Karim Maamari

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithms** | **Monte Carlo or Las Vegas?** | **Expected Running Time** | **Worst-case Running Time** | **Probability of returning a Minority Element** |
| Algorithm 1 | Las Vegas | Varies Depending on whether the element is found early/late in the population | Infinite since it is possible for the algorithm to never find a Minority element | 1 |
| Algorithm 2 | Monte Carlo | O(n) | O(n) | 1 - (1 - p) ^100 |
| Algorithm 3 | Las Vegas | O (n log n) | O (n2) | 1 |

**Justification for Algorithm 1:** It is a Las Vegas algorithm since it always returns the correct answer with probability 1. The Expected running time varies depending on how early or late the algorithm finds a minority element and the probability of returning the Minority element is 1 since the algorithm is going to keep looking until it finds it.

**Justification for Algorithm 2:** It is a Monte Carlo algorithm because it is possible for it to return an incorrect answer. The Expected running time is O(n) since we are looping 100 and every iteration takes O(n) time. Thus, O(100n) = O(n). The worst-case running time is also O(n) because it will loop 100 times checking a random element each time. Finally, the probability of returning a minority element is 1-(1-p) ^100 since the probability of not finding a minority element in any of the 100 iterations is (1 - p) ^100, the probability of selecting a non-minority element in any of the iteration is (1 - p). Therefore, the probability of finding a minority element in at least one iteration is 1 - (1 - p) ^100.

**Justification for Algorithm 3:** It is a Las Vegas algorithm because it always returns the correct answer with probability 1. The expected running time is O (n log n) because it takes O(n) to randomly rearrange the elements and it takes O (log n) to check every element. The worst-case running time is 0(n2)as, once again, it takes O(n) time to put the elements in a random order and it may take up to O(n) time to check each element. Finally, the probability of finding a Minority element is 1 as the algorithm will always provide us with the minority element if it exists.

**Next Problem:**

(a)

Pseudocode:

Count = 0

For (i=0; i<n; i++)

For (j=i+1; j<n; j++)

If (A[i] > A[j])  
 count++

Return count

The running time of this algorithm is O(n2) since we have two for loops that each go to n. All other statement is run in constant time. What we are doing is checking every possible pair (i, j) and seeing whether they satisfy the inversion or not. If they do, we add it to the count.

(b)

Pseudocode:

1. Merge sort Array A and create new array which is a copy of it we will call array B.

2. Take A [1] and find its index in array B using binary search. The number of inversions will be i-1

(i being the index of A [1] in array B)

3. add the number of inversions to the counter and remove A [1] from array A and its corresponding position in array B

4. Repeat the same process until A is empty.

The correctness of the algorithm follows from the fact that the number of inversions in A is equal to the number of times that an element in A is smaller than an element to its right. The algorithm correctly counts all such inversions because, for each element A[i], it finds its index in the sorted array B, and since B is sorted, all elements before A[i] are smaller than A[i]. Therefore, the number of inversions involving A[i] is equal to the number of elements before A[i] in B, which is (i - 1).

The merge sort would take O (n log n) time to run. Step 2 would be executed n times since there are n elements in A and we are performing a binary search each time, thus, it would run in O (n log n) time as well. So, the total running time would be of O (n log n) +O (n log n) = O (n log n)

**Next Problem:**

* Possible solution:

Step 1: Initialize three pointers, i, j, and z, so that i points to the first duck in the line, j points to the last duck in the line, and z points to the first duck in the unsorted section of the line.

Step 2: Repeat until z > j:

a. Ask the duck at position z about its favorite activity.

b. If the duck is honking, swap the duck at position i with the duck at position z, increment i and z.

c. If the duck is eating, swap the duck at position z with the duck at position j and decrement j.

d. If the duck is idling, increment z.

3. At this point the line is now sorted.

The algorithm works by maintaining three pointers: i for the first unsorted position for honking ducks, j for the first unsorted position for eating ducks, and z for the first unsorted position overall. The algorithm then checks the activity of the duck at position z, and swaps it to the appropriate side of the line based on its activity. If the duck is honking, it is swapped to the honking section of the line at position i, which is then incremented to the next unsorted position for honking ducks. If the duck is eating, it is swapped to the eating section of the line at position j, which is then decremented to the next unsorted position for eating ducks. If the duck is idling, it is simply left in place in the middle of the line, and z is incremented to the next unsorted position overall. This process is repeated until all ducks are sorted.

(b) Correctness: The algorithm works by maintaining three pointers that divide the line into three sections: honking ducks, idling ducks, and eating ducks. The algorithm then iterates through the unsorted section of the line, swapping each duck to its appropriate section based on its activity. Since the honking ducks are placed at the beginning of the line, and the eating ducks are placed at the end of the line, the algorithm successfully sorts the ducks as required.

Runtime: The algorithm maintains a constant number of pointers and performs a constant amount of work for each unsorted duck, so we can conclude that the runtime is of O(n).

Memory usage: The algorithm requires only three pointers and a constant amount of memory for each unsorted duck. Since the algorithm is iterating through the ducks one at a time, it never needs to remember more than seven integers at a time. So, it satisfies the seven integer rule.