

Pupils Plan S25

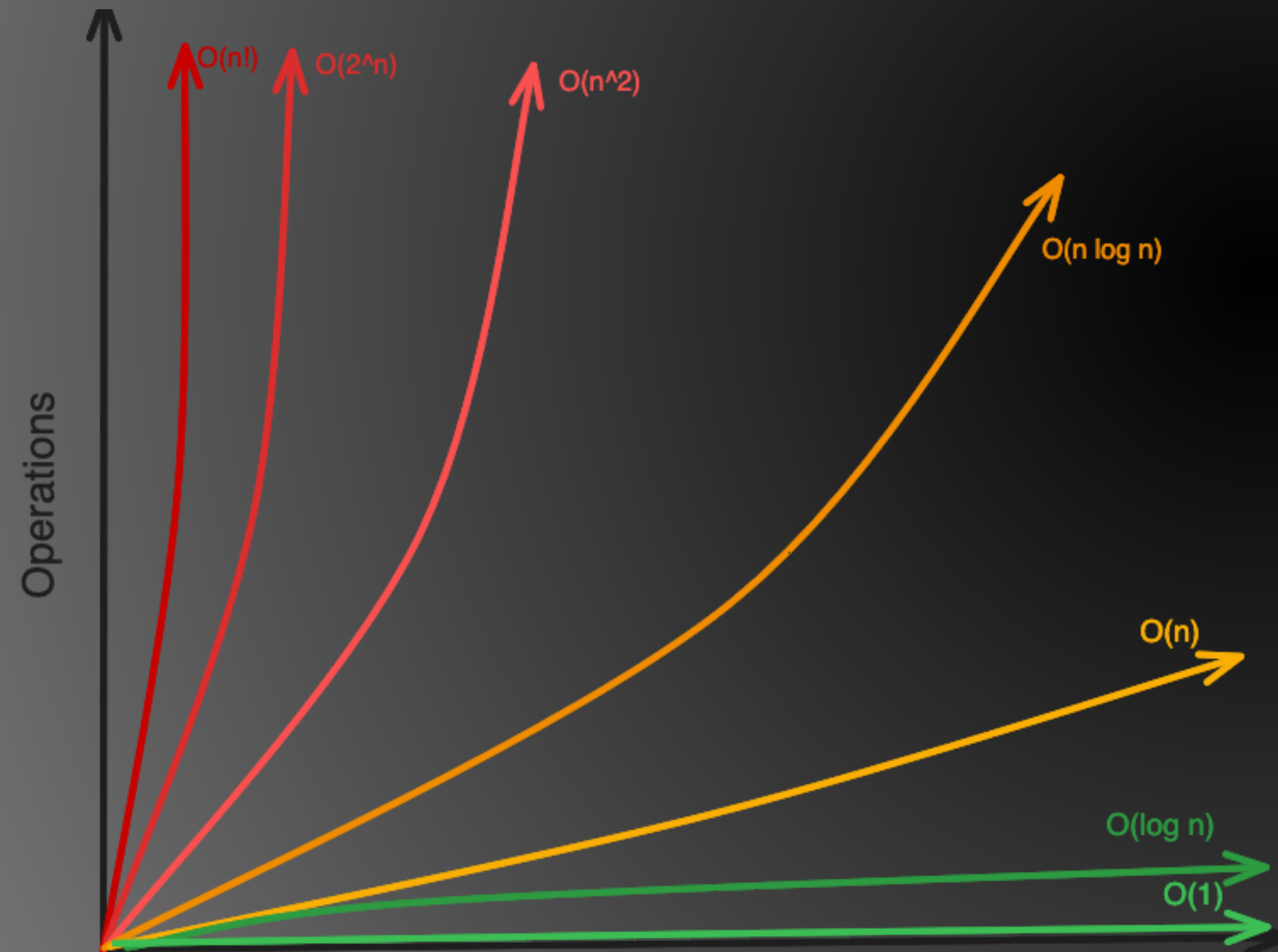
ACM GUCCPC
GUC COLLEGIATE
PROGRAMMING CONTEST

Data Structures for Competitive Programming: Efficiency,
Applications & Best Practices

2

📌 What is Time Complexity?

- ◆ Time Complexity measures how the runtime of an algorithm grows as the input size (N) increases.
It does not have to be N only !



Complexity	Performance
$O(1)$ – Constant Time	🚀 Fastest – Doesn't depend on N
$O(\log N)$ – Logarithmic Time	⚡ Very Fast – Cuts problem size each step
$O(N)$ – Linear Time	✅ Scales with input
$O(N \log N)$ – Log-Linear Time	🔥 Good for sorting
$O(N^2)$ – Quadratic Time	⚠️ Bad for large N
$O(2^n)$ – Exponential Time	❌ Very slow!

Why this matters in Competitive Programming?



Operations Per Second	C++	Java
Optimal Case	$\sim 2 \times 10^8$	$\sim 10^8$
Realistic Case	$\sim 10^8$	$\sim 10^7 - 5 \times 10^7$
Worst Case (Heavy Memory Usage, I/O Overhead, etc.)	$\sim 10^7$	$\sim 10^6$

1 O(1) – Constant Time

✓ **Definition:** The execution time does **not** depend on the input size.

 **Java Code Example:**

java

 Copy  Edit

```
int[] arr = {10, 20, 30, 40};  
System.out.println(arr[2]); // Always O(1), direct access
```

✓ **Common O(1) Operations:**

- Array indexing: `arr[i]`
- HashMap/HashSet lookup
- Stack/Queue `push()` and `pop()`

**2**

$O(\log N)$ – Logarithmic Time

✓ **Definition:** The execution time **decreases significantly** as N grows.

📌 **Java Code Example (Binary Search):**

java

Copy Ed

```
int binarySearch(int[] arr, int target) {  
    int left = 0, right = arr.length - 1;  
    while (left <= right) {  
        int mid = left + (right - left) / 2;  
        if (arr[mid] == target) return mid;  
        else if (arr[mid] < target) left = mid + 1;  
        else right = mid - 1;  
    }  
    return -1;  
}
```

✓ **Common $O(\log N)$ Operations:**

- Binary Search

3 **O(N) – Linear Time**

✓ **Definition:** Execution time grows **proportionally** with N.

✂ **Java Code Example (Find Maximum in Array):**

java

 Copy  Edit

```
int findMax(int[] arr) {  
    int max = arr[0];  
    for (int num : arr) {  
        if (num > max) max = num;  
    }  
    return max;  
}
```

✓ **Common O(N) Operations:**

- Looping through an array or list
- Finding max/min in an unsorted array
- Linear search

5 $O(N^2)$ – Quadratic Time

✓ **Definition:** Execution time grows exponentially with N^2 .

✂ **Java Code Example (Bubble Sort):**

java

 Copy  Edit

```
void bubbleSort(int[] arr) {  
    int n = arr.length;  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                int temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```



$O(2^n)$ – Exponential Time

✓ Definition: Execution time doubles with every additional input.

📌 Java Code Example (Recursive Fibonacci):

java

📄 Copy ✎ Edit

```
int fibonacci(int n) {  
    if (n <= 1) return n;  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}
```

✓ Common $O(2^n)$ Operations:

- Recursive Fibonacci


```
int search(int[] arr, int target) {  
    for (int i = 0; i < arr.length; i++) {  
        if (arr[i] == target) return i;  
    }  
    return -1;  
}
```

BEST :
WORST :

```
void checkPairs(int[] arr) {  
    for (int i = 0; i < arr.length; i++) {  
        for (int j = 0; j < arr.length; j++) {  
            System.out.println(arr[i] + ", " + arr[j]);  
        }  
    }  
}
```

BEST :
WORST :

```
int binarySearch(int[] arr, int target) {  
    int left = 0, right = arr.length - 1;  
    while (left <= right) {  
        int mid = left + (right - left) / 2;  
        if (arr[mid] == target) return mid;  
        else if (arr[mid] < target) left = mid + 1;  
        else right = mid - 1;  
    }  
    return -1;  
}
```

BEST :
WORST :

General rule of thumb

Max N Given in Problem	Allowed Complexity
$N \leq 10$	$O(N!)$ or $O(2^n)$ (Brute force is fine)
$N \leq 100$	$O(N^2)$ or $O(N \log N)$
$N \leq 10^4$	$O(N \log N)$ or $O(N)$
$N \leq 10^6$	$O(N)$ or $O(N \log N)$
$N \leq 10^9$	$O(\log N)$ or $O(1)$

1 Vectors & ArrayLists

Definition:

Both are dynamic arrays that can resize automatically.
Unlike fixed-size arrays, they grow/shrink as needed.
Stored in contiguous memory, allowing fast $O(1)$ random access.

Why Use Them Instead of Regular Arrays?



- ✓ Dynamic resizing (`push_back()` in C++, `.add()` in Java)
- ✓ Better memory management (No need to manually allocate memory)
- ✓ Supports STL & Java Collections Framework operations

```
vector<int> v = {10, 20, 30}; // C++
```

```
ArrayList<Integer> arr = new ArrayList<>(List.of(10, 20, 30)); // Java
```

Basic Syntax for Java (ArrayList<T>)

java



 Copy  Edit

```
import java.util.ArrayList;
import java.util.Arrays;

ArrayList<Integer> arr1 = new ArrayList<>(); // Empty ArrayList
ArrayList<Integer> arr2 = new ArrayList<>(5); // Initial capacity of 5
ArrayList<Integer> arr3 = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5)); // Direct initialization
ArrayList<Integer> arr4 = new ArrayList<>(arr3); // Copy another ArrayList
```

Basic Syntax for C++ (vector<T>)

cpp

 Copy  Edit

```
#include <vector>
using namespace std;

vector<int> v1; // Empty vector
vector<int> v2(5); // Vector of size 5 (default initialized to 0)
vector<int> v3(5, -1); // Vector of size 5, initialized to -1
vector<int> v4 = {1, 2, 3, 4, 5}; // Direct initialization
vector<int> v5(v4); // Copy another vector
```

Common Operations & Their Time Complexities

Operation	C++ (<code>vector</code>)	Java (<code>ArrayList</code>)	Time Complexity
Add Element at End	<code>v.push_back(x);</code>	<code>arr.add(x);</code>	O(1) amortized
Remove Last Element	<code>v.pop_back();</code>	<code>arr.remove(arr.size() - 1);</code>	O(1)
Access Element	<code>v[i]</code> or <code>v.at(i);</code>	<code>arr.get(i);</code>	O(1)
Get Size of Vector	<code>v.size();</code>	<code>arr.size();</code>	O(1)
Clear All Elements	<code>v.clear();</code>	<code>arr.clear();</code>	O(N)
Check if Empty	<code>v.empty();</code>	<code>arr.isEmpty();</code>	O(1)
Sort Elements	<code>sort(v.begin(), v.end());</code>	<code>Collections.sort(arr);</code>	O(N log N)

📌 When to Use & When to Avoid Vectors & ArrayLists in CP 🚀

✅ Use Vectors & ArrayLists When: ✓ You need dynamic resizing (unknown input size).

✓ Frequent insertions/deletions happen at the end ($O(1)$ amortized).

✓ Random access ($O(1)$) is required.

✓ Sorting and searching are needed (`sort()` in C++, `Collections.sort()` in Java).

❌ Avoid Vectors & ArrayLists When: ✓ Insertions/deletions occur at the beginning or middle ($O(N)$ shifting cost).

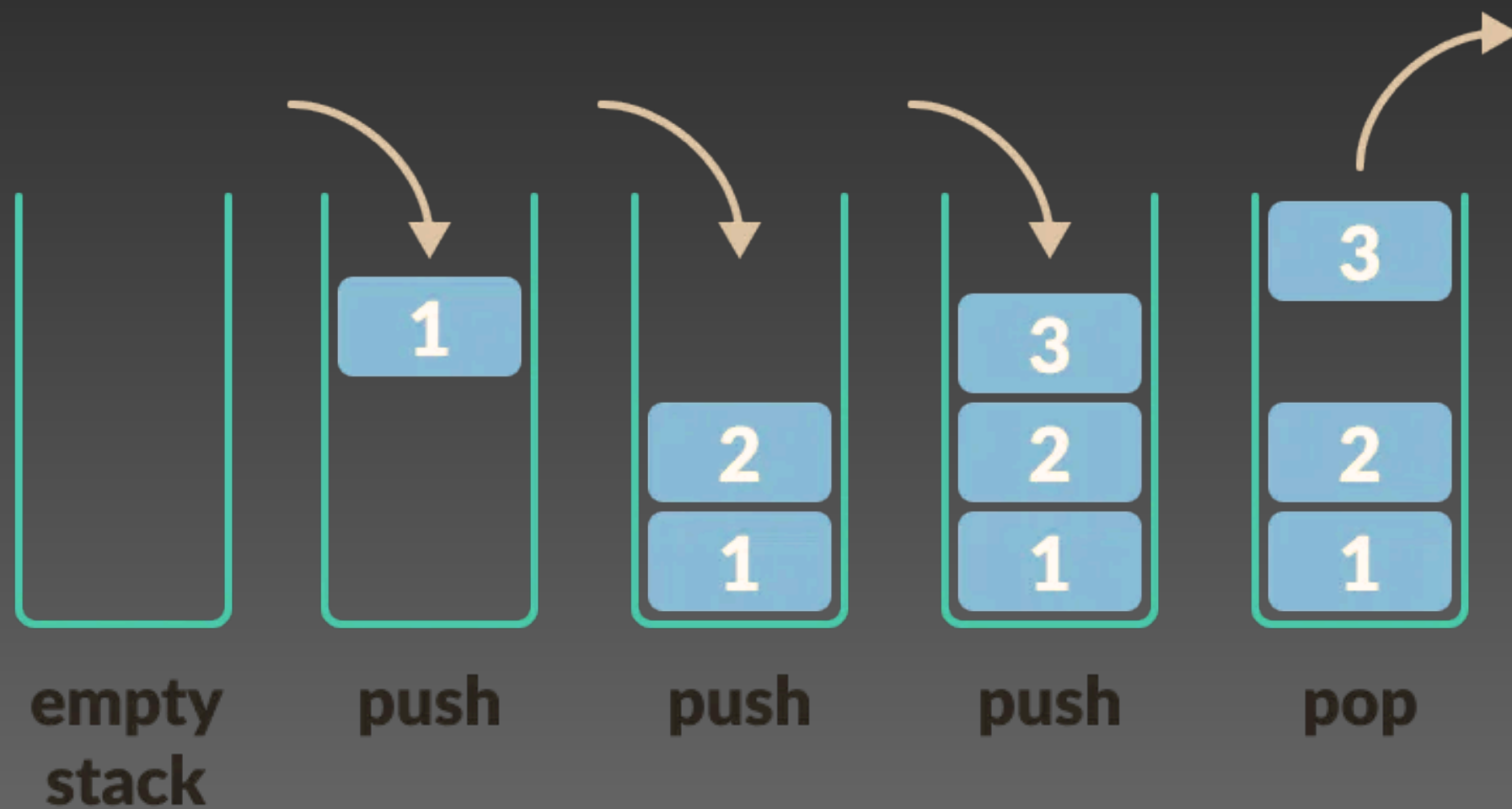
✓ You need frequent insertions/removals from both ends → Use `deque` instead.

✓ Fixed-size data is needed → Use arrays instead (less memory overhead).

2 Stack

Definition:



A Stack is a LIFO (Last In, First Out) data structure where elements are added and removed from the top.



Initialization

Java (`Stack<T>`)



java

 Copy  Edit

```
import java.util.Stack;  
Stack<Integer> stack = new Stack<>();
```

C++ (`stack<T>`)

cpp

 Copy  Edit

```
#include <stack>  
stack<int> stk;
```

Common Operations

Operation	Java (<code>Stack<T></code>)	C++ (<code>stack<T></code>)	Time Complexity
Push (Insert at Top)	<code>stack.push(x);</code>	<code>stk.push(x);</code>	O(1)
Pop (Remove from Top)	<code>stack.pop();</code>	<code>stk.pop();</code>	O(1)
Peek (View Top Element)	<code>stack.peek();</code>	<code>stk.top();</code>	O(1)
Check if Empty	<code>stack.isEmpty();</code>	<code>stk.empty();</code>	O(1)
Size of Stack	<code>stack.size();</code>	<code>stk.size();</code>	O(1)

When to Use & When to Avoid Stacks in CP

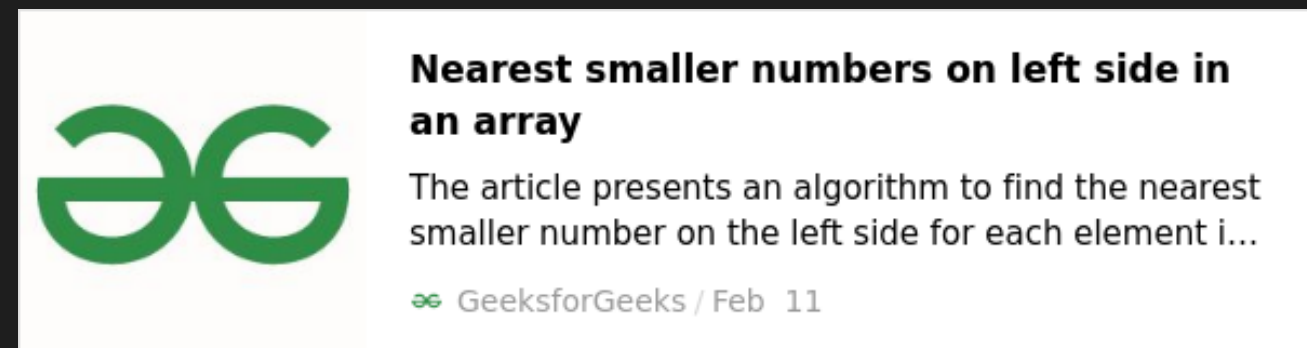
- ✓ Use a Stack When: ✓ LIFO behavior is required (last in, first out).
- ✓ Handling recursion manually (e.g., DFS iterative approach).
- ✓ Solving Next Greater Element, Stock Span, or Balanced Parentheses problems.
- ✓ Backtracking problems (e.g., Maze Solving, Bracket Matching).
- ✗ Avoid a Stack When: ✓ You need random access to elements (use `vector` or `ArrayList` instead).
- ✓ You need FIFO behavior (use a Queue instead).
- ✓ Insertion or deletion must happen from both ends (use a Deque instead).

Optimized Stack for Order-Based Problems



- A Monotonic Stack is a specialized stack that maintains elements in increasing or decreasing order.
- Used in Next Greater Element, Histogram problems, and Stock Span problems in Competitive Programming (CP).

- Given an array of N integers, find the closest smaller element to the left for each element.
- If no smaller element exists, return `-1`.

✓ Example Input & Output:



makefile

 Copy  Edit

Input: [4, 3, 2, 7, 5, 8]

Output: [-1, -1, -1, 2, 2, 5]



```

public int[] closestMinLeftBruteForce(int[] nums) {
    int n = nums.length;
    int[] result = new int[n];

    for (int i = 0; i < n; i++) {
        result[i] = -1; // Default value if no smaller element exists
        for (int j = i - 1; j >= 0; j--) {
            if (nums[j] < nums[i]) {
                result[i] = nums[j];
                break; // Closest smaller element found
            }
        }
    }
    return result;
}

```

```

vector<int> closestMinLeftBruteForce(vector<int>& nums) {
    int n = nums.size();
    vector<int> result(n, -1);

    for (int i = 0; i < n; i++) {
        for (int j = i - 1; j >= 0; j--) {
            if (nums[j] < nums[i]) {
                result[i] = nums[j];
                break; // Closest smaller element found
            }
        }
    }
    return result;
}

```

Time Complexity :

Nearest Smaller Problem (My thinking) :

Will loop over the input array until the end :

a- if i find the stack is empty --> then there is not any smaller element before me then -1 is my answer .

b- if i find a larger or equal element at the top of the stack --> i will keep popping until finding a smaller element and it is my answer or the stack will be empty and then -1 will be my answer (why ? Does anyone have a problem until here) .

c- if i find a smaller element at the top of the stack just print it .

d- add the current element to the top of the stack .

Why did we use monotonic stack :

- the point here is we only care about (nearest) smaller element to the left ...if my current number is smaller than the number at the top of the stack then it makes sense that i should pop it(get rid of it) since my current number is smaller and closer to the next elements in the array at the right side that i will encounter later ! (If you do not get , feel free to ask me to repeat in Arabic) .


```

public int[] closestMinLeftStack(int[] nums) {
    int n = nums.length;
    int[] result = new int[n];
    Stack<Integer> stack = new Stack<>();

    for (int i = 0; i < n; i++) {
        while (!stack.isEmpty() && stack.peek() >= nums[i]) {
            stack.pop(); // Remove elements that are greater or equal
        }
        result[i] = stack.isEmpty() ? -1 : stack.peek(); // Closest smaller element
        stack.push(nums[i]); // Push current element onto the stack
    }
    return result;
}

```

Time Complexity :

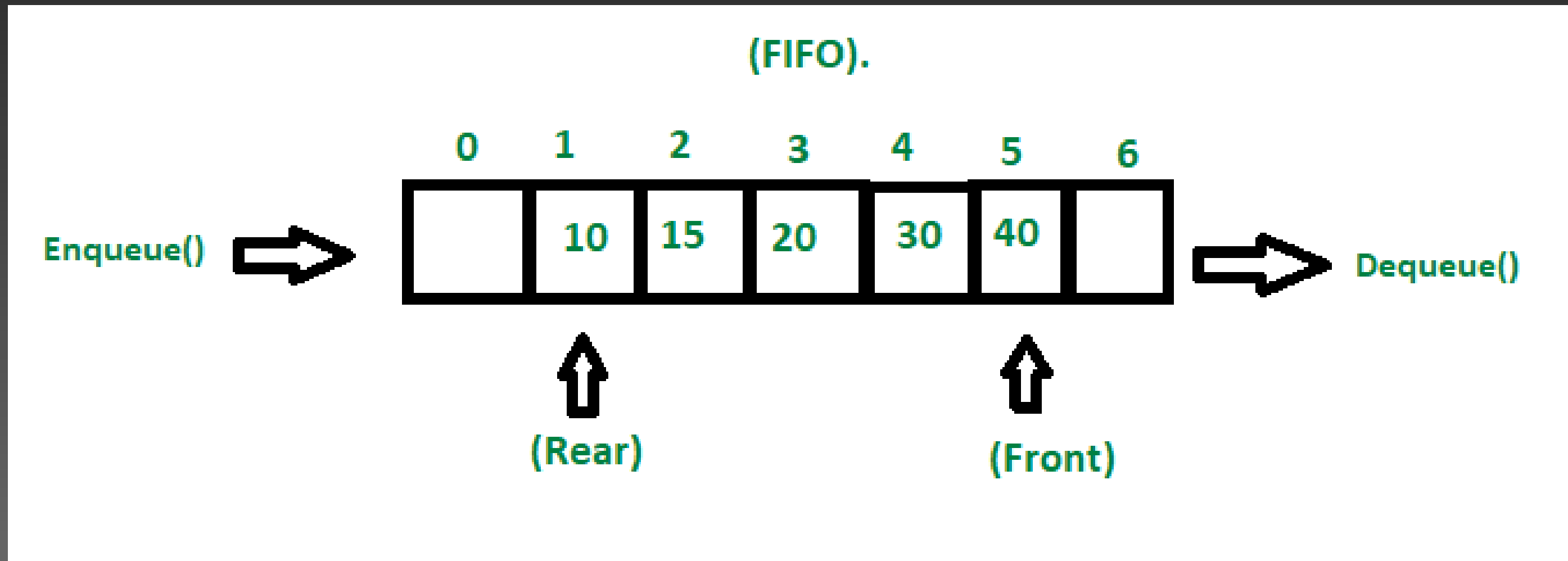
arr = 4 3 2 7 5 8

index	arr[i]	Stack	output[i]
0	4	empty → 4	-1
1	3	4 → empty ↓ 3	-1
2	2	3 → empty ↓ 2	-1
3	7	2 → 7 ↓ 2	2
4	5	7 → 2 ↓ 5 2	2
5	8	5 → 8 ↓ 5 2	5

3 Queue

Definition:

A Queue is a FIFO (First In, First Out) data structure where elements are added at the back and removed from the front.



Initialization in Java & C++

Java (`Queue<T>`)

java

 Copy  Edit

```
import java.util.LinkedList;
import java.util.Queue;
Queue<Integer> queue = new LinkedList<>();
```

C++ (`queue<T>`)

cpp

 Copy  Edit

```
#include <queue>
using namespace std;
queue<int> q;
```

Frequently Used Queue Operations in CP:

Operation	Java (<code>Queue<T></code>)	C++ (<code>queue<T></code>)	Time Complexity
Enqueue (Add Element at End)	<code>queue.add(x);</code>	<code>q.push(x);</code>	O(1)
Dequeue (Remove from Front)	<code>queue.remove();</code>	<code>q.pop();</code>	O(1)
Front Element	<code>queue.peek();</code>	<code>q.front();</code>	O(1)
Check if Empty	<code>queue.isEmpty();</code>	<code>q.empty();</code>	O(1)
Size of Queue	<code>queue.size();</code>	<code>q.size();</code>	O(1)

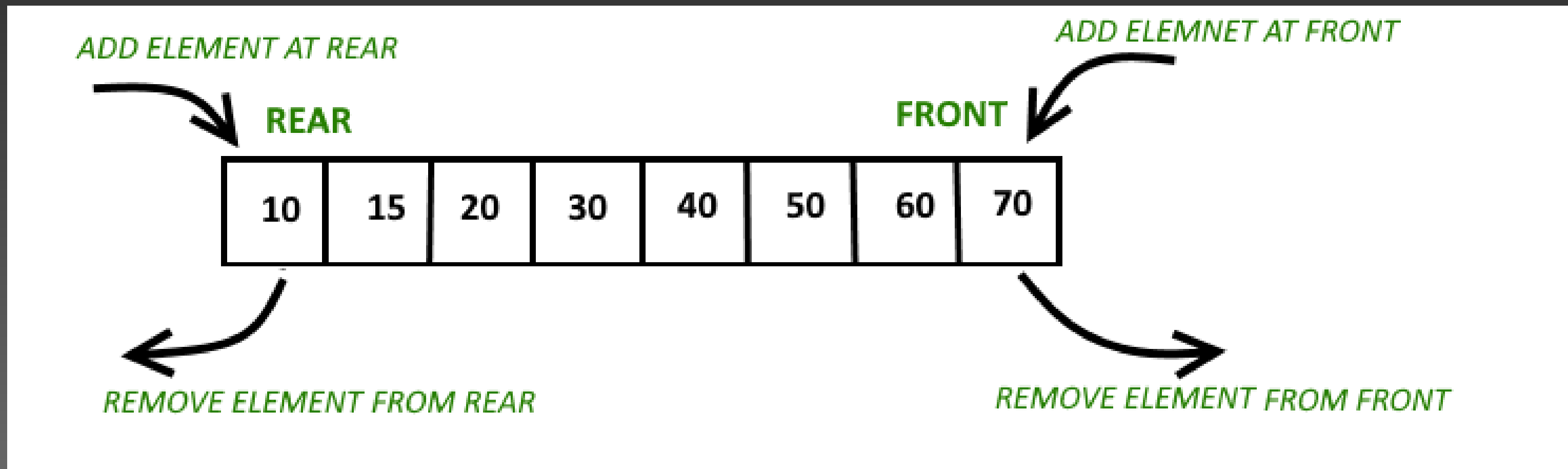
When to Use & When to Avoid Queues in CP

- ✅ Use a Queue When:
 - ✔ FIFO behavior is required (first in, first out).
 - ✔ Processing elements in order, like BFS (Breadth-First Search) or task scheduling.
 - ✔ Sliding Window problems where order matters.
- ❌ Avoid a Queue When:
 - ✔ You need random access to elements (use `vector` or `ArrayList` instead).
 - ✔ You need insertions/deletions from both ends (use a Deque instead).
 - ✔ Priority ordering is required (use a Priority Queue instead).

4 Deque (Double-Ended Queue)



Definition:

A Deque (Double-Ended Queue) is a generalized queue where elements can be added or removed from both the front and the back.



Java (Deque<T>)

java

 Copy  Edit

```
import java.util.Deque;
import java.util.LinkedList;
Deque<Integer> deque = new LinkedList<>();
```

C++ (deque<T>)

cpp

 Copy  Edit

```
#include <deque>
using namespace std;
deque<int> dq;
```

Frequently Used Deque Operations in CP:

Operation	Java (<code>Deque<T></code>)	C++ (<code>deque<T></code>)	Time Complexity
Push Front	<code>deque.addFirst(x);</code>	<code>dq.push_front(x);</code>	O(1)
Push Back	<code>deque.addLast(x);</code>	<code>dq.push_back(x);</code>	O(1)
Pop Front	<code>deque.removeFirst();</code>	<code>dq.pop_front();</code>	O(1)
Pop Back	<code>deque.removeLast();</code>	<code>dq.pop_back();</code>	O(1)
Front Element	<code>deque.peekFirst();</code>	<code>dq.front();</code>	O(1)
Back Element	<code>deque.peekLast();</code>	<code>dq.back();</code>	O(1)

📌 When to Use & When to Avoid Deques in CP 🚀

- ✅ Use a Deque When: ✓ Fast insertions and deletions from both ends are required ($O(1)$).
- ✓ Sliding Window problems where maintaining the order of elements is necessary.
- ✓ Optimizing BFS when shortest paths differ.
- ❌ Avoid a Deque When: ✓ Random access to elements is required (use `vector` or `ArrayList`).
- ✓ A simple queue or stack is sufficient for the problem.

✓ Problem Statement:

You are given a string S consisting of lowercase letters. Determine whether S is a palindrome (i.e., it reads the same forward and backward).

💡 Constraints:

- $1 \leq |S| \leq 10^5$

◆ Input:

A single string S .

◆ Output:

Print "YES" if the string is a palindrome, otherwise print "NO".

✓ Example 1:

vbnet

Copy Edit

Input: "racecar"

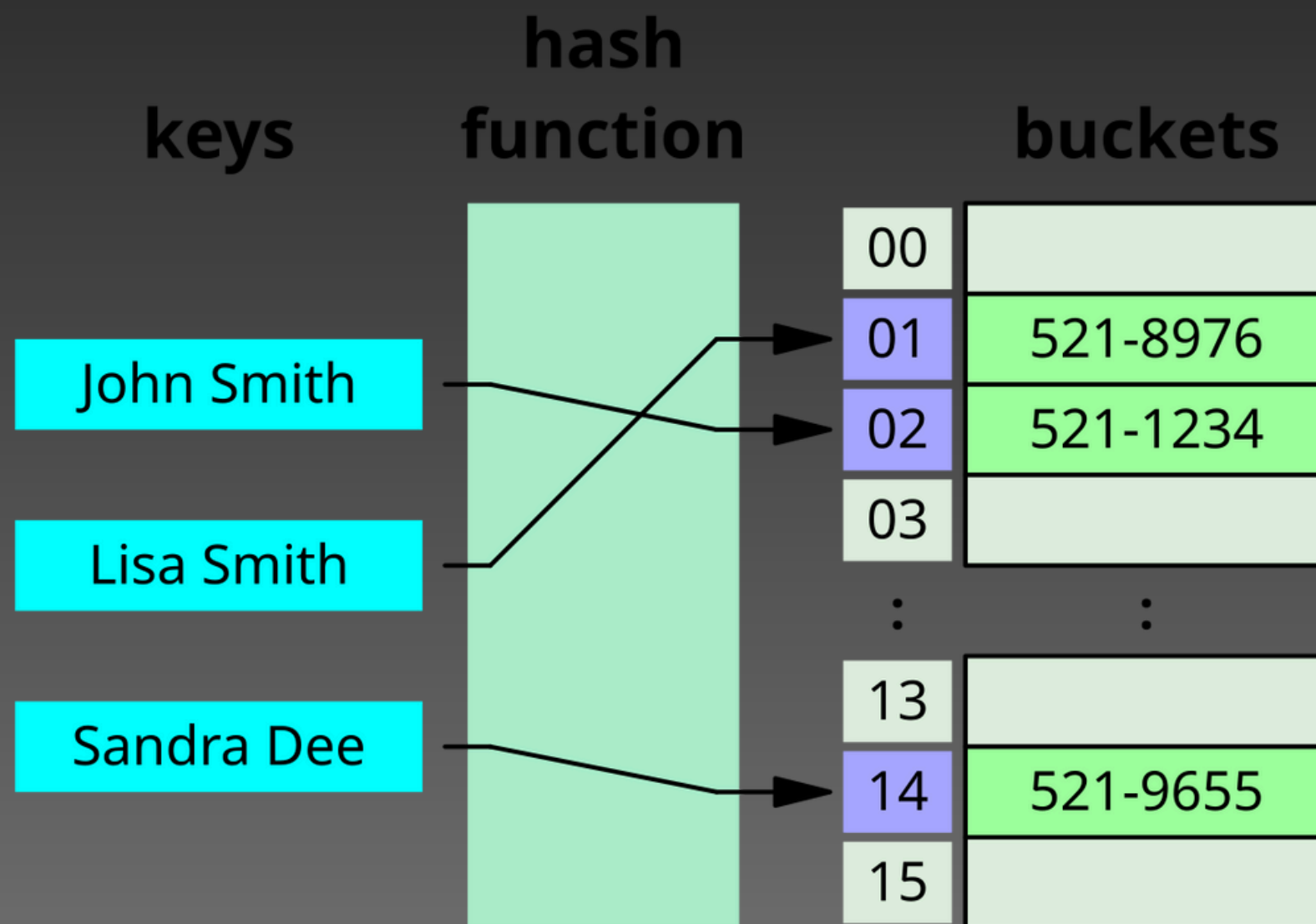
Output: "YES"



5 HashSet (Java) & Unordered Set (C++)

Definition:

A HashSet (Java) and Unordered Set (C++) store unique elements only and provide $O(1)$ average-time complexity for insertions, deletions, and lookups.



Java (HashSet<T>)



java

 Copy  Edit

```
import java.util.HashSet;  
HashSet<Integer> set = new HashSet<>();
```

C++ (unordered_set<T>)

cpp

 Copy  Edit

```
#include <unordered_set>  
using namespace std;  
unordered_set<int> uset;
```

Frequently Used HashSet Operations in CP:

Operation	Java (<code>HashSet<T></code>)	C++ (<code>unordered_set<T></code>)	Time Complexity
Insert Element	<code>set.add(x);</code>	<code>uset.insert(x);</code>	O(1) avg, O(N) worst
Check if Exists	<code>set.contains(x);</code>	<code>uset.find(x) != uset.end();</code>	O(1) avg, O(N) worst
Delete Element	<code>set.remove(x);</code>	<code>uset.erase(x);</code>	O(1) avg, O(N) worst
Get Size	<code>set.size();</code>	<code>uset.size();</code>	O(1)
Clear All Elements	<code>set.clear();</code>	<code>uset.clear();</code>	O(N)

When to Use & When to Avoid HashSets in CP

- ✅ Use a HashSet When:
 - ✓ You need to store unique elements only.
 - ✓ Fast $O(1)$ insert, delete, and search operations are required.
 - ✓ Checking membership in a dataset quickly (e.g., solving problems with duplicate constraints).
- ❌ Avoid a HashSet When:
 - ✓ You need elements in sorted order (use TreeSet instead).
 - ✓ Frequent insertions/removals are expected and hash collisions are high ($O(N)$ worst-case complexity).

G. Sum of Two Values

time limit per test: 2 seconds

memory limit per test: 256 megabytes

You are given an array of n integers, and your task is to find two values (at distinct positions) whose sum is x .

Input

The first input line has two integers n and x : the array size and the target sum.

The second line has n integers a_1, a_2, \dots, a_n : the array values.

Constraints:

- $2 \leq n \leq 2 \cdot 10^5$
- $1 \leq x, a_i \leq 10^9$

Output

Print two integers: the positions of the values. If there are several solutions, you may print any of them. If there are no solutions, print -1 .

Example

input	Copy
4 8 2 7 5 1	
output	Copy
2 4	

Problem - 102961G

Codeforces. Programming competitions and contests, programming community

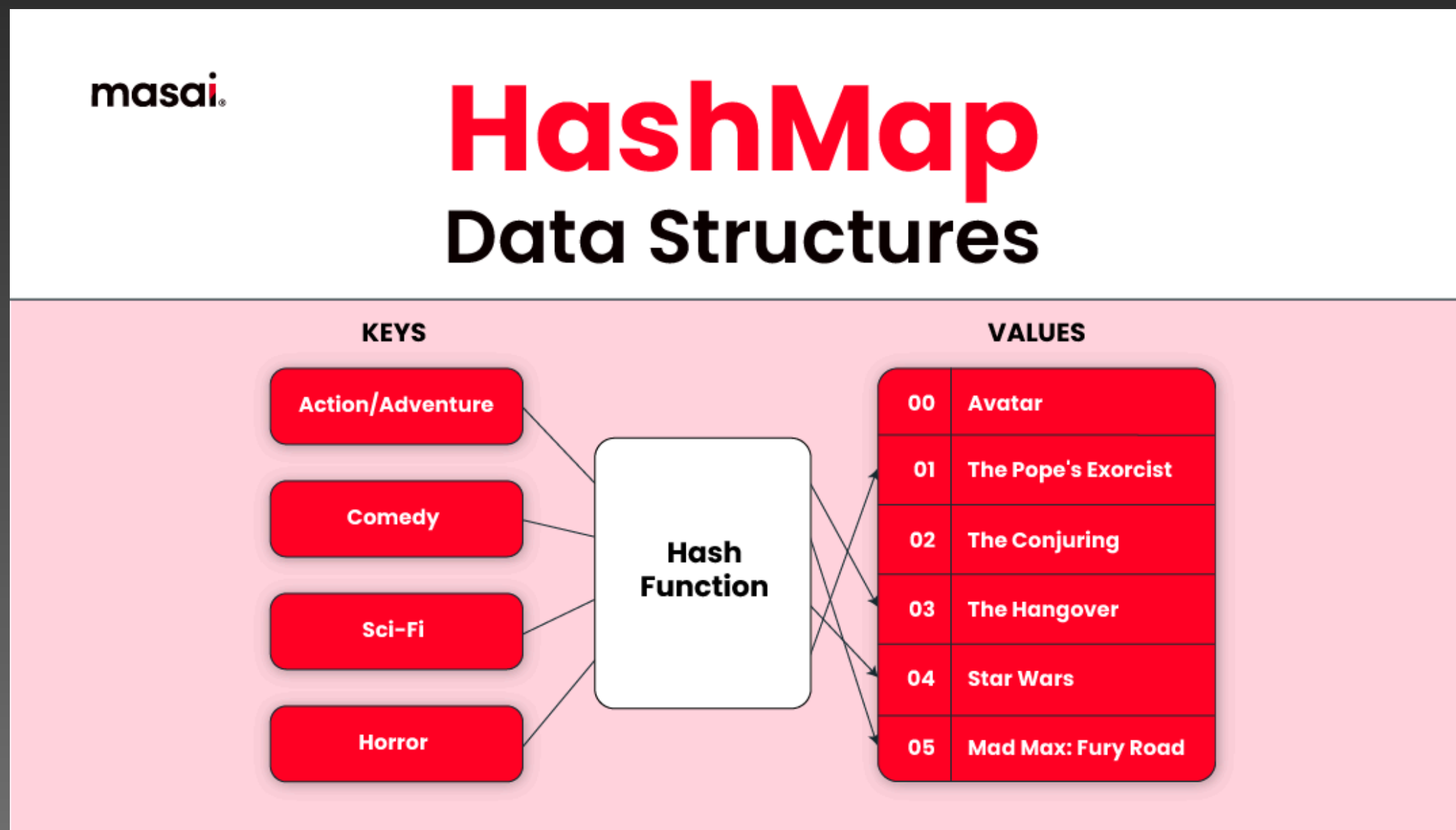
```
public static void main(String[] args) throws IOException {  
    int n=sc.nextInt();  
    long x=sc.nextInt();  
    HashMap<Long,Integer> map = new HashMap<>();  
    for(int i=0;i<n;i++) {  
        long c=sc.nextLong();  
        if(map.containsKey(x-c))  
        {  
            pw.println(i+1 + " " +map.get(x-c));  
            pw.close();  
            return;  
        }  
        map.put(c,i+1);  
    }  
    pw.println("-1");  
    pw.close();  
}
```

Time Complexity :

6 HashMap (Java) & Unordered Map (C++)

Definition:

A HashMap (Java) and Unordered Map (C++) store key-value pairs and provide $O(1)$ average-time complexity for insertions, deletions, and lookups.



Java (`HashMap<K, V>`)

java

 Copy  Edit

```
import java.util.HashMap;  
HashMap<Integer, String> map = new HashMap<>();
```

C++ (`unordered_map<K, V>`)

cpp

 Copy  Edit

```
#include <unordered_map>  
using namespace std;  
unordered_map<int, string> umap;
```

Frequently Used HashMap Operations in CP:

Operation	Java (<code>HashMap<K, V></code>)	C++ (<code>unordered_map<K, V></code>)	Time Complexity
Insert Key-Value	<code>map.put(key, value);</code>	<code>umap[key] = value;</code>	$O(1)$ avg, $O(N)$ worst
Get Value	<code>map.get(key);</code>	<code>umap[key];</code>	$O(1)$ avg, $O(N)$ worst
Delete Key	<code>map.remove(key);</code>	<code>umap.erase(key);</code>	$O(1)$ avg, $O(N)$ worst
Check If Key Exists	<code>map.containsKey(key);</code>	<code>umap.find(key) != umap.end();</code>	$O(1)$ avg, $O(N)$ worst
Size of Map	<code>map.size();</code>	<code>umap.size();</code>	$O(1)$

📌 When to Use & When to Avoid HashMaps in CP 🚀

- ✅ Use a HashMap When:
 - ✓ You need fast key-value lookups ($O(1)$ avg).
 - ✓ Frequency counting, memoization (DP), or adjacency lists in graphs are required.
 - ✓ You need a fast way to store and retrieve mapped data efficiently.
- ❌ Avoid a HashMap When:
 - ✓ You need keys in sorted order (use TreeMap instead).
 - ✓ You expect frequent insertions/removals with potential hash collisions (worst case $O(N)$).
 - ✓ Memory usage is a concern, as HashMaps can consume more memory compared to other structures.



Description: Given an array of integers `nums` and an integer `target`, return the indices of the two numbers such that they add up to `target`.

Constraints:

- Each input will have **exactly one solution**.
- You may not use the same element twice.

Example:

vbnet

 Copy  Edit

Input: `nums = [2, 7, 11, 15]`, `target = 9`

Output: `[0, 1]`

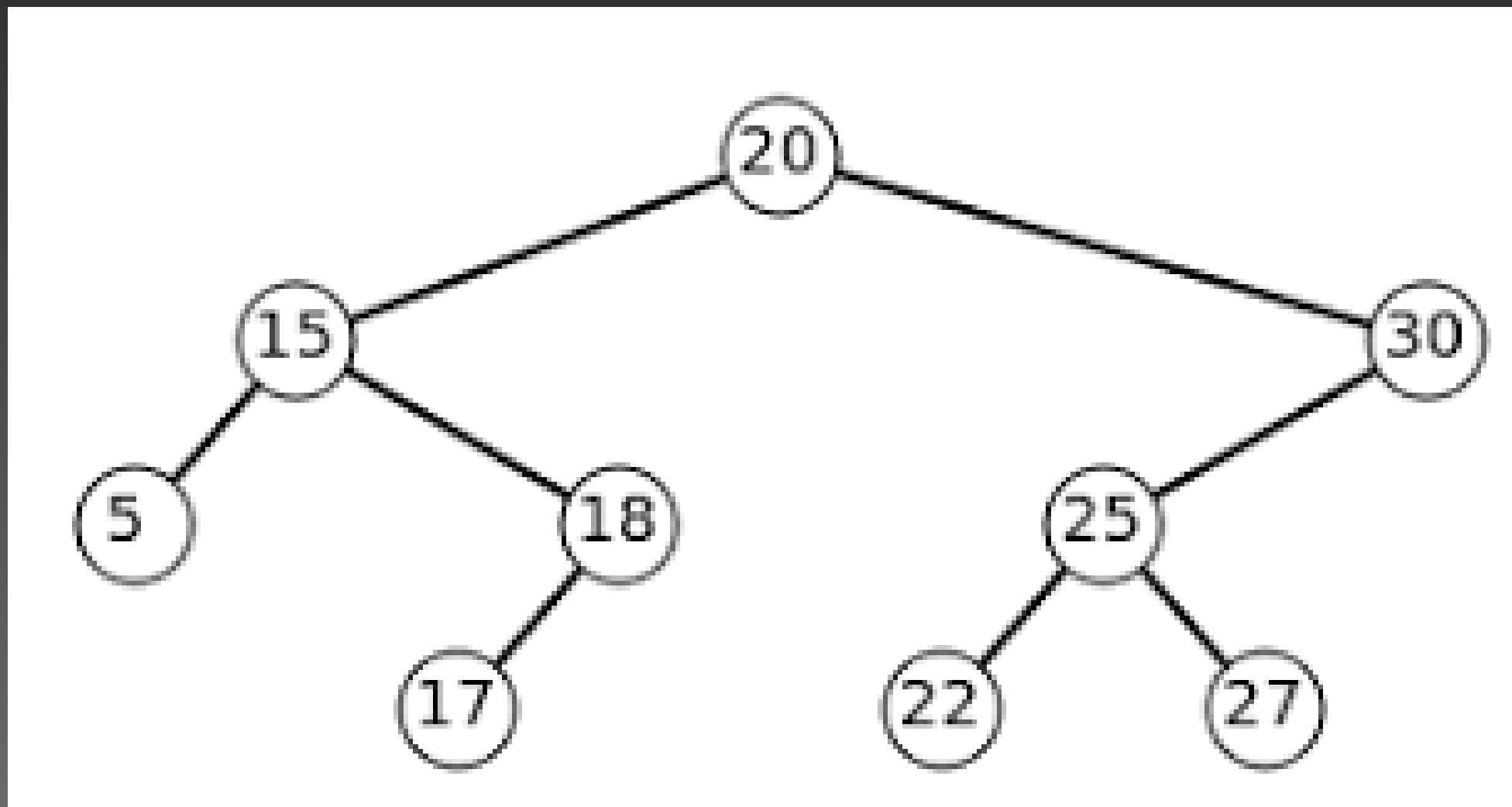
Explanation: Because `nums[0] + nums[1] == 9`, we **return** `[0, 1]`.

Time Complexity :

7 TreeSet (Java) & Set (C++)



Definition:

A TreeSet (Java) and Set (C++) store unique elements in sorted order, providing $O(\log N)$ operations for insertions, deletions, and lookups.



Java (TreeSet<T>)



java

 Copy  Edit

```
import java.util.TreeSet;  
TreeSet<Integer> set = new TreeSet<>();
```

C++ (set<T>)

cpp







 Copy  Edit

```
#include <set>  
using namespace std;  
set<int> s;
```

Frequently Used TreeSet Operations in CP:

Operation	Java (<code>TreeSet<T></code>)	C++ (<code>set<T></code>)	Time Complexity
Insert Element	<code>set.add(x);</code>	<code>s.insert(x);</code>	$O(\log N)$
Check If Exists	<code>set.contains(x);</code>	<code>s.find(x) != s.end();</code>	$O(\log N)$
Delete Element	<code>set.remove(x);</code>	<code>s.erase(x);</code>	$O(\log N)$
Find First Element	<code>set.first();</code>	<code>*s.begin();</code>	$O(1)$
Find Last Element	<code>set.last();</code>	<code>*s.rbegin();</code>	$O(1)$
Find Closest Higher Element	<code>set.higher(x);</code>	<code>s.upper_bound(x);</code>	$O(\log N)$
Find Closest Lower Element	<code>set.lower(x);</code>	<code>s.lower_bound(x);</code>	$O(\log N)$

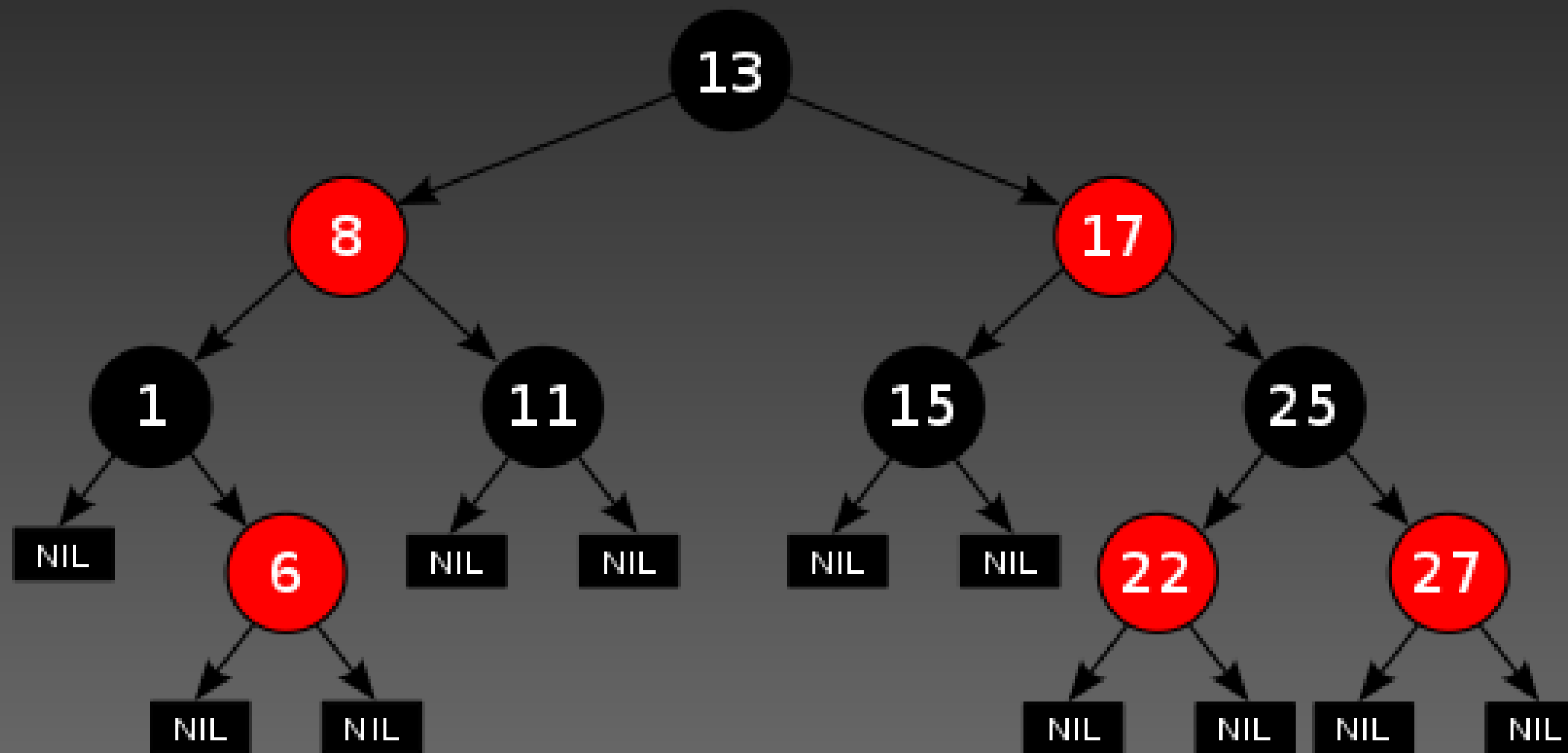
When to Use & When to Avoid TreeSet in CP

- ✅ **Use a TreeSet When:**  You need sorted unique elements.
-  Finding the smallest/largest element efficiently is required.
-  You need range-based queries where order matters.
- ❌ **Avoid a TreeSet When:**  Fast $O(1)$ insert/search is required (use HashSet instead).
-  Sorting is not necessary, making TreeSet slower than HashSet.
-  Memory constraints exist, as maintaining a balanced tree is costly.

8 TreeMap (Java) – Ordered Key-Value Storage

Definition:

A TreeMap in Java is a sorted key-value storage that maintains keys in ascending order and provides $O(\log N)$ operations for insertions, deletions, and lookups.



Java (`TreeMap<K, V>`)

java

 Copy  Edit

```
import java.util.TreeMap;  
TreeMap<Integer, String> map = new TreeMap<>();
```


C++ (`map<K, V>`)

cpp

 Copy  Edit

```
#include <map>  
using namespace std;  
map<int, string> mp;
```

Frequently Used TreeMap Operations in CP:

Operation	Java (<code>TreeMap<K, V></code>)	C++ (<code>map<K, V></code>)	Time Complexity
Insert Key-Value	<code>map.put(key, value);</code>	<code>mp[key] = value;</code>	$O(\log N)$
Get Value	<code>map.get(key);</code>	<code>mp[key];</code>	$O(\log N)$
Get Value with Default	<code>map.getDefault(key, defaultVal);</code>	<code>mp.count(key) ? mp[key] : defaultVal;</code>	$O(\log N)$
Delete Key	<code>map.remove(key);</code>	<code>mp.erase(key);</code>	$O(\log N)$
Find Closest Higher Key	<code>map.higherKey(x);</code>	<code>mp.upper_bound(x);</code>	$O(\log N)$
Find Closest Lower Key	<code>map.lowerKey(x);</code>	<code>mp.lower_bound(x);</code>	$O(\log N)$
First Key (Min Key)	<code>map.firstKey();</code>	<code>mp.begin()->first;</code>	$O(1)$
Last Key (Max Key)	<code>map.lastKey();</code>	<div> <code>mp.rbegin()->first;</code></div>	$O(1)$

When to Use & When to Avoid TreeMap in CP

- ✅ Use a TreeMap When:
 - ✓ You need sorted key-value storage with efficient retrieval.
 - ✓ Finding the smallest/largest key efficiently is required.
 - ✓ You need range-based queries where order matters.
- ❌ Avoid a TreeMap When:
 - ✓ You don't need sorted keys, making HashMap faster ($O(1)$ avg case).
 - ✓ Performance matters more than order, as TreeMap is slower than HashMap.
 - ✓ Memory constraints exist, as maintaining a balanced tree is costly.

Problem Statement: Word Frequency Counter



Description: Given a paragraph of text, write a Java program to count the frequency of each unique word. The program should display the words in alphabetical order along with their corresponding frequencies.

Example:

Input: "apple banana apple orange banana apple"

Output:

makefile

 Copy  Edit

```
apple: 3
```

```
banana: 2
```

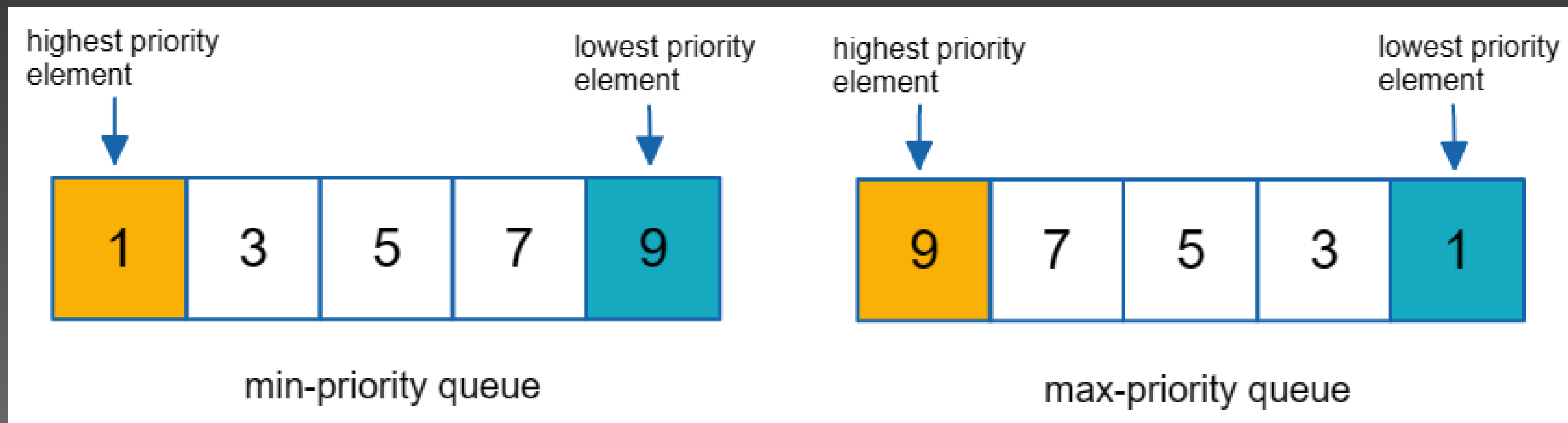
```
orange: 1
```

Time Complexity :

9 Priority Queue (Java & C++)



Definition:

- A Priority Queue is a special type of queue where elements are processed based on priority rather than order of insertion.
- By default, it behaves as a Min-Heap (smallest element at the top), but can be customized as a Max-Heap.



Java (PriorityQueue<T>)



java

 Copy  Edit

```
import java.util.PriorityQueue;
PriorityQueue<Integer> pq = new PriorityQueue<>(); // Min-Heap (default)
PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a); // Max-Heap
```

C++ (priority_queue<T>)

cpp

 Copy  Edit

```
#include <queue>
using namespace std;
priority_queue<int> maxPQ; // Max-Heap (default)
priority_queue<int, vector<int>, greater<int>> minPQ; // Min-Heap
```


Frequently Used Priority Queue Operations in CP:

Operation	Java (<code>PriorityQueue<T></code>)	C++ (<code>priority_queue<T></code>)	Time Complexity
Insert Element	<code>pq.add(x);</code>	<code>pq.push(x);</code>	$O(\log N)$
Remove Top Element	<code>pq.poll();</code>	<code>pq.pop();</code>	$O(\log N)$
Get Top Element	<code>pq.peak();</code>	<code>pq.top();</code>	$O(1)$
Check if Empty	<code>pq.isEmpty();</code>	<code>pq.empty();</code>	$O(1)$
Get Size	<code>pq.size();</code>	<code>pq.size();</code>	$O(1)$

📌 When to Use & When to Avoid Priority Queues in CP 🚀

- ✅ Use a Priority Queue When: ✓ You need dynamic ordering of elements (smallest or largest first).
 - ✓ Dijkstra's algorithm, Huffman coding, or Top-K problems require priority-based access.
 - ✓ Efficient extraction of min/max elements is required.
- ❌ Avoid a Priority Queue When: ✓ You need fast element lookup (use `HashMap` instead).
 - ✓ Sorting the entire dataset upfront is more efficient than inserting elements dynamically.
 - ✓ Memory is a constraint, as maintaining a heap requires extra space.

Problem Statement: Process Tasks with Different Priorities

You are given **N** tasks, each with a priority value. The tasks must be processed based on their priority, with the **highest priority tasks being completed first**.

If two tasks have the same priority, the one that arrived first should be processed first.

◆ **Input Format:**

- The first line contains an integer **N** (number of tasks).
- The next **N** lines contain two integers:
 - **Task ID** (unique identifier).
 - **Priority** (higher value = higher priority).

◆ **Output:**

- Process the tasks in order of priority.
- If two tasks have the same priority, process the one that appeared first in the input.

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    int N = scanner.nextInt();  
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) ->  
        b[1] != a[1] ? b[1] - a[1] : a[0] - b[0]); // Higher priority first, then earlier  
  
    for (int i = 0; i < N; i++) {  
        int id = scanner.nextInt();  
        int priority = scanner.nextInt();  
        pq.add(new int[]{id, priority});  
    }  
  
    while (!pq.isEmpty()) {  
        System.out.println(pq.poll()[0]); // Process tasks  
    }  
}
```

To be continued :

- Pairs Class in Java & C++
- Lambda Expressions in details