# Algorithms: Assignment #4

Tanvi Magdum
magdum.t@northeastern.edu

February 6, 2023

## Answers

# 1 Problem 1 Solution :

## 1.1 Problem 1a Solution -

Efficient sorting algorithm for an array C[1,...,n] whose elements are taken from the set {1,2,3,4,5,6,7}-

In this case, there can be multiple cases like input array could contain repeated elements, hence n is not limited to 7 or the number of elements in given set. But we know for sure that elements are taken from given set only and given set does not contain any negative elements. So we can use Counting sort in this case.

Algorithm-

1. Define an output array of size same as input array.
2. Define a count array of size equal to (size of set + 1), i.e in this case size=8.
3. Iterate over input array to store the count of each element into count array.
4. Iterate over count array to store cumulative count of each element.
5. Iterate over input array in reverse order and place each element correctly in output array using the cumulative count in count array.
6. Copy sorted output array to input array.

Let us calculate the cost for each step. Assume that cost is 1 for comparison, arithmetic operation, assignment operation and declaration.

1. int n = c.length; ——————————————————> This line assigns length of input array to a variable and has a time complexity of O(1).
2. int[] b = new int[n]; ——————————————————> This line creates an array of size n to store the output and has a time complexity of O(n).
3. int[] count = new int[k + 1], here k=7, size of set ——> This line creates an array of size k+1 to store the count and has a time complexity of O(k).
4. for (int i = 0; i < n; i++) ——————————————————> This line initializes the loop that will iterate over input array and has a time complexity of O(n).
5. count[c[i]]++; ——————————————————> This line increments value of count array at index c[i]. Since, increment is constant time operation, time complexity is O(1).
6. for (int i = 1; i <= k; i++) ——————————————————> This line initializes the loop that will iterate over count array and has a time complexity of O(k).

7. count[i] += count[i - 1]; ——————————————> This line performs arithmetic operation which is constant time, time complexity is O(1).
8. for (int i = n - 1; i >= 0; i–) { ——————————————-> This line initializes the loop that will iterate over input array and has a time complexity of O(n).
9. b[count[c[i]] - 1] = c[i]; ——————————————-> This line is an assignment, a constant time operation, time complexity is O(1).
10. count[c[i]]- -; } ——————————————————> This line performs decrement, a constant time operation, time complexity is O(1).
11. for (int i = 0; i < n; i++) ——————————————> This line initializes the loop that will iterate over input array and has a time complexity of O(n).
12. c[i] = b[i]; ——————————————————> This line is an assignment, a constant time operation, time complexity is O(1).

So the total cost of this algorithm becomes-

O(1)+O(n)+O(k)+O(n)*O(1)+O(k)*O(1)+O(n)*(O(1)+O(1))+O(n)*O(1) = O(n)

Since O(1) is constant time we ignore it. O(k) is also constant time because we already know the value of k in this case, which is 7. Overall what remains is O(n). The counting sort algorithm counts the frequency of each element in the input array and uses this information to determine the position of each element in the sorted output array. The sorting process itself takes a linear amount of time. Hence, the time complexity in this case is O(n).

## 1.2 Problem 1a Solution -

Efficient sorting algorithm for an array D[1, ..., n] whose elements are distinct ($D[i] \neq D[j]$, for every $i \neq j \in \{1, ..., n\}$) and are taken from the set {1, 2, ..., 2n}.

We can use Bucket sort algorithm here. We know that elements lie in between a range of 1 to 2n, and bucket sort is efficient algorithm for sorting elements that are uniformly distributed. Bucket sort works by dividing elements into several buckets and then sorting each bucket individually. Since the elements of array D[1, ..., n] belong to a particular set {1, 2, ..., 2n}, the number of buckets required would be 2n and each element would belong to a unique bucket. After sorting individual buckets, we can get final sorted array by concatenating the sorted buckets.

Algorithm-

1. Create an array of 2n+1 empty buckets(lists), each bucket contains elements corresponding to their respective array indices.
2. Loop through input array and place each element into its corresponding bucket depending upon its value, for ex. 4th bucket will contain all elements with value 4, if none is present in array then 4th bucket will be empty.
3. Loop through the buckets and concatenate the elements in each bucket to form a sorted array.

Let us calculate the cost for each step. Assume that cost is a constant 1 for adding elements in list, comparison, arithmetic operation, assignment operation and declaration. We will ignore additional constants, because they are negligible and do not affect the time complexity.

1. int n = c.length; ——————————————————> This assigns length of input array to a variable and has a time complexity of O(1).
2. List<Integer>[] buckets = new List[2*n + 1]; ————> This line creates an array of size (2n + 1) to store the buckets and has a time complexity of O(2n).

3. for (int i = 0; i < buckets.length; i++) ————————-> This line initializes the loop that will iterate over all the buckets and has a time complexity of O(2n).

4. buckets[i] = new ArrayList<>(); ————————————> This line creates an empty ArrayList for each bucket and has a time complexity of O(1).

5. for (int i = 0; i < n; i++) ————————————————> This line initializes the loop that will iterate over input array and has a time complexity of O(n).

6. buckets[c[i]].add(c[i]); ————————————————-> This line adds an element from the input array to the appropriate bucket and has an average time complexity of O(1) as adding an element to the end of an ArrayList is an a constant-time operation.

7. int j = 0; ————————————————————————-> This line initializes a variable which is O(1) time complexity.

8. for (int i = 0; i <= 2*n; i++) { ————————————-> This line initializes the loop that will iterate over all the buckets and has a time complexity of O(2n).

9. for (int k = 0; k < buckets[i].size(); k++) { ——————-> This line initializes the loop that will iterate over the elements in a bucket and has a time complexity of O(bucket_size).

10. c[j++] = buckets[i].get(k); } } ————————————————> This line takes an element from a bucket and places it in the sorted output array. The time complexity is O(1) as it is just an assignment.

So the total cost of this algorithm becomes-

O(1)+O(2n)+O(2n)*O(1)+O(n)*O(1)+O(1)+O(2n)*O(bucket_size)*O(1) = O(2n) = O(n)

Since O(1) is constant-time we ignore it. O(bucket_size) is also constant time because, elements in array are distinct. So, there are only two possibilities, either the bucket size is 0 or it is 1. Thus, the for loop condition is as good as putting it into and if-else statement for this particular problem. Overall what remains is O(2n). The gap between O(2n) and O(n) is negligible, hence the time complexity of this algorithm is O(n) i.e. linear time complexity for this specific problem.

# 2 Problem 2 Solution :

No, it doesn't mean that the lower bound for sorting of $\Omega(n \log n)$ is wrong. The lower bound of $\Omega(n \log n)$ for sorting is a theoretical lower bound, which means that any sorting algorithm will take at least this much time in the worst case scenario. It is based on mathematical proofs and is not dependent on the specific algorithm being used. Thus it applies to all general purpose sorting algorithms, sorting in ascending or in descending order. There can be certain special cases where sorting algorithms with better time complexity than $\Omega(n \log n)$ exist, but those are usually specific to certain constraints or assumptions about the input data. In our case, in previous problems we had been provided with some conditions on input like the elements of array belonged to a specific set of elements. The elements were known and well-defined, so the time complexity was linear.

We used counting algorithm in order to achieve the best time complexity. But suppose if the given set would have consisted negative numbers. Then the current approach would not be applicable because counting sort is not suitable for arrays containing negative elements. It assumes that elements are non-negative integers and uses their frequency as indices in a temporary counting array. So, negative elements would lead to negative indices in counting array, which is not supported by most programming languages. Another version of the same problem would be that consider if the array consisted of fractions. Again the indices cannot be in fractional form, so counting sort is not applicable. In second case, we used Bucket sort. Bucket sort would have costed more if the elements would have been indistinct. We saw that bucket_size scaled down to 0 or 1 in that case, that's why we got the linear time complexity. Otherwise, if there is no constraint or no data set provided, the worst-case complexity of bucket sort is O($n^2$), which is not good. Though we get a better time complexity with these algorithms, we need to make sure if our problem fits into that frame of constraints, then and only then one can think of finding a better time complexity.

In general, for most practical purposes, a sorting algorithm with a time complexity of $\Omega(n \log n)$ is considered to be the best that can be achieved for sorting an array of n elements. The lower bound assumes that a comparison-based sorting algorithm is being used, which means that algorithm relies on comparing elements in an array to determine their relative order. It has been proven that any comparison-based sorting algorithm will take atleast $\Omega(n \log n)$ time in worst-case scenario. The lower bound concept is valid for comparison-based sorting algorithms, not for all sorting algorithms.

# 3 Problem 3 Solution :

To solve Closest pair problem in 3D we can use clever divide and conquer problem, similar to the one used to solve the Closest Pair problem in 2D.

Algorithm-

1. Sort the points in x-coordinate using a sorting algorithm like merge sort. This gives us points sorted in increasing x-coordinate. Time complexity : O($n \log n$)
2. Split the points in two sets, L and R using a median of the x-coordinates. Time complexity : O(1)
3. Find the closest pair in each set L and R recursively. Let the closest pair in the left set be (A, B) and in the right set be (C, D). Time complexity : 2T(n/2), where T(n) is the time complexity of finding the closest pair
4. Find the minimum of distances between closest pair in left and right set. Let closest pair in 3D space be (E, F). Time complexity : O(1)
5. Create a strip of width 2 times the minimum of distances between the closest pair in the left and right set. Let this strip be "strip area". Time complexity : O(1)
6. Sort the points in strip by y-coordinate. Time complexity : O($n \log n$)
7. For each point in strip area, check distance of 7 points closest to it along y-axis. Time complexity : O(n)
8. Update the closest pair in strip area if necessary. Time complexity : O(1)
9. Return the closest pair found in the strip area as the closest pair in 3D space. Time complexity : O(1)
10. Repeat 3 to 8 until closest pair is found.

So, the total time complexity is-

T(n) = 2T(n/2) + (nlogn)

We can improve the algorithm by reducing the overhead nlogn. nlogn occurs due to sorting and we can avoid sorting in each step we can make recursion return a sorted list in every half and merge in linear time to get an entire sorted list. Since, the sorting occurs in linear time, we get linear time overhead i.e. n instead of nlogn. Thus, T(n) becomes-

T(n) = 2T(n/2) + n

Applying Master Theorem,

a = 2, b = 2, f(n) = n

$\therefore n^{\log_b a} = n^{\log_2 2} = n^1 = n$

f(n) = n

By using case 2 of Master theorem we can say that,

T(n) = $\Theta(n \log n)$