Walchand college of engineering

***Department of computer science***

**Batch: T3**

**Assignment No. 2**

**Searching Algorithm (Part 1)**

**Name: Piyush Dawkhare**

**PRN: 23510056**

**Problem Statement 1:**

You are an IT company's manager. Based on their performance over the last N working days, you must rate your employee. You are given an array of N integers called workload, where workload[i] represents the number of hours an employee worked on an ith day. The employee must be evaluated using the following criteria: ∙ Rating = the maximum number of consecutive working days when the employee has worked more than 6 hours. You are given an integer N where N represents the number of working days. You are given an integer array workload where workload[i] represents the number of hours an employee worked on an ith day. Task Determine the employee rating.

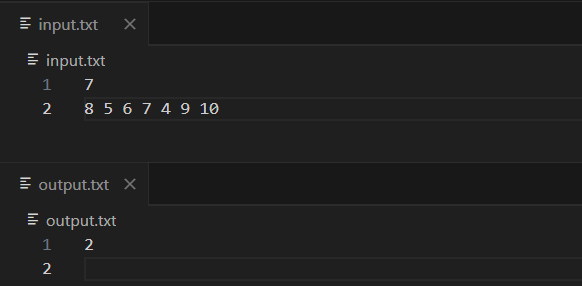
**Algorithms Used**

* **Linear Search / Sliding Window Approach Related Theory**
* The problem is essentially finding the longest consecutive subsequence of elements satisfying a condition (workload[i] ≥ 6).
* This is a classical problem that can be solved using:
  + Kadane’s-like idea for subsequences, where we maintain a counter for consecutive valid elements and reset it when the condition fails.
  + Time Complexity: O(N) since it traverses the array only once.
  + Space Complexity: O(1) since only a few variables (c, ans) are used.

**Algorithm / Procedure**

* Initialize two variables:
* c = 0 (to count current consecutive days worked ≥ 6 hours).
* ans = 0 (to store the maximum streak so far).
* Loop through each day:
* If workload[i] ≥ 6, increment c (continue the streak).
* Else, update ans = max(ans, c) and reset c = 0.
* After the loop, update ans = max(ans, c) again (to account for the last streak).
* Return ans as the employee rating.
* Print the result.

**Output**

****

**Problem Statement 2:**

You have N boxes numbered 1 through N and K candies numbered 1 through K. You put the candies in the boxes in the following order: ∙ first candy in the first box, ∙ second candy in the second box, ∙ ....... ∙ ∙ so up to N-th candy in the Nth box, ∙ the next candy in (N - 1)-th

box, ∙ the next candy in (N - 2)-th box ∙ ........ ∙ ∙ and so on up to the first box, ∙ then the

next candy in the second box ∙ and so on until there is no candy left. So you put the

candies in the boxes in the following order: Find the index of the box where you put the K-th candy.

**Algorithms Used**

* Mathematical Simulation / Modular Arithmetic

Instead of simulating every candy placement, the code uses arithmetic operations (/,

%) to directly compute the final box for the K-th candy.

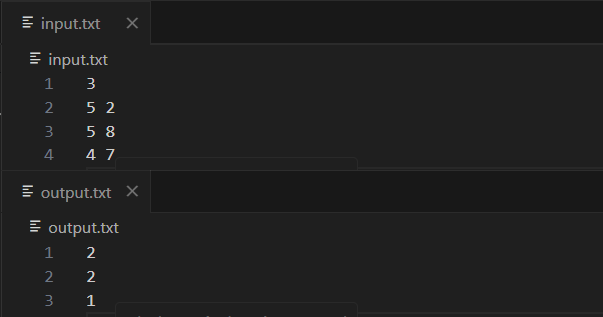
**Related Theory**

* This is a variation of zig-zag traversal or bidirectional simulation.
* Key observations:
* The first N candies fill boxes 1..N.
* After that, the filling alternates directions in blocks of (N-1) candies each.
* Therefore, the problem reduces to finding which “block” the K-th candy lies in and what its position within that block is.
* Time Complexity: O(1) per query since only arithmetic is used.
* Space Complexity: O(1) since no extra memory is needed.

**Algorithm / Procedure**

* Input n (number of boxes), k (candy number).
* If k ≤ n, return k directly (since each of the first N candies goes to a unique box).
* Otherwise:
* Subtract the first n candies (k -= n).
* Compute the block number: (k / (n-1)).
* Toggle direction depending on whether block number is even (forward) or odd (backward).
* Use modulo (k % (n-1)) to get exact position in the current block.
* Return the correct box index accordingly.
* Print the result.

**Output**

****

Implement and Explain Tower of Hanoi algorithm.

**Algorithms Used**

* Recursive Algorithm

**Related Theory**

* The Tower of Hanoi is a mathematical puzzle invented by Édouard Lucas in 1883.
* Rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the top disk from one rod and placing it on another rod.
3. A disk may not be placed on top of a smaller disk.

* Minimum number of moves:

M(n)=2n−1M(n) = 2^n - 1M(n)=2n−1

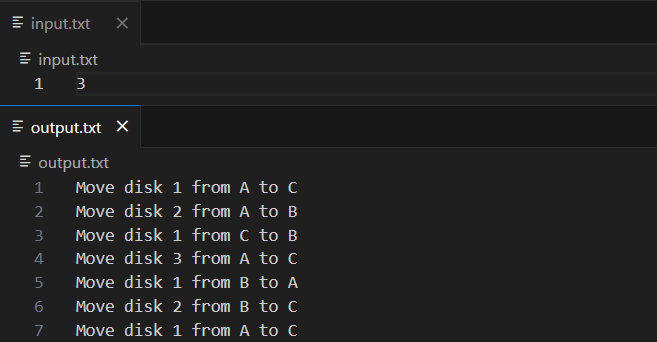
where n is the number of disks.

* Time Complexity: O(2^n) (since every disk generates two recursive calls).
* Space Complexity: O(n) (recursion depth).

**Algorithm / Procedure**

* Input number of disks n.
* If n == 1:
* Move disk 1 directly from source A to destination C.
* Otherwise:
* Recursively move n-1 disks from source A to helper B (using destination C).
* Move the largest disk n from source A to destination C.
* Recursively move n-1 disks from helper B to destination C (using source A).
* Continue until all disks are moved from source to destination.

**Output**

****

**Problem Statement 3:**

There is a frog initially placed at the origin of the coordinate plane. In exactly 1 second, the frog can either:

* move right by 1 unit,
* move up by 1 unit, or
* stay at the same position.

In other words, from position (x, y), the frog can move to:

* (x + 1, y)
* (x, y + 1)
* (x, y)

After T seconds, a villager reports that the frog lies on or inside a square of side length s with bottom-left corner at (X, Y) and top-right corner at (X + s, Y + s).

You are required to calculate how many integer coordinate points on or inside this square could be the frog’s position after exactly T seconds.

**Algorithms Used**

* Recursive Backtracking / DFS Simulation

**Related Theory**

* Time Complexity (recursive code): O(3^T) (explores all paths).

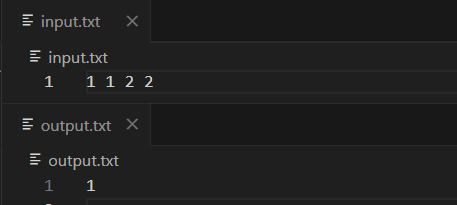
**Algorithm / Procedure**

* Read integers X, Y, s, T.
* Define square boundaries:
* x1 = X, x2 = X+s
* y1 = Y, y2 = Y+s
* Use recursion solver(x,y,t):
* If t == 0, insert (x,y) into set.
* Otherwise, recursively try (x+1,y), (x,y+1), (x,y) with t-1.
* After recursion, iterate over all collected positions.
* For each (tx,ty), check if it lies inside square: x1≤tx≤x2 and y1≤ty≤y2

If yes, increase counter.

* Print final count.

**Output**

****

**Problem Statement 4:**

Lost Package Tracker Problem Statement: A logistics company stores the scanned timestamps (in hours) of packages entering a warehouse in an array timestamps[]. Sometimes, a timestamp is repeated due to re-scanning. A package is considered "lost" if its ID (timestamp) is missing between two valid timestamps. Task: Given a sorted but incomplete list of timestamps from start to end, find the first missing timestamp in the range. Input: timestamps = [1001, 1002, 1004, 1005] Output: 1003

**Algorithms Used**

* Linear Scan Approach

**Related Theory**

* Time Complexity: O(n + m) where n = size of array, m = number of missing timestamps.
* Space Complexity: O(m) for storing **the missing numbers. Algorithm / Procedure**
* Input the number of timestamps n and the sorted array v[].
* Initialize:
* curr = v[0] (expected timestamp).
* i = 0 (index).
* While i < n:
* If curr == v[i]:
  + Increment both curr and i.
* Else:
  + Add curr to the result list (ans).
  + Increment curr (to check next possible missing number).
* After loop ends, ans contains all missing timestamps.
* Print the result.

**Output**

****

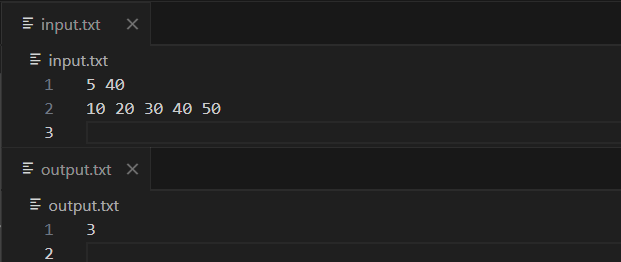
**Problem Statement 5:**

Implement linear search algorithm.

**Algorithms Used**

* Linear Search Algorithm

**Related Theory**

* Linear Search is the simplest searching technique.
* Works for both sorted and unsorted arrays.
* It does not require preprocessing (like binary search which requires sorted data).
* Time Complexity:
* Best case: O(1) (element is the first).
* Worst case: O(n) (element not present or at the end).
* Space Complexity: O(1) (only uses a few variables**). Algorithm / Procedure**
* input size of array n and target value key.
* Traverse the array from index 0 to n-1.
* For each element v[i]:
* If v[i] == key, return index i.
* If traversal finishes without finding key, return -1.
* Print the result

**Problem Statement 6:**

Implement Binary Search algorithm.

**Algorithms Used**

* Binary Search Algorithm

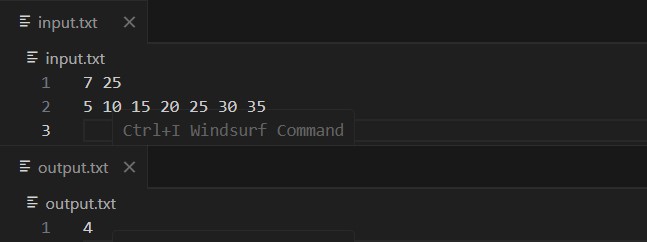
**Related Theory**

* **Time Complexity:**
* **Best case: O(1) (if middle element is the target).**
* **Worst/Average case: O(log n) (array halved each step).**
* **Space Complexity: O(1) (iterative version) or O(log n) (recursive version due to call stack).**

**Algorithm / Procedure**

* **Input array size n and target key.**
* **Initialize low = 0, high = n - 1.**
* **While low ≤ high:**
* **Compute mid = (low + high) / 2.**
* **If v[mid] == key, return mid.**
* **Else if v[mid] > key, update high = mid - 1.**
* **Else, update low = mid + 1.**
* **If not found, return -1.**
* **Print the result.**

**Output**

****

**Problem Statement 7:**

Signal Drop Detector Problem Statement: You're monitoring signal strengths over time using an array signal[]. A drop is defined as a strictly decreasing subsequence for at least 3 consecutive readings. Task: Find the number of such "signal drops" in the array. Input: signal

= [5, 4, 3, 6, 7, 4, 3, 2] Output: 2 (drops: 5→4→3 and 7→4→3→2)

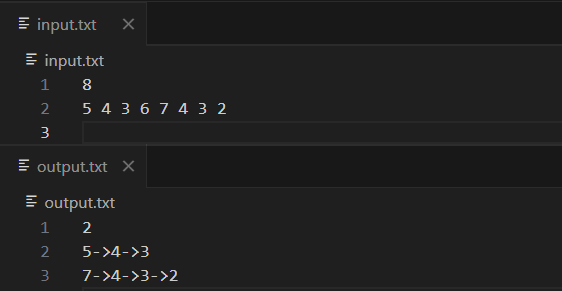
**Algorithms Used**

* Greedy Scanning / Subsequence Detection

**Related Theory**

* Greedy scanning ensures each drop is detected once.
* Time Complexity: O(n) (each element is processed once in the outer loop, inner loop advances index).
* Space Complexity: O(m \* k) where m = number of drops and k = avg length of each subsequence stored

**Algorithm / Procedure**

* Input size n and array signal[].
* Start scanning from i = 0.
* For each position:
* Initialize an empty subsequence temp.
* While elements are strictly decreasing (v[j] > v[j+1]):
  + Add elements to temp.
* If temp.size() ≥ 3, push it into result list.
* Move index i to the end of this subsequence.
* Otherwise, just increment i.
* Return the list of drops.
* Print count and each detected **subsequence. Output**