due Sunday, 24 February 2019, at 11:00 PM

Note: If you choose to work with a partner on this homework, you need to register by 5:00 PM on January 18. Instructions for doing so will be given later, but you will need to follow them exactly. You are allowed to work with a partner in a different section.

When you submit your assignment solutions please take the time to indicate the starting page of each answer. To make life easy for the TA's put the answer to each problem on a separate page (you may put multiple parts of the same problem on one page)

*If you are working with a partner, please submit **one solution only**. GradeScope allows you to specific your partner after a submission. Moodle does not, but we will grade **one solution per team only.***

Submissions for the written part of the homework, including your analysis of the programs should be submitted to GradeScope (these will be set up as two distinct assignments – `Homework 2` and `Program 2 Analysis`). The source files for your programs should be submitted to `Assignment 2` in Moodle.

1. [9 points: 3 for (a), 4 for (b), and 2 for (c)] *Purpose: Understanding Heapsort and lower bounds on algorithms.*

   Consider the special case where Heapsort is used to sort 0's and 1's. Because Heapsort is comparison-based, you should assume that Heapsort does not take any special advantage of the values of the keys. Let k be the number of 1's in the array. All bounds that follow are expressed as functions of *both* n, the total number of elements, and k, the number of 1's. Assume $k < n/2$. Recall that the MakeHeap phase takes linear time, so no need to say anything about it in your proof.

   (a) Prove that the worst case number of key comparisons in the situation described above is $O(n + k \lg n))$. This is linear unless $k \in \omega(n/\lg n)$.

   Solution: During the first k iterations of the sorting phase, either a 1 or a 0 will be placed at the top of the heap after a Remove-Max. If it's a 1, there will be only two comparisons: it will stay at the top. If it's a 0, the 0 will, in the worst case, reach the bottom of the heap, which has height at most $\lg n$.

   After these k iterations, each iteration will take constant time – all elements are 0's.

   (b) Show that the above bound is 'tight' in the sense that it is also a Ω bound for the algorithm. Recall that to prove a worst case lower bound of $\Omega(g(n))$ for an algorithm we need to show that there exists $c, n_0 > 0$, so that for each $n \geq n_0$, there exists an input of size n that causes the algorithm to take time (or, in this case, number of comparisons), $\geq cg(n)$. Where a time bound has two parameters, the argument needs to work for any valid combination of the two.

   Solution: Consider an initial array of length n and let the first k array entries be 1's. The array will not change during the Make-Heap phase. Since $k < n/2$, the leaves will all be 0's. During the sorting phase, each leaf in turn will traverse a path of length at least $h - 2$, where h is the height of the heap, until it gets past the 1's. Since $h \in \Omega(\lg n)$ this proves $\Omega(n + k \lg n)$.

   (c) Prove that the number of key comparisons depends on the original position of the 1's in the array? This is not about the worst case or an asymptotic bound. The question refers to how the exact number of comparisons depends (or doesn't) on the positions of the 1's. Since your proof will require an example with duplicate keys, you should use key,value pairs[1] to distinguish between elements.

[1] See the programming assignment below.

Solution: Consider `[1, 1, 0, 0, 0]` versus `[1, 0, 1, 0, 0]`. MAKE-HEAP takes the same number of comparisons in both cases and the array does not change in either. The difference is in the first HEAPIFY of the sorting phase. Using `|` to indicate the border between heap and sorted array, the sequence of steps for each array is ...

```
[1(a), 1(b), 0(a), 0(b), 0(c)] -> [0(c), 1(b), 0(a), 0(b) | 1(a)]
-> [1(b), 0(c), 0(a), 0(b) | 1(a)]
```

Here there are three comparisons, two for the first swap of `0(c)` with `1(b)` and one to compare `0(c)` with `0(b)`: position 2 of the array has only one child at position 4.

```
[1(a), 0(a), 1(b), 0(b), 0(c)] -> [0(c), 0(a), 1(b), 0(b) | 1(a)]
-> [1(b), 0(a), 0(c), 0(b) | 1(a)]
```

Here there are only two comparisons; once `0(c)` reaches position 3, there are no children.

2. [10 points: 2 for (a), 3 for (b), and 5 for (c)] *Purpose: understanding sorting algorithms and the sorting lower bound.* Problem 8-4 on pages 206–207 (179–180 in 2/e).

Solution: The word "comparison" below refers to the pouring operation that determines whether the blue jug has the less, the same, or greater capacity than the red one.

(a) Compare each blue jug in turn to every previously unmatched red jug. Number of comparison is $n + (n-1) + \ldots + 2$ which is $\Theta(n^2)$.

(b) Note that if we sort the red jugs in advance, any algorithm for this problem will end up sorting the blue jugs. This means that the problem is at least as hard as sorting the blue jugs. The only difference here is that comparisons are ternary (the process of filling jugs can determine whether the blue one has smaller, equal, or greater capacity to the red one) instead of binary. The decision tree therefore has height at least $\log_3 n!$ which is still $\Omega(n \lg n)$.

(c) The algorithm is very similar to Quicksort. The recursion stops when there is exactly one red and one blue jug. Every sub-instance of the problem will have the same properties as the original: an equal number of red and blue jugs for which a matching exists. All that remains is how to do the partitioning:

1. Pick a random red jug r and compare it to each blue jug. The result will be a blue jug b that matches r and two sets of blue jugs B_L and B_G, whose capacities are $< r$ and $> r$, respectively.

2. Now compare b with every red jug (except r). The result is two sets of red jugs, R_L and R_G, with capacities are $< b$ and $> b$, respectively.

3. The two sub-instances for which the algorithm is called recursively are (B_L, R_L) and (B_G, R_G).

The analysis is also very similar to that of Quicksort. Call a pair (b, r) a *pivot pair* if it takes the role of b and r in the partition procedure described above. Disregard any comparison that involves two jugs of equal capacity – there are at most n of those. Let (b_i, r_i) represent the i-th pair in increasing order of capacity and consider the pairs $(b_i, r_i), \ldots, (b_j, r_j)$ where $j > i$.

Observe that b_i is compared with r_j and b_j is compared with r_i exactly when one of (b_i, r_i) or (b_j, r_j) is a pivot pair. The rest of the argument is identical to that for Quicksort.

3. [6 points, 3 points each part] *Understanding the analysis of linear time selection.* Exercise **9.3-1** on page 223 (page 192 in 2/e).

Solution:

Groups of 7

Given the 5-step algorithm provided in the textbook, we now discuss the situation when the inputs are divided into groups of 7.

At least half of the medians of each group are greater than or equal to the median-of-medians x found in step 2. That means at least half of the $\lceil \frac{n}{7} \rceil$ groups contribute 4 elements that are greater than x, except for the one that has fewer than 7 elements and the one contains x itself. Then we can conclude that the number of elements greater than x is at least

$$4(\lceil \frac{1}{2} \lceil \frac{n}{7} \rceil \rceil - 2) \geq \frac{2n}{7} - 8$$

Similarly, the number of elements that are less than x is also at least $\frac{2n}{7} - 8$. Thus SELECT is called recursively on at most $\frac{5n}{7} + 8$ elements in step 5.

Steps 1, 2, and 4 take $O(n)$ time. Step 3 takes $T(\lceil \frac{n}{7} \rceil)$ time, and step 5 takes time at most $T(\frac{5n}{7} + 8)$. We can then get a recurrence of the form:

$$T(n) = \begin{cases} c_0 & \text{when } n < n_0 \\ T(n) = T(\lceil \frac{n}{7} \rceil) + T(\frac{5n}{7} + 8) + an & \text{when } n \geq n_0. \end{cases}$$

Here an is the total time for steps 1, 2, and 4.

We now prove by induction that $T(n) \leq cn$ for some constant c. Let an be the total time for steps 1, 2, and 4. For the basis we need to have $c \geq c_0$.

$$\begin{aligned} T(n) &\leq c\lceil \frac{n}{7} \rceil + c(\frac{5n}{7} + 8) + an \\ &\leq \frac{cn}{7} + c + \frac{5cn}{7} + 8c + an \\ &= cn + (-\frac{cn}{7} + 9c + an) \end{aligned}$$

This is at most cn if $-\frac{cn}{7} + 9c + an \leq 0$, which means $c \geq 7an/(n - 63)$ when $n > 63$. We can choose, for example, $n_0 = 70$, so that when $n > n_0$, $n/(n - 63) \leq 9$. So choose $c \geq 63a$ to satisfy $T(n) \leq cn$ when $n > 70$.

So the algorithm will work in linear time when the inputs elements are divided into groups of 7.

Groups of 3

When we have groups of 3 there is a time bound of $\Omega(n \lg n)$. Even under ideal circumstances, an odd number of groups of 3, the number of elements greater than x (elements we get to throw out) is at most $2\lceil n/6 \rceil$, or, to overestimate, $n/3 + 2$. This means a recursive call would involve at least $2n/3 - 2$ elements. A recurrence for this is:

$$T(n) = \begin{cases} c_0 & \text{when } n < n_0 \\ T(n) = T(n/3) + T(2n/3 - 2) + an & \text{when } n \geq n_0. \end{cases}$$

We now prove by induction that $T(n) \geq cn \lg n$ for some constant c. Here the basis is trivial.

$$\begin{aligned} T(n) &= T(n/3) + T(2n/3 - 2) + an \\ &\geq cn/3(\lg n - \lg 3) + 2cn/3(1 + \lg n - \lg 3) + an \\ &= cn \lg n + 2cn/3 - n \lg 3 + an \\ &\geq cn \lg n \text{ unless } a + 2c/3 < \lg 3 \end{aligned}$$

It's pretty clear that, if we're counting comparisons, a is at least 3 (it takes 3 comparisons to find the median of a group of three elements), so the "unless" part is not an issue.