CS 457, Data Structures and Algorithms I
Third Problem Set Solutions

1. Prove tight **worst-case** asymptotic upper bounds for the following recurrence equation that depends on a variable $q \in [0, n/4]$. Note that you need to prove an upper bound that is true for every value of $q \in [0, n/4]$ and a matching lower bound for a specific value of $q \in [0, n/4]$ of your choosing. Do not assume that a specific q yields the worst case input; instead, formally identify the q which maximizes the running time.

$$T(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ T(n-2q-1) + T(3q/2) + T(q/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

Solution:

We begin by guessing that $T(n) \in \Theta(n)$, i.e., that there exist constants c_1 and c_2 such that $c_1 n \leq T(n) \leq c_2 n$ for $n > n_0$. We first prove the upper bound and then the lower bound.

Given our guess, we assume that $T(n') \leq c_2 n'$ for all $n' < n$. This means that

$$T(n) = T(n-2q-1) + T(3q/2) + T(q/2) + \Theta(1) \leq c_2(n-2q-1) + c_2(3q/2) + c_2(q/2) + \Theta(1).$$

At this point, we observe that the q terms in the above equations cancel, so the function is constant with respect to q. Therefore, any q maximizes the function.

Using the fact that any function in $\Theta(1)$ is at most some constant a, this becomes $T(n) \leq c_2 n - c_2 + a$. If we let $c_2 \geq a$, then this yields $T(n) \leq c_2 n$, which concludes the proof of the upper bound.

For the lower bound, we assume that $T(n') \geq c_1 n'$ for all $n' < n$ and select $q = 0$. This means that

$$T(n) = T(n-2q-1) + T(3q/2) + T(q/2) + \Theta(1) \geq c_1(n-1) + \Theta(1).$$

Using the fact that any function in $\Theta(1)$ is at least some constant b, this becomes $T(n) \geq c_1 n - c_1 + b$. If we let $c_1 \leq b$, then this yields $T(n) \geq c_1 n$, which concludes the proof of the lower bound.

2. (24 pts) Given an array S of n distinct numbers provide $O(n)$-time algorithms for the following:

   - Given two integers $k, \ell \in \{1, \ldots, n\}$ such that $k \leq \ell$, find all the ith order statistics of S for *every* $i \in \{k, \ldots, \ell\}$.

     Solution: First, we use linear time selection to find the kth order statistic of set S. Call this value v. This takes $O(n)$ worst-case running time. Then we use linear time selection to find the ℓth order statistic of set S. Call this value w. This takes $O(n)$ worst-case running time. Then we pass through the array returning all values lying between v and w. This final pass also only takes $O(n)$ worst-case running time since we make two comparisons for every element in S.
   - Given some integer $k \in \{1, \ldots, n\}$, find the k numbers in S whose *values* are closest to that of the median of S.

Solution: First, we use linear time selection to find the median q of set S. This takes $O(n)$ worst-case running time. Then we create a distance array by calculating the absolute value of differences between each element x in the set and median, i.e., $|x - q|$. Constructing this distance array also takes $O(n)$ time. For the new distance array, we find the k-th order statistic within the set of distances, by once again using the linear time selection algorithm. Finally, passing through the whole set S, select only the elements that have distance lower than or equal to the one defined by the k-th order statistic (only including k such elements as there may be two elements with equal distance from, one larger and one smaller than, the median). The running time of this step is also $O(n)$, therefore the algorithm takes $O(n)$ time in total.

3. Consider the following silly randomized variant of binary search. You are given a sorted array A of n integers and the integer v that you are searching for is chosen uniformly at random from A. Then, instead of comparing v to the value in the middle of the array, the randomized binary search variant chooses a random number r from 1 to n and it compares v with $A[r]$. Depending on whether v is larger or smaller, this process is repeated recursively on the left sub-array or the right sub-array, until the location of v is found. Prove a tight bound on the expected running time of this algorithm.

Solution:

For the analysis of the expected running time of this algorithm, we will be using the indicator variable $X_r = \mathbb{I}\{\text{the number between 1 and } n \text{ chosen by the algorithm is } r\}$. Using the choice of r, the algorithm partitions the initial array into two sub-arrays: the length of the left sub-array is $r-1$ and the length of the right sub-array is $n-r$. Since the integer v was chosen uniformly at random from A, it lies in the left sub-array with probability $\frac{r-1}{n}$, in the right sub-array with probability $\frac{n-r}{n}$, and there is also a probability $\frac{1}{n}$ that v is in fact the element in $A[r]$. Since the cost of comparing the value of v with that of $A[r]$ is constant, we can express the running time of the algorithm using the following recurrence equation:

$$T(n) = \sum_{r=1}^{n} X_r \left(\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right)$$

Therefore, the expected running time of the algorithm can be expressed as

$$\begin{aligned}
\mathbb{E}[T(n)] =& \mathbb{E}\left[\sum_{r=1}^{n} X_r \left(\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right) \right] \\
=& \sum_{r=1}^{n} \mathbb{E}\left[X_r \left(\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right) \right] \\
=& \sum_{r=1}^{n} \mathbb{E}\left[X_r\right] \mathbb{E}\left[\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right] \\
=& \sum_{r=1}^{n} \frac{1}{n} \mathbb{E}\left[\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right] \\
=& \sum_{r=1}^{n} \frac{(r-1)\mathbb{E}[T(r-1)] + (n-r)\mathbb{E}[T(n-r)]}{n^2} + \sum_{r=1}^{n} \frac{1}{n} \Theta(1) \\
=& \sum_{r=1}^{n} \frac{2(r-1)\mathbb{E}[T(r-1)]}{n^2} + \Theta(1). \qquad (1)
\end{aligned}$$

This sequence of equations, except the last one, is due to a repeated use of linearity of expectation and of equation (C.24) from your textbook. For a very similar sequence of arguments, see the analysis of RANDOMIZED-SELECT at the top of Page 218. The last equation uses the fact that $\sum_{r=1}^{n}(r-1)\mathbb{E}[T(r-1)]$ is actually equal to $\sum_{r=1}^{n}(n-r)\mathbb{E}[T(n-r)]$.

We now guess that $\mathbb{E}[T(n)] \in O(\log n)$ and we prove it using the substitution method (we present only the upper bound as the lower bound would follow similar structure). In particular, we guess that there exist positive constants c, and n_0 such that $\mathbb{E}[T(n)] \leq c \log n$ for $n \geq n_0$. For the base case, we consider the case $n = 2$ for which $\log 2 = 1$. Since the running time of the algorithm is $\Theta(1)$ for this case, there definitely exists a constant c such that $\mathbb{E}[T(3)] \leq c$. In the rest of the proof, we also show that the inductive step holds.

To prove the upper bound, we assume that $\mathbb{E}[T(n')] \leq c \log n'$ for some constant c and every $n' < n$, which implies that

$$\begin{aligned}
\sum_{r=1}^{n} \frac{2(r-1)\mathbb{E}[T(r-1)]}{n^2} &= \sum_{r=1}^{2} \frac{2(r-1)\mathbb{E}[T(r-1)]}{n^2} + \sum_{r=3}^{n} \frac{2(r-1)\mathbb{E}[T(r-1)]}{n^2} \\
&\leq \frac{2\mathbb{E}[T(2)]}{n^2} + \sum_{r=3}^{n} \frac{2(r-1)c\log(r-1)}{n^2} \\
&\leq \frac{2\mathbb{E}[T(2)]}{n^2} + \sum_{r=3}^{\lceil n/2 \rceil} \frac{2(r-1)c\log(r-1)}{n^2} + \sum_{r=\lceil n/2 \rceil+1}^{n} \frac{2(r-1)c\log(r-1)}{n^2} \\
&\leq \frac{2c}{n^2} + \sum_{r=3}^{\lceil n/2 \rceil} \frac{2(r-1)c\log(n/2-1)}{n^2} + \sum_{r=\lceil n/2 \rceil+1}^{n} \frac{2(r-1)c\log(n-1)}{n^2} \\
&\leq \frac{2c}{n^2} + \frac{n^2 c\log(n/2-1)}{4n^2} + \frac{3n^2 c\log(n-1)}{4n^2} \\
&\leq \frac{2c}{n^2} + \frac{n^2 c\log n - n^2 c\log 2}{4n^2} + \frac{3n^2 c\log n}{4n^2} \\
&\leq c\log n - c/4 + c/(2n^2) \\
&\leq c\log n - c/8.
\end{aligned}$$

The first inequality is just by substitution of the assumption that $\mathbb{E}[T(r-1)] \leq c \log r - 1$. The second one splits the summation in order to obtain an upper bound (see Page 1152 of your textbook). The third inequality uses the fact that $\mathbb{E}[T(2)] \leq \mathbb{E}[T(3)] \leq c$, which we mentioned in our base case, as well as the fact that $\log(r-1)$ is an increasing function, so we can replace r with the largest value that it takes in the corresponding summation. We then use the fact that $\sum_{r=2}^{n/2}(r-1) \leq n^2/8$ and $\sum_{r=n/2}^{n}(r-1) \leq 3n^2/8$. Finally, the last inequality uses the fact that, for $n \geq 3$, we get $c/(2n^2) - c/4 \leq c/16 - c/4 < c/8$.

As a result, using Inequality (1) we conclude that

$$\mathbb{E}[T(n)] \leq c\log n - c/8 + \Theta(1) \leq c\log n - c/8 + a,$$

for some constant a. Choosing a constant $c \geq 8a$ therefore would ensure that $\mathbb{E}[T(n)] \leq c \log n$.

4. You are given a set S of n integers, as well as one more integer v. Design an algorithm that determines whether or not there exist two distinct elements $x, y \in S$ such that $x + y = v$. Your algorithm should run in time $O(n \log n)$, and it should return (x, y) if such elements exist and (NIL, NIL) otherwise. Prove the worst case running time bound and the correctness of the algorithm.

Solution:

There are several algorithms that solve this problem within the desired running time bound. One that is simple to write and analyze begins by sorting the elements into a list L in $O(n \log n)$ time using merge-sort or one of the other sorting algorithms with the same worst-case running time bound. Then, it considers every element i from 1 to n and for each one of them it uses binary search for the value

$v - L[i]$ in order to search whether the "pair" of $L[i]$ also exists, ensuring that if $L[i]$ is 1/2 of the target number that we ignore it (since a set has no duplicate elements there is no pair for it) to not return a false positive. The worst-case time bound is satisfied since the algorithm calls binary search $O(n)$ times and each call needs $O(\log n)$ time. Therefore, the total time of both sorting and searching is $O(n \log n)$.

What we provide below is a more interesting algorithm that instead completes the searching in linear time instead of $O(n \log n)$. We begin by presenting the algorithm's pseudocode and then provide the analysis.

Pseudocode:

```
(a) SUM-EXISTS(S, v){
(b)     L = MAKE-LIST(S)          //Creates a list from a set in O(n) time
(c)     L = MERGE-SORT(L, 1, L.length)          //Merge sort runs in O(n log(n) time
(d)     i = 1
(e)     j = L.length
(f)     while i < j do
(g)         if L[i] + L[j] = v
(h)             return (L[i],L[j])
(i)         else if L[i] + L[j] > v
(j)             j = j - 1
(k)         else
(l)             i = i + 1
(m)     return (NIL, NIL)
(n) }
```

The algorithm sorts first, but then does something more sophisticated than repeated binary search. Let $A[l]$ be the left element we are considering and $A[r]$ be the right element. If $A[l] + A[r] = v$, we are done. Otherwise, if $A[l] + A[r] > v$ then nothing further to the right of $A[l]$ can be summed with $A[r]$ for a value of v (since our array is sorted). Therefore, since we have already examined everything to the right of $A[r]$ and the left of $A[l]$, the only option is to examine things to the right of $A[r]$ (i.e., $A[r-1]$). Similarly, if $A[l] + A[r] < v$, we must examine things to the left of $A[l]$ (i.e., $A[l+1]$).

Worst-Case Bound: Since we use the merge-sort algorithm for sorting, its worst-case running time is $O(n \log n)$. Also, while the loop keeps being executed, at each iteration either i is incremented by one or j is decremented by one. Therefore, this loop cannot be executed more than $O(n)$ times, and each iteration takes $O(1)$ time. As a result, the worst-case running time of this algorithm is $O(n \log n) + O(n) = O(n \log n)$.

5. (20 pts) Suppose that you are given a sorted array A of *distinct* integers $\{a_1, a_2, \ldots, a_n\}$, drawn from 1 to m, where $m > n$.

   a) Give an $O(\log n)$ algorithm to find an integer from $[1, m]$ that is not present in A. For full credit, find the smallest such integer.

   b) Formally explain why your algorithm runs in $O(\log n)$ time.

Solution: If $A[1] > 1$, then the smallest integer missing is 1. Also, if $A[n] = n$, then the smallest integer missing is $n + 1$. Otherwise, finding the smallest integer from $[1, m]$ that is not present in A amounts to finding the smallest integer $q \in [1, n]$ such that $A[q] \neq q$. Since the array is sorted, anything

to the left of some index j such that $A[j] = j$ must also equal its index. Therefore, we are not missing any number smaller than j.

We propose algorithm FindMissing(ℓ, r), where ℓ and r are a left and a right pointer for A, i.e., $\ell \leq r$. The algorithm begins by checking whether $\ell = r$, in which case it returns ℓ. Then, the algorithm compares the value of the middle pointer $q = \lfloor(\ell + r)/2\rfloor$ to the value of the corresponding entry in A, i.e., it compares $A(q)$ to q. If $A(q) > q$, then the missing integer has to be on the left half, so the algorithm recursively calls FindMissing(ℓ, q). Otherwise, if $A(q) \leq q$, the algorithm calls FindMissing$(q + 1, r)$.

In order to get the answer to this problem, we can then just call FindMissing$(1, n)$, and the algorithm will run its recursive calls until it reaches a point where the left and the right pointer both point to the missing entry.

Complexity: To verify that the complexity of the algorithm is $O(\lg n)$ one can use the exact same arguments used in the analysis of binary search. Observe that at every recursive call the input array size is halved since we only are examining the left or right subarray of our visited index. Further, the function has a termination condition when the array has size 1. We may use these observations to define a recurrence equation which will be the same as the recurrence for binary search. In particular, the worst case running time of the algorithm can be expressed as $T(n) = T(n/2)+\Theta(1)$, whose solution is $O(\lg n)$, as can be verified using the master theorem.