CS 161
Summer 2017

Problem Set 1
Due: July 7, 2017, 11:59 p.m.

---

1. **What do you want from this course?** (1 point) What skills do you hope to learn, what topics do you think we'll cover that will stick with you five, or ten years down the road? Write *at least three sentences* describing how you expect to benefit from taking this class. The reason for this question is twofold. First, it will help us understand what you want. Second, keep your answer to this question in mind throughout the quarter as motivation if the going gets tough!

2. **New friends.** (5 points) Each of n users spends some time on a social media site. For each $i = 1, \ldots, n$, user i enters the site at time a_i and leaves at time $b_i \geq a_i$. You are interested in the question: how many distinct pairs of users are ever on the site at the same time? (Here, the pair (i, j) is the same as the pair (j, i)).

   Example: Suppose there are 5 users with the following entering and leaving times:

User	Enter time	Leave time
1	1	4
2	2	5
3	7	8
4	9	10
5	6	10

   Then, the number of distinct pairs of users who are on the site at the same time is three: these pairs are $(1, 2)$, $(4, 5)$, $(3, 5)$. (Drawing the intervals on a number line may make this easier to see).

   (a) (1 point) Given input $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ as above in no particular order (i.e., not sorted in any way), describe a straightforward algorithm that takes $\Theta(n^2)$-time to compute the number of pairs of users who are ever on the site at the same time, and explain why it takes $\Theta(n^2)$-time.

   [We are expecting pseudocode and a brief justification for its runtime.]

   (b) (4 points) Give an $\Theta(n \log(n))$-time algorithm to do the same task and analyze its running time. (**Hint:** consider sorting relevant events by time).

   [We are expecting pseudocode and a brief justification for its runtime.]

   Solution

   (a) Our algorithm will run as follows:

   - Initialize variable *count* to 0.
   - For every user i, we check every other user $j \neq i$:
     If $a_i \leq a_j \leq b_i$ or $a_j \leq a_i \leq b_j$, then user i and user j are on the site at the same time. Increment *count* by 1.

- We return $count/2$, which counts the number of distinct pairs of users who are ever on the site at the same time. We divide *count* by 2 because we double counted each pair.

Running time analysis: For each user $i = 1, ..., n$, we iterate over all other users ($\Theta(n)$ of them) to check for the above inequality which can be done in constant time. Therefore, the algorithm runs in $\Theta(n^2)$.

(b) The key here is to realize that we can decouple the start and exit times from the user. Here is our algorithm:

- Initialize variable *count* and variable *usersOnsite* to 0. Note that at time t_i, *usersOnSite* is equal to the current number of users on the site.
- We produce a combined list l of entry and exit times. l has $2n$ tuples. The first element in each tuple is the entry/exit time and the second element is a binary indicating "entry" or "exit". Thus, user i can be split into $(a_i, \text{“}enter''\text{)}$ and $(b_i, \text{“}exit''\text{)}$.
- Next, we sort list l by the first element in each tuple using MergeSort. For each tuple p in the sorted list l, we check:
  If $p[1]$ is "enter":
  - If $usersOnSite \geq 1$, increment *count* by the value of *usersOnSite*.
  - Increment *usersOnSite* by 1.

  If $p[1]$ is "exit", decrement *usersOnSite* by 1.
- Return count.

Running time analysis: Initialize list l takes $\Theta(n)$ time, since we are iterating over all user entry and exit times, generating 2 tuples for each user. Sorting the list using MergeSort takes $\Theta(n \log(n))$ time. Iterating through the sorted list l takes $\Theta(n)$ time, since there are a total of $2n$ tuples and each iteration takes constant time to execute. Overall, the algorithm takes $\Theta(n \log(n))$ time.

3. **Proof of correctness.** (6 points) Consider the following algorithm that is supposed to sort a list of integers. Provide a proof that this algorithm is correct.

[We are expecting a rigorous proof.]

Algorithm 1: `sort`

```
Input: Unsorted list A = [a_1, ..., a_n] of n items
Output: Sorted list A' = [a'_1, ..., a'_n] of n items
for i = 0, i < n − 1, i++ do
    // Find the minimum element
    min_index = i
    for j = i + 1, j < n, j++ do
        if A[j] < A[min_index] then
            min_index = j
    // Swap the minimum element with the first element
    swap(A, i, min_index)
return A
```

Solution
We use two loop invariants:

Outer loop: At the beginning of the ith iteration, elements 1 through $i-1$ are sorted in increasing order, and all of the elements i through $A.length$ are at least as large as all of the elements in $1..i-1$.

Inner loop: At the beginning of the jth iteration, minIndex is the index of the smallest element in the range $[i, j-1]$.

Outer loop:

Initialization: At the beginning of the first iteration, there are no elements in the range $[1, 0]$, so they are vacuously sorted in increasing order. The second clause of the loop invariant is also vacuously true.

Maintenance: Suppose the invariant holds before iteration i. We show that it holds before iteration $i+1$. If the inner loop invariant holds, then at the end of the inner loop, minIndex is the index of the smallest element in the array range $[i, A.length]$. Then the Swap command ensures that the smallest element in the range $[i, A.length]$ gets placed in A[i]. By the previous invocation of the loop invariant, we know that $A[1] \leq A[2] \leq \cdots \leq A[i-1]$, and $A[i]$ is at least as large as $A[i-1]$. Furthermore, from what we are arguing now, we know $A[i]$ is at least as small as all the elements in $A[i+1], \ldots, A[A.length]$. This maintains the invariant for the next iteration.

Termination: At the beginning of the $(A.length+1)$st iteration, elements 1 through $A.length$ are sorted in increasing order, so the array is sorted.

Inner loop:

Initialization: At the beginning of iteration $i+1$, minIndex is i, which is the index of the smallest element in the range $[i, i]$.

Maintenance: Suppose the invariant holds prior to iteration j. We show that it holds prior to iteration $j+1$. At the beginning of the jth iteration, minIndex is the index of the smallest element in the range $[i, j-1]$. If $A[j] < A[minIndex]$, then j is the index of the smallest element in the range $[i, j]$, so setting minIndex = j is correct. If $A[j]$ is not less than $A[minIndex]$, then $minIndex$ is the index of the smallest element in the range $[i, j]$, so it is correct to leave minIndex alone. This maintains the invariant for the next iteration.

Termination: At the beginning of the $(A.length+1)$st iteration, minIndex is the index of the smallest element in the range $[i, A.length]$, which is the condition we needed for our proof of the outer loop invariant.

4. **n-naught not needed.** (3 points) Suppose that $T(n) = O(n^d)$, and that $T(n)$ is never equal to ∞. Prove rigorously that there exists a c so that $0 \leq T(n) \leq c \cdot n^d$ for all $n \geq 1$. That is, the definition of $O(\cdot)$ holds with $n_0 = 1$.

   [We are expecting a brief, but convincing proof.]

   Solution
   Suppose that $T(n) = O(n^d)$. This means that there exists a c and an n_0 so that $T(n) \leq cn^d$ for all $n \geq n_0$.

Let

$$c' = \max\left(c, \frac{T(1)}{1^d}, \frac{T(2)}{2^d}, \ldots, \frac{T(n_0-1)}{(n_0-1)^d}\right)$$

.

Note that the first element, c in the above expression is obtained the from the case where $n \geq n_0$. The rest represent the case where $n \in [1, n_0 - 1]$. Then,

$$T(n) \leq \begin{cases} T(n) & n < n_0 \\ cn^d & n \geq n_0 \end{cases} \leq \begin{cases} c'n^d & n < n_0 \\ c'n^d & n \geq n_0 \end{cases} \leq c'n^d$$

5. **Fun with recurrences.** (6 points)

Solve the following recurrence relations; i.e. express each one as $T(n) = O(f(n))$ for the tightest possible function $f(n)$, and give a short justification. Be aware that some parts might be slightly more involved than others. Unless otherwise stated, assume $T(1) = 1$.

[To see the level of detail expected, we have worked out the first one for you.]

(z) $T(n) = 6T(n/6) + 1$. We apply the master theorem with $a = b = 6$ and with $d = 0$. We have $a > b^d$, and so the running time is $O(n^{\log_6(6)}) = O(n)$.

(a) (0.5 points) $T(n) = 2T(n/2) + 3n$

(b) (0.5 points) $T(n) = 3T(n/4) + \sqrt{n}$

(c) (0.5 points) $T(n) = 7T(n/2) + \Theta(n^3)$

(d) (2 points) $T(n) = 4T(n/2) + n^2 \log n$

(e) (0.5 points) $T(n) = 2T(n/3) + n^c$, where $c \geq 1$ is a constant (that is, it doesn't depend on n).

(f) (2 points) $T(n) = 2T(\sqrt{n}) + 1$, where $T(2) = 1$

Solution

(a) $T(n) = O(n \log n)$, using the Master Theorem with $a = 2, b = 2, d = 1$.

(b) $T(n) = O(n^{\log_4 3})$, using the Master Theorem with $a = 3, b = 4, d = 1/2$. We have $a > b^d$, so the answer is $O(n^{\log_b(a)})$.

(c) $T(n) = O(n^3)$, using the Master Theorem with $a = 7, b = 2, d = 3$. (Notice that the $\Theta(n^3)$ expression is $O(n^3)$ as well, as per the statement in class.) Then $a < b^d$, so the running time is $O(n^d) = O(n^3)$.

(d) $T(n) = O(n^2 \log^2(n))$. To get an idea, we might start working things out:

$$\begin{aligned}
T(n) &= 4T(n/2) + n^2 \log(n) \\
&= 16T(n/4) + n^2 \log(n/2) + n^2 \log(n) \\
&= \cdots \\
&= 2^{2k} T(n/2^k) + n^2(\log(n/2^{k-1}) + ... + \log(n/2) + \log(n)) \text{ where } k = \log_2(n) \\
&= n^2 + n^2 \log\left(\frac{n^k}{2^{k(k-1)/2}}\right) \\
&\leq n^2 + n^2 \log(n^k) \\
&= n^2 + n^2 \log^2(n) \\
&= O(n^2 \log^2(n)).
\end{aligned}$$

Formally (a formal proof is not required for credit, but here it is), we may proceed by induction. We use as an inductive hypothesis that for all $t > 0$.

$$T(n) = 4^t \cdot T(n/2^t) + n^2(\log(n) + \log(n/2) + \log(n/4) + \cdots + \log(n/2^{t-1})).$$

For the base case, notice that when $t = 1$ this is just

$$T(n) = 4 \cdot T(n/2) + n^2 \log(n),$$

which is the provided recurrence relation. For the inductive hypothesis, we observe that

$$T(n) = 4^{t-1} \cdot T(n/2^{t-1}) + n^2(\log(n) + \cdots + \log(n/2^{t-2})),$$

and plugging in the recurrence relation we find

$$T(n) = 4^{t-1} \cdot \left(4T(n/2^t) + (n/2^{t-1})^2 \log(n/2^{t-1})\right) + n^2(\log(n) + \cdots \log(n/2^{t-2})).$$

Rearranging establishes the inductive hypothesis for the next round. Finally, we plug in $t = \log_2(n)$ to see that

$$T(n) = 4^{\log_2(n)} T(1) + n^2(\log(n) + \cdots + \log(2)) = O(n^2 \log^2(n)),$$

as desired.

(e) The Master Theorem implies that

$$T(n) = \begin{cases} O(n^{\log_3 2} \log n) & \text{if } 2 = 3^c \\ O(n^c) & \text{if } 2 < 3^c \\ O(n^{\log_3 2}) & \text{if } 2 > 3^c \end{cases}$$

However, since $c \geq 1$, we always have $2 < 3^c$, so the answer is $O(n^c)$.

(f) To get intuition, we begin by iteratively plugging in the recurrence relation:

$$\begin{aligned} T(n) &= 2T(\sqrt{n}) + 1 \\ &= 2(2T(n^{1/4}) + 1) + 1 \\ &= 4T(n^{1/4}) + 2 + 1 \\ &= 4(2T(n^{1/8}) + 1) + 2 + 1 \\ &= 8T(n^{1/8}) + 4 + 2 + 1 \end{aligned}$$

Carrying on this way, we see that for $t \geq \log\log(n)$,

$$T(n) = 2^t T(n^{1/2^t}) + \sum_{i=0}^{t-1} 2^i.$$

(To formally prove this, we may do a proof by induction; this is not required for credit for this problem.) Now, for $t = \log\log(n)$, we have $n^{1/2^t} = 2$, and so plugging in $t = \log\log(n)$ we see

$$T(n) = 2^{\log\log(n)} T(2) + \sum_{i=0}^{\log\log(n)-1} 2^{\log\log(n)} = O(\log(n)).$$

6. **Why groups of 5?** (6 points) In the `select_k` algorithm from class, in order to find a pivot, we decompose our array of length n into $m = \lceil n/5 \rceil$ blocks of at most length 5. Why 5? In this question, we explore this decision.

   (a) (0.5 points) Prove that the recursive call inside of `select_k_with_groups_of_3` is on an array of size at most $2n/3 + 2$. Unlike in lecture, here include the values within the same group as the median of medians as elements that are guaranteed to be greater or less than it.

   [We are expecting a brief, but convincing algebraic proof.]

   (b) (0.5 points) Write a recurrence relation for the runtime of `select_k_with_groups_of_3`.

   [We are expecting a one-line answer with your recurrence relation.]

   (c) (2 points) Is `select_k_with_groups_of_3` $O(n)$? Justify your answer.

   [We are expecting a convincing argument, such as analyzing a tree, unraveling the recurrence relation to get a summation, or attempting the substitution method.]

   (d) (0.5 points) Prove that the recursive call inside of `select_k_with_groups_of_7` is on an array of size at most $5n/7 + 4$. Make the same assumptions as in part (a).

   [We are expecting the same as part (a).]

   (e) (0.5 points) Write a recurrence relation for the runtime of `select_k_with_groups_of_7`.

   [We are expecting the same as part (b).]

   (f) (2 points) Is `select_k_with_groups_of_7` $O(n)$? Justify your answer.

   [We are expecting the same as part (c).]

Solution

(a) Let $g = \lceil n/3 \rceil$ represent the number of groups.

$$\begin{aligned}
|A| &\leq n - 1 - (2 \cdot (\lceil g/2 \rceil - 2) + 1) \\
&= n + 2 - 2 \cdot \lceil g/2 \rceil \\
&\leq n + 2 - 2g/2 \\
&= n + 2 - \lceil n/3 \rceil \\
&\leq n + 2 - n/3 \\
&= 2n/3 + 2
\end{aligned}$$

(b) $T(n) \leq T(n/3) + T(2n/3 + 2) + \Theta(n)$

(c) No, it is not $O(n)$. We will imagine drawing a tree.

At the top level of our tree, we have only one problem of size n, and we do $\Theta(n)$ work. Then, at the next level, we do $\Theta(n/3) + \Theta(2n/3) = \Theta(n)$ work. In fact, if we look at the two children of a particular node, we notice that the amount of work done within each of those two children is exactly equal to the amount of work done in the parent! This is true all the way down the tree, so we can see that there is $\Theta(n)$ work done at every level. Moreover, since we either multiply the problem by $1/3$ or by $2/3$ each time, we can see that there are $\Theta(\log n)$ levels. (Depending what route you take through the tree, it could range from $\log_3(n) + 1$ at shortest to $\log_{3/2}(n) + 1$ at longest, but the difference between $\log_3(x)$ and $\log(3/2)(x)$ is just a multiplicative constant, which we ignor in the $\Theta()$ notation. Then overall we can see that the runtime is $\Theta(n \log n)$, so it is not $O(n)$.

(d) Let $g = \lceil n/7 \rceil$ represent the number of groups.

$$\begin{aligned}
|A| &\leq n - 1 - (4 \cdot (\lceil g/2 \rceil - 2) + 3) \\
&= n + 4 - 4 \cdot \lceil g/2 \rceil \\
&\leq n + 4 - 4g/2 \\
&= n + 4 - 2\lceil n/7 \rceil \\
&\leq n + 4 - 2n/7 \\
&= 5n/7 + 4
\end{aligned}$$

(e) $T(n) \leq T(n/7) + T(5n/7 + 4) + \Theta(n)$

(f) Yes, it is $O(n)$. We will prove via the substitution method. We initially guess that $T(n) \leq O(n)$.

Base Case: we must show that you can pick some d such that $T(n_0) \leq dn_0$. Thus, we pick $n_0 = 1$ and using the standard assumption $T(1) = 1 \leq d$.

Inductive Step: For the inductive hypothesis, assume that the guess is correct for $n < k$ and we prove our guess for k. Thus, consider d such that for all $n_0 \leq n <$

$k, T(n) \leq dn$. To prove for $n = k$, we solve the following equation.

$$\begin{aligned} T(k) &\leq ck + T(k/7) + T(5k/7 + 4) \\ &\leq ck + d(k/7) + d(5k/7 + 4) \\ &= ck + 6dk/7 + 4d \leq dk \\ 7ck &\leq dk - 28d \\ 7c\frac{k}{k - 28} &\leq d \end{aligned}$$

If we let $d \geq 203c$, we can see that the inductive step holds, since left side is ≤ 0 for $k \leq 28$ and decreases as $k \geq 29$. Proof complete.

7. **k-order statistic of compressed dataset.** (9 points) Suppose you have a collection of data points where there are a huge number of copies of each data point. For example, you might have a data set:

```
A, B, C, B, A, C, B, A,
A, B, A, C, A, A, C, B,
D, A, A, A, D, B, E, A,
A, F, A, B, A, A, A, B
```

Here, there are 32 data points, but there are only six distinct values. Rather than storing the data as above, suppose that you store the data as a list of tuples of the form $(value, frequency)$. For example, the above data might be stored as

```
(A, 16), (B, 8), (C, 4), (D, 2), (E, 1), (F, 1)
```

That is, there are 16 `A`'s, eight `B`'s, four `C`'s, two `D`'s, one `E`, and one `F`.

Suppose that you have a data set represented as m of these tuples, where the tuples are stored in no particular order. Design an $O(m)$-time algorithm for computing the kth order statistic of the data set. Note that m is the number of tuples rather than the total number of elements n in the uncompressed data set, so your algorithm's runtime should not depend asymptotically on n.

(a) (4 points) Describe your algorithm.
[We are expecting pseudocode.]

(b) (4 points) Provide a proof that this algorithm is correct.
[We are expecting a rigorous proof.]

(c) (1 point) Prove that your algorithm runs in $O(m)$.
[We are expecting a brief justification.]

Solution

(a) **Algorithm**

Use the linear-time selection algorithm to find the median of all the tuples (the element at position $\lfloor m/2 \rfloor$) and rearrange the tuples to place tuples with lower values to the left and higher values to the right. Let f be the frequency of the median element. Compute the sum s of the frequencies of all tuples to the left of the median and if $k < s$ and recursively apply this algorithm to the left half of the array to find element k. Otherwise, if $k < s + f$, return the key of the median element. Otherwise, recursively apply this algorithm to find order statistic $k - s - f$ of the elements in the right subarray.

(b) **Correctness Proof**

Theorem: The algorithm returns the kth order statistic in the decompressed data set.

Proof: By induction. Assume that for some $m \geq 1$ that the algorithm is correct for all m' in the range $1 \leq m' < m$. After running the linear-time selection algorithm to put tuple with the median element in place (with frequency f), the data are ordered so that all tuples with elements that would appear before the median element in sorted order appear before the tuple containing the median element, the tuple containing the median element is in the right place, and all tuples containing elements greater than the median element appear to the right of the median. Consider the sequence of elements that would be formed if we decompressed all of the data based on the order the tuples currently are placed in. There will be s elements before the first copy of the median element (which will be at position s, zero-indexed), and the first element greater than the median element will be at position $s + f$. Because of the relative ordering of the tuples, we know that the elements in the range $[0, s)$ are a permutation of the elements that would be in that range were the data to be sorted, as are the elements in the ranges $[s, s + f)$ and $[s + f, n)$. We consider three cases:

Case 1: $k < s$. This means that the kth order statistic is the element that would appear at position k in the elements in the range $[0, s)$. These elements are represented by the tuples that appear to the left of the median tuple. Therefore, when the algorithm returns the result of the recursive call to select the kth order statistic from the subarray appearing before the median (which is on an array of a smaller size and thus by the IH returns the kth order statistic of those tuples), it returns the kth order statistic overall.

Case 2: $s \leq k < s + f$. All elements in the range $[s, s + f)$ are equal to the median element, so the kth order statistic in this case is the median. This is what the algorithm returns.

Case 3: $s + f \leq k$. Then the kth order statistic would be the element at position $k - s - f$ in the subarray consisting of all elements greater than the median element. Our algorithm then recursively finds the $(k - s - f)$th order statistic of the tuples representing this subarray. Since this consists of fewer than m tuples, by the IH this recursive call returns the $(k - s - f)$th order statistic of this data set, which is the kth order statistic of the overall array. Thus in all cases the algorithm returns the kth order statistic, completing the induction.

(c) Runtime: The runtime for this algorithm is given by the following recurrence relation:

$$T(1) = \Theta(1)$$
$$T(m) \leq T(\lceil m/2 \rceil) + \Theta(m)$$

By the Master Theorem, the runtime is therefore $O(m)$.

8. **How is the course so far?** (0.5 points extra credit)

Please complete the poll at `https://goo.gl/forms/GzB9KO7j2NyddV3x2`.