**30 DAYS CODING CHALLENGE**

**DAY 1:**

**SUPER REDUCED STRING:**

Reduce a string of lowercase characters in range ascii[‘a’..’z’]by doing a series of operations. In each operation, select a pair of adjacent letters that match, and delete them.

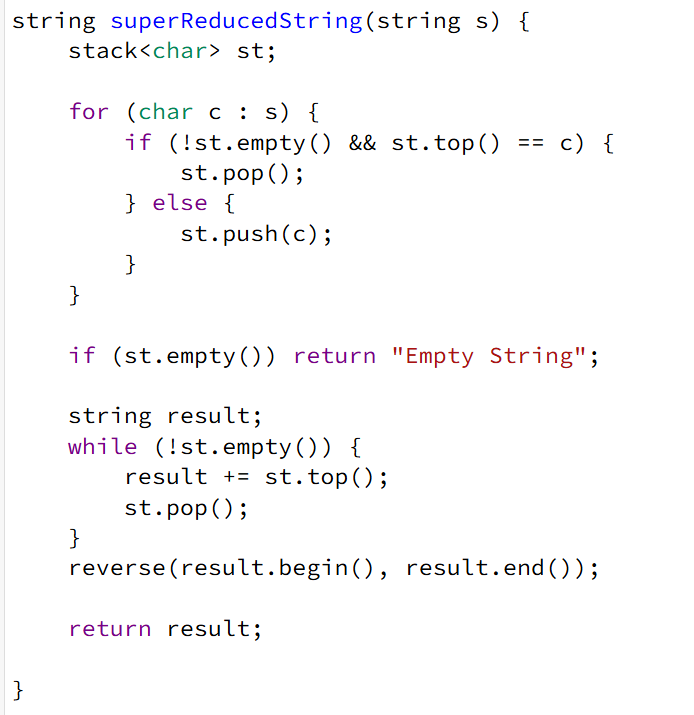
Delete as many characters as possible using this method and return the resulting string. If the final string is empty, return Empty String

**Example**.

s = aab

aab shortens to b in one operation: remove the adjacent a characters.

**SOURCE CODE:**

****

**DRY RUN:**

**Example Input:**

Lets consider the string s = "aaabccddd"

* Initialize an empty stack st.
* Iterate through the string s:
* **'a'** → Stack is empty → Push 'a' → Stack: ['a']
* **'a'** → Top of stack is 'a' (same) → Pop 'a' → Stack: []
* **'a'** → Stack is empty → Push 'a' → Stack: ['a']
* **'b'** → Top of stack is 'a' (different) → Push 'b' → Stack: ['a', 'b']
* **'c'** → Top of stack is 'b' (different) → Push 'c' → Stack: ['a', 'b', 'c']
* **'c'** → Top of stack is 'c' (same) → Pop 'c' → Stack: ['a', 'b']
* **'d'** → Top of stack is 'b' (different) → Push 'd' → Stack: ['a', 'b', 'd']
* **'d'** → Top of stack is 'd' (same) → Pop 'd' → Stack: ['a', 'b']
* **'d'** → Top of stack is 'b' (different) → Push 'd' → Stack: ['a', 'b', 'd']
* Stack after processing: ['a', 'b', 'd']
* Reverse the stack content to get the final string: "abd"

**OUTPUT:**  s=”abd”

**DAY 2:**

**GAME OF THRONE**

Dothraki are planning an attack to usurp King Robert's throne. King Robert learns of this conspiracy from Raven and plans to lock the single door through which the enemy can enter his kingdom.

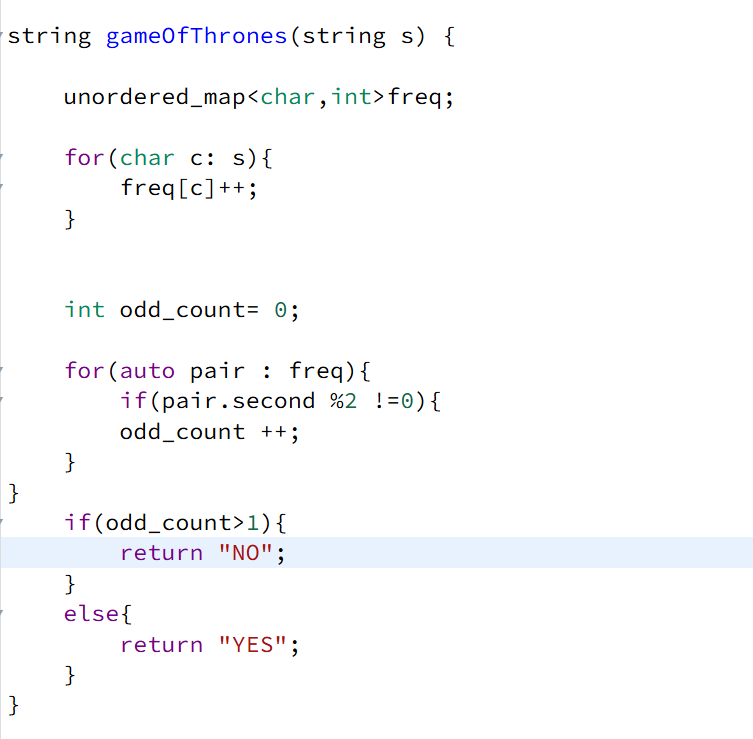
But, to lock the door he needs a key that is an [anagram](https://www.hackerrank.com/external_redirect?to=https://en.wikipedia.org/wiki/Anagram) of a [palindrome](https://www.hackerrank.com/external_redirect?to=http://en.wikipedia.org/wiki/Palindrome). He starts to go through his box of strings, checking to see if they can be rearranged into a palindrome. Given a string, determine if it can be rearranged into a palindrome. Return the string YES or NO.

**Example**

S = aabbccdd

One way this can be arranged into a palindrome is  abcddcba. Return YES.

**SOURCE CODE:**



**DRY RUN:**

**Example Input:**  s = "civic"

* **Initialize an unordered map freq** to store character frequencies.
* **Count the frequency of each character in s:**
* **'c'** → freq['c'] = 1
* **'i'** → freq['i'] = 1
* **'v'** → freq['v'] = 1
* **'i'** → freq['i'] = 2
* **'c'** → freq['c'] = 2
* **Final frequency map:**
* {'c': 2, 'i': 2, 'v': 1}
* **Count characters with odd frequencies:**
* 'c' = 2 (even) → No change in odd\_count
* 'i' = 2 (even) → No change in odd\_count
* 'v' = 1 (odd) → odd\_count = 1
* **Decision:**
* odd\_count = 1 (≤ 1) → Return **"YES"**

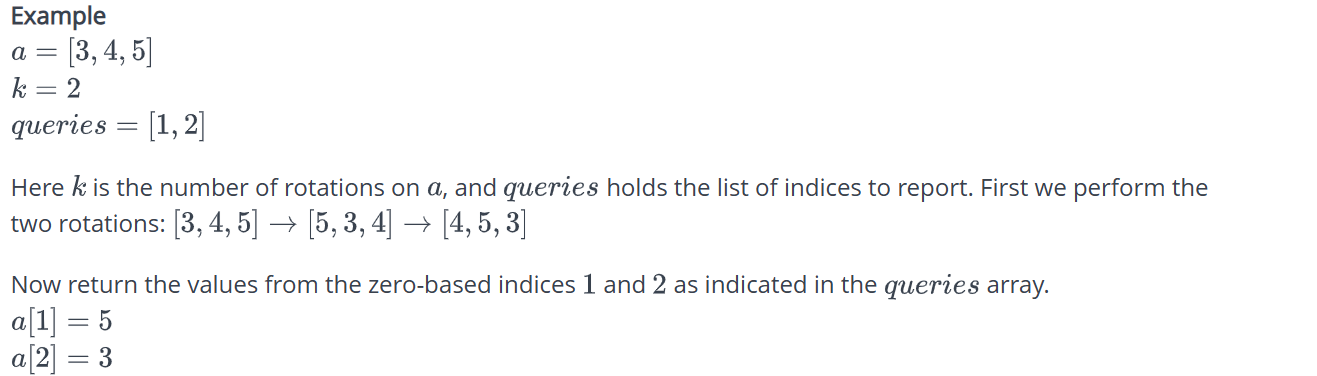
**OUTPUT:**  **"YES"**

**DAY 3:**

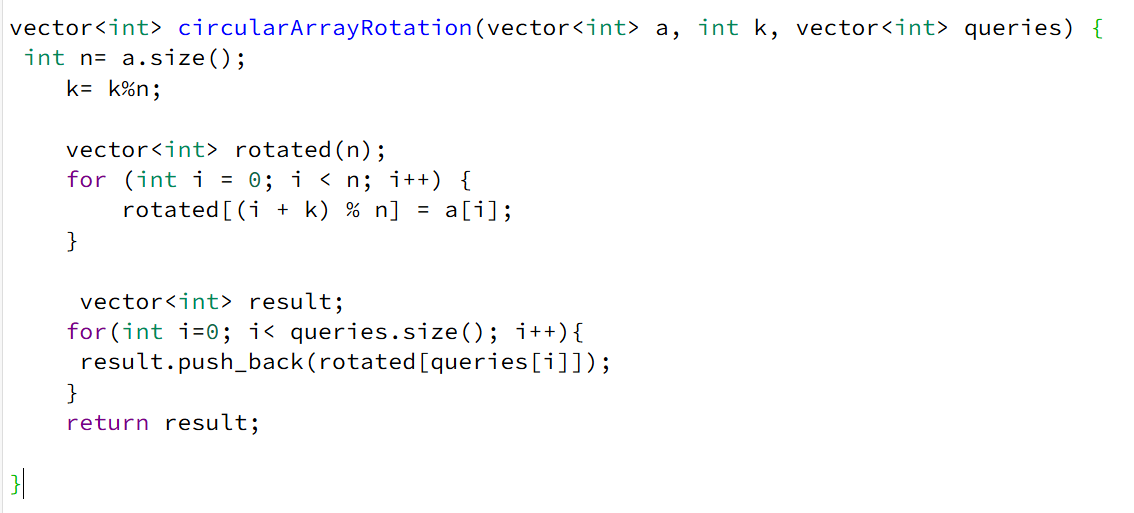
**CIRCULAR ARRAY ROTATION:**

John Watson knows of an operation called a *right circular rotation* on an array of integers. One rotation operation moves the last array element to the first position and shifts all remaining elements right one. To test Sherlock's abilities, Watson provides Sherlock with an array of integers. Sherlock is to perform the rotation operation a number of times then determine the value of the element at a given position.

For each array, perform a number of right circular rotations and return the values of the elements at the given indices.



**SOURCE CODE:**



**DRY RUN:**

**Example Input:**

a = {1, 2, 3}

k = 2

queries = {0, 1, 2}

**Step 1: Normalize k**

* n = a.size() = 3
* k = k % n = 2 % 3 = 2
* This means we rotate the array **2 times** to the right.

**Step 2: Rotate the Array**

* Initialize rotated as an empty vector of size n:

rotated = {0, 0, 0}

* Place elements in their new positions:
  + i = 0 → rotated[(0 + 2) % 3] = a[0] → rotated[2] = 1
  + i = 1 → rotated[(1 + 2) % 3] = a[1] → rotated[0] = 2
  + i = 2 → rotated[(2 + 2) % 3] = a[2] → rotated[1] = 3
* **Final rotated array:**

rotated = {2, 3, 1}

**Step 3: Process Queries**

* Initialize an empty result vector.
* Retrieve values from rotated at queried indices:
  + queries[0] = 0 → result.push\_back(rotated[0]) → result = {2}
  + queries[1] = 1 → result.push\_back(rotated[1]) → result = {2, 3}
  + queries[2] = 2 → result.push\_back(rotated[2]) → result = {2, 3, 1}

**OUTPUT:**

{2, 3, 1}

**DAY 4:**

**MAKE INTERVALS NON OVERLAPPING:**

Given an array of intervals intervals where *intervals*[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Note that intervals which only touch at a point are non-overlapping. For example, [1, 2] and [2, 3] are non-overlapping.

**Input Format**

The first line contains *n* - the number of intervals

The next n lines represent the start and end of each interval, separated by space

Example 1:

Input: *intervals* = [[1,2],[2,3],[3,4],[1,3]]

Output: 1

Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.

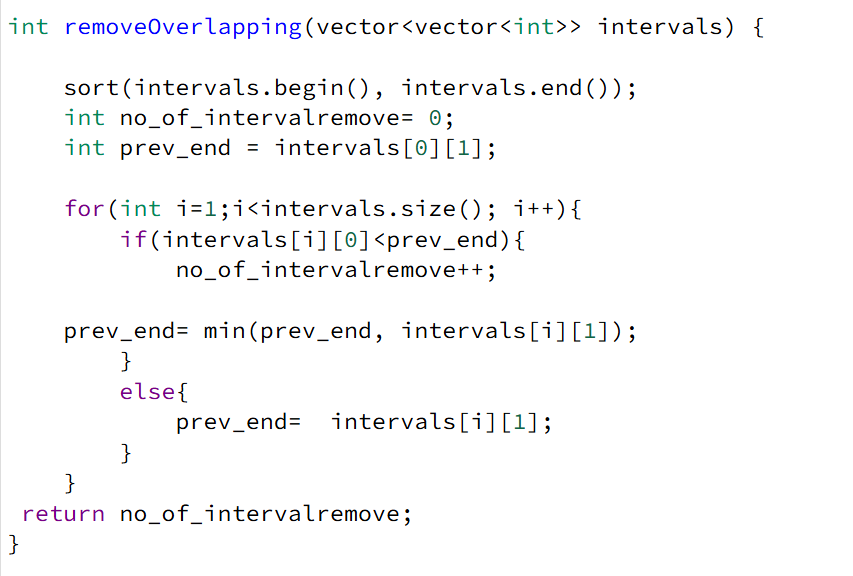
Example 2:

Input: intervals = [[1,2],[1,2],[1,2]]

Output: 2

Explanation: You need to remove two [1,2] to make the rest of the intervals non-overlapping.

**SOURCE CODE:**



**DRY RUN :**

**Example Input:**

intervals = {{1, 3}, {2, 4}, {3, 5}}

* **Sorting the Intervals:**  
  The function first sorts the intervals based on their start times. Since the input is already sorted ({{1, 3}, {2, 4}, {3, 5}}), no changes occur.
* **Initialize Variables:**
* no\_of\_intervalremove = 0 (keeps count of removed intervals)
* prev\_end = intervals[0][1] = 3 (stores the end of the last non-overlapping interval)
* **Iterate Through the Intervals:**
* **First Iteration (i = 1):**
  + intervals[1] = {2, 4}
  + Since 2 < prev\_end (3), there is an **overlap**.
  + Increment no\_of\_intervalremove = 1.
  + Update prev\_end = min(3, 4) = 3 (to remove the larger interval part).
* **Second Iteration (i = 2):**
  + intervals[2] = {3, 5}
  + Since 3 == prev\_end (3), **no overlap**, so update prev\_end = 5.
* **Final Output:** no\_of\_intervalremove = 1, meaning **one interval was removed** to avoid overlap.

**OUTPUT**: 1

**DAY 5:**

**CONNECTED CELLS IN A GRID:**

Consider a matrix where each cell contains either a  or a . Any cell containing a  is called a *filled* cell. Two cells are said to be *connected* if they are adjacent to each other horizontally, vertically, or diagonally. In the following grid, all cells marked X are connected to the cell marked Y.

XXX

XYX

XXX

If one or more filled cells are also connected, they form a *region*. Note that each cell in a region is connected to zero or more cells in the region but is not necessarily directly connected to all the other cells in the region.

**SOURCE CODE:**

int rows, cols;

vector<vector<bool>> visited;

vector<pair<int, int>> directions = {

{-1, -1}, {-1, 0}, {-1, 1},

{0, -1}, {0, 1},

{1, -1}, {1, 0}, {1, 1}

};

int dfs(vector<vector<int>>& grid, int r, int c) {

if (r < 0 || r >= rows || c < 0 || c >= cols || grid[r][c] == 0 || visited[r][c]) {

return 0;

}

visited[r][c] = true;

int size = 1;

for (auto [dr, dc] : directions) {

size += dfs(grid, r + dr, c + dc);

}

return size;

}

int getMaxRegion(vector<vector<int>>& grid) {

rows = grid.size();

cols = grid[0].size();

visited.assign(rows, vector<bool>(cols, false));

int maxRegion = 0;

for (int r = 0; r < rows; ++r) {

for (int c = 0; c < cols; ++c) {

if (grid[r][c] == 1 && !visited[r][c]) {

maxRegion = max(maxRegion, dfs(grid, r, c));

}

}

}

return maxRegion;

}

int connectedCell(vector<vector<int>> matrix) {

return getMaxRegion(matrix);

}

**DRY RUN:**

**Example Input:**

matrix = {

{1, 1, 0, 0},

{0, 1, 1, 0},

{0, 0, 1, 0},

{1, 0, 0, 0}

};

#### ****Step 1: Initialize Variables****

* rows = 4, cols = 4
* visited matrix initialized to **false** for all cells.
* maxRegion = 0
* directions array stores the 8 possible directions for DFS traversal.

#### ****Step 2: Start Traversing the Grid****

##### **(Row: 0, Col: 0)**

* **Found a 1** (matrix[0][0] = 1) and not visited.
* Call dfs(0, 0):
* Mark visited[0][0] = true, size = 1
* Explore 8 directions:
* (0,1) → matrix[0][1] = 1, call dfs(0,1)
* Mark visited[0][1] = true, size = 1
* (1,1) → matrix[1][1] = 1, call dfs(1,1)
* Mark visited[1][1] = true, size = 1
* (1,2) → matrix[1][2] = 1, call dfs(1,2)
* Mark visited[1][2] = true, size = 1
* (2,2) → matrix[2][2] = 1, call dfs(2,2)
* Mark visited[2][2] = true, size = 1
* dfs(2,2) returns 1
* dfs(1,2) returns 2
* dfs(1,1) returns 3
* dfs(0,1) returns 4
* dfs(0,0) returns 5
* maxRegion = 5 (largest region so far)
* **(Row: 0, Col: 1)**
* Already visited, skip.
* **(Row: 0, Col: 2) & (Row: 0, Col: 3)**
* matrix[0][2] = 0 and matrix[0][3] = 0, skip.
* **(Row: 1, Col: 0)**
* matrix[1][0] = 0, skip.
* **(Row: 1, Col: 1), (1,2), (2,2)**
* Already visited, skip.
* **(Row: 3, Col: 0)**
* Found new 1, call dfs(3,0), it has no connected ones.
* dfs(3,0) returns 1, maxRegion = max(5,1) = 5

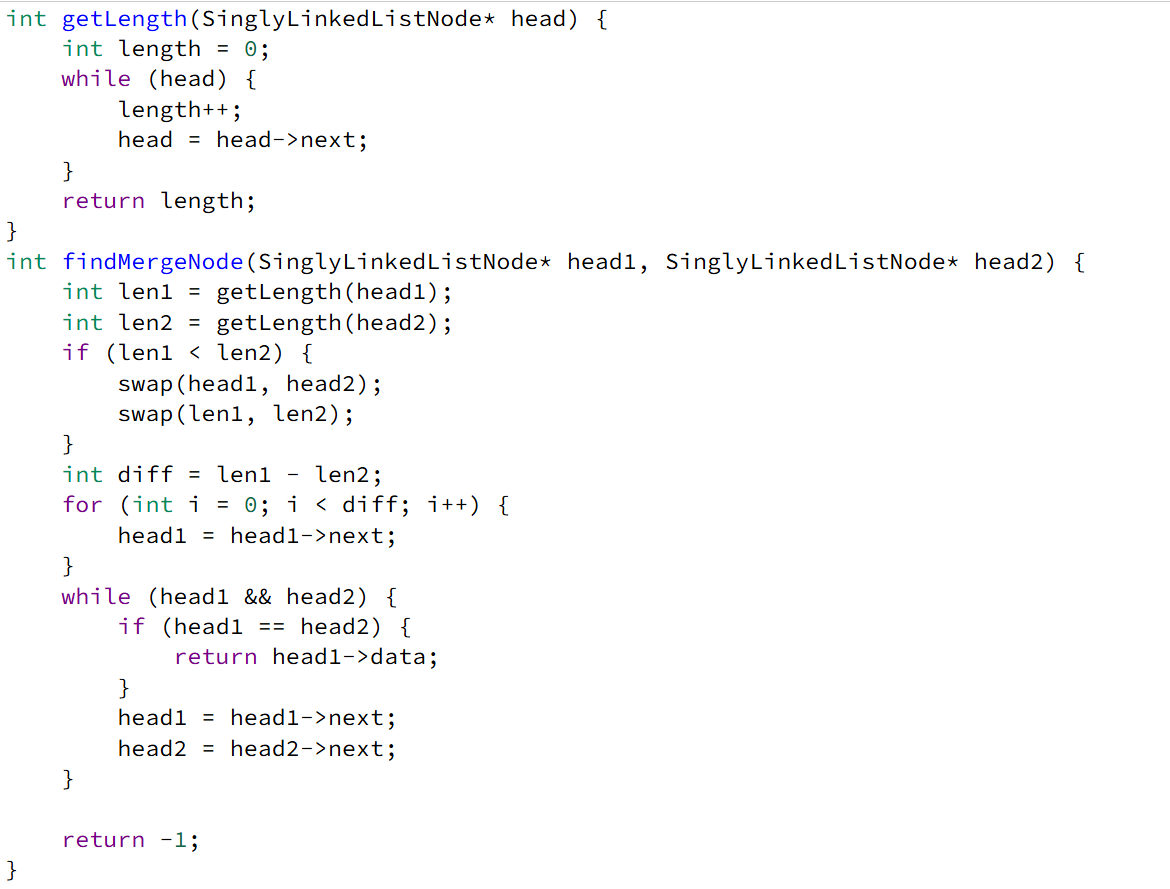
**OUTPUT:** 5

**DAY 6:**

**FIND MERGE POINT OF TWO LISTS:**

Given pointers to the head nodes of 2 linked lists that merge together at some point, find the node where the two lists merge. The merge point is where both lists point to the same node, i.e. they reference the same memory location. It is guaranteed that the two head nodes will be different, and neither will be NULL. If the lists share a common node, return that node's  data value.

**SOURCE CODE:**

****

**DRY RUN:**

**Example Input:**

List1: 1 → 2 → 3

↘

6 → 7 → 8

↗

List2: 4 → 5

The function findMergeNode first calculates the lengths of both linked lists using getLength. If one list is longer, it moves the head of the longer list forward by the difference in lengths to align both lists. Then, it traverses both lists simultaneously, comparing nodes at each step. When the two nodes are the same, it returns the data value of the merge node. If no merge point is found, it returns -1. For example, given List1: 1 → 2 → 3 → 6 → 7 → 8 and List2: 4 → 5

→ 6 → 7 → 8, after aligning the longer list, both heads move together until they meet at 6, which is returned as the merge point.

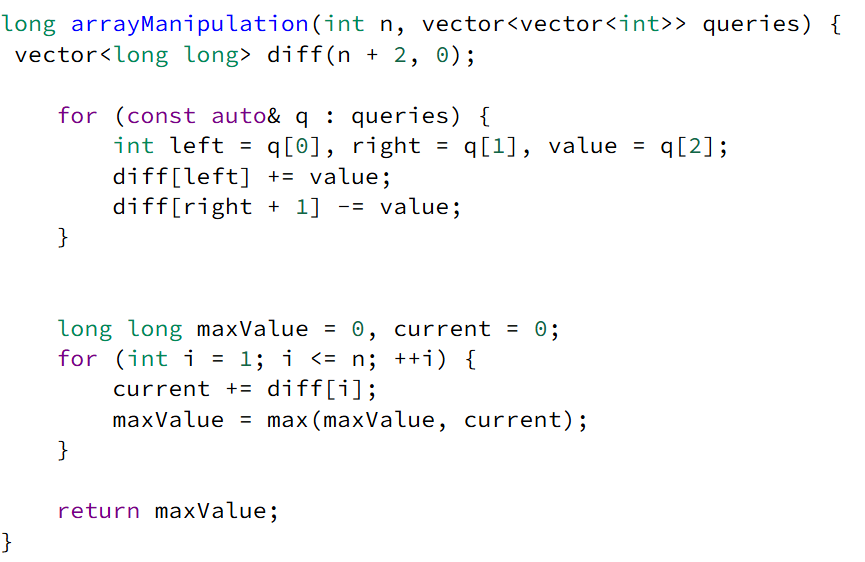
**OUTPUT:** 6

**DAY 7:**

**ARRAY MANIPULATION:**

**SOURCE CODE:**

Starting with a 1-indexed array of zeros and a list of operations, for each operation add a value to each array element between two given indices, inclusive. Once all operations have been performed, return the maximum value in the array.

****

**DRY RUN:**

**Example Input:**

n = 5

queries = {

{1, 2, 100},

{2, 5, 100},

{3, 4, 100}

}

**Step 1: Initialize Difference Array**

We create a diff array of size n+2 (i.e., 7 in this case) initialized to 0:

diff = [0, 0, 0, 0, 0, 0, 0]

### ****Step 2: Apply the Queries****

#### ****First Query (1, 2, 100)****:

* left = 1, right = 2, value = 100
* diff[1] += 100 → [0, 100, 0, 0, 0, 0, 0]
* diff[3] -= 100 → [0, 100, 0, -100, 0, 0, 0]

#### ****Second Query (2, 5, 100)****:

* left = 2, right = 5, value = 100
* diff[2] += 100 → [0, 100, 100, -100, 0, 0, 0]
* diff[6] -= 100 → [0, 100, 100, -100, 0, 0, -100]

#### ****Third Query (3, 4, 100)****:

* left = 3, right = 4, value = 100
* diff[3] += 100 → [0, 100, 100, 0, 0, 0, -100]
* diff[5] -= 100 → [0, 100, 100, 0, 0, -100, -100]