

Lab 2 - Solution

Problem 1

Look at the following function:

```
def countdown(n):  
    if n == 0:  
        return  
    print(n)  
    countdown(n - 1)
```

If we call `countdown(3)`, what will be printed? Explain step by step how the function works.

Answer

Step-by-step explanation when calling `countdown(3)`:

1. **`countdown(3)`**
 - Prints: 3
 - Calls `countdown(2)`
2. **`countdown(2)`**
 - Prints: 2
 - Calls `countdown(1)`
3. **`countdown(1)`**
 - Prints: 1
 - Calls `countdown(0)`
4. **`countdown(0)`**
 - Hits base case (`n == 0`), stops recursion, returns to previous call.

Final output printed:

3
2
1

Problem 2

What will be the output of calling `sum_numbers(3)` in the following function?

```
def sum_numbers(n):  
    if n == 0:  
        return 0  
    return n + sum_numbers(n - 1)
```

Show the calculation step by step.

Answer

Step-by-step calculation for `sum_numbers(3)`:

1. **`sum_numbers(3)`** = 3 + `sum_numbers(2)`
2. **`sum_numbers(2)`** = 2 + `sum_numbers(1)`
3. **`sum_numbers(1)`** = 1 + `sum_numbers(0)`
4. **`sum_numbers(0)`** = 0 (base case)

Now, substituting back:

- `sum_numbers(1)` = 1 + 0 = 1
- `sum_numbers(2)` = 2 + 1 = 3
- `sum_numbers(3)` = 3 + 3 = 6

Final output: 6

Problem 3

What is the Big-O complexity of the following function?

```
def find_max(arr):  
    max_value = arr[0] # Line 1  
    for num in arr: # Line 2  
        if num > max_value: # Line 3  
            max_value = num # Line 4  
    return max_value # Line 5
```

Write $T(n)$ (the exact count of operations) and then simplify it to find $O(n)$.

Answer

Step-by-step complexity analysis:

- **Line 1:** executes **once** → 1 operation.
- **Lines 2-4:** loop through the entire array of size n :
 - The loop itself runs $n + 1$ times (including the final check that exits the loop).
 - Checking condition (**Line 3**): runs n times.
 - Assignment (**Line 4**): in worst case, runs n times.
- **Line 5:** executes **once** → 1 operation.

Exact operation count $T(n)$:

$$T(n) = 1 + (n + 1) + n + n + 1 = 3n + 3$$

Simplified Big-O complexity:

- Drop constants and lower-order terms: $O(n)$

Final Answer: $O(n)$

Problem 4

Both functions below calculate the sum of numbers from 1 to n . Analyze their time complexity and determine which one is faster.

Code 1: Using a Loop

```
def sum_loop(n):  
    total = 0  
    for i in range(1, n + 1):  
        total += i  
    return total
```

Code 2: Using a Formula

```
def sum_formula(n):  
    return (n * (n + 1)) // 2
```

1. Determine which function is faster.

Answer: The **formula-based function (sum_formula)** is faster

2. Explain your reasoning based on time complexity.

Answer

The **sum_loop function** runs in **linear time complexity $O(n)$** , meaning its execution time increases as n grows.

The **sum_formula function** operates in **constant time complexity $O(1)$** , meaning its execution time stays the same regardless of the size of n .

Thus, the formula function is more efficient, especially for larger inputs.

Problem 5

Compare the growth strategies of arrays and linked lists when storing large amounts of data. Which one is better for frequent insertions and deletions?

Answer

When storing large amounts of data:

- **Arrays:**

- **Growth Strategy:** Must allocate large contiguous memory blocks as they grow.
- **Insertions/Deletions:**
 - Fast at the end: $O(1)$
 - Slow in the middle or beginning: $O(n)$, because shifting elements is required.
- **Linked Lists:**
 - **Growth Strategy:** Can allocate nodes individually, non-contiguously.
 - **Insertions/Deletions:**
 - Fast at any position (if position is known): $O(1)$
 - Finding positions can still take $O(n)$.

Conclusion:

- For frequent insertions/deletions, **linked lists** are generally better because they avoid the costly shifting of elements required by arrays.

Problem 6

Consider the following queue implementation using a list:

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        return "Queue is empty"

    def is_empty(self):
        return len(self.queue) == 0
```

What is the time complexity of the `dequeue()` operation?

Answer

- The operation `pop(0)` removes the first element in a Python list.
- Removing the first element requires shifting all subsequent elements one position forward.
- If the queue has n elements, this shifting takes **$O(n)$** time.

Conclusion:

The time complexity of the `dequeue()` operation is **$O(n)$** .

Problem 7

Consider the following stack implementation using an array:

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        return "Stack is empty"

    def is_empty(self):
        return len(self.stack) == 0
```

1. What happens when we call `pop()` on an empty stack?

Answer

When `pop()` is called:

- The method first checks if the stack is empty by calling `self.is_empty()`.
 - If the stack is indeed empty (`len(self.stack) == 0`), it returns the string "Stack is empty".
2. Modify the `pop()` method to return `None` instead of a string when the stack is empty.

Answer

```
def pop(self):
    if not self.is_empty():
```

```
return self.stack.pop()
return None # return None if the stack is empty
```

Updated implementation: Returns Python's built-in value None, clearly and concisely signaling that no element is available without needing additional text.

Problem 8

Write a recursive function to compute the product of the first n natural numbers ($n!$), i.e., factorial of n . Then, analyze its time complexity.

Example Cases:

- `factorial(5)` → 120
- `factorial(3)` → 6
- `factorial(0)` → 1

Answer

```
def factorial(n):
    if n == 0:
        return 1 # Base case: factorial(0) = 1
    return n * factorial(n - 1)
```

Complexity Analysis:

- The recursion calls itself n times.
- Each call does a constant amount of work.
- Hence, the total complexity is proportional to the number of recursive calls.

Time complexity:

- The recursion depth is n .
- Thus, time complexity is **$O(n)$** .

Problem 9

In a hospital emergency room, patients are generally treated in the order they arrive. However, patients come in with varying degrees of severity. Patients with critical conditions (urgent) must be treated before those with less severe conditions (regular). You need to design a system that always processes patients with higher severity first. Which data structure should you use?

Answer

Analysis of the Situation:

In this scenario, we have two considerations for patient treatment:

- **Arrival Order:** Normally treated as FIFO (First-In-First-Out).
- **Severity of Condition:** Higher severity patients should be prioritized, even if they arrive later.

A standard queue (FIFO) will not effectively handle severity-based prioritization because it strictly follows arrival order without considering priority.

Suitable Data Structure: Priority Queue

To handle such scenarios, the best-suited data structure is a **Priority Queue**.

Priority Queue Explained:

- A priority queue is a data structure that orders elements based on their **priority** rather than just arrival time.
- Elements with higher priority (e.g., higher severity) are processed before elements with lower priority.

How a Priority Queue works in the hospital scenario:

- Each patient entering the ER is assigned a **priority number** based on severity.
 - Higher number → Higher severity → Higher priority.
- Patients are stored in the queue based on their priority:
 - When a patient arrives, they're inserted into the queue according to their priority level.
 - The next patient to be treated is always the one with the highest severity (highest priority).

Problem 10

The newest huge maths company, South Pacific Computation Corporation (SPPC), is here! SPPC has shares that they need to distribute to their employees. Assume that you are the CEO for SPPC, and you will receive your shares over days. On day one, you receive n shares. On day two, you receive $n/2$ shares (rounded down). On day three, you receive $n/3$ shares (rounded down). One day i , you receive n/i shares (rounded down). On the final day (day n), you receive $n/n = 1$ share. For example, if $n = 3$, then you would receive $3+1+1 = 5$ shares.

How many shares in total do you receive?

Input

The input consists of a single line containing one integer n ($1 \leq n \leq 10^{12}$), which is the number of days over which you receive your shares.

Output

Display the number of shares in total that you receive.

Sample Input	Output
1	1
3	5
4	8
5	10
10	27

Answer

Step-by-step logic:

Step 1: Understand the issue

Calculating shares day-by-day from 1 to n is inefficient (linear complexity $O(n)$), especially when n is very large (10^{12}).

Step 2: Efficient method using grouping

We observe that several consecutive days can yield the same number of shares. Thus, we group them to calculate faster.

Example: For $n=10$:

Day	Shares ($n//\text{day}$)
1	10 ($10//1$)
2	5 ($10//2$)
3	3 ($10//3$)
4	2 ($10//4$)
5	2 ($10//5$)
6	1 ($10//6$)
7	1 ($10//7$)

Day	Shares (n//day)
8	1 (10//8)
9	1 (10//9)
10	1 (10//10)

Notice: day 1 (1 day): 10 shares/day
day 2 (1 day): 5 shares/day
day 3 (1 day): 3 shares/day
days 4–5 (2 days): 2 shares/day
days 6–10 (5 days): 1 share/day

But when n is huge, calculating day-by-day would be inefficient.

Step 2: Group days receiving the same number of shares

- On day i , you get $n//i$ shares.
- To efficiently calculate, we find a range of days that yield the same shares.
- The last day that you receive the same shares as day i is: $j=n // (n//i)$

Formula breakdown clearly explained:

- On day i , shares received = $n//i$.
- Last day receiving $(n//i)$ shares: $\text{last_day} = n // \text{shares}$
- **Days in group:** $\text{days_in_group} = \text{last_day} - i + 1$
- **Total shares added in each group:** $\text{shares per day} * \text{days_in_group}$
- **Next day with fewer shares:** $\text{last_day} + 1$

Step 3: Python Implementation

```
def total_shares(n):
    total = 0
    i = 1
    while i <= n:
        shares = n // i
        last_day = n // shares
        days_in_group = last_day - i + 1
        total += shares * days_in_group
        i = last_day + 1
```



```
i = last_day + 1  
return total
```

Lab 3

Problem 1

A university is ranking students based on their exam scores. The scores must be sorted in **descending order**, maintaining **stability** (students with the same score stay in their original order).

What is Stability in Sorting?

- **Definition:** A sorting algorithm is **stable** if it preserves the relative order of equal elements after sorting.
- **Example:**
 - **Unsorted:** `[("Alice", 85), ("Charlie", 85)]`
 - **Stable Sort Output:** `[("Alice", 85), ("Charlie", 85)]`  (Bob remains before Charlie)
 - **Unstable Sort Output:** `[("Charlie", 85), ("Alice", 85)]`  (Order changed, unfair in ranking systems)

You are given a list of students with their scores:

```
students = [("Alice", 85), ("Bob", 90), ("Charlie", 85), ("David", 92), ("Eve", 90)]
```

Explain **step by step** how **Insertion Sort** sorts this list.

1. **Describe each iteration** of Insertion Sort, showing comparisons, shifts, and insertions.
2. **Show the list's state** after each pass.

Answer

Step-by-Step Execution:

- **Pass 1 (Bob, 90) is inserted at the correct position**
 - Compare "Bob" (90) with "Alice" (85).
 - Since $90 > 85$, shift "Alice" and insert "Bob" before "Alice".

List after Pass 1: `[("Bob", 90), ("Alice", 85), ("Charlie", 85), ("David", 92), ("Eve", 90)]`

- **Pass 2 (Charlie, 85) is inserted at the correct position**

- Compare "Charlie" (85) with "Alice" (85).
- Since `85 == 85`, **no change** (stability is maintained).

List after Pass 2: `[("Bob", 90), ("Alice", 85), ("Charlie", 85), ("David", 92), ("Eve", 90)]`

- **Pass 3 (David, 92) is inserted at the correct position**
 - Compare "David" (92) with "Charlie" (85) → Shift "Charlie".
 - Compare "David" (92) with "Alice" (85) → Shift "Alice".
 - Compare "David" (92) with "Bob" (90) → Shift "Bob".
 - "David" is inserted at the beginning.

List after Pass 3: `[("David", 92), ("Bob", 90), ("Alice", 85), ("Charlie", 85), ("Eve", 90)]`

- **Pass 4 (Eve, 90) is inserted at the correct position**
 - Compare "Eve" (90) with "Charlie" (85) → Shift "Charlie".
 - Compare "Eve" (90) with "Alice" (85) → Shift "Alice".
 - Compare "Eve" (90) with "Bob" (90) → **No shift** (stability maintained).
 - "Eve" is placed **after "Bob" and before "Alice"**.

List after Pass 4 (Final Sorted List): `[("David", 92), ("Bob", 90), ("Eve", 90), ("Alice", 85), ("Charlie", 85)]`

Problem 2

You are managing an **e-commerce platform** and need to display the **top K cheapest products** to customers. Instead of sorting the entire product list, you only need to identify the **K lowest-priced items efficiently**.

You are given the following product prices:

`prices = [299, 150, 89, 199, 49, 120]`

For **K = 3**, the three cheapest products should be:

`[49, 89, 120]`

1. **Explain step by step** how Selection Sort is used to find the **K smallest prices** without sorting the entire list.

2. **Show the intermediate states** of the list after each iteration until the K smallest elements are selected.

Answer

Step-by-Step Execution Using Selection Sort

Selection Sort works by selecting the smallest element and swapping it to its correct position. Instead of sorting the entire list, we only run the first K iterations to find the K smallest elements.

- **Pass 1: Finding the 1st Smallest Price**
 - Scan the entire list to find the **smallest element**.
 - **Smallest value:** 49
 - Swap 49 with the first element (299).

Updated list after Pass 1: [49, 150, 89, 199, 299, 120]

- **Pass 2: Finding the 2nd Smallest Price**
 - Scan the remaining unsorted portion [150, 89, 199, 299, 120].
 - **Smallest value:** 89
 - Swap 89 with 150 (second element).

Updated list after Pass 2: [49, 89, 150, 199, 299, 120]

- **Pass 3: Finding the 3rd Smallest Price**
 - Scan the remaining unsorted portion [150, 199, 299, 120].
 - **Smallest value:** 120
 - Swap 120 with 150 (third element).

Updated list after Pass 3: [49, 89, 120, 199, 299, 150]

Final Selected Prices: [49, 89, 120]

We stop after $K = 3$ iterations instead of fully sorting the list, improving efficiency.

- Standard Selection Sort sorts the entire list ($O(n^2)$).
- By stopping after K iterations, we reduce unnecessary comparisons, improving efficiency to $O(nK)$.
- Useful for ranking scenarios where only a few top elements are needed.

Problem 3

A **restaurant review platform** wants to display customer ratings **in ascending order** to help users easily find the lowest-rated restaurants. However, the ratings are currently **unsorted**, and the platform is using **Bubble Sort** to process them.

You are given the following **restaurant ratings (out of 5 stars)**:

```
ratings = [4.5, 3.2, 5.0, 2.8, 4.0, 3.8]
```

After sorting, the ratings should appear as:

```
[2.8, 3.2, 3.8, 4.0, 4.5, 5.0]
```

1. **Explain step by step** how Bubble Sort sorts this list, showing each full pass and the swaps that occur.
2. How can **Bubble Sort be optimized** for a nearly sorted list? Analyze the time complexity.

Answer

Standard Bubble Sort Execution

Bubble Sort works by repeatedly **swapping adjacent elements** if they are in the wrong order. This continues until the list is fully sorted.

Pass 1: Move the largest element to the end

- Compare 4.5 and 3.2 → **Swap** → [3.2, 4.5, 5.0, 2.8, 4.0, 3.8]
- Compare 4.5 and 5.0 → No swap
- Compare 5.0 and 2.8 → **Swap** → [3.2, 4.5, 2.8, 5.0, 4.0, 3.8]
- Compare 5.0 and 4.0 → **Swap** → [3.2, 4.5, 2.8, 4.0, 5.0, 3.8]
- Compare 5.0 and 3.8 → **Swap** → [3.2, 4.5, 2.8, 4.0, 3.8, 5.0]

State after Pass 1: [3.2, 4.5, 2.8, 4.0, 3.8, 5.0]

Pass 2: Move the second-largest element to its correct position

- Compare 3.2 and 4.5 → No swap
- Compare 4.5 and 2.8 → **Swap** → [3.2, 2.8, 4.5, 4.0, 3.8, 5.0]
- Compare 4.5 and 4.0 → **Swap** → [3.2, 2.8, 4.0, 4.5, 3.8, 5.0]

- Compare 4.5 and 3.8 → **Swap** → [3.2, 2.8, 4.0, 3.8, 4.5, 5.0]

State after Pass 2: [3.2, 2.8, 4.0, 3.8, 4.5, 5.0]

Pass 3: Move the third-largest element to its correct position

- Compare 3.2 and 2.8 → **Swap** → [2.8, 3.2, 4.0, 3.8, 4.5, 5.0]
- Compare 3.2 and 4.0 → No swap
- Compare 4.0 and 3.8 → **Swap** → [2.8, 3.2, 3.8, 4.0, 4.5, 5.0]

State after Pass 3: [2.8, 3.2, 3.8, 4.0, 4.5, 5.0]

At this point, the list is already sorted, but **Bubble Sort will continue checking** for two more passes, even though no swaps are needed.

Pass 4: Move the fourth-largest element to its correct position

- Compare 2.8 and 3.2 → No swap
- Compare 3.2 and 3.8 → No swap

State after Pass 4: [2.8, 3.2, 3.8, 4.0, 4.5, 5.0]

Pass 5: Move the fifth-largest element to its correct position

- Compare 2.8 and 3.2 → No swap

State after Pass 5: [2.8, 3.2, 3.8, 4.0, 4.5, 5.0]

Optimized Bubble Sort (Stopping Early)

- **Issue with Standard Bubble Sort:** Runs $n-1$ passes even if already sorted, causing unnecessary comparisons.
- **Optimization:** Use a swapped flag:
 - Set `swapped = False` before each pass.
 - If any swap occurs, set `swapped = True`.
 - If no swaps occur, stop early (list is sorted).
- **Efficiency Gain:**
 - Prevents redundant iterations, reducing passes.
 - Best-case complexity improves from $O(n^2)$ to $O(n)$.
 - Ideal for sorted or nearly sorted lists, making Bubble Sort more efficient.

Problem 4

Different sorting algorithms perform **differently** depending on the structure of the input data. Your task is to **analyze and compare** the behavior of sorting algorithms when applied to different types of input.

Given the following input cases:

1. **Nearly sorted array:** [1, 2, 3, 4, 6, 5, 7, 8, 9, 10]
2. **Completely reversed array:** [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
3. **Array with many duplicate elements:** [4, 2, 2, 8, 3, 3, 3, 7, 4, 2]

Which **sorting algorithm (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort)** performs **best and worst** for each input type? Explain why.

Answer

Case 1: Nearly Sorted Array

Example: [1, 2, 3, 4, 6, 5, 7, 8, 9, 10]

Best Algorithm: Insertion Sort

- Insertion Sort is highly efficient when the array is nearly sorted.
- **Few swaps needed:** It only moves elements that are out of place. In this case, only 6 and 5 need to be adjusted, minimizing the number of operations.
- **Reaches $O(n)$ complexity:** In the best case (fully sorted), each element is compared once and no swaps are required.

Worst Algorithm: Selection Sort

- Selection Sort **always** performs $O(n^2)$ comparisons, regardless of input order.
- **Inefficient for nearly sorted input:** Even if only one element is misplaced, it still scans the entire list in every iteration.

Case 2: Completely Reversed Array

Example: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Best Algorithm: Merge Sort

- Merge Sort performs $O(n \log n)$ operations consistently, regardless of whether the array is sorted or reversed.
- **Efficient for worst-case scenarios:** Unlike Insertion Sort and Bubble Sort, which perform poorly on reversed data, Merge Sort divides and processes subarrays efficiently.

Worst Algorithm: Bubble Sort/Insertion Sort

- Every element must be swapped into its correct position, leading to $O(n^2)$ swaps.
- **Highly inefficient:** The maximum number of swaps and comparisons occur when the list is in descending order.

Case 3: Array with Many Duplicates

Example: [4, 2, 2, 8, 3, 3, 3, 7, 4, 2]

Best Algorithm: Merge Sort

- Merge Sort maintains $O(n \log n)$ efficiency and does not depend on unique values.
- **Efficiently handles repeated elements:** Sorting performance remains consistent, even when duplicate values are present.

Worst Algorithm: Selection Sort

- Selection Sort **compares every element $O(n^2)$ times**, making it inefficient when sorting a dataset with repeated values.
- **Unnecessary operations:** Even with duplicates, the algorithm does not improve its performance by recognizing similar values.

Problem 5

An e-commerce platform processes thousands of orders daily. Each order contains a **unique order ID** and a **total purchase amount**. To generate financial reports efficiently, the orders need to be **sorted in ascending order by total purchase amount** before analysis.

Given a list of **unsorted orders**, you must choose and explain the steps of a **sorting algorithm with $O(n \log n)$ complexity** to efficiently organize the data. **Describe step by step** how the sorting algorithm processes the list.

Answer

Using Merge Sort

- This sorting algorithm follows a **divide-and-conquer** approach.
- The list is **recursively divided into smaller sublists** until each contains only one element.
- Then, the sublists are **merged back together in sorted order**.

Example: orders = [320, 150, 450, 280, 500, 200, 100]

The goal is to **sort these orders in ascending order** using a **sorting algorithm with $O(n \log n)$ complexity**.

Step-by-Step Sorting Execution

1. Splitting the List (Divide Phase)

The list is **split into two halves repeatedly** until each sublist contains only one element.

Original list: [320, 150, 450, 280, 500, 200, 100]

First Split

Left: [320, 150, 450, 280]

Right: [500, 200, 100]

Second Split

Left (split again): [320, 150] | [450, 280]

Right (split again): [500, 200] | [100]

Final Split into Single Elements

[320] [150] [450] [280] [500] [200] [100]

2. Merging the Sorted Sublists (Conquer Phase)

After dividing, the algorithm **merges sublists back together in sorted order**.

Merging the smallest elements first

[320] and [150] → [150, 320]

[450] and [280] → [280, 450]

[500] and [200] → [200, 500]

Merging the next level of sublists

[150, 320] and [280, 450] → [150, 280, 320, 450]

[200, 500] and [100] → [100, 200, 500]

Final Merge (Sorted List)

[150, 280, 320, 450] and [100, 200, 500] → [100, 150, 200, 280, 320, 450, 500]

Final Sorted Order:

[100, 150, 200, 280, 320, 450, 500]

Problem 6

You are developing a digital photo management system. Users can add new photos to their albums in two ways:

1. **A few new photos** are added to an already sorted album.
2. **A large batch of photos** is imported at once, often in random order.

Which sorting algorithm would be most suitable for **each scenario**, and why? Consider factors such as **time complexity, efficiency, and adaptability** to different input conditions.

Answer

Scenario 1: A Few New Photos Added to an Already Sorted Album

Best Algorithm: Insertion Sort

- **Efficient for nearly sorted data:** Insertion Sort performs in $O(n)$ time when only a few elements are out of place.
- It minimizes unnecessary comparisons and swaps.
- It doesn't require extra space, making it efficient for maintaining the sorted structure.

Scenario 2: A Large Batch of Photos Imported at Once (Unsorted Data)

Best Algorithm: Merge Sort

- **Handles large, unordered datasets efficiently:** Merge Sort guarantees $O(n \log n)$ time complexity in all cases.

- If photos have timestamps, Merge Sort maintains their original order. Without this, photos with identical timestamps could be **rearranged unpredictably**, disrupting the intended order in albums or galleries.

Lab 4

Problem 1

You are given the following nearly sorted list of integers:

[10, 9, 8, 7, 6, 4, 5, 2, 1]

Your task is to **track the number of recursion calls** made during the partitioning steps of **Quick Sort** using different **pivot selection strategies** (first element, last element, and median-of-three). Compare how each pivot selection strategy impacts the resulting subarrays, the number of partition steps, and explain why one strategy might lead to better or worse performance.

*Note: The **median-of-three** method selects the pivot as the **middle value** of the first, middle, and last elements of the array*

Answer

Partition Steps for Each Pivot Selection Method

1. First Element as Pivot:

- **Partition 1:** Pivot = 10, Subarrays = [9, 8, 7, 6, 4, 5, 2, 1] (left), [] (right)
- **Partition 2:** Pivot = 9, Subarrays = [8, 7, 6, 4, 5, 2, 1] (left), [] (right)
- **Partition 3:** Pivot = 8, Subarrays = [7, 6, 4, 5, 2, 1] (left), [] (right)
- **Partition 4:** Pivot = 7, Subarrays = [6, 4, 5, 2, 1] (left), [] (right)
- **Partition 5:** Pivot = 6, Subarrays = [4, 5, 2, 1] (left), [] (right)
- **Partition 6:** Pivot = 4, Subarrays = [2, 1] (left), [5] (right)
- **Partition 7:** Pivot = 2, Subarrays = [1] (left), [] (right)
- **Partition 8:** Pivot = 5, Subarrays = [] (left), [] (right)
- **Total Partitions:** 8

2. Last Element as Pivot:

- **Partition 1:** Pivot = 1, Subarrays = [] (left), [10, 9, 8, 7, 6, 4, 5, 2] (right)
- **Partition 2:** Pivot = 2, Subarrays = [] (left), [10, 9, 8, 7, 6, 4, 5] (right)
- **Partition 3:** Pivot = 5, Subarrays = [4] (left), [10, 9, 8, 7, 6] (right)
- **Partition 4:** Pivot = 4, Subarrays = [] (left), [] (right)
- **Partition 5:** Pivot = 6, Subarrays = [] (left), [10, 9, 8, 7] (right)
- **Partition 6:** Pivot = 7, Subarrays = [] (left), [10, 9, 8] (right)
- **Partition 7:** Pivot = 8, Subarrays = [] (left), [10, 9] (right)

- **Partition 8:** Pivot = 9, Subarrays = [] (left), [10] (right)
- **Total Partitions:** 8

3. Median-of-Three as Pivot:

- **Partition 1:** Pivot = 6 (median of 10, 6, 1), Subarrays = [4, 5, 2, 1] (left), [7, 8, 9, 10] (right)
- **Partition 2: Left Subarray (Pivot = 2, from median of 4, 2, 1),** Subarrays = [1] (left), [4, 5] (right)
- **Partition 3: Right Subarray (Pivot = 9, from median of 7, 9, 10),** Subarrays = [7, 8] (left), [10] (right)
- **Partition 4:** Pivot = 5, Subarrays = [4] (left), [] (right)
- **Partition 5:** Pivot = 8, Subarrays = [7] (left), [] (right)
- **Total Partitions:** 5

Analysis and Comparison

1. First Element as Pivot

- **Unbalanced partition**, with almost all elements placed in the left subarray.
- The right subarray is empty, meaning Quick Sort will continue working on a large, nearly unsorted left portion.
- This leads to inefficient recursion and increases the number of comparisons, resulting in $O(n^2)$ worst-case complexity when the list is already sorted or reverse-sorted.

2. Last Element as Pivot

Again, an **unbalanced partition**, but in the opposite direction—this time, all elements go to the right subarray.

- Just like the first strategy, this leads to excessive recursion depth and poor performance.
- The partitioning does not effectively reduce the problem size, resulting in $O(n^2)$ worst-case complexity.

3. Median-of-Three as Pivot

- **Balanced partition**, with elements distributed relatively evenly between left and right subarrays.
 - Since Quick Sort works best when partitions are balanced, this method reduces recursion depth and keeps the algorithm closer to $O(n \log n)$ complexity.
 - This is the most efficient pivot selection strategy in this case, preventing extreme unbalanced partitions.
-

Problem 2

Imagine you're working for an e-commerce company, and you need to identify the **2nd lowest price** of products from an unsorted list of product prices. To achieve this, you can use the **partition step** of Quick Sort, which is often used in selecting the k-th smallest element. In this case, we'll apply the **median-of-three pivot selection method** to find the **2nd smallest price** in the list.

You are given the following unsorted list of product prices in dollars:

[12, 3, 5, 7, 19, 1, 10, 15]

- Explain how the partitioning process of Quick Sort helps identify the k-th smallest element by using the pivot's position to narrow down the search space.
- Apply the median-of-three pivot selection method to partition the array and progressively narrow down the subarray to find the 2nd smallest element.

Note: Why Use Median-of-Three Instead of Other Methods?

- Picking the first, middle, and last gives a good approximation of the true median. Only **3 comparisons** are needed, avoiding extra overhead.
- **Median-of-Four/Five** – More comparisons (**5-7**) with **minimal improvement**.
- The **average may not exist** in the array, requiring extra operations.

Answer

How the Partitioning Process Helps Find the k-th Smallest Element

1. **Partitioning organizes elements into two groups:**
 - Left: Elements **smaller** than the pivot.
 - Right: Elements **greater** than the pivot.
 - The pivot itself is in its **final sorted position**.
2. **Instead of sorting the entire list, we reduce the problem size:**
 - If the pivot index = $k - 1$, we **found the k-th smallest element**.
 - If $k - 1$ is in the left subarray, we **recur on the left**.
 - If $k - 1$ is in the right subarray, we **recur on the right**.

Step to find 2nd smallest price element

Step 1. Selecting the Median-of-Three Pivot

The **median-of-three method** picks the pivot as the **median of the first, middle, and last elements**.

- **First element = 12**

- **Middle element** (index $\text{len}(\text{arr}) // 2 = 8 // 2 = 4$) = 19
- **Last element** = 15
- **Median of (12, 19, 15) = 15** → **Pivot = 15**

Step 2. Partitioning Around Pivot (15)

We rearrange elements so that:

- Elements **less than 15** go to the left.
- Elements **greater than 15** go to the right.

Partitioning Process:

1. Compare 12 (left) → **keep (less than 15)**.
2. Compare 3 (left) → **keep (less than 15)**.
3. Compare 5 (left) → **keep (less than 15)**.
4. Compare 7 (left) → **keep (less than 15)**.
5. Compare 19 (right) → **move right (greater than 15)**.
6. Compare 1 (left) → **keep (less than 15)**.
7. Compare 10 (left) → **keep (less than 15)**.

Resulting Partitioned List: [12, 3, 5, 7, 1, 10, 15, 19]

Pivot 15 is placed at index 6.

Step 3. Identify the 2nd Smallest Element

- The left subarray: [12, 3, 5, 7, 1, 10] (elements less than 15)
- The right subarray: [19] (elements greater than 15)
- Since we need the **2nd smallest element**, we continue partitioning the **left subarray** [12, 3, 5, 7, 1, 10].

Step 4. Recursive Partitioning on Left Subarray

New subarray: [12, 3, 5, 7, 1, 10]

Selecting the New Pivot (Median-of-Three)

- **First element** = 12
- **Middle element** = 7
- **Last element** = 10
- **Median of (12, 7, 10) = 10** → **New Pivot = 10**

Partitioning Around Pivot (10)

Rearrange elements so that:

- **Less than 10** → [3, 5, 7, 1]
- **Greater than 10** → [12]

Resulting Partitioned List: [3, 5, 7, 1, 10, 12]

Pivot 10 is placed at index 4.

Step 5. Identify the 2nd Smallest Element

- The left subarray: [3, 5, 7, 1] (elements less than 10)
- Since we need the **2nd smallest element**, we continue partitioning **this subarray**.

New Pivot Selection (Median-of-Three)

- **First element** = 3
- **Middle element** = 7
- **Last element** = 1
- **Median of (3, 7, 1) = 3** → **Pivot = 3**

Partitioning Around Pivot (3)

Rearrange elements so that:

- **Less than 3** → [1]
- **Greater than 3** → [5, 7]

Resulting Partitioned List: [1, 3, 5, 7]

Pivot 3 is placed at index 1.

Step 6. Find the 2nd Smallest

- **The 2nd smallest element is at index 1** → "3".

Problem 3

In a class, students' marks are recorded in a list, but many marks are repeated. Standard sorting algorithms like Quick Sort can become inefficient when dealing with numerous duplicates, as they result in excessive comparisons and swaps, leading to **O(n²)** time complexity in the worst case.

To address this, you are tasked with partitioning the list of marks efficiently using the **Three-Way Quick Sort** partitioning method, which reduces unnecessary comparisons by grouping duplicate marks together.

Input:

- `arr = [10, 20, 10, 5, 30, 20, 15, 5, 25, 30, 20]`

Note on Three-Way Quick Sort:

Unlike the standard Quick Sort, which divides the list into two partitions, the **Three-Way Quick Sort** method partitions the list into three sections:

- **Marks less than the pivot** in the first section.
- **Marks equal to the pivot** in the second section.
- **Marks greater than the pivot** in the third section.

Task:

1. **Explain step-by-step** how the Three-Way Quick Sort partitioning method works on the input list. **Write the final list** after applying the method.
2. How can **Three-Way Quick Sort** improve the time complexity in this case compared to **standard Quick Sort**?
3. Based on your analysis, compare how **Three-Way Quick Sort** handles duplicates in comparison to other sorting algorithms like **Merge Sort** and **Heap Sort**. Which method handles duplicates more efficiently, and why?

Answer

Step-by-Step Explanation of Three-Way Quick Sort Partitioning:

Input list:

`arr = [10, 20, 10, 5, 30, 20, 15, 5, 25, 30, 20]`

1. **Choose the pivot:**

We select **20** as the pivot. The goal is to partition the list into three sections:

- Marks less than the pivot.
- Marks equal to the pivot.
- Marks greater than the pivot.

2. **Partitioning Process:**

- We initialize three pointers:
 - `low` (start of the list).
 - `mid` (traverses through the list).
 - `high` (end of the list).
- **Step-by-step partitioning:**

- `arr[mid] = 10` is less than 20. We swap `arr[mid]` and `arr[low]`, then increment both `low` and `mid`.
 - `arr[mid] = 20` is equal to 20. We just increment `mid`.
 - `arr[mid] = 10` is less than 20. We swap `arr[mid]` and `arr[low]`, then increment both `low` and `mid`.
 - `arr[mid] = 5` is less than 20. We swap `arr[mid]` and `arr[low]`, then increment both `low` and `mid`.
 - `arr[mid] = 30` is greater than 20. We swap `arr[mid]` and `arr[high]`, then decrement `high`.
 - `arr[mid] = 20` is equal to 20. We just increment `mid`.
 - This process continues until the array is fully partitioned, with elements less than 20 in the first section, equal to 20 in the second, and greater than 20 in the third.
3. **Final Partitioned List:** After partitioning, the final array will look like this:
- **Marks less than 20:** `[10, 10, 5, 5, 15]`
 - **Marks equal to 20:** `[20, 20, 20]`
 - **Marks greater than 20:** `[30, 30, 25]`
4. **Partitioned list:** `[10, 10, 5, 5, 15, 20, 20, 20, 30, 30, 25]`
5. Now, Quick Sort will recursively sort the **lower part** (elements less than 20) and the **higher part** (elements greater than 20).
6. Sorting the Lower Part `[10, 10, 5, 5, 15]` with pivot 10 will result in `[5, 5, 10, 10, 15]`
7. Sorting the Higher Part `[30, 30, 25]` with pivot 30 will result in `[25, 30, 30]`
8. **Final sorted list:** `[5, 5, 10, 10, 15, 20, 20, 20, 25, 30, 30]`

Time Complexity of Three-Way Quick Sort:

- In this case, **standard Quick Sort** can degrade to $O(n^2)$ when many duplicates exist, as it repeatedly partitions the same elements without meaningful reductions. This happens because standard Quick Sort partitions into only two sections, leading to excessive swaps and comparisons.
- **Three-Way Quick Sort**, however, efficiently handles duplicates by grouping all equal elements together in one pass. This reduces the number of recursive calls, leading to improved performance, ensuring the average time complexity remains at $O(n \log n)$

Comparison with Merge Sort and Heap Sort:

1. Merge Sort:

- **Time Complexity:** Merge Sort has a time complexity of $O(n \log n)$, even in the worst case, as it divides the array into two halves and recursively merges them. It does not handle duplicates any differently than other elements, but the algorithm will still perform $O(n \log n)$ comparisons and merge steps, even when duplicates are present.

- **Handling Duplicates:** Merge Sort does not optimize for duplicates. It will still perform comparisons for each duplicate element during the merge phase, making it less efficient when there are many duplicates compared to Three-Way Quick Sort.

2. Heap Sort:

- **Time Complexity:** Heap Sort also has a time complexity of $O(n \log n)$ in all cases. It builds a heap and then repeatedly extracts the maximum (or minimum) element.
- **Handling Duplicates:** Heap Sort does not have a specific optimization for handling duplicates. While it performs $O(\log n)$ operations for each element during the heapification process, it does not minimize the comparisons for duplicate elements as Three-Way Quick Sort does.

3. Three-Way Quick Sort:

- Three-Way Quick Sort excels in scenarios with many duplicate values, such as sorting student exam scores, product ratings. In large datasets, many students may have the same score, numerous products may share identical ratings. By grouping duplicates in a single pass, this method reduces redundant comparisons and swaps, minimizing recursive calls. This can achieve $O(n)$ in the best case. Compared to Merge Sort and Heap Sort, it handles duplicates more efficiently, making it ideal for datasets with frequent repeated values.

Conclusion: **Three-Way Quick Sort** is the most efficient in handling duplicates, especially when compared to **Merge Sort** and **Heap Sort**, as it specifically optimizes for duplicates by grouping equal elements together, reducing unnecessary work and comparisons.

Problem 4

You are tasked with implementing a sorting algorithm for a **database system** that manages large numbers of **customer records** (each containing customer information such as names, ages, and addresses). The records are stored in an array, and you need to sort the records based on **customer age in ascending order**.

Scenario 1: Sorting User Records for a Fast Real-Time Application

- The database contains a **large dataset of user records**, and the system is a **real-time application** (e.g., an e-commerce platform). The data is **frequently updated**, and sorting must occur efficiently. The primary goal is to ensure the **sorting operation is as fast as possible**, with **minimal memory overhead**.

Scenario 2: Sorting Large External Files for Storage

- The database contains **very large datasets** (e.g., **terabytes of data**) stored across **external storage devices** (e.g., hard drives, cloud storage). The data is **too large to**

fit into memory all at once and needs to be sorted in **chunks**. **Stability** is important, as the relative order of records with the same age must be preserved, especially when multiple sorting operations (e.g., sorting by name after sorting by age) are required.

Which algorithm (Merge Sort or Quick Sort) would you choose for sorting the records for each scenario, and why?

Answer

Scenario 1: Sorting User Records for a Fast Real-Time Application

- For a real-time application that requires fast sorting with minimal memory overhead, **Quick Sort** is the preferred algorithm. Quick Sort has an average-case time complexity of $O(n \log n)$ and is generally faster than Merge Sort for in-memory sorting due to lower memory usage. Since the data is frequently updated, Quick Sort's in-place nature is beneficial, reducing memory overhead compared to Merge Sort, which requires additional space for merging.

Scenario 2: Sorting Large External Files for Storage

- For large datasets that cannot fit into memory and require external sorting, **Merge Sort** is the better choice. Merge Sort has a consistent $O(n \log n)$ time complexity and is a **stable sorting algorithm**, preserving the relative order of records with the same key (age). This is particularly useful when secondary sorting operations, such as sorting by name after sorting by age, are required.
- Since Merge Sort processes data in **chunks (divide-and-conquer approach)**, it is well-suited for **external sorting**, where data is read from and written to storage efficiently. The **External Merge Sort** variant is commonly used in database systems to handle massive datasets, as it allows sorting large files efficiently by breaking them into smaller sorted segments and merging them iteratively.

Problem 5

```
def heapify(arr, n, i):
    swaps = 0
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
```

```

    arr[i], arr[largest] = arr[largest], arr[i]
    swaps += 1
    swaps += heapify(arr, n, largest)
return swaps

```

Manually simulate the call `heapify(arr, 5, 0)` for the array `[4, 10, 3, 5, 1]`.

- List the sequence of comparisons and swaps that occur.
- How many swaps occur in total during this call?

Answer

Step-by-Step Execution of `heapify(arr, 5, 0)`

1. **Initial Setup**
 - Root ($i = 0$) = 4
 - Left child ($left = 2 * 0 + 1 = 1$) = 10
 - Right child ($right = 2 * 0 + 2 = 2$) = 3
2. **First Comparison**
 - Compare `arr[left] = 10` with `arr[largest] = 4`
 - $10 > 4$, so `largest = 1`
3. **Second Comparison**
 - Compare `arr[right] = 3` with `arr[largest] = 10`
 - $3 < 10$, so `largest` remains 1
4. **First Swap**
 - `arr[i] (4)` swaps with `arr[largest] (10)`
 - Updated array: `[10, 4, 3, 5, 1]`
 - **Total swaps = 1**

Recursive Call: `heapify(arr, 5, 1)`

1. **New Root ($i = 1$) = 4**
 - Left child ($left = 2 * 1 + 1 = 3$) = 5
 - Right child ($right = 2 * 1 + 2 = 4$) = 1
2. **Comparison**
 - Compare `arr[left] = 5` with `arr[largest] = 4`
 - $5 > 4$, so `largest = 3`
3. **Swap**
 - `arr[i] (4)` swaps with `arr[largest] (5)`
 - Updated array: `[10, 5, 3, 4, 1]`
 - **Total swaps = 2**
4. **Recursive Call: `heapify(arr, 5, 3)`**
 - $i = 3$ has **no children**, so recursion stops.

Problem 6

A company runs a **real-time task scheduling system** where multiple tasks are processed each minute. Each task has an associated **priority**, and the system must ensure that tasks are processed based on their urgency. New tasks arrive constantly, and the system must be able to **add**, **remove**, and **process tasks dynamically**. The system needs to decide how to efficiently manage the task queue to ensure that tasks are processed in the correct order, respecting their priorities.

Your goal is to design a system where:

1. Guarantees that each task operation (insertion, removal) happens in **$O(\log n)$** time
2. Tasks with **higher priority** are processed first.

Tasks:

1. **Describe how you would implement an efficient task scheduling system** that processes tasks based on their priority satisfying the above system goal.
2. **Simulate a scheduling process** with the following list of tasks, ensuring tasks are processed based on their priority:
 - tasks = [(3, 'Task A'), (5, 'Task B'), (2, 'Task C'), (4, 'Task D'), (1, 'Task E')]

Hint:

Consider using a **priority queue** implemented with a **max-heap**, where tasks are stored so that the highest-priority task is always easily accessible and processed first. This structure allows efficient addition, removal, and retrieval of tasks based on priority.

Answer

Designing an Efficient Task Scheduling System

To efficiently manage task scheduling where tasks with higher priority are processed first and operations (insertion, removal) occur in **$O(\log n)$** time, we can use a **Priority Queue** implemented with a **Binary Heap (Max-Heap)**. A Max-Heap ensures that the highest-priority task (largest priority value) is always at the top and can be efficiently extracted in **$O(\log n)$** time.

Implementation Details

- In a **max heap**, tasks are organized such that for every parent node, its priority is **greater than or equal** to the priorities of its child nodes.
- The heap structure ensures that the highest-priority task is always at the root.
- Inserting a new task takes **$O(\log n)$** time as the heap maintains its order by bubbling up the new task.
- Removing (processing) the highest-priority task also takes **$O(\log n)$** by swapping it with the last element and re-heapifying.

Heapify-Up (Insertion Process)

When inserting a new task (`priority, task_name`):

1. **Insert the task at the end** of the heap (to maintain the complete binary tree property).
2. **Compare the task with its parent**: If it has a higher priority, **swap it with the parent**.
3. **Repeat the process** until the task is in its correct position (either at the root or when its parent has a higher priority).
4. **Time Complexity: $O(\log n)$** since at most the height of the heap needs adjustments.

Heapify-Down (Processing/Removal Process)

When removing the highest-priority task (root node):

1. **Replace the root** with the last element in the heap.
2. **Compare the new root with its children**: Swap it with the highest-priority child if necessary.
3. **Repeat until the heap property is restored**, meaning the parent is always greater than both children.
4. **Time Complexity: $O(\log n)$** since the tree is restructured at most along its height.

Simulating Task Scheduling: Given tasks:

```
tasks = [(3, 'Task A'), (5, 'Task B'), (2, 'Task C'), (4, 'Task D'), (1, 'Task E')]
```

We insert them into a **Max-Heap**, where each task is represented as (`priority, task_name`). The heap maintains the order dynamically.

Heap Construction (Insertion Process)

1. Insert (3, 'Task A') → Heap: [(3, 'Task A')]
2. Insert (5, 'Task B') → Heap: [(5, 'Task B'), (3, 'Task A')]
3. Insert (2, 'Task C') → Heap: [(5, 'Task B'), (3, 'Task A'), (2, 'Task C')]
4. Insert (4, 'Task D') → Heap: [(5, 'Task B'), (4, 'Task D'), (2, 'Task C'), (3, 'Task A')]
5. Insert (1, 'Task E') → Heap: [(5, 'Task B'), (4, 'Task D'), (2, 'Task C'), (3, 'Task A'), (1, 'Task E')]

Processing Tasks in Priority Order

Using **heap extraction**, we process tasks in the order of highest priority:

1. **Extract (5, 'Task B')** → Heap after removal: [(4, 'Task D'), (3, 'Task A'), (2, 'Task C'), (1, 'Task E')]
2. **Extract (4, 'Task D')** → Heap after removal: [(3, 'Task A'), (1, 'Task E'), (2, 'Task C')]

3. **Extract (3, 'Task A')** → Heap after removal: [(2, 'Task C'), (1, 'Task E')]
4. **Extract (2, 'Task C')** → Heap after removal: [(1, 'Task E')]
5. **Extract (1, 'Task E')** → Heap is now empty.

Final Processing Order: Task B → Task D → Task A → Task C → Task E

Lab 5

Problem 1

You are given an array of integers `arr[]`, where the elements first increase, reach a peak, and then decrease. The peak element is defined as an element that is greater than or equal to its neighbors. The array guarantees that there is exactly one peak element. Your goal is to determine the index of the peak element in the array.

You should solve this problem using an efficient approach with time complexity $O(\log n)$. Justify your answer based on this input `[1, 2, 4, 5, 7, 8, 3]`

Answer

We will use a **binary search** approach, which allows us to find the peak element in $O(\log n)$ time complexity. Here's the reasoning behind the approach:

1. **Start by defining the middle element:** In a binary search, we begin by checking the middle element. If this element is greater than or equal to its neighbors, it is a peak.
2. **Check the neighbors:** If the middle element is smaller than its right neighbor, this indicates that the peak element lies in the right half of the array. Therefore, we move the search to the right half by updating the left boundary (`low = mid + 1`).
3. **Otherwise, check the left half:** If the middle element is smaller than its left neighbor, the peak element lies in the left half of the array. We then move the search to the left half by updating the right boundary (`high = mid - 1`).
4. **Repeat the process:** We keep halving the array until we find a peak element. Since we are reducing the search space by half with each step, the time complexity is $O(\log n)$.

Solution Steps:

- **Step 1:** Initialize the search range with `low = 0` and `high = n - 1` (where `n` is the length of the array).
- **Step 2:** Calculate the middle index `mid = (low + high) / 2`.
- **Step 3:** Compare the middle element with its neighbors:

- If `arr[mid] >= arr[mid - 1]` and `arr[mid] >= arr[mid + 1]`, we have found a peak element at index `mid`.
 - If `arr[mid] < arr[mid + 1]`, the peak must lie in the right half of the array, so update `low = mid + 1`.
 - If `arr[mid] < arr[mid - 1]`, the peak must lie in the left half of the array, so update `high = mid - 1`.
- **Step 4:** Continue adjusting the search range until a peak element is found.

Example Walkthrough (for input [1, 2, 4, 5, 7, 8, 3]):

- **Step 1:** Initial setup
`low = 0, high = 6` (size of array - 1).
 Calculate `mid = (0 + 6) / 2 = 3`.
 The element at index 3 is 5.
 - **Step 2:** Compare `arr[3] = 5` with its neighbors:
 - `arr[3] = 5` is less than `arr[4] = 7` and greater than `arr[2] = 4`.
 - Since `arr[3] < arr[4]`, the peak is likely in the right half. Therefore, update `low = mid + 1 = 4`.
 - **Step 3:** New range `low = 4, high = 6`
 Recalculate `mid = (4 + 6) / 2 = 5`.
 The element at index 5 is 8.
 - **Step 4:** Compare `arr[5] = 8` with its neighbors:
 - `arr[5] = 8` is greater than `arr[4] = 7` and `arr[6] = 3`.
 - Since `arr[5] >= arr[4]` and `arr[5] >= arr[6]`, 8 is a peak element, and its index is 5.
-

Problem 2

You are given two sorted arrays $a[]$ and $b[]$. Your task is to find and return the **median** of the combined array formed by merging $a[]$ and $b[]$.

The median is the middle value in an odd-length array, or the average of the two middle values in an even-length array.

You must solve this problem using an efficient approach with a time complexity of $O(\log(\min(n, m)))$, where n and m are the lengths of the two arrays.

Justify your answer based on these inputs $a[] = [1, 3, 8]$ and $b[] = [7, 9, 10, 11]$

Answer

We aim to solve this problem using a binary search approach on the smaller array, which gives us an efficient time complexity of $O(\log(\min(n, m)))$.

Explanation of the Approach:

1. Ensure $a[]$ is the Smaller Array:

- We apply binary search on the smaller array to minimize the search space, ensuring $O(\log(\min(n, m)))$ complexity.
- If the length of $a[]$ is greater than $b[]$, swap the arrays so that $a[]$ is always the smaller one.

2. Partitioning the Arrays:

- We want to partition both arrays such that:
 - The total number of elements in the left partition is half the combined total $((n + m + 1) // 2)$.
 - The elements on the left side of the partition should be less than or equal to the elements on the right side of the partition.

3. Finding the Partition Points:

- Define $mid1$ as the partition point in $a[]$ and $mid2$ as the partition point in $b[]$.
- To ensure the total number of elements on the left side is correct, we calculate $mid2$ as:
$$mid2 = ((n + m + 1) // 2) - mid1$$
- This formula for $mid2$ ensures that the total number of elements in the left partition is exactly half of the combined length (rounded up in case of an odd total number of elements).

4. Binary Search on the Smaller Array (`a[]`):

- We perform binary search on `a[]` by adjusting `mid1` and calculating `mid2` accordingly to ensure the left and right partitions are valid.
- For the partitions to be valid:
 - `a[mid1-1] <= b[mid2]` (ensures the left part of `a[]` is less than or equal to the right part of `b[]`).
 - `b[mid2-1] <= a[mid1]` (ensures the left part of `b[]` is less than or equal to the right part of `a[]`).

5. Calculating the Median:

- Once the correct partition is found, if the total number of elements is odd, the median is the maximum element in the left partition.
- If the total number of elements is even, the median is the average of the maximum element from the left partition and the minimum element from the right partition.

Example Walkthrough (for `a[] = [1, 3, 8]` and `b[] = [7, 9, 10, 11]`):

Step 1: Ensure `a[]` is the Smaller Array

- `a[]` has 3 elements, and `b[]` has 4 elements. So no need to swap them.

Step 2: Binary Search on `a[]`

- Set `lo = 0` and `hi = 3` (since `a[]` has 3 elements).
- Start the binary search.

Step 3: First Iteration of Binary Search

- Calculate `mid1 = (lo + hi) // 2 = 1`.
- Using the formula for `mid2`: `mid2 = ((n + m + 1) // 2) - mid1 = 3`
- Now, partition the arrays:
 - `a[mid1 - 1] = a[0] = 1`, `a[mid1] = a[1] = 3`.
 - `b[mid2 - 1] = b[2] = 10`, `b[mid2] = b[3] = 11`.

Step 4: Check Partition Validity

- To ensure the partition is valid:
 - We check that `a[mid1-1] <= b[mid2]` (i.e., `1 <= 11`), which is **true**.
 - We check that `b[mid2-1] <= a[mid1]` (i.e., `10 <= 3`), which is **false**.

Since the second condition is false, we need to adjust our partition by increasing `lo`.

Step 5: Second Iteration of Binary Search

- Update `lo = mid1 + 1 = 2`, and re-run the binary search.

Step 6: Second Iteration Calculation

- Now, `mid1 = (2 + 3) // 2 = 2`.
- Using the formula for `mid2`: `mid2 = 2`
- Now, partition the arrays:
 - `a[mid1 - 1] = a[1] = 3`, `a[mid1] = a[2] = 8`.
 - `b[mid2 - 1] = b[1] = 9`, `b[mid2] = b[2] = 10`.

Step 7: Check Partition Validity

- To ensure the partition is valid:
 - We check that `a[mid1-1] <= b[mid2]` (i.e., `3 <= 10`), which is **true**.
 - We check that `b[mid2-1] <= a[mid1]` (i.e., `9 <= 8`), which is **false**.

Again, we adjust `lo`.

Step 8: Final Iteration

- Update `lo = mid1 + 1 = 3`, and re-run the binary search.
- Now, `mid1 = (3 + 3) // 2 = 3`.
- Using the formula for `mid2`: `mid2 = 1`
- Now, partition the arrays:
 - `a[mid1 - 1] = a[2] = 8`, `a[mid1] = a[3] = +inf`.
 - `b[mid2 - 1] = b[0] = 7`, `b[mid2] = b[1] = 9`.
- `a[mid1] = a[3]`, but this would go out of bounds for `a[]` because `a[]` has only 3 elements. So, we treat `a[mid1]` as **positive infinity (+inf)**, meaning it is larger than any other element.

Step 9: Check Partition Validity

- To ensure the partition is valid:
 - We check that `a[mid1-1] <= b[mid2]` (i.e., `8 <= 9`), which is **true**.
 - We check that `b[mid2-1] <= a[mid1]` (i.e., `7 <= +inf`), which is **true**.

Step 10: Final Calculation

- In this case, the correct partition results in:

- The total number of elements in the combined array is 7 (odd), so the median is the maximum element on the left side of the partition: **Median= max(7,8) = 8**
-

Problem 3

You are given a hash table with **5 buckets** (indexed from 0 to 4).

Each bucket uses **separate chaining** to handle collisions (i.e., each bucket contains a linked list of records).

The **hash function** is defined as: $h(\text{key}) = \text{key} \% 5$

Each record contains a **student ID** and **name**.

Initial Insertions:

The following student records are inserted **in order**:

Student ID	Name
7	Alice
12	Bob
17	Carol
3	David
8	Eva

Tasks:

- 1) **Build the Hash Table**
 - a) Use the given hash function to place each student into the correct bucket.
 - b) For each insertion, **show the bucket index** and whether a **collision occurs**.
 - c) Draw the **final state** of the hash table, showing linked lists where collisions happen.
- 2) **Perform Operations** Using the final hash table:
 - a) **Search for student ID 12:**
 - Which bucket is accessed?
 - How is the student found?
 - b) **Insert a new student:** (ID = 22, Name = Frank)
 - Which bucket does this go to?
 - Does a collision occur?
 - Update the hash table accordingly.
 - c) **Delete student ID 17:**
 - Which bucket is accessed?
 - Explain how the record is removed from the linked list.
 - Show the updated state of the bucket.

Answer

1. Build the Hash Table

Insertion 1: ID = 7, Name = Alice

$$h(7) = 7 \% 5 = 2$$

- Bucket 2 is empty → Insert **Alice** at index 2.

Insertion 2: ID = 12, Name = Bob

$$h(12) = 12 \% 5 = 2$$

- Bucket 2 already contains Alice → **Collision!**
- Insert **Bob** into the linked list at index 2 (e.g., append after Alice).

Insertion 3: ID = 17, Name = Carol

$$h(17) = 17 \% 5 = 2$$

- Bucket 2 already has Alice → Bob → **Collision again!**
- Insert **Carol** at the end of the list in bucket 2.

Insertion 4: ID = 3, Name = David

$$h(3) = 3 \% 5 = 3$$

- Bucket 3 is empty → Insert **David** at index 3.

Insertion 5: ID = 8, Name = Eva

$$h(8) = 8 \% 5 = 3$$

- Bucket 3 has David → **Collision!**
- Insert **Eva** after David in the linked list at index 3.

Final Hash Table Structure:

Index Linked List (Chained Records)

0	(Empty)
1	(Empty)
2	Alice → Bob → Carol
3	David → Eva
4	(Empty)

2. Perform Operations

a) Search for Student ID = 12

- Compute hash: $h(12) = 12 \% 5 = 2$
- Go to bucket 2: Alice → Bob → Carol
- Traverse the linked list:
 - Check Alice (ID 7): Not a match

- Check Bob (ID 12): Match found

- **Result:** Student found → Name: **Bob**

b) Insert Student ID = 22, Name = Frank

- Compute hash: $h(22) = 22 \% 5 = 2$
- Bucket 2 already has: Alice → Bob → Carol → **Collision**
- Insert **Frank** at the end of the list in bucket 2

Updated Bucket 2: Alice → Bob → Carol → Frank

c) Delete Student ID = 17

- Compute hash: $h(17) = 17 \% 5 = 2$
- Traverse linked list in bucket 2:
Alice → Bob → Carol → Frank
- Locate Carol (ID 17)
- Remove Carol from the list by adjusting Bob's **next** pointer to skip Carol

Updated Bucket 2: Alice → Bob → Frank

Updated Hash Table Structure After All Operations:

Index Linked List (Chained Records)

0	(Empty)
1	(Empty)
2	Alice → Bob → Frank
3	David → Eva
4	(Empty)

Problem 4

You are tasked with selecting a **hash table collision resolution technique** for an application that stores **user IDs** as keys and **user information** as values. The two techniques you are considering are:

1. **Separate Chaining (Open Hashing)**: Uses linked lists for handling collisions.
2. **Open Addressing (Closed Hashing)**: Resolves collisions by searching for the next available slot.

Task:

1. **Efficiency Analysis**: Which technique is more efficient when the hash table is **sparsely populated** (low load factor) versus when it is **nearly full** (high load factor)? Discuss their performance in these two scenarios.
2. How does each technique perform in scenarios with **frequent deletions**? Consider how the deletion operation might affect **open addressing** (e.g., clustering issues).
3. If the application involves **frequent insertions**, which collision resolution technique would you recommend? Why?

Answer

Efficiency Analysis:

- **Sparsely Populated (Low Load Factor)**: When the hash table is sparsely populated, **open addressing** tends to be more efficient because there are fewer collisions, and most slots are empty. This results in fast insertions and lookups, as the probe sequence for finding an available slot is usually short. **Separate chaining**, on the other hand, requires more memory because each bucket stores a linked list, which could lead to increased memory overhead.
- **Nearly Full (High Load Factor)**: When the table is near full, **separate chaining** performs better. Open addressing suffers from increased clustering (especially with linear probing), which significantly degrades performance as the load factor grows. Separate chaining handles high load factors more gracefully since linked lists can grow dynamically within each bucket without the need for resizing or probing. This results in more stable performance, even as the table becomes nearly full.

Frequent Deletions:

- **Open Addressing:** Deletion in open addressing is more complex because after a deletion, there might be gaps in the probe sequence. This can disrupt the search for future elements, especially in **linear probing**, leading to **clustering**. While this can be mitigated by using **double hashing** or **quadratic probing**, the performance still degrades as deletions can cause additional probing and rehashing.
- **Separate Chaining:** Deletion is straightforward in separate chaining because elements are stored in linked lists, and a deletion simply removes an element from the list at the corresponding bucket. However, if many deletions occur, the linked lists could become sparse, and managing the memory for empty slots may require additional cleanup operations. Overall, separate chaining tends to handle deletions more efficiently than open addressing.

Recommendation for Frequent Insertions:

- For **frequent insertions**, **separate chaining** is recommended. This is because, with open addressing, frequent insertions can lead to a **high load factor**, causing clustering issues that degrade performance, especially as the table fills up. Separate chaining allows for more flexibility in handling insertions because the linked lists can grow dynamically, and each bucket can hold multiple values without affecting other elements. Additionally, separate chaining does not suffer from clustering, which can be a significant issue with open addressing when insertion rates are high.

Problem 5

Imagine you're part of a company's performance evaluation team. The organization is structured hierarchically like a **binary tree**, where:

- Each **node** represents an **employee**, including team leads and individual contributors.
- Each **employee node** contains a **performance score** for a specific quarter.
- A **leaf node** is an individual contributor with no subordinates.
- **Internal nodes** (team leads or managers) manage up to two direct reports (left and right child in the tree).

Now, to **validate fair performance reporting**, the company defines a rule:

"Each team lead's performance score must equal the **total performance score** of their **entire team** (i.e., the sum of their direct subordinates' performance scores and all levels below)."

Such a tree is called a **"Sum Tree"**.

Task:

Your job is to **verify** whether the company's hierarchy satisfies the **Sum Tree property**, i.e., whether the performance score of every team lead equals the **sum of the scores of all their subordinates**.

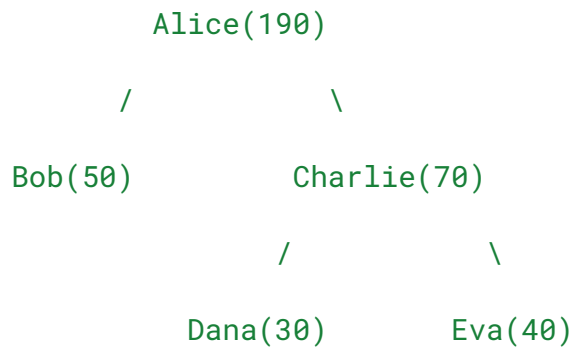
You must describe:

- The **algorithm runs in $O(n)$ time complexity** you'd use to validate the Sum Tree property.
- A **step-by-step walkthrough** using the provided example tree.

Note:

- A leaf employee (individual contributor) is considered valid by default.
- An empty team (no employees) is considered to contribute 0 to the total.

Example Hierarchy (Binary Tree):



Expected Output: Yes, this is a valid Sum Tree.

Answer

1. Algorithm Explanation:

We use a **bottom-up (post-order) recursive approach** to check whether the binary tree is a Sum Tree in $O(n)$ time:

Key Ideas:

1. **Leaf Nodes** (individual contributors) are **always valid** Sum Trees.
2. **Empty Nodes** (no employee/subtree) contribute **0** to the total sum.

3. For any **non-leaf node** (manager):
 - First, **recursively check** if the left and right subtrees are Sum Trees.
 - If either is not, the whole tree isn't.
4. If both subtrees are valid:
 - We can **calculate the total sum in O(1)** using the following:
 - If a child is a leaf → use its **value**
 - If a child is an internal node and a Sum Tree → its subtree sum is **2 × node value** (because child's value = sum of its subordinates)
5. Check if **node.value == left_sum + right_sum**. If yes, this node is valid.

Why it's O(n):

- Every node is visited **exactly once**, and at each step, we do constant-time checks and calculations.
- No repeated subtree sum computations.

2. Step-by-Step Process:

Step 1: Start with Leaf Nodes

- **Dana(30)** and **Eva(40)** are leaves → **Valid Sum Trees**
 - Subtree sum of Dana = 30
 - Subtree sum of Eva = 40

Step 2: Charlie(70)

- Charlie is not a leaf.
- Left = Dana (leaf) → contributes 30
- Right = Eva (leaf) → contributes 40
- Expected: **Charlie.value = 30 + 40 = 70** → **Valid**

- Since both children are leaves, their values are used directly.
- **Subtree sum under Charlie = $70 + 30 + 40 = 140$** , but for the algorithm we return only **70** to its parent, because Charlie is treated as one unit now (since it passed the check).

Step 3: Bob(50)

- Bob is a leaf (no children) → **Valid Sum Tree**
- Subtree sum under Bob = 50

Step 4: Alice(120)

- Left = Bob (leaf) → contributes 50
- Right = Charlie (a non-leaf but confirmed Sum Tree) → use $2 \times \text{Charlie.value} = 2 \times 70 = 140$
- Check: $\text{Alice.value} = 50 + 140 = 190 \rightarrow \text{Valid}$

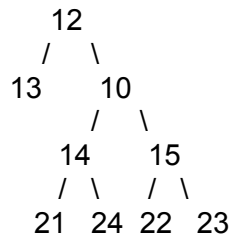
Final Output: Yes, this is a valid Sum Tree.

Problem 6

We have a network of devices represented by a binary tree, where each device is connected to others, with parent-child relationships indicating direct connections. When a **failure** (such as a network outage, malfunction, or device failure) **occurs at a particular device** (the target node), **the network can no longer reach the failed device or its directly connected devices**. Our goal is to determine the order in which devices fail as the failure progresses throughout the network. By determining this order, we can understand how the failure will affect connectivity, identify critical devices whose failure could cause widespread disruption, and prioritize actions to prevent further failures or mitigate their impact on the overall network.

1. **Describe the Algorithm** you would use to simulate the failure spreading through the binary tree.
2. **Justify your approach** by showing the steps with the following example.

Example Tree:



Target Node = 14

Expected Output:

14

21, 24, 10

15, 12

22, 23, 13

Answer

Algorithm Explanation:

To simulate the failure spreading in the binary tree, follow these steps:

1. Find the Target Node:

- First, we need to locate the target node in the tree. We can use **Depth-First Search (DFS)** or **Breadth-First Search (BFS)** to traverse the tree and find the node where the failure will start.
- We will use **DFS** for this explanation.

2. Failure Spreads to Connected Nodes:

- Once the target node is found, print the target node (this is the first failed device).
- The failure spreads to all directly connected nodes. These are the **parent** (if the target node is not the root) and **children** of the current node.

3. Use a Queue for Level-by-Level Spread:

- We will use a **queue** to manage the spread of failure. After the initial failure starts at the target node, we add all its neighbors (children and parent) to the queue.
- For each subsequent level, we process the queue, print the devices that fail simultaneously, and add their children (if any) to the queue for the next step.

4. Repeat the Process:

- This process continues until all devices in the network have failed, i.e., until the queue is empty.

Step-by-Step Process with Example:

Step 1: Find the Target Node (14)

DFS Traversal:

- We perform a **DFS** traversal to find the target node. Start at the root (12), traverse down to the left child (13), and then move to the right child (10).
- From node **10**, we move down to node **14**, which is the target.
- Once we find node **14**, we print it as it fails first.

Queue state after the failure starts at 14:

- We add **14's** neighbors (parent and children) to the queue.
 - Left Child (21)
 - Right Child (24)
 - Parent (10)

Queue: [21, 24, 10]

Output so far: 14

Step 2: Spread Failure to Neighbors (Level 1)

Failure Spreads to Neighbors:

- The failure spreads to **21, 24, and 10** in the first step.
- These nodes fail simultaneously, so we print them all.
- After spreading, we add their connected neighbors to the queue (if they exist).

Processing the queue:

- **Node 21** has no children, so no new nodes are added to the queue.
- **Node 24** has no children, so no new nodes are added to the queue.
- **Node 10** has **children 15** and **parent 12**.
- We add **15** and **12** to the queue.

Queue state after Level 1:

- **Queue:** [15, 12]

Output so far: 14, 21, 24, 10

Step 3: Spread Failure to Neighbors (Level 2)

Failure Spreads to Neighbors of 15 and 12:

- The failure spreads to **15** and **12** in the next level.
- We print them simultaneously.
- After spreading, we add their connected neighbors to the queue (if they exist).

Processing the queue:

- **Node 15** has **children 22, 23**.
 - Add **22** and **23** to the queue.
- **Node 12** has no children but has **parent 13**.
 - Add **13** to the queue.

Queue state after Level 2:

- **Queue:** [22, 23, 13]

Output so far: 14, 21, 24, 10, 15, 12

Step 4: Spread Failure to Neighbors (Level 3)

Failure Spreads to Neighbors of 22, 23, and 13:

- The failure spreads to **22, 23, and 13**.
- We print these nodes simultaneously.
- After spreading, we add any connected neighbors to the queue.

Processing the queue:

- **Node 22** has no children.
- **Node 23** has no children.
- **Node 13** has no children, as it is a leaf node.

Queue state after Level 3:

- The queue is empty.

Output so far: 14, 21, 24, 10, 15, 12, 22, 23, 13

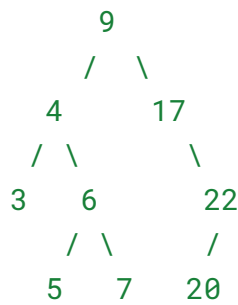
Lab 6

Problem 1

Consider an e-commerce platform where the product prices are organized in a Binary Search Tree (BST). Each node represents a product's price, and the tree is structured based on the prices. As an e-commerce system administrator, your task is to quickly find the product with the price closest to a customer's target price.

Example:

Given the following BST of product prices:



You need to implement an algorithm that finds the product with the minimum absolute price difference to the given target price K .

1. **Input:** Target price $K = 4$
Output: Closest price = 4
2. **Input:** Target price $K = 18$
Output: Closest price = 17
3. **Input:** Target price $K = 2$
Output: Closest price = 3

Task:

1. Describe an approach to solve this problem with $O(h)$ time complexity, where h is the height of the tree.
2. Walk through the algorithm step by step for the example with the target price $K = 19$ and explain the comparisons made in the tree to find the closest price.

Answer

Approach to Solve the Problem in $O(h)$ Time Complexity:

To find the closest price, we need to traverse the tree and compare the target price K with the node's price at each step. At each node, we will track the closest value found so far. Since the BST structure helps us eliminate half of the tree in each comparison, this approach works in $O(h)$ time, where h is the height of the tree.

Step-by-Step Algorithm:

1. **Start with the root node.** Initialize a variable `closest` to store the closest value to the target price found so far. Initially, set `closest` to the root's value.
2. **Traverse the tree.**
 - While the current node is not `null`, compare the absolute difference between the target price K and the current node's price with the absolute difference between K and the closest price found so far.
 - If the current node's value is closer to K than the `closest` value, update `closest` to the current node's value.
3. **Move to the left or right subtree.**
 - If the current node's price is greater than K , move to the left child, as the left subtree contains smaller values.
 - If the current node's price is smaller than K , move to the right child, as the right subtree contains larger values.
 - If the current node's price is equal to the target price K , stop the search immediately as we have found the exact match.
4. **Repeat the process** until we reach a `null` node or find a node with an exact match (i.e., the price equals K).
5. **Return the closest value** found after completing the traversal.

Example Walkthrough: Target Price $K = 19$

Step 1:

- We start at the root, which has a price of 9 .
- The absolute difference between 9 and 19 is 10 , so we initialize `closest = 9`.

Step 2:

- Since $19 > 9$, we move to the right child (value 17).
- The absolute difference between 17 and 19 is 2, which is smaller than the previous difference of 10. We update `closest = 17`.

Step 3:

- Since $19 > 17$, we move to the right child (value 22).
- The absolute difference between 22 and 19 is 3, which is larger than the previous difference of 2, so we do not update `closest`.

Step 4:

- Since $19 < 22$, we move to the left child (value 20).
- The absolute difference between 20 and 19 is 1, which is smaller than the previous difference of 2. We update `closest = 20`.

Step 5:

- Since $19 < 20$, we move to the left child, which is `null`. The traversal stops here.

Step 6:

- The closest price found is 20.

Problem 2

A financial auditing system is analyzing a company's past transactions. Each transaction amount is stored as a node in a **Binary Search Tree (BST)**, where the BST structure ensures that the left child of a node contains a smaller amount, and the right child contains a larger amount.

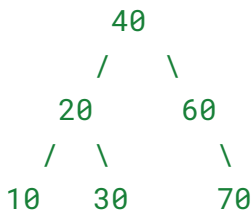
As part of a fraud detection protocol, the auditor wants to know: **Are there two transaction amounts in the BST that add up exactly to a suspicious target amount?**

Task:

Given the root of a Binary Search Tree containing **distinct positive integers** (transaction amounts) and a target sum (suspicious amount), determine whether **there exists a pair of nodes in the BST** whose sum equals the target.

1. **Explain an efficient algorithm** to solve this problem in $O(n)$ time using BST properties.
2. **Step-by-step walkthrough:** Use the BST below and target = 90. Show how the algorithm works.

Example tree:



Answer

Algorithm Approach

1. Inorder Traversal of the BST:

- Since the BST follows the property where the left subtree contains smaller values and the right subtree contains larger values, an **inorder traversal** will give us the values of the tree in **ascending order**.
- This gives us a sorted array of transaction amounts, and we can use the **two-pointer technique** to find two values whose sum is equal to the target.

2. Two-Pointer Technique on Sorted Array:

- Once we have the sorted array (from the inorder traversal), we can use two pointers: one starting at the beginning of the array and the other starting at the end.
- The two-pointer technique works because:
 - If the sum of the two pointers is **less than the target**, we need a larger sum, so we move the left pointer to the right (increase it).
 - If the sum is **greater than the target**, we need a smaller sum, so we move the right pointer to the left (decrease it).
 - If the sum equals the target, we have found the two values that sum to the target and can return **True**.

3. Algorithm Complexity:

- **Time Complexity: $O(n)$** , where **n** is the number of nodes in the BST. This is because:
 - We perform **inorder traversal** once, which takes **$O(n)$** time.
 - The **two-pointer technique** also runs in **$O(n)$** time on the sorted array.

Step-by-Step Walkthrough Using the Given BST and Target = 90:

Step 1: Perform Inorder Traversal

We start by performing an inorder traversal of the BST, which will give us the node values in sorted order.

- **Inorder traversal steps:**

- Traverse left subtree of root (40), which is node 20.
- Traverse left subtree of node 20, which is node 10.
- Visit node 10.
- Visit node 20.
- Traverse right subtree of node 20, which is node 30.
- Visit node 30.
- Visit root node 40.
- Traverse right subtree of root node 40, which is node 60.
- Visit node 60.
- Traverse right subtree of node 60, which is node 70.
- Visit node 70.

Result of Inorder Traversal:

The sorted array obtained is:

[10, 20, 30, 40, 60, 70]

Step 2: Use Two-Pointer Technique to Find the Pair

Now that we have the sorted array [10, 20, 30, 40, 60, 70], we can apply the two-pointer technique to find if there exists a pair whose sum is **90**.

- **Initial Pointers:**

1. **left = 0** (points to 10)
2. **right = 5** (points to 70)

- **Step-by-step Process:**

1. **Check the sum of left and right:**

- `currentSum = 10 + 70 = 80`
- Since `80 < 90`, move the `left` pointer to the right (`left = 1`).

2. Check the sum of `left` and `right`:

- `currentSum = 20 + 70 = 90`
 - **Match found!** The sum of `20` and `70` equals `90`. We return `True`.
-

Problem 3

Imagine an e-commerce platform that manages a large and dynamic product catalog where the prices of the products are frequently updated (products are added or removed). Initially, the platform stores these prices in a **sorted array** for efficient querying of individual product prices using **binary search** in $O(\log n)$ time. While the sorted array is efficient for searching individual prices, the platform faces performance challenges with **frequent insertions and deletions**. Specifically, maintaining the sorted order during each update operation (insertion or deletion) in a sorted array requires shifting elements, leading to $O(n)$ time complexity for each operation.

To optimize the performance of the platform's product catalog, the platform needs to **convert the sorted array into a Balanced Binary Search Tree (BST)**. This will allow the platform to perform **insertions and deletions** efficiently in $O(\log n)$ time, while maintaining fast querying capabilities for product prices.

Task:

1. Explain an efficient algorithm to convert a sorted array into a Balanced Binary Search Tree in $O(n)$ time.
2. **Step-by-step walkthrough:** Show how the algorithm works with the array [10, 20, 30, 40, 50, 60, 70]

Answer

Approach:

The key idea is to use **recursion** to traverse the array and create a BST:

1. **Find the Middle Element:** The middle element of the array is selected as the root node. This guarantees that the tree will be balanced, as it divides the array into two roughly equal halves.
2. **Recursively Build the Left and Right Subtrees:**
 - The left half of the array becomes the left subtree of the root.

- The right half of the array becomes the right subtree of the root.
3. **Base Case:** The recursion stops when the range of the array becomes invalid (i.e., when `start > end`), indicating that no more nodes need to be created.

Steps:

1. **Set the middle element** of the array as the root.
2. **Recursively apply the same operation** to the left half of the array to create the left child of the root.
3. **Recursively apply the same operation** to the right half of the array to create the right child of the root.

Step-by-Step Walkthrough Using the Array `[10, 20, 30, 40, 50, 60, 70]`:

Step 1: Initial Array:

`[10, 20, 30, 40, 50, 60, 70]`

- The **middle element** is `40` (index 3). This becomes the root of the tree.

Tree so far:

`40`

Step 2: Create Left Subtree:

- The left half of the array is `[10, 20, 30]`.
- The **middle element** of this subarray is `20` (index 1), so `20` becomes the left child of `40`.

Tree so far:

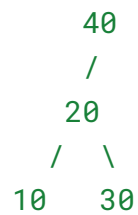
```

  40
 /
20

```

- Now, the left half of `[10, 20, 30]` is `[10]`, and the middle element is `10`. So, `10` becomes the left child of `20`.
- The right half of `[10, 20, 30]` is `[30]`, and the middle element is `30`. So, `30` becomes the right child of `20`.

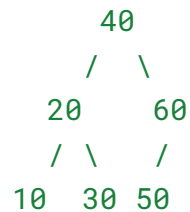
Tree so far:



Step 3: Create Right Subtree:

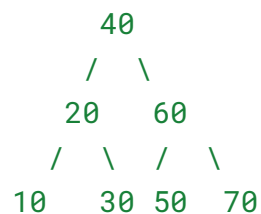
- The right half of the array is [50, 60, 70].
- The **middle element** is 60 (index 5), so 60 becomes the right child of 40.

Tree so far:



- The left half of [50, 60, 70] is [50], and the middle element is 50. So, 50 becomes the left child of 60.
- The right half of [50, 60, 70] is [70], and the middle element is 70. So, 70 becomes the right child for 60.

Final Tree:

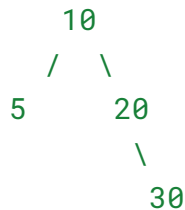


<https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/>

Problem 4

In an e-commerce platform, product listings are managed using an **AVL tree** based on product **ID**. The tree must remain balanced after every **insertion** and **deletion** to ensure efficient searching and updating.

The initial AVL tree is shown below:



Tasks:

- Given the following sequence of insertions:
 - Insert Product ID = 25**
 - Insert Product ID = 35**

After each insertion, analyze the AVL tree balance and identify any rotations needed. Explain the reason for the rotation (left/right, single/double)

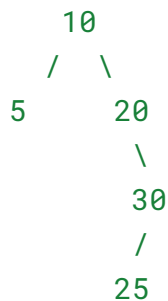
- Now, delete the following products:
 - Delete Product ID = 35**
 - Delete Product ID = 30**

After each deletion, analyze the tree's balance and identify any required rotations. Justify the rotation based on the AVL property violated.

Answer

Insert Product ID = 25

- Insert **25** as the left child of **30**:



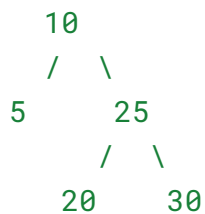
2. Balance Factor Analysis:

- Node **30**: Left subtree height = 1, Right subtree height = 0 → **Balance Factor = +1** (Balanced)
- Node **20**: Left subtree height = 0, Right subtree height = 2 → **Balance Factor = -2** (Unbalanced!)

3. Rotation Required:

- This is a **Right-Left (RL)** case because **25** was inserted into the left subtree of **30**, which is the right child of **20**.
- **Step 1**: Perform **Right Rotation (R)** on **30**.
- **Step 2**: Perform **Left Rotation (L)** on **20**.

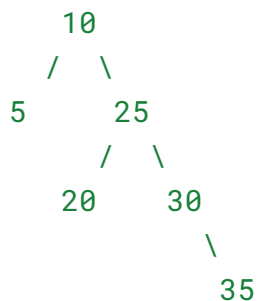
4. After RL Rotation:



Now, the tree is balanced.

Insert Product ID = 35

1. Insert **35** as the right child of **30**:



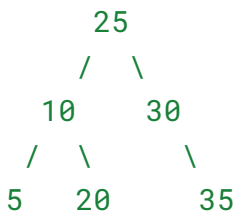
2. Balance Factor Analysis:

- Node **30**: Left subtree height = 0, Right subtree height = 1 → **Balance Factor = -1** (Balanced)
- Node **25**: Left subtree height = 1, Right subtree height = 2 → **Balance Factor = -1** (Balanced)
- **Node 10**: Left subtree height = 1, Right subtree height = 3 → **Balance Factor = -2** (Unbalanced!)

3. Rotation Required:

- This is a **Right-Right (RR) case** because **35** was inserted into the right subtree of **25**, which is the right subtree of **10**.
- **Perform Left Rotation (L) on 10.**

4. After Left Rotation (L) on 10:

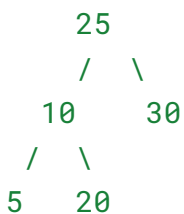


Now, the tree is balanced.

Deletion Operations:

Delete Product ID = 35

1. **Remove 35** (Leaf node):



2. **Balance Factor Analysis:**

- Node **30**: Left subtree height = 0, Right subtree height = 0 → **Balance Factor = 0** (Balanced)
- Node **25**: Left subtree height = 2, Right subtree height = 1 → **Balance Factor = +1** (Balanced)

Since the tree remains balanced, **no rotations are required**.

Delete Product ID = 30

1. **Remove 30**, leading to:



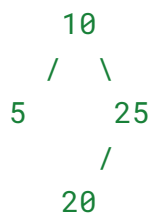
2. **Balance Factor Analysis:**

- Node **25**: Left subtree height = 2, Right subtree height = 0 → **Balance Factor = +2 (Unbalanced!)**
- Node **10**: Left subtree height = 1, Right subtree height = 1 → **Balance Factor = 0** (Balanced)

3. **Rotation Required:**

- This is a **Left-Left (LL) case** because **10** is in the left subtree of **25**, and the imbalance is caused by the left subtree being too tall.
- **Perform Right Rotation (R) on 25.**

4. **After Right Rotation (R) on 25:**



Now, the tree is balanced.

Problem 5

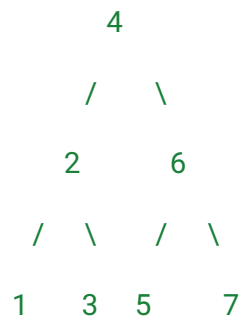
Consider an e-commerce platform that stores product data in a **Binary Search Tree (BST)** based on product prices. To optimize **product recommendations** for the **highest-priced products**, the platform plans to **convert the BST into a Special Max Heap**. This conversion allows for **efficient retrieval of top-priced products** by maintaining the **Max Heap property**, where the largest element is always at the root.

The platform also wishes to maintain the **BST property** within each subtree, ensuring that all values in the left subtree are smaller than those in the right subtree. This **Special Max Heap** structure balances **fast access to the highest-priced products** while preserving an **ordered structure** for efficient searching and querying within the tree.

Task:

1. Explain an efficient algorithm to convert a Binary Search Tree (BST) into a **Special Max Heap** in $O(n)$ time.
2. **Step-by-step walkthrough:** Show how the algorithm works with the below BST

Example BST:



Answer

Algorithm Approach to Convert a BST into a Special Max Heap

The process of converting a **Binary Search Tree (BST)** into a **Special Max Heap** follows a structured approach to ensure that the resulting tree satisfies both:

1. The **Max Heap property**, where each parent node is greater than its children.
2. The **BST structure in its subtrees**, where the left child is smaller than the right child.

Step 1: Perform Inorder Traversal to Store BST Elements

- **Inorder traversal (Left → Root → Right)** is performed on the BST.

- This traversal visits nodes in ascending order and stores their values in an auxiliary array (`arr[]`).
- The array `arr[]` now contains all BST elements in sorted order.
- This ensures that when we later assign values back to the BST, we preserve the order needed for the heap.

Step 2: Perform Postorder Traversal to Reassign Values

- **Postorder traversal (Left → Right → Root)** is used to modify the BST while maintaining heap properties.
- During this traversal, each node is assigned a value from `arr[]`, starting from the smallest to the largest.
- The traversal order ensures that child nodes are processed **before** their parent, helping maintain the heap property.

Step 3: Preserve the Heap Structure

- By using postorder traversal, values are assigned such that every parent node has a greater value than its children.
- Since `arr[]` contains sorted values, assigning them sequentially during postorder traversal ensures that:
 - Parent nodes always receive larger values than their children.
 - The resulting tree maintains the **Max Heap property**.

Step 4: Maintain BST Subtree Order

- While converting the BST into a Max Heap, the tree structure itself is not altered.
- This means that in each subtree, the left child remains smaller than the right child, thus preserving the BST property within its subtrees.

Step-by-Step Walkthrough

Step 1: Perform Inorder Traversal

Inorder traversal of BST yields a **sorted array**: `arr = [1, 2, 3, 4, 5, 6, 7]`

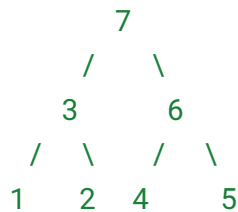
Step 2: Modify BST in Postorder Traversal

Now, we traverse the tree in **postorder (left → right → root)** and replace values using `arr[]` in order.

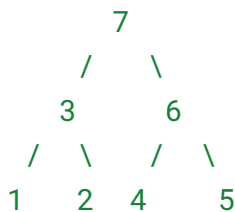
1. Visit left subtree (**1 → 3 → 2**)
2. Visit right subtree (**5 → 7 → 6**)

3. Assign values to the nodes:
Postorder traversal order: [1, 3, 2, 5, 7, 6, 4]

Replace nodes with values from arr[]:



Final Max Heap:



Now, the tree satisfies **Max Heap** conditions:

- Every parent is greater than its children.
- The original **BST structure** is maintained within subtrees.

Problem 6

In a **customer support center**, each incoming request is labeled with a **type**, represented by lowercase letters (e.g., 'b' for billing, 't' for technical). Some request types occur more frequently than others. To **reduce operator fatigue** and **optimize efficiency**, the system must be configured so that **no two requests of the same type are handled consecutively**.

Given a string **s**, where each character represents a request type, your task is to **rearrange the characters** so that **no two adjacent characters are the same**. If such an arrangement is **not possible**, return an **empty string** ("").

To ensure performance, your algorithm must run in **O(n log n)** time, where **n** is the length of the string.

Example

Input: $s = \text{"aaabbc"}$

Output: "ababac"

Tasks:

1. **Describe an algorithm** to solve this problem that runs in $O(n \log n)$ time.
2. **Demonstrate your approach** with a step-by-step explanation for the input $s = \text{"aaabbc"}$.

Hint: Think about how to always select the most frequent request types first, while temporarily holding back recently used types.

Answer

Main Idea:

Always choose the character with the **highest frequency** (greedy choice), but ensure that it's **not the same** as the one added in the previous step.

Steps:

1. **Count the frequency** of each character in the string.
2. **Build a max heap** (priority queue) where each element is a pair $(-frequency, character)$.
 - We use negative frequency because Python's `heapq` is a **min-heap**, and we need **max-heap behavior**.
3. **Initialize:**
 - An empty result list `res`.
 - A variable `prev` to store the character used in the previous step (with remaining frequency) so that it can be re-added later.
4. **Loop:**
 - While the heap is not empty:
 - Pop the character with the **highest frequency** (call it `curr`).
 - Append it to the result.
 - If `prev` still has remaining frequency, push it back into the heap.
 - Update `prev` to be the current character, with frequency decreased by 1.
5. After the loop, if the length of the result is **equal to the input**, return the joined result. Otherwise, return `""` (not possible).

Step-by-Step Example:

Let's take `s = "aaabbc"`.

Step 1: Count frequencies

`a → 3`

`b → 2`

`c → 1`

Step 2: Build max heap

We store frequencies as negatives for max-heap behavior:

`[(-3, 'a'), (-2, 'b'), (-1, 'c')]`

Step 3: Initialize

- `res = []`
- `prev = (0, '')` → No previous character initially

Iteration 1:

- Pop `(-3, 'a')` → Most frequent character
- Append `'a'` to `res`: → `res = ['a']`
- `prev = (-2, 'a')` → Decrease frequency of `'a'` by 1
- Nothing to push back yet (prev count was 0)

Heap: `[(-2, 'b'), (-1, 'c')]`

Iteration 2:

- Pop `(-2, 'b')`
- Append `'b'` to `res`: → `res = ['a', 'b']`

- Push back `prev = (-2, 'a')` → heap becomes `[(-2, 'a'), (-1, 'c')]`
- `prev = (-1, 'b')`

Heap: `[(-2, 'a'), (-1, 'c')]`

Iteration 3:

- Pop `(-2, 'a')`
- Append 'a' to `res`: → `res = ['a', 'b', 'a']`
- Push back `prev = (-1, 'b')` → heap becomes `[(-1, 'b'), (-1, 'c')]`
- `prev = (-1, 'a')`

Heap: `[(-1, 'b'), (-1, 'c')]`

Iteration 4:

- Pop `(-1, 'b')`
- Append 'b' to `res`: → `res = ['a', 'b', 'a', 'b']`
- Push back `prev = (-1, 'a')` → heap becomes `[(-1, 'a'), (-1, 'c')]`
- `prev = (0, 'b')` → No more `b` left

Heap: `[(-1, 'a'), (-1, 'c')]`

Iteration 5:

- Pop `(-1, 'a')`
- Append 'a' to `res`: → `res = ['a', 'b', 'a', 'b', 'a']`
- `prev = (-1, 'c')` → Save to reinsert in next step

Heap: `[(-1, 'c')]`

Iteration 6:

- Pop `(-1, 'c')`
- Append `'c'` to `res`: $\rightarrow res = ['a', 'b', 'a', 'b', 'a', 'c']$
- `prev = (0, 'c')` \rightarrow No more `c`

Heap is now empty.

Output: `"ababac"`

Lab 7&8

Problem 1

Alex, a transportation planner at a city transport company, is tasked with reviewing the city's subway network to ensure it's designed as a tree. A tree is a connected graph with no cycles and exactly $N-1$ edges, where N is the number of stations (vertices). Each station in the subway system is connected to other stations by specific routes (edges).

The planning department has provided Alex with the number of stations (N) and the number of routes each station connects to (degree). Alex needs to determine whether the subway network can be considered a tree based on this information.

Input:

- The number of stations, N , in the subway network.
- A list of N integers, where each integer represents the number of routes (degree) each station is connected to.

Output:

Print "Yes" if the subway system forms a tree, otherwise print "No".

Questions:

1. **Describe the algorithm approach to solve this problem.**
(Provide a step-by-step explanation of how Alex can determine if the subway system forms a tree based on the number of stations and the degree of each station.)
2. **Consider the following inputs:**
 - a. $N=4$, Degrees = [1, 2, 1, 2]
 - b. $N=5$, Degrees = [2, 3, 2, 2, 3]

Do these subway networks represent a tree? Explain your reasoning.

Answer

A tree in graph theory has two main properties:

1. **It is connected:** All stations (vertices) are reachable from any other station.
2. **It has no cycles:** There are no loops or redundant routes in the system.
3. **It has exactly $N-1$ edges:** This is crucial. For a graph to be a tree, the number of edges (routes) must always be one less than the number of stations, because any more edges would create a cycle, and fewer would mean some stations are disconnected.

Given that we are only provided with the degree (number of edges connected to each station), we can check if these conditions hold true for the system. Specifically:

- The sum of the degrees of all stations should be **exactly twice the number of edges** in the graph (since each edge connects two stations).
- There should be **exactly $N-1$ edges** in total.
- If the total number of edges is less than $N-1$, we know that the graph is disconnected, so it's not a tree.
- If the total number of edges is more than $N-1$, we know that there's a cycle in the graph, which violates the acyclic property of trees.

With this in mind, the task becomes checking these two properties using the given degree information to determine if the system is a tree.

Algorithm Approach

1. Check if the number of edges is correct:

- First, compute the total sum of the degrees of all stations.
Each edge connects two stations, so the sum of the degrees is always twice the number of edges in the graph. That is, if the sum of the degrees is S , then the number of edges E should be $E=S/2$
- If S is odd, the graph cannot form a valid structure, because the sum of degrees should always be even (since each edge contributes to two degrees).

2. Check if the graph has exactly $N-1$ edges:

- After finding the total number of edges, check if it equals $N-1$. If it does, then the graph could potentially be a tree (i.e., it's a connected acyclic graph with the right number of edges). If not, the graph cannot be a tree.

3. Final check:

- If the sum of the degrees is even and the number of edges is $N-1$, then the graph is a tree. If either of these conditions fails, then the graph is not a tree.

Step-by-Step Algorithm

1. Input:

- Let N be the number of stations.
- Let **degrees** be the list of degrees of all N stations.

2. Calculate the sum of degrees:

- Let **sum_of_degrees = sum(degrees)**.

- If `sum_of_degrees` is odd, print "No" because a valid graph cannot have an odd sum of degrees.

3. Calculate the number of edges:

- Let $E = \text{sum_of_degrees} / 2$. This is the number of edges in the graph.
- If E is not equal to $N-1$, print "No" because the graph doesn't have the correct number of edges for a tree.

4. Output:

- If both checks pass, print "Yes", as the graph forms a tree.
- Otherwise, print "No".

Example Walkthrough:

Example 1

4

1 2 1 2

- Sum of degrees = $1 + 2 + 1 + 2 = 6$
- Number of edges = $6/2=3$
- Expected number of edges for a tree with 4 vertices = $4-1=3$

Since the number of edges is correct, and the sum of degrees is even, the graph satisfies the conditions for being a tree. **The output is "Yes".**

Example 2

5

2 3 2 2 3

- Sum of degrees = $2 + 3 + 2 + 2 + 3 = 12$
- Number of edges = $12/2=6$
- Expected number of edges for a tree with 5 vertices = $5-1=4$

The number of edges is incorrect (6 instead of 4), so **the output is "No"**.

Problem 2

Sophie, a marine biologist, has arrived at an archipelago of N islands. Each island is connected to one or more others by bidirectional bridges. Sophie is currently at Island #1 and needs to

reach Island #N to conduct her research. However, she wants to minimize the number of bridges she has to cross because the bridges are old and require a lot of effort to cross.

Sophie is wondering how many bridges she will need to cross to get to Island #N, taking the optimal route. Can you help Sophie find the minimum number of bridges she must cross to reach her destination?

You are given:

- The total number of islands N and the total number of bridges connecting them.
- A list of connections between the islands in the form of bidirectional bridges.
- The index of the starting island, which is always Island #1.

You need to determine how many bridges Sophie will cross on the shortest route, or if it's even possible to reach Island #N.

Questions:

1. Describe the approach Sophie should use to find the minimum number of bridges to cross.
2. Given the following connections between the islands:
 - Islands: 6
 - Bridges: 7
 - Connections: 1-2, 2-3, 3-4, 1-4, 5-4, 6-5, 4-6
 - Sophie starts at Island 1.

What is the optimal route Sophie should take to reach Island #6, and how many bridges will she cross? Explain your reasoning.

Answer

Question 1: Describe the approach Sophie should use to find the minimum number of bridges to cross.

To find the **minimum number of bridges** Sophie needs to cross, we can model the islands and their bridges as a **graph**. Here:

- **Islands** are represented as **nodes**.
- **Bridges** are represented as **edges** connecting these nodes.

The problem Sophie faces is essentially a **shortest path problem** in an unweighted graph, where the goal is to find the **minimum number of edges** (bridges) between the starting node (Island #1) and the destination node (Island #N).

To solve this efficiently, we use **Breadth-First Search (BFS)**. Here's why BFS is a suitable algorithm for this problem:

1. **Unweighted Graph:** Each bridge is considered to have the same weight (effort), so the shortest path in terms of the **number of bridges** is equivalent to finding the path with the fewest edges.
2. **Layered Exploration:** BFS explores all nodes level by level (or layer by layer), ensuring that the first time it reaches a node, it does so using the minimum number of edges. This makes BFS ideal for finding the shortest path in terms of the number of edges.
3. **Optimal Path:** By starting from the source island (Island #1), BFS will traverse through all possible routes. When BFS reaches Island #N, it will have traversed the fewest possible edges to get there, ensuring that Sophie crosses the minimum number of bridges.

The steps for this approach are:

1. Start BFS from Island #1.
2. Use a queue to explore all neighboring islands, marking each island as **visited** and tracking the number of bridges crossed.
3. If BFS reaches Island #N, the number of bridges crossed so far is the answer.
4. If BFS completes and Island #N isn't reached, it means there's no path to Island #N.

Question 2: Given the following connections between the islands:

- **Islands:** 6
- **Bridges:** 7
- **Connections:** 1-2, 2-3, 3-4, 1-4, 5-4, 6-5, 4-6
- **Starting Island:** Island #1
- **Destination Island:** Island #6

What is the optimal route Sophie should take to reach Island #6, and how many bridges will she cross? Explain your reasoning.

Let's apply the BFS algorithm to find the shortest path from **Island #1** to **Island #6**.

1. **Graph Representation:** The graph can be represented as an adjacency list:

- 1: [2, 4]
- 2: [1, 3]
- 3: [2, 4]
- 4: [1, 3, 5, 6]
- 5: [4, 6]
- 6: [5, 4]

2. **BFS Process:**

- **Start at Island #1:**
 - Current Queue: [1]
 - Visited: {1}
 - Distance: {1: 0}
- **Explore neighbors of Island #1:**
 - From Island #1, we can go to Island #2 and Island #4.
 - Update Queue: [2, 4]
 - Visited: {1, 2, 4}
 - Distance: {1: 0, 2: 1, 4: 1}
- **Explore neighbors of Island #2:**
 - From Island #2, we can go to Island #1 and Island #3. But Island #1 is already visited, so we move to Island #3.
 - Update Queue: [4, 3]
 - Visited: {1, 2, 3, 4}
 - Distance: {1: 0, 2: 1, 4: 1, 3: 2}
- **Explore neighbors of Island #4:**

- From Island #4, we can go to Island #1, Island #3, Island #5, and Island #6.
- Island #1 and Island #3 are already visited, so we move to Island #5 and Island #6.
- Update Queue: [3, 5, 6]
- Visited: {1, 2, 3, 4, 5, 6}
- Distance: {1: 0, 2: 1, 4: 1, 3: 2, 5: 2, 6: 2}

3. Reaching Island #6:

- At this point, we have reached **Island #6** after crossing **2 bridges**.

4. Optimal Route: The **optimal route** to Island #6 is **1 → 4 → 6** (2 bridges).

Answer: The optimal route is 1 → 4 → 6, and Sophie will cross 2 bridges.

Reasoning:

- BFS guarantees that the first time we reach a node (Island #6), we will do so with the minimum number of bridges, which in this case is 2.
- Since BFS explores all nodes level by level, we find the shortest path in terms of the number of bridges. The queue ensures that each node is explored with the minimum number of edges (bridges) from the starting point.

Problem 3

You are working as a network engineer for a large communication company. The company's data center is designed as a network of nodes, with each node representing a server. These servers are connected by edges, which represent communication links between them. Each server in the data center is identified by a node, and there is one special node called the **head node**, which is responsible for coordinating the network.

You need to check how many servers (nodes) are unreachable from the head node. Some servers might be isolated or disconnected from the head node due to network issues. The graph representing the network may have multiple edges between the same nodes or self-loops, and it's undirected. Your task is to identify how many nodes are unreachable from the head node.

Input:

- The total number of nodes (servers) in the network, followed by the total number of communication links (edges).
- A list of M communication links between servers. Each link is represented as a pair of nodes aaa and b, where there is a communication link between node a and node b.
- The index of the head node.

Output:

- Print a single integer, denoting the number of servers (nodes) that are unreachable from the head node.

Questions:

1. Describe the algorithm approach to solve this problem.
2. Given the following input:
10 servers in total, and 10 communication links between them.
Connections between the servers are as follows:
8-1, 8-3, 7-4, 7-5, 2-6, 10-7, 2-8, 10-9, 2-10, 5-10
The head node is 2.

How many servers are unreachable from the head node (node 2)? What are these unreachable servers? Explain your reasoning.

Answer

Question 1: Describe the algorithm approach to solve this problem.

To solve the problem of determining the number of **unreachable servers** from the **head node** in a **network** of servers (represented as a graph), we can apply **Depth-First Search (DFS)**. Let's break this down step by step:

1. Graph Representation:

- The servers are represented as **nodes** in a graph.
- The communication links between servers are represented as **edges**.
- The **head node** is the starting point from which we need to find all reachable nodes.

2. DFS Approach:

- **DFS** is a natural choice for this problem because it explores nodes by going as deep as possible before backtracking. This ensures that we can explore all

reachable servers from the head node.

- The reason for using **DFS** over **BFS** or other approaches is its ability to explore all nodes in a connected component in a recursive manner. When we reach a node, we can check if it is connected to other servers. If it is, we recursively explore those connections.
- In DFS, we use a **visited list** to keep track of nodes that have already been explored. This prevents revisiting the same node and ensures we don't get stuck in cycles.

3. Why DFS:

- **Graph traversal:** Since the problem involves finding all reachable servers from the head node, **DFS** efficiently solves this by visiting nodes along the depth of the graph. Once DFS finishes, we can identify which nodes are reachable and which are not.

4. Algorithm Approach:

- Start by building the graph using an adjacency list.
- Perform **DFS** starting from the head node. Mark each node that is visited as **reachable**.
- After the DFS traversal, count the number of nodes that were not marked as visited. These are the **unreachable nodes**.
- Finally, return the number of unreachable nodes and list them.

By applying **DFS**, we ensure that every node that is connected to the head node is explored, and we can easily identify and count the unreachable nodes.

Question 2: Given the following input, how many servers are unreachable from the head node (node 2)? What are these unreachable servers? Explain your reasoning.

Input:

- 10 servers (nodes) in total.
- 10 communication links (edges) between servers.
- The connections between servers are:
8-1, 8-3, 7-4, 7-5, 2-6, 10-7, 2-8, 10-9, 2-10, 5-10

- The **head node** is **2**.

Step-by-Step Reasoning:

1. **Graph Representation:** We can construct an adjacency list from the provided connections:
1: [8]
2: [6, 8, 10]
3: [8]
4: [7]
5: [7, 10]
6: [2]
7: [4, 5, 10]
8: [1, 2, 3]
9: [10]
10: [2, 5, 7, 9]
2. **DFS Traversal:** We start DFS from the head node **2** and explore all reachable nodes:
 - From node **2**, we can go to nodes **6**, **8**, and **10**.
 - From node **6**, we can only go back to **2**, which we already visited.
 - From node **8**, we can go to nodes **1**, **2**, and **3**. Since **2** is already visited, we go to **1** and **3**.
 - From node **1**, we go to node **8**, which is already visited.
 - From node **3**, we go to node **8**, which is already visited.
 - From node **10**, we can go to nodes **2**, **5**, **7**, and **9**. Since **2** is visited, we explore **5**, **7**, and **9**.
 - From node **5**, we go to nodes **7** and **10**, but **7** is visited, and **10** is already explored.
 - From node **7**, we go to nodes **4**, **5**, and **10**, but **5** and **10** are already visited. We explore **4**.
 - From node **4**, we can only go to node **7**, which is already visited.

3. **Visited nodes:** {2, 6, 8, 1, 3, 10, 5, 7, 4, 9}

4. **Unreachable Nodes:**

- All nodes are reachable from node **2**, so there are **no unreachable nodes**.

Answer:

- **Number of unreachable nodes: 0.**
 - **Unreachable nodes:** None.
-

Problem 4

In a small village, there are n villagers, each labeled from 1 to n . A rumor is spreading that one of the villagers is secretly the village mayor. The mayor has a special role in the village and is trusted by everyone except themselves. The mayor trusts no one.

You are tasked with identifying the mayor based on a series of trust relationships. A trust relationship between two villagers is represented as a pair $[a,b]$, meaning that villager a trusts villager b . If a trust relationship does not exist between two villagers, no trust relationship exists between them.

The mayor must meet two criteria:

1. The mayor trusts nobody.
2. Everyone else (except for the mayor) trusts the mayor.

If a mayor exists, return the label of the mayor; otherwise, return -1.

Example:

Input:

$n = 3, \text{ trust} = [[1, 3], [2, 3]]$

Output:

3

Explanation:

Villagers 1 and 2 both trust villager 3, and villager 3 trusts no one. Therefore, villager 3 is the mayor of the village.

Questions:

1. Describe the algorithm approach to solve this problem.
2. Given the following input:

$n = 4$, $trust = [[1, 4], [2, 4], [3, 4], [4, 1]]$

Can you identify the mayor of the village? If so, who is it? If not, explain why.

Answer

To identify the mayor in a village, we need to rely on two clear behaviors:

1. **The mayor trusts no one** – they don't rely on anyone else.
2. **Everyone else trusts the mayor** – because they believe the mayor is in charge.

This situation can be modeled using a **directed graph**, where each villager is a node, and a trust relationship is a directed edge from one node to another.

In graph theory:

- **Outdegree** of a person = number of people they trust (edges going out).
- **Indegree** of a person = number of people who trust them (edges coming in).

So, the mayor must have:

- **Outdegree = 0** (they trust no one).
- **Indegree = $n - 1$** (everyone else trusts them).

By tracking the difference between indegree and outdegree for each person, we can efficiently find the one who fits both conditions.

Algorithm Approach:

1. **Create a `count` array** of size $n + 1$ (because people are labeled from 1 to n).
 - `count[i]` will track the net trust score: how many people trust person i minus how many they trust.
2. **Iterate through all trust relationships:**
 - For a pair $[a, b]$, person a trusts person b .
 - So, **decrease** `count[a]` by 1 (they trust someone).
 - And **increase** `count[b]` by 1 (they are trusted).
3. **Look for the mayor:**
 - The person who has `count[i] === n - 1` is the one who is trusted by everyone else and trusts no one.
4. **Return their label**, or return -1 if no such person exists.

Step by step example

Input:

```
n = 4
trust = [[1, 4], [2, 4], [3, 4], [4, 1]]
```

Step-by-step reasoning:

- Let's compute the trust changes:

Person	Trusts Someone (Outdegree -1)	Is Trusted By (Indegree +1)	Net Trust
1	-1	0	-1
2	-1	0	-1
3	-1	0	-1
4	-1	+3	+2

- Now, check for the person with **net trust = n - 1 = 3**:
 - Person 4 has a net trust of +2, not +3, because they **trusted person 1**, violating the rule that the mayor trusts no one.

Answer:

No, there is **no mayor** in this village. Although person 4 is trusted by everyone else, they also **trust person 1**, which disqualifies them.

Problem 5

You are tasked with evaluating a system of interconnected devices in a smart home network. Each device is represented as a node in a directed graph, and connections between devices are represented by edges. The devices communicate with each other based on these connections, and some devices are terminal devices, meaning they do not communicate with any other device.

A device is considered **safe** if every possible communication starting from that device eventually leads to a terminal device or another safe device. Your task is to find all the safe devices in the network and return them in ascending order.

The system is represented by a 0-indexed 2D array called **graph**, where each index i represents a device, and **graph[i]** contains the list of devices that device i communicates with. A device is a **terminal device** if it does not communicate with any other devices.

Input:

- An integer array **graph** of size n , where **graph[i]** is an array representing the devices that device i communicates with.
- Each device is represented by a node labeled from 0 to $n-1$.

Output:

- Return an array containing all the safe devices, sorted in ascending order.

Example:

Input:

```
graph = [[1, 2], [2, 3], [5], [0], [5], [], []]
```

Output:

```
[2, 4, 5, 6]
```

Explanation:

- Devices 5 and 6 are terminal devices because they do not communicate with any other devices.
- Every communication starting at devices 2, 4, 5, and 6 eventually leads to either device 5 or 6, which are terminal devices.
- Therefore, the safe devices are 2, 4, 5, and 6.

Questions:

1. **Describe the approach to solve the problem of identifying safe devices.** (Explain how you can determine which devices are safe based on the communication paths and terminal devices in the network.)
2. **Consider the following graph:** `graph = [[1, 2, 3, 4], [1, 2], [3, 4], [0, 4], []]`

Which devices are safe in this network? Explain why.

Answer:

To identify safe devices in the given smart home network represented as a directed graph, we use **Depth-First Search (DFS)**. The main goal is to explore whether each device eventually leads to a **terminal device** (safe) or to a **cycle** (unsafe). DFS is particularly useful here because it allows us to traverse the graph deeply and detect cycles in the process.

Why DFS?

- **Cycle Detection:** DFS excels in detecting cycles in a directed graph, which is crucial for identifying unsafe devices. If a device leads to a cycle, it is considered unsafe because the communication from that device will not terminate — it will loop infinitely.
- **Path Exploration:** DFS explores each path starting from a device, allowing us to determine whether that path leads to a terminal device (safe) or an unsafe cycle. This depth-first exploration helps fully capture the communication chain of each device.
- **Backtracking:** One of the core features of DFS is backtracking — once all neighbors of a device have been explored, the search backtracks to the previous device to explore other paths. This makes it easy to mark devices as either safe or unsafe after fully exploring all potential communication routes.

State Tracking: The Key to Identifying Safe and Unsafe Devices

In DFS, state tracking is essential for determining the safety of each device. We use **two arrays** to manage the state of each device during the DFS traversal:

1. `visited[]` Array:

- **Purpose:** The `visited[]` array tracks whether a device has been fully explored. This ensures that each device is processed only once.
- **How it Works:**
 - Initially, all devices are marked as unvisited (`visited[i] = False`).
 - When we begin DFS on a device `i`, we mark it as visited (`visited[i] = True`).
 - Once all neighbors of the device are explored (i.e., the DFS path has been completely followed), we consider the device fully explored. If the device has been explored before, we skip further DFS on it.
- **When it's Used:**
 - The `visited[]` array helps avoid **redundant DFS calls**. If a device has been visited and fully explored, we don't need to start DFS from it again, ensuring efficiency.

2. `inStack[]` Array:

- **Purpose:** The `inStack[]` array keeps track of whether a device is part of the **current DFS recursion stack**. This is essential for detecting cycles. If we encounter a device that is already in the recursion stack, it means we have visited it during the current DFS path, and thus, a cycle has been detected.
- **How it Works:**
 - When we start exploring a device `i`, we mark it as part of the recursion stack (`inStack[i] = True`).
 - As we recursively explore its neighbors, the device remains in the stack until all its neighbors have been fully explored.
 - Once we finish exploring all neighbors of a device, we mark it as **no longer part of the recursion stack** (`inStack[i] = False`) and

backtrack to explore other paths.

- **Cycle Detection:**

- **Key role:** If we encounter a device that is already in the `inStack` during our DFS exploration, it means that device is involved in a cycle. This is because we're revisiting a device that's still being explored as part of the current path, indicating that we've looped back to it, forming a cycle.
- **Unsafe Devices:** Any device that is part of a cycle or leads to a cycle is marked as **unsafe**.

- **When it's Used:**

- The `inStack[]` array is crucial for **cycle detection** and helps us identify unsafe devices quickly.

Step-by-Step Example:

We will perform a Depth-First Search (DFS) for each device, and during the DFS, we will track:

- Whether a device is part of a cycle (`inStack[]`).
- Whether a device has been fully explored (`visited[]`).

1. Initialize `visited[]` and `inStack[]`:

- `visited = [False, False, False, False, False]` (no devices have been explored yet).
- `inStack = [False, False, False, False, False]` (no devices are in the DFS recursion stack).

2. DFS on Device 0:

- Mark device 0 as visited and part of the recursion stack:
`visited[0] = True, inStack[0] = True`
- Explore its neighbors: `[1, 2, 3, 4]`.

3. DFS on Device 1 (Neighbor of Device 0):

- Mark device 1 as visited and part of the recursion stack:
`visited[1] = True, inStack[1] = True`
- Explore its neighbors: [1, 2].

4. DFS on Device 1 (Self-loop):

- **Cycle Detected:** Device 1 points to itself, so we detect a cycle.
- **Device 1 is unsafe** because it is part of a cycle.
- Backtrack from device 1, marking it as no longer part of the recursion stack (`inStack[1] = False`).

5. DFS on Device 2 (Neighbor of Device 0):

- Mark device 2 as visited and part of the recursion stack:
`visited[2] = True, inStack[2] = True`
- Explore its neighbors: [3, 4].

6. DFS on Device 3 (Neighbor of Device 2):

- Mark device 3 as visited and part of the recursion stack:
`visited[3] = True, inStack[3] = True`
- Explore its neighbors: [0, 4].

7. DFS on Device 0 (Neighbor of Device 3):

- **Cycle Detected:** Device 0 is already in the recursion stack (`inStack[0] = True`), indicating a cycle.
- **Device 3 is unsafe** because it leads to a cycle (via device 0).
- Backtrack from device 3, marking it as no longer part of the recursion stack (`inStack[3] = False`).

8. DFS on Device 4 (Neighbor of Device 3):

- Mark device 4 as visited and part of the recursion stack:
`visited[4] = True, inStack[4] = True`
- Device 4 is a **terminal device** (it has no outgoing edges).
- **Device 4 is safe** because it is a terminal device.
- Backtrack from device 4, marking it as no longer part of the recursion stack (`inStack[4] = False`).

9. Backtrack to Device 2:

- No more neighbors to explore for device 2.
- **Device 2 is unsafe** because it leads to device 3, which is part of a cycle.
- Backtrack from device 2, marking it as no longer part of the recursion stack (`inStack[2] = False`).

10. Backtrack to Device 0:

- No more neighbors to explore for device 0.
- **Device 0 is unsafe** because it leads to a cycle (via device 3).
- Backtrack from device 0, marking it as no longer part of the recursion stack (`inStack[0] = False`).

Final Result:

- **Device 0:** Unsafe because it leads to a cycle (via device 3).
- **Device 1:** Unsafe because it has a self-loop (a cycle).
- **Device 2:** Unsafe because it leads to device 3, which is part of a cycle.
- **Device 3:** Unsafe because it leads to device 0, which is part of a cycle.
- **Device 4:** Safe because it is a terminal device and has no outgoing edges.

Safe Devices:

The only **safe device** is **device 4**.

Lab 9

Problem 1

You are helping to design a **navigation system** for a small city. The city has **5 important locations**, connected by **5 one-way roads**. Your task is to calculate the **shortest time (in minutes)** it takes to reach each location from the **Central Station** (Location 1).

Each road goes **from one location to another**, and has a certain **travel time**.

Here is the road map:

Number of locations: 5

Number of roads: 5

Roads:

1 → 2 (5 minutes)

1 → 3 (2 minutes)

3 → 4 (1 minute)

1 → 4 (6 minutes)

3 → 5 (5 minutes)

Question:

Use **Dijkstra's Algorithm** to find the **shortest travel time** from the **Central Station (Location 1)** to each of the other locations (Locations 2 to 5).

Show step-by-step how the algorithm updates distances.

At the end, write the final travel times separated by spaces. If a location can't be reached, write "INF".

Answer

Step 0: Initialization

We apply **Dijkstra's Algorithm** starting from **Location 1 (Central Station)**.

We initialize the shortest distances from Location 1 as follows:

Distance[1] = 0 (starting point)

Distance[2..5] = ∞ (unknown at the beginning)

We also maintain a **priority queue (min-heap)** that always selects the next closest location to process.

Road Map:

1 → 2 (5)

1 → 3 (2)

3 → 4 (1)

1 → 4 (6)

3 → 5 (5)

Step-by-Step Execution

Step 1: Start at Location 1

- Distance[1] = 0
- Explore neighbors from Location 1:
 - 1 → 2: $0 + 5 = 5$ → update Distance[2] = 5
 - 1 → 3: $0 + 2 = 2$ → update Distance[3] = 2
 - 1 → 4: $0 + 6 = 6$ → update Distance[4] = 6

Priority queue: [(2, 3), (5, 2), (6, 4)]

Current distances:

[0, 5, 2, 6, ∞]

Step 2: Visit Location 3 (distance 2)

- Explore neighbors from Location 3:
 - 3 → 4: $2 + 1 = 3$ → update Distance[4] = 3 (better than previous 6)
 - 3 → 5: $2 + 5 = 7$ → update Distance[5] = 7

Priority queue: [(3, 4), (5, 2), (7, 5)]

Current distances:

[0, 5, 2, 3, 7]

Step 3: Visit Location 4 (distance 3)

- Location 4 has no outgoing roads → nothing to update

Priority queue: [(5, 2), (7, 5)]

Step 4: Visit Location 2 (distance 5)

- Location 2 has no outgoing roads → nothing to update

Priority queue: [(7, 5)]

Step 5: Visit Location 5 (distance 7)

- Location 5 has no outgoing roads → nothing to update

Priority queue: []

Final Shortest Travel Times:

From Location 1 to:

- Location 2: 5 minutes
- Location 3: 2 minutes
- Location 4: 3 minutes
- Location 5: 7 minutes

Final Output: 5 2 3 7

Problem 2

You are managing a project that consists of **5 tasks**, and some tasks can only be started **after others are completed**. These dependencies form a **Directed Acyclic Graph (DAG)** — meaning the tasks must follow a one-way, non-circular sequence.

Each task is represented by a number from 1 to 5. If task **X** must be completed before task **Y**, it is written as a directed edge **X → Y**.

Below is the structure of your project:

Number of tasks: 5

Number of dependencies: 6

Dependencies:

1 → 2

1 → 3

2 → 3

2 → 4

3 → 4

3 → 5

Your goal is to determine a **valid order to complete all tasks** such that every task is started **only after all its prerequisite tasks** have been completed.

If there are **multiple valid orders**, print the one that is **lexicographically smallest** (i.e., appears first if you sort all valid results like words in a dictionary).

Question:

Simulate the **topological sort step by step**.

At each step, choose the available task with the **smallest number** (lowest task ID) among those with no remaining prerequisites.

At the end, print the topological order — a single line of task numbers in the required order.

Answer

Why Topological Sort?

We are given a list of **tasks with dependencies**, meaning certain tasks must be completed before others. This kind of dependency structure forms a **Directed Acyclic Graph (DAG)**. Our goal is to find an **ordering of tasks** such that **every task is scheduled only after all its prerequisites are done**.

This is exactly what **topological sort** is designed to do:

A topological sort of a DAG is a linear ordering of its vertices such that for every directed edge $u \rightarrow v$, u comes before v in the ordering.

Why Use a Min-Heap?

In this problem, **multiple valid topological orders** may exist. For example, if both Task 2 and Task 3 are available at the same time, either could be chosen.

But the question specifically asks for the **lexicographically smallest** order — this means we should always choose the **lowest-numbered available task** at each step.

To achieve this, we use a **min-heap (priority queue)**:

- It allows us to always access the **smallest available task number** in $O(\log N)$ time.
- This ensures that at every step, among all the available tasks with no prerequisites, we choose the one with the **smallest ID**.

Step-by-Step Execution

Given:

Number of tasks: 5

Dependencies:

1 → 2

1 → 3

2 → 3

2 → 4

3 → 4

3 → 5

Step 1: Build Graph and In-Degree Array

We represent the graph using an adjacency list and track the number of prerequisites (**in-degree**) for each task.

Adjacency list:

1: [2, 3]

2: [3, 4]

3: [4, 5]

In-degree:

Task 1: 0

Task 2: 1 (from 1)

Task 3: 2 (from 1 and 2)

Task 4: 2 (from 2 and 3)

Task 5: 1 (from 3)

Step 2: Initialize the Min-Heap

We start with all tasks that have in-degree = 0 → only Task 1
→ Min-Heap = [1]

The Core Idea: Why We Decrease In-Degree and Push to Heap

Each task's **in-degree** shows how many tasks must be completed **before** it can start.
When we **process a task** (i.e., pop it from the heap), we treat it as “completed.”

For each neighbor **v** of that task (each task that depends on it), we do:

```
in_degree[v] -= 1
```

This reflects that **one prerequisite has been completed**.

If `in_degree[v]` becomes 0, that means **all prerequisites are done**, and it's now **ready to be scheduled** — so we **push it to the heap**.

Using the **min-heap** ensures we always choose the **smallest task ID** among available options, which guarantees the **lexicographically smallest topological order**.

Step-by-Step Topological Sort

1. **Pop 1** from heap → result: [1]

- Task 2: `in_degree[2] -= 1` → becomes 0 → push 2 to heap
- Task 3: `in_degree[3] -= 1` → becomes 1
→ Heap = [2]

2. **Pop 2** → result: [1, 2]

- Task 3: `in_degree[3] -= 1` → becomes 0 → push 3
- Task 4: `in_degree[4] -= 1` → becomes 1
→ Heap = [3]

3. **Pop 3** → result: [1, 2, 3]

- Task 4: `in_degree[4] -= 1` → becomes 0 → push 4

- Task 5: `in_degree[5] -= 1` → becomes 0 → push 5
→ Heap = `[4, 5]`
- 4. **Pop 4** (smallest of 4 and 5) → result: `[1, 2, 3, 4]`
 - No neighbors to process
→ Heap = `[5]`
- 5. **Pop 5** → result: `[1, 2, 3, 4, 5]`
 - No neighbors to process
→ Heap is empty

Final Output: `1 2 3 4 5`

This is a **valid topological order**, and because we used a **min-heap**, it's also the **lexicographically smallest** possible.

Problem 3

In the near future, a **smart city** is equipped with **automated roads** that **disappear after a certain time**. These roads are used for transportation, and their existence is time-dependent — they automatically become **inactive** at certain times.

Your task is to determine the **minimum time required to reach each district** from the **central hub (District 0)** before the **road to that district disappears**. If a district is unreachable before the road disappears, return `-1`.

You are given:

1. A **city map** consisting of **districts** connected by **automated roads** (edges) with traversal times.
2. An array indicating when each **district becomes inactive** (its corresponding road disappears).

The problem is to compute the **minimum time to reach each district** from the central hub (District 0) before the roads to them become unavailable.

Example Input:

`Number of districts: 3`

Number of roads: 3

Starting district: 1

Target district: 3

Roads:

0 → 1 (2 minutes)

1 → 2 (1 minutes)

0 → 2 (4 minute)

Disappear times:

District 0: 1

District 1: 3

District 2: 5

Questions:

1. **Describe the algorithm** used to find the minimum time required to reach each district from the Central District (node 0) before the road disappears.
2. **Simulate the process step-by-step** for the given example, showing how time is updated and how the disappearance times of the roads affect the journey.

Answer

Why Use Dijkstra's Algorithm?

In this problem, we need to find the minimum time required to reach each district from the central district (District 0) before the road to that district disappears. This introduces a time-dependent constraint: roads become inactive after certain times, so we need to ensure that we reach each district before its road becomes unavailable.

We can solve this problem using **Dijkstra's algorithm** because:

1. **Shortest Path:** Dijkstra's algorithm is designed to find the shortest path in a graph with non-negative edge weights. In this case, the edge weights are the travel times for the roads.
2. **Priority Queue:** The algorithm uses a priority queue (min-heap) to always process the node with the smallest accumulated time first. This is ideal for ensuring we always find the quickest path to each district.
3. **Time Constraints:** We can modify Dijkstra's algorithm to take the road disappearance times into account by skipping nodes whose current time exceeds their disappearance time.

In short, Dijkstra's algorithm is efficient for finding the shortest path in graphs, and with slight modification, it can handle the time-sensitive nature of this problem.

Step-by-Step Solution

1. **Graph Representation:**
 - We represent the city's districts as a graph, where nodes represent the districts and edges represent the roads between them, with the associated travel times.
2. **Priority Queue (Min-Heap):**
 - We use a priority queue to process nodes starting from the central district (District 0), ensuring that we always process the district with the smallest time required to reach it.
3. **Visited and Disappearance Check:**
 - We track which districts have been visited to avoid processing the same district multiple times.
 - For each district, we also check if the current time to reach it exceeds its disappearance time. If it does, we skip that district.
4. **Neighbor Exploration:**
 - For each district, we explore its neighbors (connected districts), calculate the time it takes to reach them, and update the minimum time if a shorter path is found.
5. **Final Answer:**

- After running the algorithm, we return the minimum time to reach each district. If a district is unreachable before the road disappears, we return -1 .

Example Walkthrough

1. Initialization:

- We start with **District 0** at time 0 in the priority queue.
- The **dist** array (minimum time to reach each district) is initialized as $[0, \text{inf}, \text{inf}]$, where **inf** represents that the district is initially unreachable, and 0 is the time to reach District 0.
- The **disappear** times are $[1, 3, 5]$.

2. First Step - Process District 0 (Starting Point):

- We pop District 0 from the priority queue with a time of 0 .
- We explore its neighbors:
 - **To reach District 1:** The travel time is 2 minutes, so the new time is $0 + 2 = 2$. Since District 1 disappears at time 3 , we can reach it before it disappears. We update the time for District 1 to 2.
 - **To reach District 2:** The travel time is 4 minutes, so the new time is $0 + 4 = 4$. Since District 2 disappears at time 5 , we can reach it before it disappears. We update the time for District 2 to 4.

3. After this step, the **dist** array is $[0, 2, 4]$, meaning:

- District 0 (starting district) has a time of 0 .
- District 1 can be reached in 2 minutes.
- District 2 can be reached in 4 minutes.

4. Second Step - Process District 1:

- We now pop District 1 from the priority queue with a time of 2 .

- We explore its neighbors:
 - **To reach District 0:** The travel time is 2 minutes, so the new time would be $2 + 2 = 4$, which is greater than the current time to reach District 0 (which is 0). Therefore, no update is made for District 0.
 - **To reach District 2:** The travel time is 1 minute, so the new time is $2 + 1 = 3$. Since District 2 disappears at time 5, we can reach District 2 before it disappears. We update the time for District 2 to 3.

5. After this step, the **dist** array is updated to $[0, 2, 3]$.

6. Final Step - Process District 2:

- We now pop District 2 from the priority queue with a time of 3.
- We explore its neighbors:
 - **To reach District 0:** The travel time is 4 minutes, so the new time would be $3 + 4 = 7$, which is greater than the current time to reach District 0 (which is 0). Therefore, no update is made for District 0.
 - **To reach District 1:** The travel time is 1 minute, so the new time would be $3 + 1 = 4$, which is greater than the current time to reach District 1 (which is 2). Therefore, no update is made for District 1.

7. After this step, the **dist** array remains $[0, 2, 3]$.

Final Answer:

- **District 0 (Starting point):** We are already at District 0, so the time is 0.
- **District 1:** We can reach District 1 in 2 minutes before it disappears at time 3, so the answer is 2.
- **District 2:** We can reach District 2 in 3 minutes before it disappears at time 5, so the answer is 3.

Thus, the minimum times to reach each district are $[0, 2, 3]$.

Problem 4

You are exploring an archipelago of **5 islands**, each connected by a network of **one-way bridges**. These bridges let you travel from one island to another — but **only in the forward direction**. Once you leave an island, **you cannot come back**.

You start on **Island 1**. From there, at every step, you choose **uniformly and randomly** one of the islands directly reachable via a bridge and move to it. You repeat this process until you land on an island with **no outgoing bridges** — in which case you're **stuck there forever**.

You are given the number of islands, the list of bridges (each from island **X** to island **Y**), and the island you start on. Your task is to **simulate the process** and determine:

Which island(s) you are **most likely** to end up stuck on.

If there are multiple islands with the **same maximum probability**, return them in **increasing order**.

Example Input:

Number of islands: 5

Number of one-way bridges: 7

Start island: 1

Bridges:

1 → 2

1 → 3

1 → 4

1 → 5

2 → 4

2 → 5

3 → 4

Question:

- Describe the algorithm used to find the island(s) where you are most likely to get stuck.
- Simulate the process step by step using the example and show how probabilities are distributed.

Answer

The **problem** is that we need to simulate the movement across islands where the probability of ending up at each island depends on the probability of reaching the **previous islands** that can move into it.

However, some islands are **dependent** on others — for example, you can't calculate the probability of being at island 4 until you know the probabilities from island 1, 2, and 3 that contribute to it.

So the challenge is: if we update islands in the wrong order, we might calculate a probability **before** knowing all the possible sources that flow into it — leading to **incomplete or incorrect values**.

So we need to use Topological Traversal because:

- It processes each island **only after all its incoming sources** have already been processed.
- It guarantees that for any directed edge $u \rightarrow v$, we process u **before** v .
- Since the graph is a **DAG**, topological ordering is always possible and reliable for forward-only propagation.
- This solves the **dependency issue**, ensuring that each probability value is fully updated before it's passed on.

Once we've computed probabilities in topological order, we simply check the **sink islands** (with no outgoing bridges) and return the one(s) with the **highest final probability**.

How Do We Calculate Probability?

- We start by assigning $\text{prob}[\text{start}] = 1.0$, meaning you are 100% sure to start from the given island.
- For every island u , if it has k outgoing bridges (neighbors), then it distributes its current probability equally: $\text{prob}[u] / k$ to each neighbor.
- We repeat this for every island, in topological order.
- The final probabilities at the **sink islands** (those with no outgoing bridges) represent the **likelihood of being stuck there**.
- We return the island(s) with the **highest probability** among the sinks.

Simulate the process step by step

Step 1: Topological Traversal

We apply **Kahn's Algorithm** to generate a valid topological order:

1. Calculate **in-degree** (number of incoming edges) for each island:

```
in-degree[1] = 0
in-degree[2] = 1 (from 1)
in-degree[3] = 1 (from 1)
in-degree[4] = 3 (from 1, 2, 3)
in-degree[5] = 2 (from 1, 2)
```

2. Start with nodes having in-degree 0 → queue = [1]

3. Process node 1:

- Add to topo order: [1]
- Decrease in-degrees:
in-degree[2] = 0 → enqueue
in-degree[3] = 0 → enqueue
in-degree[4] = 2
in-degree[5] = 1

4. Process node 2:

- Topo order: [1, 2]
- Update in-degrees:
in-degree[4] = 1
in-degree[5] = 0 → enqueue

5. Process node 3:

- Topo order: [1, 2, 3]
- Update in-degree:
in-degree[4] = 0 → enqueue

6. Process node 5 → [1, 2, 3, 5]

7. Process node 4 → [1, 2, 3, 5, 4]

Final topological order: [1, 2, 3, 5, 4]

Step 2: Probability Propagation (based on topo order)

1. Initialize probabilities:

```
prob[1] = 1.0 # starting point
prob[2..5] = 0.0
```

2. Process island 1

Has 4 outgoing edges → each neighbor gets $1.0 / 4 = 0.25$

`prob[2] = 0.25`

`prob[3] = 0.25`

`prob[4] = 0.25`

`prob[5] = 0.25`

3. Process island 2

Has 2 outgoing edges → each gets $0.25 / 2 = 0.125$

`prob[4] += 0.125 → prob[4] = 0.375`

`prob[5] += 0.125 → prob[5] = 0.375`

4. Process island 3

Has 1 outgoing edge → all 0.25 goes to 4

`prob[4] += 0.25 → prob[4] = 0.625`

5. Process islands 5 and 4

Both have no outgoing edges (sink nodes) → we do not propagate further.

Final Answer: 4

Island 4 has the **highest probability** of being the final destination, so it is the island you're most likely to get stuck on.

Problem 5

You are leading a rescue operation in a **smart building** where each room is equipped with **automated fire suppression systems**. The building is laid out as an $n \times m$ grid, and you're given a 2D list `moveTime`, where `moveTime[i][j]` represents the **earliest time (in seconds)** at which the room (i, j) becomes safe to enter after the fire suppression is activated.

You start at the **emergency control room** located at the **top-left corner** $(0, 0)$ at **time 0**. You can move to **adjacent rooms** (up, down, left, or right), and **each move takes exactly 1 second**. However, you **cannot enter a room before its safety system finishes** (i.e., current time must be $\geq \text{moveTime}[i][j]$).

Your mission is to reach the **trapped survivors** at the **bottom-right room** $(n-1, m-1)$ in the **minimum time possible**.

Example Inputs:

Example 1:

```
moveTime = [
    [0, 4],
    [4, 4]
]
```

Example 2:

```
moveTime = [
    [0, 0, 0],
    [0, 0, 0]
]
```

Questions:

1. Describe the algorithm used to find the minimum time to reach the survivors in the bottom-right room.
2. Simulate the path step-by-step for each example and show how time is updated at each move.

Answer

Question 1: Describe the algorithm used to find the minimum time to reach the survivors in the bottom-right room.

You are in a smart building where **each room has a safety lock**, and you are **not allowed to enter** until the room is **ready** (`moveTime[i][j]` seconds after the start).

You start from `(0, 0)` and want to reach the bottom-right room `(n-1, m-1)` as **quickly as possible**, moving one cell at a time (up, down, left, right) — each move takes exactly **1 second**.

But there's a twist: *Even if a room is next to you, you **must wait** if it's not yet safe to enter.*

Why use Dijkstra's Algorithm?

This is a **shortest-path problem with dynamic movement costs**:

- In a normal grid, you'd use **BFS** (uniform cost).
- But here, **some rooms make you wait longer than others**, so the cost to enter depends on the **current time** and the room's **unlock time**.

That makes it a **weighted shortest path problem**, where:

- Nodes = rooms
- Edge weight = $\max(\text{current time}, \text{room unlock time}) + 1$
- Goal = minimize time to reach $(n-1, m-1)$

Dijkstra's Algorithm is the best tool when:

- You want the **fastest route**.
- Travel cost depends on **where you are now**, and **where you go next**.

How the algorithm works:

1. **Use a priority queue (min-heap)** to always explore the room that can be reached the **earliest**.
2. Keep a `minTime[i][j]` matrix to record the **earliest time you can reach each room**.
3. Start from $(0, 0)$ at time 0 and push it into the heap.
4. While the heap is not empty:
 - Pop the room with the **lowest current time**.
 - Try to move to all 4 adjacent rooms:
 - You can only move into (ni, nj) **at or after** `moveTime[ni][nj]`
 - Wait if needed: $\text{nextTime} = \max(\text{currTime}, \text{moveTime[ni][nj]}) + 1$
 - If `nextTime` is better than the recorded time, update it and push it to the heap.
5. When you reach $(n-1, m-1)$, return the current time.

Question 2: Step-by-step simulation

Example 1

`moveTime = [`

$[0, 4],$
 $[4, 4]$
 $]$

Start at room (0, 0) at time 0.

Step 1:

Try moving to adjacent rooms:

- Room (0, 1): it is locked until time **4**, so you must wait. You can enter it at $\max(0, 4) + 1 = 5$.
- Room (1, 0): also locked until time **4**, so you must wait. You can enter it at $\max(0, 4) + 1 = 5$.

Push both of these into the heap:

$(5, 0, 1), (5, 1, 0)$

Step 2:

Pop the room with the lowest time $\rightarrow (5, 0, 1)$

From here, try moving to adjacent rooms:

- Room (1, 1): it is available at time 4, and you're already at time 5, so you can enter immediately after 1 second: $5 + 1 = 6$.

Push into the heap:

$(6, 1, 1)$

Step 3:

Pop the next room $\rightarrow (5, 1, 0)$

From here, try moving to room (1, 1) again. But we already plan to reach room (1, 1) at time 6, so we **skip it** (no improvement).

Step 4:

Pop room (6, 1, 1) → this is the destination.
You have reached the bottom-right room at **time 6**.

Output: 6

Example 2

```
moveTime = [  
    [0, 0, 0],  
    [0, 0, 0]  
]
```

Start at room (0, 0) at time 0.

Step 1:

Try moving to adjacent rooms:

- Room (1, 0): it is already available (time 0), so you can enter at $\max(0, 0) + 1 = 1$
- Room (0, 1): also available, enter at $\max(0, 0) + 1 = 1$

Push both:

```
(1, 1, 0), (1, 0, 1)
```

Step 2:

Pop room (1, 0, 1)

From here, move to:

- Room (0, 2): available, reach at $1 + 1 = 2$

Push:

```
(2, 0, 2)
```

Step 3:

Pop room (1, 1, 0)

From here, move to:

- Room (1, 1): available, reach at $1 + 1 = 2$

Push:

(2, 1, 1)

Step 4:

Pop room (2, 0, 2)

From here, move to:

- Room (1, 2): available, reach at $2 + 1 = 3$

Push:

(3, 1, 2)

Step 5:

Pop room (2, 1, 1) — from here, we could also reach (1, 2), but it's already scheduled at time 3, so we skip.

Step 6:

Pop room (3, 1, 2) → this is the destination.

You have reached the bottom-right room at **time 3**.

Output: 3

Lab 10

Problem 1

You are tasked with helping a maintenance worker move between floors in a building. Each floor of the building has a cost associated with moving to it. The worker can either go up one floor at a time or skip one floor and go up two floors at once.

You need to determine the minimum cost for the worker to reach the top of the building, given that they can either start at the first floor or the second floor.

Input:

- A list `cost` of integers, where `cost[i]` is the cost for moving to floor `i`. The worker can either move one floor up (paying the cost of the current floor) or skip one floor and go two floors up (paying the cost of the skipped floor and the next one).

Output:

- The minimum total cost for the worker to reach the top of the building.

Task:

1. Explain how you would use dynamic programming to calculate the minimum cost for the worker to reach the top of the building.
2. With the given example `cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`, explain step-by-step how the dynamic programming approach works to calculate the minimum cost of reaching the top.

Answer

Algorithm Approach

To solve this problem efficiently, dynamic programming (DP) is an ideal approach because the problem involves optimal substructure and overlapping subproblems. Let's break it down:

Why Dynamic Programming?

1. **Subproblems:**
The problem involves calculating the minimum cost to reach the top of the building, but instead of calculating the total cost in one go, we can break it into smaller steps. The worker can either:

- Move one floor up and pay the cost of the current floor, or
 - Skip a floor and move two floors up, paying the cost of the skipped floor and the next one.
2. For each floor, we need to make a decision based on the costs of the two previous floors, so each floor's minimum cost depends on the minimum costs of the two previous floors.
 3. **Optimal Substructure:**
The problem exhibits optimal substructure, meaning that the minimum cost to reach a given floor is the minimum of the costs to reach the previous two floors. This allows us to break the problem into smaller subproblems that are easier to solve.
 4. **Overlapping Subproblems:**
The cost to reach any given floor will be recalculated multiple times if solved naively. By using dynamic programming, we can store the results of each subproblem (the minimum cost to reach each floor) in the `cost` array, which prevents redundant calculations and improves efficiency.

Dynamic Programming Approach:

1. **Initialization:**
We start by defining the `cost[]` array, where `cost[i]` will store the minimum cost to reach floor `i`. This is done in-place, so we update the original `cost[]` array.
2. **Recurrence Relation:**
For each floor `i` (starting from the 2nd step), the minimum cost can be computed as:
`cost[i] += min(cost[i-1], cost[i-2])`

This means that the cost to reach floor `i` is the current floor's cost `cost[i]` plus the minimum of the costs to reach the previous two floors.
3. **Final Step:**
The worker can either reach the top from the last floor (`cost[n-1]`) or from the second-to-last floor (`cost[n-2]`). Thus, the final result is:
`min(cost[n-1], cost[n-2])`

Step by step example

Let's walk through the dynamic programming approach with the example `cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`:

1. **Initialization:**

- Start by initializing the `cost` array, where `cost[i]` represents the minimum cost to reach the `i`th floor. The `cost[]` array will be updated in-place.
`cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`

2. Base Cases:

- The cost of the first floor (`cost[0]`) is already given as 1.
- The cost of the second floor (`cost[1]`) is 100, as given.

3. Iterating through the floors and updating the cost array:

- **Floor 2 (i=2):** The worker can either come from floor 1 or floor 0. The cost will be:
`cost[2] += min(cost[1], cost[0]) # cost[2] = 1 + min(100, 1) = 2`
 Updated `cost` array: `[1, 100, 2, 1, 1, 100, 1, 1, 100, 1]`
- **Floor 3 (i=3):** The worker can either come from floor 2 or floor 1:
`cost[3] += min(cost[2], cost[1]) # cost[3] = 1 + min(2, 100) = 3`
 Updated `cost` array: `[1, 100, 2, 3, 1, 100, 1, 1, 100, 1]`
- **Floor 4 (i=4):** The worker can either come from floor 3 or floor 2:
`cost[4] += min(cost[3], cost[2]) # cost[4] = 1 + min(3, 2) = 3`
 Updated `cost` array: `[1, 100, 2, 3, 3, 100, 1, 1, 100, 1]`
- **Floor 5 (i=5):** The worker can either come from floor 4 or floor 3:
`cost[5] += min(cost[4], cost[3]) # cost[5] = 100 + min(3, 3) = 103`
 Updated `cost` array: `[1, 100, 2, 3, 3, 103, 1, 1, 100, 1]`
- **Floor 6 (i=6):** The worker can either come from floor 5 or floor 4:
`cost[6] += min(cost[5], cost[4]) # cost[6] = 1 + min(103, 3) = 4`
 Updated `cost` array: `[1, 100, 2, 3, 3, 103, 4, 1, 100, 1]`
- **Floor 7 (i=7):** The worker can either come from floor 6 or floor 5:
`cost[7] += min(cost[6], cost[5]) # cost[7] = 1 + min(4, 103) = 5`

Updated `cost` array: [1, 100, 2, 3, 3, 103, 4, 5, 100, 1]

- **Floor 8 (i=8):** The worker can either come from floor 7 or floor 6:
`cost[8] += min(cost[7], cost[6])` # `cost[8] = 100 + min(5, 4) = 104`

Updated `cost` array: [1, 100, 2, 3, 3, 103, 4, 5, 104, 1]

- **Floor 9 (i=9):** The worker can either come from floor 8 or floor 7:
`cost[9] += min(cost[8], cost[7])` # `cost[9] = 1 + min(104, 5) = 6`

Updated `cost` array: [1, 100, 2, 3, 3, 103, 4, 5, 104, 6]

4. **Final Step:** After processing all the floors, the minimum cost to reach the top of the building is:

`min(cost[9], cost[8])` # `min(6, 104) = 6`

The minimum cost to reach the top is **6**.

Problem 2

You are working on a network monitoring tool that tracks the number of active devices in a network at any given time. Each device in the network has a unique identifier, and the identifiers are represented as integers. These identifiers can be seen as a sequence of numbers, where each number's binary representation shows which devices are active at specific time intervals.

Your task is to write a function that, for each integer from 0 to `n`, returns the number of devices that are active, which corresponds to the number of 1s in the binary representation of each number.

The function should generate a list where each index `i` (from 0 to `n`) tells you how many devices are active (i.e., how many 1s are in the binary representation of `i`).

Input:

- An integer `n` which represents the highest device identifier you want to track.

Output:

- A list `ans` of length `n + 1`, where `ans[i]` is the number of active devices (the number of 1s) in the binary representation of the number `i`.

Example:

Input: $n = 2$

Output: $[0, 1, 1]$

Explanation:

- The binary representation of 0 is 0, which has 0 1s (no active devices).
- The binary representation of 1 is 1, which has 1 1 (one active device).
- The binary representation of 2 is 10, which has 1 1 (one active device).

Task:

1. How can you solve this problem efficiently using dynamic programming that runs in $O(n)$ time?
2. For the given example $n = 5$, explain how you would calculate the binary representations of each number from 0 to 5 and count the number of 1s in each binary number to build the final output array.

Answer

How can you solve this problem efficiently in $O(n)$ time? Explain how to reach that approach.

We are asked to count how many 1s appear in the binary representation of each number from 0 to n . A straightforward way is to convert each number to binary and count the 1s one by one. However, that takes more time—about $O(n \log n)$ because converting to binary and counting takes \log time per number.

To make it faster, we can use **dynamic programming** by recognizing a simple pattern in binary numbers.

Let's think step-by-step:

1. **Observe the pattern:**
 - Every time you divide a number by 2 (which in binary means shifting bits to the right), you remove the least significant bit (LSB).
 - If the number is **even**, that bit is 0, so the number of 1s stays the same.

- If the number is **odd**, that bit is **1**, so the number of **1s** is **1 more** than its half.

2. Example:

- 2 in binary is 10 → same number of 1s as 1 (1 in binary is 1)
- 3 in binary is 11 → one more 1 than 1
- 5 is 101 → one more 1 than 2 (10)

So we see:

`countBits(2) = countBits(1) → because 2 is even`

`countBits(3) = countBits(1) + 1 → because 3 is odd`

`countBits(4) = countBits(2) → even`

`countBits(5) = countBits(2) + 1 → odd`

3. From this pattern, we can define the formula:

`countBits(i) = countBits(i // 2) + (i % 2)`

- If **i** is **even** → `i % 2 = 0`, so it reuses the same count as `i // 2`
- If **i** is **odd** → `i % 2 = 1`, so we add one more to the count of `i // 2`

4. Dynamic programming idea:

- We build an array **t** where `t[i]` stores the number of 1s for number **i**
- Start from `t[0] = 0` and build up to `t[n]` using the formula above
- Since we reuse already computed values, this gives us **O(n)** time complexity

Using the input `n = 5`, explain step-by-step how to calculate the result.

Let's build the result step-by-step using the rule:

`t[i] = t[i // 2] + (i % 2)`

We initialize `t[0] = 0` and go up to `t[5]`:

- `t[0] = 0` (binary 0)
- `t[1] = t[0] + 1 = 0 + 1 = 1` (binary 1)
- `t[2] = t[1] + 0 = 1 + 0 = 1` (binary 10)
- `t[3] = t[1] + 1 = 1 + 1 = 2` (binary 11)

- $t[4] = t[2] + 0 = 1 + 0 = 1$ (binary 100)
- $t[5] = t[2] + 1 = 1 + 1 = 2$ (binary 101)

So the final result is:

[0, 1, 1, 2, 1, 2]

Problem 3

Imagine you are programming a robot to move across a factory floor. The floor is divided into checkpoints, and each checkpoint indicates how far the robot can jump forward. The robot starts at the first checkpoint and aims to reach the last one.

Each checkpoint has a number that represents the **maximum steps** the robot can jump from there. The robot can jump from 1 up to the maximum number of steps allowed at each checkpoint. You need to determine if the robot can reach the last checkpoint starting from the first one.

Input:

- A list `nums` of integers where `nums[i]` is the maximum number of steps the robot can take from checkpoint `i`.

Output:

- A boolean value: `True` if the robot can reach the last checkpoint, otherwise `False`.

Task:

1. Explain how you would solve this problem using dynamic programming.
2. With the given examples `nums = [2, 3, 1, 1, 4]` and `nums = [3, 2, 1, 0, 4]`, explain step-by-step how the approach works to determine if the robot can reach the last checkpoint.

Answer

This problem is about determining if a robot can reach the last checkpoint, starting from the first, where each checkpoint specifies how far the robot can jump.

Why Use Dynamic Programming?

1. **Optimal Substructure:**
The ability to reach a checkpoint depends on whether you can reach any earlier

checkpoint. Each subproblem (checkpoint) contributes to the overall solution (reaching the last checkpoint).

2. **Overlapping Subproblems:**

Instead of recalculating whether each checkpoint is reachable multiple times, we store the farthest index (`max_reach`) the robot can reach at any point. This avoids redundant work, similar to DP where results of subproblems are stored and reused.

How It Relates to Greedy:

- **Greedy Aspect:**

At each checkpoint, the robot chooses to jump the maximum possible steps. This is a greedy choice that aims to reach the farthest point, maximizing the chance of reaching the last checkpoint.

- **DP Aspect:**

While we don't use a traditional DP table, we keep track of the maximum reachable index (`max_reach`), which acts like storing subproblem results. By updating `max_reach` as we go, we avoid recomputing reachability for each checkpoint.

How the Code Works:

In the code, we track the farthest point the robot can reach using the variable `max_reach`. Let's break down the approach:

1. **Initialization:**

We start with `max_reach = 0`, meaning we begin at the first checkpoint.

2. **Iterate through the list:** We loop through each checkpoint (`i`), and at each checkpoint, we check if it is reachable:

- If `i > max_reach`, it means we can't reach this checkpoint, so we return `False`.

3. **Update `max_reach`:**

At each checkpoint `i`, we update `max_reach`:

- `max_reach = max(max_reach, i + nums[i])`: This line means that from the current checkpoint `i`, the robot can jump as far as `i + nums[i]`, and we update `max_reach` to reflect the farthest point we can now reach.

4. **Check if we can reach the last checkpoint:**

If `max_reach >= len(nums) - 1`, it means we can reach or exceed the last checkpoint, so we return `True`.

5. **Final Return:**

If after iterating through all the checkpoints we never reach the last index, we return `False`.

Step-by-step Explanation Using the Given Examples `nums = [2, 3, 1, 1, 4]` and `nums = [3, 2, 1, 0, 4]`:

Example 1: `nums = [2, 3, 1, 1, 4]`

1. Initialization:

- `max_reach = 0` (starting at the first checkpoint).

2. Step 1 (i = 0):

- The robot is at checkpoint 0, and it can jump 2 steps.
- Update `max_reach = max(0, 0 + nums[0]) = 2`.
- The robot can now reach up to checkpoint 2.

3. Step 2 (i = 1):

- The robot is at checkpoint 1, and it can jump 3 steps.
- Update `max_reach = max(2, 1 + nums[1]) = 4`.
- The robot can now reach up to checkpoint 4, which is the last checkpoint.

4. Early Termination:

- Since `max_reach = 4` is greater than or equal to the last index, return `True`.

Example 2: `nums = [3, 2, 1, 0, 4]`

1. Initialization:

- `max_reach = 0` (starting at the first checkpoint).

2. Step 1 (i = 0):

- The robot is at checkpoint 0, and it can jump 3 steps.
- Update `max_reach = max(0, 0 + nums[0]) = 3`.

- The robot can now reach up to checkpoint 3.
3. **Step 2 (i = 1):**
- The robot is at checkpoint 1, and it can jump 2 steps.
 - Update `max_reach = max(3, 1 + nums[1]) = 3`.
 - The robot can still reach up to checkpoint 3.
4. **Step 3 (i = 2):**
- The robot is at checkpoint 2, and it can jump 1 step.
 - Update `max_reach = max(3, 2 + nums[2]) = 3`.
 - The robot can still reach up to checkpoint 3.
5. **Step 4 (i = 3):**
- The robot is at checkpoint 3, and it can jump 0 steps.
 - Since `i = 3` is the farthest the robot can reach and `max_reach` is 3, the robot is stuck and cannot move forward.
6. **Final Check:**
- Since `max_reach` never exceeds or equals the last index (4), return `False`.

Problem 4

You are managing a rectangular garden represented as a 2D grid. Each cell in the grid is either:

- `1` (indicating that the plot is **healthy and green**), or
- `0` (indicating a **damaged or unusable** plot).

Your goal is to find out how many **square plots** (subgrids) of any size can be formed such that **all cells in the square are green (1)**.

A square plot must be fully green — all cells in it must be 1s — and can be of any size 1×1 , 2×2 , ..., up to the size that fits within the garden.

Example:

Input:

```
garden = [
    [0, 1, 1, 1],
    [1, 1, 1, 1],
    [0, 1, 1, 1]
]
```

Output: 15

Explanation:

- There are 10 squares of size 1×1 (each individual 1).
- 4 squares of size 2×2 .
- 1 square of size 3×3 .
- Total = $10 + 4 + 1 = 15$

Task:

1. How would you solve this problem using dynamic programming?
2. Use the example `garden = [[1, 0, 1], [1, 1, 0], [1, 1, 0]]` to walk through how the DP table is built step-by-step and how the final result is calculated.

Answer

Understanding the Common Approach

Let's begin by understanding what we're trying to do.

We are given a **binary matrix** — that is, a 2D list of 0s and 1s — for example:

```
matrix = [
    [1, 1, 1],
    [1, 1, 1],
    [0, 1, 1]
```

]

Each cell represents a tile of land (or garden plot, image pixel, etc.). A 1 means the cell is *usable*, and a 0 means it is *not usable*.

Our Goal

We want to count how many **square submatrices** (small square-shaped blocks) we can find in the matrix such that **all elements in the square are 1**.

These squares can be of size:

- 1×1 (single cell),
- 2×2 (block of 4),
- 3×3 (block of 9), etc.

What does it mean for a square to “end at (i, j)”?

This is the key idea in the DP approach, so let's clarify:

Example:

If we say **a square ends at position (2, 2)**, we mean:

- The **bottom-right corner** of the square is at cell (2, 2) (i.e., `matrix[2][2]`).
- The rest of the square stretches **upwards** and **to the left** from that cell.

For example, a 2×2 square ending at (2, 2) looks like this:

```
[1, 1]
[1, 1] ← ends here at (2, 2)
```

This square includes:

- (1, 1), (1, 2)
- (2, 1), (2, 2)

So we say: **a square of size 2×2 ends at (2, 2)**.

Core DP Idea

We define a DP table where:

`dp[i][j]` represents the **size of the largest square submatrix of all 1s that ends at cell (i, j)**.

This square must include cell `(i, j)` as the **bottom-right corner**.

Example:

Let's say:

`dp[2][2] = 3`

This means: there is a **3×3 square** of all 1s that ends at `matrix[2][2]`.

How do we build the DP table?

We use the rule:

```
if matrix[i][j] == 1:
    dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
else:
    dp[i][j] = 0
```

Why these three neighbors?

To form a square ending at `(i, j)`, you need:

- The cell **above**: `dp[i-1][j]` (same column, one row up)
- The cell **to the left**: `dp[i][j-1]` (same row, one column left)
- The **top-left diagonal** cell: `dp[i-1][j-1]`

These three cells determine the **maximum possible square size** you can extend to `(i, j)`.

How to get the total number of squares?

Once we've built the DP table:

- Each `dp[i][j]` tells you how many squares end at that position.
 - A `dp[i][j] = 3` means:
 - One 3×3 square ends here,
 - One 2×2 square (inside the 3×3) also ends here,
 - One 1×1 square also ends here.

So we add all `dp[i][j]` values across the matrix to get the **total number of square submatrices with all 1s**.

Summary of the Common DP Approach

- Build a DP table where `dp[i][j] = size of largest square ending at (i, j)`
- Use the recurrence:


```
dp[i][j] = 1 + min(top, left, top-left)  if matrix[i][j] == 1
dp[i][j] = 0                             if matrix[i][j] == 0
```
- Sum all `dp[i][j]` values to get the final answer

Step-by-Step Walkthrough

Input matrix (garden):

```
[
  [1, 0, 1],
  [1, 1, 0],
  [1, 1, 0]
]
```

We will initialize a DP table of the same size:

```
dp = [
  [0, 0, 0],
  [0, 0, 0],
  [0, 0, 0]
]
```

We also initialize `total = 0` to store the number of squares.

Fill the DP table

Row 0:

- **(0,0):** `garden[0][0] = 1` → first row, so `dp[0][0] = 1` → total = 1
- **(0,1):** `garden[0][1] = 0` → `dp[0][1] = 0` → total = 1
- **(0,2):** `garden[0][2] = 1` → first row, so `dp[0][2] = 1` → total = 2

dp after row 0:

```
[
  [1, 0, 1],
  [0, 0, 0],
  [0, 0, 0]
]
```

Row 1:

- **(1,0):** `garden[1][0] = 1` → first column, so `dp[1][0] = 1` → total = 3
- **(1,1):** `garden[1][1] = 1`
→ check min of top (`dp[0][1]=0`), left (`dp[1][0]=1`), and top-left (`dp[0][0]=1`)
→ `dp[1][1] = 1 + min(0, 1, 1) = 1` → total = 4
- **(1,2):** `garden[1][2] = 0` → `dp[1][2] = 0` → total = 4

dp after row 1:

```
[
  [1, 0, 1],
  [1, 1, 0],
  [0, 0, 0]
]
```

Row 2:

- **(2,0):** `garden[2][0] = 1` → first column, so `dp[2][0] = 1` → total = 5

- **(2,1):** garden[2][1] = 1
 → check min of top (dp[1][1]=1), left (dp[2][0]=1), top-left (dp[1][0]=1)
 → $dp[2][1] = 1 + \min(1, 1, 1) = 2 \rightarrow \text{total} = 7$
- **(2,2):** garden[2][2] = 0 → $dp[2][2] = 0 \rightarrow \text{total} = 7$

Final dp table:

```
[
  [1, 0, 1],
  [1, 1, 0],
  [1, 2, 0]
]
```

Final Calculation

To get the **total number of square submatrices with all 1s**, sum all values in the **dp** table:

$$1 + 0 + 1 + 1 + 1 + 0 + 1 + 2 + 0 = 7$$

Problem 5

You are working on a video compression algorithm. A video is broken into **n** frames, and each frame has a **complexity score** represented by an integer in an array **frames**. To compress the video, you can **group contiguous frames** into segments, with each segment being **at most k frames long**.

For every segment you create, the compression algorithm simplifies all frames in that segment by assigning them the **maximum complexity** within that segment (since the hardest frame to compress defines the difficulty of the segment). The **total cost** of compression is the **sum of all new frame values after partitioning**.

Your task is to find the **maximum total cost** you can get by optimally partitioning the video into segments of length at most **k**.

Example

Input:

```
frames = [1, 15, 7, 9, 2, 5, 10]
k = 3
```

Output: 84

Explanation:

- One optimal way to partition is: `[15,15,15] [9] [10,10,10]`
- Sum = $15+15+15 + 9 + 10+10+10 = 84$

Task:

1. Explain how you would solve this problem using dynamic programming in $O(nk)$ time
2. Use the input `frames = [1, 4, 1, 5, 7, 3, 6, 1, 9, 9, 3]`, `k = 4` to walk through how your approach calculates the result step by step.

Answer

To solve this problem efficiently, we use **dynamic programming**. But before jumping into code or formulas, let's understand how to think about the problem and how this approach naturally fits.

How to think about the problem

We are given an array of frame complexities, and we can group **contiguous frames** into segments of at most `k` frames. Each frame in a segment is replaced by the **maximum value** of that segment, and our goal is to **maximize the total sum** of the resulting array.

When you're at a certain index `i` in the array (say frame `i`), you don't know yet how long your current segment should be — it could be:

- just `frames[i]`
- or `[frames[i], frames[i+1]]`
- or up to `k` frames (as long as it doesn't go past the end).

For each of these choices, you:

1. **Calculate the group score:** take the max value in the group \times the number of frames in it.
2. **Add the best total score** from whatever comes **after** this group — that's what we've already stored in `dp[i + length]`.

The best choice at i is the one that gives you the highest **combined score**: current group score **plus** best future score.

This is why we use **dynamic programming**:

- We store answers for future positions (like $dp[i+1]$, $dp[i+2]$...) so we don't have to recalculate them.
- We build the best answer for each index by **reusing** the best answers we've already computed for later positions.

Defining the DP Approach

- Let $dp[i]$ represent the **maximum total compression cost** starting from index i .
- We build this solution **from the end to the beginning** of the array.

At each position i :

- Try all possible group lengths from 1 to k .
- For each possible group:
 - Keep track of the **maximum frame value** within the group.
 - Compute the group cost = $\text{max_value} \times \text{group_size}$.
 - Combine it with the **previously computed** $dp[i + \text{group_size}]$.
 - Take the **maximum** total score among all options.

Why solve from the end?

To compute $dp[i]$, we need $dp[i+1]$, $dp[i+2]$, ..., depending on the group size. That means we must solve the **later parts of the array first**, then work our way backward.

Step by step with input $\text{frames} = [1, 4, 1, 5, 7, 3, 6, 1, 9, 9, 3]$, $k = 4$

Let's trace the dynamic programming computation step by step.

We'll go **from the end to the beginning**, updating $dp[start]$ at each step. Here's a high-level view of what happens:

- At each $start$ index, we try all segment lengths 1 to k (up to $k = 4$), calculate the best segment ending at that point, and store the best result.

Example highlights:

- **start = 10** (value = 3):
Only one possible segment $[3]$, cost = $1 \times 3 = 3 \rightarrow dp[10] = 3$
- **start = 9** (value = 9):
Try:
 - $[9] \rightarrow \text{cost} = 9 + dp[10] = 9 + 3 = 12$
 - $[9, 3] \rightarrow \text{max} = 9, \text{len} = 2 \rightarrow 18$
 $\rightarrow dp[9] = 18$
- **start = 8** (value = 9):
Try:
 - $[9] \rightarrow 9 + dp[9] = 9 + 18 = 27$
 - $[9, 9] \rightarrow 9 \times 2 + dp[10] = 18 + 3 = 21$
 - $[9, 9, 3] \rightarrow 9 \times 3 + dp[11] = 27 + 0 = 27$
 $\rightarrow dp[8] = 27$
- **start = 7** (value = 1):
Try:
 - $[1] \rightarrow 1 + dp[8] = 1 + 27 = 28$
 - $[1, 9] \rightarrow \text{max} = 9, \text{len} = 2 \rightarrow 18 + dp[9] = 18 + 18 = 36$
 - $[1, 9, 9] \rightarrow 9 \times 3 + dp[10] = 27 + 3 = 30$
 - $[1, 9, 9, 3] \rightarrow 9 \times 4 = 36 + dp[11] = 36$
 $\rightarrow dp[7] = 36$

- And so on, all the way to `start = 0`.

Finally, `dp[0] = 83`, which is the **maximum total compression score** achievable.

Lab 11

Problem 1

You are given a short story and a pattern. Your task is to find all occurrences of the pattern in the story using the brute-force string matching algorithm. The story is a string of text, and you need to find the starting indices where the pattern appears in the story.

Input:

- **Story:** "Alice walked into the forest. Alice wondered what she would find. Suddenly, Alice saw a rabbit."
- **Pattern:** "Alice"

Task:

Describe **step by step** how you would apply the brute-force string matching algorithm to find all occurrences of the pattern "Alice" in the given story.

Answer

Step 1: Align the pattern at the beginning of the text

We start by aligning the pattern "Alice" with the first position of the text, which is the beginning of the string "Alice walked into the forest. Alice wondered what she would find. Suddenly, Alice saw a rabbit.".

Step 2: Compare each character of the pattern to the corresponding character in the text

We will now compare each character of the pattern "Alice" with the corresponding character in the text.

- **First alignment** (at the start of the text):
 - Compare "A" (pattern) with "A" (text) → match.
 - Compare "l" (pattern) with "l" (text) → match.
 - Compare "i" (pattern) with "i" (text) → match.

- Compare "c" (pattern) with "c" (text) → match.
- Compare "e" (pattern) with "e" (text) → match.
- All characters match. So, the pattern "Alice" matches at the **first position** of the text, which is index **0** (0-based index).

Step 3: Move the pattern one position to the right

Now that we have matched the first occurrence, we move the pattern one position to the right and repeat the comparison.

- **Second alignment** (shift the pattern one position to the right):
 - The second alignment starts at position 1 in the text: "lice walked into the forest. Alice wondered what she would find. Suddenly, Alice saw a rabbit.".
 - Compare "A" (pattern) with "l" (text) → mismatch.
- Since there's a mismatch, we move the pattern one more position to the right and start over.

Step 4: Continue shifting and comparing

We continue shifting the pattern one position to the right and comparing until we reach the next successful match.

- **Alignment** (shift the pattern to position 30):
 - The fourth alignment starts at position 30 in the text: "Alice wondered what she would find. Suddenly, Alice saw a rabbit.".
 - Compare "A" (pattern) with "A" (text) → match.
 - Compare "l" (pattern) with "l" (text) → match.
 - Compare "i" (pattern) with "i" (text) → match.
 - Compare "c" (pattern) with "c" (text) → match.
 - Compare "e" (pattern) with "e" (text) → match.

- All characters match again. So, the pattern "Alice" matches at position **30** (0-based index).

5. **Alignment** (shift the pattern to position 76):

- The fifth alignment starts at position 62 in the text: "Alice saw a rabbit."
- Compare "A" (pattern) with "A" (text) → match.
- Compare "l" (pattern) with "l" (text) → match.
- Compare "i" (pattern) with "i" (text) → match.
- Compare "c" (pattern) with "c" (text) → match.
- Compare "e" (pattern) with "e" (text) → match.

6. All characters match again. So, the pattern "Alice" matches at position **76** (0-based index).

Conclusion

- The pattern "Alice" is found at positions **0**, **30**, and **76** in the text.

Problem 2

You are analyzing customer reviews to check if any of them mention a specific complaint keyword.

- The **pattern** you're searching for is: LATE
- The **text** from a feedback entry is:
"THE DELIVERY WAS VERY LATE AND DAMAGED THE PACKAGE"

To efficiently find the pattern, you decide to apply **Horspool's Algorithm**, which is optimized for fast string searching using a **shift table** and right-to-left comparisons.

Tasks:

1. **Build the shift table** for the pattern LATE using Horspool's rule:
2. **Step by step**, describe how Horspool's Algorithm compares characters and performs shifts on the given text.

Answer

Build the Shift Table for Pattern **LATE**

Pattern: **LATE**

Length of the pattern: 4

We compute shift values for all characters **except** the last character (**E**), using the formula:

$$\text{shift} = \text{pattern length} - \text{position} - 1$$

Character	Position	Shift
L	0	$4 - 0 - 1 = 3$
A	1	$4 - 1 - 1 = 2$
T	2	$4 - 2 - 1 = 1$

For all other characters not in the pattern, the default shift value is 4 (the length of the pattern).

Final Shift Table:

L: 3

A: 2

T: 1

All others: 4

Step-by-Step Execution of Horspool's Algorithm

Text: "THE DELIVERY WAS VERY LATE AND DAMAGED THE PACKAGE"

Pattern: "LATE"

Pattern length: 4

We align the pattern with the text and compare characters from **right to left**. The algorithm shifts the pattern based on the character in the text that aligns with the last character of the pattern (**E**).

Step 1

Text index 0–3: THE

Pattern: LATE

- Compare **E** (pattern) with ' ' (text[3]): mismatch
- ' ' is not in the pattern → shift = 4

- New alignment starts at index 4

Step 2

Text index 4–7: DELI

Pattern: LATE

- Compare E with I (text[7]): mismatch
- I is not in the pattern → shift = 4
- New alignment starts at index 8

Step 3

Text index 8–11: VERY

Pattern: LATE

- Compare E with Y (text[11]): mismatch
- Y is not in the pattern → shift = 4
- New alignment starts at index 12

Step 4

Text index 12–15: WAS

Pattern: LATE

- Compare E with S (space at text[15]): mismatch
- S is not in the pattern → shift = 4
- New alignment starts at index 16

Step 5

Text index 16–19: VER

Pattern: LATE

- Compare E with R (text[19]): mismatch
- R is not in the pattern → shift = 4
- New alignment starts at index 20

Step 6

Text index 16–19: Y LA

Pattern: LATE

- Compare E with A (text[23]): mismatch
- A is in the pattern → shift = 2
- New alignment starts at index 22

Step 6

Text index 22–25: LATE

Pattern: LATE

- Compare E with E (text[25]): match
- Compare T with T (text[24]): match
- Compare A with A (text[23]): match
- Compare L with L (text[22]): match
- All characters match → pattern found at index 22

Conclusion: The pattern LATE is found at position 22 in the text using Horspool's Algorithm.

Problem 3

You are implementing an auto-complete system that uses a **Trie** data structure. The system must support inserting words, checking if a word exists, and verifying whether any word starts with a given prefix.

Given the following list of words:

["cat", "cap", "can", "camp", "camera"]

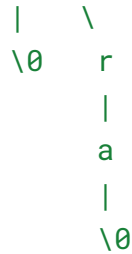
Tasks:

1. Draw the **Trie structure** that results from inserting all the given words.
2. Use the Trie to determine whether the prefix "cam" exists in the structure.
3. Check whether the word "capo" exists in the Trie.

Answer

Trie Structure

```
(root)
  |
  c
  |
  a
 /  /  \  \
t p  n  m
|  |  |  |  \
\0\0 \0 p  e
```



Check if the Prefix "cam" Exists

Perform a prefix traversal from the root, following the characters 'c' → 'a' → 'm'. Since this is a prefix check, we only need to confirm that each character node exists; we do not need to find a \0 marker.

Steps:

1. Start at the root.
2. Go to child 'c' → exists.
3. From 'c', go to child 'a' → exists.
4. From 'a', go to child 'm' → exists.

Conclusion: All characters of the prefix "cam" are found.

Result: The prefix "cam" **exists** in the Trie.

Check if the Word "capo" Exists

Perform a word-level traversal: 'c' → 'a' → 'p' → 'o', and ensure that the last character leads to a \0 marker to denote a full word. If any character is missing or no termination is found, the word does not exist.

Steps:

1. Start at the root.
2. Go to child 'c' → exists.
3. From 'c', go to child 'a' → exists.
4. From 'a', go to child 'p' → exists.
5. From 'p', attempt to go to child 'o' → **not found**.

Conclusion: The character 'o' is missing in the path.

Result: The word "capo" **does not exist** in the Trie.

Problem 4

In a genetics research lab, scientists are analyzing DNA sequences to detect specific **harmful gene patterns** associated with increased risk of hereditary disorders. You are given a DNA sequence and a known **harmful gene pattern**, and your task is to find all locations where this pattern occurs in the sequence using an **efficient string matching algorithm**.

Each occurrence of the harmful pattern corresponds to a potential mutation site. If the pattern appears **more than twice** in the sequence, the DNA is considered at risk for **genetic instability**, which may contribute to cancer or other genetic disorders.

Input:

- **DNA sequence:**
"ATCGTACGATCGGATCATCG"
- **Harmful gene pattern:**
"ATCG"

Task:

1. Use an **efficient string matching algorithm** to find all starting indices (0-based) where the pattern occurs in the DNA sequence.
2. Describe **step by step** how your algorithm searches for the pattern in the DNA string. Based on the number of matches found, determine whether the sequence should be flagged for **genetic instability**.

Answer

We are given:

- DNA sequence: "ATCGTACGATCGGATCATCG"
- Harmful gene pattern: "ATCG"

We will use an efficient string matching algorithm, specifically **Horspool's Algorithm**, to find all occurrences of the pattern in the DNA sequence.

Step 1: Construct the Shift Table

Horspool's algorithm uses a shift table to determine how far the pattern can be moved when a mismatch occurs. The shift value for each character in the alphabet (in this case, A, T, C, G) is calculated based on its rightmost position in the pattern, excluding the last character.

Pattern: "ATCG" (length = 4)

We exclude the last character 'G' when building the table.

We compute the shift values as follows:

- 'A' is at index 0 \rightarrow shift = 3 ($4 - 1 - 0$)
- 'T' is at index 1 \rightarrow shift = 2 ($4 - 1 - 1$)
- 'C' is at index 2 \rightarrow shift = 1 ($4 - 1 - 2$)
- 'G' does not appear in the first three characters \rightarrow shift = 4 (pattern length)

Final shift table:

A: 3

T: 2

C: 1

others (including G): 4

Step 2: Apply Horspool's Algorithm

We align the pattern with the text and compare characters from right to left. If the characters all match, we record the position. If a mismatch occurs, we shift the pattern according to the shift table using the character aligned with the last character of the pattern.

We compare substrings of length 4 from index 0 to 16.

Detailed steps:

1. Index 0 \rightarrow Text window: "ATCG"
 - $G == G \rightarrow$ match
 - $C == C \rightarrow$ match
 - $T == T \rightarrow$ match
 - $A == A \rightarrow$ match
Pattern found at index 0
Shift by $\text{shift}[G] = 4 \rightarrow$ Next index: 4

2. Index 4 → Text window: "TACG"

- $G == G \rightarrow \text{match}$
- $C == C \rightarrow \text{match}$
- $A \neq T \rightarrow \text{mismatch}$
Shift by $\text{shift}[G] = 4 \rightarrow \text{Next index: } 8$

3. Index 8 → Text window: "ATCG"

- $G == G \rightarrow \text{match}$
- $C == C \rightarrow \text{match}$
- $T == T \rightarrow \text{match}$
- $A == A \rightarrow \text{match}$
Pattern found at index 8
Shift by $\text{shift}[G] = 4 \rightarrow \text{Next index: } 12$

4. Index 12 → Text window: "GATC"

- $C \neq G \rightarrow \text{mismatch}$
Shift by $\text{shift}[C] = 1 \rightarrow \text{Next index: } 13$

5. Index 13 → Text window: "ATCA"

- $A \neq G \rightarrow \text{mismatch}$
Shift by $\text{shift}[A] = 3 \rightarrow \text{Next index: } 16$

6. Index 16 → Text window: "ATCG"

- $G == G \rightarrow \text{match}$
- $C == C \rightarrow \text{match}$
- $T == T \rightarrow \text{match}$
- $A == A \rightarrow \text{match}$
Pattern found at index 16
Shift by $\text{shift}[G] = 4 \rightarrow \text{End of string}$

Step 3: Final Result

The pattern "ATCG" is found at the following indices in the DNA sequence:
[0, 8, 16]

Step 4: Interpretation

Since the harmful gene pattern appears more than two times, the DNA sequence is flagged as potentially unstable.