CSOR 4246 Algorithms for Data Science (Fall 2022)
Problem Set #1t

Xinhao Li - xl2778@columbia.edu

2022/10/03

Problem 1

(a)

In this question, we are asking to do a Mergesort recursively break lists into 3 equal-sized sublists instead of 2. We know that the running times of many recursive algorithms can be expressed by the following recurrence:

$$T(n) = aT(n/b) + cn^k, for\ a,\ c > 0,\ b > 1,\ k \geq 0 \tag{1}$$

where a is the branching factor, b is the factor by which the size of each subproblem shrinks, and the cn^k term represents the cost of dividing the problem and combining the results of the subproblems. Therefore, we know that $a = b = 3, k = 1$(based on the running time of merge function). Finally based on the master theorem, we have:

$$a = b^k \rightarrow T(n) = O(nlogn) \tag{2}$$

(b)

Intro

In this question, we are given a set S of n distinct integers and another integer x. We need to design an algorithm to determine whether or not there exist two elements in S whose sum is exactly x. Basically we can design the algorithm like this: (1) First, we need to do Mergesort on the given input set S and this can make sure that the given input set S is sorted. (2) Then, we can use two pointers and they are put on the most left (lp) and most right (rp) of the input set S, respectively. (3) By having the two pointers initialized, we can start to compare the sum of the two elements that two pointers pointed to: $a = sum(S[lp] + S[rp])$. (4) If a == x, then the algorithm will directly return **True**. (5) If a > x, this means that we have to move the right pointer (rp) to the left by just one index (rp -1) and recalculate the sum of two elements (one element is updated): $a = sum(S[lp] + S[rp - 1])$ (6) If a $\leq$ 0, this means that we have to move the left pointer (rp) to the right by just one index (lp + 1) and recalculate the sum of two elements (one element is updated): $a = sum(S[lp + 1] + S[rp])$. (7) Finally if we do not see the sum of any two elements in the set S is equal to x, then the algorithm will return **False**.

Pseudocode

The pseudocode can be:

```
def checkingtwosum(S, x):
        Mergesort(S)
        lp, rp = 0, len(S)
        while lp < rp:
                if S[lp] + S[rp] == x:
                        return True
                elif S[lp] + S[rp] < x:
                        lp += 1
                else:
                        rp -= 1
        return False
```

The correctness of Mergesort algorithm has been proved in the class. So the only thing here we need to prove here is the algorithm that checking the sum of two elements in the set.

Proof of correctness by induction and contradiction

First of all, let's still have the two pointers on the most left (lp) and most right of the set. Let's first assume that

$$solution[i,j] \in S(i \geq lr, j \leq rp, \text{and } i > j). \tag{3}$$

There are three possible cases, the first case will be: S[lp] + S[rp] == x. And the algorithm will return **True**. For the second case, we have S[lp] + S[rp] > x. As mentioned in the algorithm designed above, the right pointer (rp) will move to the left by one index (rp-1). In other words, this means that the most right element has been removed from the set S and now the set S becomes to set S$'$. Now let's prove this by using the contradiction and introduce the second assumption:

$$solution[i,j] \notin S'(S' = S[lp, rp-1]) \tag{4}$$

On the basis of the hypothesis 1 and 2 (equation 3 and 4), we know that the solution will definitely be in the set S and it is not in the set S$'$. Therefore, we can say that the solution has to include the only element that has been removed by us previously. So now, we have (remember we are still discussing about the second case):

$$S[i] + S[rp] = x < S[lp] + S[rp] \tag{5}$$

Let's subtract S[rp] from both side and we have:

$$S[i] < S[lp] \tag{6}$$

So now, the problem is pretty obvious. It is not possible to have index i smaller than index lp since lp is already pointed to the most left element in the set S. Thus this means that our hypothesis 2 (equation 4) is not correct and we conclude that:

$$solution[i,j] \in S'(S' = S[lp, rp-1]) \tag{7}$$

Finally for the last case, it is just the symmetrical case of case 2, so it can be proved by the same argument.

Running time

The worst case running time for Mergesort has been proved in the class and it is $O(nlogn)$. For the algorithm that designed to check the two sum, the pointers initialization step takes constant time and we only loop once for the whole set S. So the running time for this part should be $O(n)$. Then the running time for the whole algorithm is $O(nlogn)$.

Problem 2

Pseudocode

In this question, we are asked to design a new recursive algorithm for sorting that runs in three phases.

```
def newmergesort(A, left, right):
        if left == right:
                return A
        first = left
        second = left + 2/3 * (right - left)
        third = left + 1/3 * (right - left)
        newmergesort(A, first, second)
        newmergesort(A, third, right)
        newmergesort(A, first, second)
        merge(A, first, second, right)
```

Proof of correctness

Basically, first we will sort the first [2/3n] elements of the list. This means that for any element in the first [1/3n] of the list must be equal or smaller than any element in between [1/3n + 1] and [2/3n] of the list. The second phase will then compare the elements in between [1/3n + 1] and [2/3n] of the list with the last [1/3n] of the list. After the second phase, we can make sure that any element in the first [2/3n] of the list must be equal or smaller than any element in the last [1/3n] elements of the list because the second phase is actually comparing the "large" elements found in phase 1 with the rest elements in the list to determine the actual "large" elements and put them in the last [1/3n] of the list. Lastly, the third phase is required to sort again the first [1/3n] elements of list since we do not know whether the "small" elements found in phase 2 are actually all smaller than "small" elements found in phase 1.
To proof by induction:

- Base case: assume n = 3^k for integer k $\geq$ 0. For k = 0, the input consists of 1 item; **newmergesort** returns the item.

- Induction hypothesis: For k $\geq$ 0, assume the **newmergesort** correctly sorts any list of size 3^k.
- Induction step: Let's show that the **newmergesort** correctly sorts any list A of size 3^{k+1}.
  - Line 3: first takes the value 1
  - Line 4: second takes the value 2/3 * 3^{k+1}
  - Line 5: third takes the value 1/3 * 3^{k+1}
  - Line 6: **newmergesort**(A, 1, 2/3 * 3^{k+1}) can not correctly sort the first [2/3n] elements of the list by the induction hypothesis since the number of elements now $(2 * 3^k)$ is larger than 3^k. However, if we only sort first half of the first [2/3n] elements of the list, it will be done successfully. Now we still have [1/3n] elements in the middle of the list are not sorted.
  - Line 7: Same as Line 6, but this time the last [1/3n] elements of the list will be sorted. So again, we still have [1/3n] elements in the middle of the list are not sorted.
  - Line 8: Same as Line 6, but this time the middle [1/3n] elements of the list will be sorted.
  - Line 9: Merge correctly merges three sorted lists into one sorted output.
- **newmergesort** correctly sorts any input of size 3^{k+1}

Running time

The running time for the merge function is still $O(n)$. For the **newmergesort** function, we will the input will be separated as 3 sub-list and each sub-list has size 2/3n. Let T(n) be the running time for the whole algorithm:

$$T(n) = 3T(\frac{2}{3}n) + O(n) \tag{8}$$

Based on master theorem, we have:

$$a = 3, b = \frac{3}{2}, k = 1 \rightarrow a > b^k \rightarrow T(n) = O(n^{log_{\frac{3}{2}}3}) \tag{9}$$

We know that $log_{\frac{3}{2}}3 > log_{\frac{3}{2}}\frac{9}{4} = 2$. This means that:

$$T(n) = O(n^{log_{\frac{3}{2}}3}) > O(n^2) \tag{10}$$

Based on the running time calculated above, I will not use this algorithm to do any sorting since it is slower than normal merge sort.

Problem 3

In this problem, we are given a sequence of n distinct numbers $x_1, x_2, ..., x_n$ and want us to understand how far this sequences is from being in ascending order.

(a)

For the brute force algorithm, it is pretty simple. So basically we will just compare each element in the given sequence with other elements from the beginning of the sequence. If the latter element is smaller than front element in the for loop, then the disorder will plus 1.

Pseudocode

Let's have the sequence and the length of sequence as A and n, respectively.

```
def disorder_counter1(A, n):
        disorder = 0
        for i in range(n-1): # exclude the last element
                for j in range (i+1, n):
                        if A[i] > A[j]:
                                disorder += 1
        return disorder
```

Proof

- Base case: n = 1. This means that in the sequence there is only one number, so the disorder number is 0.
- Inductive hypothesis: For n $\geq$ 1, disorder can be calculated correctly.
- Inductive step: Show that the statement also holds *True* for n + 1.
  - When we have a new element at the end of sequence, we need to compare all existing elements (A[1], A[2], ..., A[x], ..., A[n]) in the sequence with the new added element (A[n+1]): if A[x] > A[n+1], we will add 1 into the disorder counter.
  - After comparing all existing elements with the new added element, the disorder should be calculated correctly.

Running time

In the above algorithm, it is pretty clear that we have two nested for loops. So the running time should be $O(n^2)$.

(b)

As for the faster algorithm to compute the disorder, we can simply use the idea from merge-sort and calculate the disorder recursively. So basically, we first split the input sequence into two sequences of equal size. Then we will sort each sequence recursively (stop when sequences have size 2). During the merge process, we will have two sequences, one sequence on the left and one sequence on the right. If the element in the right sequence will be added to output, then the number of remaining elements in the left sequence will be counted as the disorder number. Since adding an element from the right sequence means that this element is smaller than some/all elements in the left sequence.

Pseudocode

Let A, left, right be the input sequence, the first element in the sequence, and the last element in the sequence, respectively.

```
def disorder_counter2(A, left, right):
        disorder = 0
        if left == right:
                return 0
        mid = left + (right - left) / 2
        disorder += disorder_counter2(A, left, mid)
        disorder += disorder_counter2(A, mid+1, right)
        disorder += counter(A, left, mid, right)
        return disorder
def counter(A, left, mid, right):
        L = A[left: mid]
        R = A[mid: right]
        lp, rp, n = 0, 0, 0
        num = 0
        while both two lists are not empty:
                if L[lp] <= R[rp]:
                        A[n] = L[lp]
                        lp += 1
                        n += 1
                else:
                        A[n] = R[rp]
                        num += len(L[lp:]
                        rp += 1
                        n += 1
        Append the non-empty list to the output
        return num
```

Proof

(a) First need to prove function **counter** is correct:

- Base case: n = 1. This means that in the sequence there is only one number, so the disorder number is 0.
- Inductive hypothesis: For n $\geq$ 1, disorder can be calculated correctly.
- Inductive step: Show that the statement also holds *True* for n + 1.
  - Based on the inductive hypothesis and the above algorithm, for the first n elements, we know that by applying the algorithm, we will have the sequence sorted and disorder can be calculated correctly.
  - ...

- ...
- ...

(b) Then need to prove function **disorder_counter2** is correct:

- Base case: assume n = 2^k for integer k $\geq$ 0. For k = 0, the input consists of 1 item; **disorder_counter2** returns the disorder to be 0.
- Induction hypothesis: For k $\geq$ 0, assume the **disorder_counter2** correctly sorts any list of size 2^k.
- Induction step: Let's show that the **disorder_counter2** correctly returns disorder for any list A of size 2^{k+1}.
  - Line 4: mid takes the value 2^k.
  - Line 5: **disorder_counter2**(A, 1, 2^k) can correctly return the disorder for the leftmost half of the input by the induction hypothesis.
  - Line 6: Same as Line 5, **disorder_counter2**(A, 2^k+1, 2^{k+1}) can correctly return the disorder for the rightmost half of the input by the induction hypothesis.
  - Line 7: **counter** correctly returns the disorder when merges the two sorted lists into one sorted output of size $2^k + 2^k = 2^{k+1}$.
- **disorder_counter2** correctly returns the disorder for any input of size 2^{k+1}.

Running time

This algorithm is basically the same as mergesort. The only difference in the *merge* function is to define a counter when any element from the right part of the input has been added to the output. So the total running time is $O(nlogn)$.

Problem 4

(a) Let P be the set of n points.

Pseudocode

```
def distance(p1, p2):
        # p1 and p2 are two points in P.
        # Each consists of x and y.
        return sqrt((p1[0]- p2[0])**2 + (p1[1]- p2[1])**2)

def find_closest_pair1(P, n):
  if n <= 1:
    return False
  else:
```

```
    closest_pair = INIFINITE LARGE
      for i in range(n-1): # exclude the last element
        for j in range(i+1, n):
          dist = distance(P[i], P[j])
          if dist < closest_pair:
            closest_pair = dist
  return closest_pair
```

Proof

- Base case: n = 1. This means that in the plane there is only one point, so it will return False.
- Inductive hypothesis: For n > 1, disorder can be calculated correctly.
- Inductive step: Show that the statement also holds *True* for n + 1.
  - Based on the induction hypothesis, let's assume now the *cloest_pair* is a.
  - When we have a new point on the plane, we need to calculate distance between all existing points (p_1, p_2, ..., p_x, ..., p_n) and the new added point (p_{n+1}): if distance(p_x, p_{n+1}) < a, we will update the *cloest_pair* to distance(p_x, p_{n+1}).
  - After comparing all existing points with the new added point, the final *cloest_pair* should be correct.

Running time

In the above algorithm, it is pretty clear that we have two nested for loops. So the running time should be $O(n^2)$.

(b) First of all, let's assume that for all sorting algorithm used in the above algorithm has running time of $O(nlogn)$. We will divide the whole problem recursively, so the a and b parameters in the master theorem should be **2** and **2**, respectively. For the process that finding the closest pair in L and R should take $O(n)$ time since we will loop it for once. Then sorting the remaining points in the $x - d < x_i < x + d$ region by y coordinate takes again $O(nlogn)$ time. Finally, in the $x - d < x_i < x + d$ region, since we will only consider seven subsequence points in the list, the running time to find the closest point reduces from $O(n^2)$ to $O(n)$. Therefore, let T(n) be the total running time of the algorithm and we have:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + O(n) + O(nlogn) + O(n) \\ &= 2T(\frac{n}{2}) + O(nlogn) \end{aligned} \tag{11}$$

If we decompose the above T(n), we will have:

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + O(nlogn) \\
&= 4T(\frac{n}{4}) + 2O(\frac{n}{2}log\frac{n}{2}) + O(nlogn) \\
&= 8T(\frac{n}{8}) + 2O(\frac{n}{2}log\frac{n}{2}) + O(nlogn) \\
&= ..... \\
&= 2^kO(\frac{n}{2^k}log\frac{n}{2^k}) + ... + 2O(\frac{n}{2}log\frac{n}{2}) + O(nlogn) \\
&= O(nlog\frac{n}{2^k}) + ... + O(nlog\frac{n}{2}) + O(nlogn) \\
&= O\bigg(n\Big(log\frac{n}{2^k} + ... + log\frac{n}{2} + logn\Big)\bigg)
\end{aligned} \tag{12}$$

Since the recursion tree we used will divide the problem into two subproblems, so we will have 2^k subproblems in total and the depth of the tree should be $k = logn$. Let's substitute n by 2^k:

$$\begin{aligned}
T(n) &= O\bigg(n\Big(log\frac{n}{2^k} + ... + log\frac{n}{2} + logn\Big)\bigg) \\
&= O\bigg(n\Big(log1 + ... + log2^{k-1} + log2^k\Big)\bigg) \\
&= O\bigg(n\Big(0 + 1 + ... + k - 1 + k\Big)\bigg) \\
&= O\Big(\frac{k(k+1)}{2}n\Big)
\end{aligned} \tag{13}$$

Finally, substitute k by $logn$ and we have:

$$T(n) = O\Big(n(logn)^2\Big) \tag{14}$$