Romerico David, COSC483.001, Midterm

Professor Nguyen

10/16/2024

Question 1

**Step-by-Step Thought Process:**

1. **Initialize a Temporary Array (temp):**
   1. Create an array temp[] of size arr[].length to store the indices of all valid k-centers.
2. **Loop Through the Array:**
   1. Start from index k and loop until arr.length - k - 1, ensuring that you're not out of bounds when checking elements k positions before and after index i.
   2. For each index i, compare the element k positions before (arr[i - k]) and k positions after (arr[i + k]) the current index i.
   3. **If valid k center:**
      1. If arr[i−k]==arr[i+k], store the current index i in the temp[] array, and increment the j pointer to keep track of valid indices.
3. **Find the Maximum Value at k-centers:**
   1. Once all valid k-centers are identified and their indices are stored in temp[], initialize two variables:
      1. max: Stores the maximum value found at those positions.
      2. maxIdx: Stores the index of the element with the maximum value in temp[].
   2. Loop through the temp[] array and determine the max value and maxIdx of arr[]
4. **Return the index of the maximum k-center**

**Brute Force Pseudocode/Main Approach Solution:**

public static int getIndexOfElemKHopsAway(int[] arr, int k) {

int[] temp = new int[arr.length];

for (int i = k, j = 0; i < arr.length - k - 1; i++) {

int minusKHop = arr[i - k];

int plusKHop = arr[i + k];

if (minusKHop == plusKHop) temp[j++] = i;

}

int max = temp[0];

int maxIdx = 0;

for (int i = 1; i < temp.length; i++) {

if (arr[temp[i]] > max) {

max = arr[temp[i]];

maxIdx = i;

}

}

return temp[maxIdx];

}

**Proof of Correctness:**

The algorithm begins by identifying valid k-centers in the array. A k-center is defined as an index i where the elements k positions before and after i are equal, i.e., arr[i−k]==arr[i+k]. The first loop in the algorithm starts at index i = k and runs until i = arr.length - k - 1, ensuring that the loop does not go out of bounds when checking the elements at i−k and i+k. During each iteration, the algorithm checks whether the value at arr[i−k] is equal to arr[i+k]. If they are equal, the index i is stored in a temporary array temp[], which holds all valid k-centers. The loop invariant guarantees that by the end of the first loop, temp[] will contain all valid k-centers found in the array.

Once the valid k-centers are identified, the algorithm proceeds to find the k-center with the maximum value in the array. The second loop iterates through the indices stored in temp[] and compares the values at those positions. It initializes the maximum value (max) as the value at the first index in temp[] and then compares each subsequent value at positions in temp[]. If a larger value is found, the maximum is updated, and the corresponding index (maxIdx) is recorded. By the end of this loop, max holds the largest value among all k-centers, and maxIdx holds the index in temp[] where this maximum value was found.

The algorithm concludes by returning the index of the k-center with the largest value, which is stored in temp[maxIdx]. This ensures that the algorithm correctly identifies and returns the position of the k-center with the highest value, satisfying the problem's requirements. The bounds of the array are carefully managed, and the algorithm works even if no valid k-centers are found, as the second loop will still function correctly in that case. Therefore, the algorithm is correct and solves the problem as intended.

**Big-O Analysis:**

The first loop in the algorithm iterates from index k to arr.length−k−1, covering n−2k elements, where n is the length of the input array arr[]. During each iteration, the algorithm performs a constant-time comparison between two elements: arr[i−k] and arr[i+k]. Since each comparison takes O(1) time, the total time complexity of the first loop is O(n−2k). Given that k is typically much smaller than n, this simplifies to O(n).

In the second loop, the algorithm iterates through the temp[] array, which holds the indices of all valid k-centers. In the worst case, if every element in the array is a valid k-center, temp[] could contain up to n−2k elements. For each element in temp[], the algorithm performs a constant-time comparison to find the maximum value, resulting in a total time complexity of O(n−2k), which again simplifies to O(n).

Since the two loops run sequentially, the overall time complexity of the algorithm is the sum of the time complexities of both loops. Therefore, the total time complexity of the algorithm is O(n)+O(n)=O(n), where n is the length of the input array.

In terms of space complexity, the input array arr[] requires O(n) space, as it contains n elements. The temporary array temp[], which is used to store the indices of valid k-centers, can hold up to n−2k elements in the worst case, meaning it also requires OO(n) space. Additionally, the algorithm uses a few constant-space variables such as i, j, max, and maxIdx, which require O(1) space. Therefore, the overall space complexity of the algorithm is O(n), dominated by the input array and the temporary array.

­

Question 2

**Step-by-Step Thought Process:**

**Brute Force Solution:**

**Main Approach:**

**Proof of Correctness:**

**Big-O Analysis:**

Question 4

**Step-by-Step Thought Process:**

When I first approached the problem of removing duplicates from an array, I instinctively went with a brute-force method. The basic idea was simple: I would iterate through each element in the array and compare it against the elements I had already seen. If it wasn’t a duplicate, I would store it in a temporary array.

To implement this, I created a new array temp where I would store unique values. For each element in the original array, I checked if it was already in temp by looping through the array I had built so far. If it wasn’t there, I added it and kept track of how many unique elements I had stored. Once I had processed all elements, I copied the non-duplicate values from temp into a result array of the correct size.

This solution worked, but I quickly noticed a major inefficiency: every time I added a new element, I had to loop through the existing unique elements to check for duplicates. This resulted in a nested loop, making the time complexity O(n^2). While this approach was conceptually straightforward, I realized it was not scalable for large datasets.

That’s when it hit me: I was doing redundant work by manually checking for duplicates. There had to be a more efficient way to handle this problem. I thought about using a Set, which by design only allows unique elements. If I could leverage a Set, I could avoid the need for manual duplicate checking entirely.

By switching to a LinkedHashSet, I knew I could both maintain the order of elements and ensure uniqueness. This meant I could simply iterate through the array once, adding elements directly into the Set. The Set would automatically discard duplicates for me, eliminating the need for the inner loop.

With this optimization, the process became much simpler and faster. I iterated through the array, added each element to the LinkedHashSet, and then converted the Set into a result array. This reduced the time complexity to O(n), a significant improvement.

In hindsight, using a Set was an obvious solution, but going through the brute-force approach helped me understand the inefficiencies, leading me to this more elegant and efficient solution.

**Brute Force Solution:**

public static double[] bruteRemoveDuplicates(double arr[]) {

double[] temp = new double[arr.length];

int size = 0;

for (int i = 0; i < arr.length; i++) {

boolean isDuplicate = false;

for (int j = 0; j < size; j++) {

if (arr[i] == temp[j]) {

isDuplicate = true;

break;

}

}

if (!isDuplicate) {

temp[size++] = arr[i]Q2q;

}

}

double[] result = new double[size];

for (int i = 0; i < size; i++) {

result[i] = temp[i];

}

return result;

}

**Main Approach:**

public static double[] optimalRemoveDuplicates(double arr[]) {

LinkedHashSet<Double> temp = new LinkedHashSet<Double>();

for (int i = 0; i < arr.length; i++) {

temp.add(arr[i]);

}

double[] result = new double[temp.size()];

int i = 0;

for (double val : temp) {

result[i++] = val;

}

return result;

}

**Proof of Correctness:**

For both the brute-force and optimized solutions, the goal is to remove duplicates from the array and return only unique elements while maintaining the original order of appearance.

1. Brute Force Solution: The brute-force solution iterates through each element in the input array and checks if it has already been added to a temporary array (temp). If the element is not a duplicate (i.e., it hasn’t been encountered before), it is added to temp. Once all elements have been processed, the unique elements are transferred into the final result array. Since every element is compared to previously encountered elements and only unique values are stored, this approach correctly removes duplicates.
2. Optimized Solution: The optimized solution uses a LinkedHashSet to store unique elements. The Set data structure inherently ensures that no duplicates are stored, and the LinkedHashSet preserves the insertion order of elements. As the array is traversed, elements are added to the Set, which automatically discards duplicates. The final result array is constructed from this Set, guaranteeing that all elements are unique and in their original order. Thus, the optimized solution also correctly removes duplicates.

**Big-O Analysis:**

1. Brute Force Solution: The brute-force solution involves a nested loop. For each element in the array, the algorithm checks whether the element already exists in the temp array. This leads to a worst-case time complexity of O(n^2), where n is the number of elements in the input array. This is because for each element, the inner loop may need to iterate over all previously stored elements, resulting in quadratic behavior. The space complexity is O(n) because the algorithm creates an extra array (temp) to store unique elements, which can grow to the size of the input array in the worst case, and the final result array is also of size n.
2. Optimized Solution: The optimized solution uses a Set, which allows for average constant-time checks and insertions. Therefore, the time complexity of this solution is O(n), as each element in the array is processed once and inserted into the Set in constant time on average. After that, copying the elements from the Set to the result array takes O(n) time, so the overall time complexity remains O(n). The space complexity is O(n) as well, since the LinkedHashSet stores up to n unique elements, and the result array also requires O(n) space.

Question 5

**Step-by-Step Thought Process:**

**Brute Force Solution:**

**Main Approach:**

**Proof of Correctness:**

**Big-O Analysis:**