Subham Adhikari

B.Sc. (Hons.) Computing, Softwarica College of IT and E-commerce, Coventry University

ST5008CEM: Programming for Developers

Shrawan Thakur

March 02, 2025

**1. a) Ans:**

The main goal is to find the minimum number of measurement or attempts to find the critical temperature for the identical materials to change their properties.

**Approach**

To solve this, we use a dynamic programming (dp) approach. We have to determine the optimal plan to minimize the number of measurements/attempts needed to find the critical temperature. We consider the following given condition:

* If the material changes properties at a given temperature, it cannot be used for the further measurement and we cannot go past that temperature level.
* If the material does not change properties, the material can be reused for further tests.

Let **dp[k][n]** is the maximum number of temperatures that can be checked with given **n** temperature levels and **k** identical sample materials to find the critical temperature **f**.

**Algorithm and Steps**

* Initialize a 2D array **dp[k][n]** where **k** represents number of sample materials and **n** represents the number of temperature levels to find the critical temperature **f**.
* Enter a loop until **k** materials is iterated. For each material **k**, iterate for each temperature levels **n**. Make sure to start the iteration from 1, since material and levels cannot be 0.
* Create a **DP** table for the easy understanding. The row represents **n** temperature level and column represents **k** sample materials.
* Inside the nested loop, for each combination of **i** (samples) and **j** (temperature levels) check whether **i** > 1 and **j** > 1. If true then enter a loop which tracks a **cj** current row and **pj** previous row. And if **i** = **j** = 1 then, assign **dp[i][j]** = **j** or **dp[i][j]** = 1 respectively.
* For each possible temperature levels **x** (assumed) from 1 to **j**, calculate the cases:
* If material does not change its property, then use same number of materials and test temperature levels below. Then the number of measurements becomes **value1** = **dp[i][cj]** or **value1** = **dp[i][j-x]**.
* If material changes its property, then use one less material and test temperature levels below. Then the number of measurements becomes **value2** = **dp[i-1][pj]** or **value2** = **dp[i-1] [x-1]**.
* Take the maximum measurements **val** of these two cases to handle the worst-case.
* Find the minimum number of measurements **min** (minimum of maximum measurements **val**) across all possible temperature levels to find the optimal measurement or attempts.
* Populate the table, for each **i** and **j**, set **dp[i][j]** = **min** + 1, where **min** is the minimum worst-case attempts found and extra addition of 1 represents the attempts taken before hand.
* After filling the DP table, the final result is stored in **dp[k][n]**, which gives the minimum measurements required to find the **f** critical temperature with **k** sample materials and **n** temperature levels.

**Dynamic Programming (DP) Table**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DP** | **n** temperature levels | | | | | | | | | | | |
| **k** sample materials |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **2** | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 4 |
| **3** | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 |
| **4** | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 |
| **5** | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 |

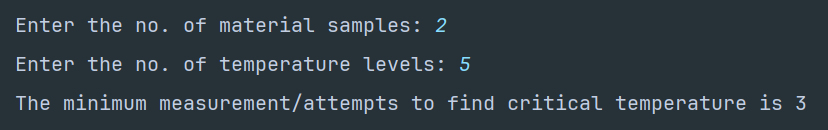
**Code Implementation**



**Example-1**

Input: k = 2, n = 5

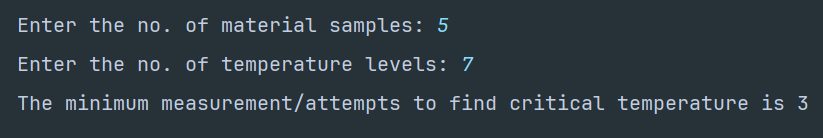
Output (no of attempts): 3



**Example-2**

Input: k = 5, n = 7

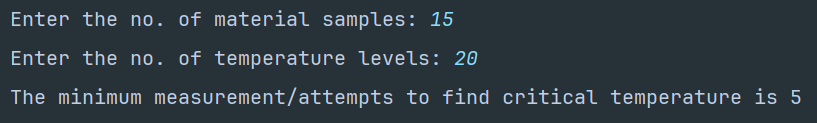
Output (no of attempts): 3



**Example-3**

Input: k = 15, n = 20

Output (no of attempts): 5



**1. b) Ans:**

The goal is to find the **k**th lowest combined return where we select one element from each of the two sorted arrays and calculating their product.

There are two sorted arrays: **returns1** and **returns2** given to calculate kth smallest combined return from both arrays.

**Approach**

To solve this problem, we can use a Binary Search Tree (BST). The main approach is to

* Insert all the possible combinations into the BST.
* Traverse through nodes in sorted order from smallest product to largest.
* The **k**th node visited during the traversal will give the **k**th lowest combined return.

**Algorithm and Steps**

* Initialize a Binary Search Tree (BST) class **BST** and node class **BSTNode** where **i**, **j** represents indices of **returns1** and **returns2** respectively and **product** represents the product between the indices of **returns1** and **returns2** i.e., **product = returns1[i] \* returns2[j]** and **left**, **right** represent child nodes – left and right of BST.
* In **BST** class, define methods **insert()** and **insertRec()** which adds a new node into the BST based on product value. The smaller products go to the left side and larger products go to the right side of the tree.
* Similarly in **BST** class, define another method **kthSmallest(int k)** which performs an in-order traversal by visiting every nodes in increasing order. It uses a counter **count** to track the number of nodes visited. When **count** equals **k**, it records the product of current node.
* In main class, define a method **findKthSmallestProduct()** where it uses nested loops to iterate through all indices **i** from **returns1** and **j** from **returns2**. Insert all possible product combinations into the BST.
* For each pair, compute **product = returns1[i] \* returns2[j]** and insert a new **BSTNode(i, j, product)** into the BST. If **product** is smaller, insert into the left subtree and if **product** is larger, insert into the right subtree.
* After building the BST with all possible combinations, an in-order traversal is performed to get the **k**th smallest product return. In-order traversal approach:
* Traverse the left subtree.
* Visit the current node and increment a count.
* If count equals **k**, store the product of node.
* Traverse the right subtree.
* The **k**th visited node in in-order traversal holds the **k**th smallest product.
* The product at the **k**th lowest product return which is based on sorted order is returned as the output.

**Time and Space Complexity Analysis**

1. Time Complexity:

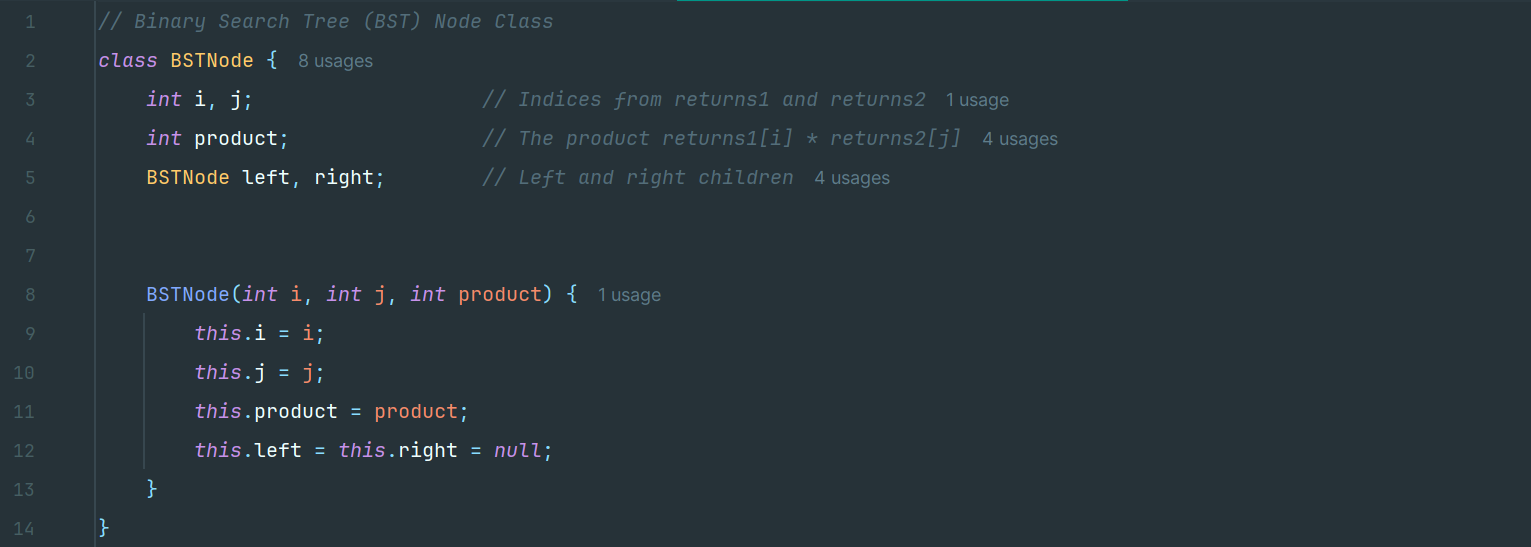
* To build BST:
* There is **m \* n** possible products where **m = returns1.length**, **n = returns2.length**.
* Each insertion into the BST takes **O(log(m \* n))** in the worst case.
* Total BST insertions take **O(m \* n log(m \* n))**.
* To find the kth smallest product, in-order traversal takes O(k) time to reach the kth node.

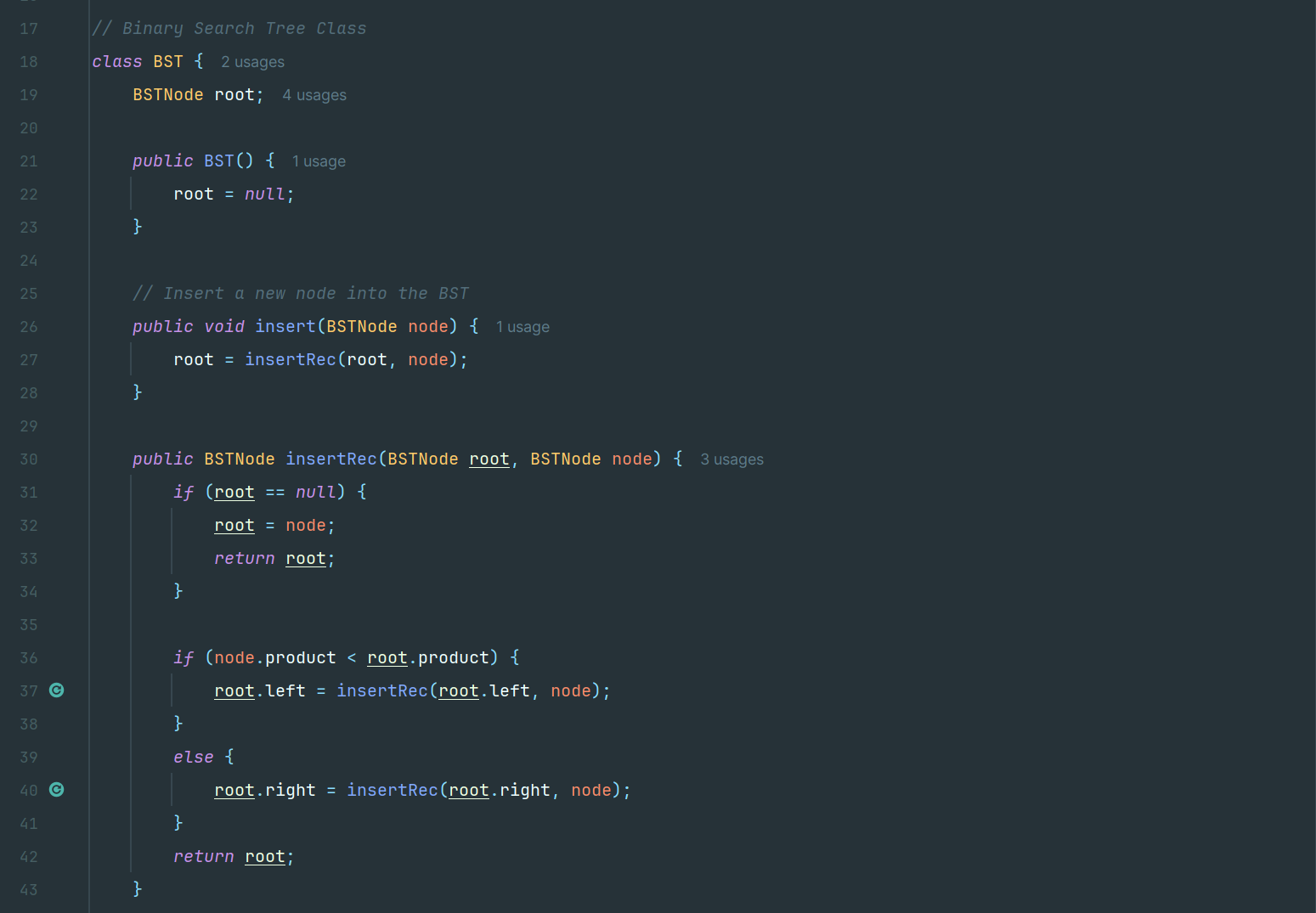
Total Time Complexity in the worst case: **O(m \* n log(m \* n) + k) ≈ O(m \* n log(m \* n))**.

1. Space Complexity:

* To store BST in the memory, it takes up to **O(m \* n)** nodes in all product pairs.
* The recursive stack for in-order traversal requires **O(m \* n)** space in worst case.

**Code Implementation**



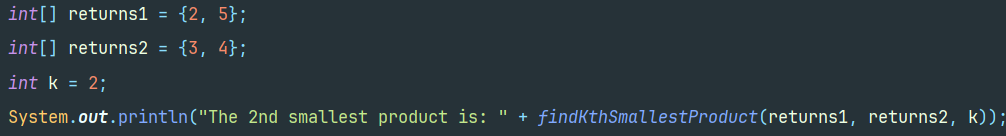






**Example-1**

Input: returns1 = {2, 5}, returns2 = {3, 4}, k = 2

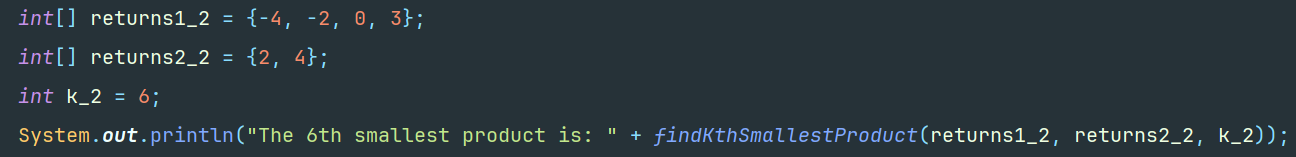


Output:8



**Example-2**

Input: returns1 = {-4, -2, 0, 3}. returns2 = {2, 4}, k = 6

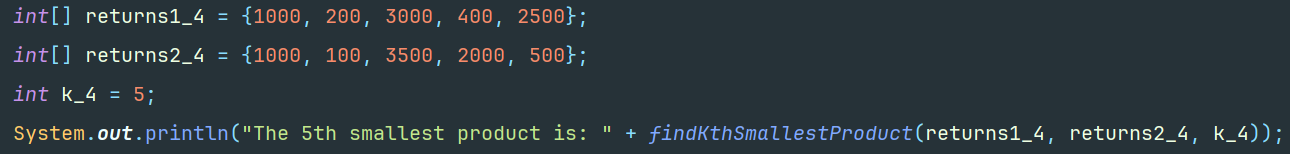


Output: 0



**Example-3**

Input: returns1= {1000, 200, 3000, 400, 2500}, returns2= {1000, 100, 3500, 2000, 500}, k = 5



Output: 200000



**2. a) Ans:**

The goal is to assign the minimum number of rewards to a team of **n** employees where each employee is given at least one reward and the employees with a high-performance rating more rewards than other employees.

There is given an integer array rating: **ratings**

**Approach**

In this, the two-pass greedy approach is used. We assume an array **ratings** such that the rating of each employee is stored.

* Left-to-Right Pass:

If the rating of an employee is higher than the previous employee, they receive one more reward than the previous employee.

* Right-to-Left Pass:

If the rating of an employee is higher than the next employee, they receive at least one more reward than the next employee.

The two-pass greedy method ensures that both left and right conditions are satisfied.

**Algorithm and Steps**

* Define a method **calculateMinRewards()** which takes a integer array **ratings** as parameter. First the validation of array **ratings** is checked i.e.,
* If **ratings** null or empty, return 0.
* If the length of **ratings** is equal to, return 1.
* Inside this method, initialize an integer array **rewards** of length **n** which represent rewards received by employees and number of employees respectively.
* Start a left to right pass iteration which iterates through each employee from **1** to **n-1** and checks whether **ratings[i] > ratings[i - 1]** then sets **rewards[i] = rewards[i - 1] + 1**. This ensures that any employee with a higher rating than the previous one receives one more reward.
* Similarly, Start a right to less pass iteration which iterates through each employee from **n-2** down to **0** and checks whether **ratings[i] > ratings[i + 1]** then sets **rewards[i] = Math.max(rewards[i], rewards[i + 1] + 1)**. This ensures that employee with a higher rating than the next one receives at least one more reward than that neighbor.
* Then, sum up all values in the rewards array to find the total minimum rewards needed. Finally, the output is returned.

**Time and Space Complexity Analysis**

1. Time Complexity:

* The iteration through array takes **O(n)** in worst case where **n** is the number of employees.

1. Space Complexity:

* The space used by an array of size **n** to store rewards is **O(n)** in worst case.

**Code Implementation**

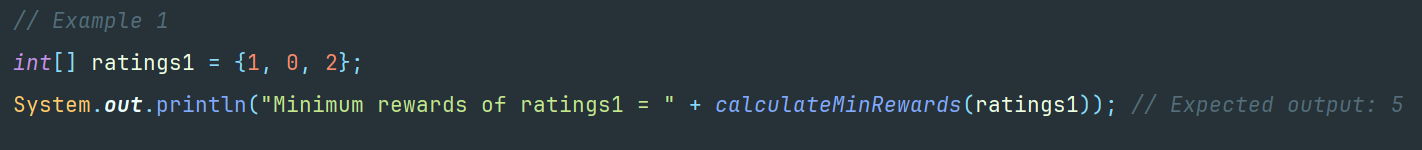






**Example-1**

Input: ratings = {1, 0, 2}

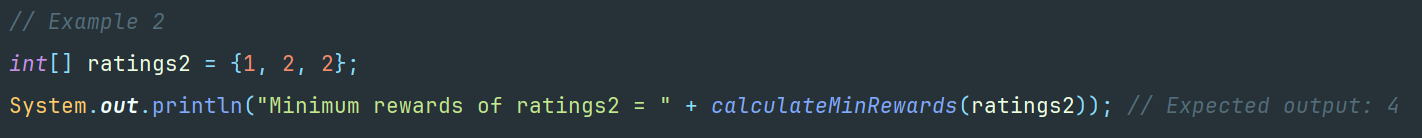


Output: 5



**Example-2**

Input: ratings = {1, 2, 2}

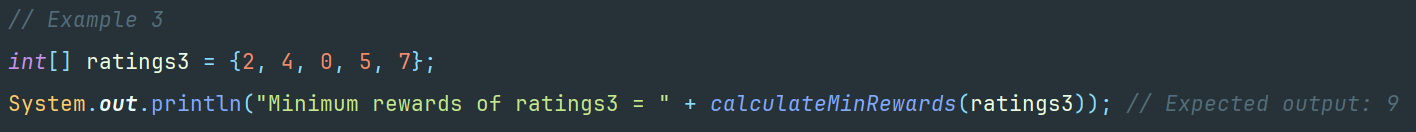


Output: 4



**Example-2**

Input: ratings = {2, 4, 0, 5, 7}



Output: 9



**2. b) Ans:**

The main goal is to find out the lexicographically close pair of points with the smallest distance where smallest distance is calculated using:

**| x\_coords [i] - x\_coords [j]| + | y\_coords [i] - y\_coords [j]|**

where **x\_coords** and **y\_coords** represent array of x-coordinates and y-coordinates of the points respectively.

**Approach**

In this problem, we apply Brute-force because this approach directly computes distances and is optimal for small to moderate points. It iterates through all pairs **(i, j)** with **i < j**, compute their distance and track the pair with the smallest distance. If multiple pairs have the same distance, then it selects the lexicographically smallest one.

**Algorithm and Steps**

* Declare a method **findClosestPair()** inside initialize variables **minDistance** to a large value **Integer.MAX\_VALUE**, **bestI** and **bestJ** to store the indices of the best (closest) pair.
* Start a nested iteration that iterates over all the pairs **(i, j)** with **i < j**. The outer loop runs from **i=0** to **n-1** and inner loop from **j=i+1** to **n-1**.
* For each iteration in the nested loop, the distance is calculated i.e.

**distance = Math.abs(x\_coords[i] - x\_coords[j]) + Math.abs(y\_coords[i] - y\_coords[j])**

* Similarly, check the conditions in each iteration:
* If **distance < minDistance**, then update **minDistance=distance** and set **bestI = i**, **bestJ = j**.
* If **distance == minDistance**, check lexicographical order i.e., if **i<bestI** or **i==bestI** and **j<bestJ**, update **bestI=i** and **bestJ=j**.
* After checking all pairs, return the array pair **[bestI, bestJ]** as the result.

**Time and Space Complexity**

1. Time Complexity:

* The nested loops to compare every pair through two arrays co-ordinates takes **O(n2)** in worst case where **n** is the length of array co-ordinates.

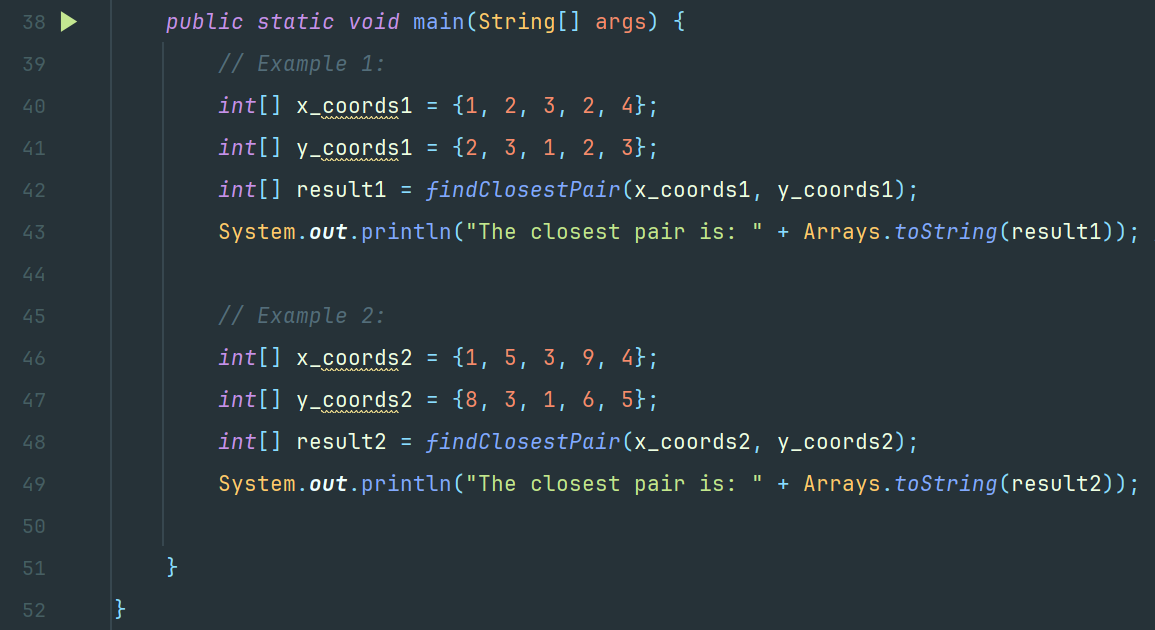
1. Space Complexity:

* The space used is **O(1)** in worst case.

**Code Implementation**







**Example-1**

Input: x\_coords = {1, 2, 3, 2, 4}, y\_coords= {2, 3, 1, 2, 3}



Output: result = [1, 3]



**Example-2**

Input: x\_coords = {1, 5, 3, 9, 4}, y\_coords= {8, 3, 1, 6, 5}



Output: result = [1, 4]



**3. a) Ans:**

The goal is to find the minimum total cost to connect **n** devices in the network where each device can either install a communication module at a given cost which is specified by the **modules** array or be connected using a direct connection which is specified in **connections** where **connections[j] = [device1j, device2j, costj]** which represents the cost to connect devices **device1j and device2j**.

**Approach**

For this problem, we assume a Minimum Spanning Tree (MST) problem on an augmented graph.

* Add a special virtual node (node 0) which represents module installation.
* For each device **i**, add an edge from the virtual node to device **i** with weight **modules[i]**.
* Add the given bidirectional connections as edges between devices.

We use Kruskal’s algorithm to build an MST that connects all nodes (the virtual node and all devices) at the minimum total cost and Disjoint Set Union (DSU) structure the algorithm.

**Algorithm and Steps**

* Define classes **Edge** represents an edge in the graph with endpoints u, v and a connection cost and **DSU** to perform efficient union-find operations in Kruskal's algorithm.
* Declare a method in main class **minCostToConnect()** where the minimum total cost is calculated to connect all devices using an MST approach.
* Start a loop such that for each device **i** from 1 to **n**, add an edge between the virtual node (node 0) and device **i**. The cost of this edge is the module installation cost given in **modules[i-1]**.
* Similarly, for each connection provided (an array like [u, v, cost]), add an edge between device **u** and device **v** with the given cost. The connections should be bidirectional.
* Sort all edges in ascending order of cost to ensure we pick the lowest cost edges first.
* Initialize a **DSU** to manage which devices are connected.
* Start an iteration through sorted edges where for each **DSU** edge, check whether the two nodes are in different sets. If they are not connected i.e., adding the edge does not create a cycle, then add the cost of edge to the total cost and merge the two sets. Stop when **n** edges are added where edges connect **n+1** nodes.
* Return the **totalCost** as minimum total cost required all the devices.

**Time and Space Complexity**

1. Time Complexity:

* The nested loops to compare every pair through two arrays co-ordinates takes **O(n2)** in worst case where **n** is the length of array co-ordinates.
* To build graph, it took **O(v + e)** where **v=n+1** and **e** is the total number of edges (the virtual node and the provided connections).
* To sort edges, it took **O(e log e)**.
* The Kruskal’s Algorithm took **O(e log e)**.

Overall time complexity is **O(e log e)** in worst where **e** is the number of edges.

1. Space Complexity:

* The space used to store graph which is edges list is **O(e)**.
* The space used by DSU is **O(v)**.

Overall space complexity is **O(v + e)** in worst case where v = n+1 and e is the number of edges.

**Code Implementation**

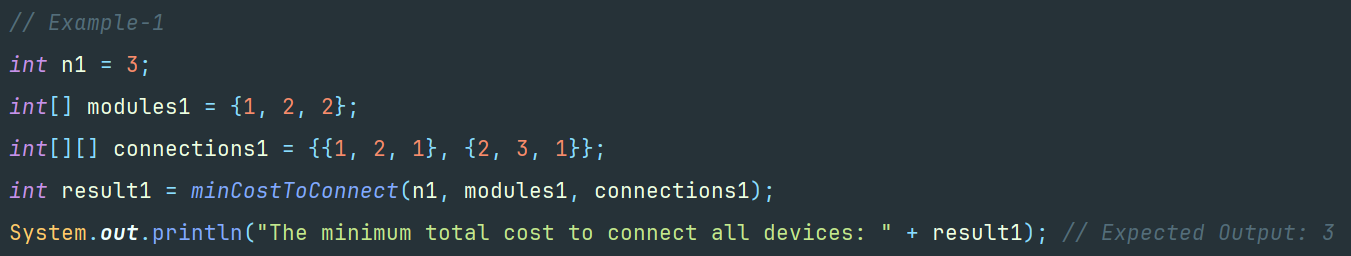
****

****

****

**Example-1**

Input: n = 3, modules = {1, 2, 3}, connections = {{1, 2, 1}, {2, 3, 1}}

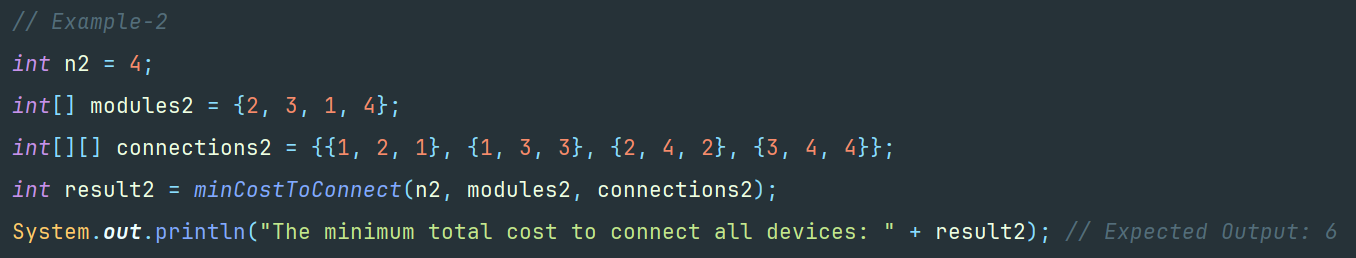


Output: 3



**Example-2**

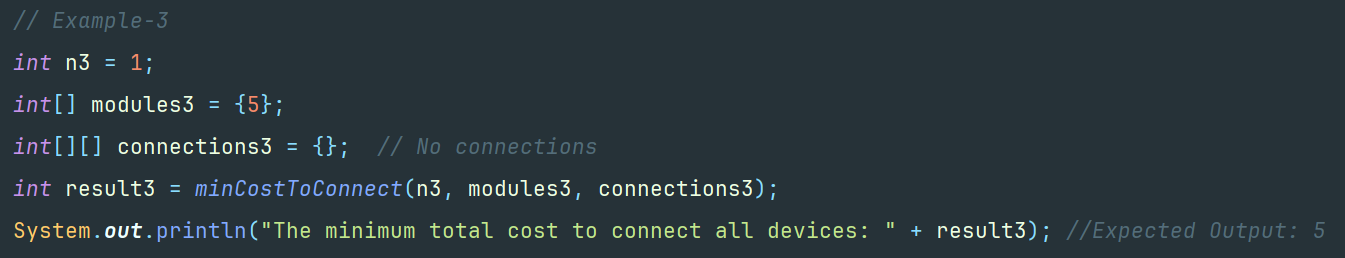
Input: n = 4, modules = {2, 3, 1, 4}, connections = {{1, 2, 1}, {1, 3, 3}, {2, 4, 2}, {3, 4, 4}}



Output: 6

**Example-3**

Input: n = 1, modules = {5}, connections = {}



Output: 5



**4. a) Ans:**

The main goal is to return the result table ordered by count of hashtag which is in descending order. An input data (in table) is passed with **userId**, **tweetId**, **tweetDate**, **tweetText** where **userId** and **tweetId** are primary keys which uniquely identify the user and tweet respectively.

**Approach**

To solve this, we use file handling to persist auto-incrementing primary keys i.e., userId and tweetId so that every new tweet gets a unique ID even when the program restarts. Each tweet is represented by a **Tweet** object that extracts hashtags from its text. The program then prints the tweets in a table and computes the top 3 trending hashtags for February 2024 by counting every hashtag occurrence. When two hashtags have the same count, they are sorted alphabetically in ascending order.

**Algorithm and Steps**

* Initialization a class **PrimaryKeyGenerator** which reads from a file ("ids.txt") that contains two lines in the format "userId: <value>" and "tweetId: <value>". If the file exists, it initializes the counters from it. Otherwise, it starts with 0.
* Start a loop where each time a new **Tweet** is created, the generator returns the next available userId and tweetId, increments the counters and updates the file.
* Initialize a class **Tweet**. When a **Tweet** object is created, it uses the **PrimaryKeyGenerator** to get unique IDs.
* Declare a **printInputTable()** method which formats and prints each tweet along with its details (IDs, text, hashtags, and date) in an table.
* Declare a method **findTopTrendingHashtags()** which performs:
* It filters the tweets to only those dated in February 2024.
* It then counts every occurrence of each hashtag with duplicates ones using a HashMap.
* The hashtags are sorted by count in descending order. if two hashtags have the same count, they are sorted alphabetically in ascending order.
* Hashtags with maximum count are returned.
* Finally, the top 3 trending hashtags and their counts are printed in another output table.

**Time and Space Complexity**

1. Time Complexity:

* For **PrimaryKeyGenerator**, reading and writing the file is constant time (**O(1)**) since the file is very small.
* For **Tweet**, extracting hashtags from a tweet takes O(m), where m is the length of the tweet.
* For **findTopTrendingHashtags**:
* Filtering and counting hashtags over n tweets takes **O(n \* m)** time in the worst case.
* Sorting the unique hashtags takes **O(k log k)**, where **k** is the number of unique hashtags.

Overall time complexity is **O(n \* m + k log k)** in worst case.

1. Space Complexity:

* The tweet objects and their extracted hashtags require **O(n \* m)** space.
* The counting HashMap uses **O(k)** space.

Overall space complexity is **O(n \* m + k)** in worst case.

**Code Implementation**







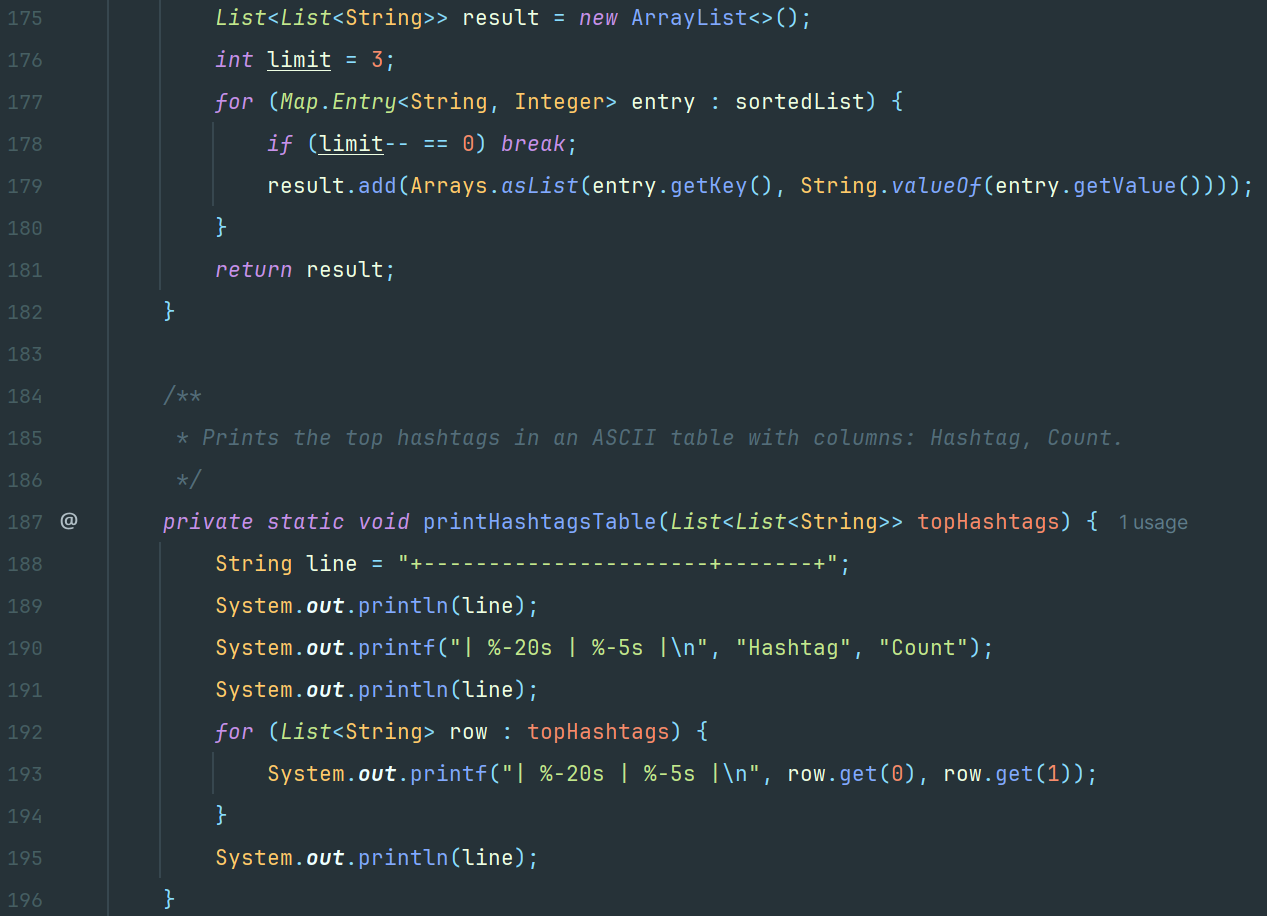




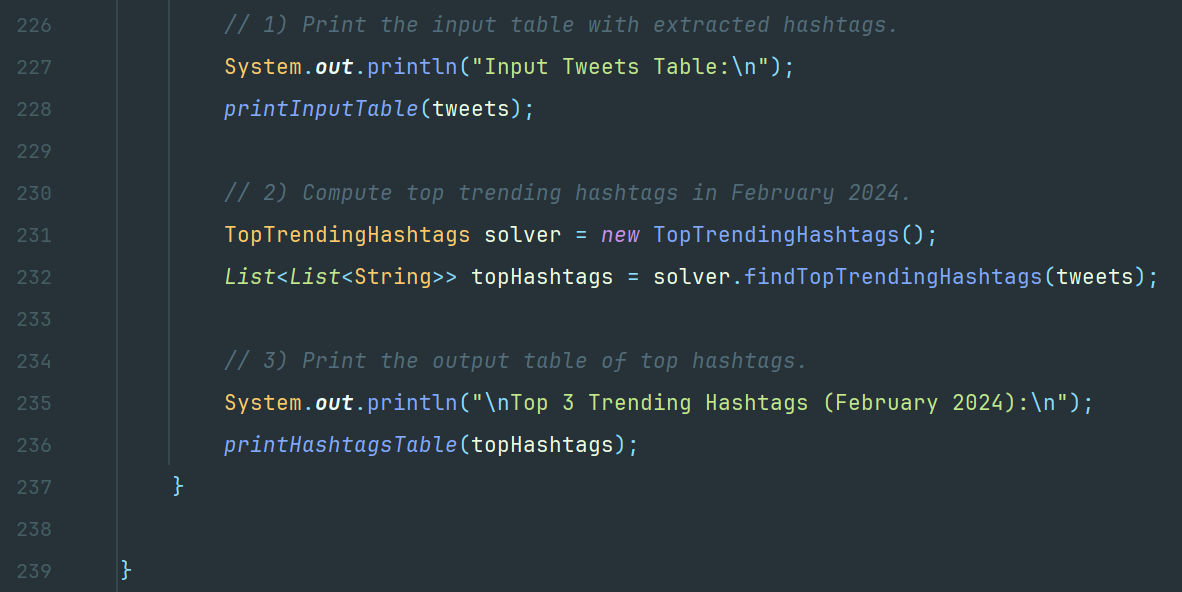






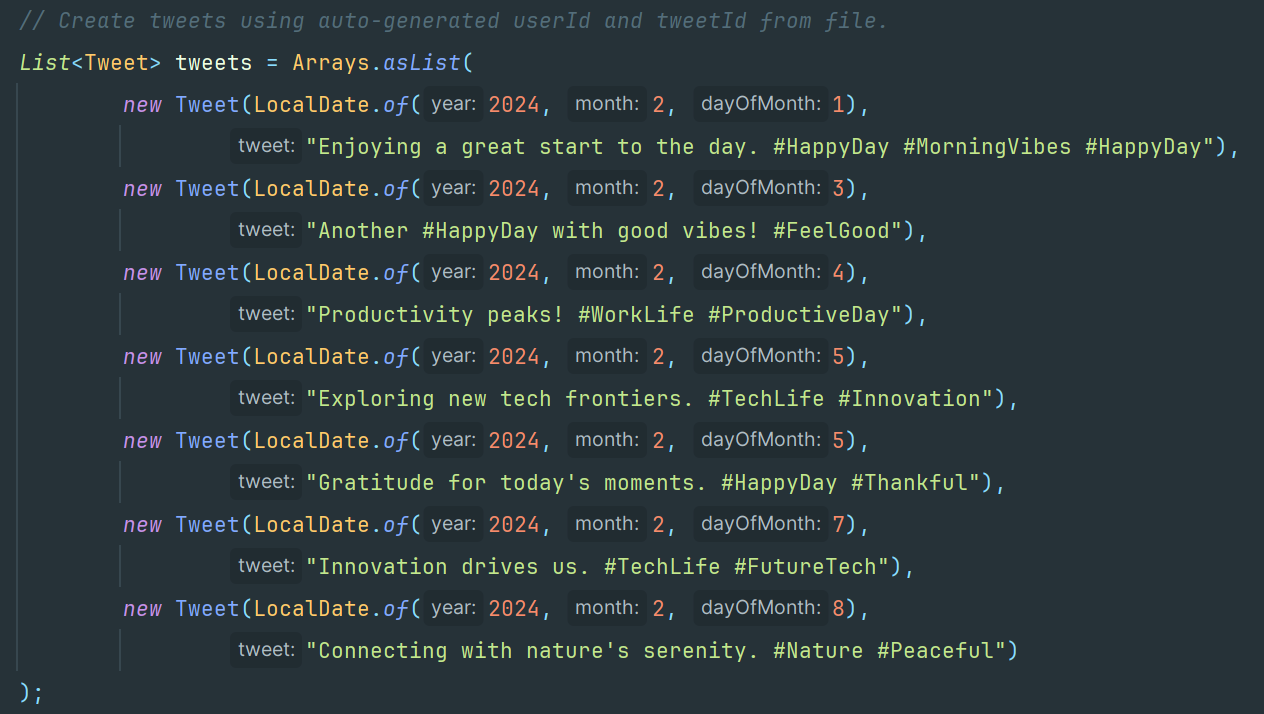


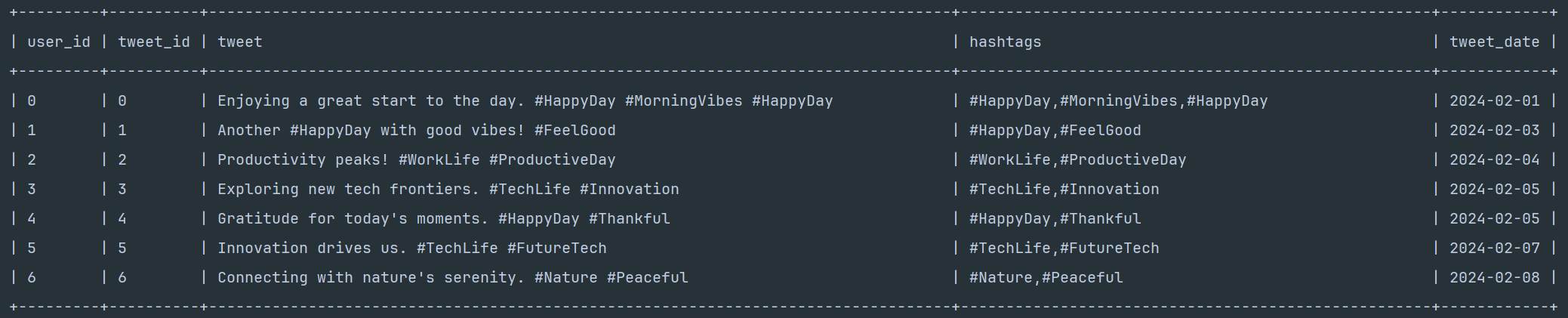




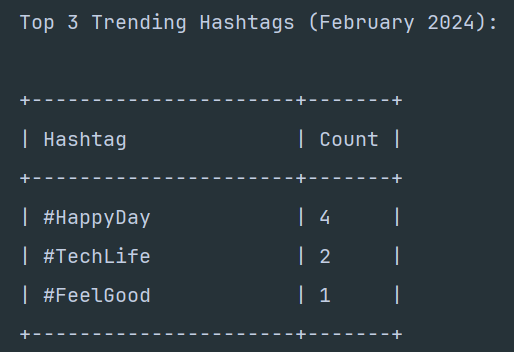
**Example-1**

Input:





Output:



**6. a) Ans:**

The goal is to sequence "0102030405..." up to number **n** using three threads **ZeroThread**, **EvenThread** and **OddThread**. A class **NumberPrinter** is given with three methods: **printZero()**, **printEven()** and **printOdd()** to print zeros, even numbers and odd numbers respectively in a correct order.

**Approach**

To solve this problem, we use a shared state with synchronization through **Lock** and **Condition**. The state transitions control which thread is allowed to execute next.

* **ZeroThread**: It prints "0" and then signals **OddThread** if the next number is odd or signals **EvenThread** if even.
* **OddThread** and **EvenThread**: Wait for the turn, print the corresponding number, increment the counter and signal ZeroThread.

**Algorithm and Steps**

* Initialize variables **current = 1** and **state = 0**. Also initialize a class **NumberPrinter** and a controller with the maximum number **n** and a **NumberPrinter** instance as parameters.
* For **ZeroThread**, start a loop which iterates while **current <= n**.
* If **state != State.ZERO**, it waits on the condition.
* After the condition is met, **printer.printZero()** is called with prints “0”.
* Check if **current** is odd, it sets state to **State.ODD**; otherwise, it sets it to **State.EVEN**. Then, **condition.signalAll()** is called which signals all waiting threads so that the correct thread can proceed.
* For **OddThread**, start a loop which iterates while **current <= n**.
* If **state != State.ZERO** and **current** is odd, it waits on the condition.
* After the condition is met, **printer.printOdd(current)** is called with prints “0”.
* The **current** is incremented i.e., **current++**
* Set the state to **ZERO** i.e., **state = State.ZERO** and signals all threads so that the **ZeroThread** can print the next zero.
* Similarly, just like for **EvenThread**, same process is repeated.
* The threads keep on repeating until **current** exceeds **n**.
* Finally, any remaining threads are signaled to ensure proper termination and the output is printed.

**Time and Space Complexity**

1. Time Complexity:

* The loops through **n** takes **O(n).**
* Each thread operation (print and state update) is done in **O(1)**.

Overall time complexity is **O(n)** in worst where **n** is the **nth** of the number.

1. Space Complexity:

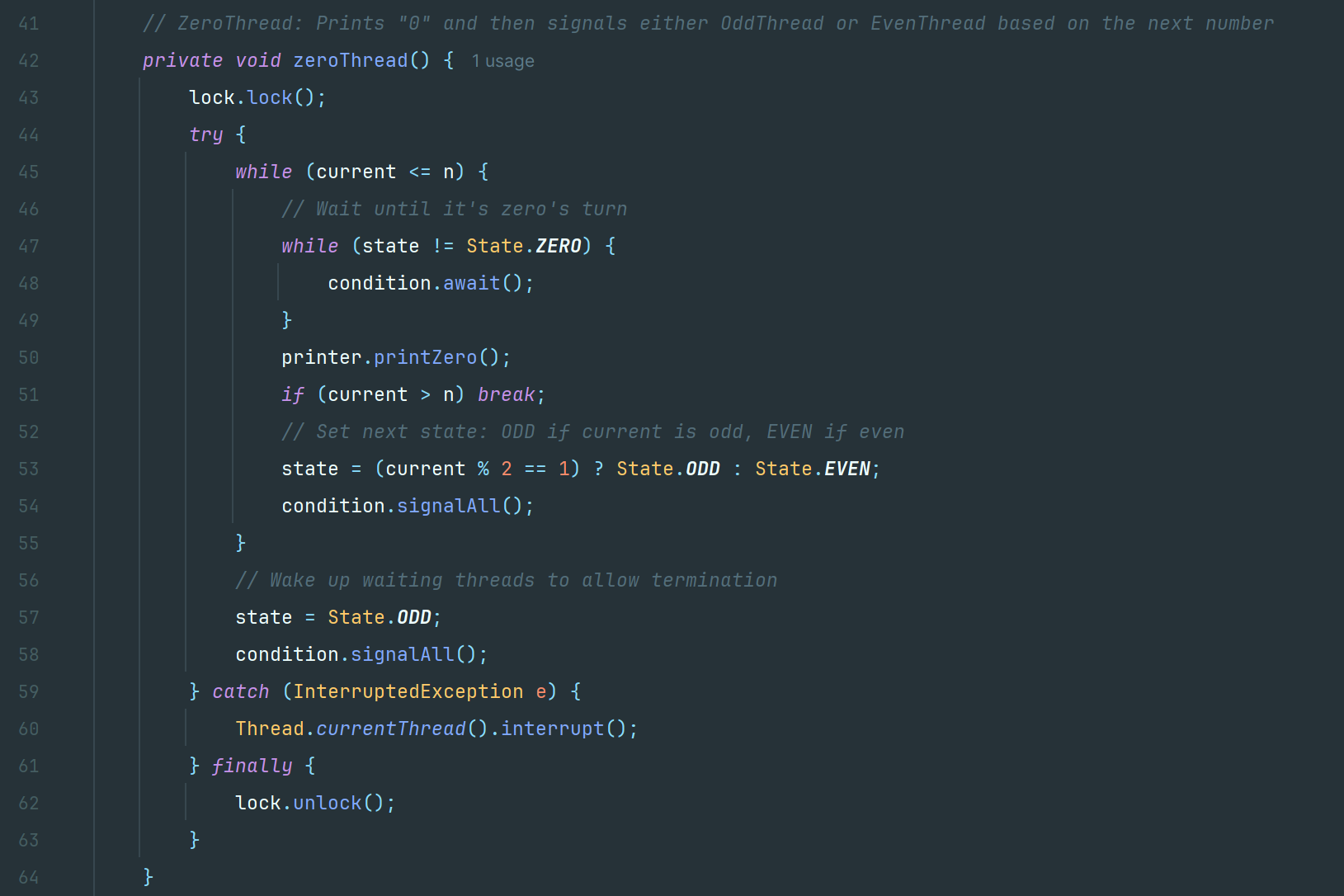
* A fixed number of variables are such as lock, condition, state, and counter which takes **O(1)**.

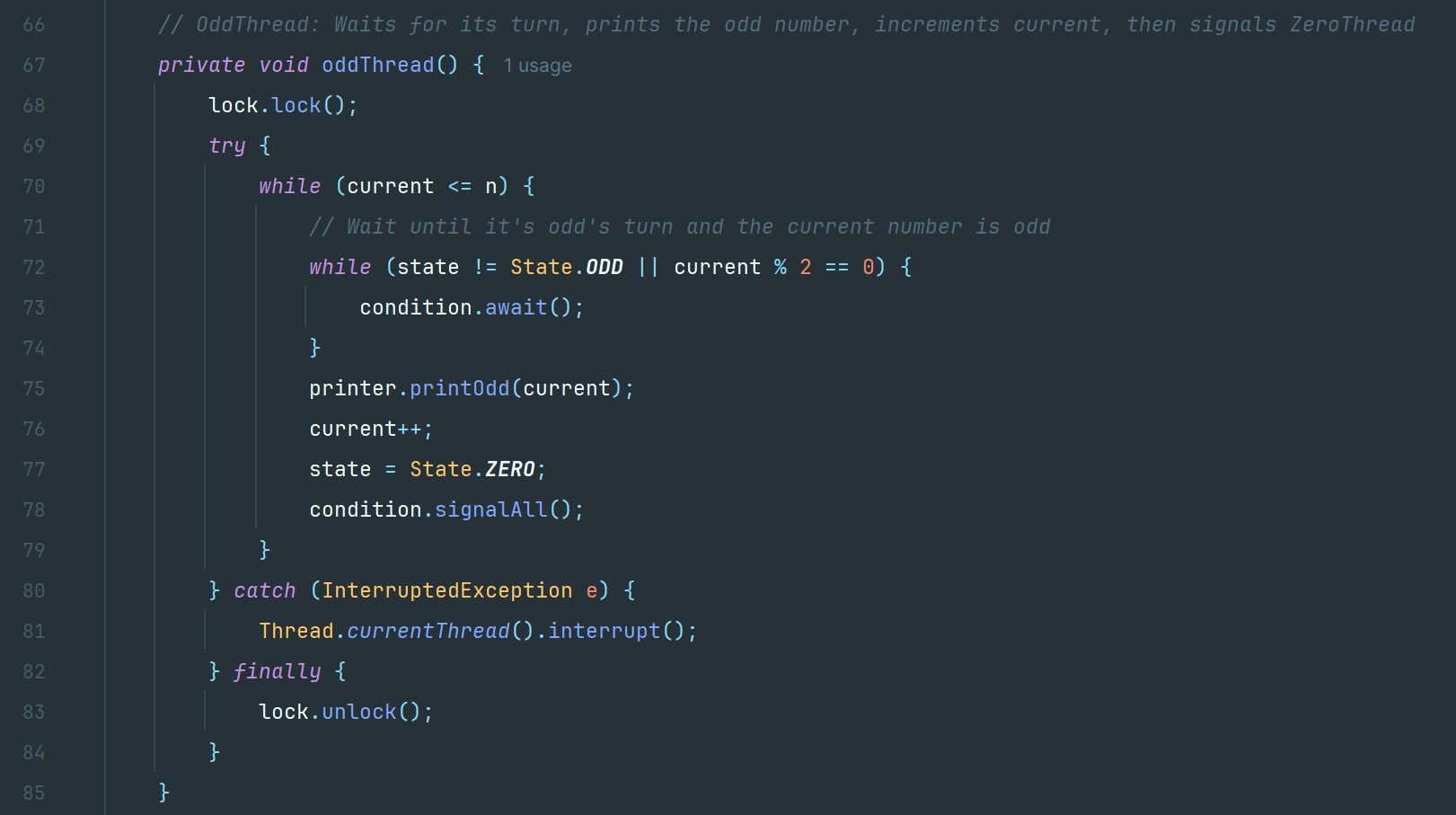
Overall space complexity is **O(1)** in worst case.

**Code Implementation**

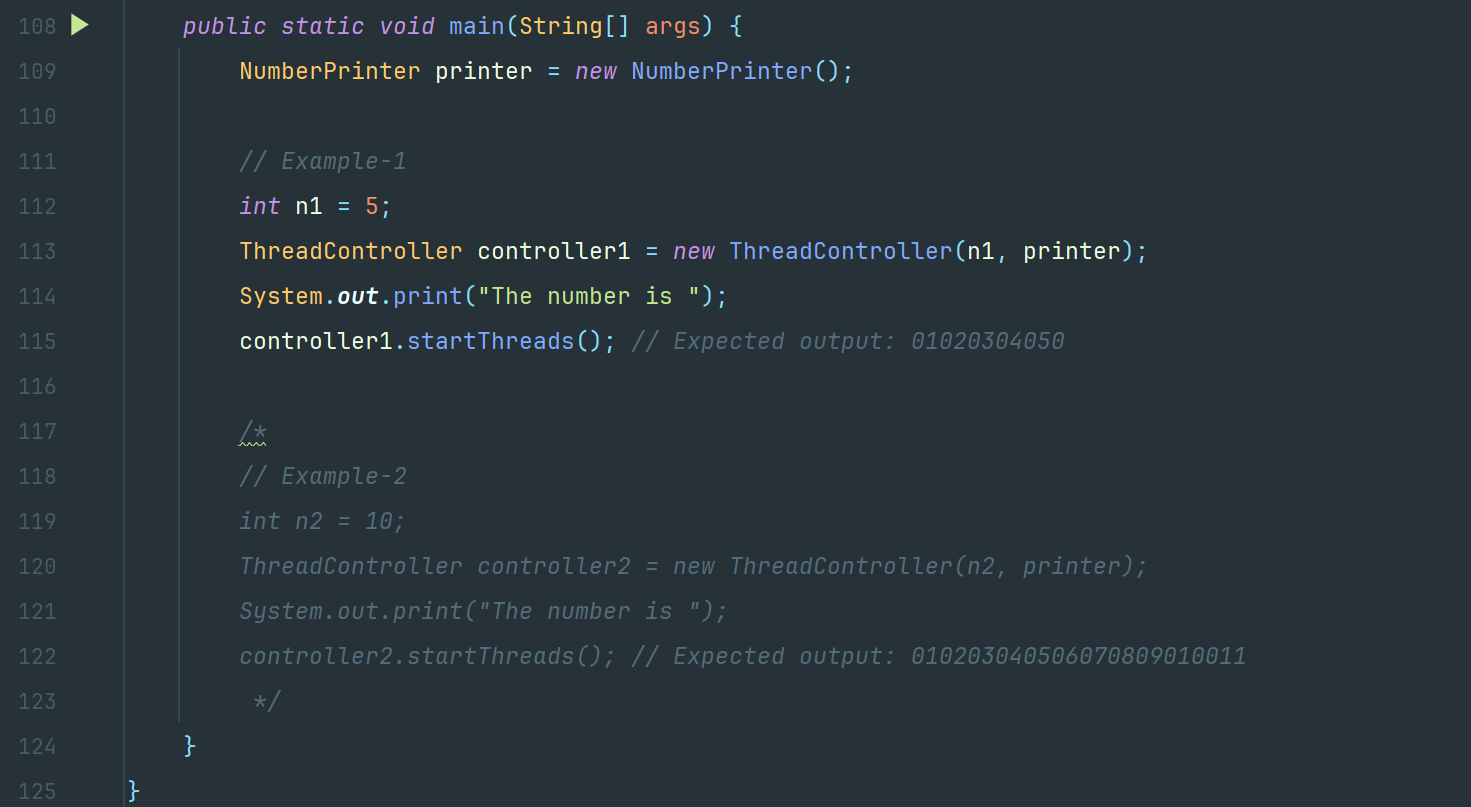






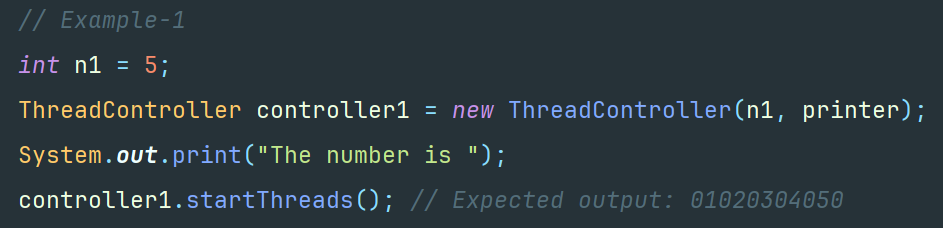






**Example-1**

Input: n = 5

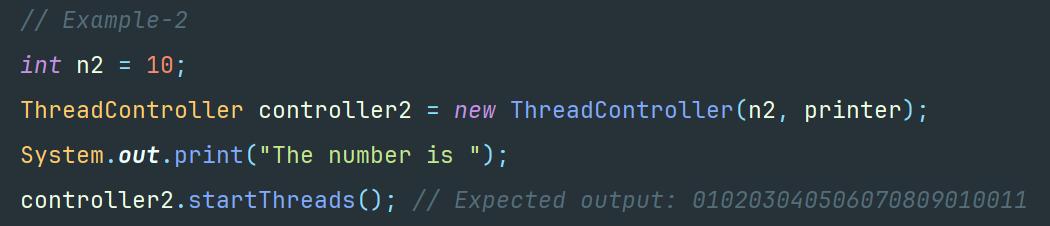


Output: 01020304050



**Example-2**

Input: n = 10



Output: 010203040506070809010011

