1. [worth 6 points] *understanding Heapsort and algorithm lower bounds*

Give (and prove) a Θ bound for the *average* number of comparisons used by Heapsort on an array of n elements. As with the average case analysis of Quicksort, all permutations are equally likely.

Actually the assumption about permutations being equally likely is not relevant. You will end up showing that, as long as all keys are distinct initially, even the best case number of comparisons for heapsort is $\Omega(n \lg n)$ and the average case can obviously be no better than the best case. Oddly, converting the keys to 0's and 1's, which are far from distinct, makes the argument much easier.

The gist of your argument, if you want to pursue this the way I did it, is as follows. Argue that, in any configuration, not too many 0's can be close to the bottom, i.e., there must be enough 1's close to the bottom to force enough 0's to trickle most of the way down when they are swapped into the root position.

We can ignore comparisons that take place in the initial building of the heap in getting a lower bound since this can be accomplished in $O(n)$ comparisons.

Suppose that we are sorting arbitrary elements in a random permutation, all initial permutations equally likely. Doing so requires no fewer comparisons than sorting an array of 0's and 1's derived by letting $x =$ the median of the elements and replacing all elements $\leq x$ with 0 and all elements $> x$ with 1. This is because, when $y > z$ in the original heap and y is a child of z, we may have $y > z$ when they are replaced with 0/1 or they may be equal. The former case leads to behavior just like in the original instance; the latter saves a recursive call in the HEAPIFY function.

Now all we need to do is show that all possible heap-ordered arrangements of 0's and 1's in which there are the same number of 0's and 1's result in $\Omega(n \lg n)$ comparisons. If they all yield $\Omega(n \lg n)$ comparisons then the average is also $\Omega(n \lg n)$.

Obvious observation: The two children of a node with key = 0 must also have keys = 0.

Consider the nodes that are parents of leaves. In a heap n nodes there are at least $n/4$ of these. The number p of parents of leaves that can have keys = 0 is governed by the inequality

$$\begin{aligned} 3p - 1 &\leq n/2 && \text{Each parent with key = 0, except possibly one,} \\ & && \text{has two children with key = 0.} \\ p &\leq n/6 + 1/3 \\ &\leq n/6 + 1 \end{aligned}$$

That means at least $n/4 - n/6 - 1$ parents of leaves have keys = 1. By the time the original leaves are no longer part of the heap, all keys must be 0's: there are at most $n/4 + n/8 + \ldots + 1 < n/2$ nodes remaining, less than the number of original 0's, and the 1's are all removed from the heap before any 0's are.

So the parents of leaves that used to be 1's have been replaced by 0's. A 0 can only replace a 1 if it has also been swapped with all the 1's along the path to it. So we have at least $n/4 - n/6 - 1 = n/12 - 1$ nodes that required $h - 2$ swaps, where h is the height of the original heap, to have their keys converted from 1's to 0's. Even if we grossly underestimate and assume at least one comparison per swap (all but possibly the last require two), we get at least $(n/12 - 1)(h - 2)$ comparisons, which is clearly in $\Omega(n \lg n)$.

The average number of comparisons is clearly at most the worst case, so it's $O(n \lg n)$, giving us the desired $\Theta(n \lg n)$ bound.

Note that the lower bound argument applies even to the *best case.*

2 points for explaining why 0's and 1's are sufficient for the analysis, 4 for proving the lower bound, 1 for proving the upper bound and completing the argument. That makes a total of 7 points – there's a built-in extra credit point if you nail it.

In the lower bound argument, 3 points should be given to any solution that is based on a bound of either the number of 0's or the number of 1's in a key part of the heap, whether it be leaves, parents of leaves, subtrees consisting of 0's only, etc.

2. [worth 8 points] *understanding Quicksort*

(a) Describe a modification of the PARTITION procedure, call it EQ-PARTITION that splits the array into three parts as follows: Return two indices r and t so that $p \leq q \leq t \leq r$, and

- all elements of $A[q \ldots t]$ are equal,
- each element of $A[p \ldots q-1]$ is less than $A[q]$, and
- each element of $A[t+1 \ldots r$ is greater than $A[q]$.

Your procedure must do $\mathrm{O}(r-p)$ comparisons and swaps and be in-place, i.e., no additional space other than a few variables may be used.

Solution: The easy way to approach this is to partition twice (let x be the pivot), once to divide the array so that elements with keys $< x$ precede ones with keys $\geq x$. The latter part of the array can be partitioned again using essentially the same algorithm.

A single pass algorithm uses a solution to the well-known "Dutch National Flag" problem.

The invariant of the following loop is as the three conditions listed above, with an important modification. A new variable u – the upper boundary of the "unknown" region – is introduced. The last invariant in the above list is replaced by: each element of $A[u+1 \ldots r]$ is greater than $A[q]$. This makes the status of the elements in $A[q \ldots u]$ unknown.

The algorithm begins by swapping the pivot into position p (or assuming it's already there). To make the invariant correct initially it makes $q = p+1$ (since the pivot is already known to be in position p) and $u = r$. For readability, let x denote $A[p]$, the pivot.

```
while $t \leq u$ do
    if $A[t] < x$ then
     (1) swap $A[q]$ and $A[t]$; $q = q+1$; $t = t+1$
    else if $A[t] = x$ then
     (2) $t = t+1$
    else ($A[t] > x$)
     (3) swap $A[t]$ and $A[u]$; $u = u-1$
    endif
end do
```

This is a clever combination of the two partition algorithms presented in class. Lines (1) and (2) emulate the book's algorithm to divide the elements less than the pivot from those equal to it. These then combine with line (3) to emulate the "start at both ends and work toward the middle" approach.

(b) Assuming that EQ-PARTITION is used for partitioning in QUICKSORT and no recursive calls are done for elements with keys equal to the pivot, what is the worst-case number of comparisons required to sort an array whose elements have keys that are entirely 0's and 1's. Your answer should give the *exact* number of comparisons based on your solution to part (a) and describe what the array for the worst case looks like. Note that the algorithm does not "know" that the inputs are 0's and 1's.

Solution: An array that has either all 0's or all 1's will require exactly $n-1$ comparisons: each element is compared to the pivot and the computation stops. Now suppose there is a mix of 0's and 1's, m of which are 0's and $n-m$ are 1's. Without loss of generality let the pivot be a 1. Then the $n-1$ initial comparisons will be followed by $m-1$ comparisons during the recursive call with all 0's. The worst case occurs when $m = n-1$, leading to $2n-3$ total comparisons. It's easy to see that there can never be more than two calls – the recursive call will always have all elements equal. Thus the worst case number of comparisons is $\mathrm{O}(n)$.

(c) Give a Θ bound for the worst case number of comparisons when the keys are from the set $\{1, \ldots, k\}$. Again your bound should assume that EQ-PARTITION is used. Your bound should be in terms of both n and k. In this situation a Θ bound would be defined as:

> $f(n,k)$ is $\Theta(g(n,k))$ if there exist four positive constants c_1, c_2, n_0 and k_0 so that
> $c_1 g(n,k) \leq f(n,k) \leq c_2 g(n,k)$ for all $n \geq n_0$ and $k \geq k_0$.

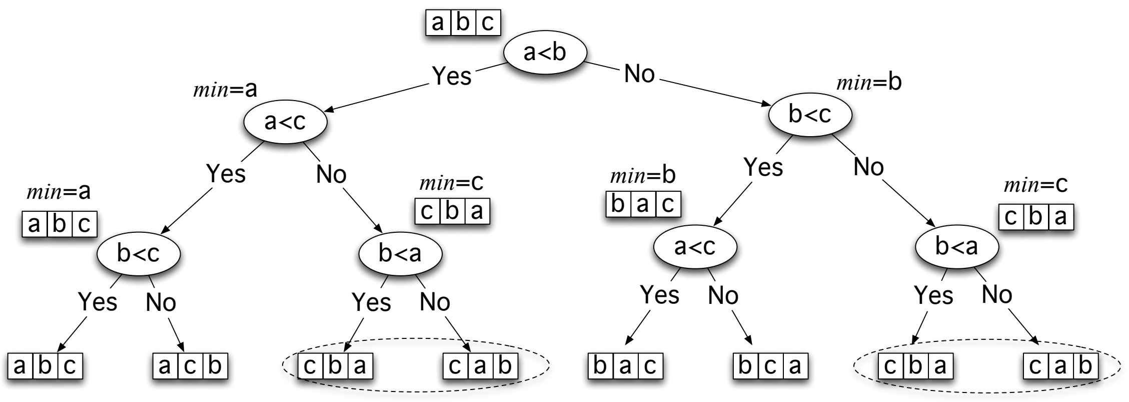

Figure 1: Decision tree for seclection sort on three elements.

For the lower bound, you should describe what a general worst-case example looks like.

Solution: To get a big-oh bound, note that there can be at most k calls to the function along any branch of the recursion tree. Each call eliminates one of the k possible keys from consideration (because it was the pivot). Clearly no more than $n-1$ comparisons occur during any call. This means the algorithm makes $O(nk)$ comparisons.

For the Ω bound consider a situation where each of the elements $1, \ldots, k-1$ occurs exactly once and k occurs $n-k+1$ times. Consider a sequence of k calls where key i becomes the pivot for the i-th call. At the beginning of the i-th call there will be $n-i+1$ keys from the set $\{i, \ldots, k\}$. The total number of comparisons is therefore

$$\sum_{1 \le i \le k} (n-i) = nk - \sum_{1 \le i \le k} i = k(n-(k+1)/2) \ge k(n-k/2)$$

Because we can assume $n \ge k$, the number of comparisons is $\ge nk/2$ and therefore $\Omega(nk)$.

3 points for part (a), 2 points for (b), 3 points for (c)

3. [worth 4 points] *understanding decision trees*

Draw the decision tree for *selection sort* on 3 elements. What is the worst case number of comparisons? What is the average case?

The tree is shown in Fig. 1. Every branch has height 3; so both the average and the worst case number of comparisons is 3. The dotted ovals indicate that there are duplicate leaves (accounting for the fact that there are 8 leaves instead of the expected 6). These arise because there are two ways for c to end up being the minimum element during the first pass.

4. [worth 6 points] *understanding the linear-time selection algorithm*

Exercise **9.3-1** on page 223 (page 192 in 2/e).

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:

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

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 $c \geq c_0$.

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

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:

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

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

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

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.

3 points for each part