CSC505 HW2

Liam Adams

February 24 2019

Problem 1

a) The heapsort algorithm from the book is

```
HeapSort(A)
    Build-Max-Heap(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size-1
        Max-Heapify(A, 1)
```

Build-Max-Heap runs in linear time and is responsible for the first n term in $O(n + klgn)$.
Now for the for loop - the worst case runtime for Max-Heapify is lgn, when the array is not a heap and it has to traverse every level of the tree to make it a heap. This worst case scenario can only occur k times with this input however. Each call to Max-Heapify will place a 1 at the root of the heap, which is then removed before the next call to Max-Heapify. After the for loop runs k times, all of the 1's will have been removed from the heap and all the remaining elements will all be 0's, forming a heap. Max-Heapify will take constant time when every element of the array is equal. Therefore the worst case runtime is $O(n + klgn)$

b) This heap can be thought of as a binary decision tree that is done sorting when all of the 1's have been removed. MaxHeapify must run at least k times in order to remove all the 1's. For any input size, when all of the 1's are at the bottom level of the tree, maxHeapify will take at least $klgn$ time. Therefore, with one execution of BuildHeap, the algorithm is $\Omega(n + klgn)$ with these inputs.

c) Excluding the Build-Max-Heap operation as the problem states, when the for loop begins with the heap array [1, 1, 0, 1, 0] it performs 6 comparisons. When the heap array is rearranged to [1, 1, 0, 0, 1] it only performs 5 comparisons.

In figure 1 the first run through the for loop takes 3 comparisons because [0,5] must be moved all the way down and compared to [1,4] at the third level. So it performs 6 comparisons total
In figure 2 the first run through the loop only takes 2 comparisons because it is still a valid heap after swapping [1,4] to the root. After the first run through the loop the heaps in both cases are identical so the same number of comparisons are made. So it performs 5 comparisons total.

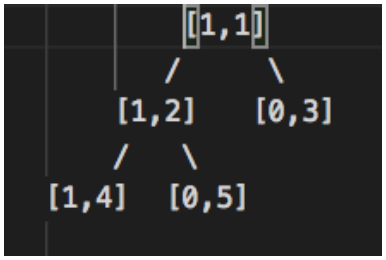

Figure 1: [1, 1, 0, 1, 0]

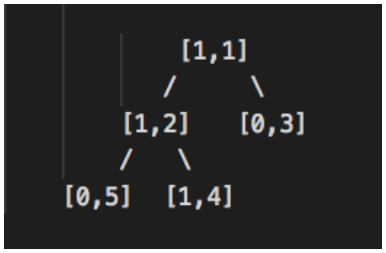

Figure 2: [1, 1, 0, 0, 1]

Problem 2

a) The algorithm below will test every possible pair in $\Theta(n^2)$ comparisons. R is an array holding the red jugs, B is an array holding the blue jugs, and Pairs is an array holding the output of the algorithm which is a group of pairs. CompareJugs encapsulates the operation of filling the red jug with water and pouring it into the blue jug. It returns true if the jugs have the same volume. CompareJugs takes one time unit.

```
GroupJugs(R, B, Pairs)
    for i = 0 to n
        for j = 0 to n
            if CompareJugs(R[i], B[j]) == true add (R[i], B[j]) to Pairs
```

b) This is a kind of comparison based sorting algorithm. If, for example, you hold the blue array constant, you can think of the problem as finding the right permutation of the red array so that Volume(R[i]) == Volume(B[i]) for all i from 0 to n-1. You can construct a decision tree that includes all the permutations, and as you follow a path the number elements of R in the correct position increase until you hit a leaf. The permutation at the leaf is the correct ordering. So the tree has at least n! leaves and the height must be at least lg(n!), which means the worst case number of compares is at least lg(n!). Therefore we have

$$lg(n!) = \sum_{i=1}^{n} lg(i) \geq \frac{n}{2} lg \frac{n}{2} = \frac{1}{2} nlgn - \frac{n}{2} = \Omega(nlgn)$$

as a lower bound for comparisons.

c) Since you know that for any blue jug, there is a red jug with the same volume and vice versa, you can perform quicksort on the blue array by randomly selecting a red jug as the partition element and partitioning the blue array based on that jug. You know that one blue jug will be equal to the red partition jug, so you will be able to put that blue jug in its correct position on each iteration. The position of the blue jug will also be the correct position of the red jug in its array, so you can sort

both at once. Then you can access pairs by using the same index in both arrays. By choosing each partition element at random, the expected number of comparisons for the simultaneous quicksort of both arrays is $O(nlgn)$.
Borrowing the proof of randomized quicksort from the book, the expected value on the number of comparisons made, where $\frac{2}{j-i+1}$ is the probability that elements x_j and x_i are compared is equal to the expected value of the following summation

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

which is $O(nlgn)$ using $k = j - i + 1$. The worst case number of comparisons is n^2 if every partition jug chosen is the minimum or maximum.

Problem 3

The algorithm will work in linear time if the input elements are divided into groups of 7. At least half of the $\lceil \frac{n}{7} \rceil$ groups contribute at least 4 elements that are greater than the median of medians x, except for the group containing the median of medians and the group with less than 7 elements. This gives at least

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

elements greater than x. So in the worst case step 5 of SELECT is recursively called on $n-(\frac{2n}{7}-8) = \frac{5n}{7} + 8$ elements. Of the 5 steps in SELECT, steps 1, 2, and 4 take $O(n)$ time and step 3 takes $T(\lceil \frac{n}{7} \rceil)$ time. So we have

$$T(n) \leq c\lceil \frac{n}{7} \rceil + T(\frac{5n}{7} + 8) + O(n) \leq \frac{cn}{7} + c + \frac{5cn}{7} + 8c + an$$

This can be rewritten as $cn + (-\frac{cn}{7} + 9c + an)$ which is at most cn if $-\frac{cn}{7} + 9c + an \leq 0$. This is true for $c = 14a$ and $n \geq 126$, Therefore we have $T(n) = O(n)$.

SELECT does not run in linear time if groups of 3 are used because dividing into 3 groups doesn't reduce the problem size enough compared to the worst case number of elements SELECT will have to process recursively. Following the same steps above we have $T(n) \leq cn + 3c + an$, and with T monotonically increasing you will never have $T(n)$ at most cn.