Kushagr,Gupta & Rayan,Dasoriya
kgupta7,rdasori@ncsu.edu

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 $k \in \omega(n/\lg n)$.

   Taking n = $2^{h+1} - 1$ and k = $n/2 - 1$
   After the MakeHeap phase of HeapSort (Which runs in linear time),let us assume we will have a binary heap of height 'h' (due to the selected value of n) with the leaf nodes all 0's and all interior nodes as 1's. The total comparisons in this phase are $\leq$ 4n
   Now, for the 2nd phase of HeapSort, we exchange the root with the last value in the array (or any linear data structure) and reduce the array size in consideration by one till all elements have been moved through the root to their correct position.
   For the 1st k elements, we will exchange the 1 at root with the final element in consideration, which due to the relation of k and n will always be a 0. This 0 will then be percolated down to maintain the heap invariant of the parent being larger than the child, till it reaches a leaf node, or a node with both children 0.
   The number of key comparisons will be $\leq 2 \lg n$ per 1 that is encountered, which makes the number of comparisons $\leq 2k \lg n$
   For the remaining 0's there will be exactly 2 comparisons, since there will only be 0's in the heap. Hence the number of comparisons $\leq 2 * n/2$
   Therefore total comparisons $\leq 5n + 2k \lg n \in O(n + k \lg n)$
   This will remain linear as long as $k \lg n \in O(n)$ i.e the $k \lg(n)$ term grows slower than n
   Conversely, the term will not be linear if $k \lg n \in \omega(n)$ i.e $k \in \omega(n/\lg n)$

   (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.

   Since we have already proved the upper bound, we just need to prove the lower bound for this, i.e., $\Omega(n + k \lg n)$. To prove:

   $$0 \leq cg(n) \leq f(n) \text{ for some } c, n_0 \geq 0$$

   Let us assume two variables c and n

This gives us :

$$c(n + k \lg n) \leq (5n + 2k \lg n)$$
$$(5 - c)n + (2 - c)k \lg n) > 0$$

So for all $c < 2$, $n_0 > 4$ and $1 \leq k < n/2$, we can say that the total number of comparisons $= \Omega(n + k \lg n)$

(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 on the positions of the 1's. Since your proof will require an example with duplicate keys, you should use key,value pairs to distinguish between elements. Make your example as small as possible.

The original ordering of the 1's in the input array data structure affects the total number of comparisons by affecting the Heapify phase of HeapSort, where the number of comparisons would directly depend on the number of 1's that are children of 1's (Runs of 1) when a 0 is put at the root position.

This can be illustrated using the following 2 cases:
Taking Element of Array as (key, value)
A : [(1,1),(1,2),(1,3),(0,1),(0,2)]
B : [(1,1),(1,2),(0,1),(1,3),(0,2)]

Running HeapSort on these will result in the following operations on the above arrays:

Initial							
Key	1	1	1	0	0		Comparisons
Value	1	2	3	1	2		0
MakeHeap							
Key	1	1	1	0	0	Compare	Comparisons
Value	1	2	3	1	2	1,2 & 1,3	2
MakeHeap							
Key	1	1	1	0	0	Compare	Comparisons
Value	1	2	3	1	2	0,1 & 0,2	4
Extract-Max							
Key	0	1	1	0	1		Comparisons
Value	2	2	3	1	1		4
Heapify							
Key	1	0	1	0	1	Compare	Comparisons
Value	2	2	3	1	1	0,1 & 0,2	6
Extract-Max							
Key	0	0	1	1	1		Comparisons
Value	1	2	3	2	1		8
Heapify							
Key	1	0	0	1	1	Compare	Comparisons
Value	3	2	1	2	1	0,1 & 0,2	8
Extract Max							
Key	0	0	1	1	1		Comparisons
Value	1	2	3	2	1		8
Heapify							
Key	0	0	1	1	1	Compare	Comparisons
Value	1	2	3	2	1	0,1	9
Extract-Max							
Key	0	0	1	1	1		Comparisons
Value	2	1	3	2	1		9
Heapify							
Key	0	0	1	1	1		Comparisons
Value	2	1	3	2	1		9
Extract-Max							
Key	0	0	1	1	1		Comparisons
Value	2	1	3	2	1		9

Figure 1: For Array A

Initial							
Key	1	1	0	1	0		Comparisons
Value	1	2	1	3	2		0
MakeHeap							
Key	1	1	1	0	0	Compare	Comparisons
Value	1	2	3	1	2	1,2 & 1,3	2
MakeHeap							
Key	1	1	1	0	0	Compare	Comparisons
Value	1	2	3	1	2	0,1 & 0,2	4
Extract-Max							
Key	0	1	0	1	1		Comparisons
Value	2	2	1	3	1		4
Heapify							
Key	1	0	0	1	1	Compare	Comparisons
Value	2	2	1	3	1	0,1 & 0,2	6
Heapify							
Key	1	1	0	0	1	Compare	Comparisons
Value	2	3	1	2	1	1,2 & 1,3	8
Extract-Max							
Key	0	1	0	1	1		Comparisons
Value	2	3	1	2	1		8
Heapify							
Key	1	0	0	1	1	Compare	Comparisons
Value	3	2	1	2	1	0,1 & 0,2	10
Extract-Max							
Key	0	0	1	1	1		Comparisons
Value	1	2	3	2	1		10
Heapify							
Key	0	0	1	1	1	Compare	Comparisons
Value	1	2	3	2	1	0,1	11
Extract-Max							
Key	0	0	1	1	1		Comparisons
Value	2	1	3	2	1		11
Heapify							
Key	0	0	1	1	1		Comparisons
Value	2	1	3	2	1		11
Extract-Max							
Key	0	0	1	1	1		Comparisons
Value	2	1	3	2	1		11

Figure 2: For Array B

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): red and blue jugs.

   (a) $\Theta(n^2)$ Deterministic Algorithm
   A deterministic algorithm to pair n Red and n Blue jugs of equal size will be the brute force method of picking a random red jug and comparing it with blue jugs till we find one. This will be repeated for n red jugs, which in the worst case will have to be compared with n blue jugs, which will result in nxn comparison , which is of form $\Theta(n^2)$

   (b) $\Omega(n \lg n)$ lower bound.
   So for a distinct ordering of red jugs, we have to find a distinct ordering of blue jugs with the minimum possible comparisons. We can imagine this as a decision tree, where the i^{th} red jug is compared to the i^{th} blue jug at the i^{th} height of the tree from the root.
   Since the i^{th} blue jug can be only one of the following: larger than, smaller than or equal to the i^{th} red jug, the tree being formed has 3 children at every node.
   For n distinct items, the number of possible permutations of all items is $n!$
   Hence there are $n!$ leaf nodes to the tree, since each permutation is a possible reordering candidate.
   Given 3 children per internal node and $n!$ leaf nodes in a complete tree, the height of the tree will be $\log_3(n!)$
   Using Sterling's Approximation, the number of comparison will be $\Omega(n \lg n)$

   (c) $O(n \lg n)$ Random Algorithm
   Given that there are n red jugs and n blue jugs, each of distinct capacity from jugs of the same color and having exactly one jug of other color with equal capacity, we could use a randomised quicksort style algorithm to sort n elements of each color and have the i^{th} item of sorted red jugs be of equal capacity to the i^{th} item of the sorted blue jugs.
   Assuming random and unequal orderings of the red and blue jugs, first we pick a random jug from red jugs, and use it to partition the blue jugs.
   Using n comparisons we will find a set of blue jugs smaller than the red jug, a set of blue jugs larger than the red jug, and a single blue jug equal to the red jug.
   We can use the single blue jug to similarly partition the red jugs in n-1 comparisons.
   At this point we have identified a pair of red and blue, and a set each of red jugs larger and smaller than the identified red jug, and a set each of blue jugs smaller and larger than the identified blue jug.
   Since there exists an equal red jug for every blue jug, the set of "larger" blue jugs will be of equal size to the set of "larger" red jugs, but possibly in a different order. The same is true for the set of "smaller" jugs for each color.
   Now for the base case, when the size of the set is 1, we have found a pair.
   By induction we can prove that this algorithm is correct.
   Since the comparison of any two jugs is taken as one time unit, it is logically similar to the cost of comparing two numbers. Hence we can take the expected runtime of randomised quicksort which is $O(n \lg n)$ as the expected runtime of the algorithm.
   In the worst case, where we always take one of the extremes (largest or smallest) of the set as the pivot we will make $\sum_{i=1}^{n} 2(i) - 1 = n^2$ comparisons.

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): selection with groups of size other than five.

   In this exercise question, we are supposed to consider two cases.
   1. Number of elements = 7
   2. Number of elements = 3

   Case 1: Number of elements = 7
   The number of elements which will be greater than x will be

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

   So, the equation turns out to be $T(n) \leq T(n/7) + T(5n/7) + O(n)$.

   We know that if $T(n) = T(an) + T(bn) + c(n)$and if a+b <1, then T(n)= $\Theta(n)$. And since $1/7 + 5/7 = 6/7 < 1$. Hence, the algorithm works in linear time.

   Case 2: Number of elements = 3
   The number of elements which will be greater than x will be

$$2\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{3}\right\rceil\right\rceil-2\right)\geq\frac{n}{3}-4,$$

   So, the equation turns out to be $T(n) \leq T(n/3) + T(2n/3) + O(n)$.

   We know that if $T(n) = T(an) + T(bn) + c(n)$ and if a+b <1, then T(n)= $\Theta(n)$. But here, we can see that $a + b = 1/3 + 2/3 \not< 1$. Hence, the algorithm will not work in linear time. Moreover, it will take $n \lg n$ running time.