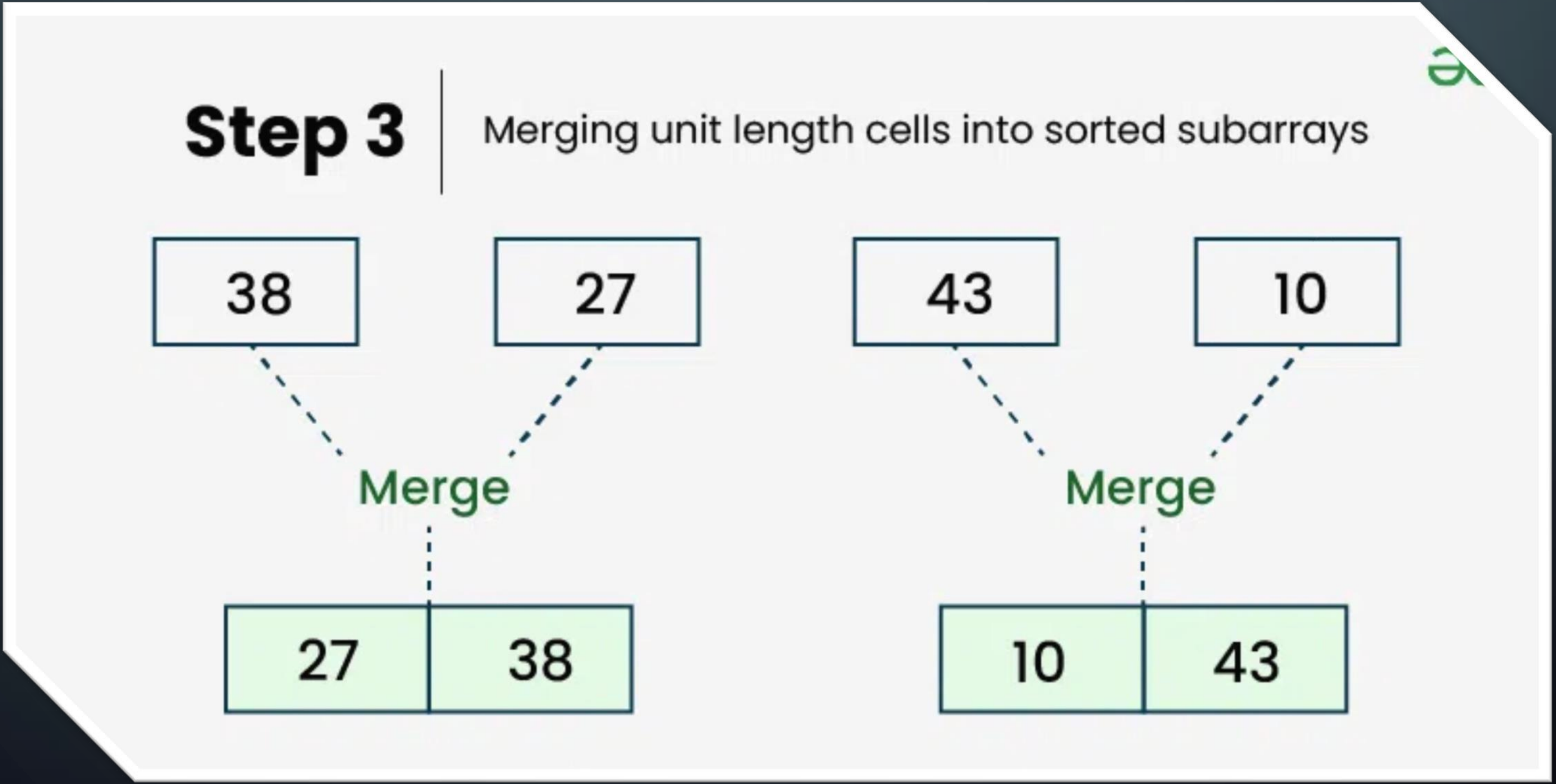
WORKING PROCEDURE WITH EXAMPLE

Let’s sort the array or list **[38, 27, 43, 10]** using Merge Sort.

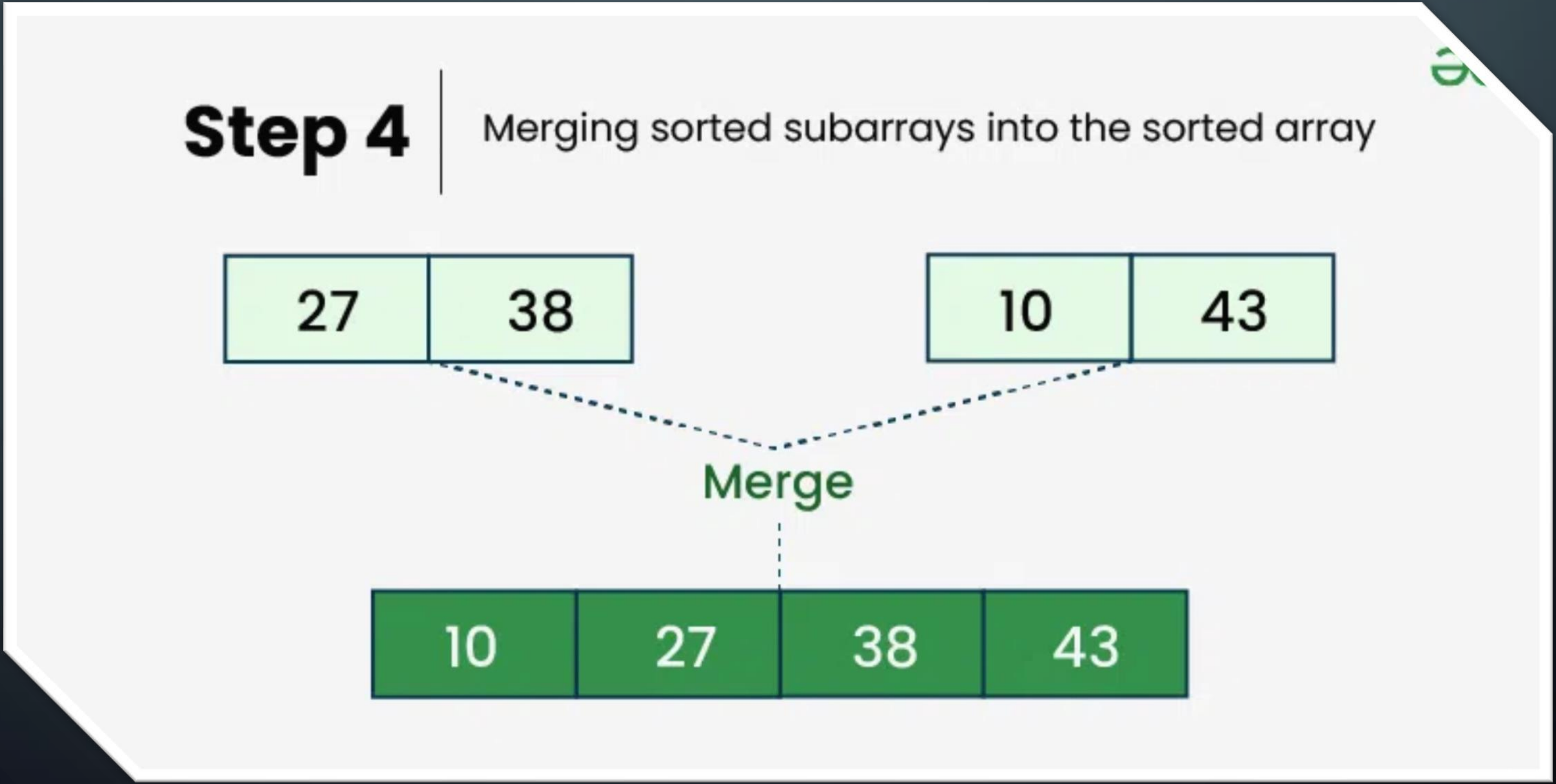


Step 2 | Splitting the subarrays into two halves





Step 3 | Merging unit length cells into sorted subarrays


Step 4
Merging sorted subarrays into the sorted array
27
38
10
43
Merge
10
27
38
43

Recurrence Relation of Merge Sort:

- The recurrence relation of merge sort is:
  T(n)={Θ(1)if n=12T(n2)+Θ(n)if n>1*T*(*n*)={Θ(1)2*T*(2*n*)+Θ(*n*)if *n*=1if *n*>1
- T(n) Represents the total time time taken by the algorithm to sort an array of size n.
- 2T(n/2) represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has n/2 elements, we have two recursive calls with input size as (n/2).
- O(n) represents the time taken to merge the two sorted halves.

TIME & SPACE COMPLEXITY

- **Time Complexity:**
  - **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
  - **Average Case:** O(n log n), When the array is randomly ordered.
  - **Worst Case:** O(n log n), When the array is sorted in reverse order.

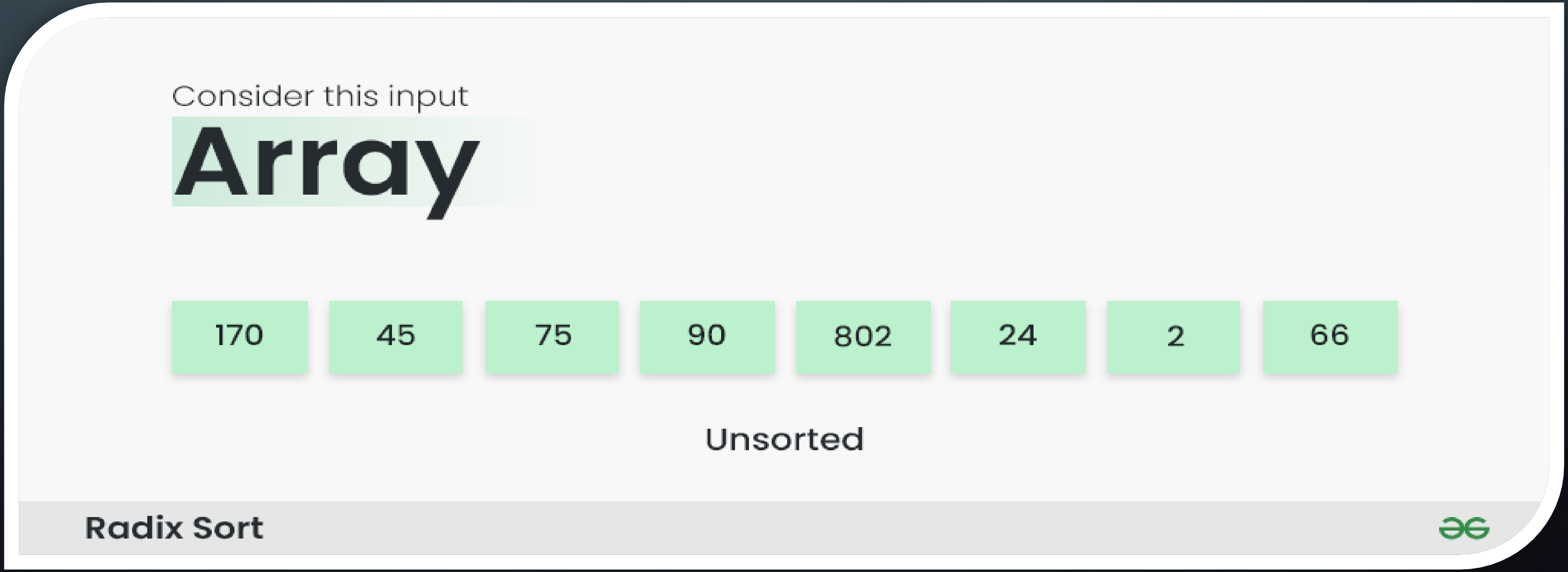
- **Auxiliary Space:** O(n), Additional space is required for the temporary array used during merging

RADIX SORT

Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

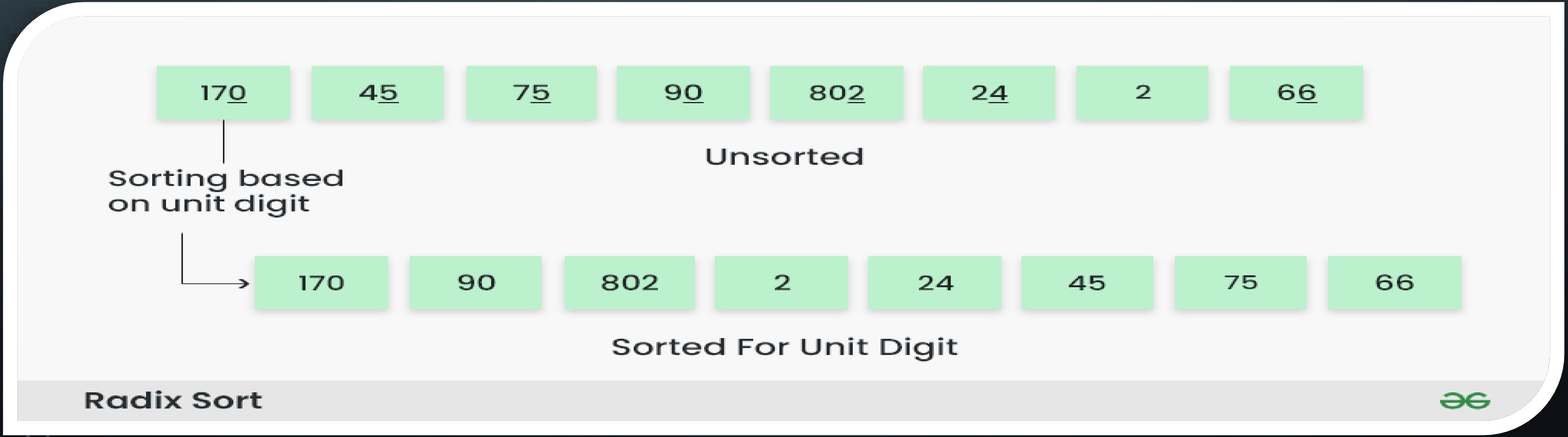
How does Radix Sort Algorithm work?

To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:



- **Step 1:** Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.
- **Step 2:** Sort the elements based on the unit place digits (X=0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place. It's important to understand that the default implementation of counting sort is unstable i.e. same keys can be in a different order than the input array. To solve this problem, We can iterate the input array in reverse order to build the output array. This strategy helps us to keep the same keys in the same order as they appear in the input array.

Sorting based on the unit place:

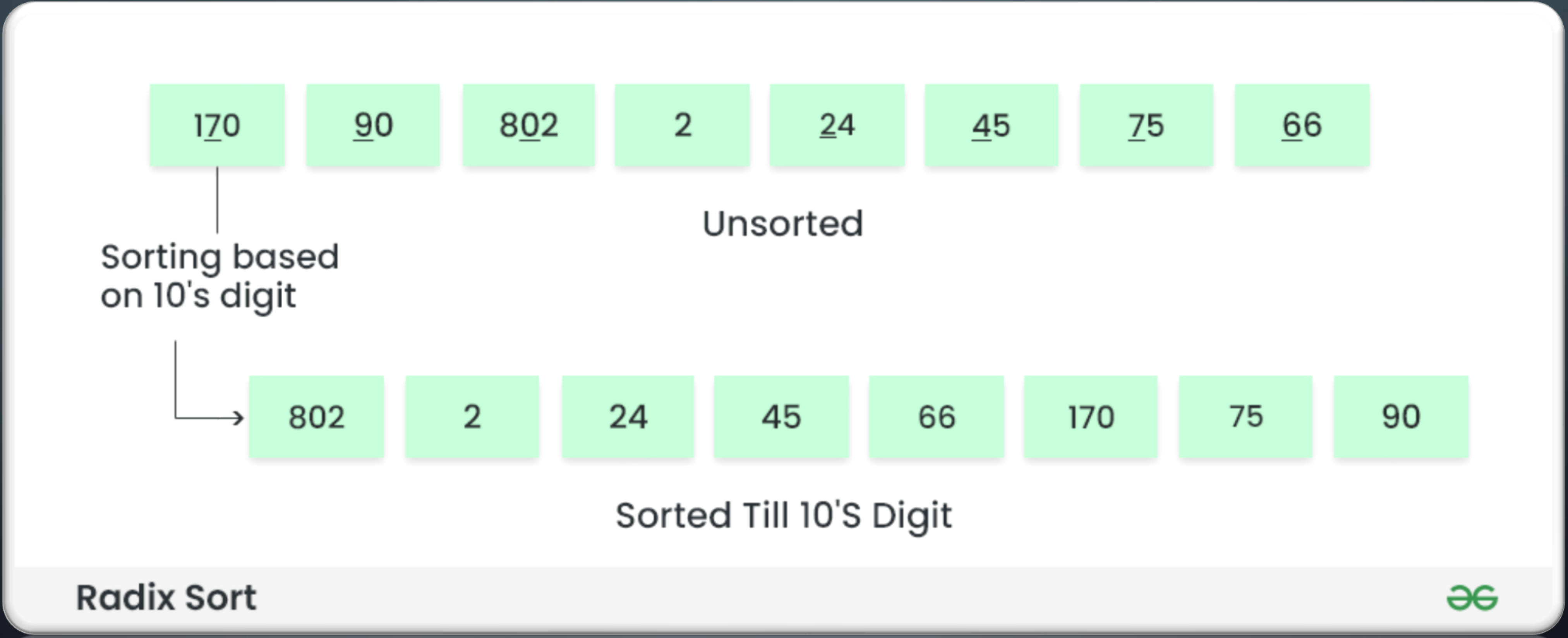


Step 3: Sort the elements based on the tens place digits.

Sorting based on the tens place:

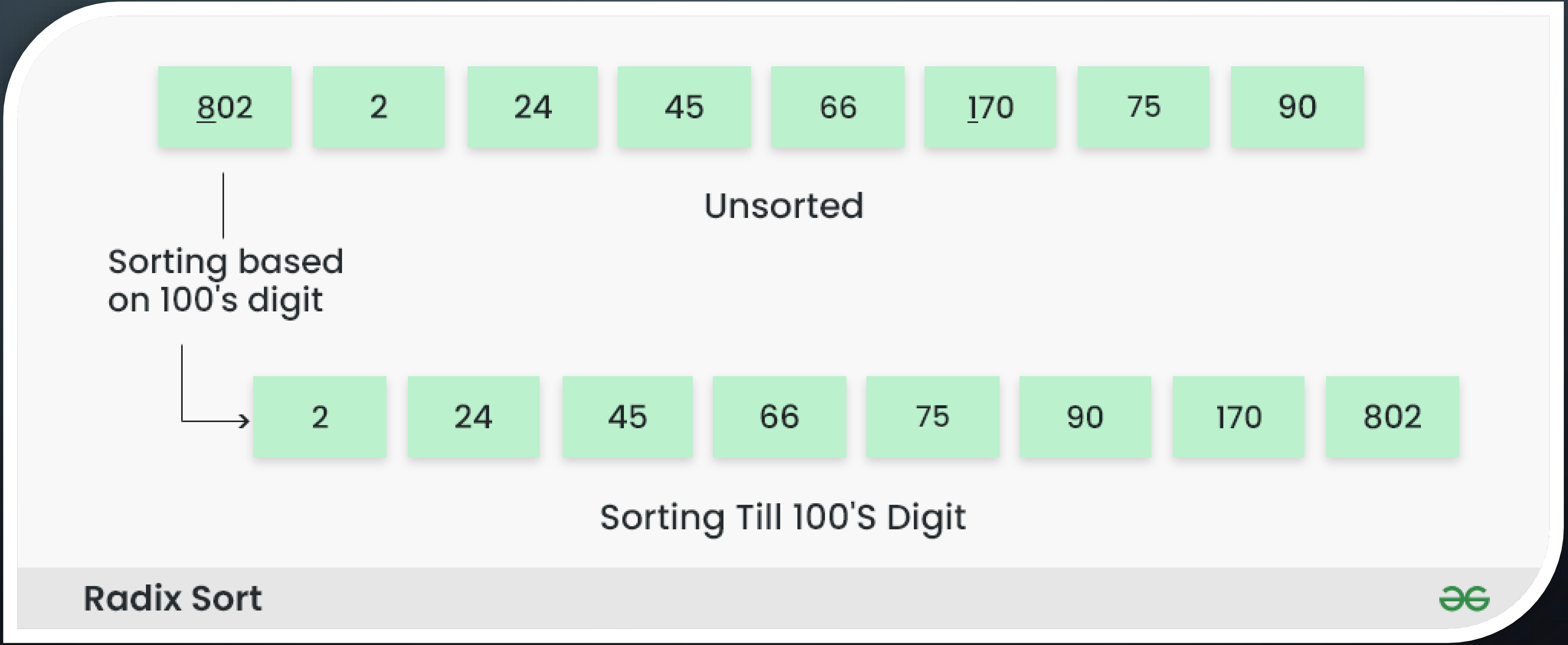
Perform counting sort on the array based on the tens place digits.

The sorted array based on the tens place is [802, 2, 24, 45, 66, 170, 75, 90].



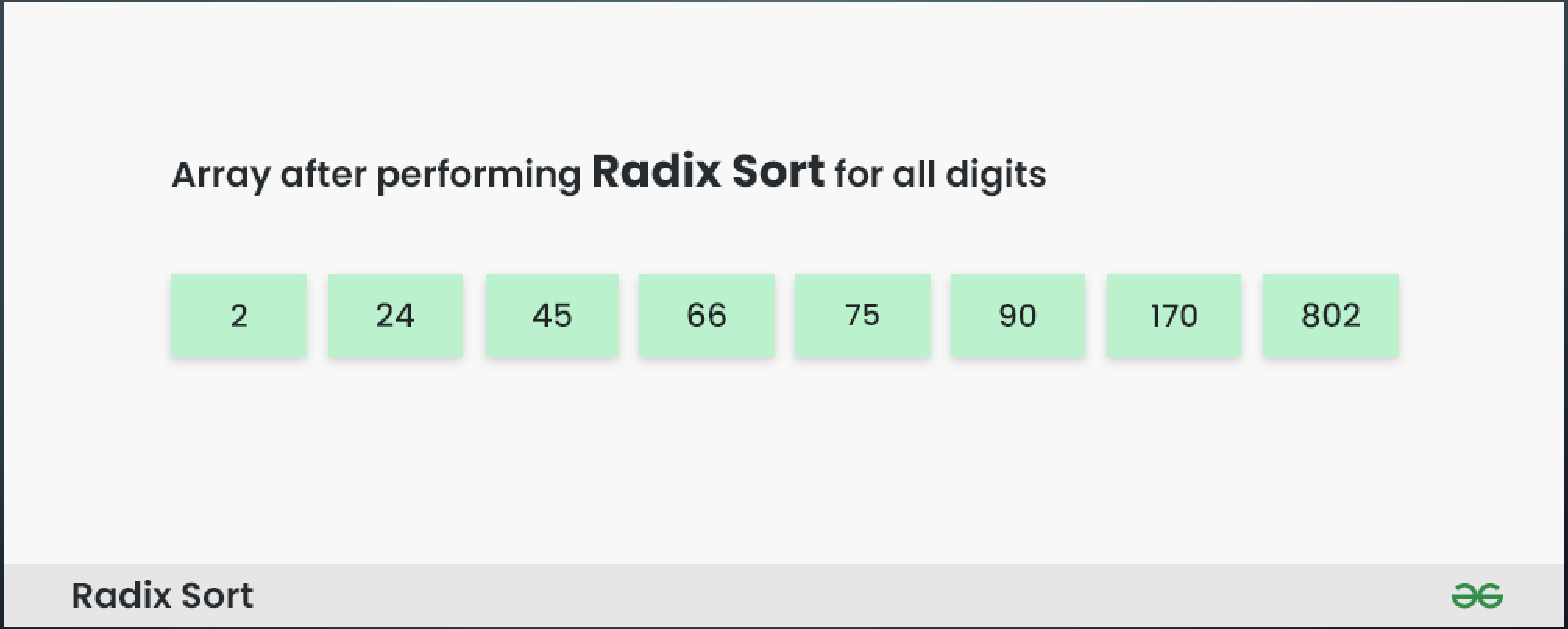
Step 4: Sort the elements based on the hundreds place digits.

- **Sorting based on the hundreds place:**
- Perform counting sort on the array based on the hundreds place digits.
- The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802].



Step 5: The array is now sorted in ascending order.

- The final sorted array using radix sort is [2, 24, 45, 66, 75, 90, 170, 802].



TIME & SPACE COMPLEXITY

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is **Ω(n+k)**.
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Radix sort is **θ(nk)**.
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Radix sort is **O(nk)**.

Self Study

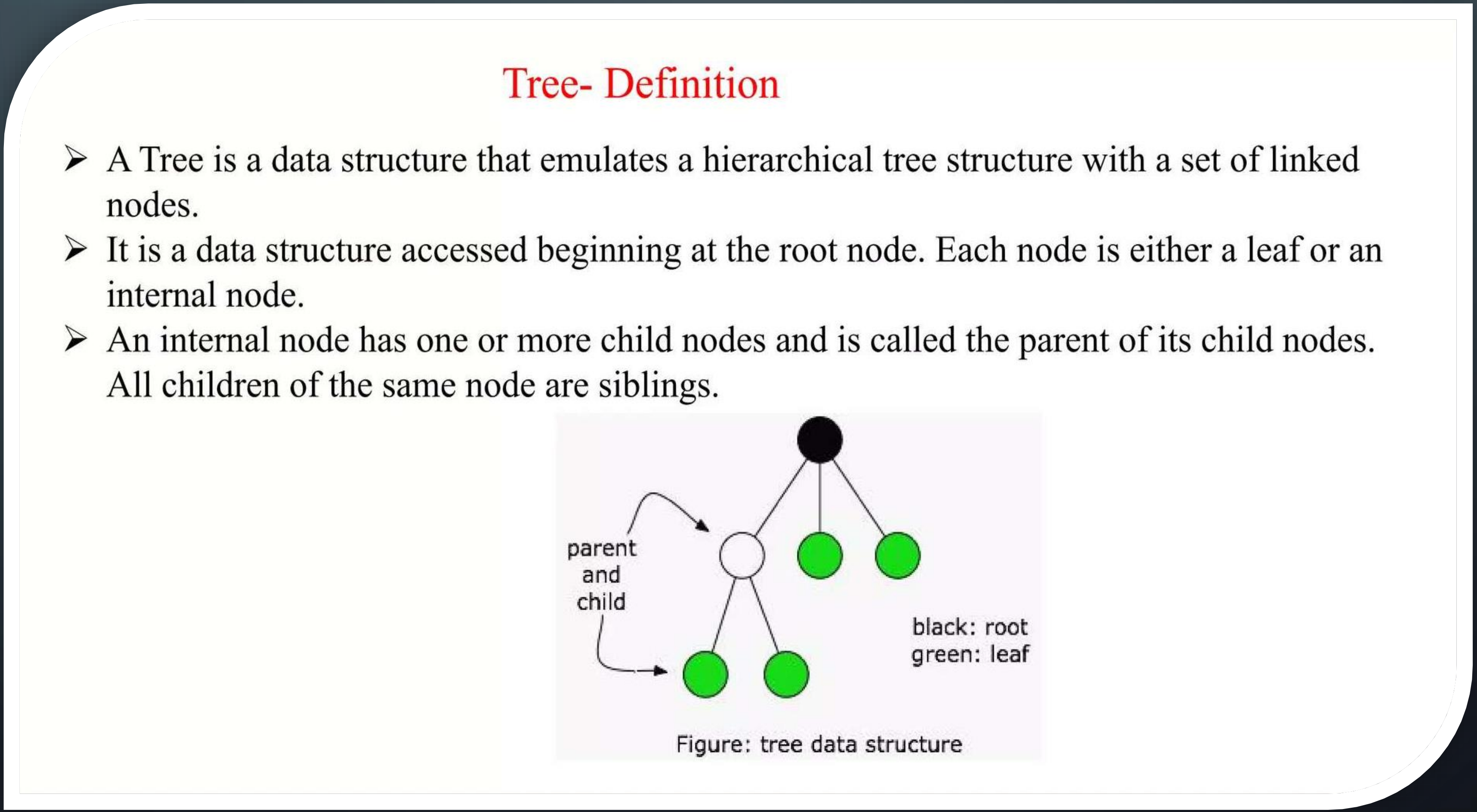
Practice coding & difference of them

Week 6

Introduction to Tree & Tree Traversal

Page
71-85

TREE



Tree- Definition

- A Tree is a data structure that emulates a hierarchical tree structure with a set of linked nodes.
- It is a data structure accessed beginning at the root node. Each node is either a leaf or an internal node.
- An internal node has one or more child nodes and is called the parent of its child nodes. All children of the same node are siblings.

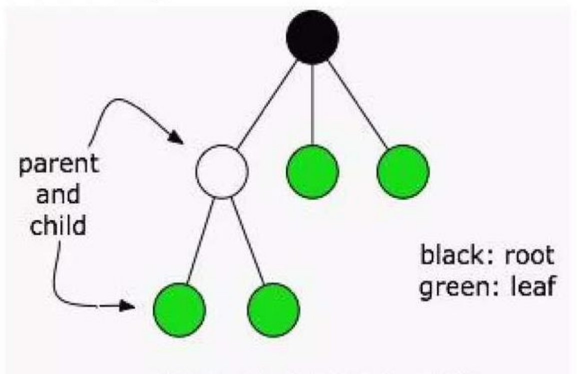
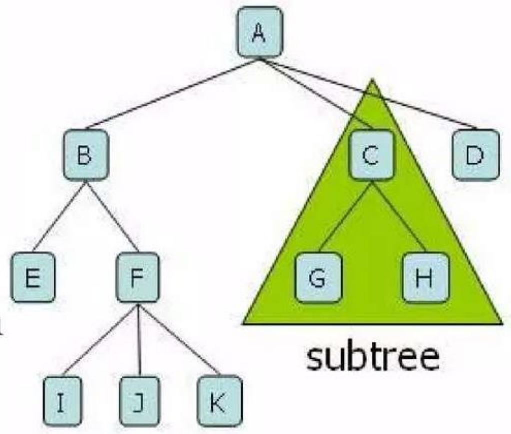


Figure: tree data structure

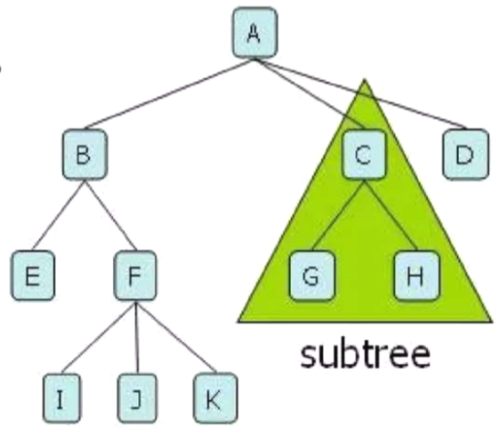
Tree Terminologies

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node: a kind of leaf node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc
- Degree of an element: no. of children it has
- Sub tree: tree consisting of a node and its descendants
- Path: traversal from node to node along the edges that results in a sequence
- Root: node at the top of the tree
- Parent: any node, except root has exactly one edge running upward to another node. The node above it is called parent.

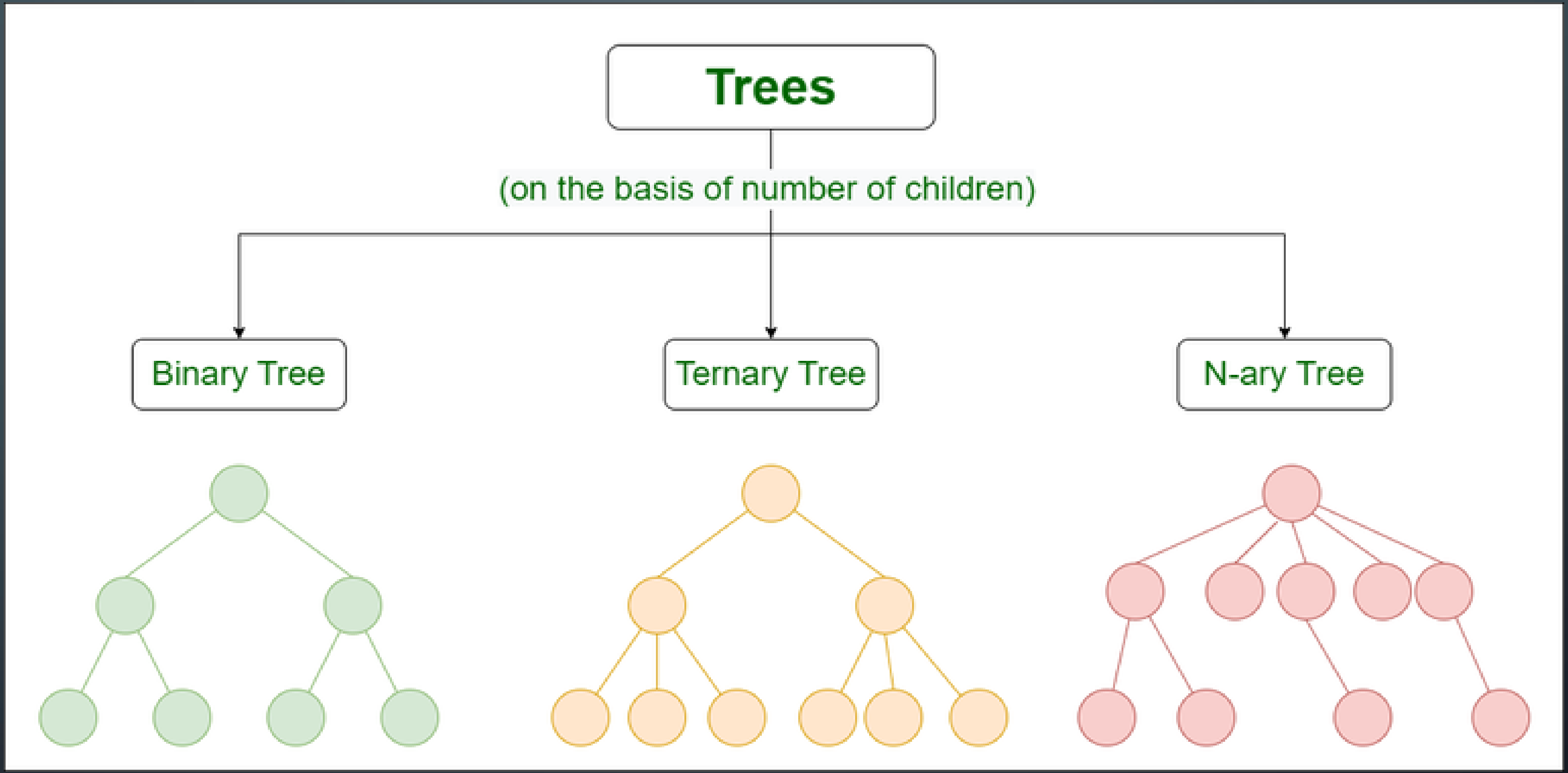


Tree Terminologies

- Child: any node may have one or more lines running downward to other nodes. Nodes below are children.
- Leaf: a node that has no children
- Sub tree: any node can be considered to be the root of a subtree, which consists of its children and its children's children and so on.
- Visiting: a node is visited when program control arrives at the node, usually for processing.
- Traversing: to traverse a tree means to visit all the nodes in some specified order.
- Levels: the level of a particular node refers to how many generations the node is from the root. Root is assumed to be level 0.



TYPES OF TREES

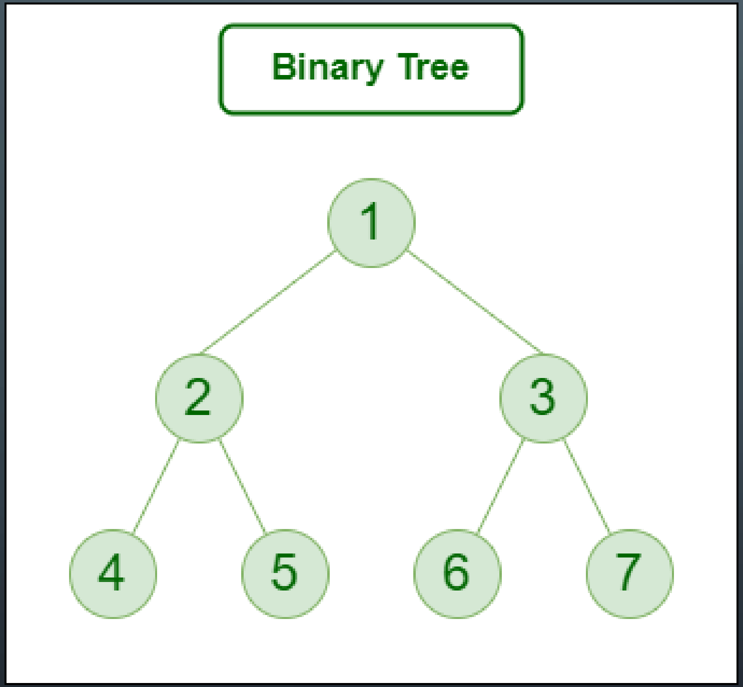


BINARY TREE

A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Example:

Consider the tree below. Since each node of this tree has only 2 children, it can be said that this tree is a Binary Tree

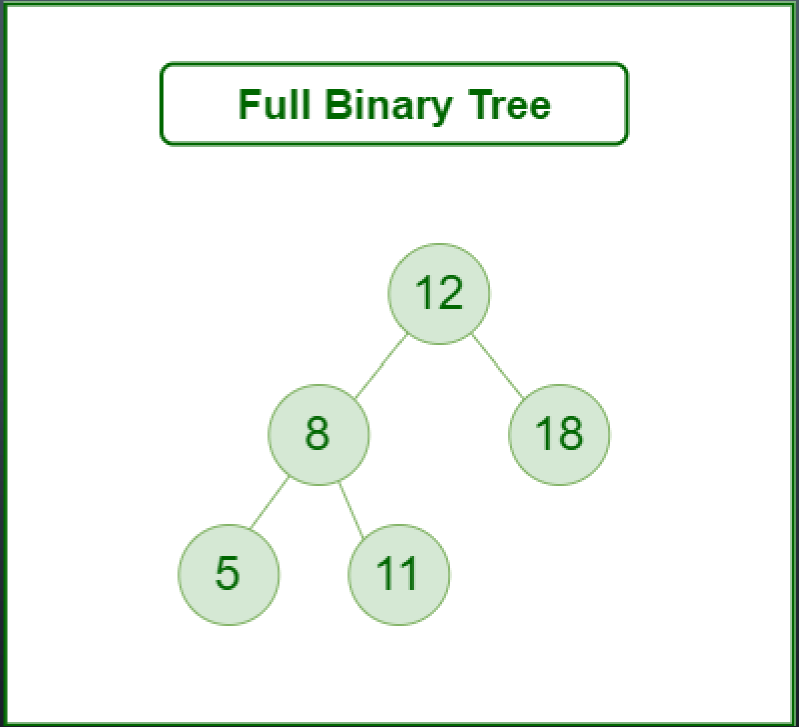


TYPES OF BINARY TREE

1. Full Binary Tree

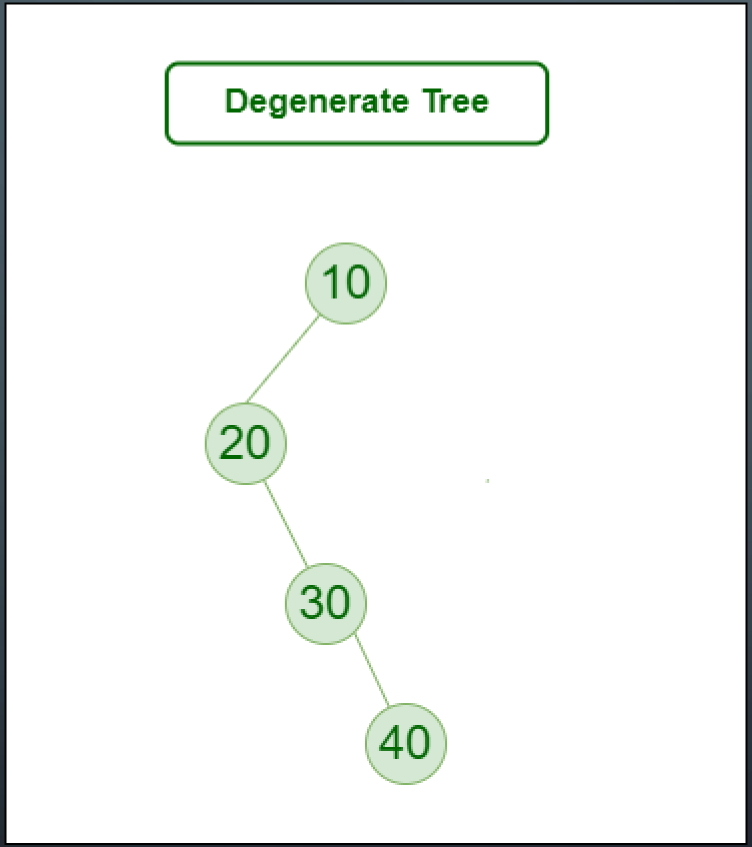
A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.



2. Degenerate (or pathological) tree

A Tree where every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is a tree having a single child either left or right.

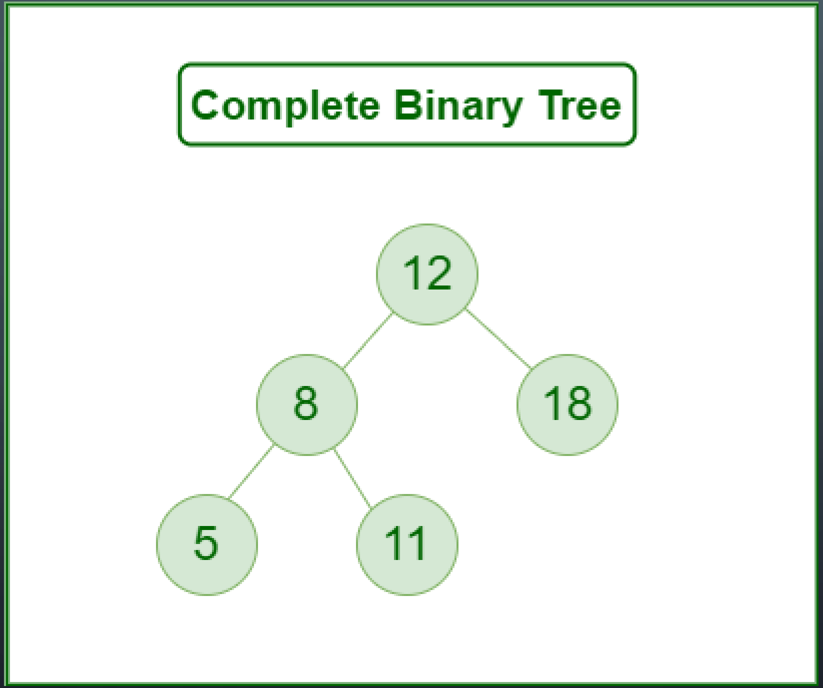


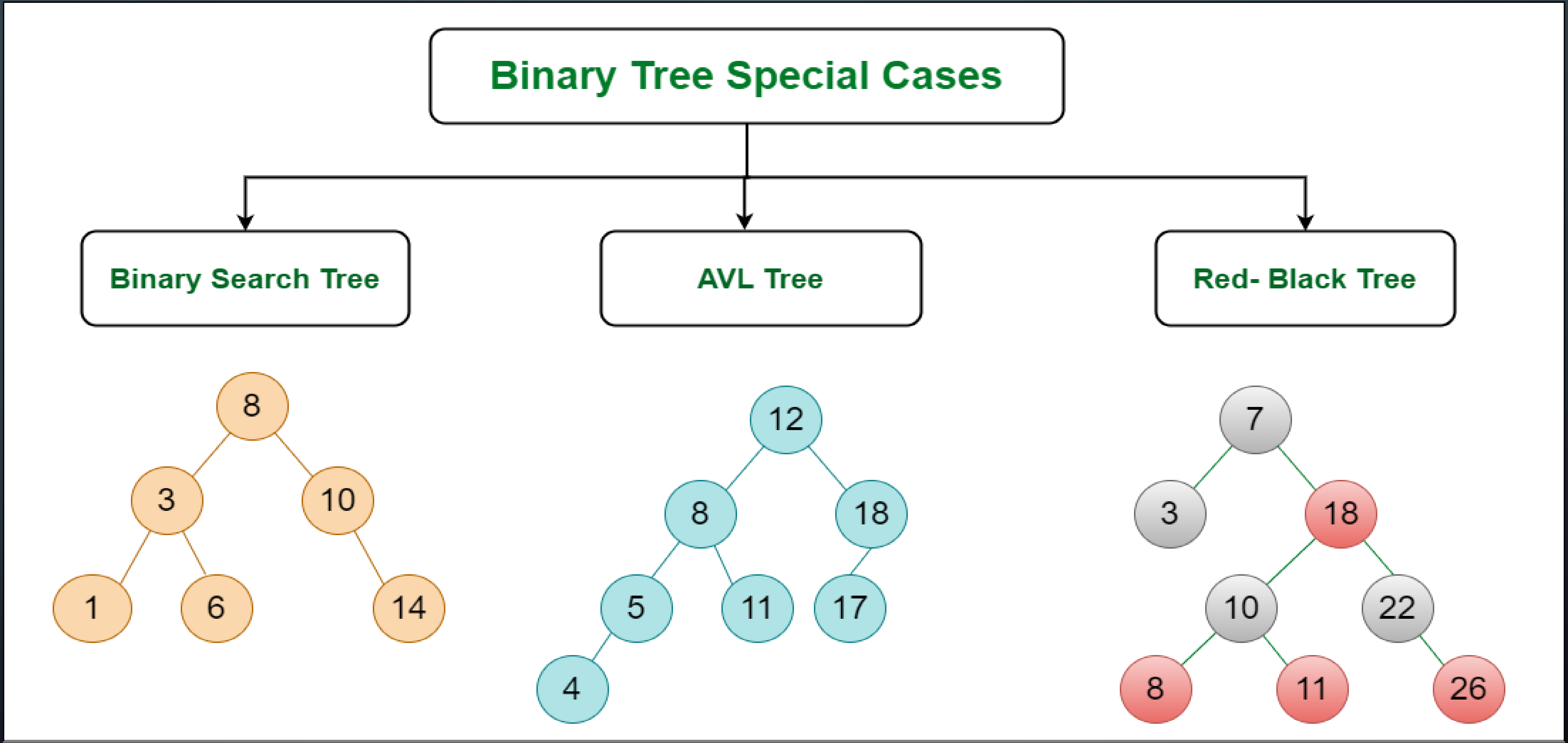
3. Complete Binary Tree

A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

A complete binary tree is just like a full binary tree, but with two major differences:

- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

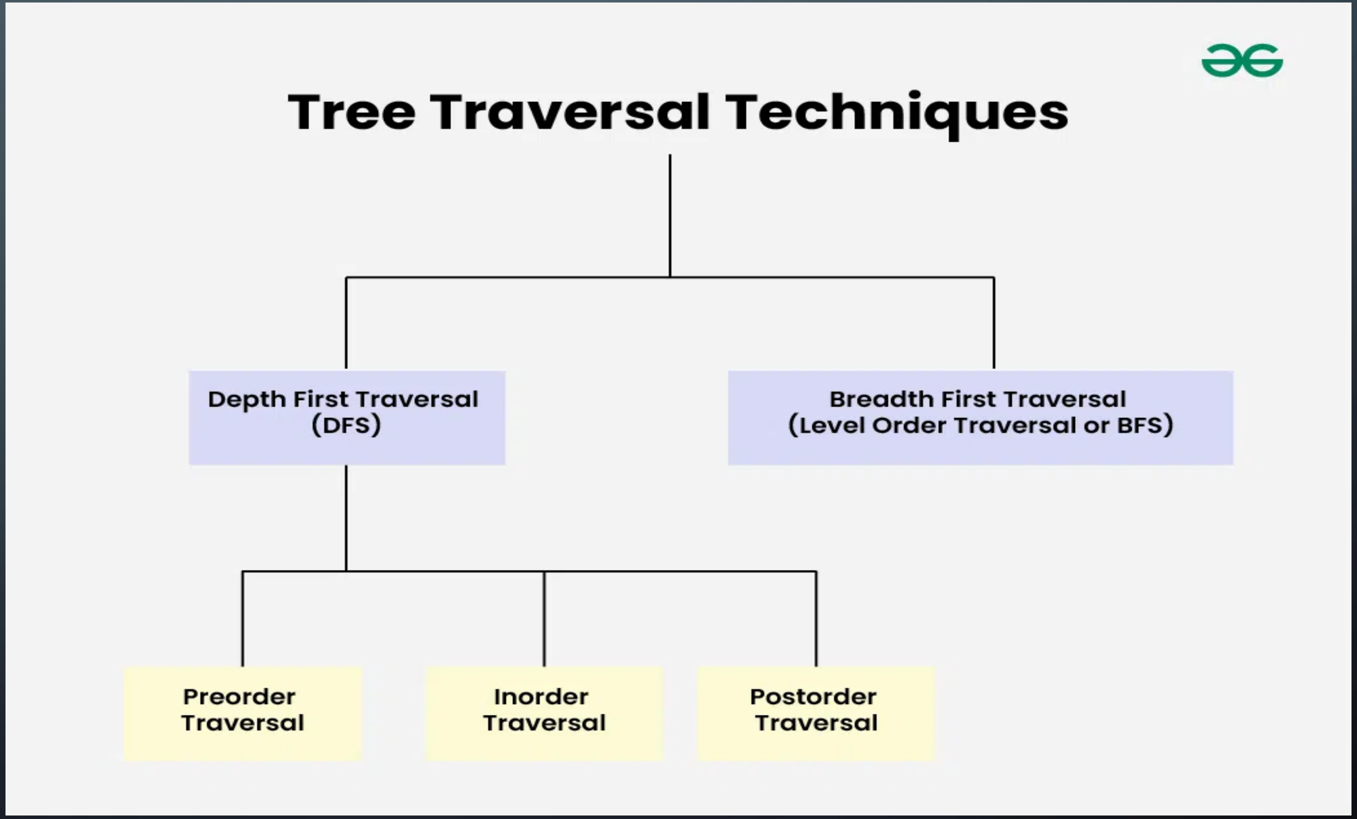



Binary Tree Special Cases
Binary Search Tree
AVL Tree
Red- Black Tree
8
3
10
1
6
14
12
8
18
5
11
17
4
7
3
18
10
22
8
11
26

TREE TRAVERSAL

Tree Traversal techniques include various ways to visit all the nodes of the tree. Tree Traversal refers to the process of visiting or accessing each node of the tree exactly once in a certain order. Tree traversal algorithms help us to visit and process all the nodes of the tree. Since tree is not a linear data structure, there are multiple nodes which we can visit after visiting a certain node. There are multiple tree traversal techniques which decide the order in which the nodes of the tree are to be visited.

Tree Traversal Techniques:


Tree Traversal Techniques
Depth First Traversal (DFS)
Breadth First Traversal (Level Order Traversal or BFS)
Preorder Traversal
Inorder Traversal
Postorder Traversal

PREORDER TRAVERSAL

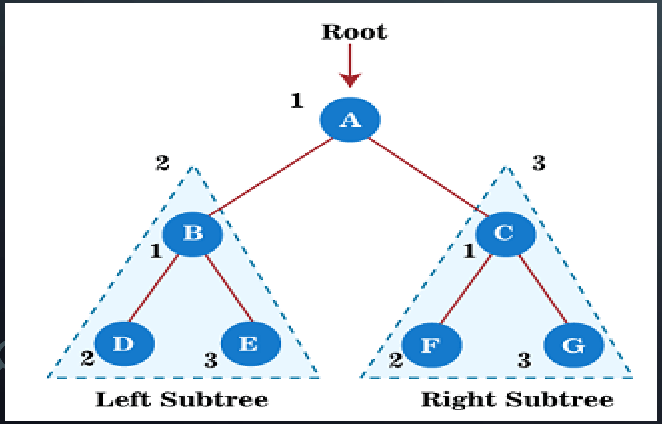
This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

So, in a preorder traversal, each node is visited before both of its subtrees.

Algorithm

Until all nodes of the tree are not visited

1. Step 1 - Visit the root node
2. Step 2 - Traverse the left subtree recursively.
3. Step 3 - Traverse the right subtree recursively.



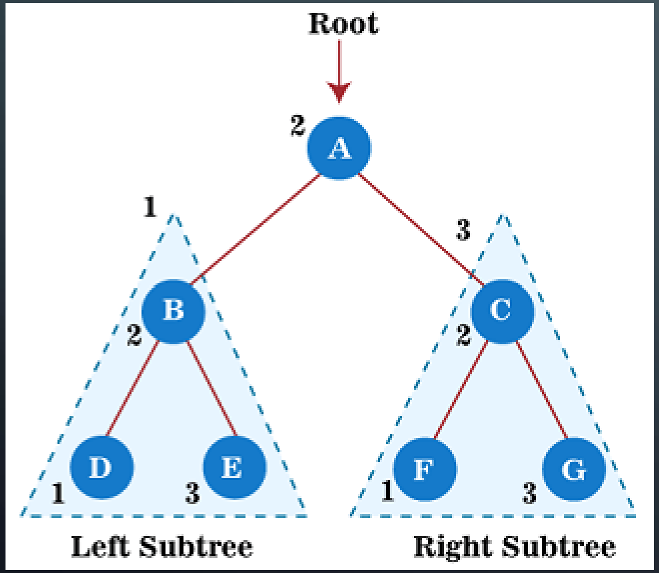
So, the output of the preorder traversal of the above tree is -
A → B → D → E → C → F → G

INORDER TRAVERSAL

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

Algorithm

```
Until all nodes of the tree are not visited
    Step 1 - Traverse the left subtree recursively.
    Step 2 - Visit the root node.
    Step 3 - Traverse the right subtree recursively.
```



The output of the inorder traversal of the above tree is -

D → B → E → A → F → C → G

POSTORDER TRAVERSAL

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

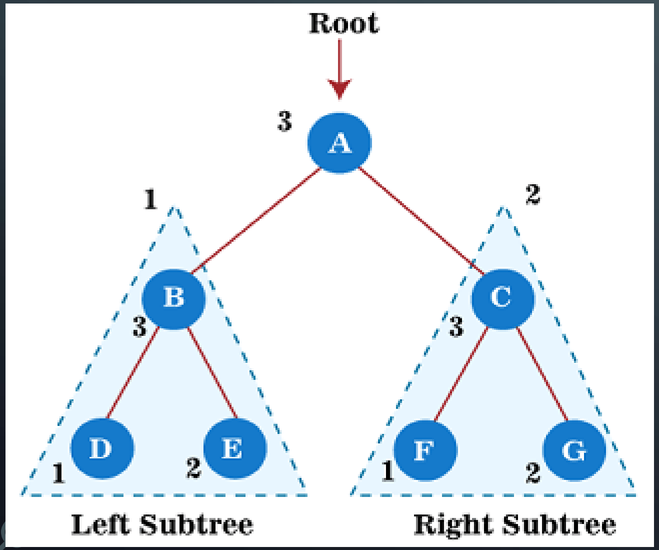
Algorithm

Until all nodes of the tree are not visited

Step 1 - Traverse the left subtree recursively.

Step 2 - Traverse the right subtree recursively.

Step 3 - Visit the root node.

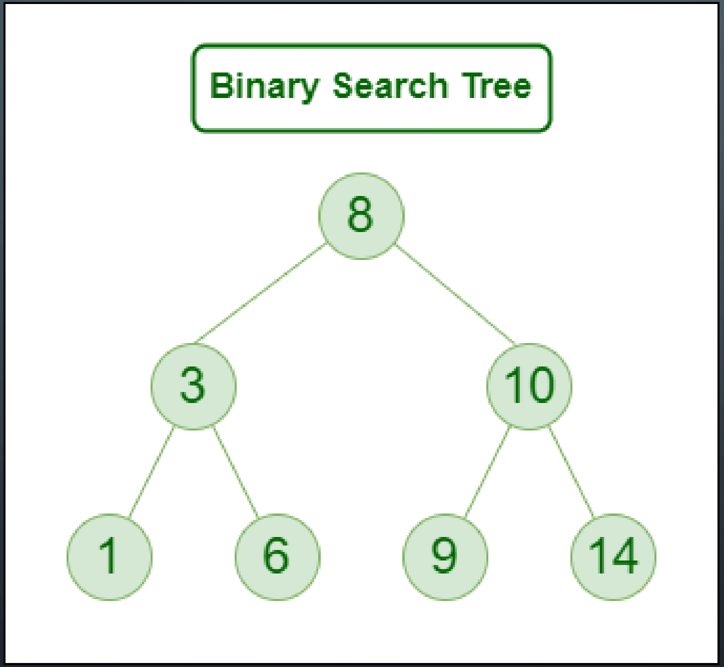


So, the output of the postorder traversal of the above tree is -
D → E → B → F → G → C → A

BST

A **binary Search Tree** is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.



Self Study

Practice

Week 7

Introduction to Graph & Representation

Page
86-100

GRAPH

A **Graph** is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E)**.

Representations of Graph

Here are the two most common ways to represent a graph.

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix Representation

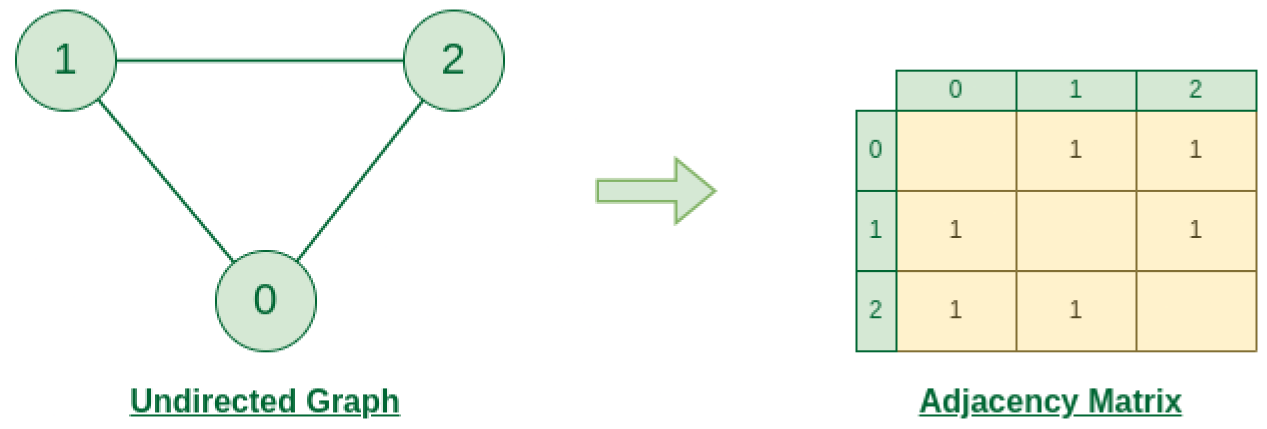
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)

Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- If there is an edge from vertex i to j, mark adjMat[i][j] as 1.
- If there is no edge from vertex i to j, mark adjMat[i][j] as 0.

Representation of Undirected Graph as Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat**[**destination])** because we can go either way.

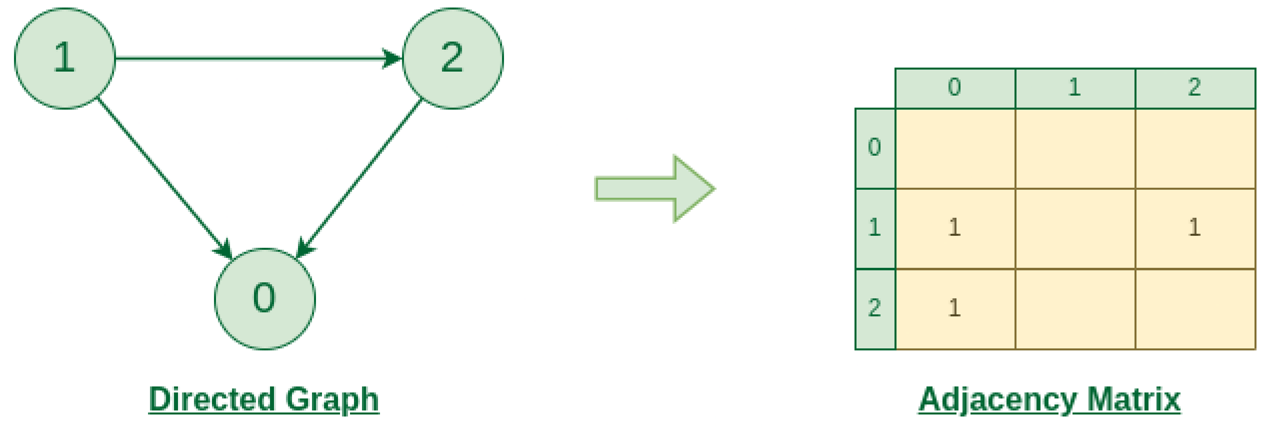


	0	1	2
0		1	1
1	1		1
2	1	1	

Graph Representation of Undirected graph to Adjacency Matrix

Representation of Directed Graph as Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



	0	1	2
0			
1	1		1
2	1		

Graph Representation of Directed graph to Adjacency Matrix

Adjacency List Representation

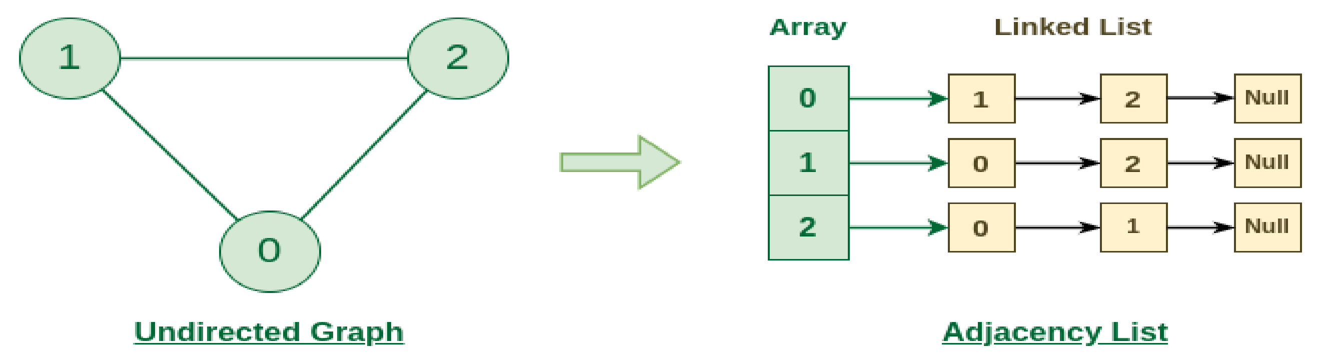
An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.

Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- *adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.*
- *adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.*

Representation of Undirected Graph as Adjacency list:

The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Undirected graph to Adjacency List

TYPES OF GRAPH

1. **Undirected Graphs:** A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal. Example: A social network graph where friendships are not directional.
2. **Directed Graphs:** A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal. Example: A web page graph where links between pages are directional.
3. **Weighted Graphs:** A graph in which edges have weights or costs associated with them. Example: A road network graph where the weights can represent the distance between two cities.
4. **Unweighted Graphs:** A graph in which edges have no weights or costs associated with them. Example: A social network graph where the edges represent friendships.
5. **Complete Graphs:** A graph in which each vertex is connected to every other vertex. Example: A tournament graph where every player plays against every other player.
6. **Bipartite Graphs:** A graph in which the vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set. Example: A job applicant graph where the vertices can be divided into job applicants and job openings.

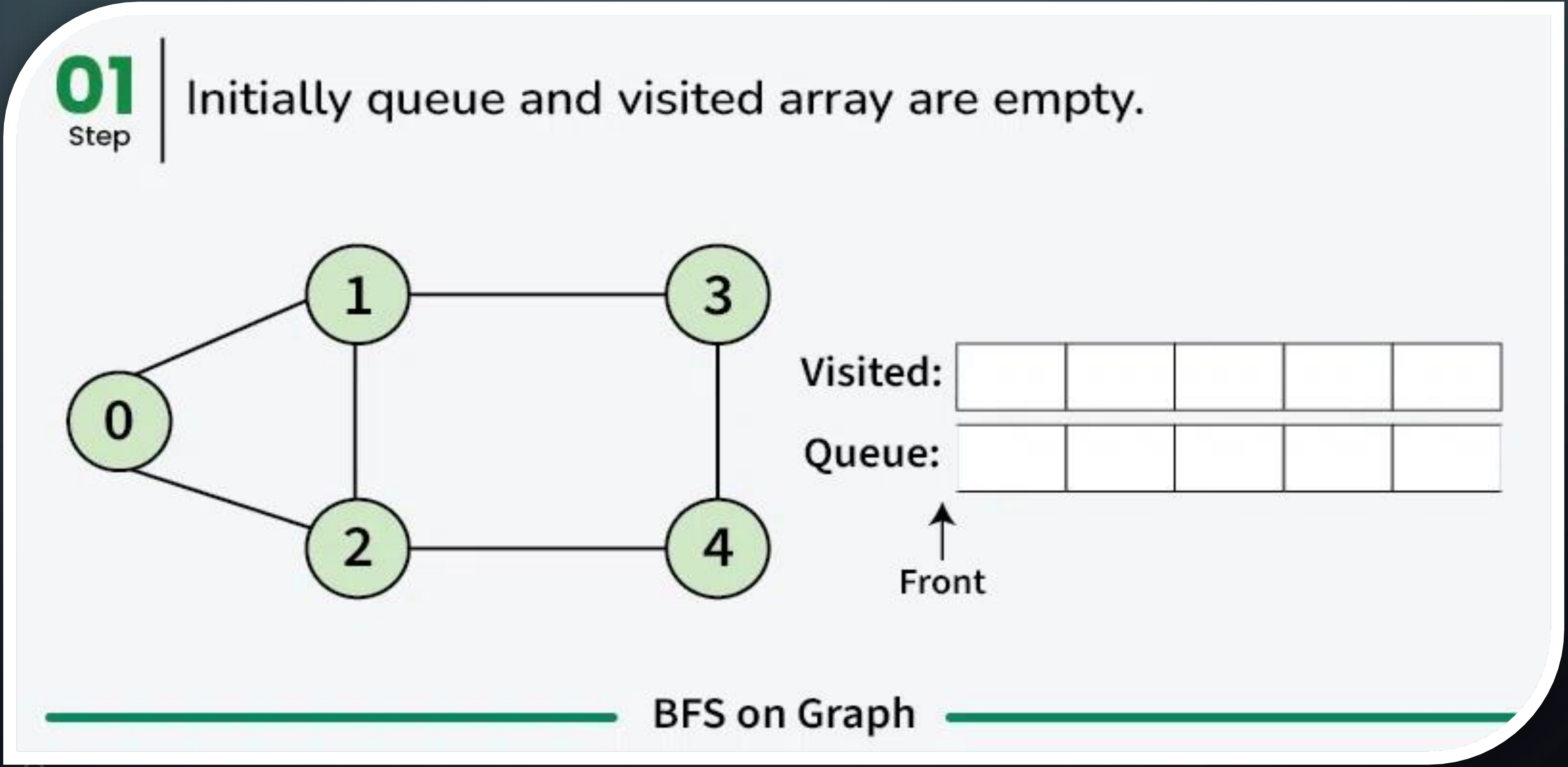
BFS

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent.

Algorithm:

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until QUEUE is empty
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]
Step 6: EXIT
```

DEMONSTRATION WITH EXAMPLE

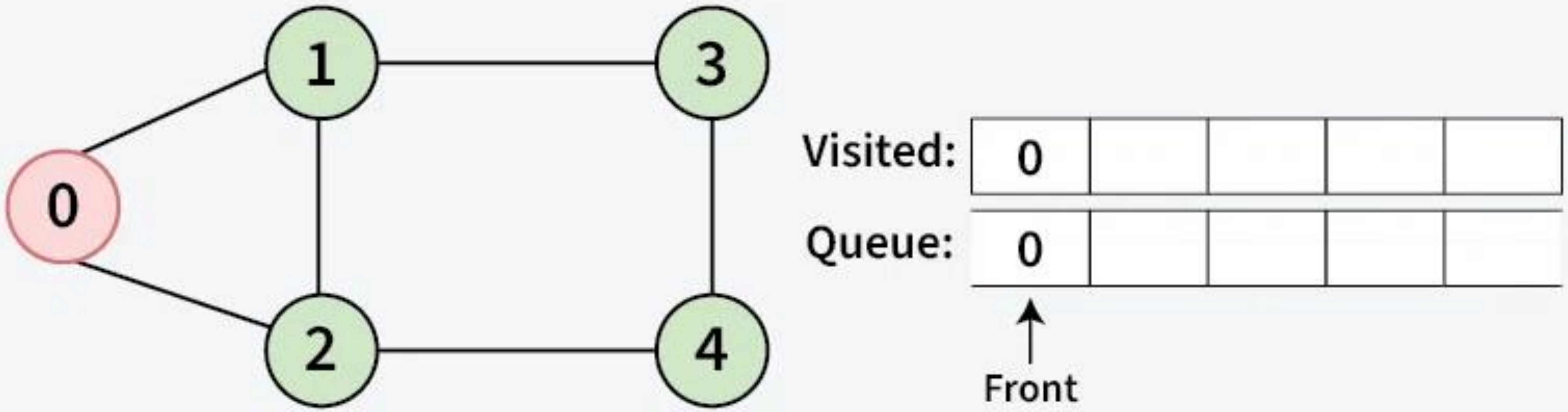

01 Step

Initially queue and visited array are empty.

BFS on Graph

02 Step

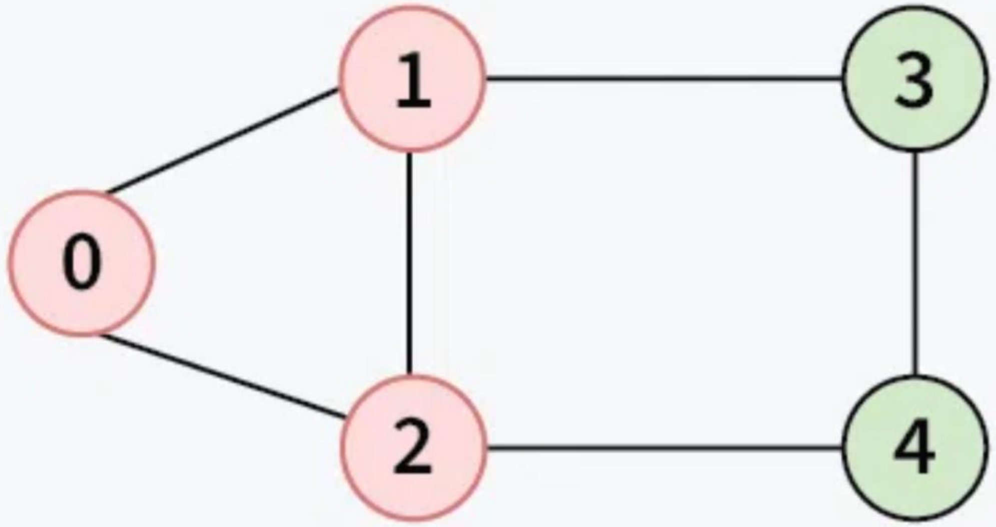
Push 0 into queue and mark it visited.



BFS on Graph

03 Step

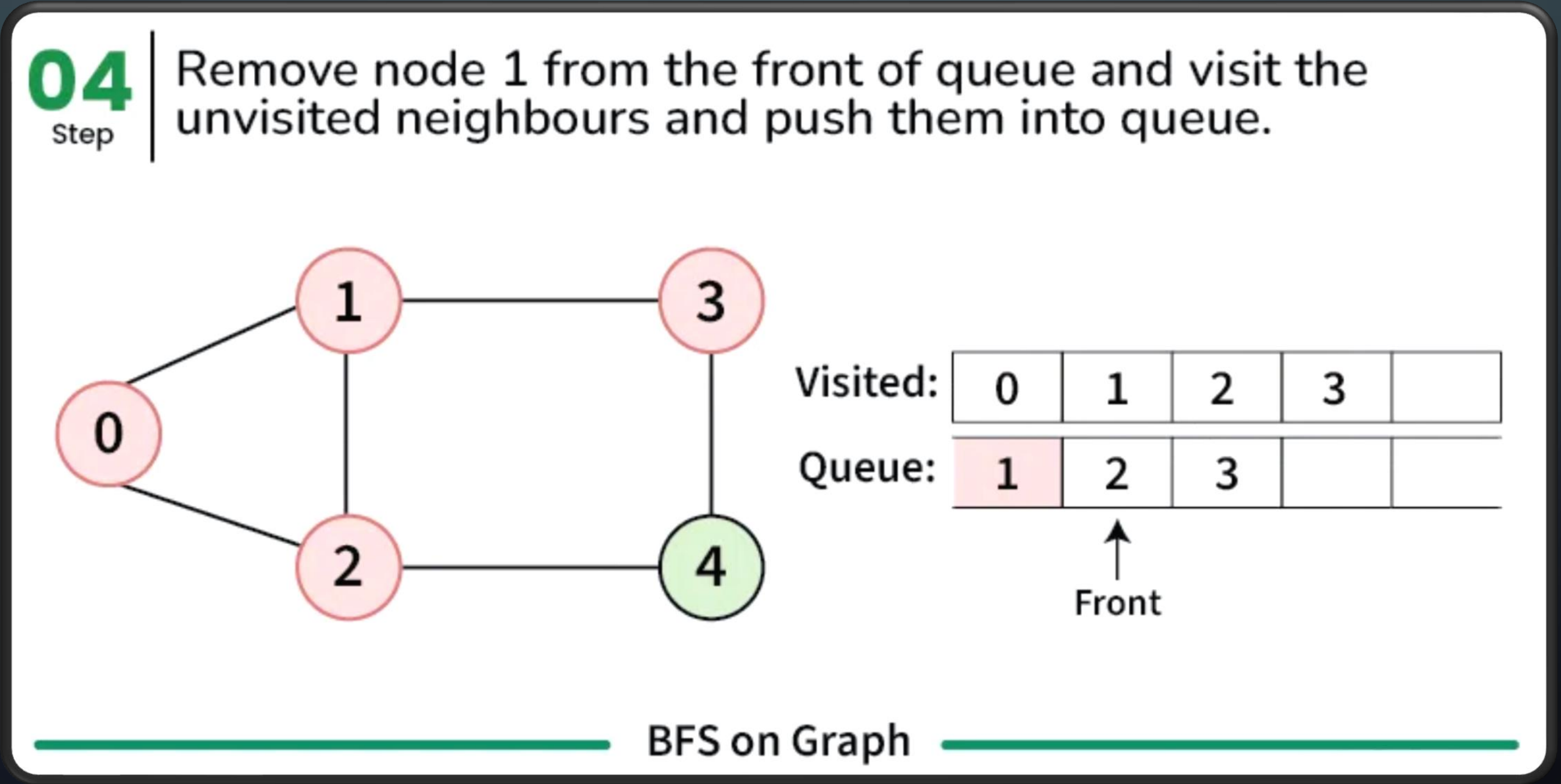
Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2		
Queue:	🟥 0	1	2		

↑ Front

BFS on Graph



04 Step

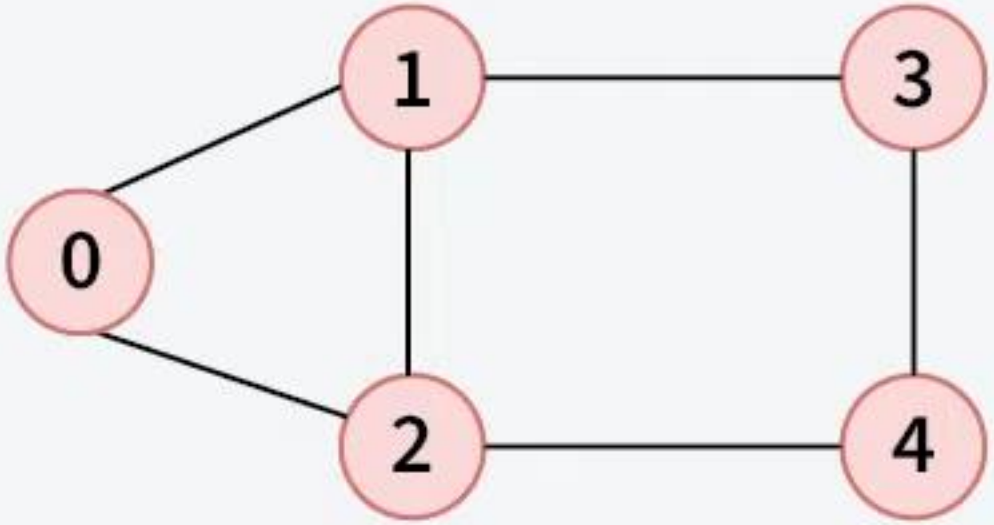
Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.

Visited:	0	1	2	3	
Queue:	1	2	3		

BFS on Graph

05 Step

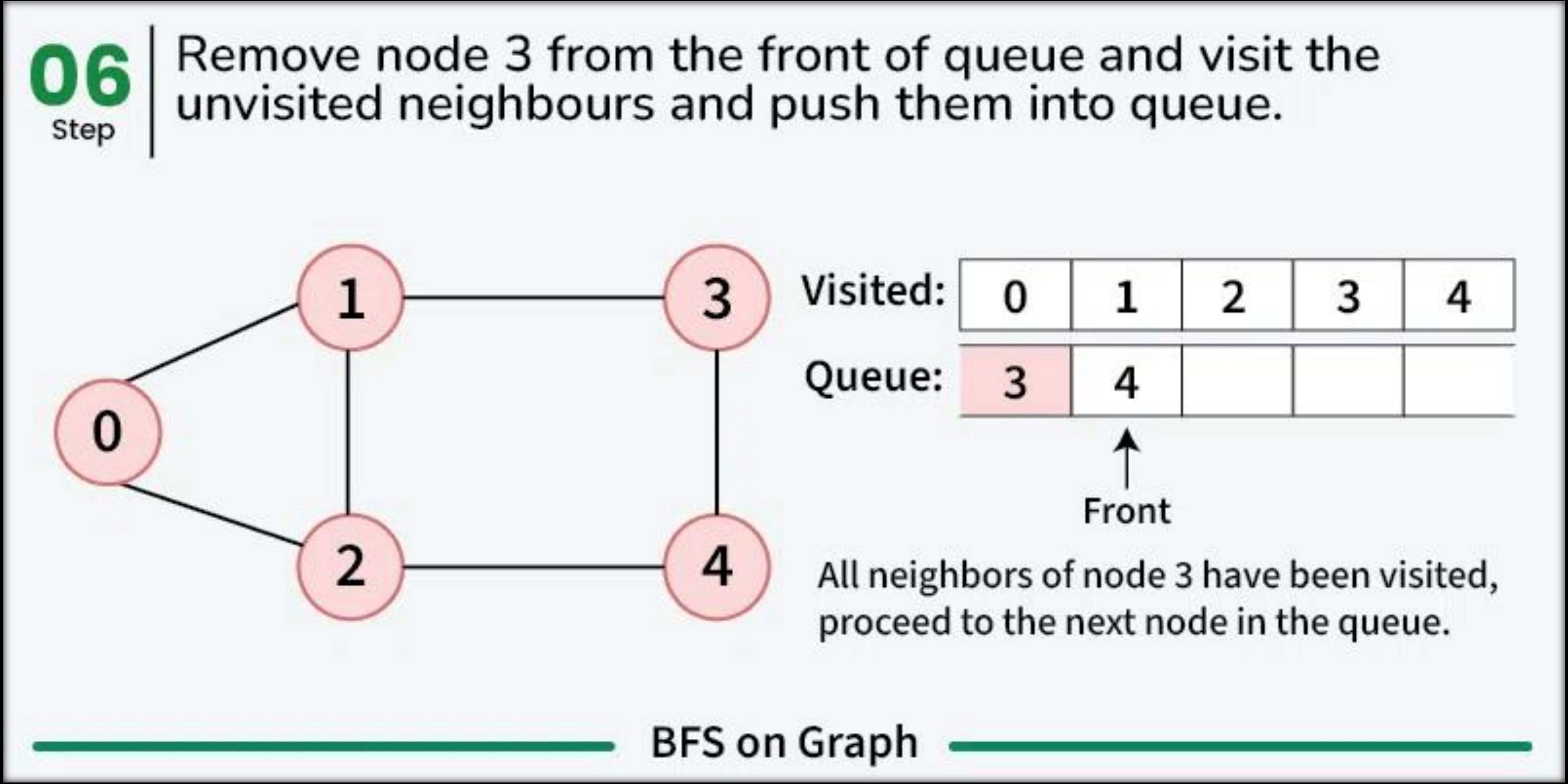
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2	3	4
Queue:	2	3	4		

↑ Front

BFS on Graph



06 Step

Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

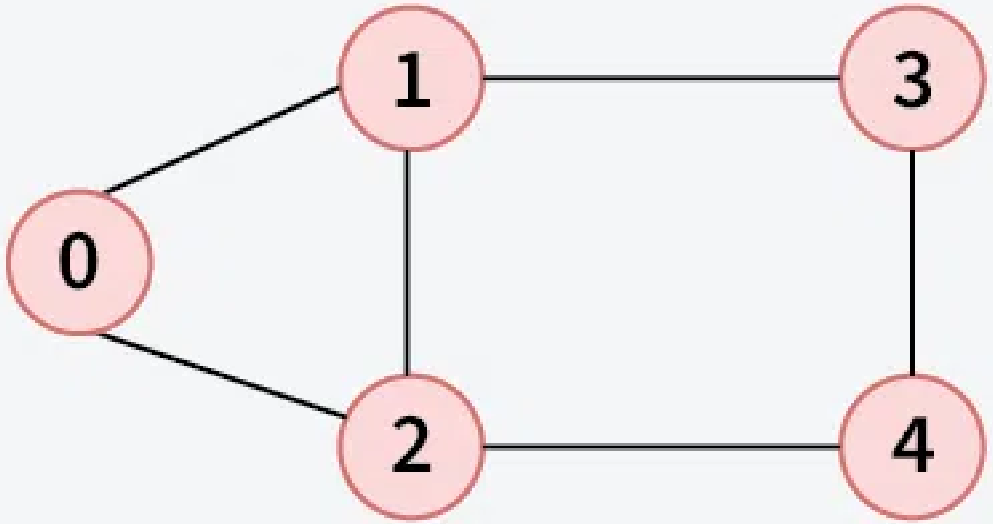
Visited:	0	1	2	3	4
Queue:	3	4			

All neighbors of node 3 have been visited, proceed to the next node in the queue.

BFS on Graph

07
Step

Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2	3	4
Queue:	4				

↑
Front

All neighbors of node 4 have been visited, proceed to the next node in the queue.

BFS on Graph

Self Study

Practice

Week 8

DFS & Shortest Path Algorithm

Page
102-109

DFS

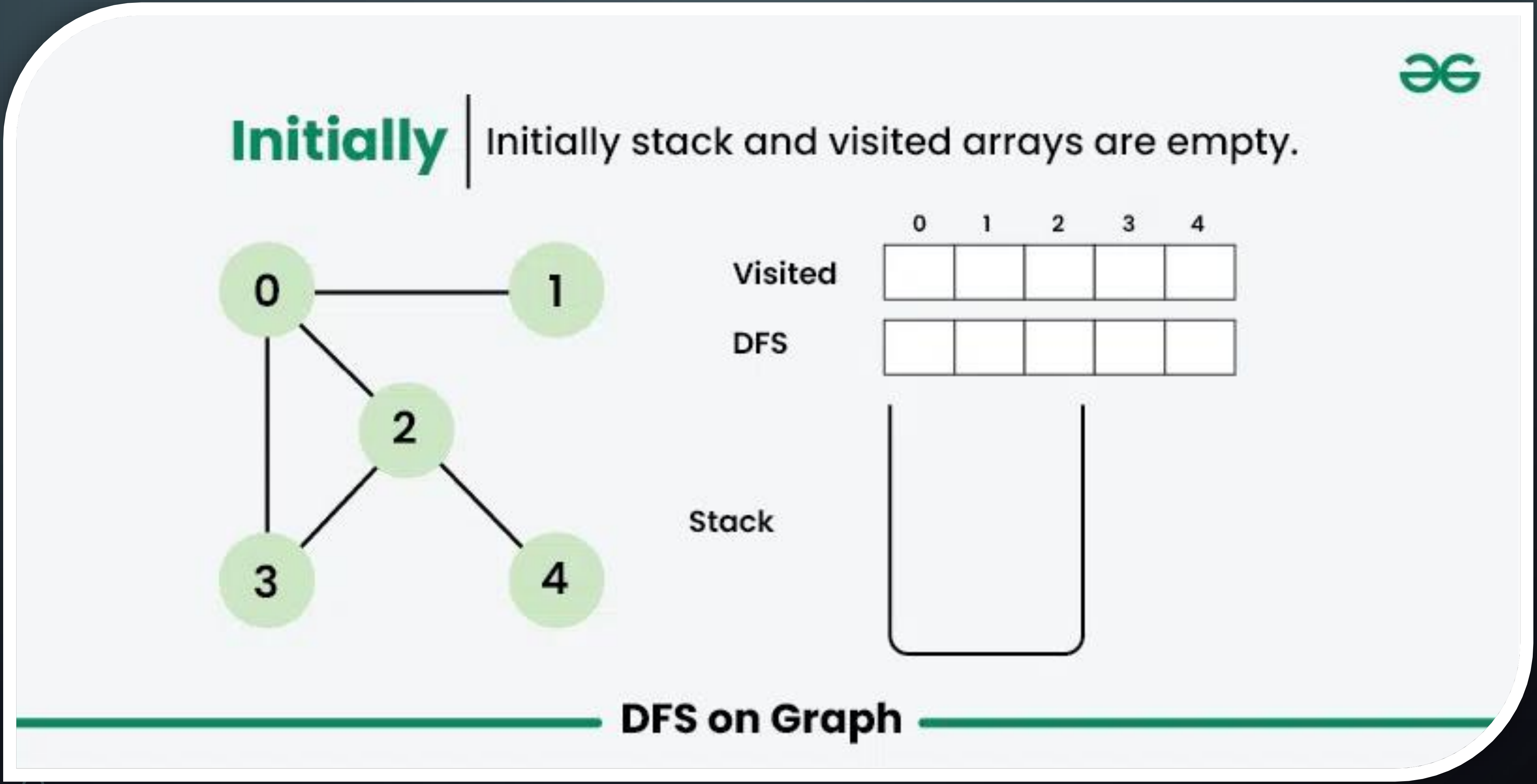
Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Algorithm:

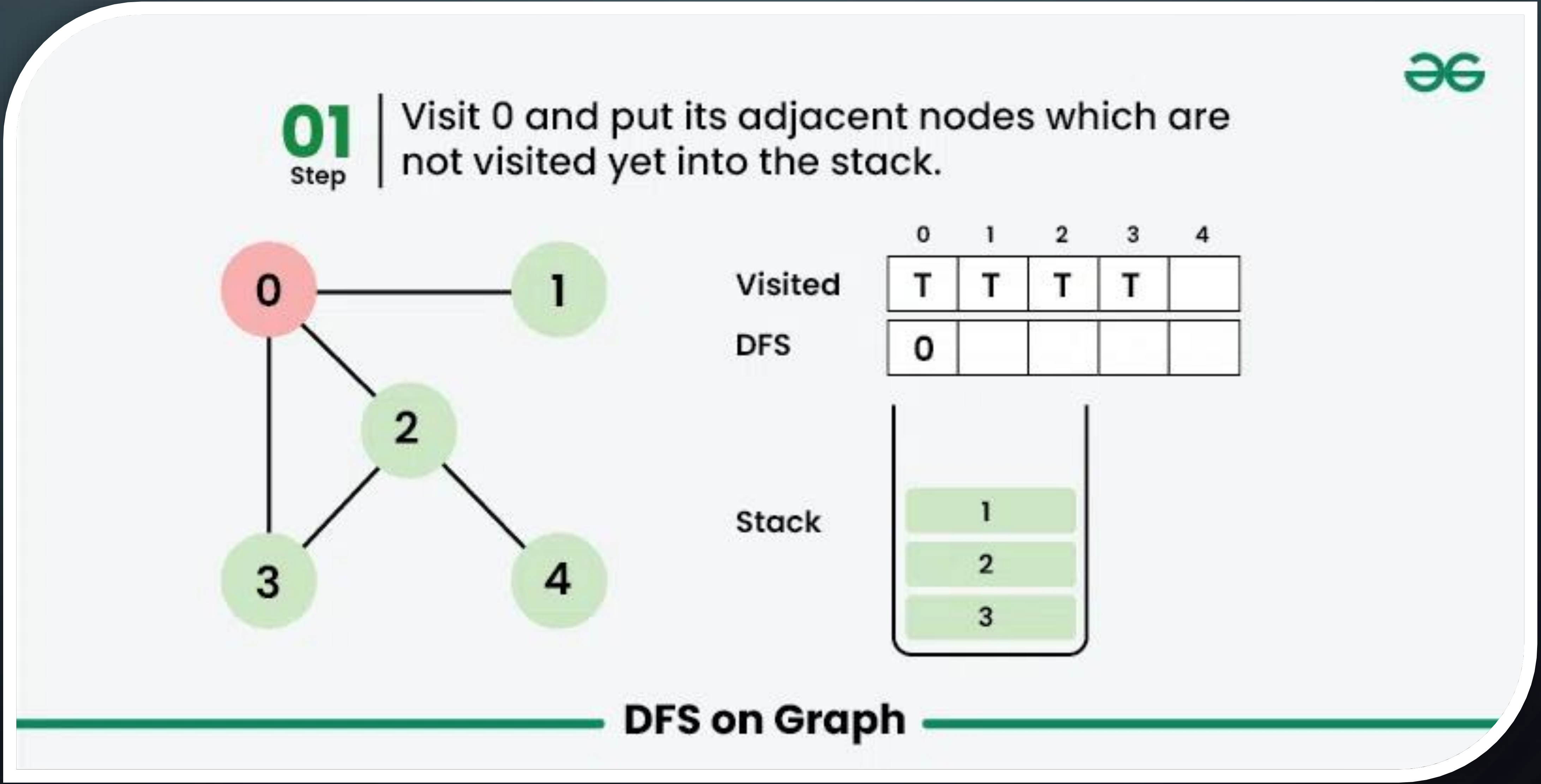
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

DEMONSTRATION WITH EXAMPLE


Initially | Initially stack and visited arrays are empty.

DFS on Graph

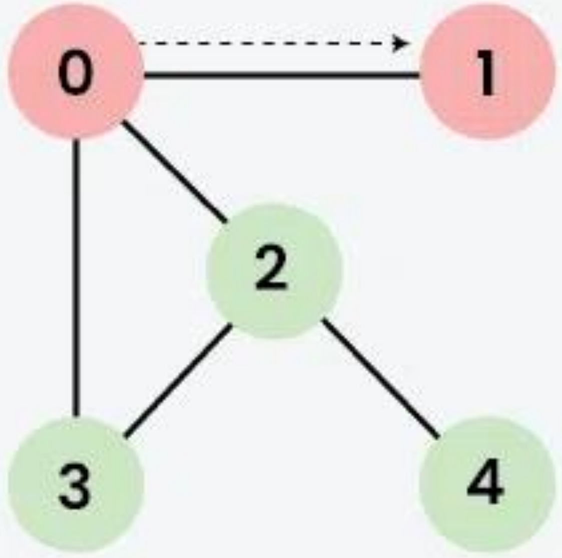


01 Step | Visit 0 and put its adjacent nodes which are not visited yet into the stack.

	0	1	2	3	4
Visited	T	T	T	T	
DFS	0				

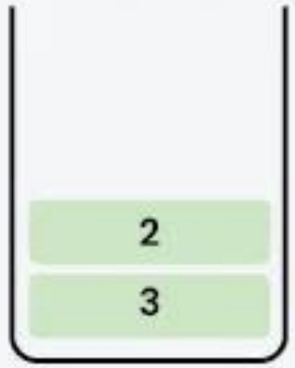
DFS on Graph

02 Step | Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



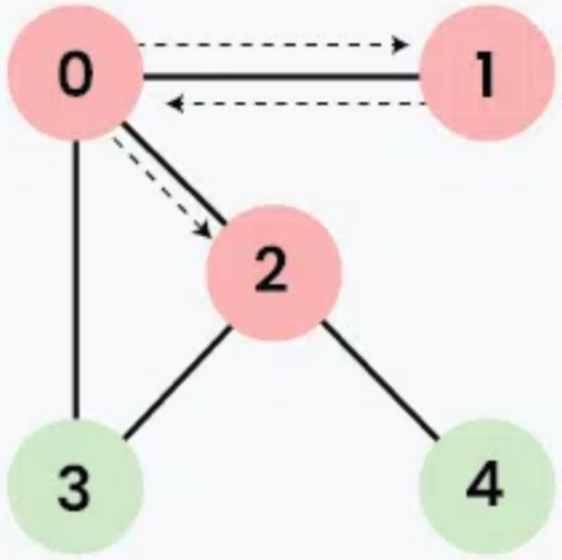
	0	1	2	3	4
Visited	T	T	T	T	
DFS	0	1			

Stack

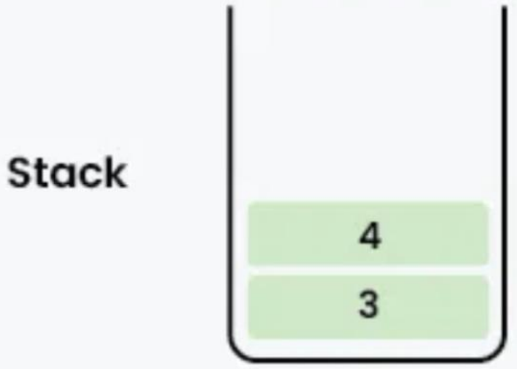


DFS on Graph

03 Step | Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

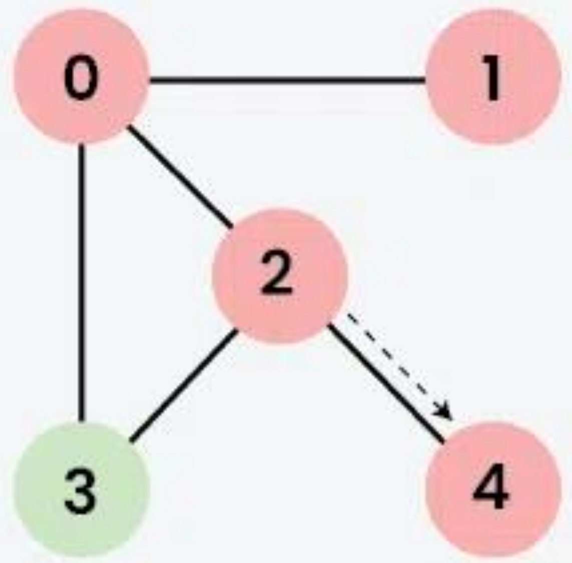


	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2		

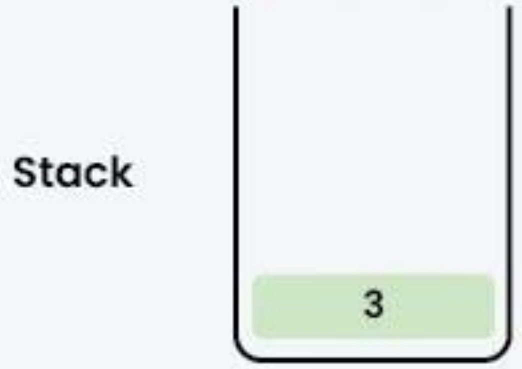


DFS on Graph

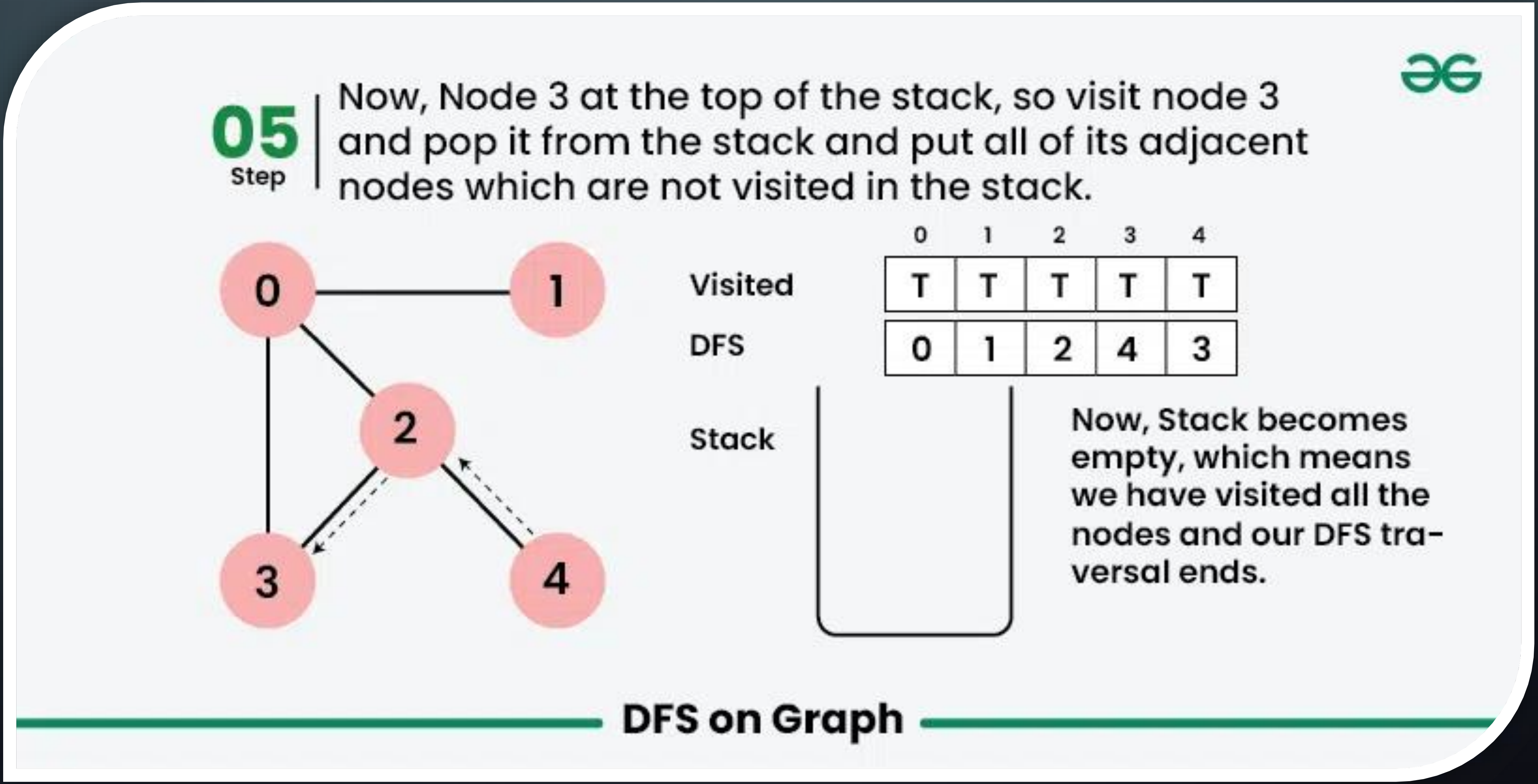
04 Step | Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	

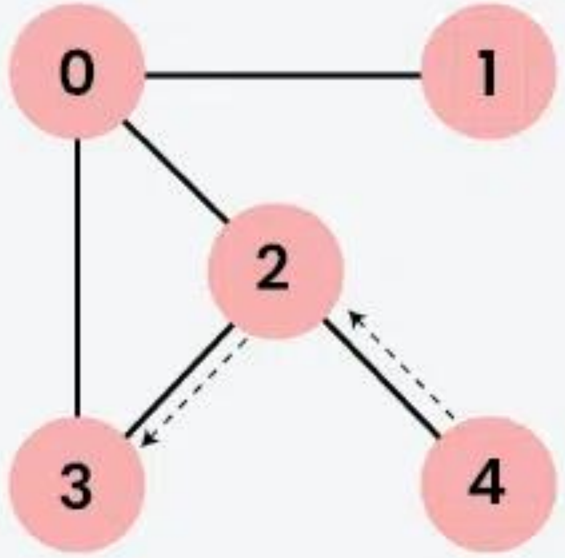


DFS on Graph



05 Step

Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	3

Stack

Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

DFS on Graph

SELF STUDY

CODING PRACTICE

Week 9

Minimum Spanning Tree

Page
111-130

MST

A minimum spanning tree (MST) is defined as a spanning tree that has the minimum weight among all the possible spanning trees.

There are several algorithms to find the minimum spanning tree from a given graph, some of them are listed below:

- Kruskal’s Minimum Spanning Tree Algorithm
- Prim’s Minimum Spanning Tree Algorithm

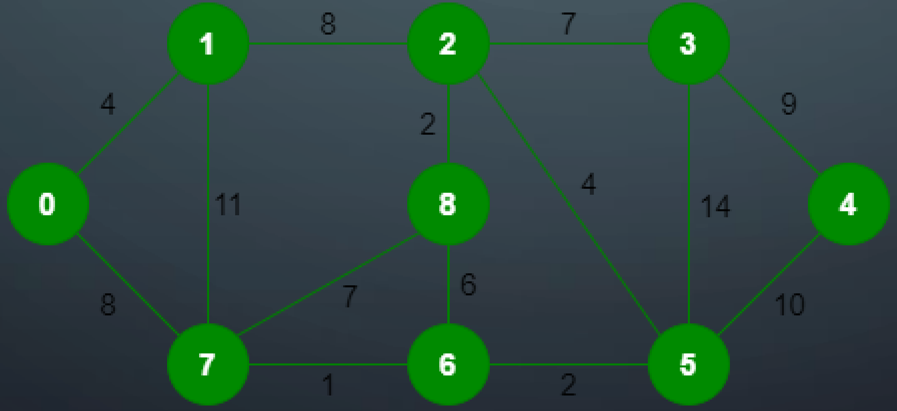
KRUSKAL'S MINIMUM SPANNING TREE ALGORITHM

Below are the steps for finding MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

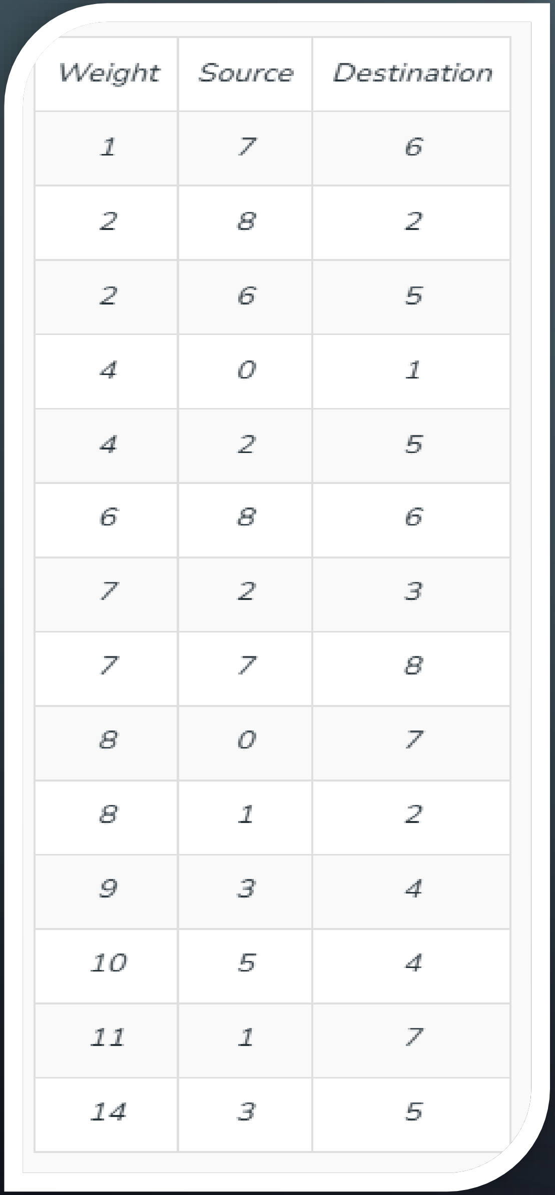
Illustration:

Below is the illustration of the above approach:



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

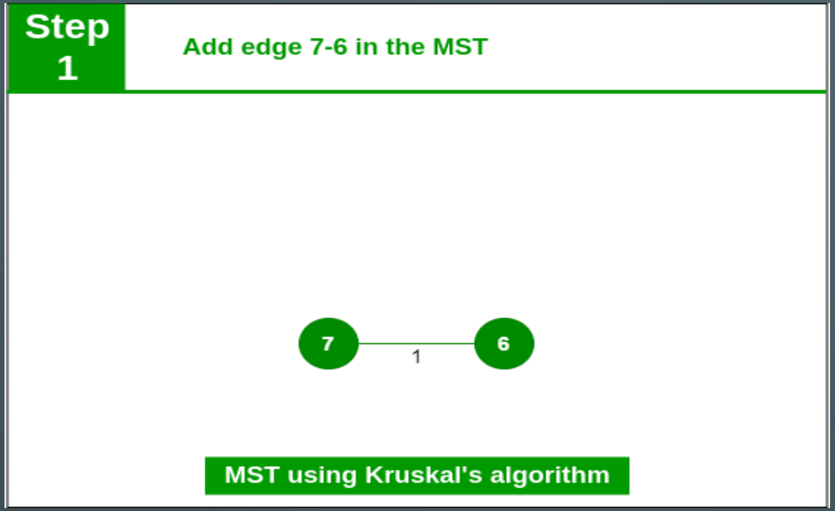
After sorting:



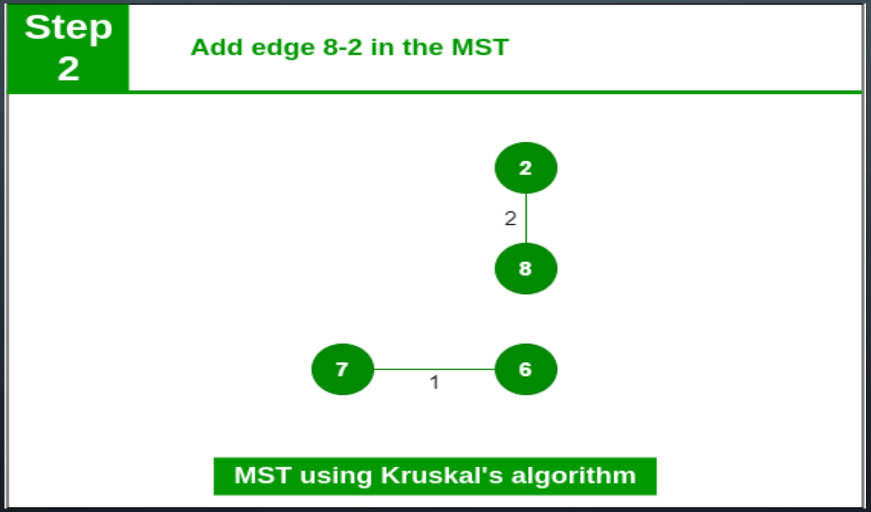
Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from the sorted list of edges

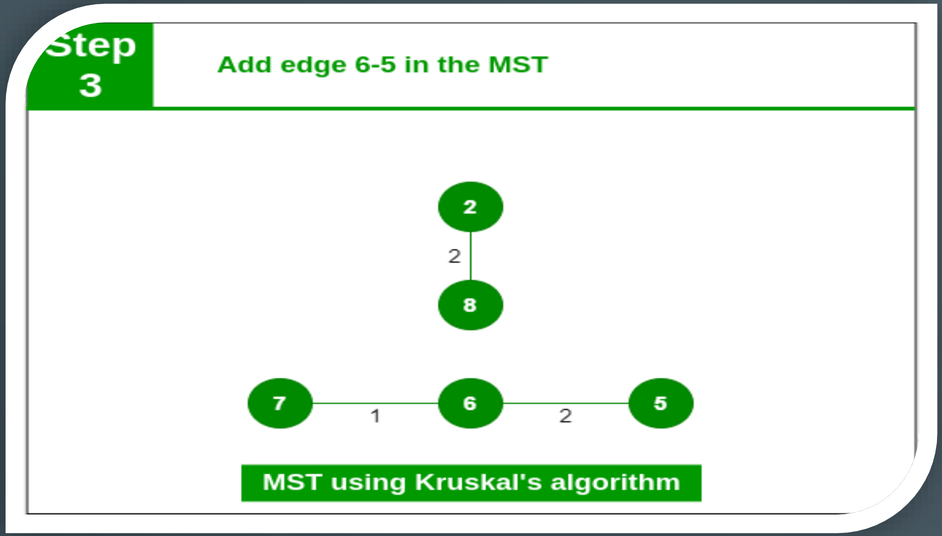
Step 1: Pick edge 7-6. No cycle is formed, include it.



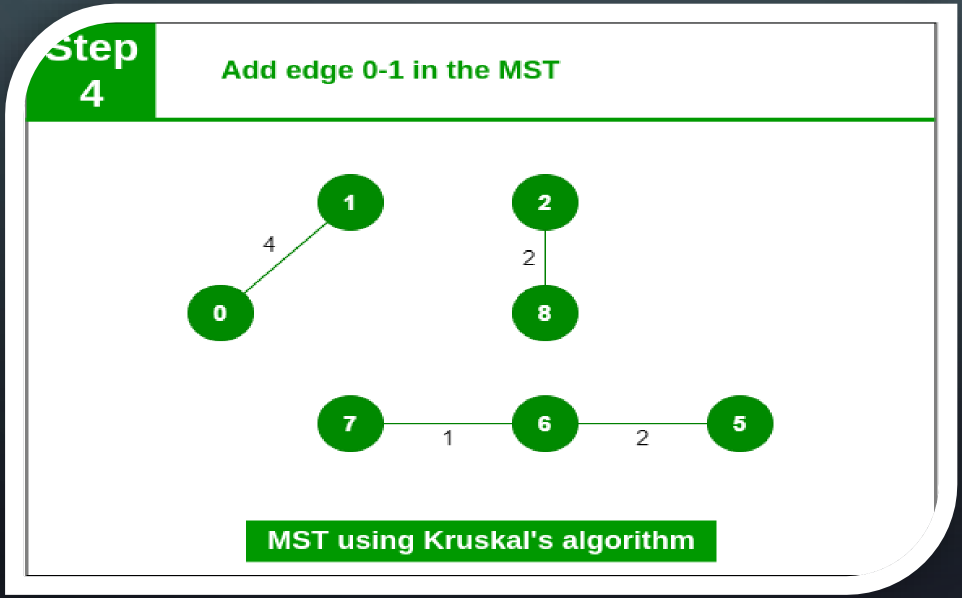
Step 2: *Pick edge 8-2. No cycle is formed, include it.*



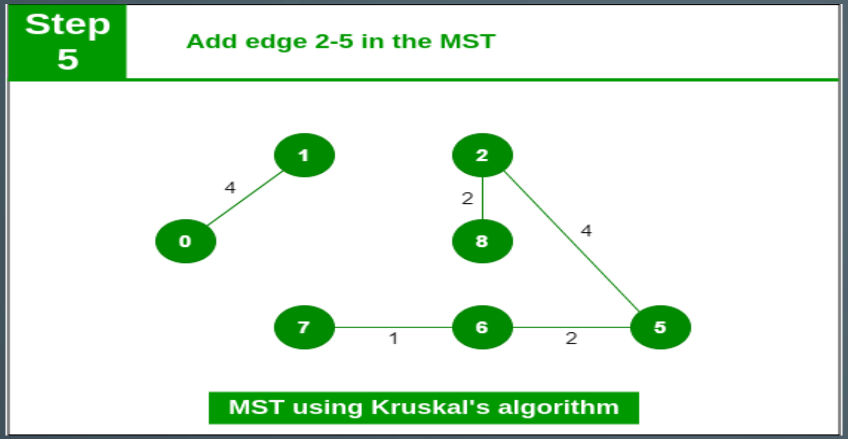
Step 3: *Pick edge 6-5. No cycle is formed, include it.*



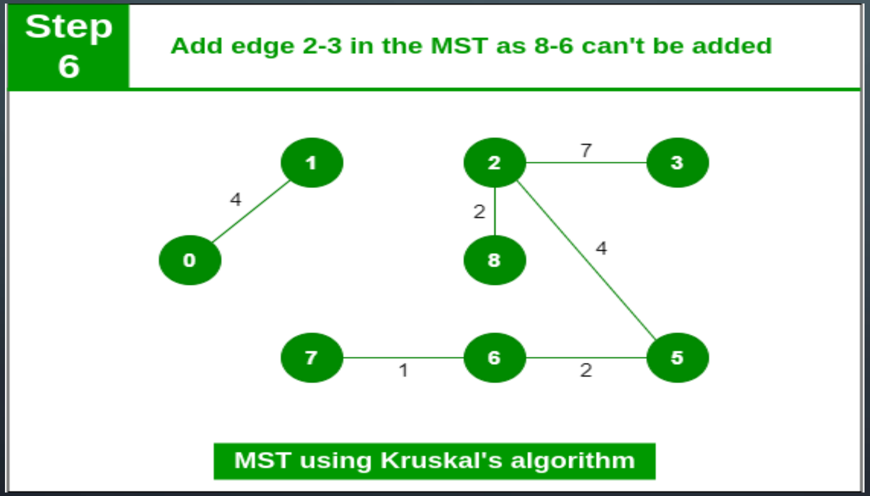
Step 4: Pick edge 0-1. No cycle is formed, include it.



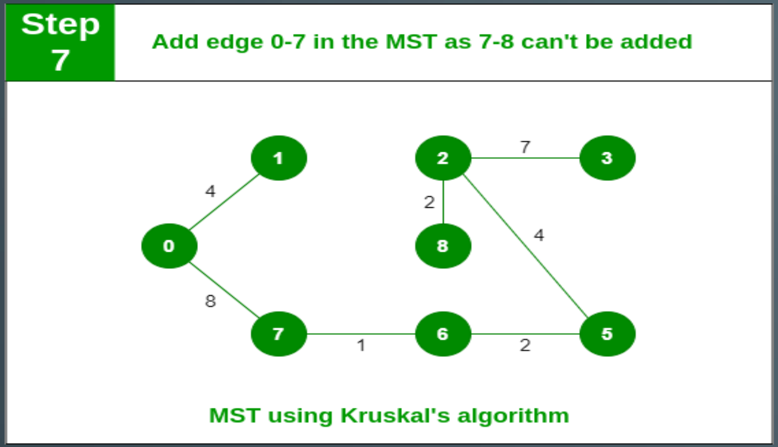
Step 5: *Pick edge 2-5. No cycle is formed, include it.*



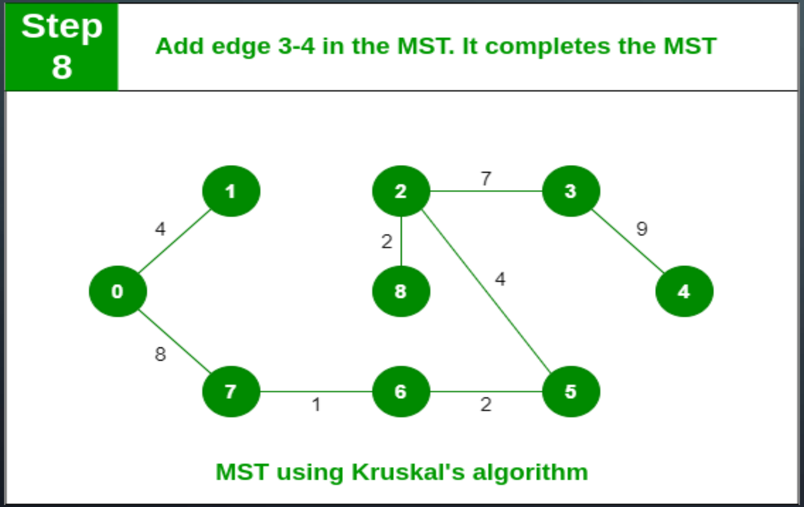
Step 6: *Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.*


Step 6
Add edge 2-3 in the MST as 8-6 can't be added
1
2
7
3
4
2
0
8
4
7
6
5
1
2
MST using Kruskal's algorithm

Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.


Step 7
Add edge 0-7 in the MST as 7-8 can't be added
1
2
7
3
4
2
0
8
4
8
7
1
6
2
5
MST using Kruskal's algorithm

***Step 8:** Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.*


Step 8
Add edge 3-4 in the MST. It completes the MST
1
2
7
3
9
4
2
0
8
4
4
8
7
1
6
2
5
MST using Kruskal's algorithm

PRIM’S MINIMUM SPANNING TREE ALGORITHM

Step 1: *Determine an arbitrary vertex as the starting vertex of the MST.*

Step 2: *Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).*

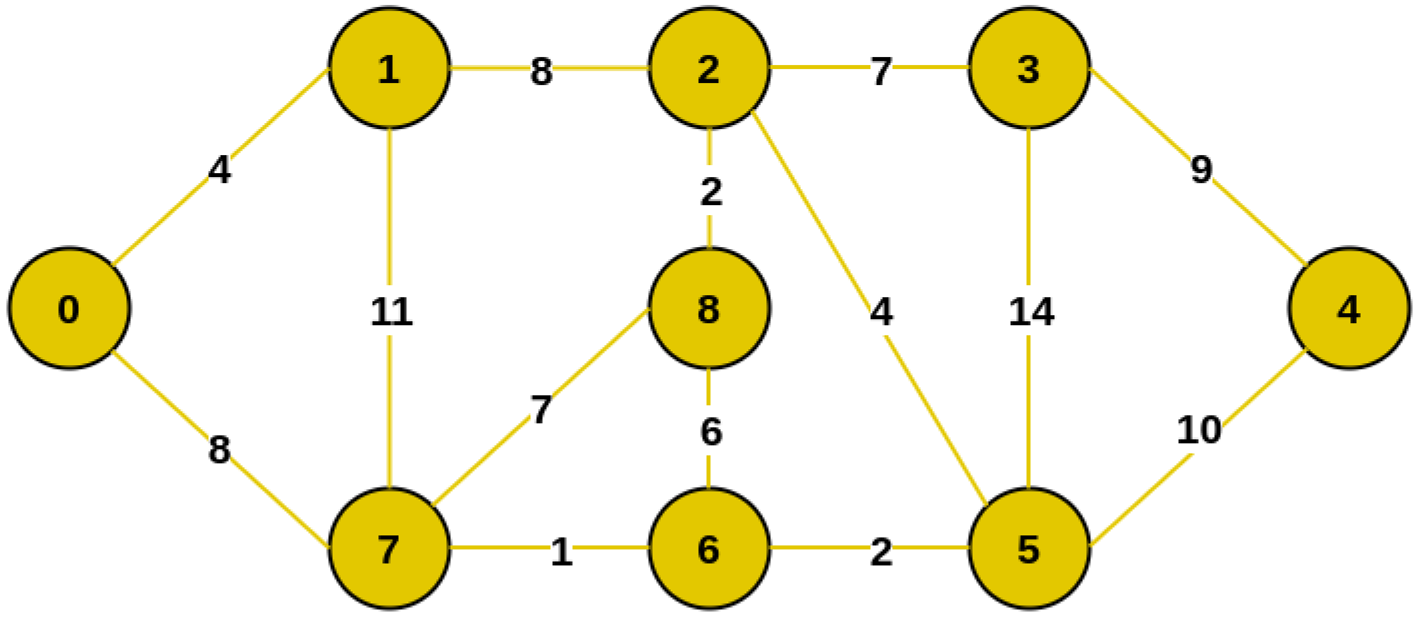
Step 3: *Find edges connecting any tree vertex with the fringe vertices.*

Step 4: *Find the minimum among these edges.*

Step 5: *Add the chosen edge to the MST if it does not form any cycle.*

Step 6: *Return the MST and exit*

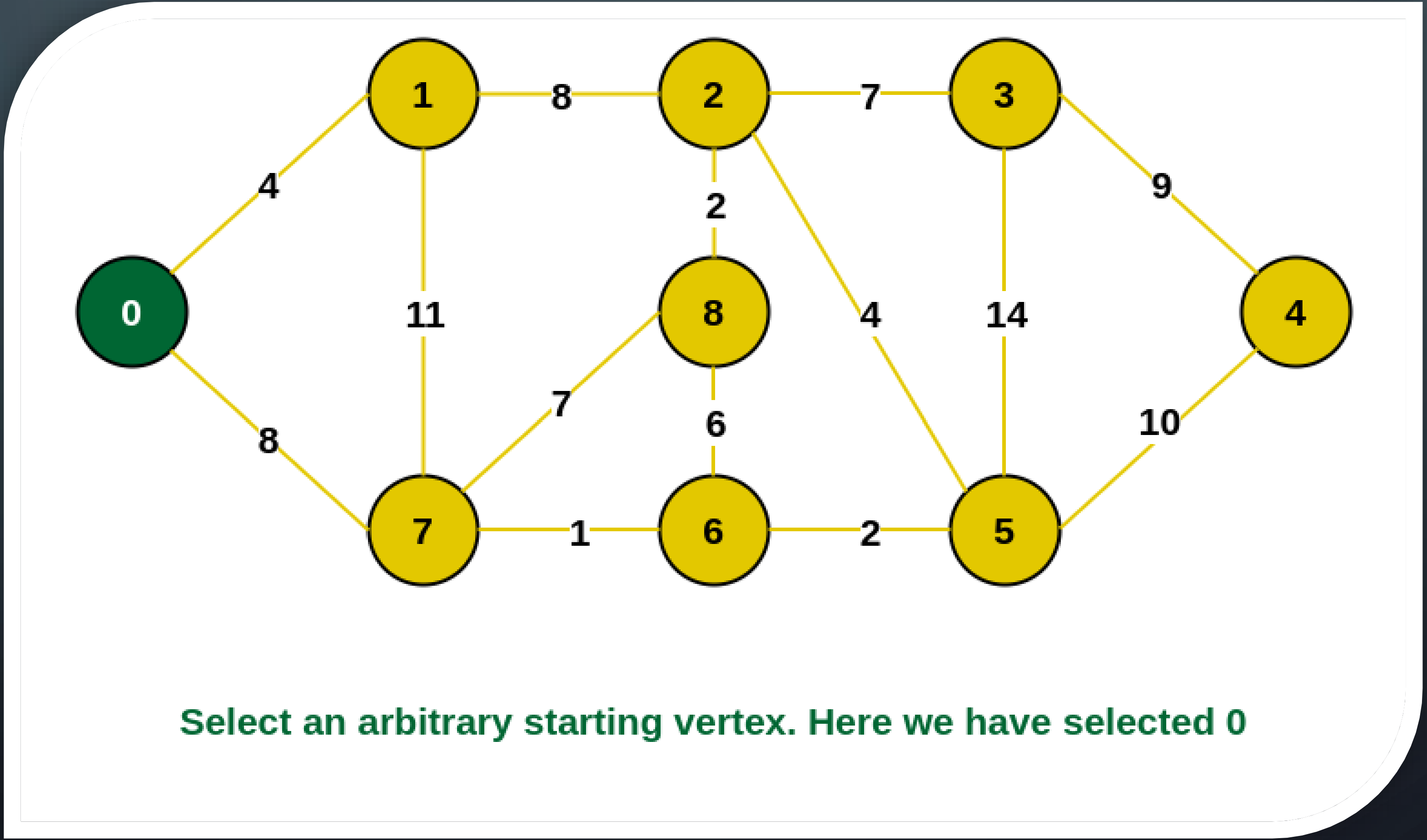
Consider the following graph as an example for which we need to find the Minimum Spanning Tree (MST).



Example of a Graph

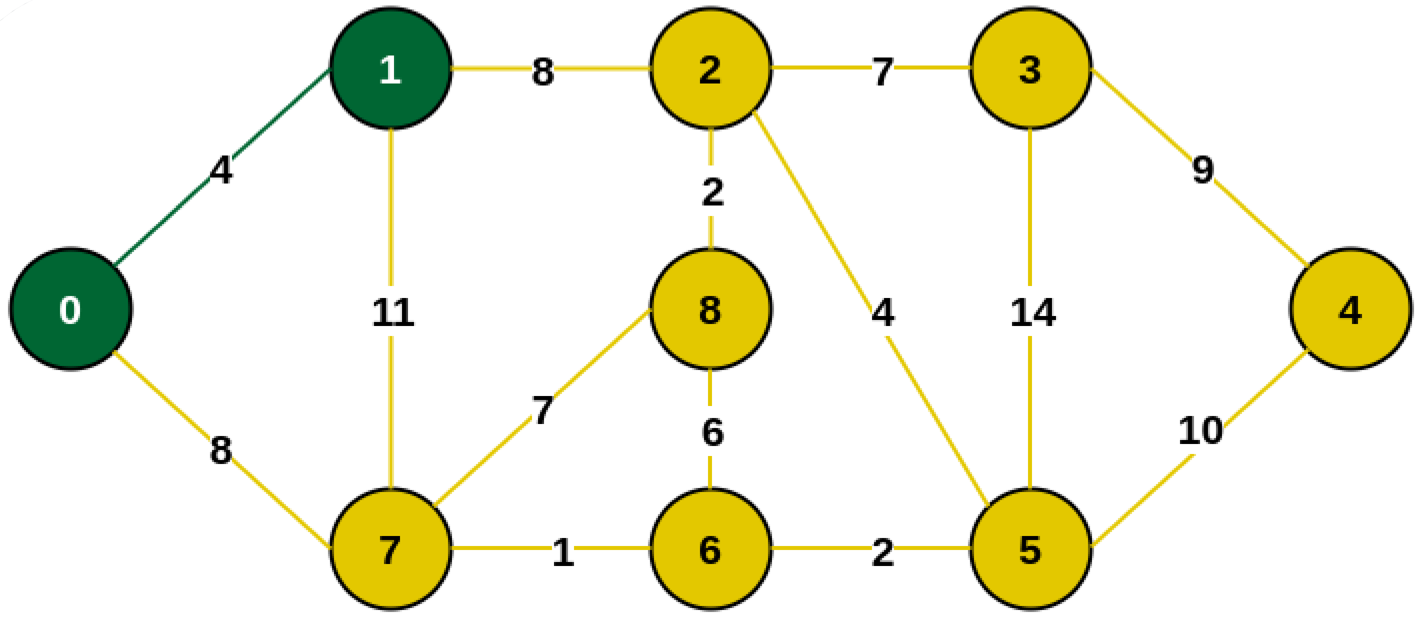
Example of a Graph

Step 1: *Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex* ***0*** *as the starting vertex.*



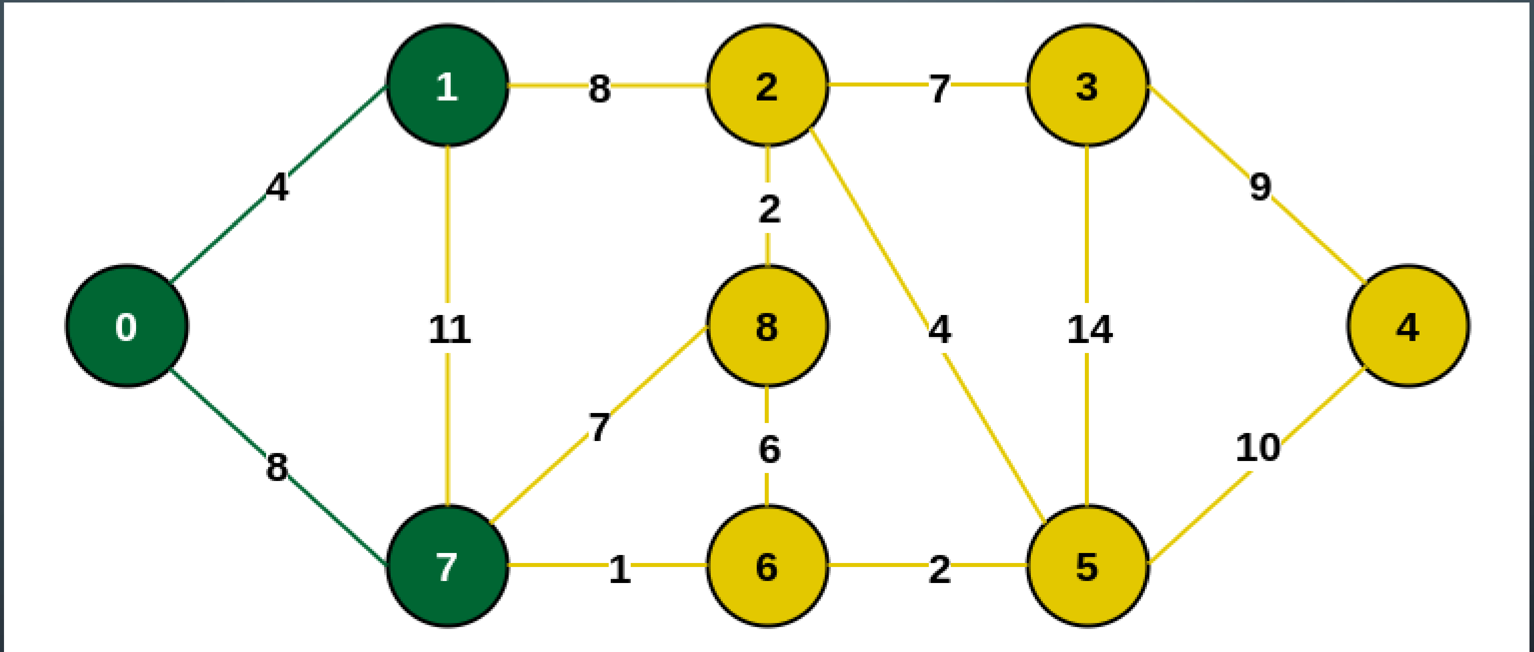
Select an arbitrary starting vertex. Here we have selected 0

<u>Step 2:</u> *All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.*



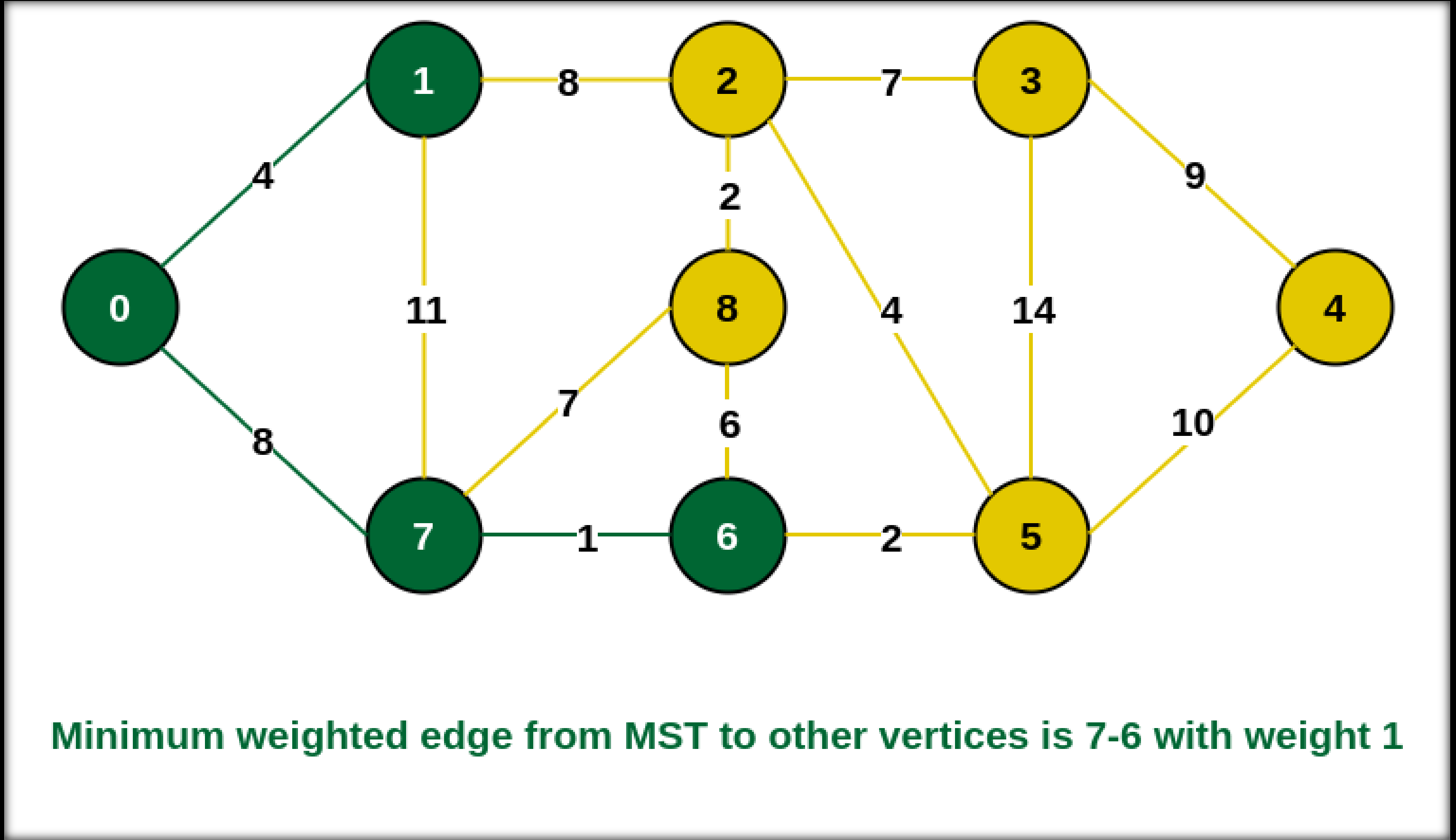
Minimum weighted edge from MST to other vertices is 0-1 with weight 4

<u>Step 3:</u> The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].



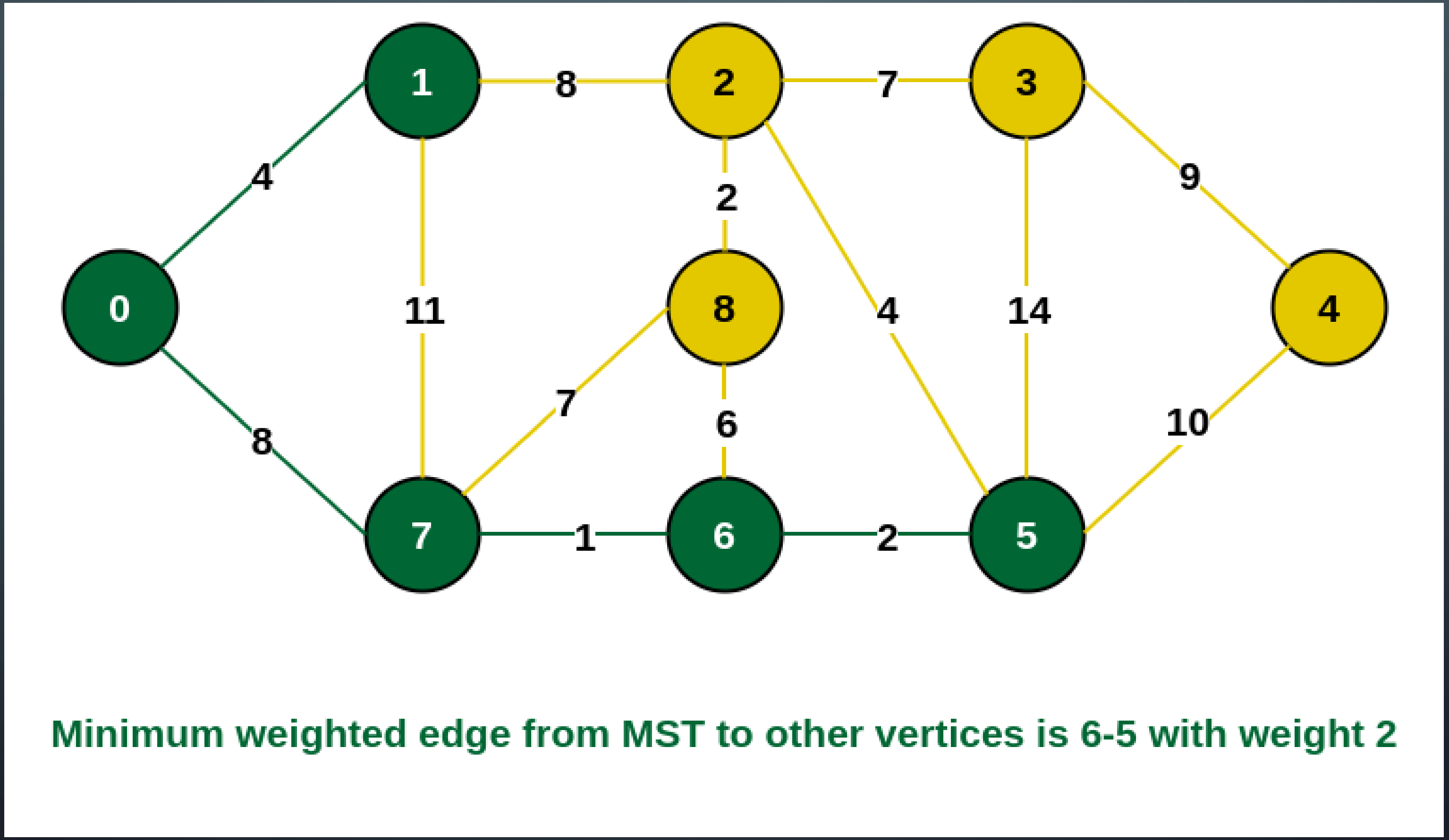
Minimum weighted edge from MST to other vertices is 0-7 with weight 8

Step 4: *The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).*

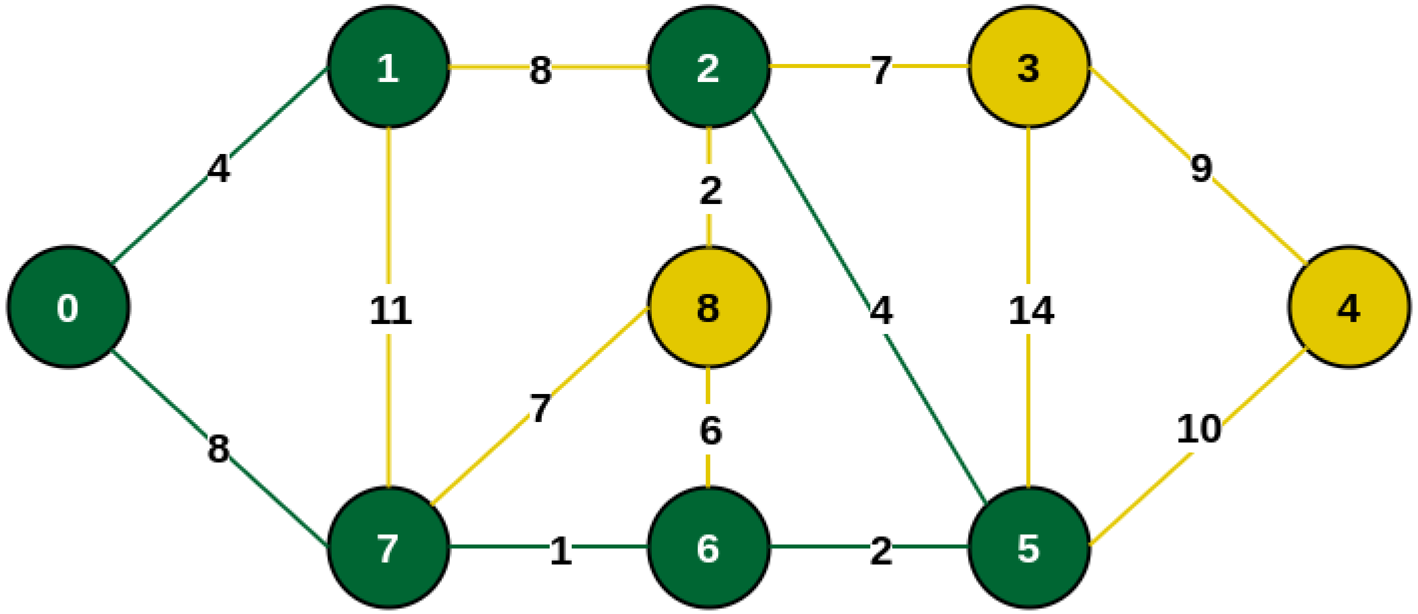


Minimum weighted edge from MST to other vertices is 7-6 with weight 1

Step 5: *The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.*

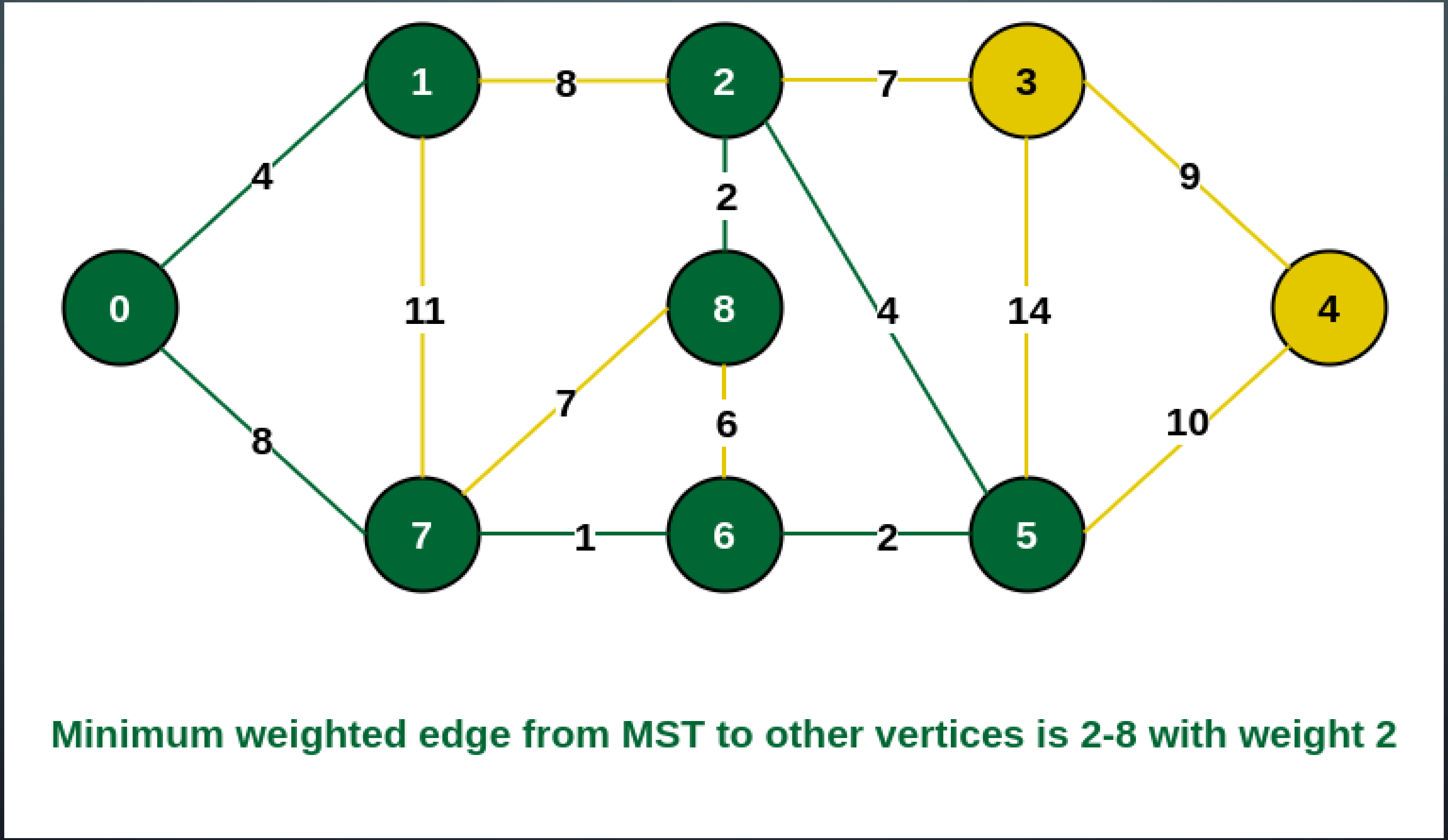

0
1
2
3
4
5
6
7
8
4
8
11
8
7
2
7
6
1
2
4
14
9
10
Minimum weighted edge from MST to other vertices is 6-5 with weight 2

Step 6: *Among the current connecting edges, the edge {5, 2} has the minimum weight. So include that edge and the vertex 2 in the MST.*



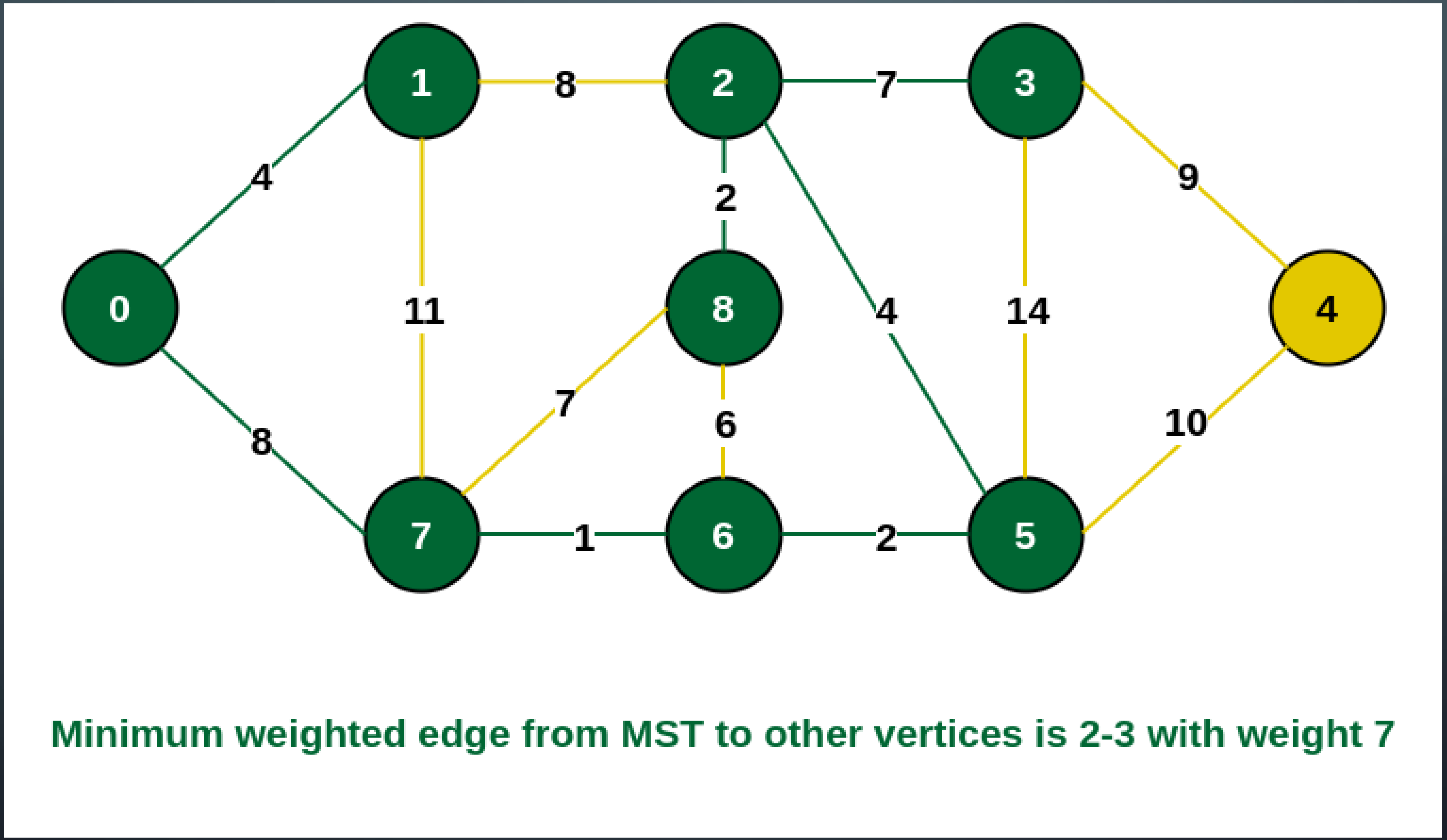
Minimum weighted edge from MST to other vertices is 5-2 with weight 4

Step 7: *The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.*



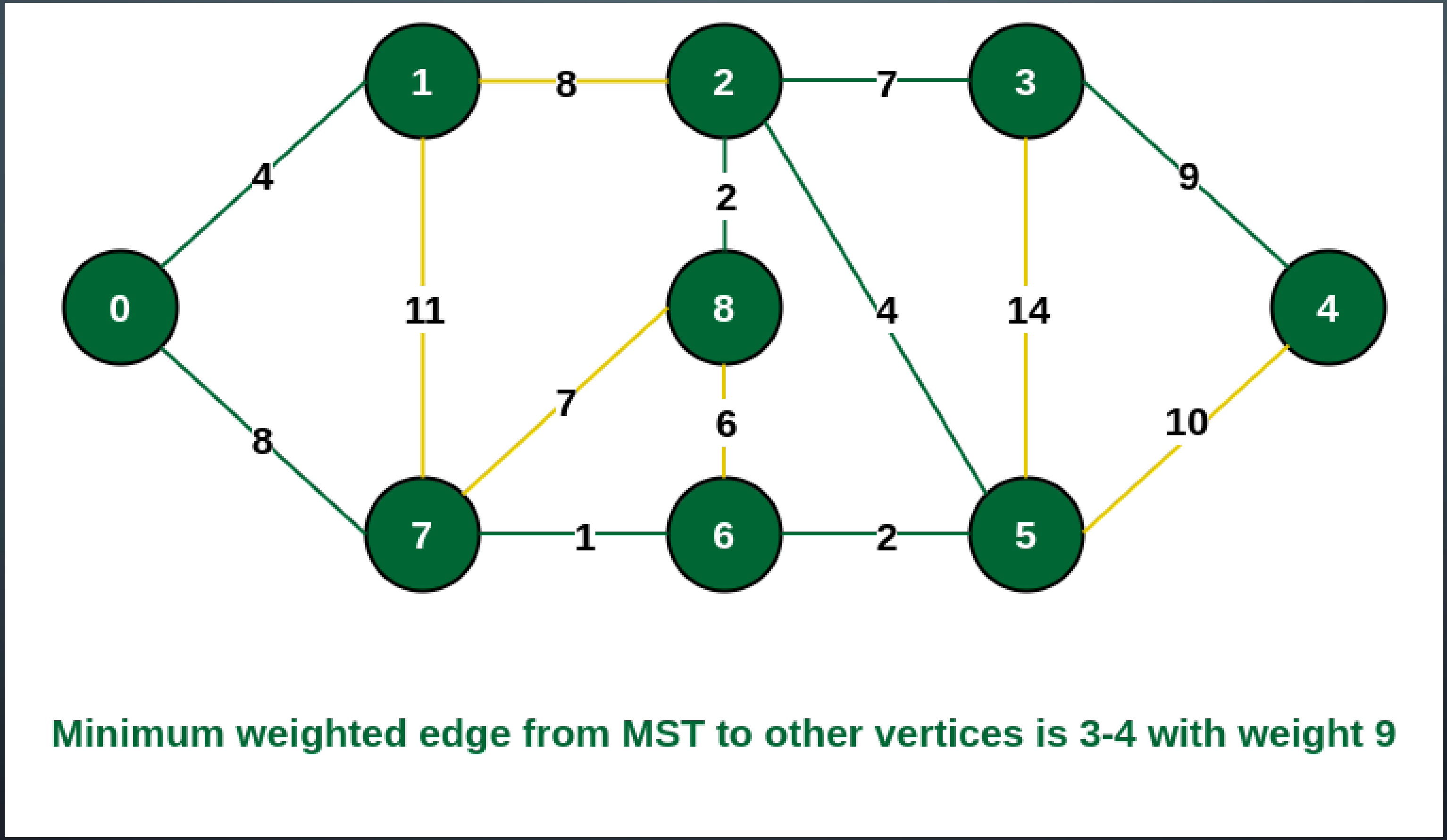
Minimum weighted edge from MST to other vertices is 2-8 with weight 2

Step 8: *See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.*



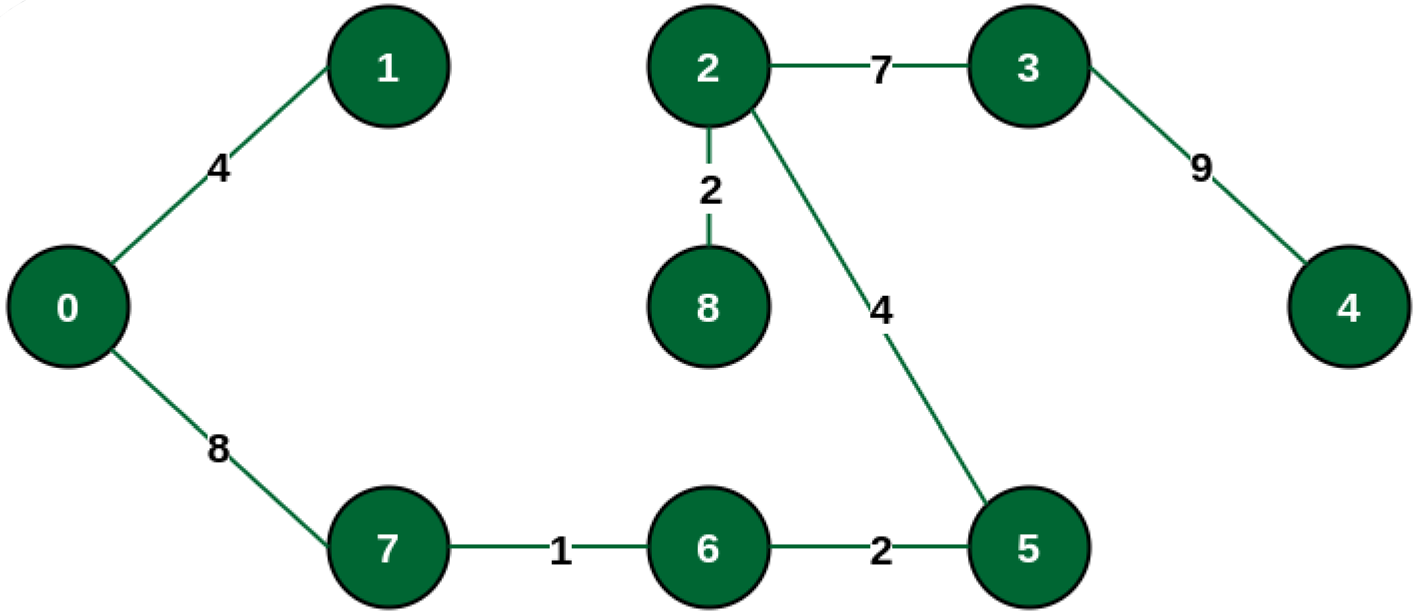
0
1
2
3
4
5
6
7
8
4
8
7
2
11
9
4
14
7
6
10
8
1
2
Minimum weighted edge from MST to other vertices is 2-3 with weight 7

<u>Step 9:</u> *Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4 is {3, 4}.*



Minimum weighted edge from MST to other vertices is 3-4 with weight 9

The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = ***37****.*



The final structure of MST

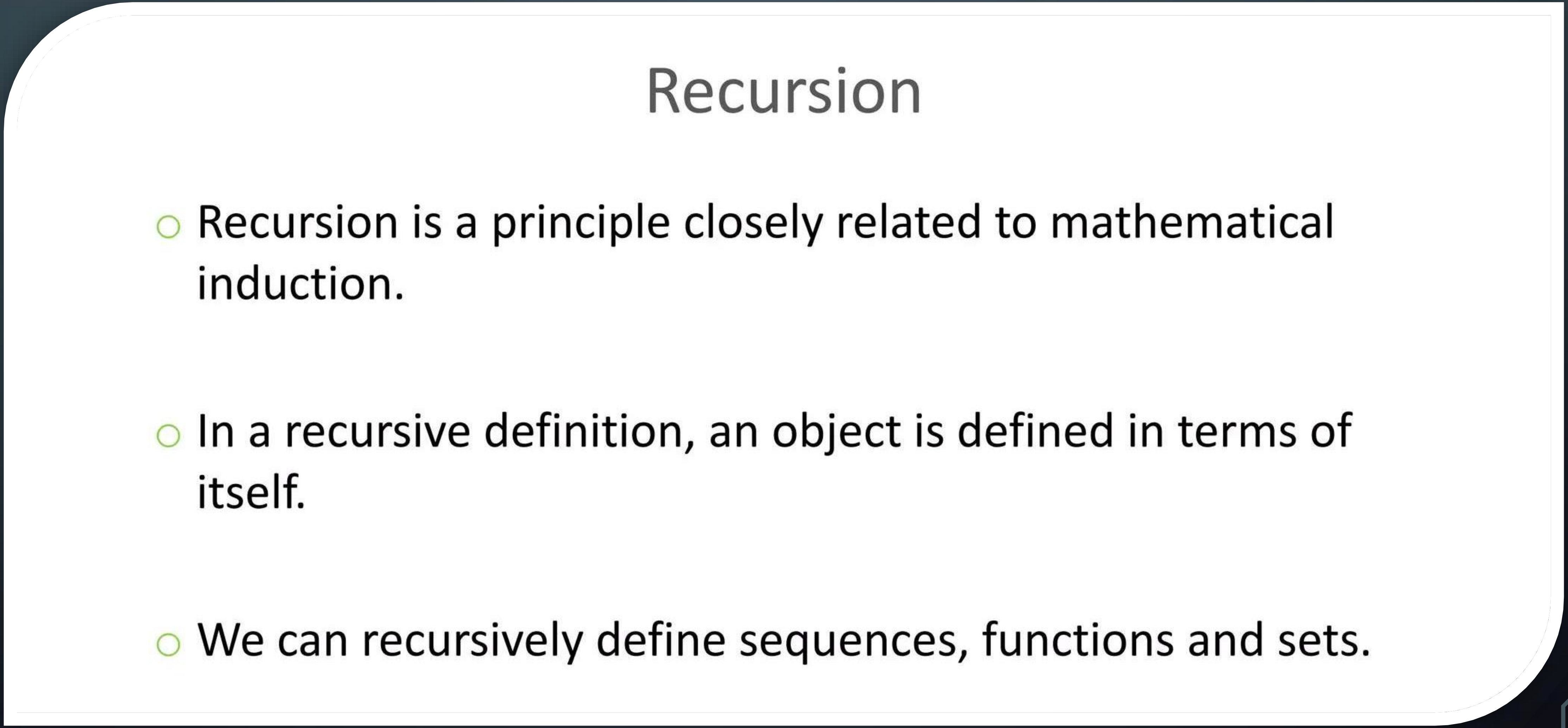
SELF STUDY

PRACTICE

Week 10

Recursion & DP

Page
132-156



Recursion

- Recursion is a principle closely related to mathematical induction.
- In a recursive definition, an object is defined in terms of itself.
- We can recursively define sequences, functions and sets.

Recursion

- RECURSION

  The process of defining an object in terms of smaller versions of itself is called recursion.

- A recursive definition has two parts:

  1. BASE:

  An initial simple definition which cannot be expressed in terms of smaller versions of itself.

  2. RECURSION:

  The part of definition which can be expressed in terms of smaller versions of itself.

Recursion

- BASE:

  1 is an odd positive integer.

- RECURSION:

  If k is an odd positive integer, then k + 2 is an odd positive integer.

  - Now, 1 is an odd positive integer by the definition base.
  - With k = 1, 1 + 2 = 3, so 3 is an odd positive integer.
  - With k = 3, 3 + 2 = 5, so 5 is an odd positive integer
  - and so, 7, 9, 11, … are odd positive integers.

- The main idea is to “reduce” a problem into smaller problems.

Recursion

$$f(0) = 3$$

$$f(n + 1) = 2f(n) + 3$$

- Find $f(1), f(2), f(3)$ and $f(4)$

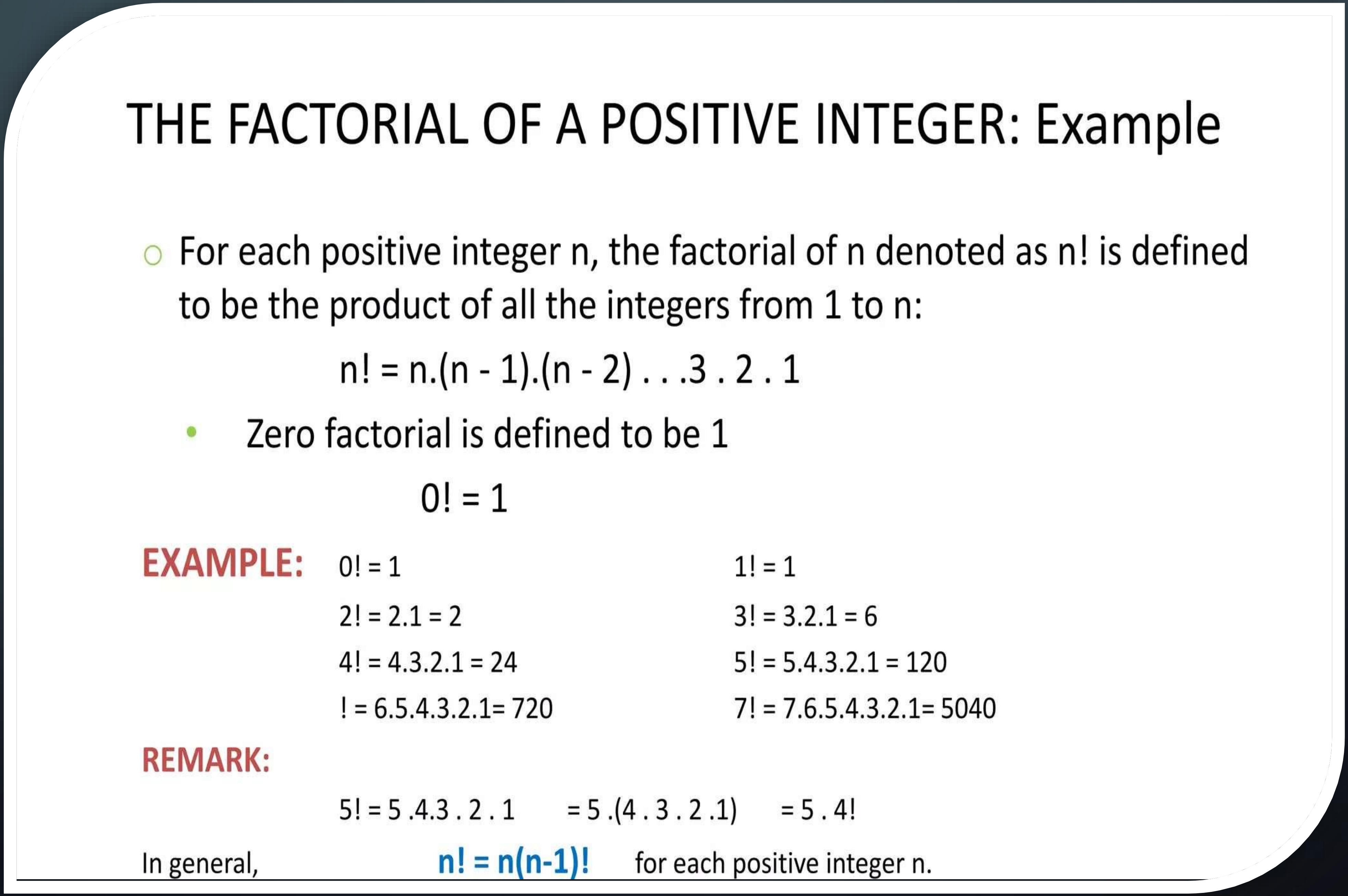
➢ From the recursive definition it follows that

$$f(1) = 2\ f(0) + 3 = 2(3) + 3 = 6 + 3 = 9$$

$$f(2) = 2\ f(1) + 3 = 2(9) + 3 = 18 + 3 = 21$$

$$f(3) = 2\ f(2) + 3 = 2(21) + 3 = 42 + 3 = 45$$

$$f(4) = 2\ f(3) + 3 = 2(45) + 3 = 90 + 3 = 93$$



THE FACTORIAL OF A POSITIVE INTEGER: Example

- For each positive integer n, the factorial of n denoted as n! is defined to be the product of all the integers from 1 to n:

$$n! = n.(n - 1).(n - 2) \ldots .3 . 2 . 1$$

- Zero factorial is defined to be 1

$$0! = 1$$

EXAMPLE:

0! = 1   1! = 1

2! = 2.1 = 2   3! = 3.2.1 = 6

4! = 4.3.2.1 = 24   5! = 5.4.3.2.1 = 120

! = 6.5.4.3.2.1= 720   7! = 7.6.5.4.3.2.1= 5040

REMARK:

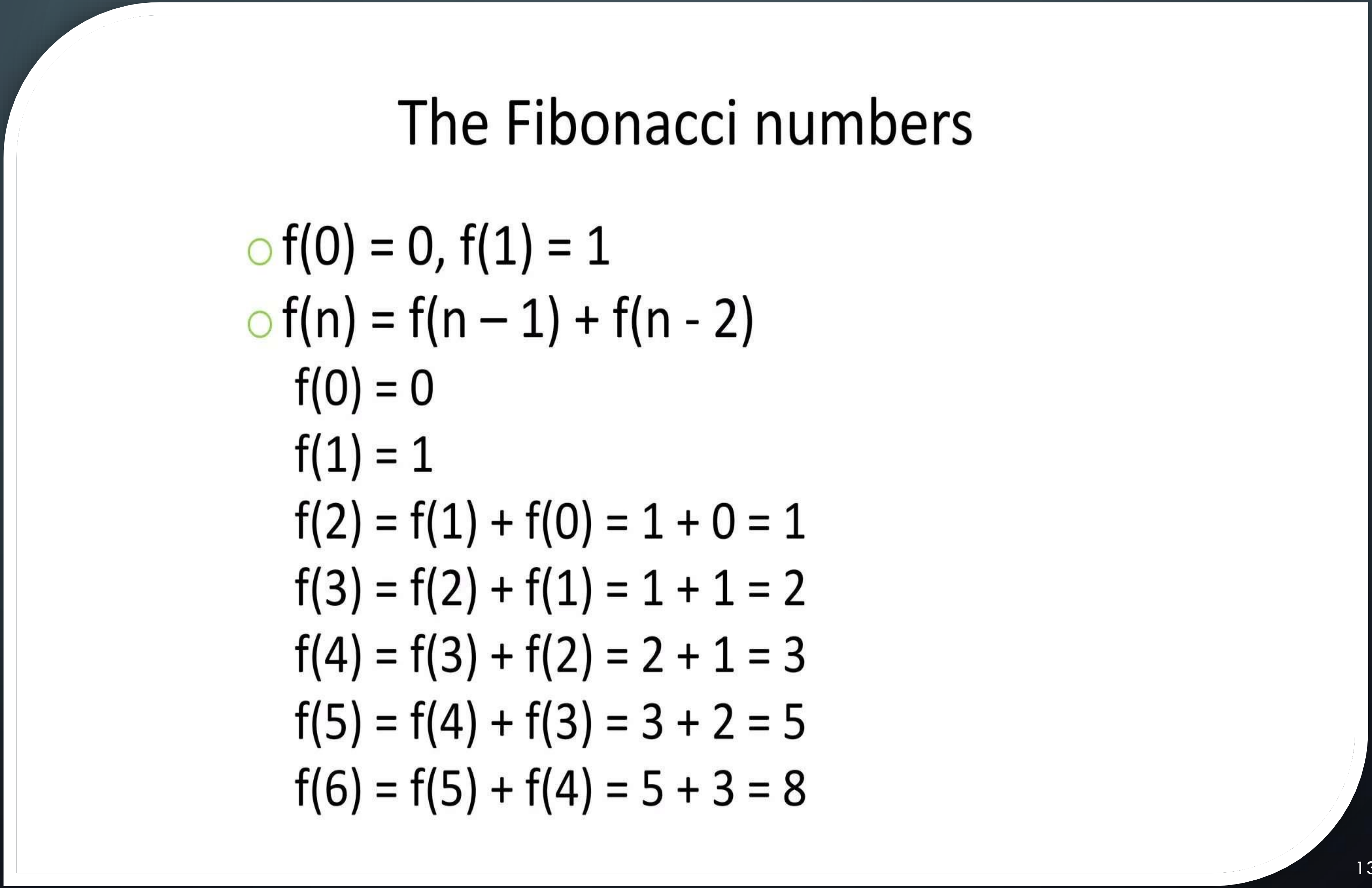
5! = 5 .4.3 . 2 . 1 = 5 .(4 . 3 . 2 .1) = 5 . 4!

In general, **n! = n(n-1)!** for each positive integer n.

THE FACTORIAL OF A POSITIVE INTEGER: Example

- Thus, the recursive definition of factorial function F(n) is:

1. $F(0) = 1$
2. $F(n) = n\,F(n-1)$



The Fibonacci numbers

- $f(0) = 0, f(1) = 1$
- $f(n) = f(n – 1) + f(n - 2)$

  $f(0) = 0$

  $f(1) = 1$

  $f(2) = f(1) + f(0) = 1 + 0 = 1$

  $f(3) = f(2) + f(1) = 1 + 1 = 2$

  $f(4) = f(3) + f(2) = 2 + 1 = 3$

  $f(5) = f(4) + f(3) = 3 + 2 = 5$

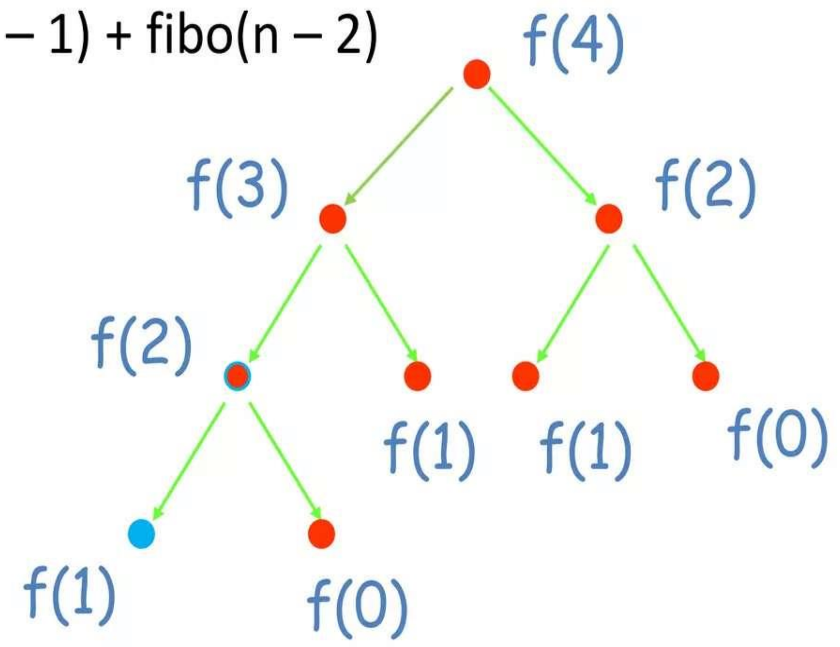
  $f(6) = f(5) + f(4) = 5 + 3 = 8$

RECURSIVELY DEFINED FUNCTIONS

- A function is said to be recursively defined if the function refers to itself such that
  1. There are certain arguments, called base values, for which the function does not refer to itself.
  2. Each time the function does refer to itself, the argument of the function must be closer to a base value.

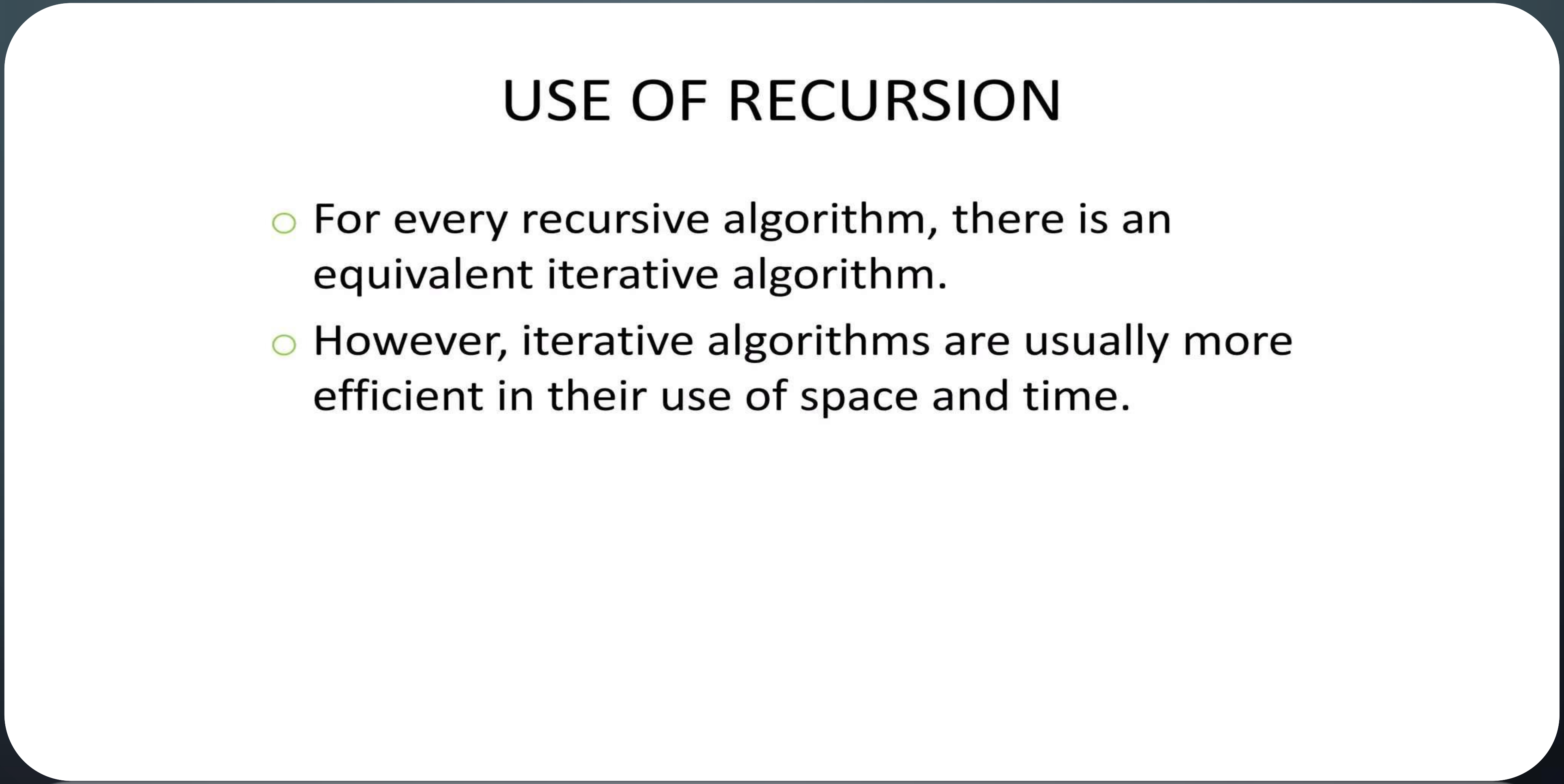
RECURSIVELY DEFINED FUNCTIONS

```
procedure fibo(n: nonnegative integer)
    if n = 0 then fibo(0) := 0
    else if n = 1 then fibo(1) := 1
    else fibo(n) := fibo(n – 1) + fibo(n – 2)
end
```



USE OF RECURSION

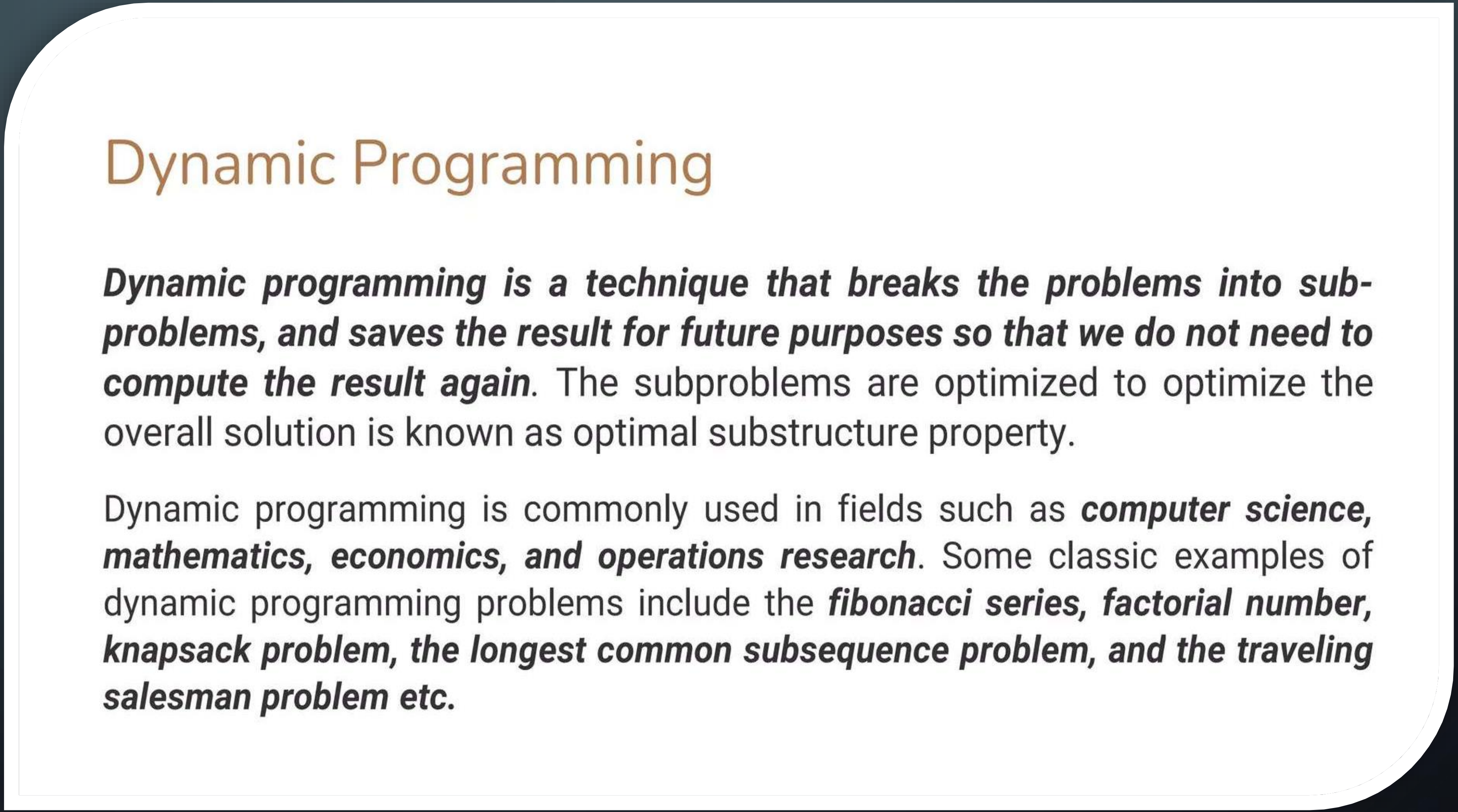
- At first recursion may seem hard or impossible, may be magical at best. However, recursion often provides elegant, short algorithmic solutions to many problems in computer science and mathematics.
  - Examples where recursion is often used
    - math functions
    - number sequences
    - data structure definitions
    - data structure manipulations
    - language definitions



USE OF RECURSION

- For every recursive algorithm, there is an equivalent iterative algorithm.
- However, iterative algorithms are usually more efficient in their use of space and time.

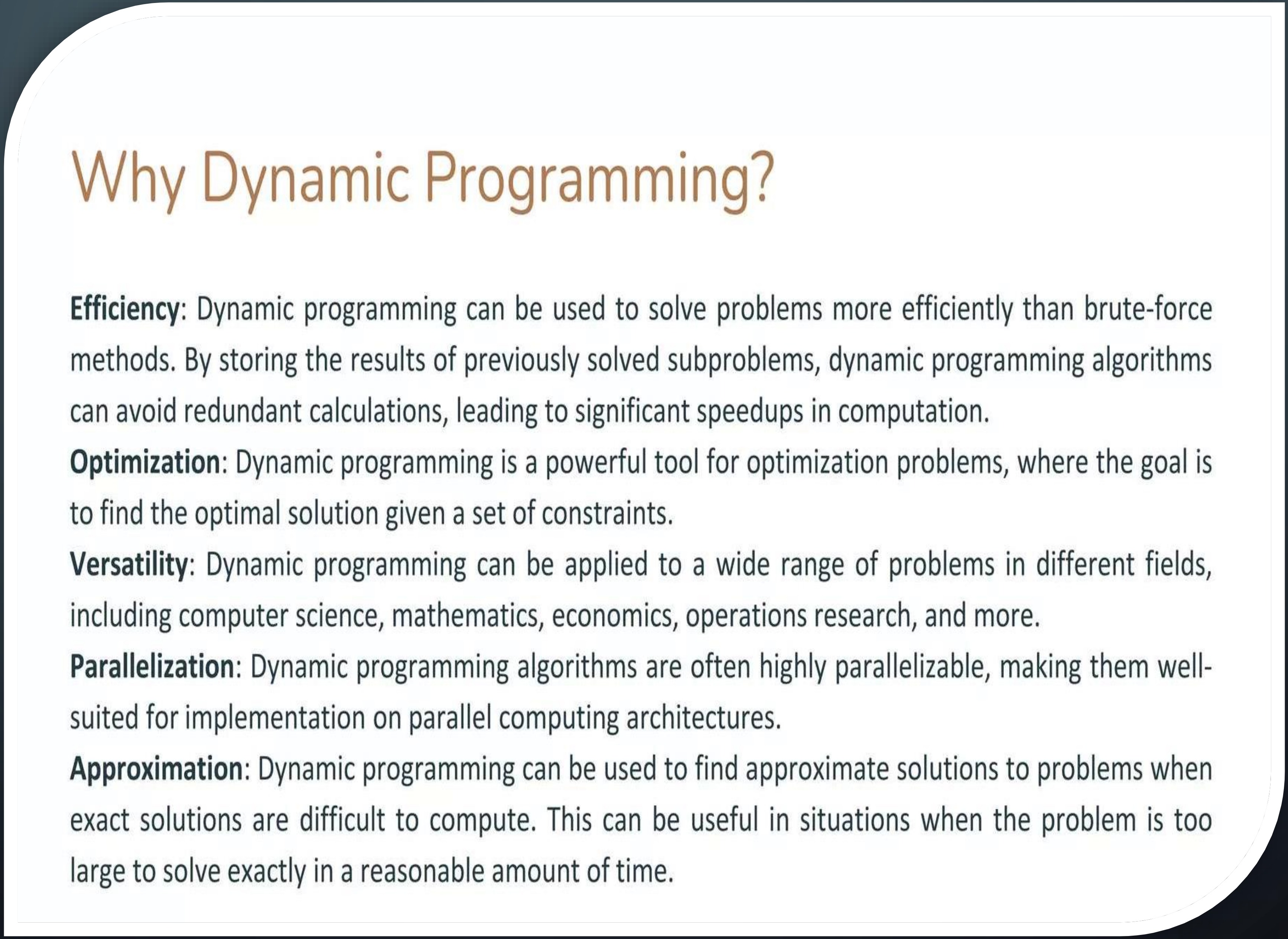
DP



Dynamic Programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property.

Dynamic programming is commonly used in fields such as ***computer science, mathematics, economics, and operations research***. Some classic examples of dynamic programming problems include the ***fibonacci series, factorial number, knapsack problem, the longest common subsequence problem, and the traveling salesman problem etc.***



Why Dynamic Programming?

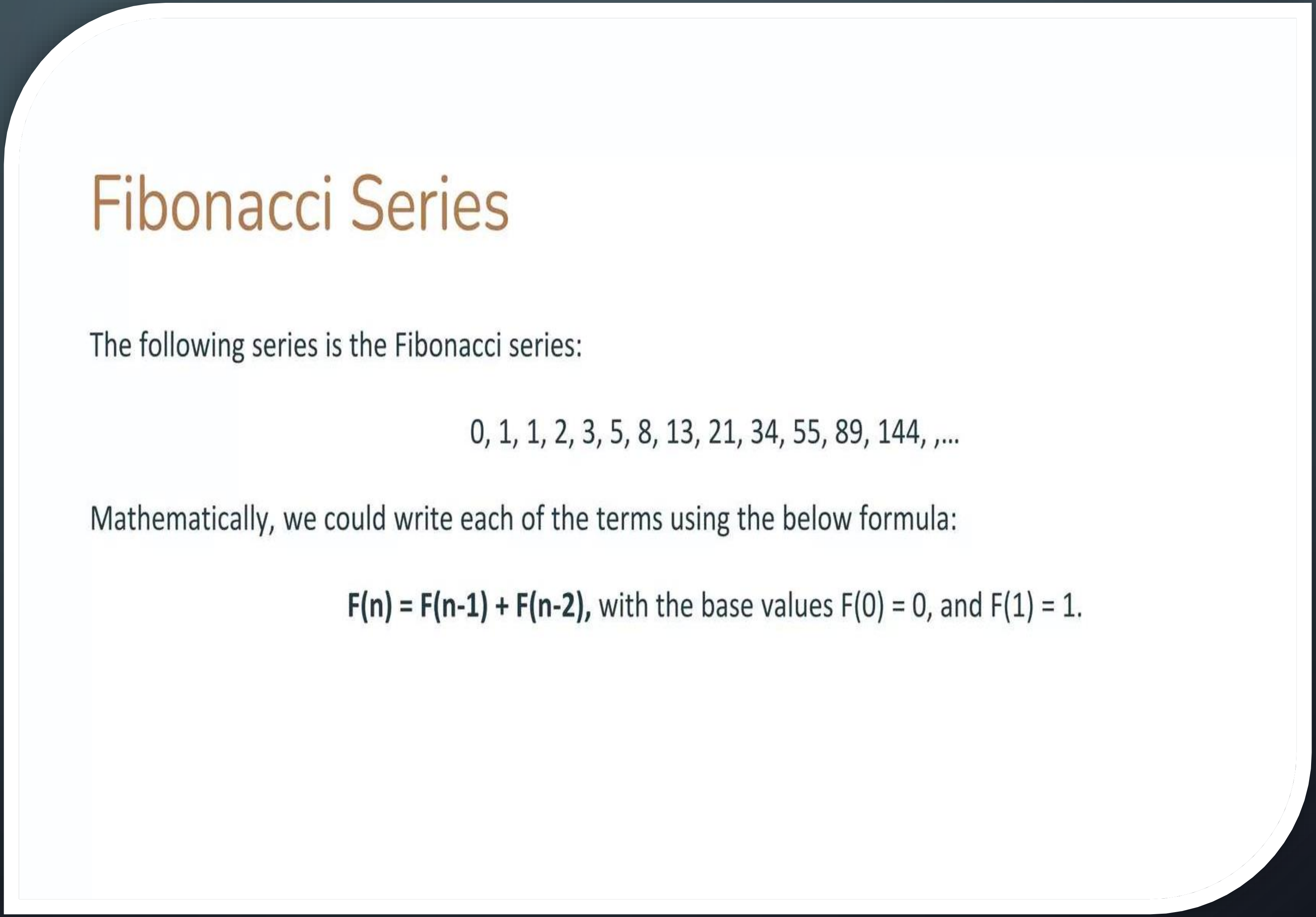
Efficiency: Dynamic programming can be used to solve problems more efficiently than brute-force methods. By storing the results of previously solved subproblems, dynamic programming algorithms can avoid redundant calculations, leading to significant speedups in computation.

Optimization: Dynamic programming is a powerful tool for optimization problems, where the goal is to find the optimal solution given a set of constraints.

Versatility: Dynamic programming can be applied to a wide range of problems in different fields, including computer science, mathematics, economics, operations research, and more.

Parallelization: Dynamic programming algorithms are often highly parallelizable, making them well-suited for implementation on parallel computing architectures.

Approximation: Dynamic programming can be used to find approximate solutions to problems when exact solutions are difficult to compute. This can be useful in situations when the problem is too large to solve exactly in a reasonable amount of time.



Fibonacci Series

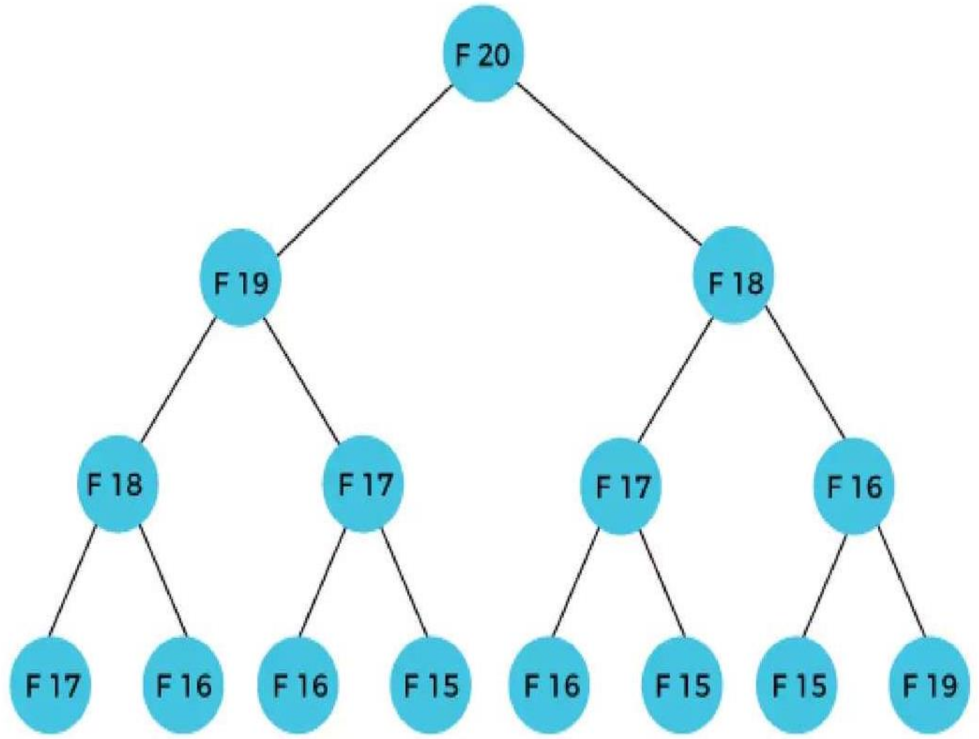
The following series is the Fibonacci series:

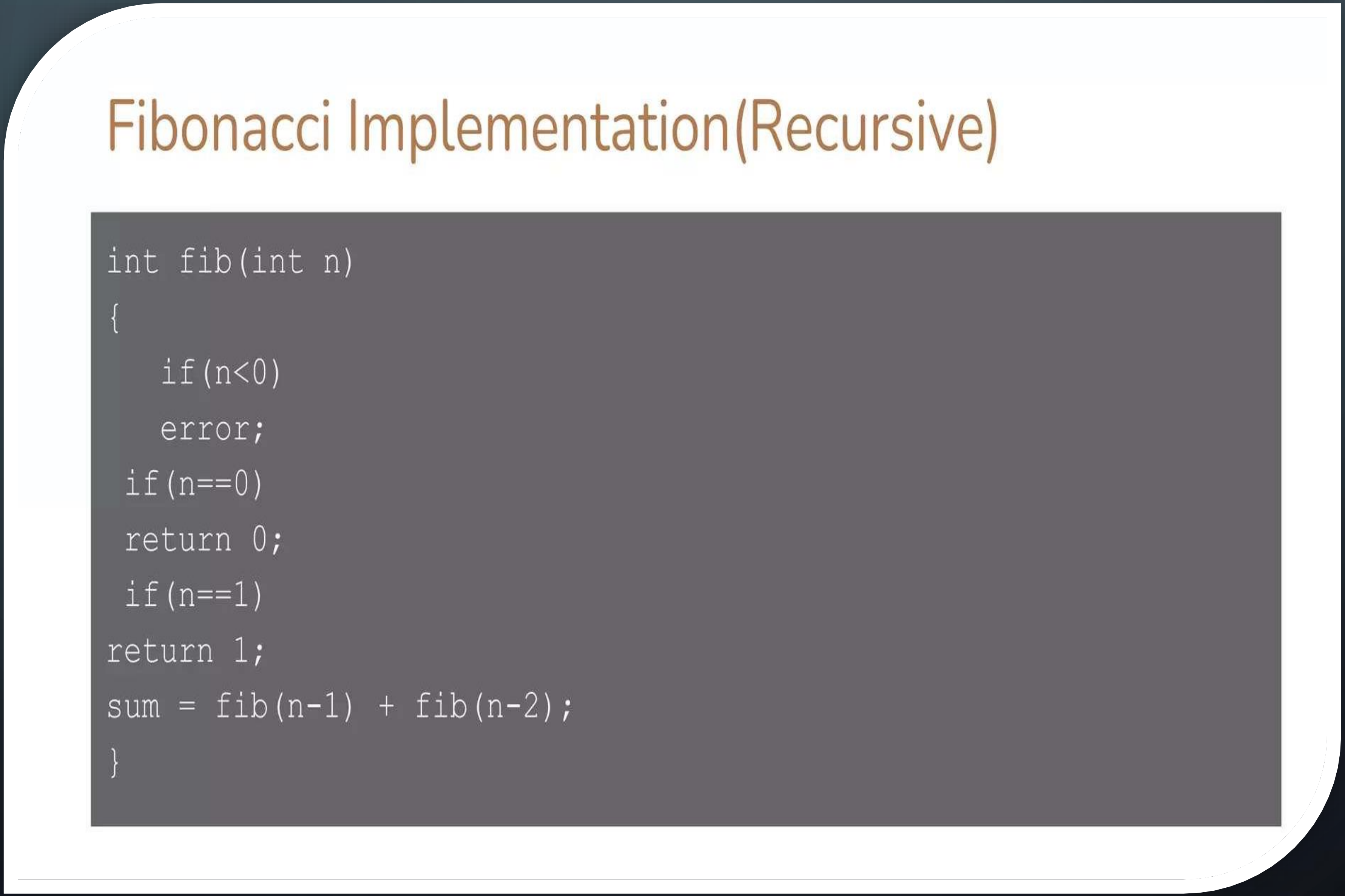
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,...

Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2), with the base values F(0) = 0, and F(1) = 1.

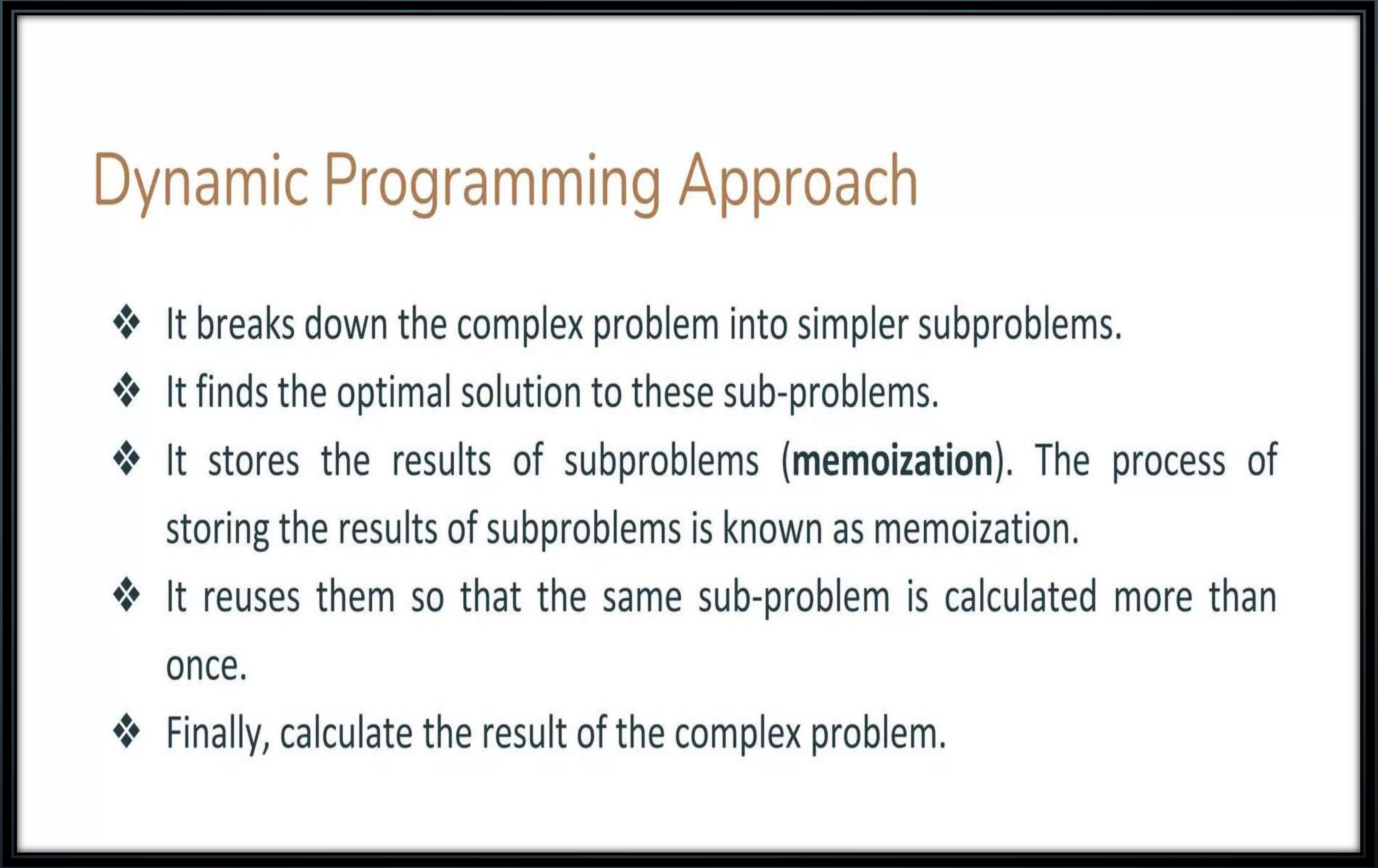
How can we calculate F(20)?





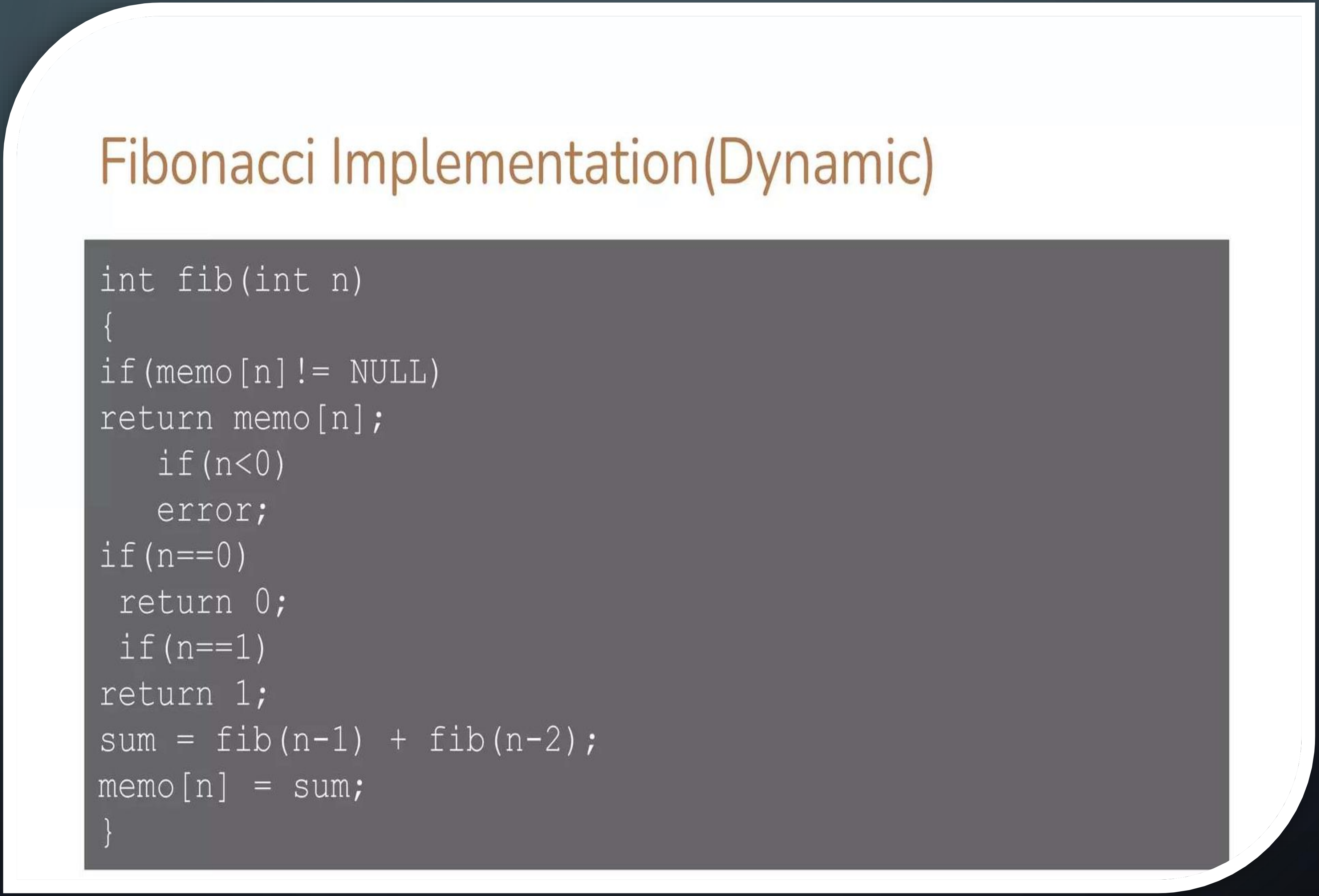
Fibonacci Implementation(Recursive)

```
int fib(int n)
{
   if(n<0)
   error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
}
```



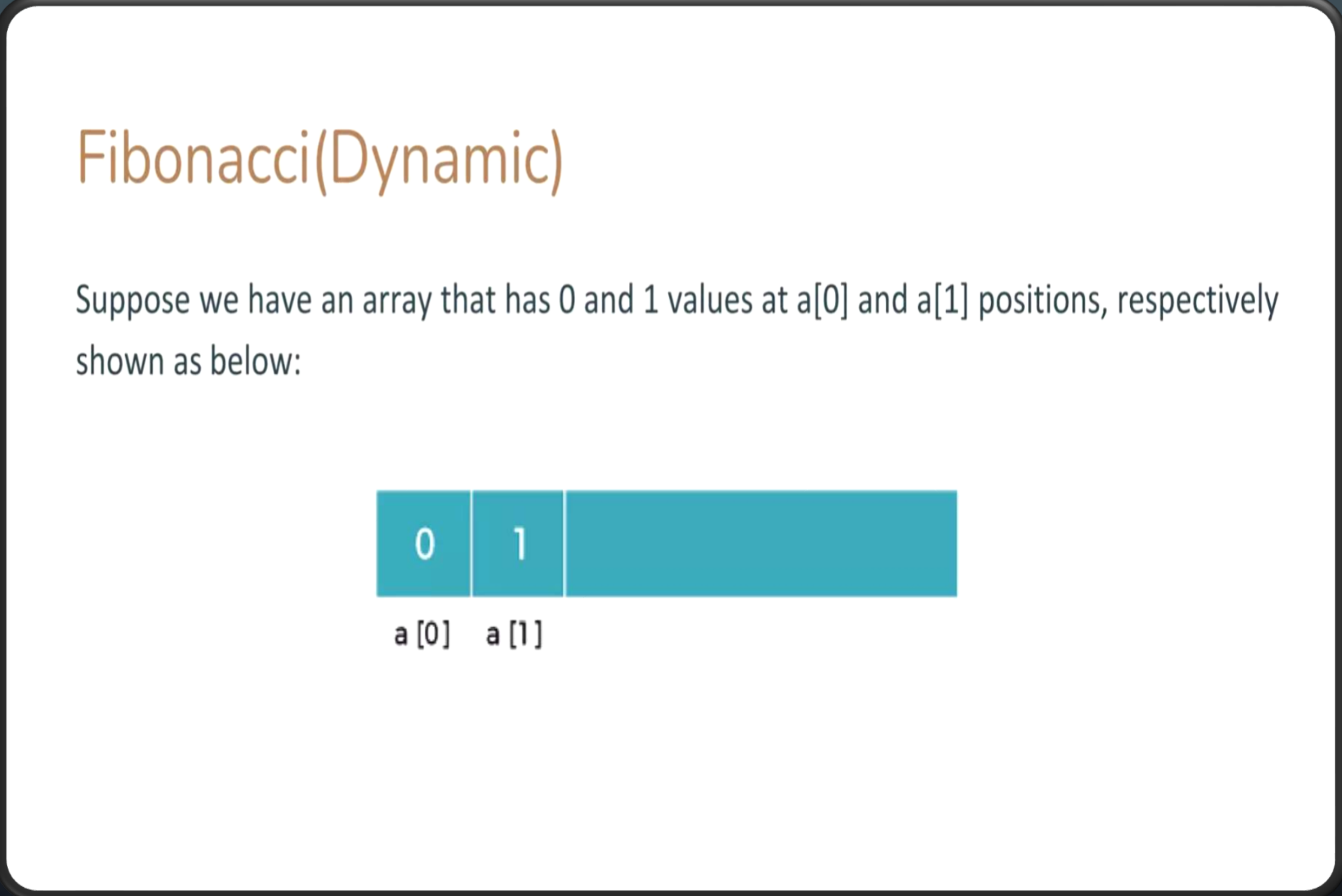
Dynamic Programming Approach

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (**memoization**). The process of storing the results of subproblems is known as memoization.
- It reuses them so that the same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.



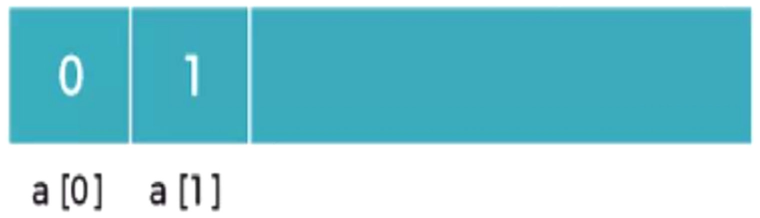
Fibonacci Implementation(Dynamic)

```
int fib(int n)
{
if(memo[n]!= NULL)
return memo[n];
   if(n<0)
   error;
if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
memo[n] = sum;
}
```



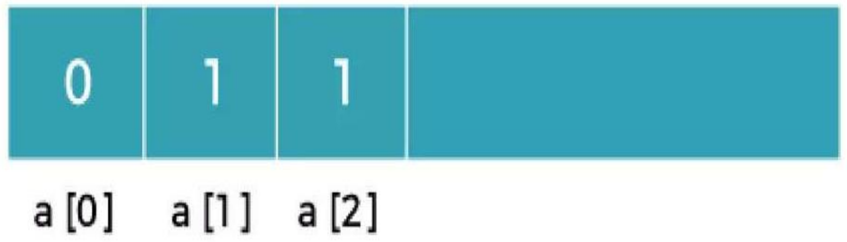
Fibonacci(Dynamic)

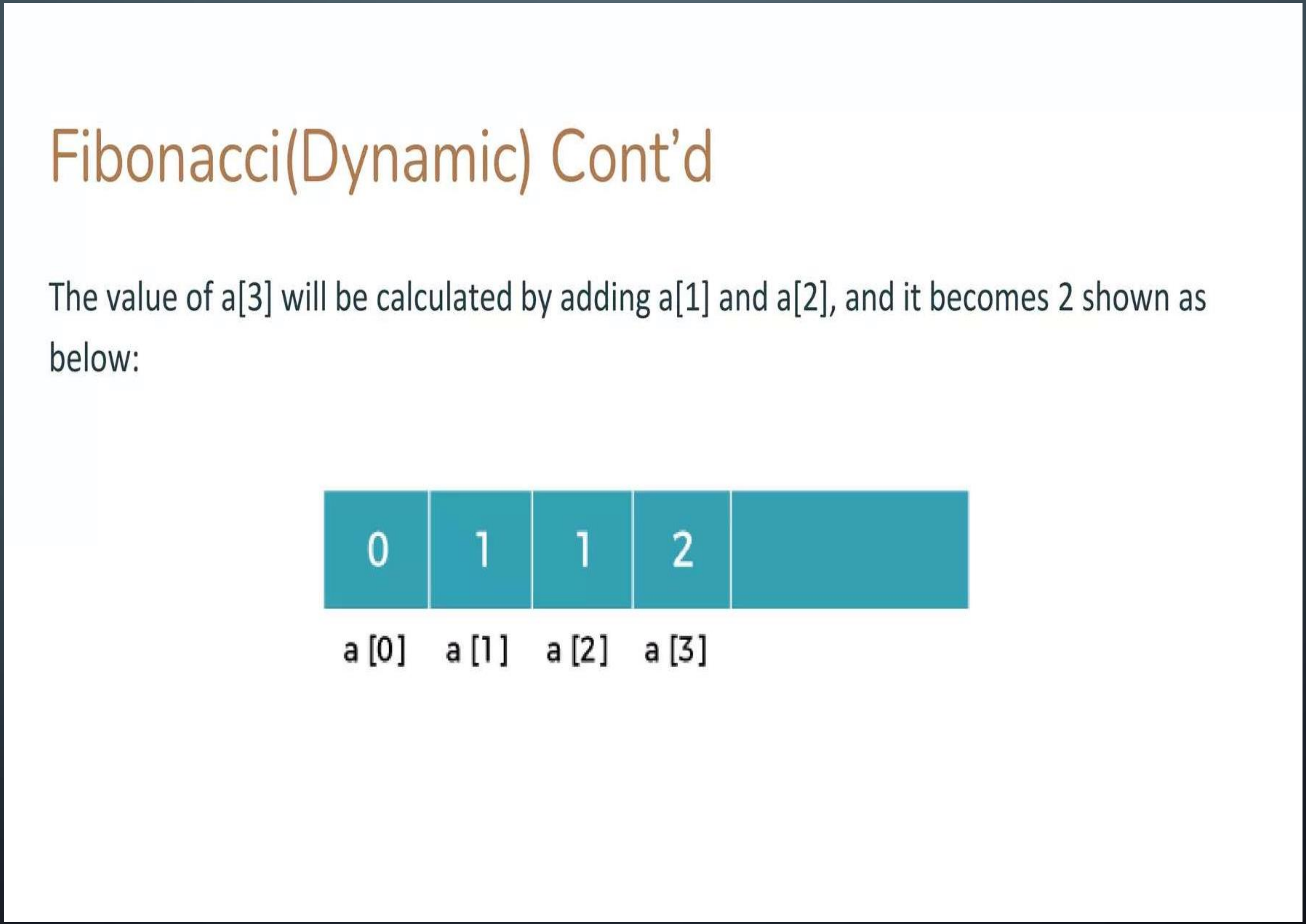
Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:



Fibonacci(Dynamic) Cont'd

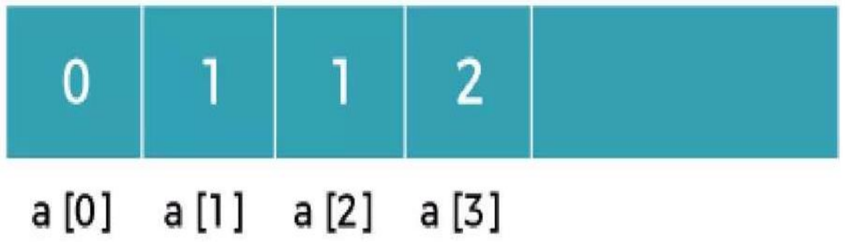
Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value of a[2] shown as below:





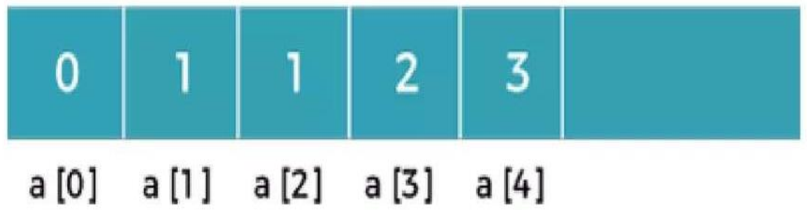
Fibonacci(Dynamic) Cont'd

The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:



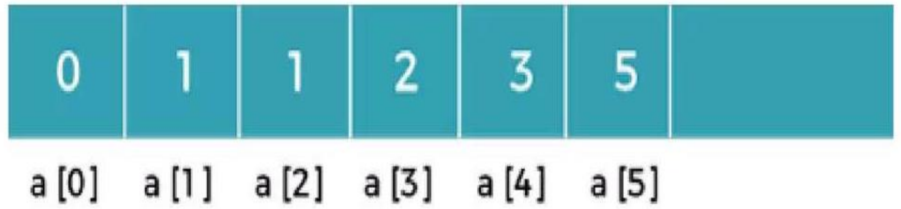
Fibonacci(Dynamic) Cont’d

The value of a[4] will be calculated by adding a[2] and a[3], and it becomes 3 shown as below:



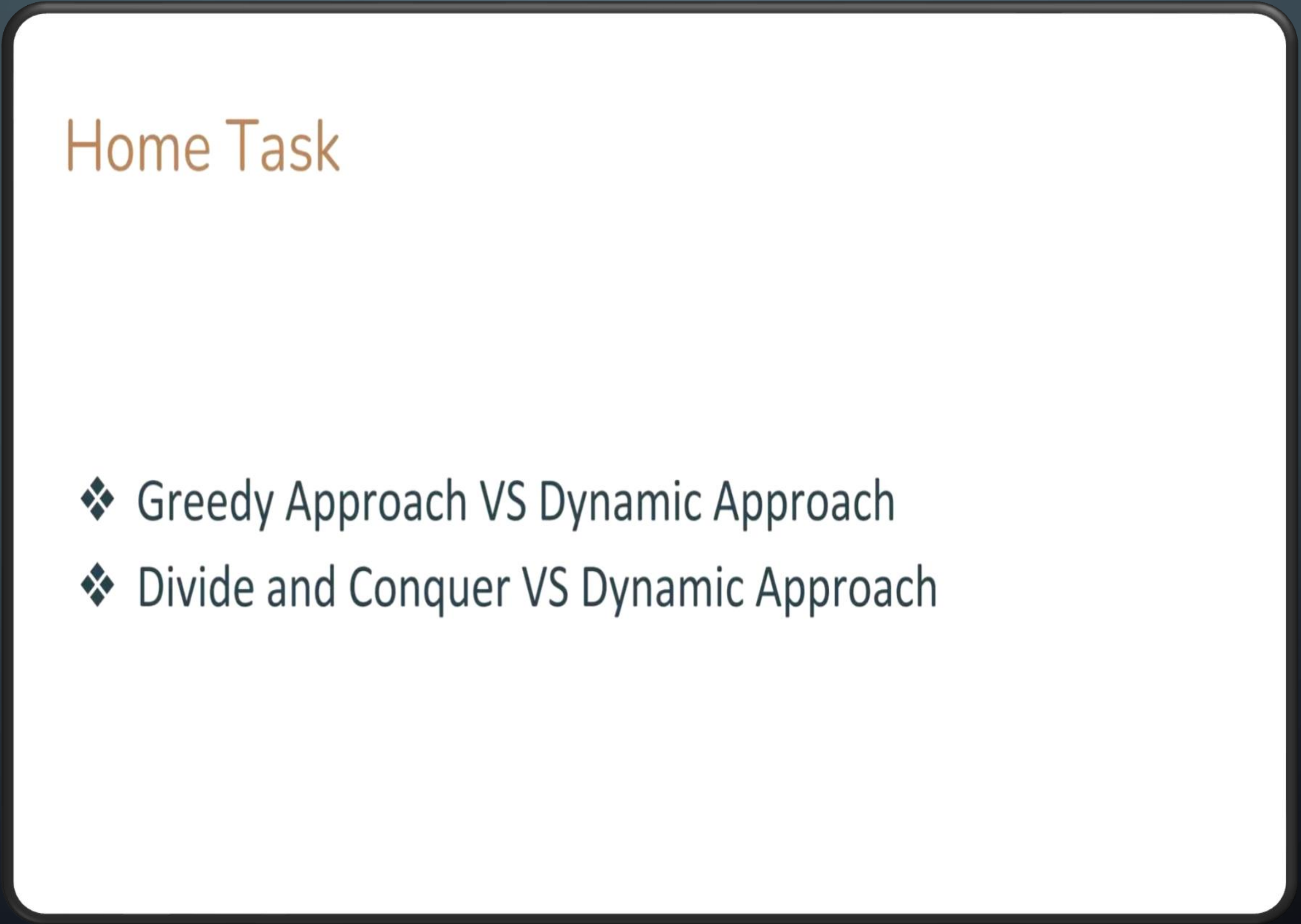
Fibonacci(Dynamic) Cont’d

The value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:



Statistics

	Recursive	Dynamic
N=20	21,891	39
N=40	331,160,281	79



Home Task

- ❖ Greedy Approach VS Dynamic Approach
- ❖ Divide and Conquer VS Dynamic Approach

WEEK 11

BELLMAN FORD ALGORITHM

Page
157-174

BELLMAN–FORD ALGORITHM

- Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" (a cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence.

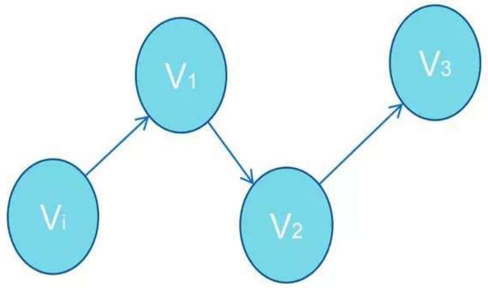
SHORTEST PATH PROBLEM

- ❖Shortest path network
  - Driected graph
  - Source s, Destination t
  - Cost(V-U) cost of using edge from v to u

Shortest path problem

Find shortest directed path from s to t

Cost of path = sum of arc cost in path

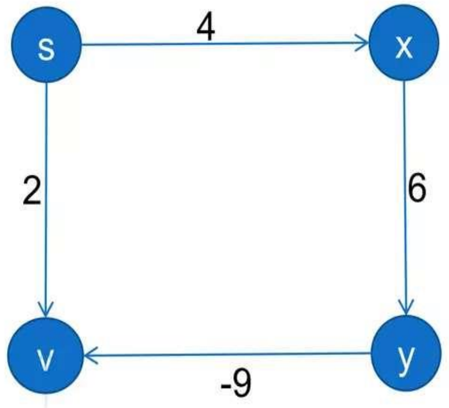


$$\delta(S, V_i) = \delta(S, V_{i-1}) + W(V_{i-1}, V_i)$$

BELLMAN-FORD

❖Dijkstra's Algorithm fails when theme is negative edge

Ex: Selects vertex v immediately after s . But short path s to v is



S-X-Y-V

Solution is Bellman Ford Algorithm which can work on negative edge.

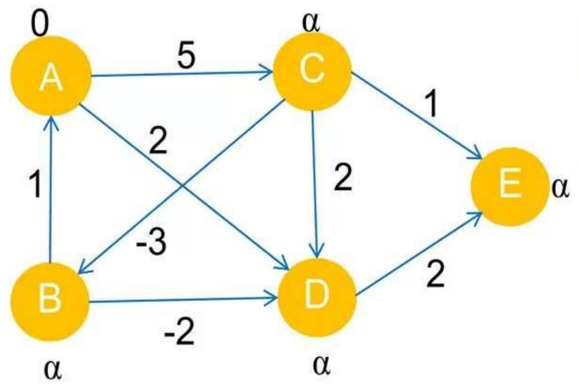
Solution is Bellman Ford Algorithm which can work on negative edge.

PSUEDOCODE

```
BELLMAN-FOR D ( G ,s )
   INITIALIZE   SINGLE – SOURCE   ( G , s )
    for   i ←  1 to |V|-1  do                                computaion
        for    each  edge  ( u, v ) €   G.E do
         RELAX  ( U ,  V )
      for    each  edge  ( u, v ) €  E do
         if d[V] > d[U] + W( U , V ) then                    check
         return   FALSE
    return    TRUE
```

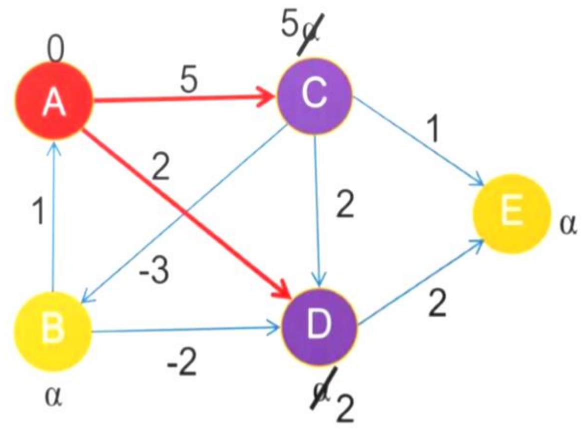
BELLMAN–FORD EXAMPLE

1 st :



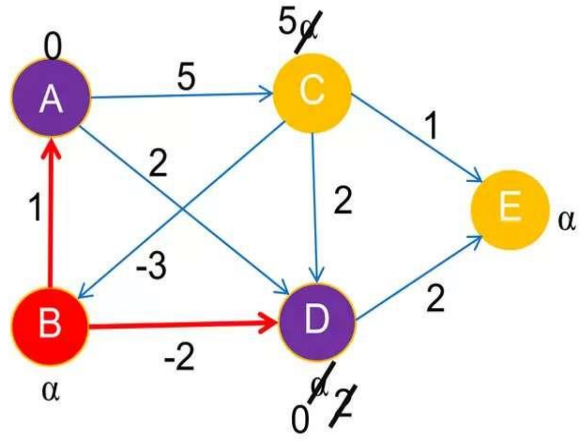
Vertices	A	B	C	D	E
Cost	0	α	α	α	α
Prevertices	—	—	—	—	—

BELLMAN–FORD EXAMPLE



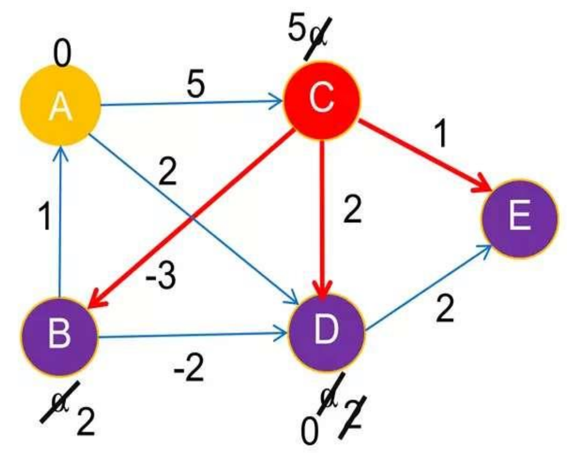
Vertices	A	B	C	D	E
Cost	0	α	5 ~~α~~	2 ~~α~~	α
Pre	–	–	A	A	–

BELLMAN–FORD EXAMPLE



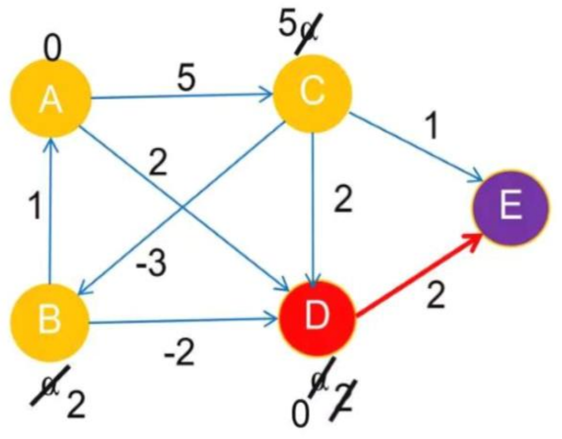
Vertices	A	B	C	D	E
Cost	0	α	5 ~~α~~	0 ~~2~~ ~~α~~	α
Pre	–	–	A	~~A~~ B	–

BELLMAN–FORD EXAMPLE



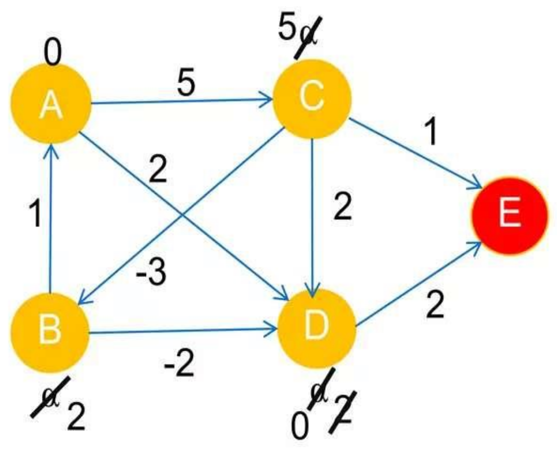
Vertices	A	B	C	D	E
Cost	0	~~α~~ 2	5 ~~α~~	0 ~~2~~ ~~α~~	~~α~~ 6
Pre	–	C	A	~~A~~ B	C

BELLMAN–FORD EXAMPLE



Vertices	A	B	C	D	E
Cost	0	~~α~~ 2	5 ~~α~~	0 ~~2~~ ~~α~~	~~α~~ ~~6~~ 2
Pre	–	C	A	~~A~~ B	~~C~~ D

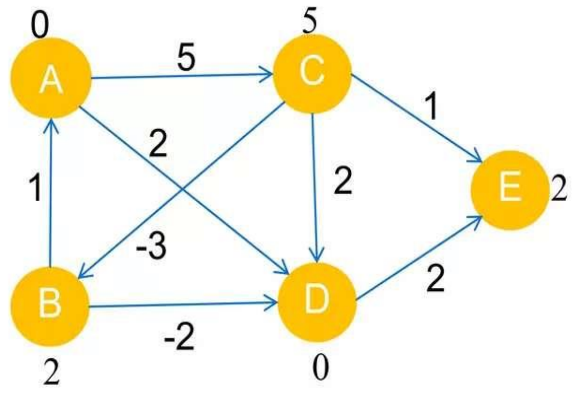
BELLMAN–FORD EXAMPLE



Vertices	A	B	C	D	E
Cost	0	α ~~2~~	5 ~~α~~	0 ~~2~~ ~~α~~	~~α~~ ~~6~~ 2
Pre	–	C	A	~~A~~ B	C ~~D~~

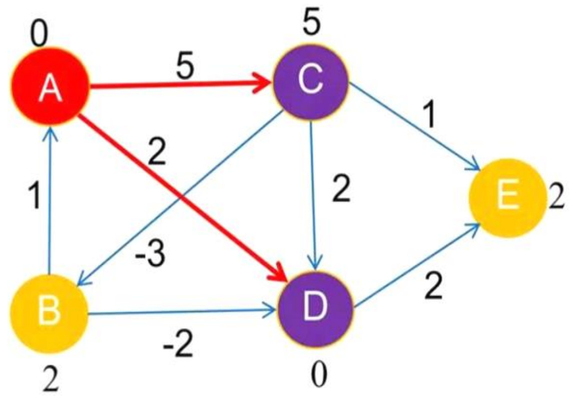
BELLMAN–FORD EXAMPLE

2nd:



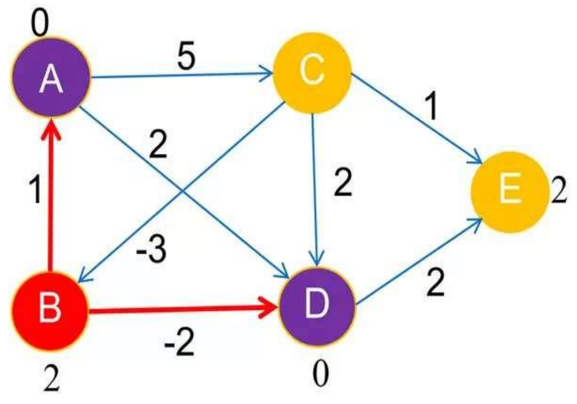
Vertices	A	B	C	D	E
Cost	0	2	5	0	2
Pre	–	C	A	B	D

BELLMAN–FORD EXAMPLE



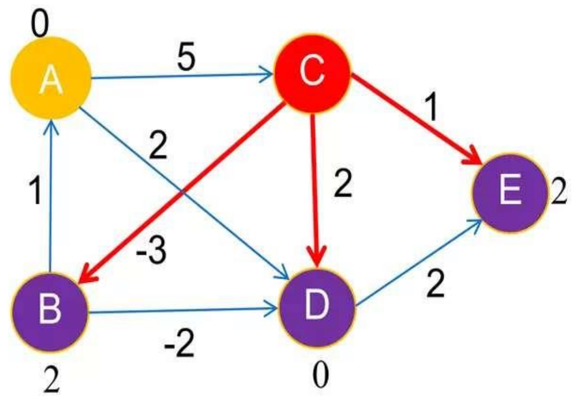
Vertices	A	B	C	D	E
Cost	0	2	5	0	2
Pre	–	C	A	B	D

BELLMAN–FORD EXAMPLE



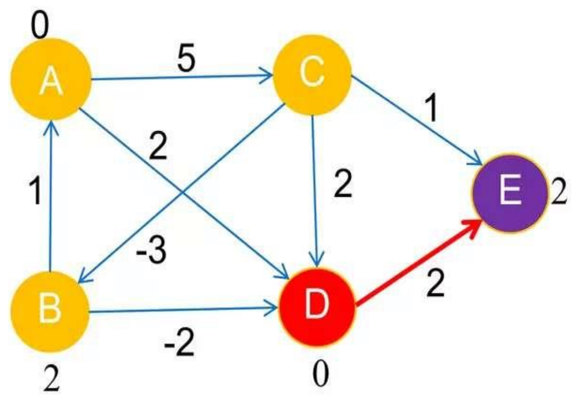
Vertices	A	B	C	D	E
Cost	0	2	5	0	2
Pre	–	C	A	B	D

BELLMAN–FORD EXAMPLE



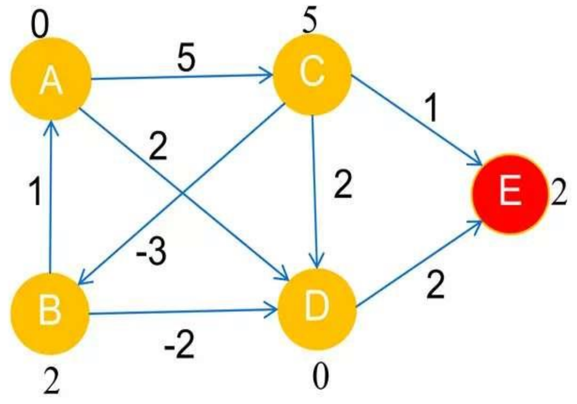
Vertices	A	B	C	D	E
Cost	0	2	5	0	2
Pre	–	C	A	B	D

BELLMAN–FORD EXAMPLE



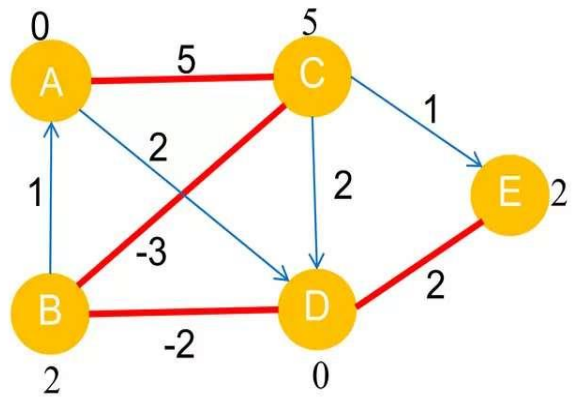
Vertices	A	B	C	D	E
Cost	0	2	5	0	2
Pre	–	C	A	B	D

BELLMAN–FORD EXAMPLE



Vertices	A	B	C	D	E
Cost	0	2	5	0	2
Pre	–	C	A	B	D

BELLMAN–FORD EXAMPLE



Vertices	A	B	C	D	E
Cost	0	2	5	0	2
Pre	–	C	A	B	D

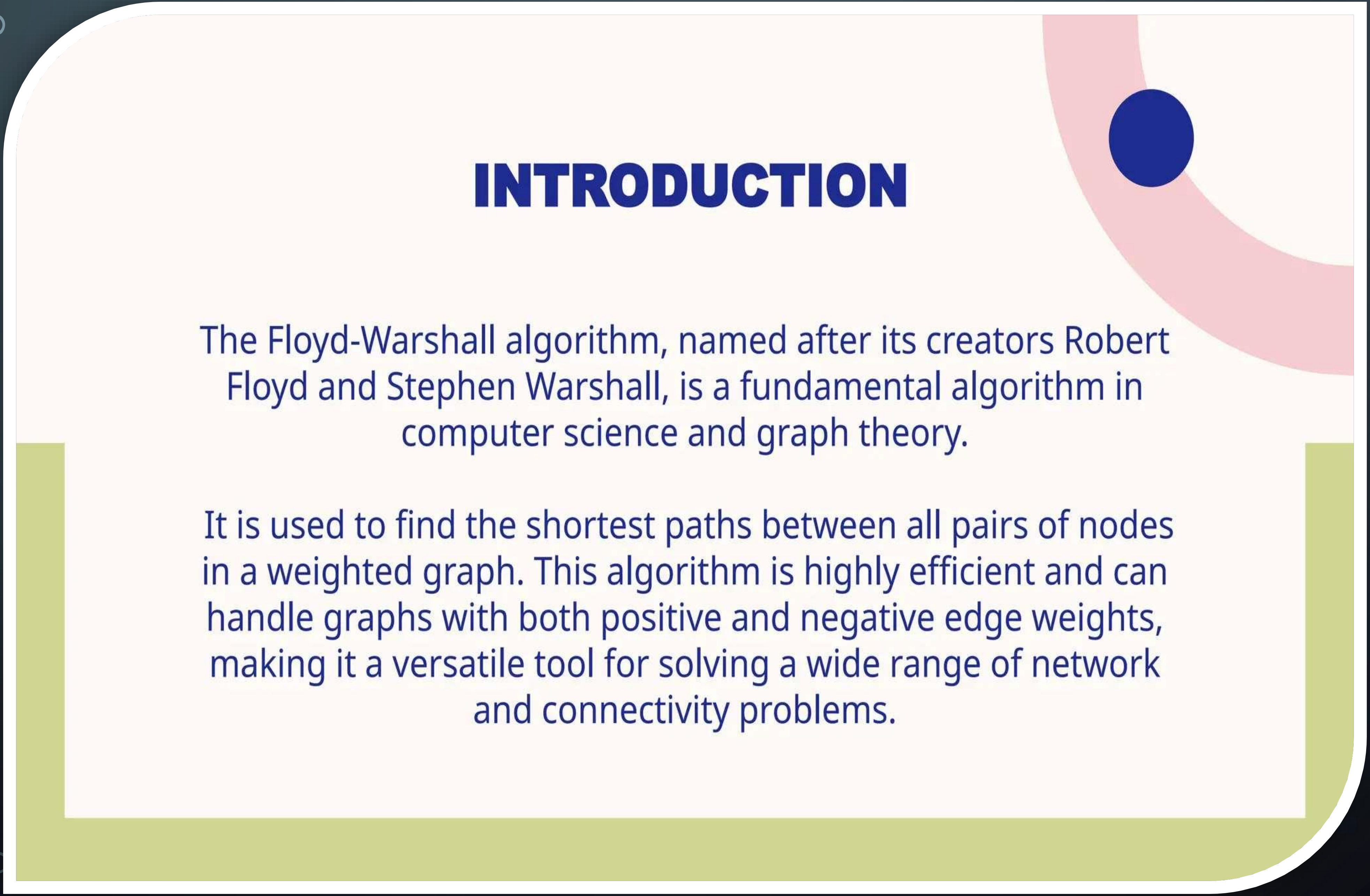
Shortest path are –

E = ACBDE = {5+(-3)+(-2)+2 }= 2

WEEK 12

FLOYD WAR SHALL & KNAPSACK 0/1

Page
176-193

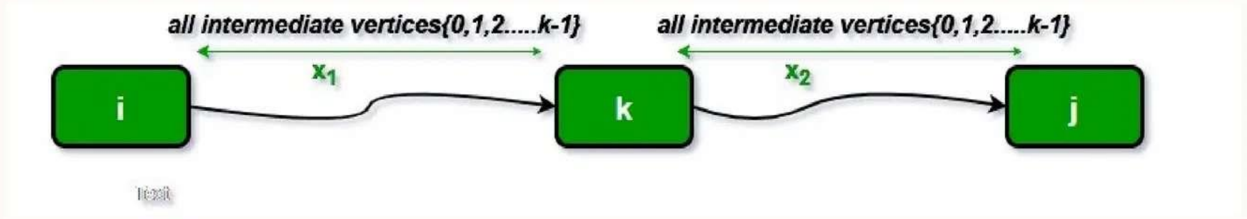


INTRODUCTION

The Floyd-Warshall algorithm, named after its creators Robert Floyd and Stephen Warshall, is a fundamental algorithm in computer science and graph theory.

It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

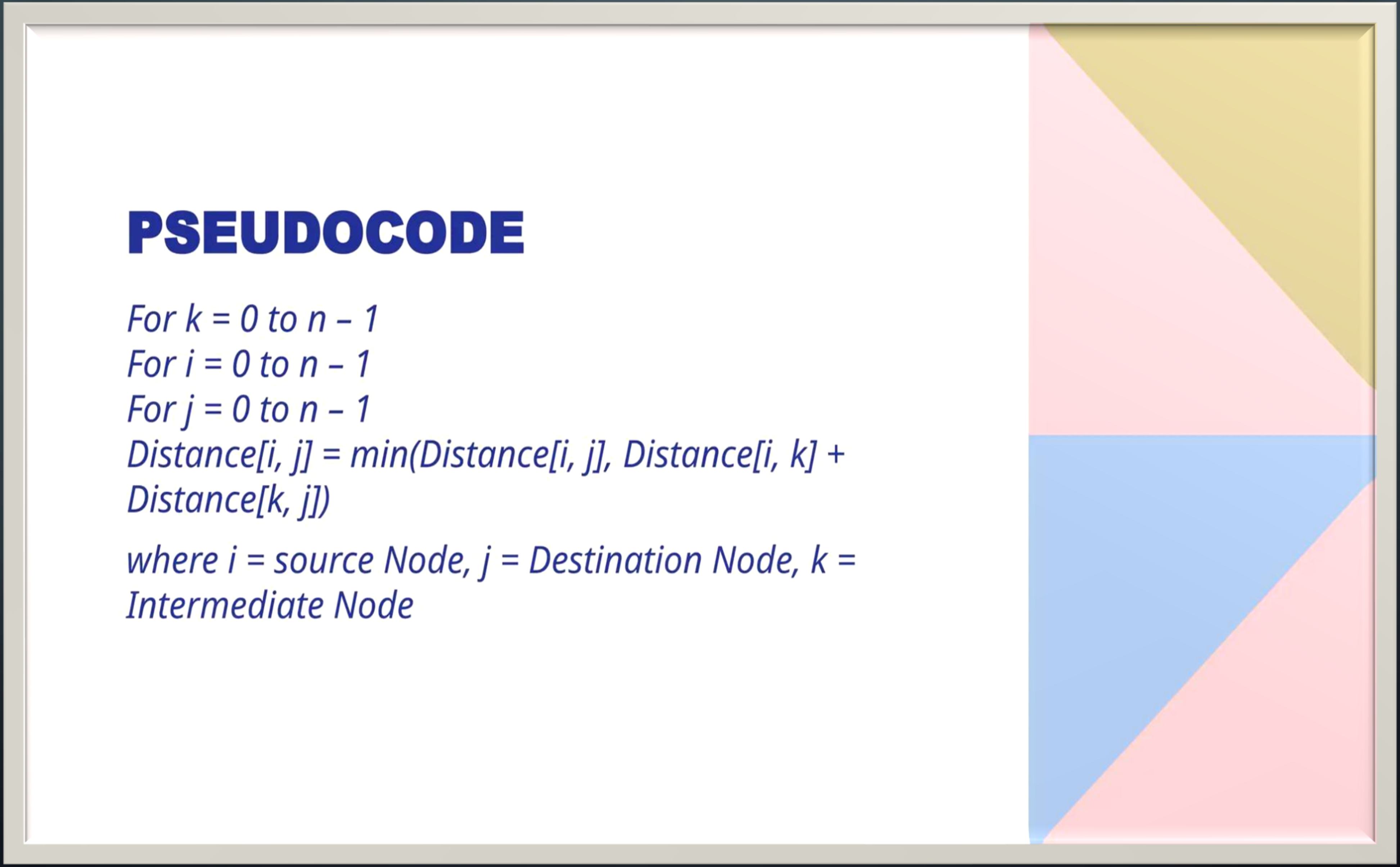
The following figure shows the above optimal substructure property in floyd warshall algorithm:


all intermediate vertices{0,1,2.....k-1}
all intermediate vertices{0,1,2.....k-1}
x1
x2
i
k
j
Text

ALGORITHM

•Initialize the solution matrix same as the input graph matrix as a first step.
•Then update the solution matrix by considering all vertices as an intermediate vertex.
•The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
•When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
•For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

- k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
- k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j], if dist[i][j] > dist[i][k] + dist[k][j]

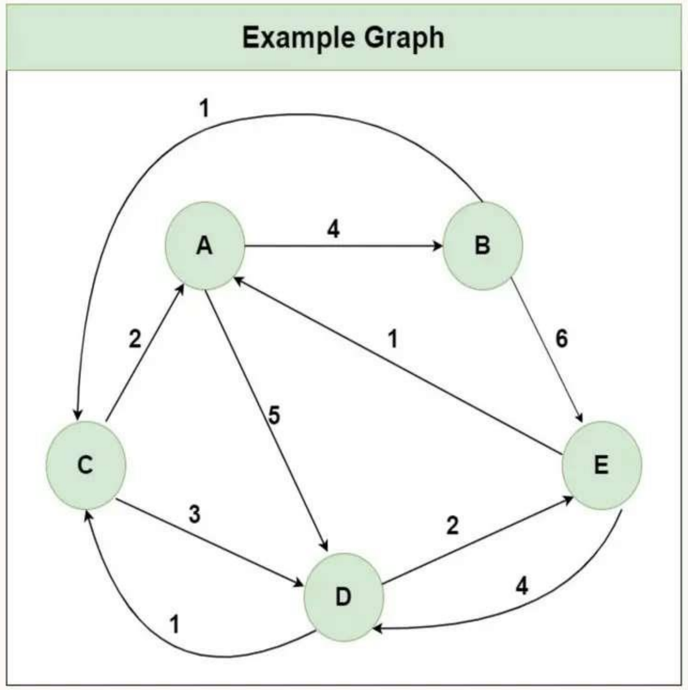


PSEUDOCODE

```
For $k = 0$ to $n - 1$
    For $i = 0$ to $n - 1$
        For $j = 0$ to $n - 1$
            $Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])$
```

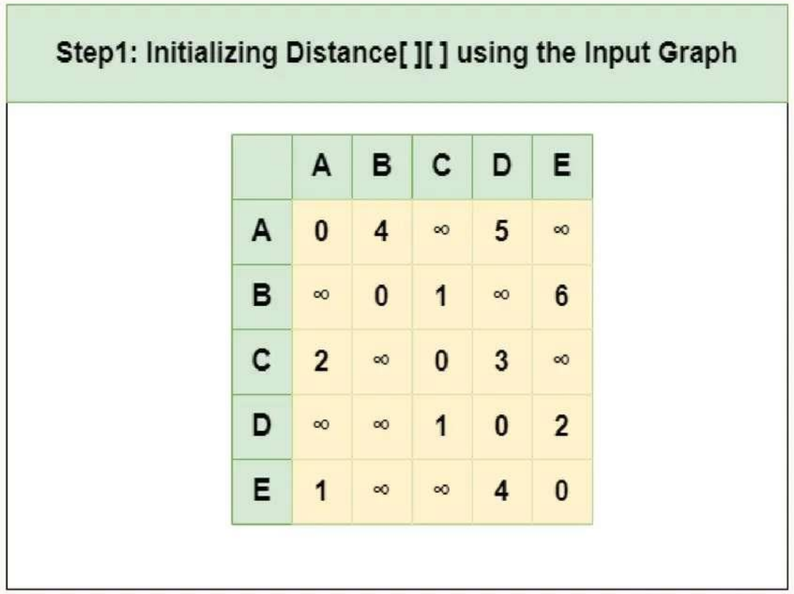
where i = source Node, j = Destination Node, k = Intermediate Node

EXAMPLE



STEP-1

Step 1: Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from i to j, also Distance[i][j] = Infinity if there is no edge from i to j.

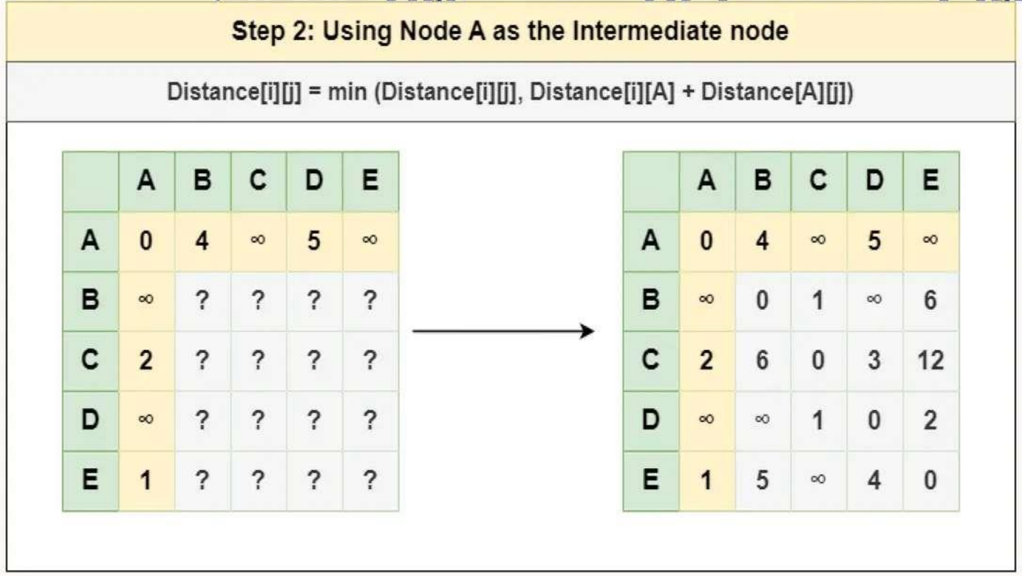


Step1: Initializing Distance[ ][ ] using the Input Graph

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

STEP-2

Step 2: Treat node A as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to A) + (Distance from A to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][A] + Distance[A][j])



Step 2: Using Node A as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

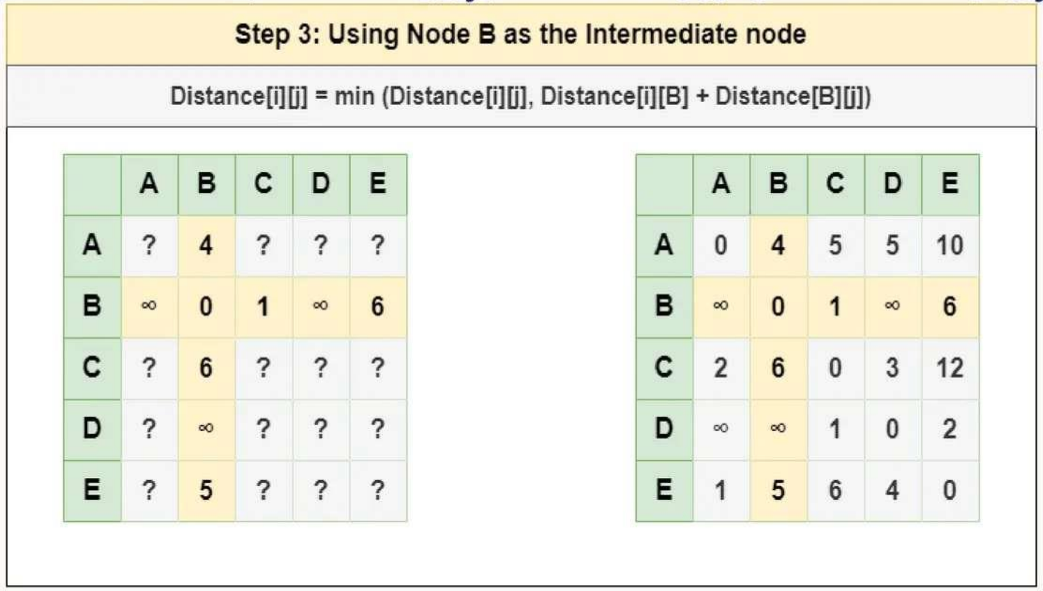
	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

$\longrightarrow$

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

STEP-3

Step 3: Treat node B as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to B) + (Distance from B to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][B] + Distance[B][j])



Step 3: Using Node B as the Intermediate node

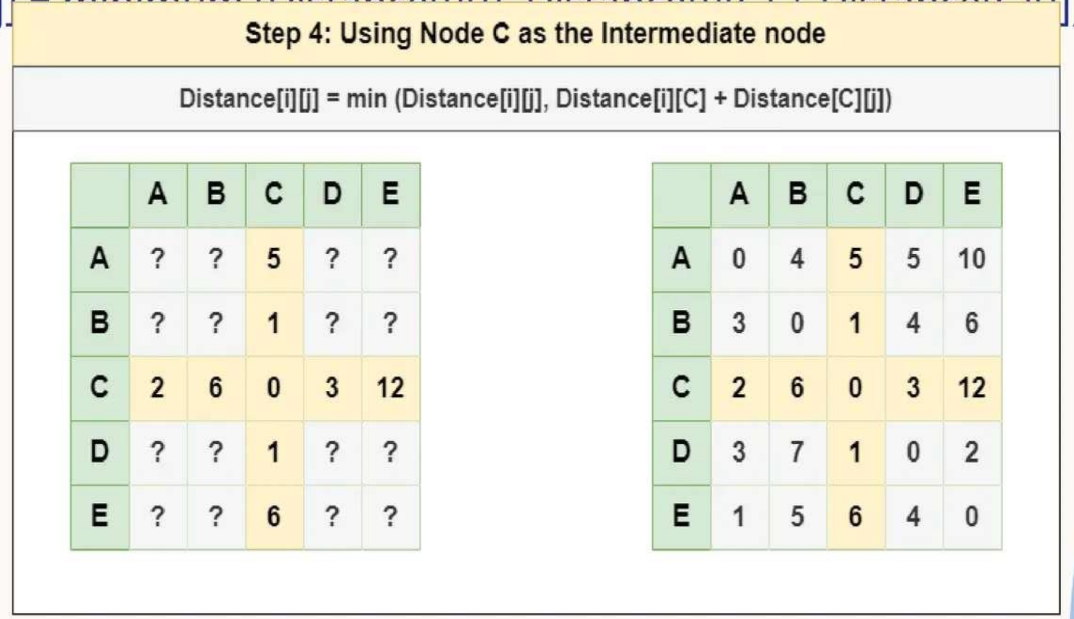
Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

STEP-4

Step 4: Treat node C as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to C) + (Distance from C to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][C] + Distance[C][j])



Step 4: Using Node C as the Intermediate node

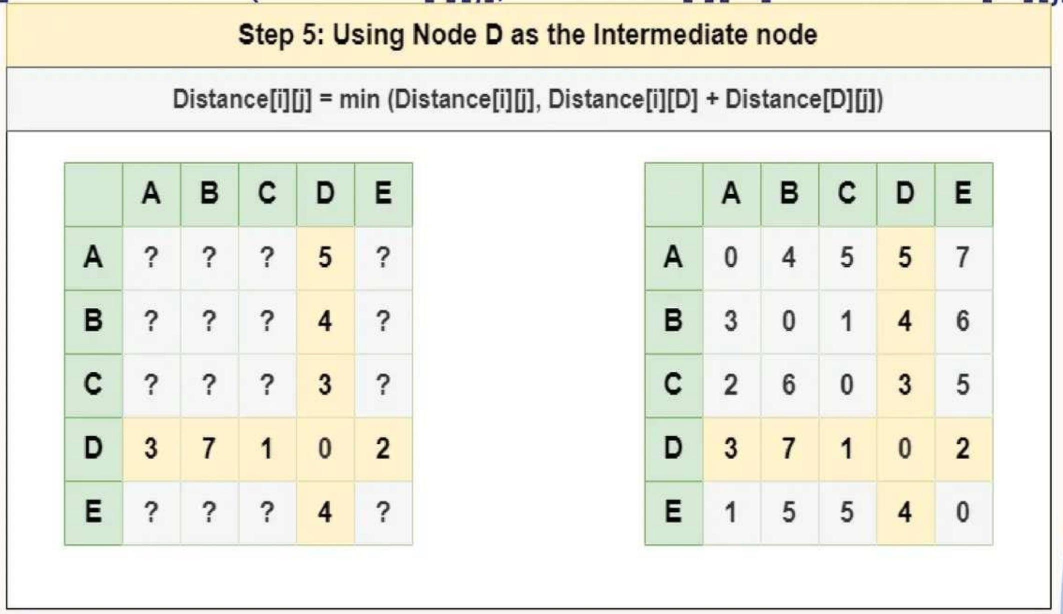
Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

STEP-5

Step 5: Treat node **D** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **D**) + (Distance from **D** to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][**D**] + Distance[**D**][j])



Step 5: Using Node D as the Intermediate node

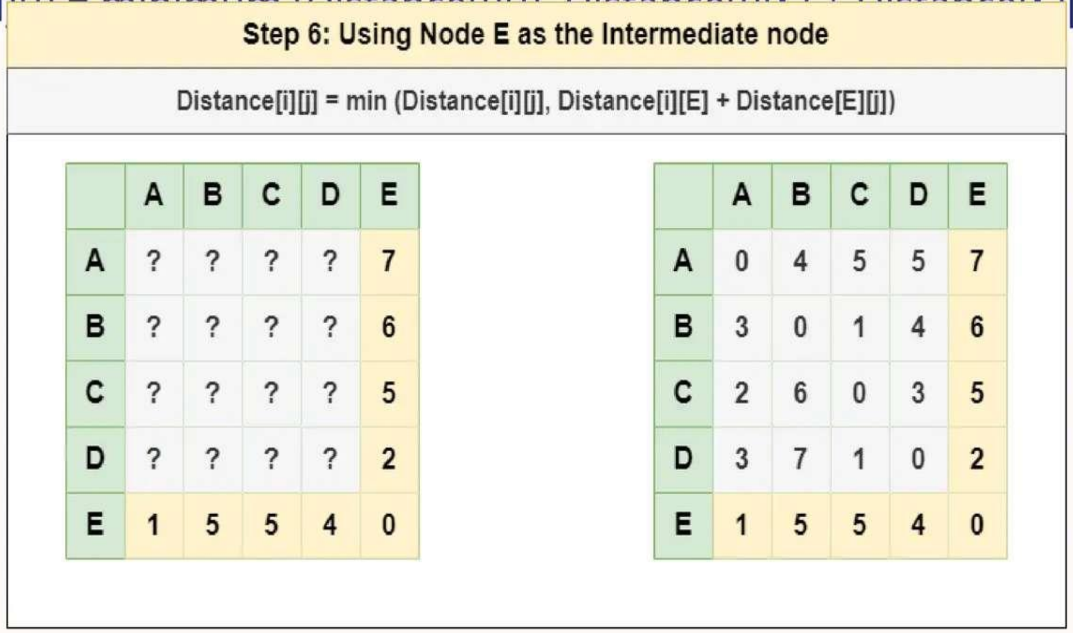
Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

STEP-6

Step 6: Treat node **E** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **E**) + (Distance from **E** to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][**E**] + Distance[**E**][j])


Step 6: Using Node E as the Intermediate node

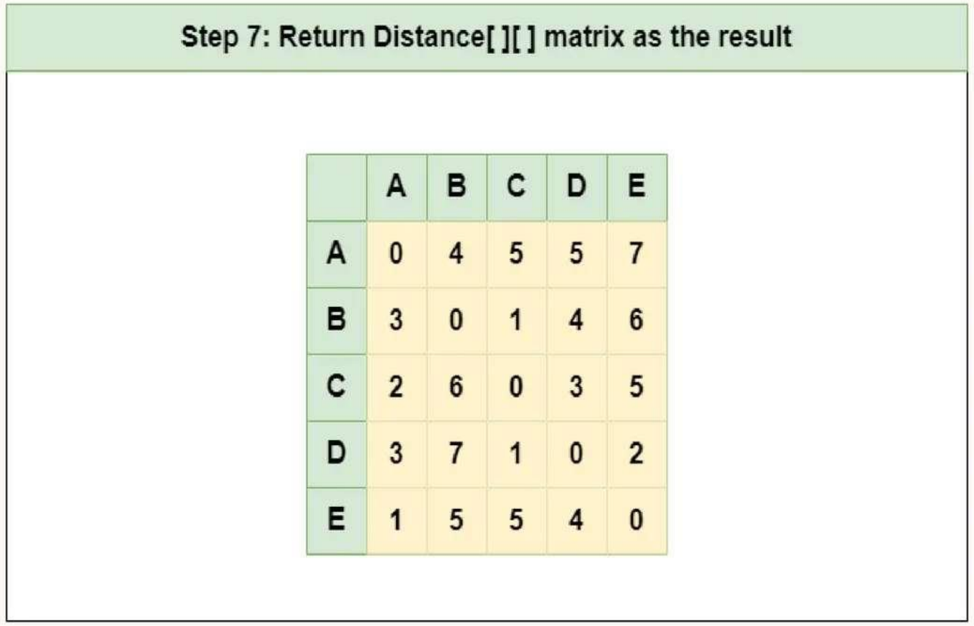
Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

STEP-7

Step 7: Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.



Step 7: Return Distance[ ][ ] matrix as the result

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

0/1 DP KNAPSACK PROBLEM

- Knapsack is basically means bag. A bag of given capacity. We want to pack n items in your luggage.

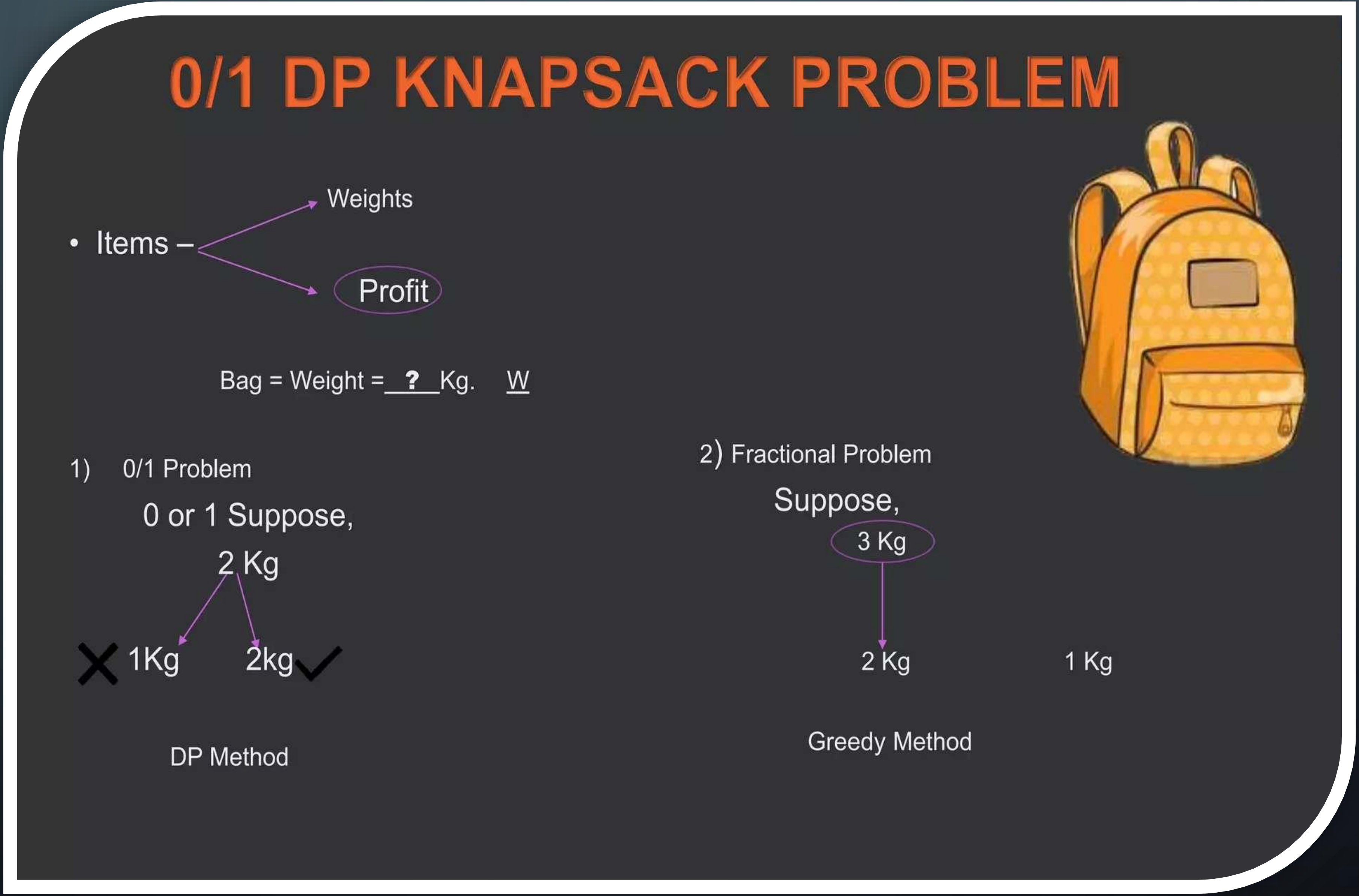
- **Input:**

  Knapsack of capacity

  List (Array) of weight and their corresponding value.

- **Output:** To maximize profit and minimize weight in capacity.

The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.



0/1 DP KNAPSACK PROBLEM

$$\begin{tikzcd} & \text{Weights} \\ \text{• Items –} \arrow[ur] \arrow[dr] & \\ & \boxed{\text{Profit}} \end{tikzcd}$$

Bag = Weight = __?__ Kg. W

1) 0/1 Problem

0 or 1 Suppose,

$$\begin{tikzcd} & \text{2 Kg} \arrow[dl] \arrow[dr] & \\ \text{✗ 1Kg} & & \text{2kg ✓} \end{tikzcd}$$

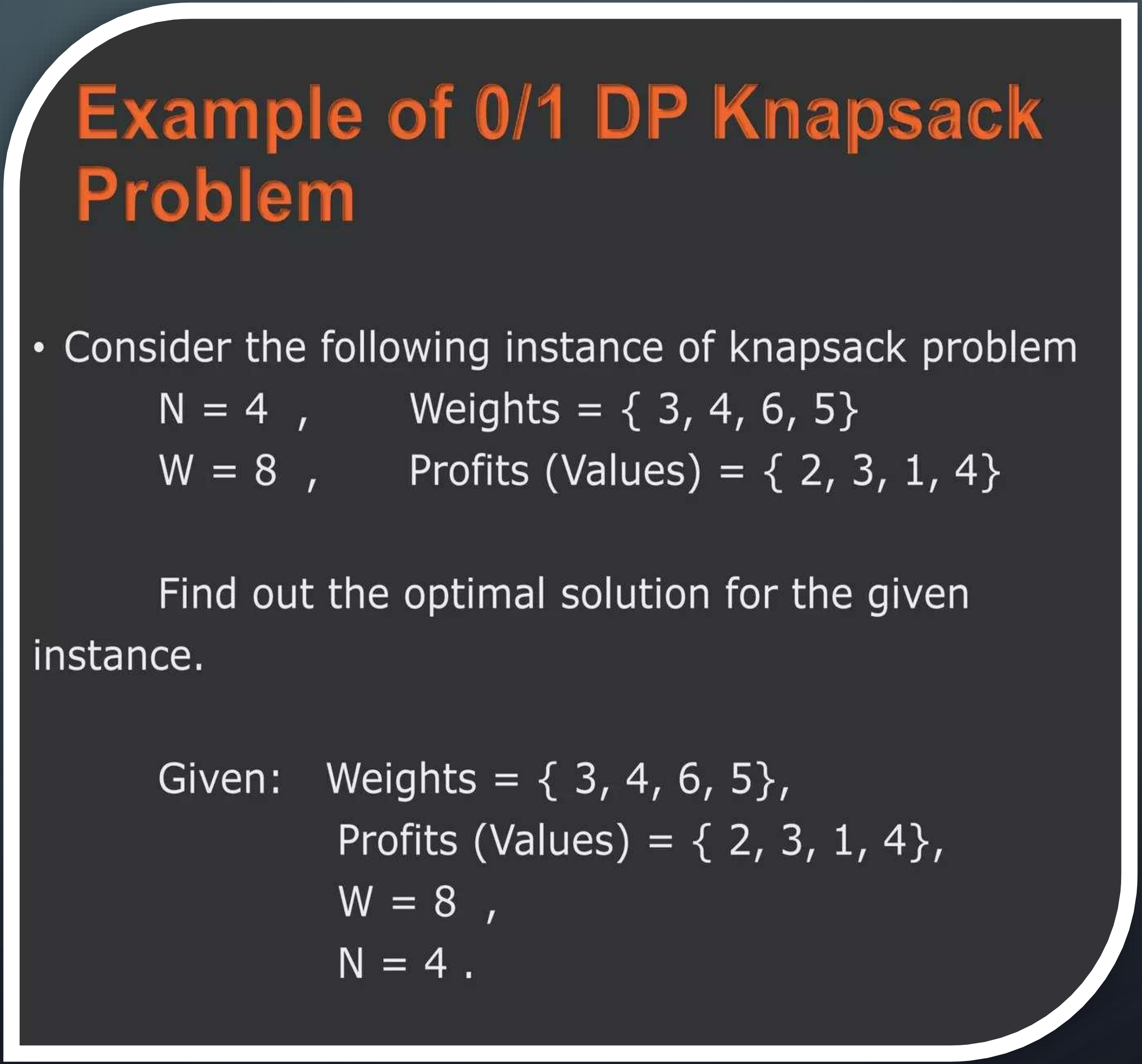
DP Method

2) Fractional Problem

Suppose,

$$\begin{tikzcd} \boxed{\text{3 Kg}} \arrow[d] & \\ \text{2 Kg} & \text{1 Kg} \end{tikzcd}$$

Greedy Method



Example of 0/1 DP Knapsack Problem

- Consider the following instance of knapsack problem

  N = 4 , Weights = { 3, 4, 6, 5}

  W = 8 , Profits (Values) = { 2, 3, 1, 4}

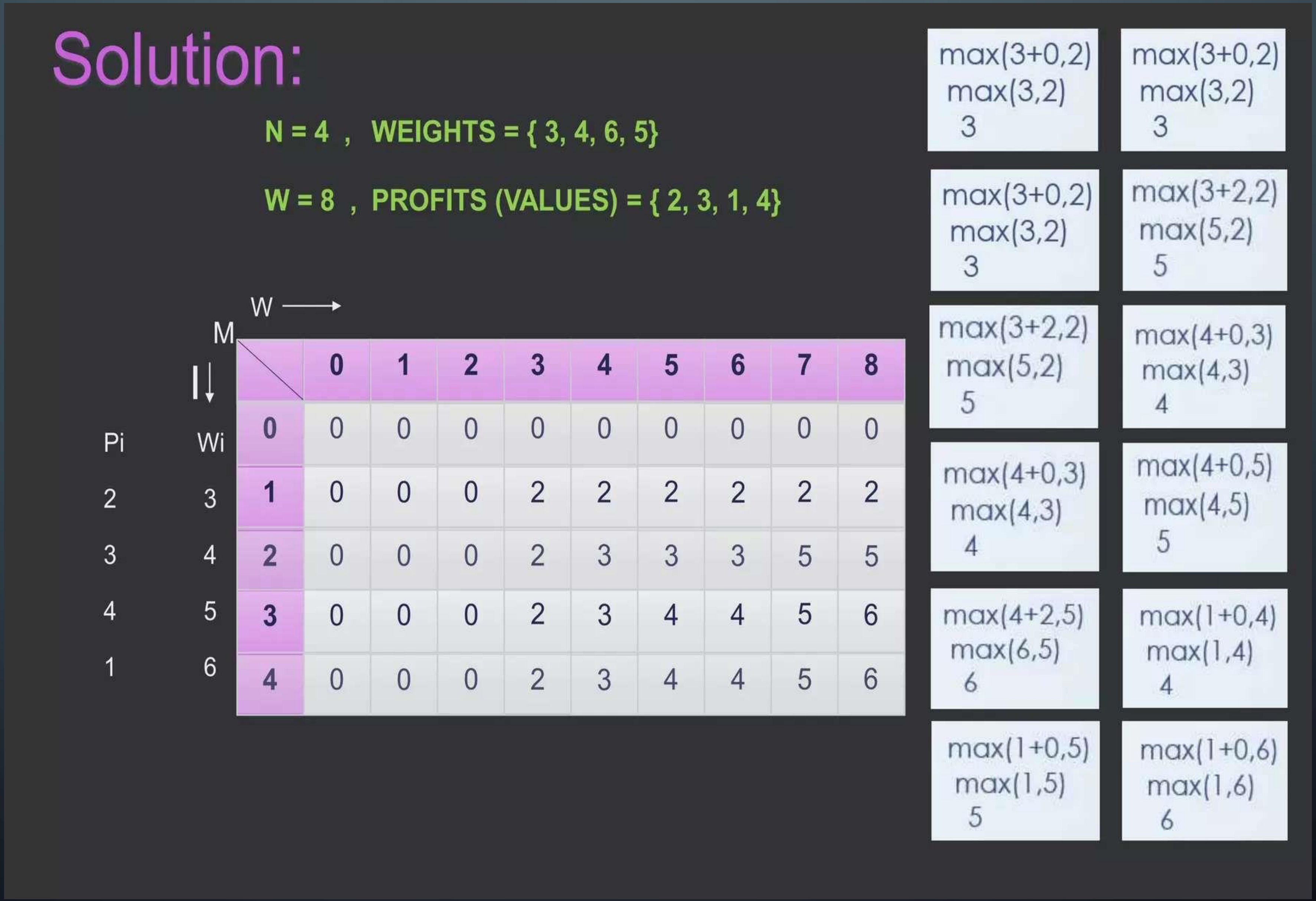
  Find out the optimal solution for the given instance.

  Given: Weights = { 3, 4, 6, 5},

  Profits (Values) = { 2, 3, 1, 4},

  W = 8 ,

  N = 4 .



Solution:

N = 4 , WEIGHTS = { 3, 4, 6, 5}

W = 8 , PROFITS (VALUES) = { 2, 3, 1, 4}

W ⟶

M ↓

Pi	Wi	M \ W	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
2	3	**1**	0	0	0	2	2	2	2	2	2
3	4	**2**	0	0	0	2	3	3	3	5	5
4	5	**3**	0	0	0	2	3	4	4	5	6
1	6	**4**	0	0	0	2	3	4	4	5	6

max(3+0,2)
max(3,2)
3

max(3+0,2)
max(3,2)
3

max(3+0,2)
max(3,2)
3

max(3+2,2)
max(5,2)
5

max(3+2,2)
max(5,2)
5

max(4+0,3)
max(4,3)
4

max(4+0,3)
max(4,3)
4

max(4+0,5)
max(4,5)
5

max(4+2,5)
max(6,5)
6

max(1+0,4)
max(1,4)
4

max(1+0,5)
max(1,5)
5

max(1+0,6)
max(1,6)
6

0/1 DP KNAPSACK PROBLEM

Algorithm of Knapsack Problem

```
Dynamic-0-1-knapsack (v, w, n, W)
{
   for w = 0 to W do
      c[0, w] = 0
   for i = 1 to n do
      c[i, 0] = 0
   for w = 1 to W do
      if wi ≤ w then
         if vi + c[i-1, w-wi] then
            c[i, w] = vi + c[i-1, w-wi]
         else c[i, w] = c[i-1, w]
      else
         c[i, w] = c[i-1, w]
}
```

SELF STUDY

PRACTICE PROBLEM & CODING

WEEK 13
LCS

Page
195-206

LONGEST COMMON SUBSEQUENCE

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

In LSC , we have to find Longest common Subsequence that is in same relative order.

String of length n has 2^n different possible subsequences.

E.g.--

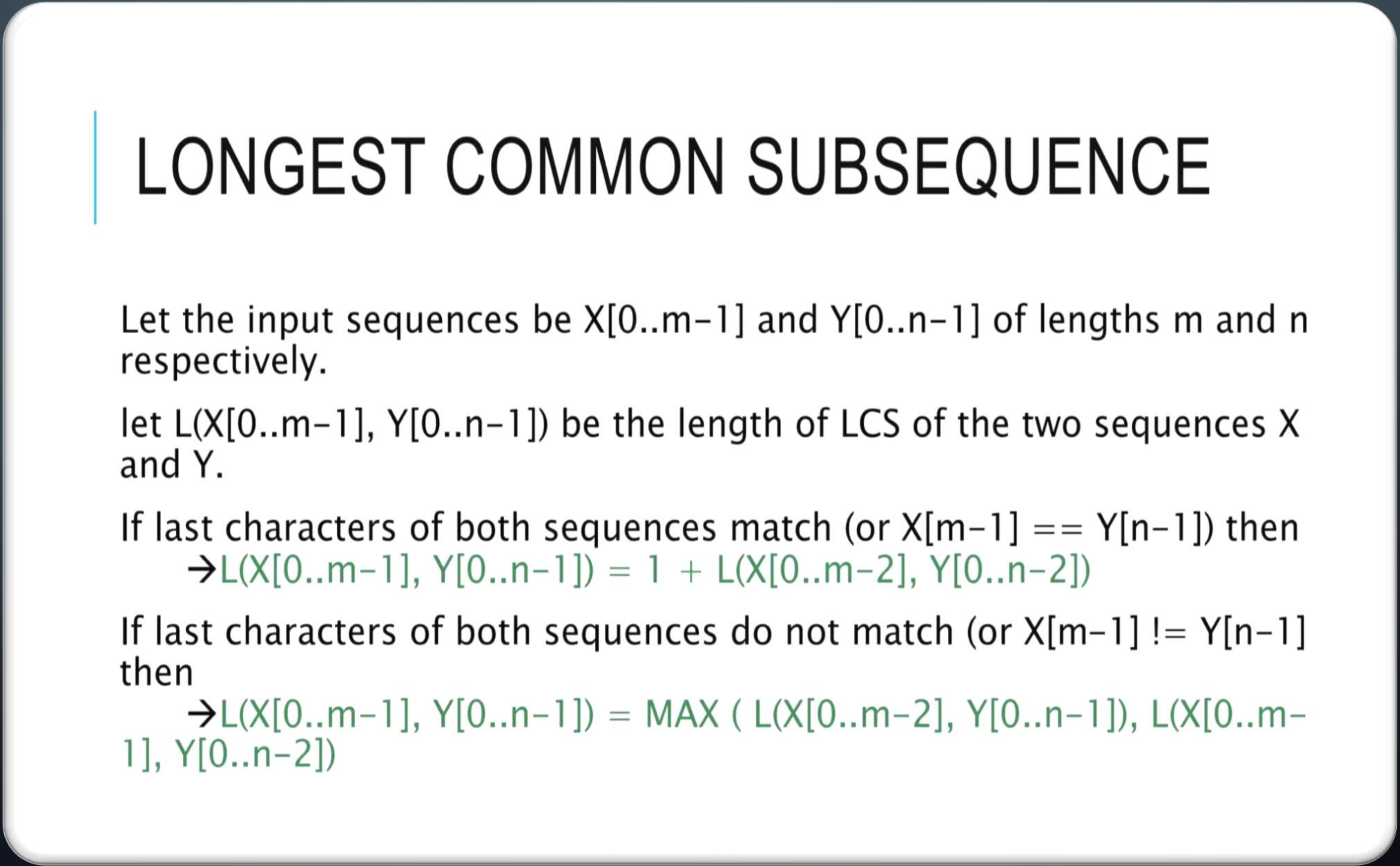
Subsequences of “ABCDEFG”.

“ABC”, “ABG”, “BDF”, “AEG”, ‘”ACEFG”, …

EXAMPLE

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.



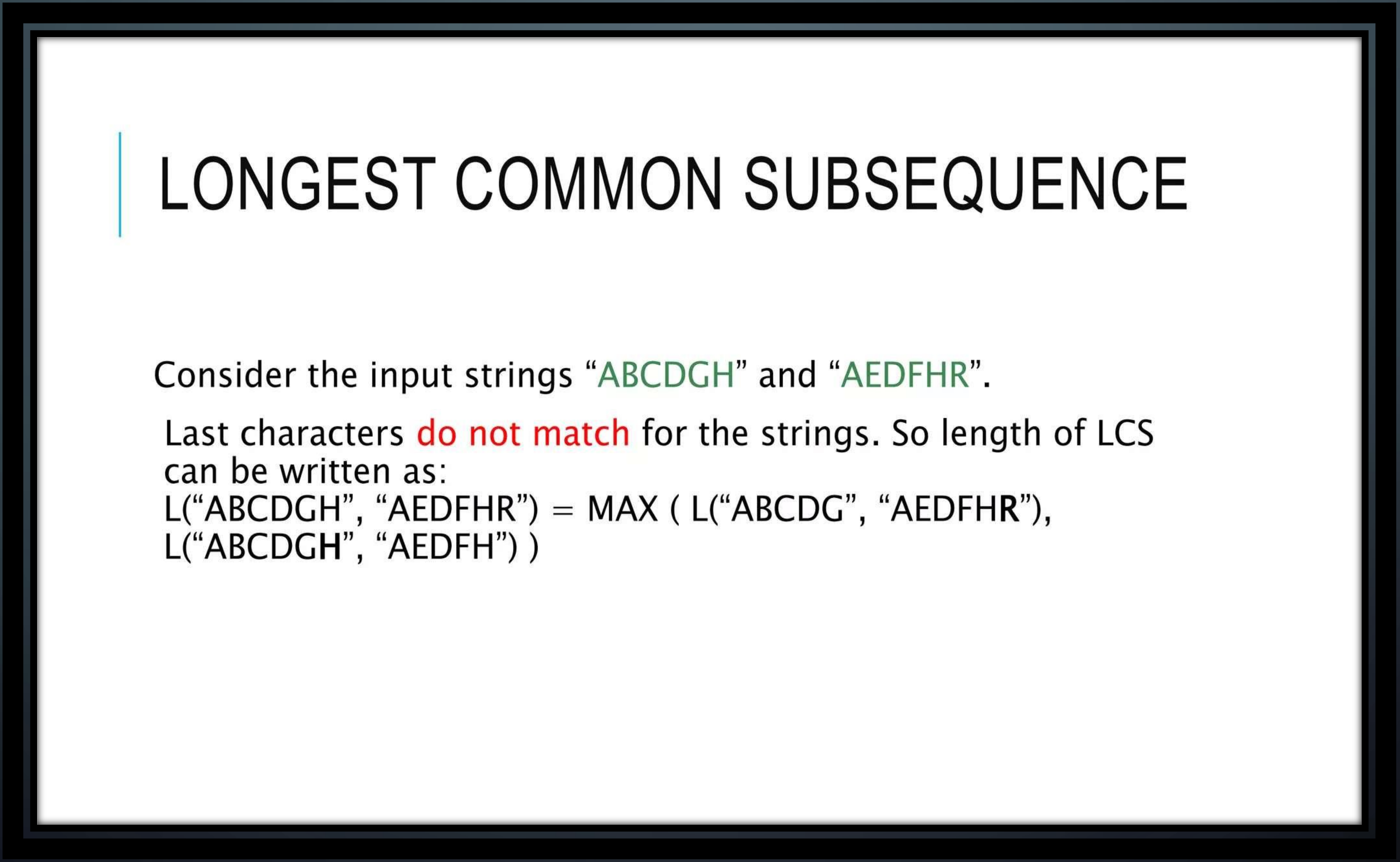
LONGEST COMMON SUBSEQUENCE

Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively.

let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y.

If last characters of both sequences match (or X[m-1] == Y[n-1]) then
→L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1] then
→L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])



LONGEST COMMON SUBSEQUENCE

Consider the input strings “ABCDGH” and “AEDFHR”.

Last characters do not match for the strings. So length of LCS can be written as:
L(“ABCDGH”, “AEDFHR”) = MAX (L(“ABCDG”, “AEDFH**R**”), L(“ABCDG**H**”, “AEDFH”))

EXAMPLE

Following is a partial recursion tree for input strings “AXYT” and “AYZX”

```
                        lcs("AXYT", "AYZX")
                      /                 \
         lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
         /              \                /               \
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

EXAMPLE

X=M,Z,J,A,W,X,U

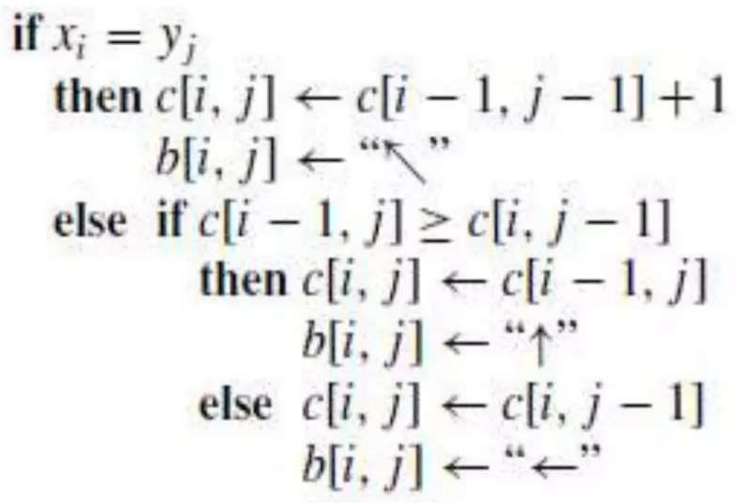
Y=X,M,J,Y,A,U,Z

And Longest Common Subsequence is,

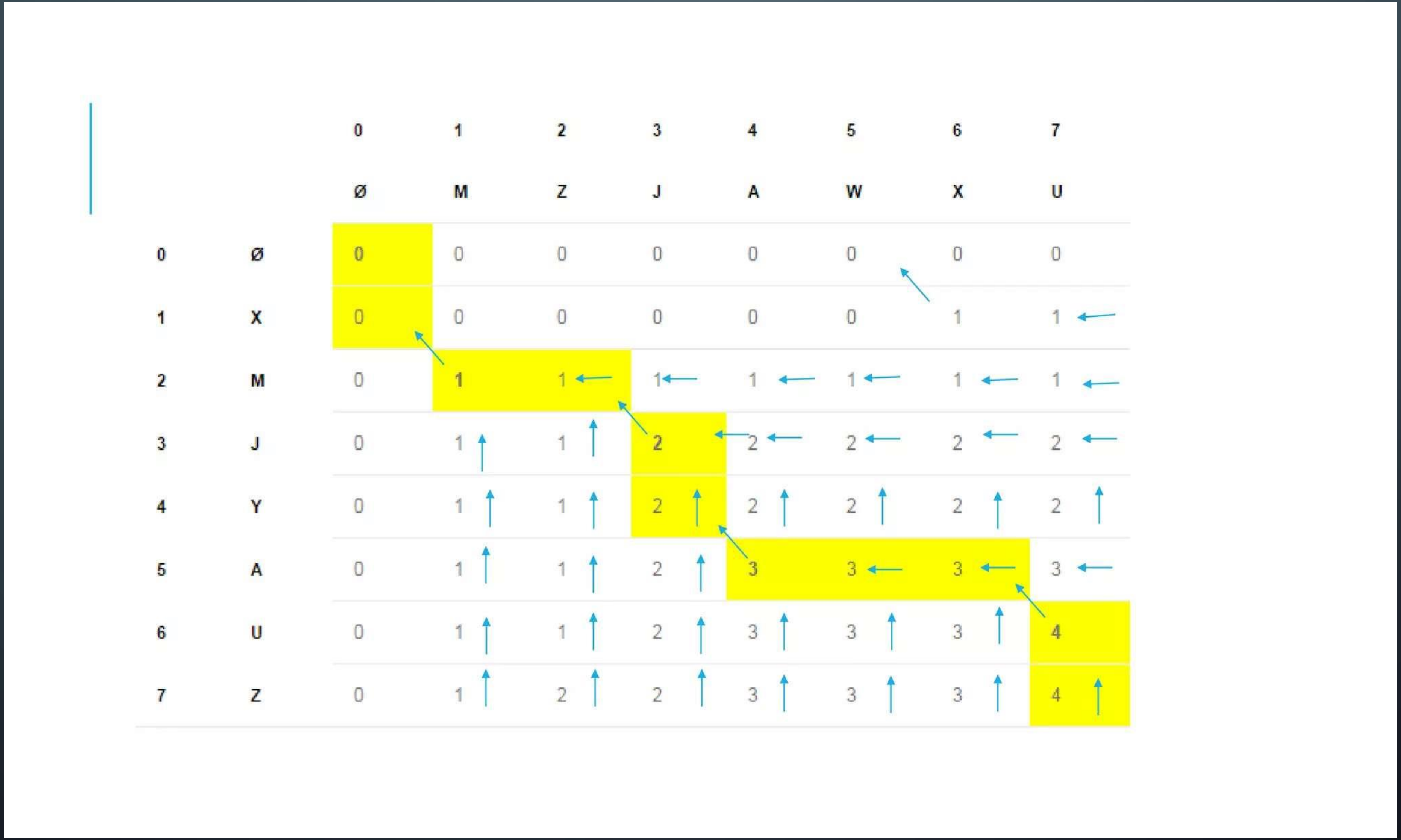
LCS(X,Y)=M,J,A,U

Now we will see table from which we can find LCS of Above sequences.

CONDITIONS FOR ARROWS

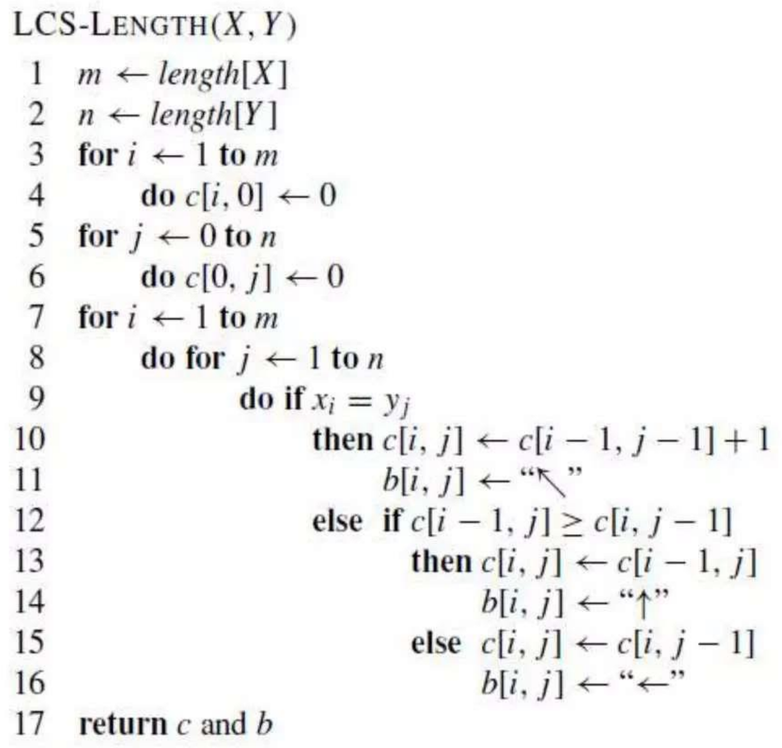


```
if $x_i = y_j$
  then $c[i, j] \leftarrow c[i-1, j-1] + 1$
       $b[i, j] \leftarrow$ "$\nwarrow$"
  else if $c[i-1, j] \geq c[i, j-1]$
         then $c[i, j] \leftarrow c[i-1, j]$
              $b[i, j] \leftarrow$ "$\uparrow$"
         else $c[i, j] \leftarrow c[i, j-1]$
              $b[i, j] \leftarrow$ "$\leftarrow$"
```



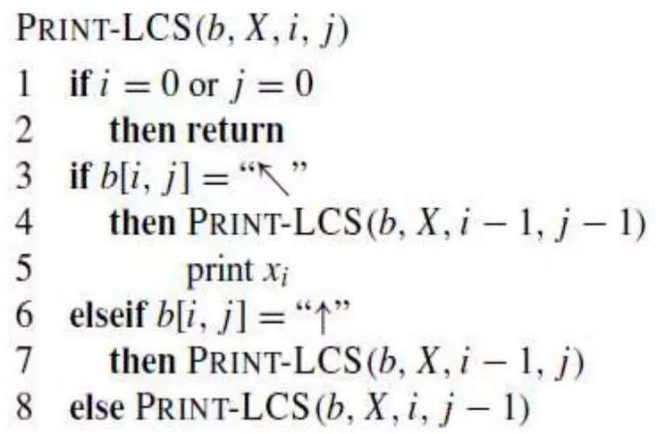
		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	↖ 1	1 ←
2	M	0	↖ 1	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←
3	J	0	1 ↑	1 ↑	↖ 2	2 ←	2 ←	2 ←	2 ←
4	Y	0	1 ↑	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑
5	A	0	1 ↑	1 ↑	2 ↑	↖ 3	3 ←	3 ←	3 ←
6	U	0	1 ↑	1 ↑	2 ↑	3 ↑	3 ↑	3 ↑	↖ 4
7	Z	0	1 ↑	2 ↑	2 ↑	3 ↑	3 ↑	3 ↑	4 ↑

LCS ALGORITHM



```
LCS-LENGTH($X, Y$)
$m \leftarrow length[X]$
$n \leftarrow length[Y]$
for $i \leftarrow 1$ to $m$
    do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ to $n$
    do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ to $m$
    do for $j \leftarrow 1$ to $n$
        do if $x_i = y_j$
            then $c[i, j] \leftarrow c[i-1, j-1] + 1$
                 $b[i, j] \leftarrow$ "$\nwarrow$"
            else if $c[i-1, j] \geq c[i, j-1]$
                then $c[i, j] \leftarrow c[i-1, j]$
                     $b[i, j] \leftarrow$ "$\uparrow$"
                else $c[i, j] \leftarrow c[i, j-1]$
                     $b[i, j] \leftarrow$ "$\leftarrow$"
return $c$ and $b$
```

CONSTRUCTING AN LCS



```
PRINT-LCS($b, X, i, j$)
if $i = 0$ or $j = 0$
    then return
if $b[i, j] =$ "$\nwarrow$"
    then PRINT-LCS($b, X, i - 1, j - 1$)
        print $x_i$
elseif $b[i, j] =$ "$\uparrow$"
    then PRINT-LCS($b, X, i - 1, j$)
else PRINT-LCS($b, X, i, j - 1$)
```

COMPLEXITY

Complexity of Longest Common Subsequence is O(mn).

Where m and n are lengths of the two Strings.

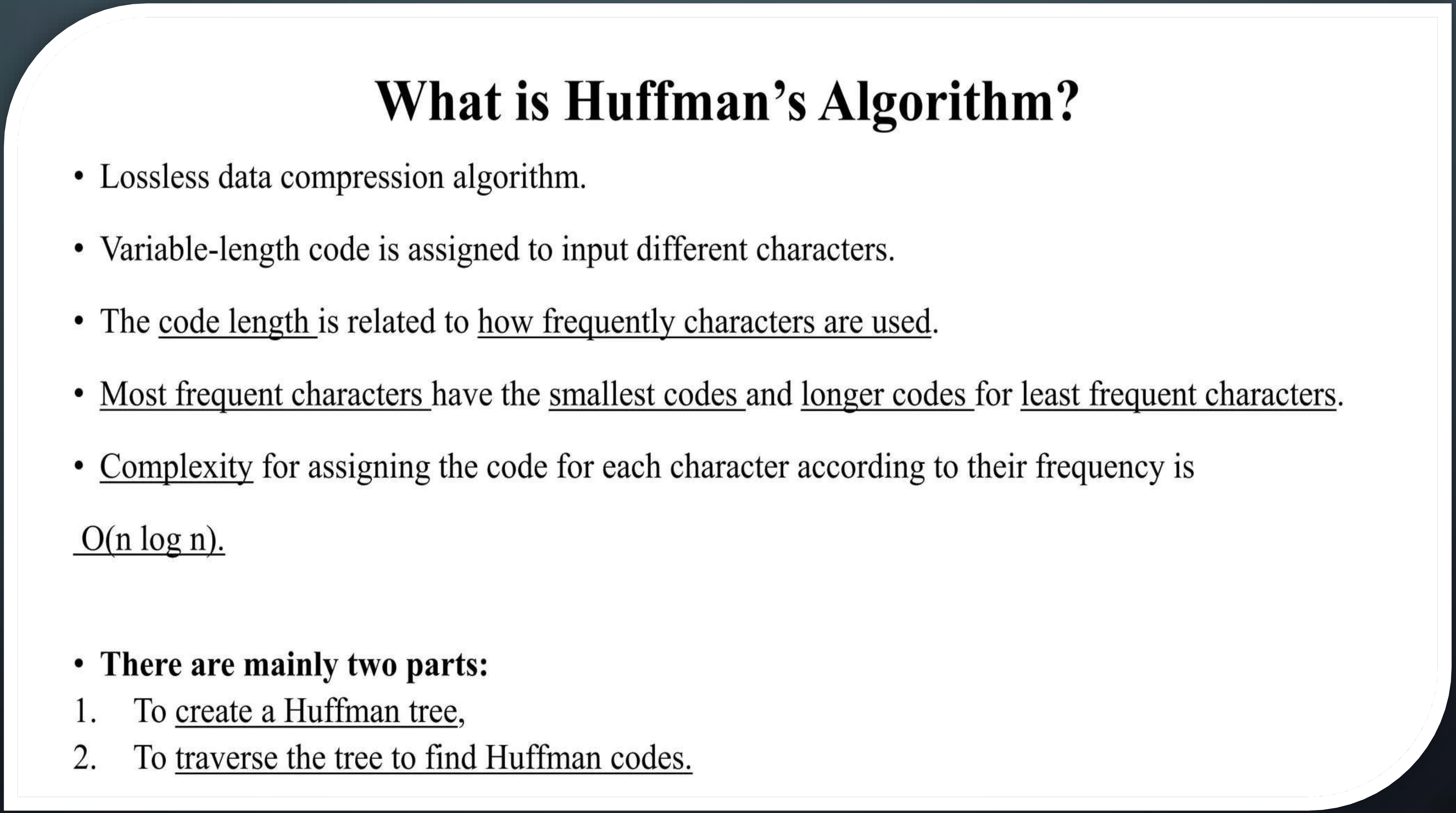
SELF STUDY

PRACTICE CODING

Week 14

Huffman Coding

Page
207-215



What is Huffman's Algorithm?

- Lossless data compression algorithm.
- Variable-length code is assigned to input different characters.
- The code length is related to how frequently characters are used.
- Most frequent characters have the smallest codes and longer codes for least frequent characters.
- Complexity for assigning the code for each character according to their frequency is

O(n log n).

- **There are mainly two parts:**

1. To create a Huffman tree,
2. To traverse the tree to find Huffman codes.

Steps to build Huffman Tree:

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root).

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Steps to Building Huffman's Tree:

Let's understand the algorithm with example:

Example 1:

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

1. 5+9=14(Internal Node)
2. 12+13=25(Internal Node)
3. 14+16=30(Internal Node)
4. 25+30=55.(Internal Node)
5. 55+45=100.(Root Node)

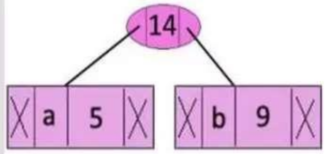
Continue..

Step 1: Build Min Heap DS, Which contains 6 Nodes(Given example having 6 Char.) Each node represent as root.

Step 2: Extract two minimum frequency nodes from min heap. Add a new internal node with frequency

5 + 9 = 14. Now 14 becomes internal root node.

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

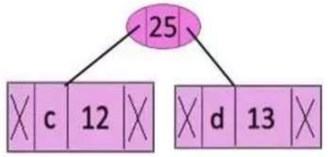


Step 3: Again extract two minimum frequency nodes from heap.

Add a new internal node with frequency

12 + 13 = 25. Now 25 becomes internal root node.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

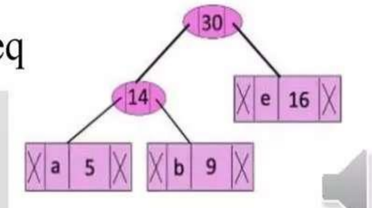


Step 4: Again extract two minimum frequency nodes. Add a new internal node with freq

Now 30 becomes internal root node.

Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45



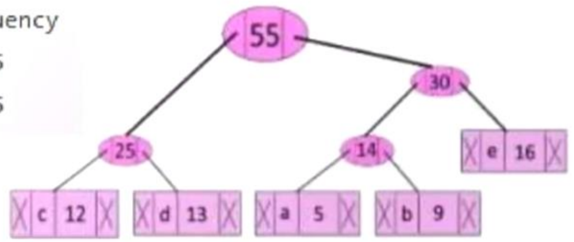
Continue..

Step 5: Extract two minimum frequency nodes.

Add a new internal node with frequency 25 + 30 = 55.

Now min heap contains 2 nodes.

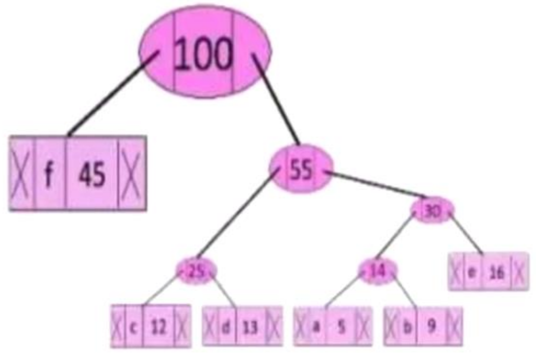
character	Frequency
f	45
Internal Node	55



Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100.

Step 7: Now min heap contains only one node.

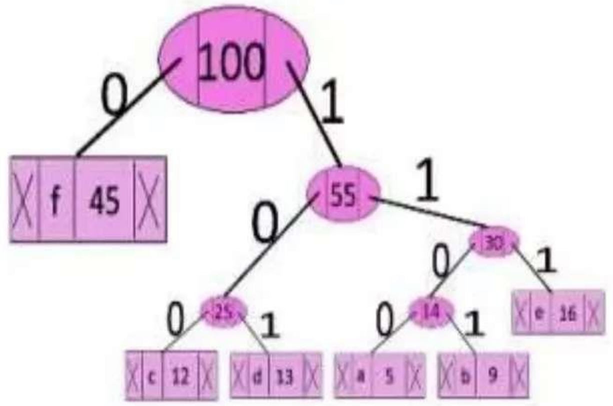
character	Frequency
Internal Node	100



Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

1. Traverse the tree formed starting from the root.
2. Maintain an auxiliary array.
3. While moving to the left child, write 0 to the array.
4. While moving to the right child, write 1 to the array.
5. Print the array when a leaf node is encountered.

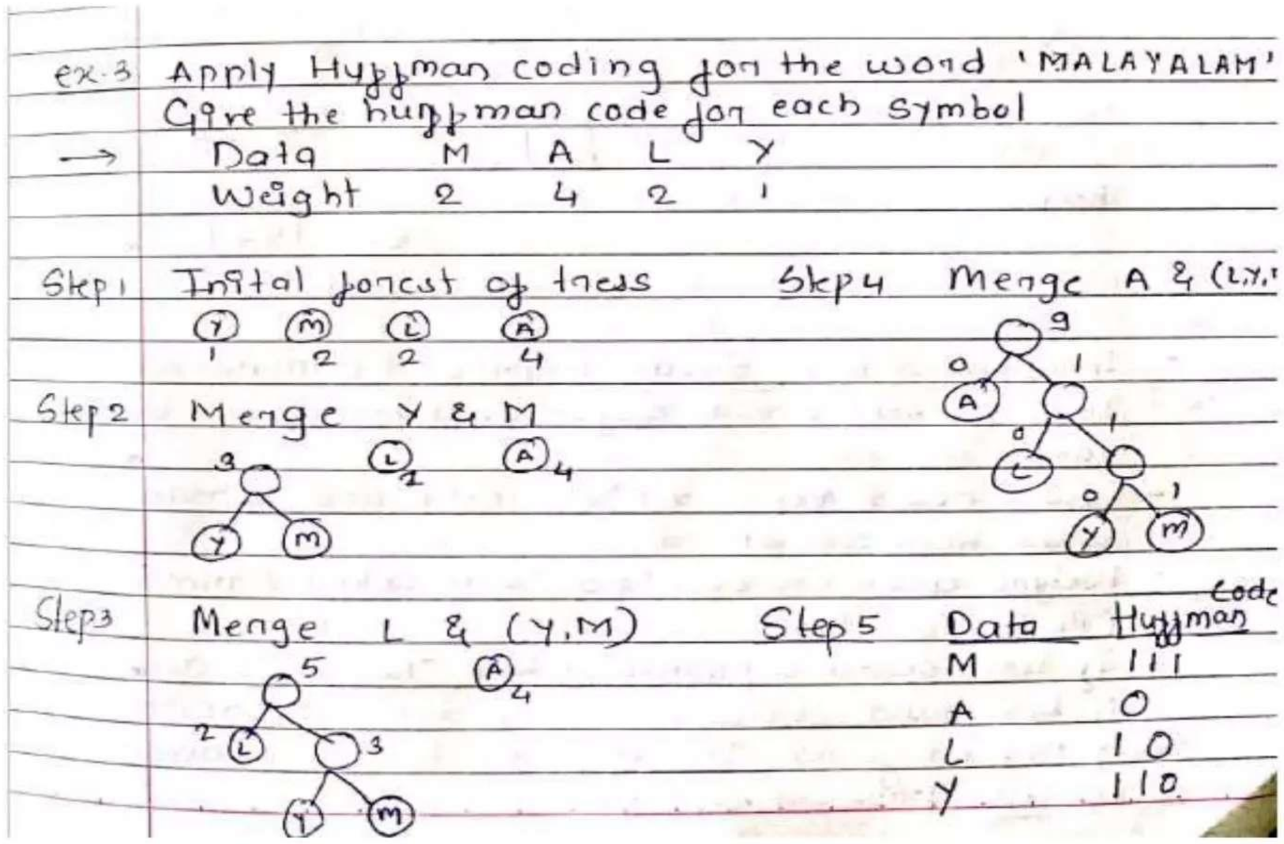


- The codes are as follows:

Characters	**Code-word**
f	0
c	100
d	101
a	1100
b	1101
e	111

Example 2:

Apply Huffman Coding for the word ‘MALAYALAM’. Give Huffman Code for each symbol.



ex.3 Apply Huffman coding for the word 'MALAYALAM'
Give the huffman code for each symbol

→

Data	M	A	L	Y
Weight	2	4	2	1

Step1 Initial forest of trees

Step2 Merge Y & M

Step3 Merge L & (Y,M)

Step4 Merge A & (L,Y,

Step5

Data	Huffman Code
M	111
A	0
L	10
Y	110

Important Questions

- By SPPU Exam Pattern

1. Explain Huffman algorithm with example. 4M

1. Construct Huffman Tree and Code with following data

Character	A	B	C	D	E
Frequency	20	10	10	30	30

5M

3. Apply Huffman Coding for the word ‘ENGINEERING’.
Give Huffman Code for each symbol. 6M

WEEK 15

MAXIMUM SUBARRAY SOLUTION USING DIVIDED & CONQUER

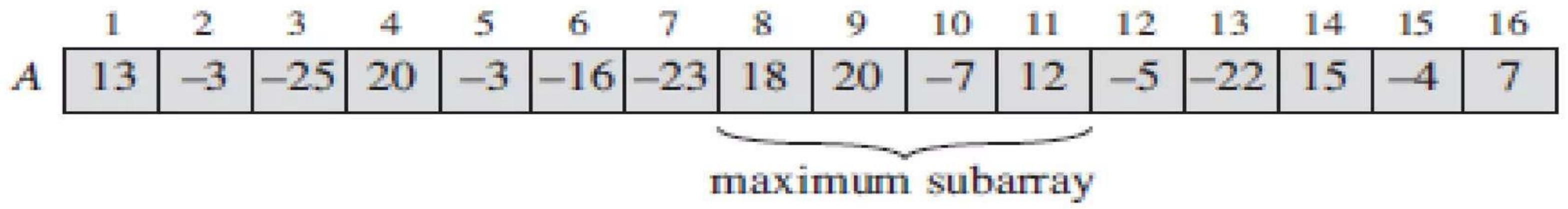
Page
217-224

Formal Problem Definition

- Given a sequence of numbers <a1,a2,.....an> we work to find a subsequence of A that is contiguous and whose values have the maximum sum.
- Maximum Subarray problem is only challenging when having both positive and negative numbers.

Example…

Here, the subarray A[8…11] is the Maximum Subarray, with sum 43, has the greatest sum of any contiguous subarray of array A.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

Brute Force Solution

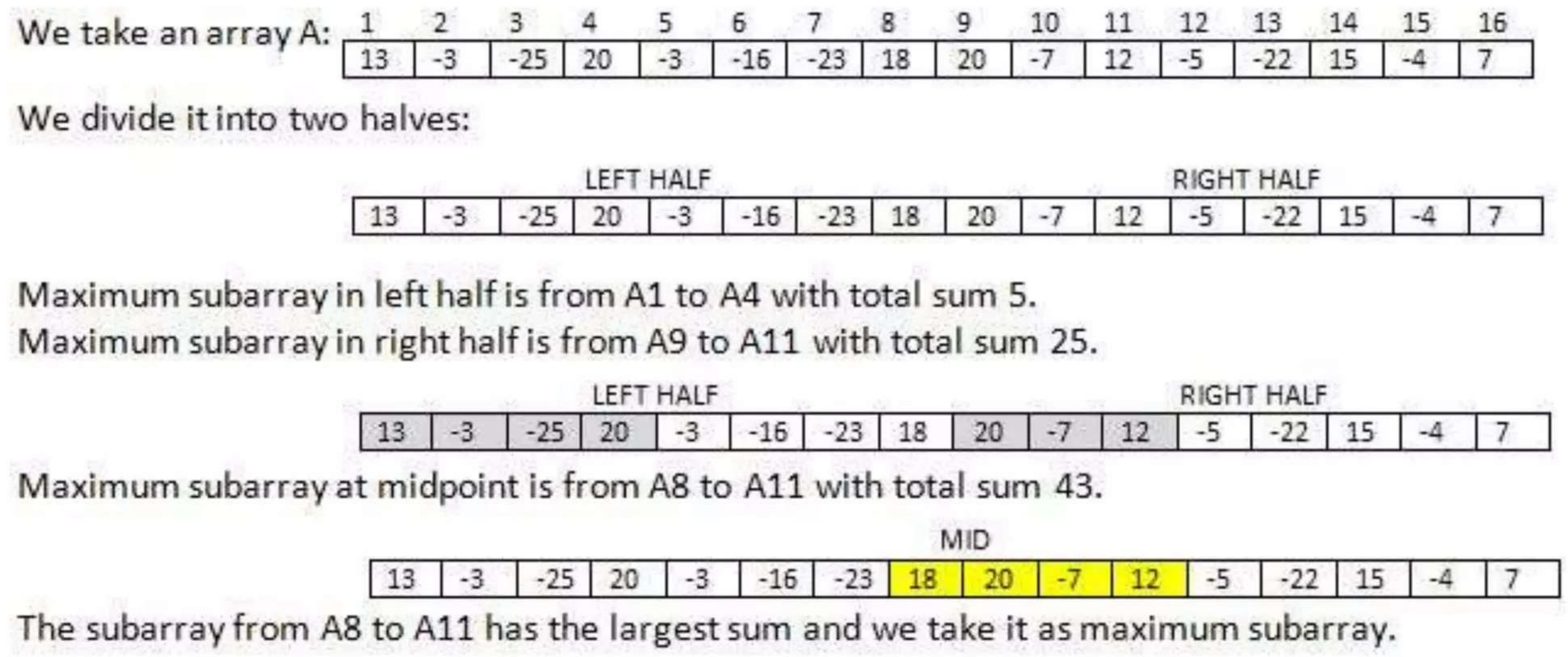
To solve the Maximum Subarray Problem we can use brute force solution however in brute force we will have to compare all possible continuous subarray which will increase running time. Brute force will result in $\Theta(n^2)$, and the best we can hope for is to evaluate each pair in constant time which will result in $\Omega(n^2)$.

Divide and Conquer is a more efficient method for solving large number of problems resulting in $\Theta(n\log n)$.

Divide and Conquer Solution

- First we Divide the array A [low … high] into two subarrays A [low … mid] (left) and A [mid +1 … high] (right).
- We recursively find the maximum subarray of A [low … mid] (left), and the maximum of A [mid +1 … high] (right).
- We also find maximum crossing subarray that crosses the midpoint.
- Finally we take a subarray with the largest sum out of three.

Example...



We take an array A:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

We divide it into two halves:

LEFT HALF RIGHT HALF

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Maximum subarray in left half is from A1 to A4 with total sum 5.
Maximum subarray in right half is from A9 to A11 with total sum 25.

LEFT HALF RIGHT HALF

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Maximum subarray at midpoint is from A8 to A11 with total sum 43.

MID

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

The subarray from A8 to A11 has the largest sum and we take it as maximum subarray.

Time Analysis

- Find-Max-Cross-Subarray takes: θ (n) time
- Two recursive calls on input size n/2 takes: 2T(n/2) time
- Hence:

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

  $T(n) = \theta(n \log n)$

Pseudo Code

```
Max-subarray(A, Left, Right)
if (Right == Left)
        return (left, right, A[left])
else mid= [(left+right)/2]
L1=Find-Maximum-Subarray(A,left,mid)
R1=Find-Maximum-Subarray(A,mid+1,right)
M1=Find-Max-Crossing-Subarray(A,left,mid,right)
If sum(L1) > sum(R1) and sum(L1) > sum(M1)
Return L1
elseif sum(R1) > sum(L1) and sum(R1) > sum(M1)
Return R1
Else return M1
```

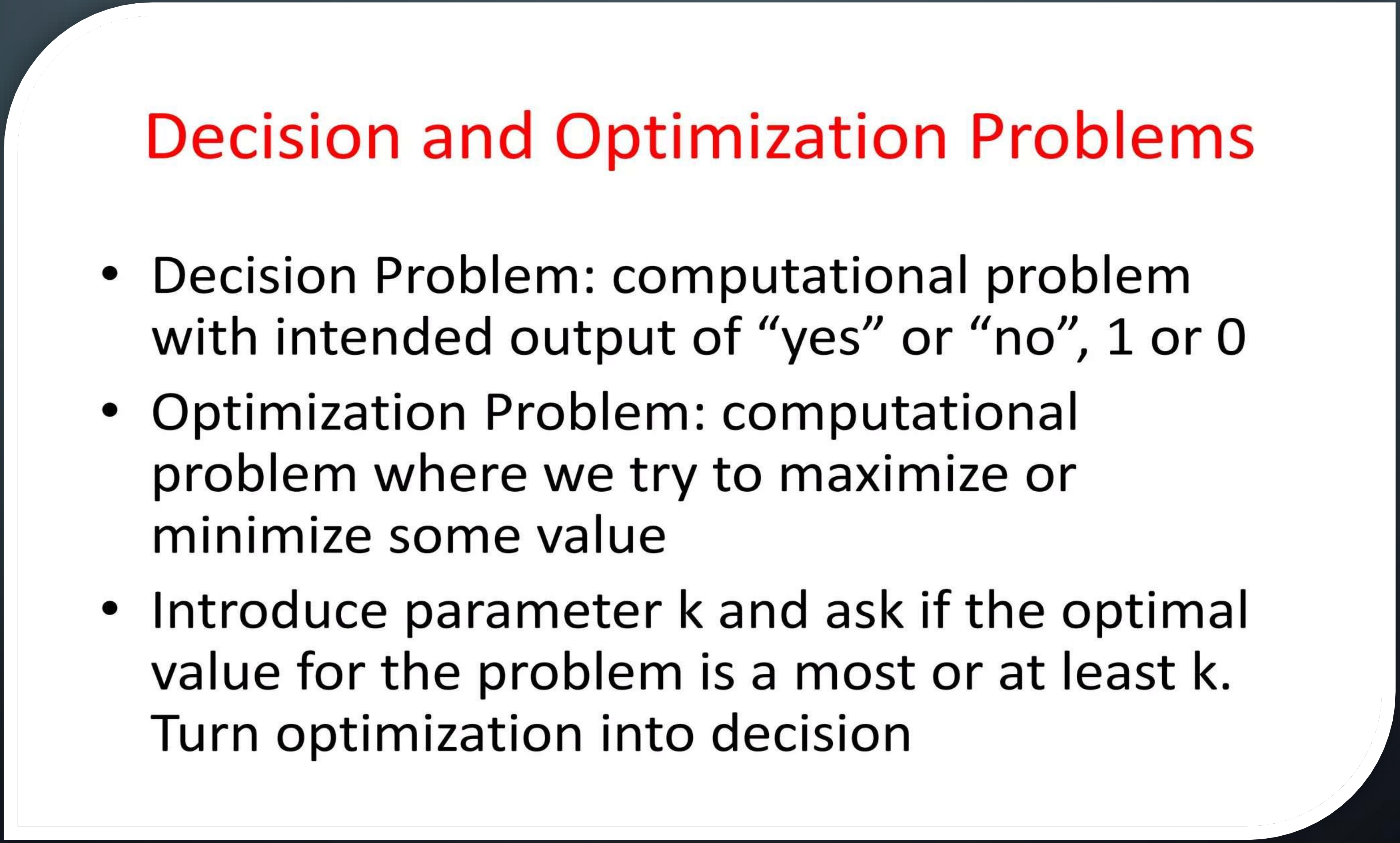
SELF STUDY

PRACTICE CODING

Week 16

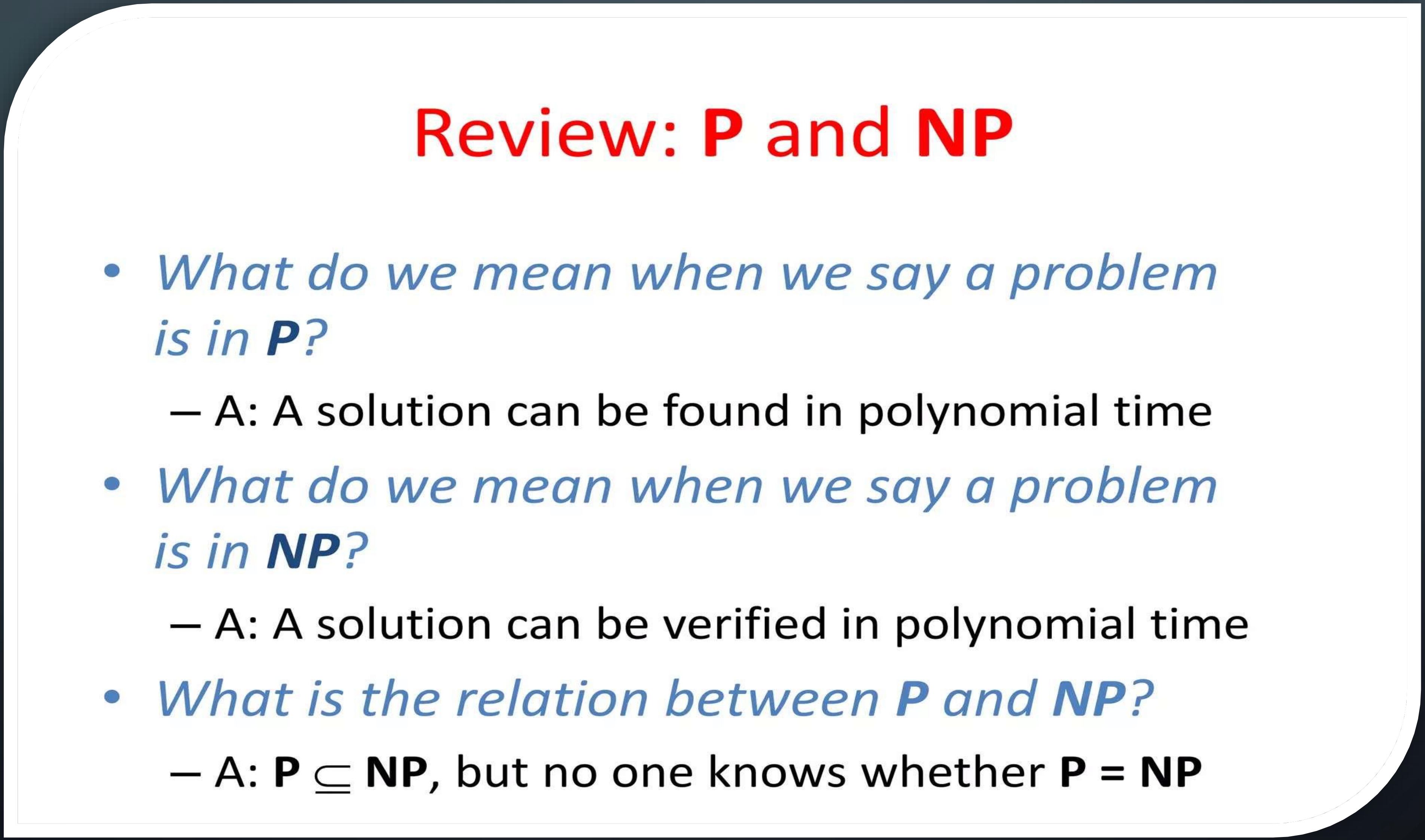
NP-Hard Problem

Page
226-241



Decision and Optimization Problems

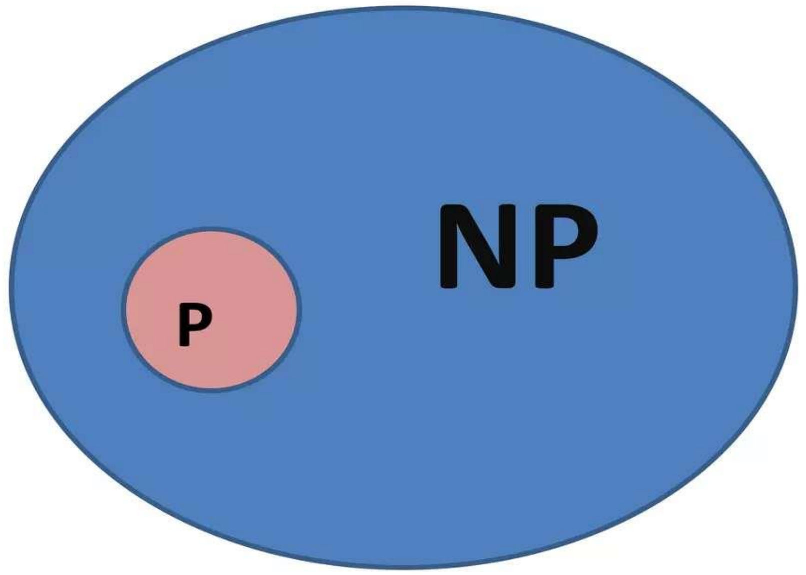
- Decision Problem: computational problem with intended output of “yes” or “no”, 1 or 0
- Optimization Problem: computational problem where we try to maximize or minimize some value
- Introduce parameter k and ask if the optimal value for the problem is a most or at least k. Turn optimization into decision



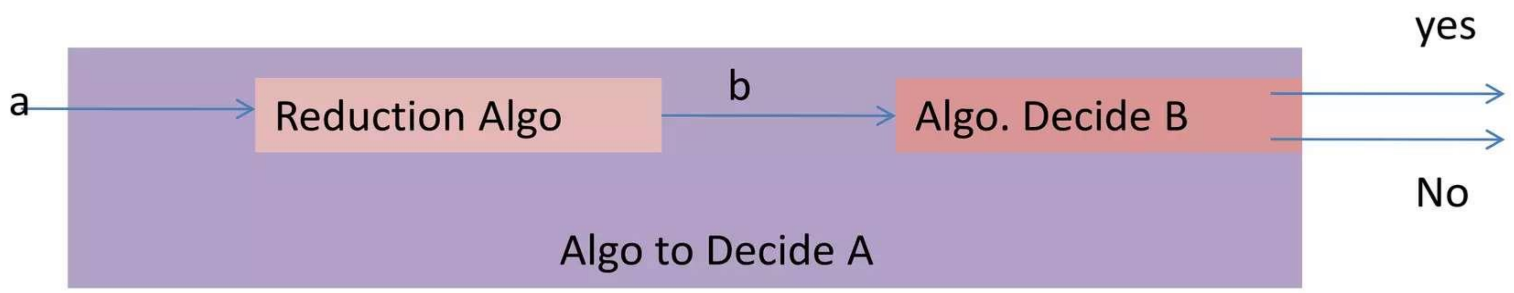
Review: **P** and **NP**

- *What do we mean when we say a problem is in* ***P****?*
  - A: A solution can be found in polynomial time
- *What do we mean when we say a problem is in* ***NP****?*
  - A: A solution can be verified in polynomial time
- *What is the relation between* ***P*** *and* ***NP****?*
  - A: **P** $\subseteq$ **NP**, but no one knows whether **P = NP**

Commonly Believed Relationship between P and NP



Reductions



Given two sets A and B
– Procedure is called polynomial-time reduction algorithm and it provides us with a way to solve problem A in polynomial time
 -Also known as **Turing reduction**
- Given an instance a of A, use a polynomial-time reduction algorithm to transform it to an instance b of B
- Run the polynomial-time decision algorithm on instance b of B
 -Use the answer b of a as the answer for b

Satisfiability(SAT)

- I/P : Boolean formula
- O/P : Is formula satisfiable?

SAT ∈ NP

Proof : Assignment to variables

Verifier: Uses these assignments and checks that the formula evaluates to true (T).

COOK's Theorem

- If SAT has an efficient algorithm then so does other problems in NP.

There is an efficient algo. for SAT

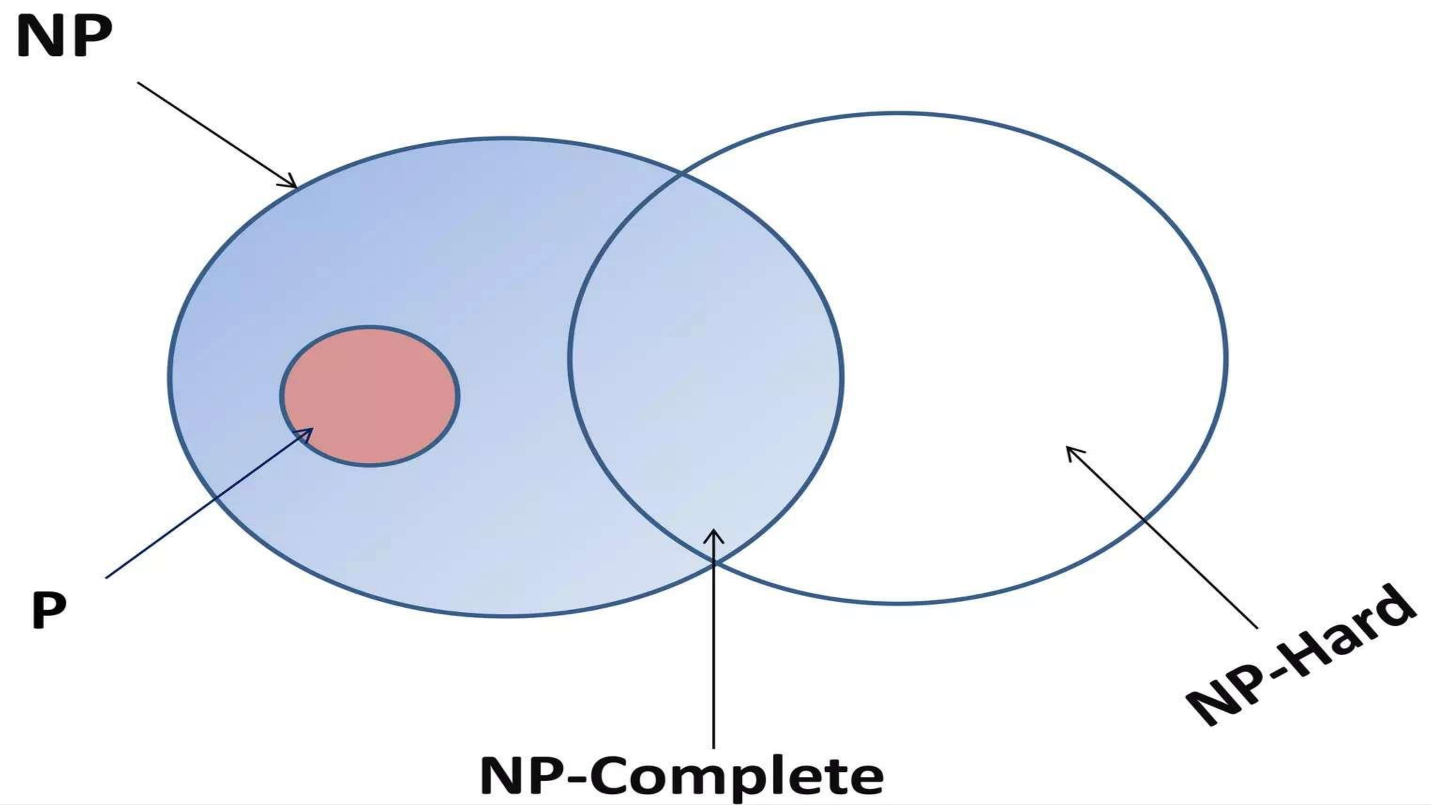
$\downarrow$

There is an efficient algo for all problems in NP.

NP Hard

- A problem A is **NP-hard** if and only if satisfiability reduces to A (satisfiability α A).
- A problem A is **NP-complete** if and only if A is NP-hard and A Є NP.

Relationship among P,NP,NP-Complete and NP-Hard

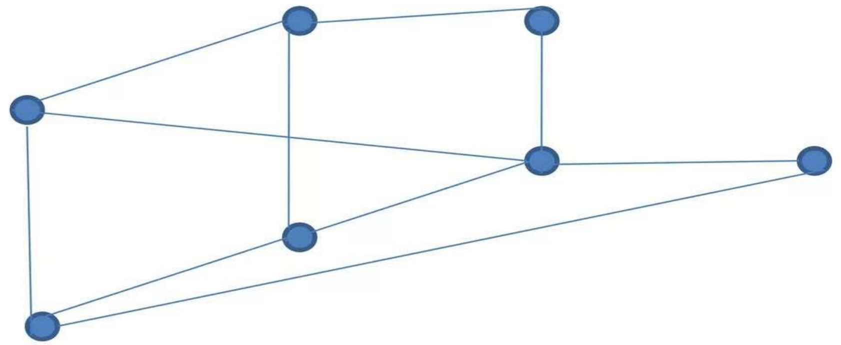


Vertex Cover is NP-Hard

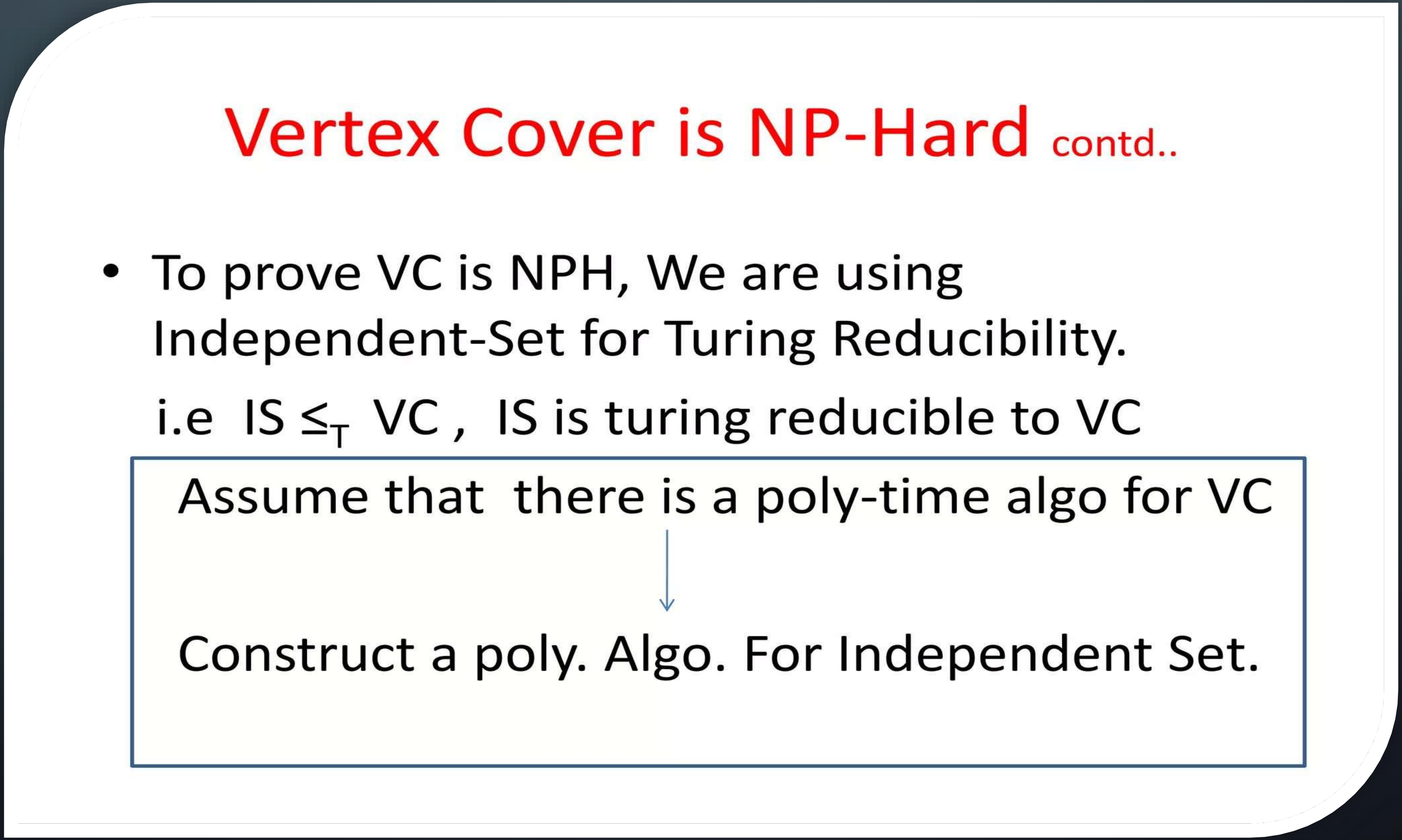
- What is Vertex Cover ?

  Vertex Cover in a graph G is a set of vertices such that every edge has atleast one end point in it.

Ex.



Vertex Cover (Search) converted to VC(decision)



Vertex Cover is NP-Hard contd..

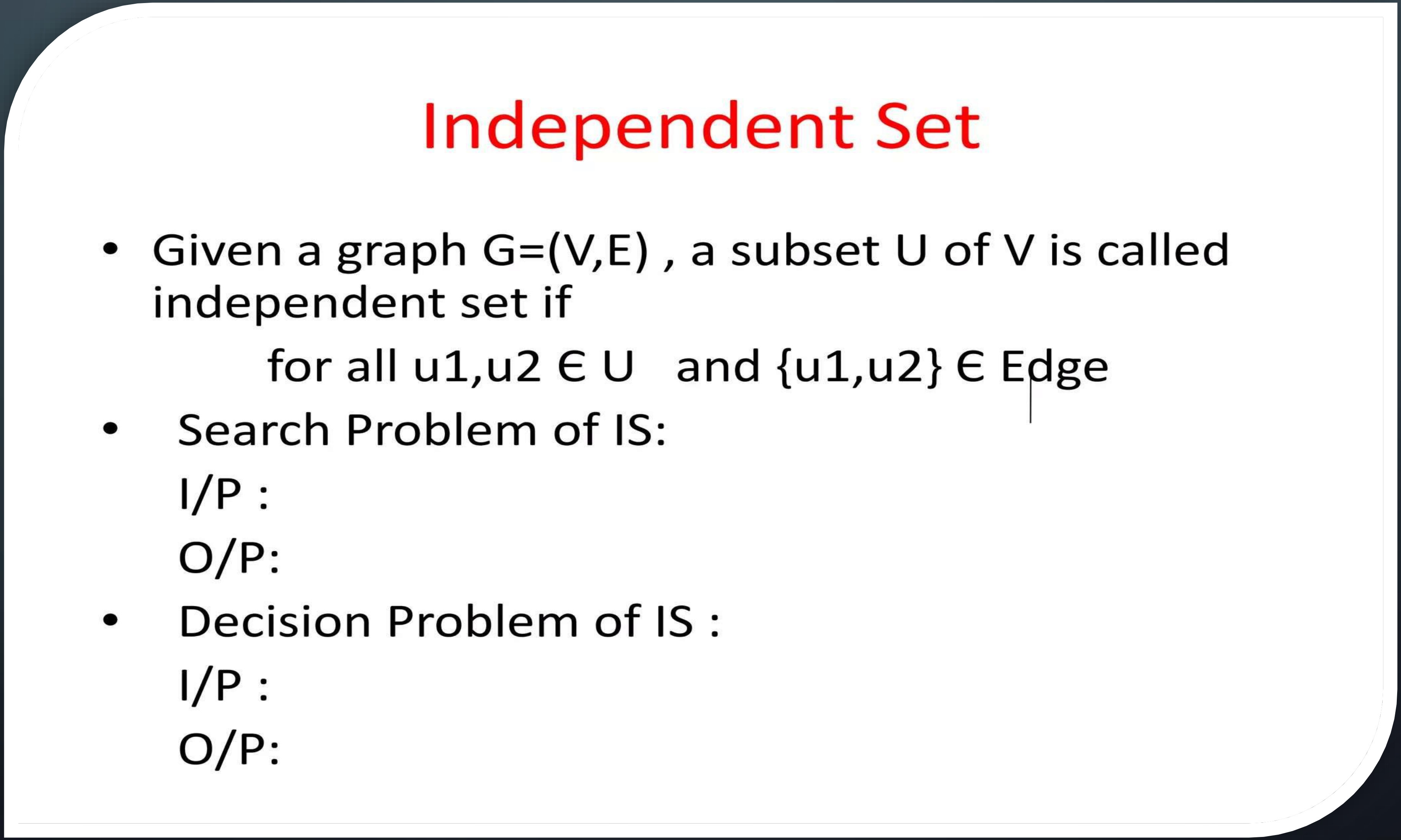
- To prove VC is NPH, We are using Independent-Set for Turing Reducibility.

  i.e IS $\leq_T$ VC , IS is turing reducible to VC

Assume that there is a poly-time algo for VC

$\downarrow$

Construct a poly. Algo. For Independent Set.



Independent Set

- Given a graph G=(V,E) , a subset U of V is called independent set if

  for all u1,u2 Є U and {u1,u2} Є Edge

- Search Problem of IS:

  I/P :

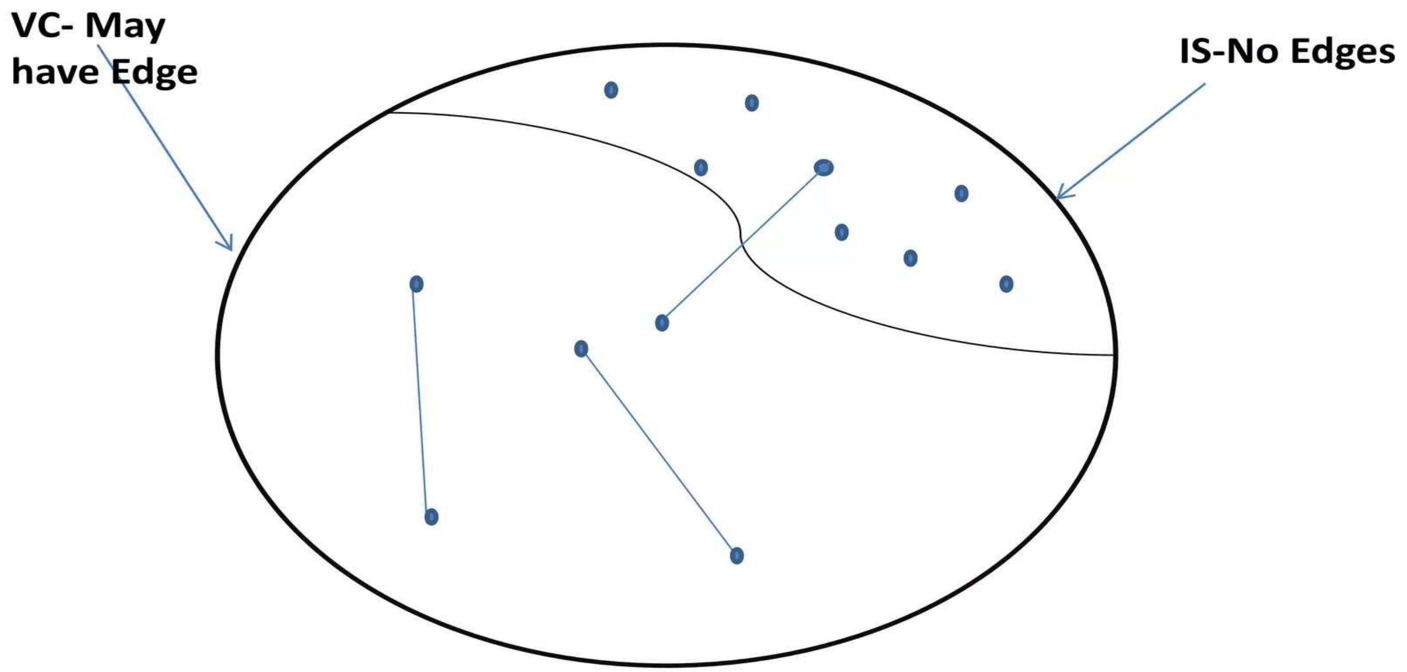
  O/P:

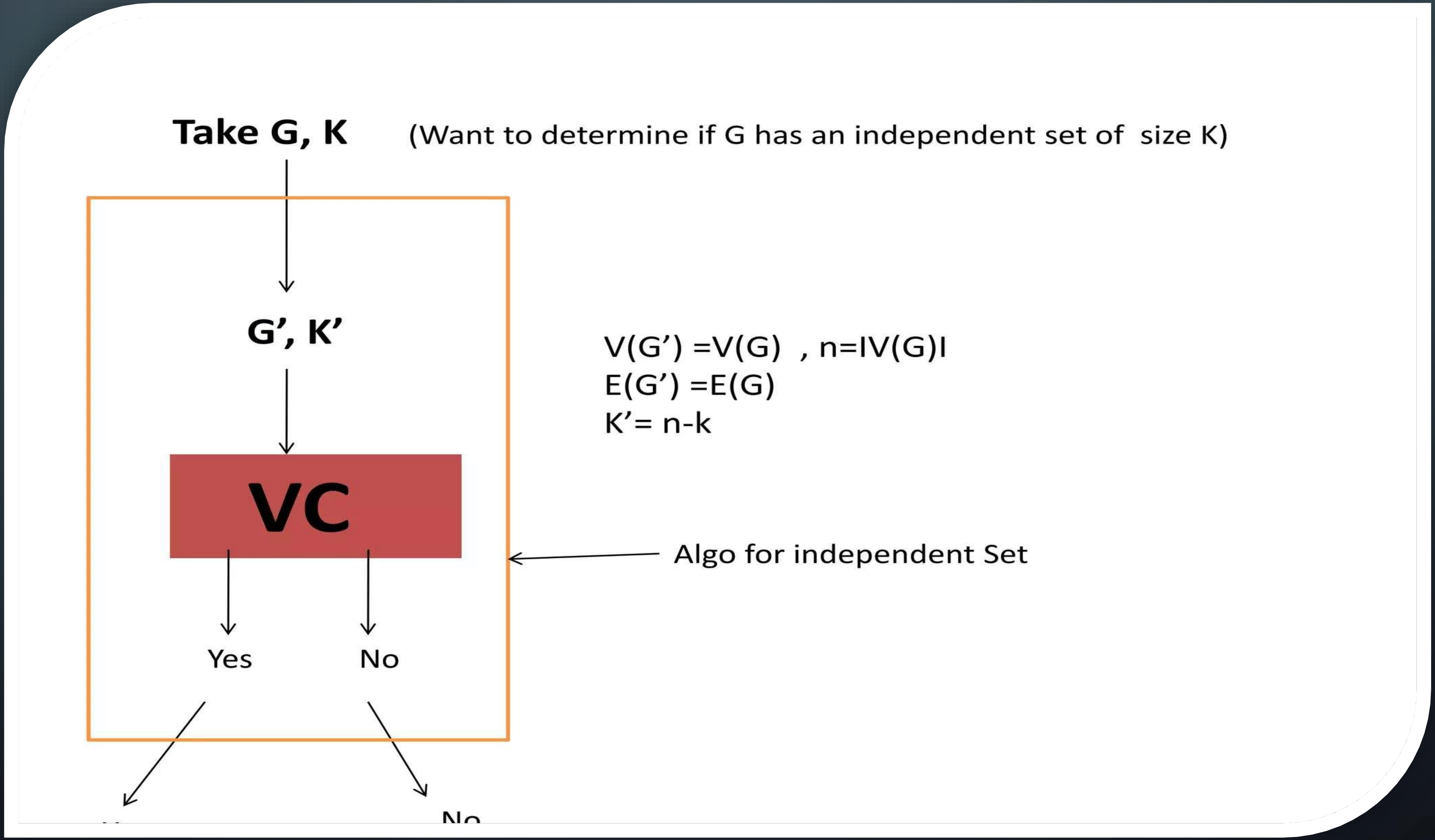
- Decision Problem of IS :

  I/P :

  O/P:

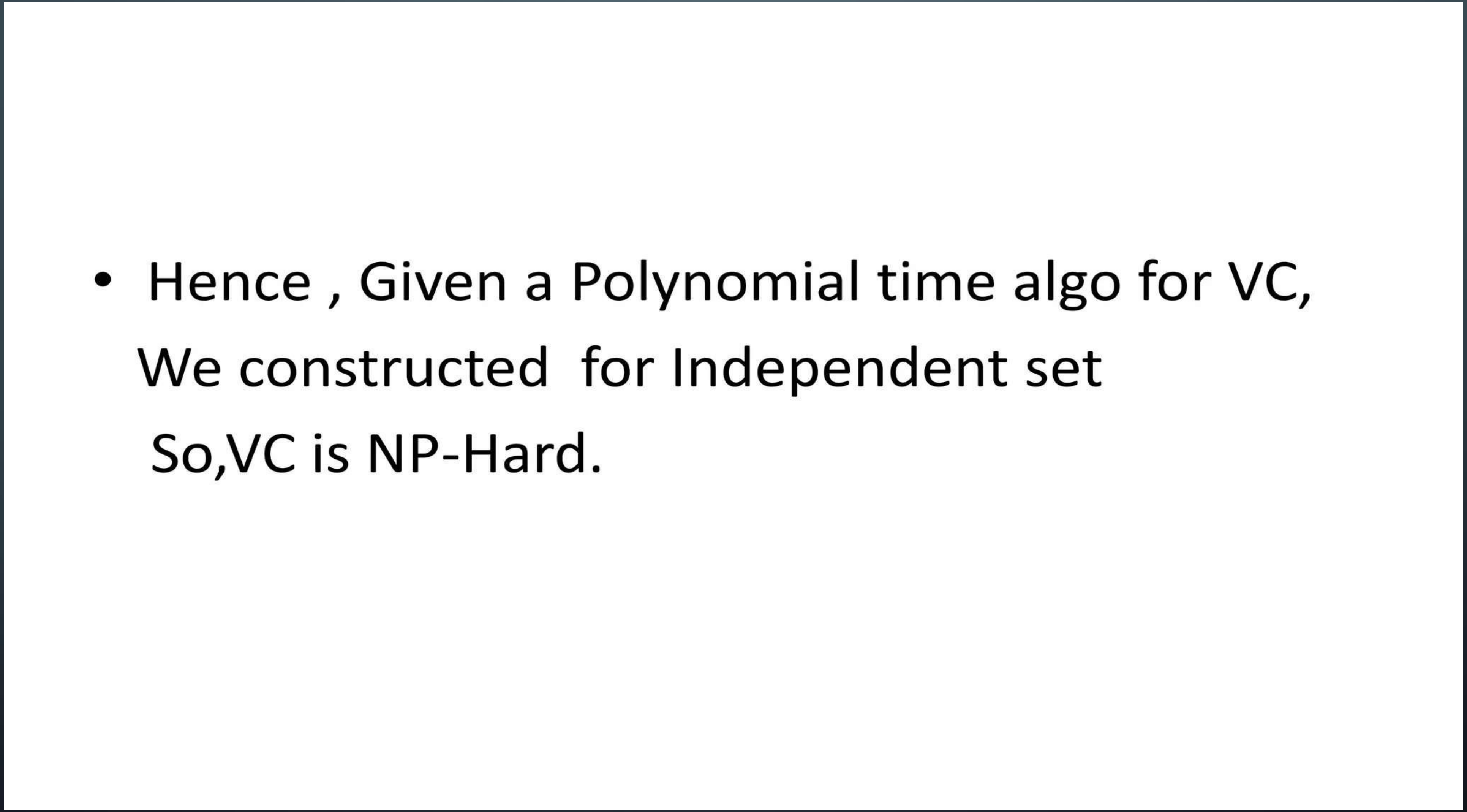
Relation between VC and IS





Take G, K (Want to determine if G has an independent set of size K)

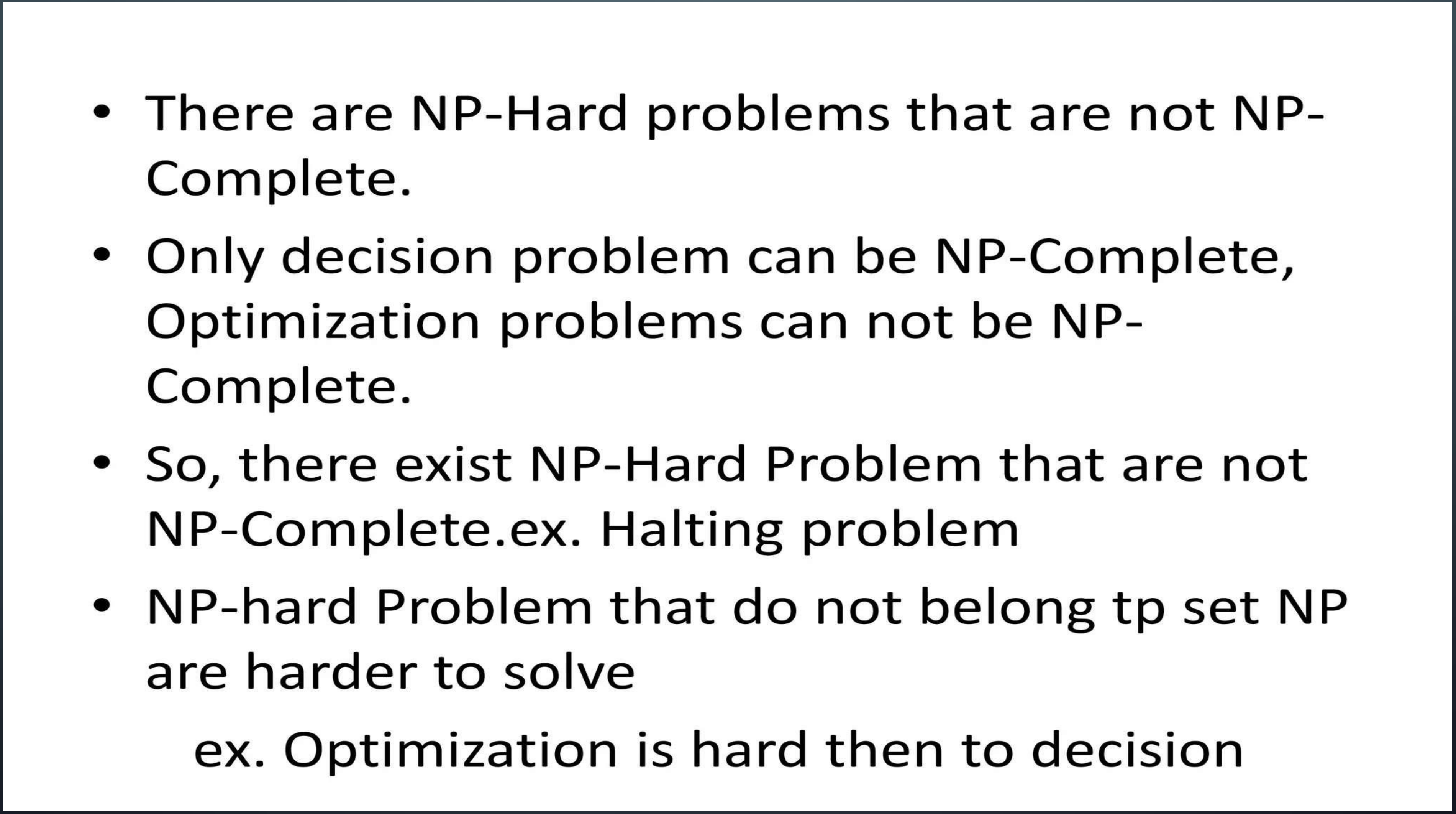
V(G') =V(G) , n=IV(G)I
E(G') =E(G)
K'= n-k



- Hence , Given a Polynomial time algo for VC,

  We constructed for Independent set

  So,VC is NP-Hard.



- There are NP-Hard problems that are not NP-Complete.
- Only decision problem can be NP-Complete, Optimization problems can not be NP-Complete.
- So, there exist NP-Hard Problem that are not NP-Complete.ex. Halting problem
- NP-hard Problem that do not belong tp set NP are harder to solve

  ex. Optimization is hard then to decision

SELF STUDY

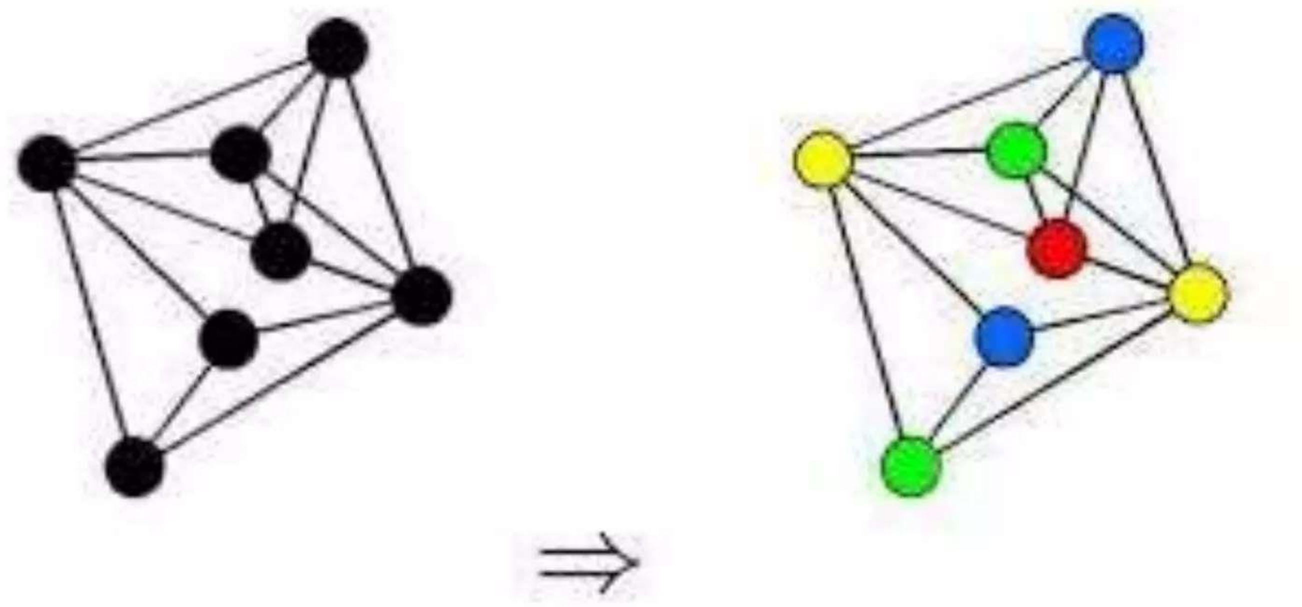
PRACTICE

Week 17

Graph Coloring Problem

Page
243-257

What is graph coloring?



- The objective is to minimize the number of colors while coloring a graph.
- The smallest number of colors required to color a graph G is called its chromatic number of that graph.
- Graph coloring problem is a NP Complete problem.
- No efficient algorithm is available to implement graph coloring mainly we use Greedy and Backtracking algorithm.

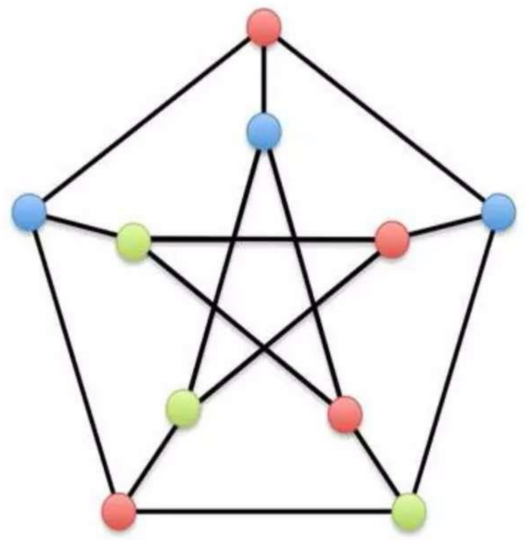
Method to Color a Graph

1. Arrange the vertices of graph in the same order.
2. Choose the first vertex and color it with the first color.
3. Then choose next vertex and color it with the lowest numbered color that has not been colored on any vertices which is adjacent to it.
4. If all the adjacent vertices are colored with the color, assign the new color to it.
5. Repeat the same process until all the vertices are colored.
6. If any complication occurs backtrack before step and repeat the process.

Graph Colouring Backtracking

Goal: Given a graph G and an integer m, find if we can satisfy the problem description using at most m colours.

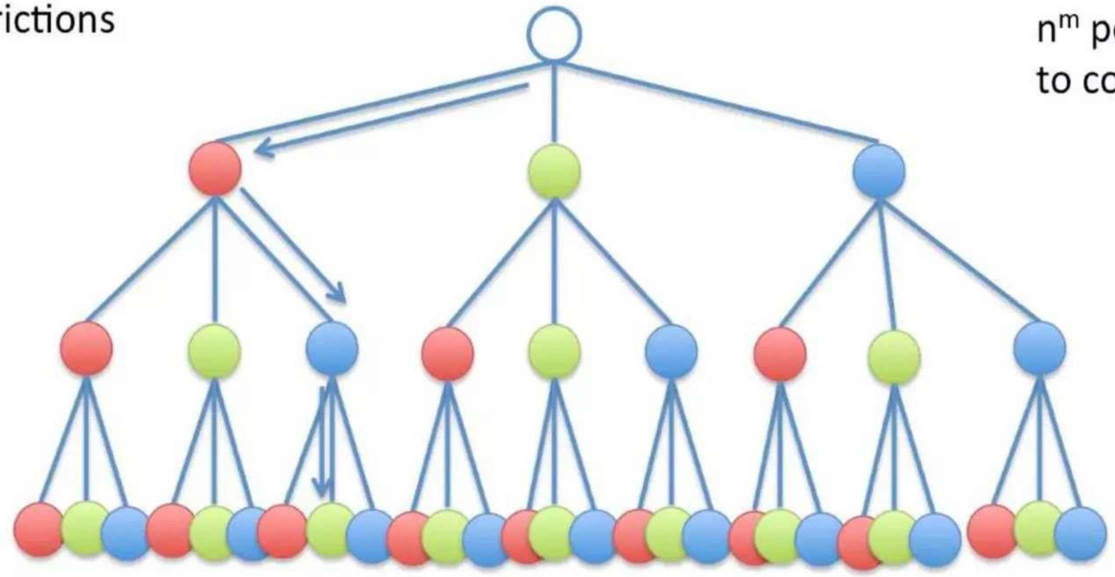
n = 10, m = 3



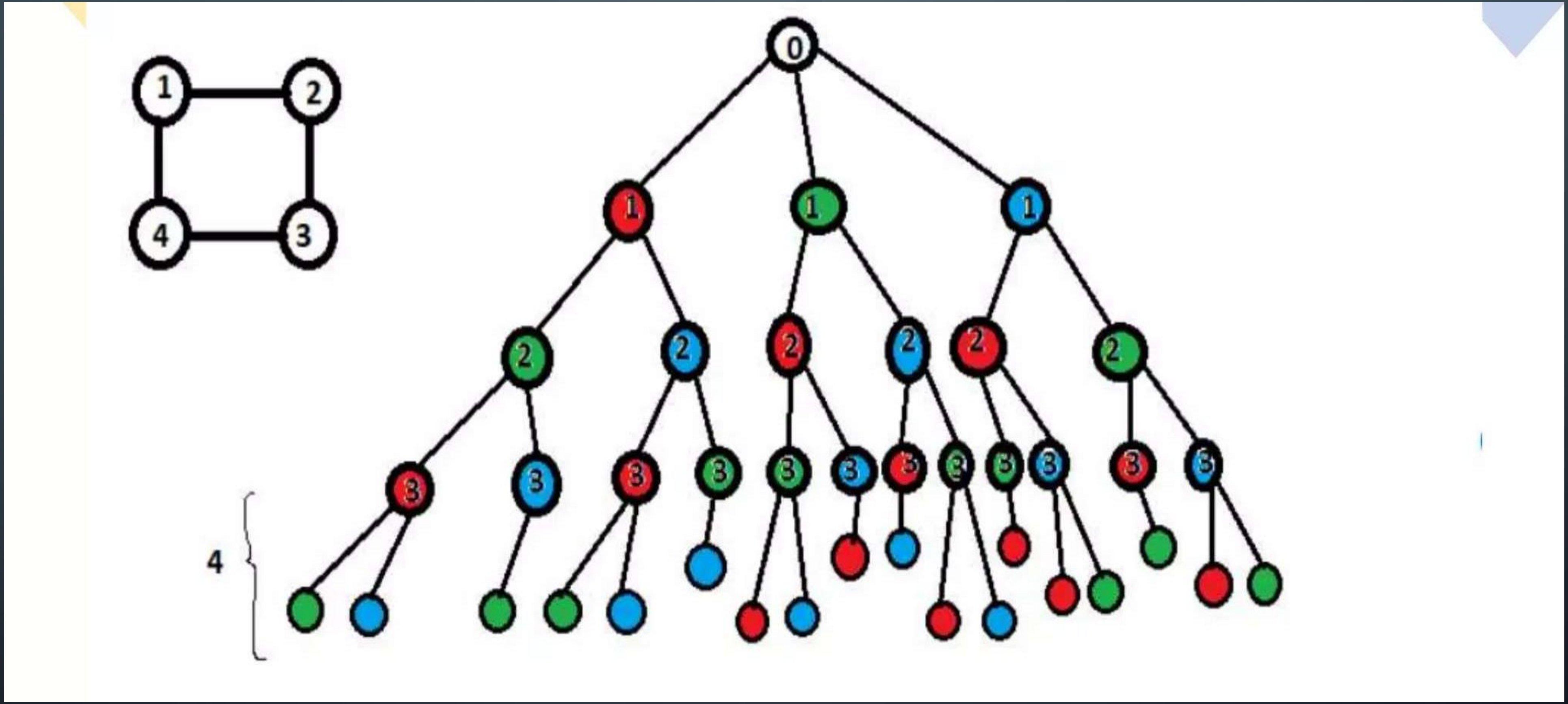
Graph Colouring Backtracking

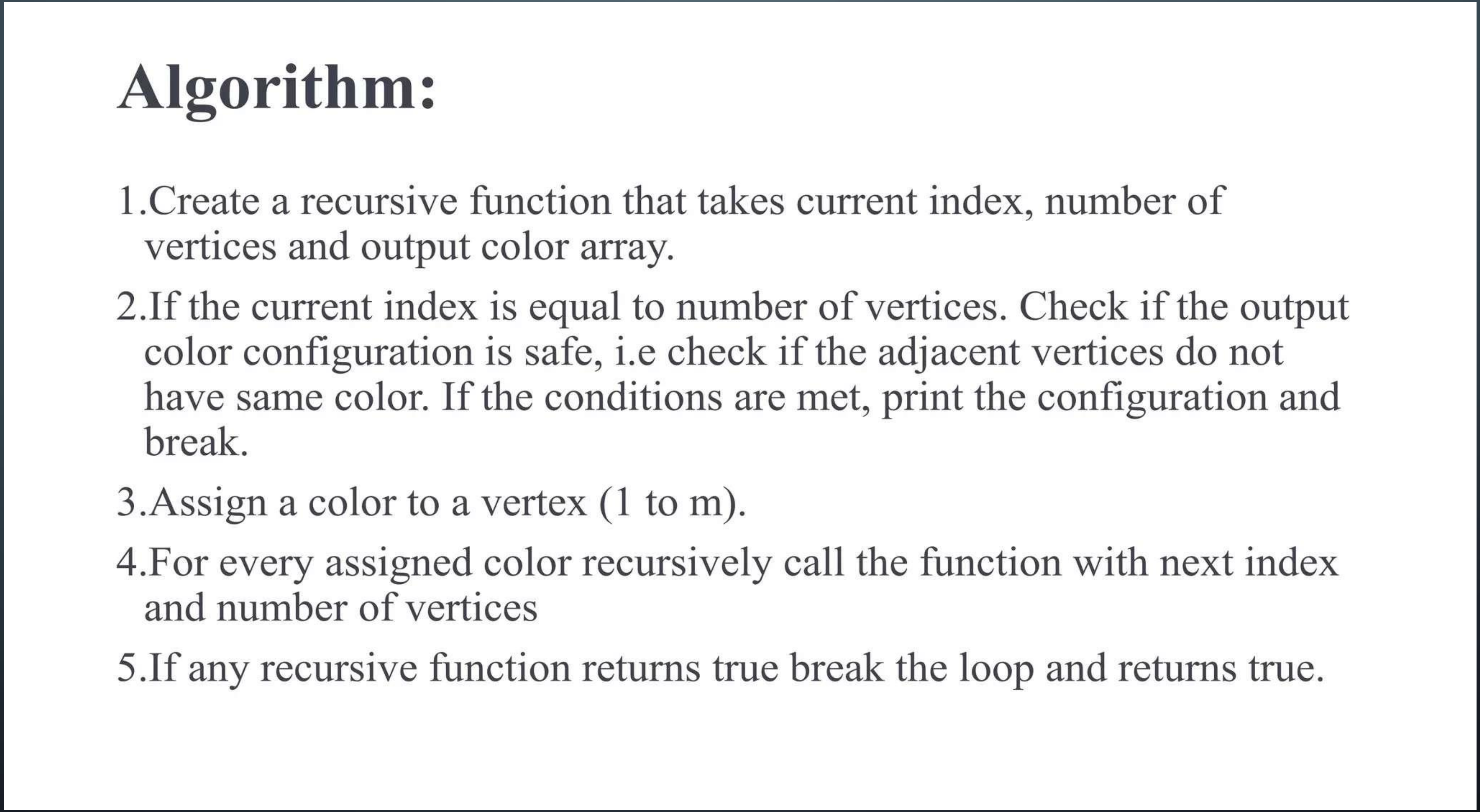
Let's look at a smaller problem: $n = 3, m = 3$

No Restrictions



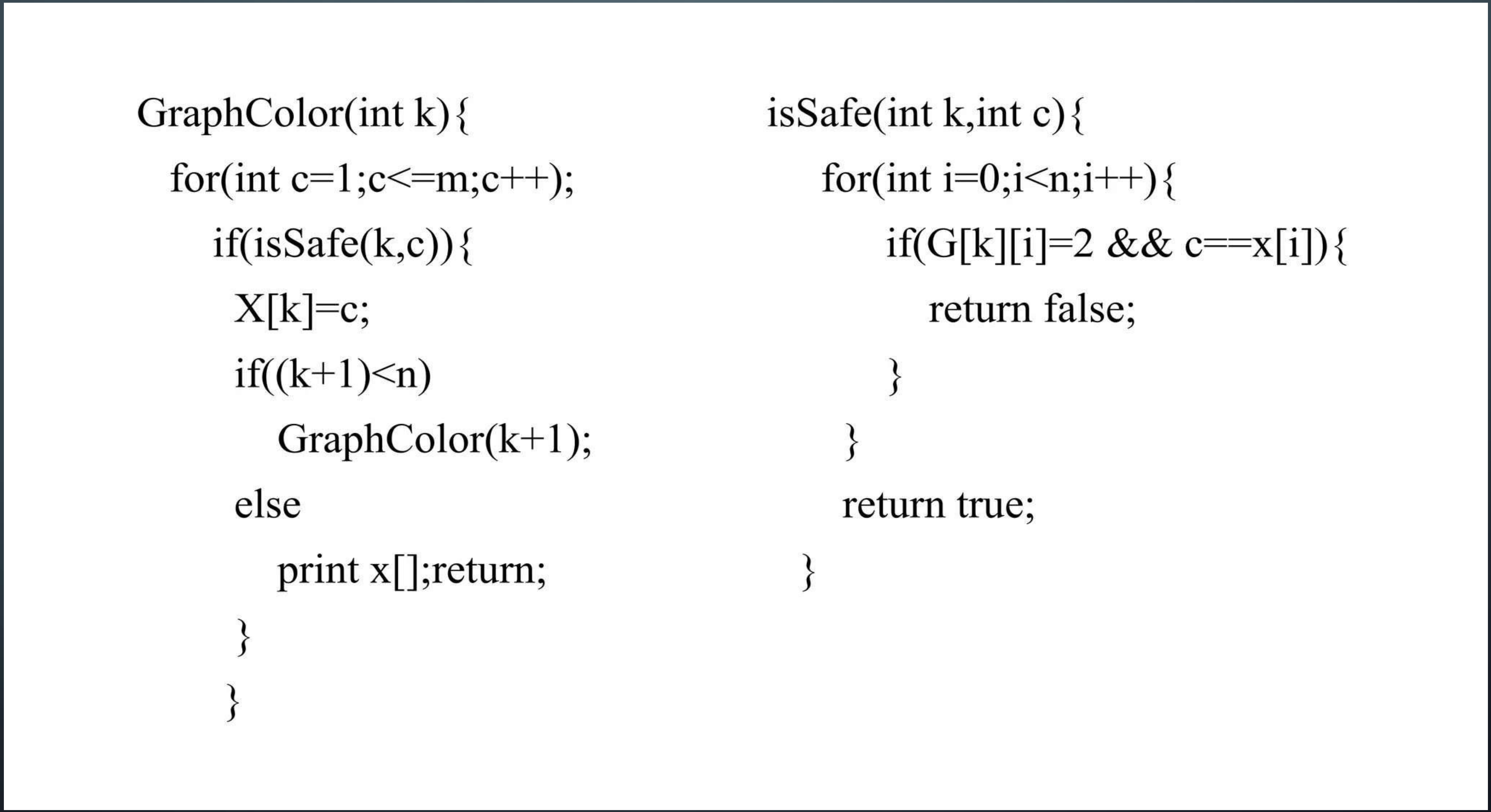
n^m possible ways to colour the 3 nodes


1
2
3
4
0
1
2
3
4



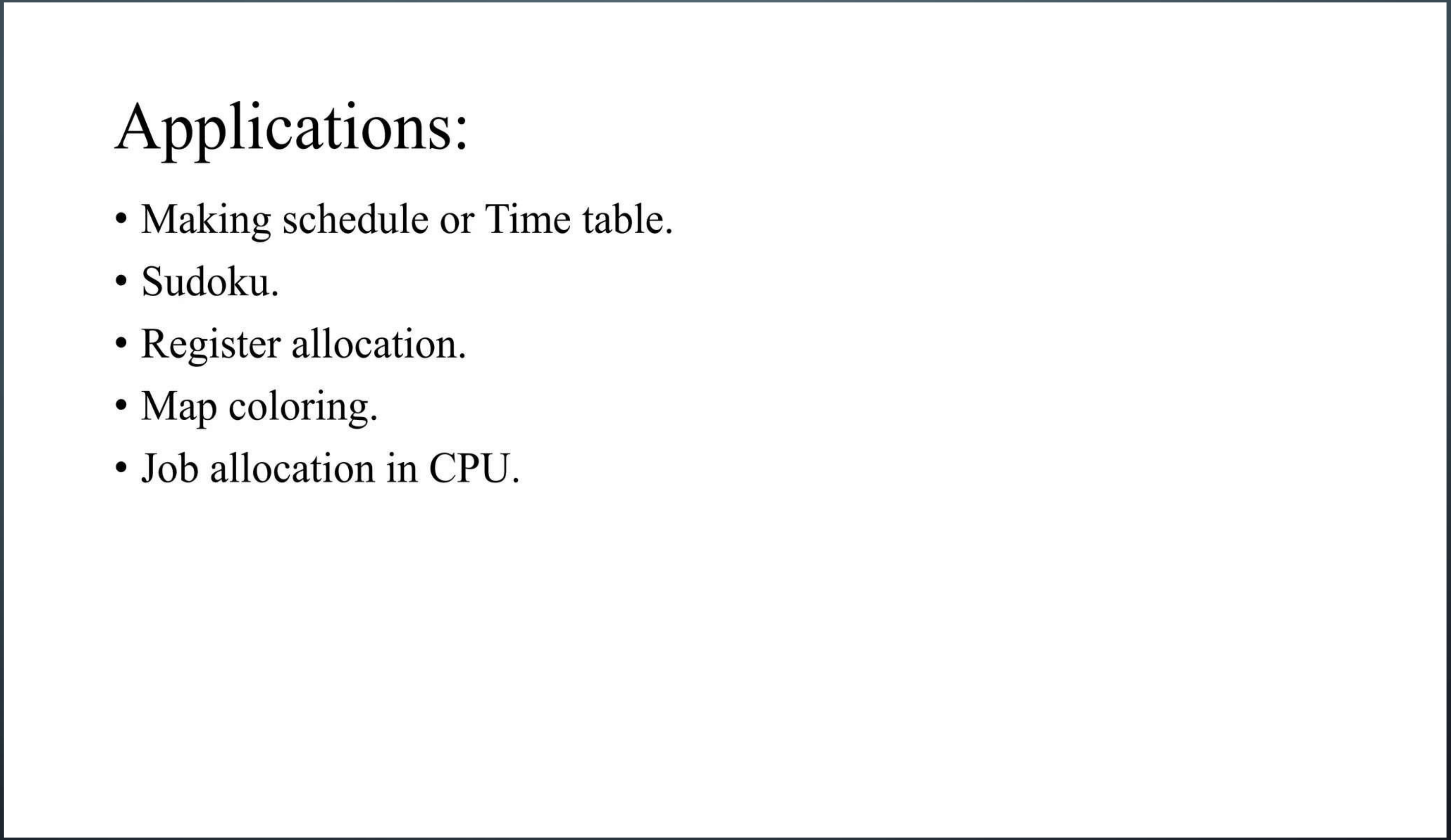
Algorithm:

1. Create a recursive function that takes current index, number of vertices and output color array.
2. If the current index is equal to number of vertices. Check if the output color configuration is safe, i.e check if the adjacent vertices do not have same color. If the conditions are met, print the configuration and break.
3. Assign a color to a vertex (1 to m).
4. For every assigned color recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and returns true.



```
GraphColor(int k){
  for(int c=1;c<=m;c++);
     if(isSafe(k,c)){
      X[k]=c;
      if((k+1)<n)
         GraphColor(k+1);
      else
         print x[];return;
      }
     }
```

```
isSafe(int k,int c){
   for(int i=0;i<n;i++){
       if(G[k][i]=2 && c==x[i]){
          return false;
       }
    }
    return true;
  }
```



Applications:

- Making schedule or Time table.
- Sudoku.
- Register allocation.
- Map coloring.
- Job allocation in CPU.

SELF STUDY

PRACTICE

Thank You