Homework Assignment #3

Due: Noon, 6th Nov, 2017
Submit **printed** solutions via eClass
Submit **handwritten** solutions at dropbox on CSC level 1

CMPUT 204
Department of Computing Science
University of Alberta

Note: All logs are base-2 unless stated otherwise.
Your maximum score on this exercise is 110pts.

Exercise I. What is the result of `Build-Max-Heap` on the input $A = [16, 4, 3, 9, 1, 44, 29]$?
Illustrate all intermediate steps.

What is the result of increasing the key 1 to 34 once the max-heap is built?

Exercise II. What is the result of the following sequence of operations on a (i) standard BST, (ii) an AVL-tree and (iii) a RB-tree: `Insert(16)`, `Insert(4)`, `Insert(3)`, `Insert(9)`, `Insert(1)`, `Insert(44)`, `Insert(29)`, `Delete(1)`, `Delete(4)`, `Insert(34)`.

Exercise III. The *Birthday Paradox*: If we assume that each person's birthday is picked uniformly at random (and independently of everyone else in the room), then, even though there are 365 days in a year (disregarding leap-day) — in a party of 30 random people it holds with overwhelming probability that at least one pair of people have the same birthday. In this question, you will show why this holds.

Suppose that the are N possible colors, and each person is assigned a color uniformly at random and independently from anyone else.

- Show that for any two people i and j it holds that $\Pr[i \text{ and } j \text{ have same color}] = \frac{1}{N}$.

So intuitively, in a set of N people, we are likely to see at least one pair of people with the same color. The Birthday Paradox is the fact that this happens for much smaller sets — of the order of $O(\sqrt{N})$.

- Show that if we take a set of $n = \lceil \sqrt{2N} + 1 \rceil$ people, then we can expect to have at least one pair of people with the same color. (Make sure you define the suitable Bernoulli random variables.)

Exercise IV. Sorting Problems:
(a) Suppose that we want to sort an array A that contains $n + k$ elements, where the first n elements are sorted, and the latter k elements are arbitrary.
(An example with $n = 8$ and $k = 2$ is $A = [\underbrace{1, 4, 17, 19, 31, 33, 34, 73}_{n}, \underbrace{72, 12}_{k}]$)

Assume n is very large (say, $n = 10^6$) and that k is a small constant (say, $k = 10$). Which of the sorting methods we have seen so far (InsertionSort, MergeSort, HeapSort, QuickSort) would be especially suitable for such a task? Explain why and justify your answer.

(b) Give an algorithm using the techniques and data structures you have seen in the course so far, for the following problem. The input is an array A of size n of integers and some $k \leq n$. The output is the k largest elements of A. Your algorithm must run in $O(n + k \log n)$. Explain your algorithm and analyze its running time. What is the asymptotic upper bound on k for which your algorithm is still linear in n?

Problem 1. (45 pts) The **Selection** problem takes two inputs: an array (unsorted) A of n pairwise comparable elements, and an integer $k \in [1, n]$; and outputs the element x s.t. $\#\{y \in A : y \leq x\} = k$.
If we assume all elements in A are unique (which you may assume for the rest of the question), then your goal is to return the unique element which is strictly greater than exactly $k-1$ other elements and strictly smaller than exactly $n-k$ other element.
Note that you cannot assume k is constant. In fact, a common application of the **Selection** problem is to set $k = \frac{n}{2}$ and return the median of the elements in A.

(a) (3 pts) Give an algorithm that takes $\Theta(n \log(n))$ time for the **Selection** problem.
Answer. The algorithm sorts all elements in A, then returns $A[k]$. Sorting takes $\Theta(n \log(n))$ time with, say, HeapSort, and outputting the k-th element takes $O(1)$ times. Thus, the algorithm's overall running time is $\Theta(n \log(n))$.

In the remainder of the question, our goal is to design an algorithm for the **Selection** problem that runs in linear time in the number of elements in the array, namely runtime of $O(n)$ where $n = r - p + 1$. Here is attempt #1:

```
Select(A, p, r, k)          ** precondition: (p − 1) + k ≤ r
s ← Partition(A, p, r)
index ← s − p + 1
if (index = k) then
   return A[s]
if (index > k) then
   return Select(A, p, (s − 1), k)
if (index < k) then
   return Select(A, (s + 1), r, k − index)
```

(b) (5 pts) Prove that `Select` indeed solves the **Selection** problem. I.e., that `Select` indeed returns x that would appear in the k-th place had we sorted A.
Answer. Prove by induction of $p - r$.
Base case: $p - r = 0$, thus $r = p$ and so $k = 1$. In this case `Partition` must return $s = p (= r)$ and so we return $A[p]$ which is the 1st element in $A[p, ..., r]$.
Induction step: Assuming the claim holds for any p, r such that $r - p < i$ and any k satisfying $(p-1)+k \leq r$ we show it holds for $r - p = i$.
`Partition` puts $s - 1$ elements that are $\leq A[r]$ in $A[p, ..., s-1]$; puts $r - s$ elements that are $\geq A[r]$ in $A[s+1, ..., r]$; and puts $A[r]$ is position s.

- If $s = p + k - 1$: then it must holds that $A[r]$ is the k-th element in $A[p, .., r]$ and we indeed return it.
- If $s > p + k - 1$: Note that all elements in $A[s, ..., r]$ are greater than the $(s-1) - p + 1$ elements in $A[p, p+1, .., s-1]$. Thus, all elements in $A[s, ..., r]$ are strictly greater than more than $s - 1 - p + 1 > (p + k - 1) - p = k - 1$ elements in A – i.e., from at least k other elements. Thus, all of the elements in $A[s, ..., r]$ are too big to be the k-th element. Therefore the k-th element in $A[p, ..., r]$ is the k-th element in $A[p, .., s-1]$. By induction, our recursive call returns precisely this element.
- If $s < p + k - 1$: Note that all elements in $A[p, ..., s]$ are strictly smaller than the $r - (s+1) + 1$ elements in $A[s+1, s+2, ..., r]$. Thus, all the elements in $A[p, ..., s]$ are strictly smaller than more than $r - (s+1) + 1 > r - (p + k - 1) = (r - p + 1) - k$ other elements, and so they are too small to be the k-th element. Therefore, the k-th element (which is smaller than exactly $(r - p + 1) - k$ elements) is found in $A[s+1, ..., r]$. Note that $A[s]$ is the $(s - p + 1)$th element in the sorted $A[p, .., r]$ array, thus, the k-th element in the sorted array $A[p, ..., r]$ appears $k - (s - p + 1)$ places after $A[s]$. By induction, this is precisely the element we return. ■

(c) (2 pts) We invoke `Select`$(A, 1, n, k)$. Show that the best-case running time of `Select` is $O(n)$ and give an example of a best-case instance.
Answer. In the best-case, we never invoke a recursive call to `Select`, thus our runtime is merely the runtime of `Partition` which is $O(n)$. A best-case instance is such that the k-th element happens to be in the r-th (last) position.

(d) (3 pts) We invoke `Select`$(A, 1, n, k)$. Show that the worst-case running time of `Select` is $\Omega(n^2)$ and give an example of a worst-case instance.
Answer. Assume $k \leq n/2$. In the worst-case, r is the last element in the array, and so we keep recursing `Select` on a sub-instance with $n-1$ elements. Thus we have $T(n) = c \cdot n + T(n-1)$ for some constant c and $T(k) = c \cdot k$. This recursion solves to $\sum_{i=k}^{n} c \cdot i = c \cdot \frac{(n-k+1)(n+k)}{2} \geq \frac{c}{8} n^2$. An example of a worst-case instance is when $A[1, .., n]$ is sorted to begin with.

The randomized version of `Select` is described below.

```
RSelect(A, p, r, k)          ** precondition: p − 1 + k ≤ r
i ← uniformly chosen random integer in {p, p + 1, ..., r}
exchange A[i] ↔ A[r]
s ← Partition(A, p, r)       ** From here on the code is like Select
index ← s − p + 1
if (index = k) then
   return A[s]
if (index > k) then
   return RSelect(A, p, (s − 1), k)
if (index < k) then
   return RSelect(A, (s + 1), r, k − index)
```

(e) (8 pts) Let $f(n)$ denote the expect runtime of `RSelect` *in the worst-case*, i.e. $f(n) = \mathbf{E}[T(n)]$. Prove that there exists a constant c s.t.

$$f(1) \leq c \quad \text{, and} \qquad f(n) \leq c \cdot n + \frac{2}{n} \sum_{i=n/2}^{n-1} f(i)$$

Hint: In the worst case, we always recurse on the largest of the two subarrays `Partition` creates.
Answer. Given large enough n we have that first we pick the random pivot (which takes $O(\log(n))$ as we need to pick $\lceil \log(n) \rceil$ random bits), we then run `Partition` which takes $O(n)$ time, and then check which condition holds and recurse once on a subarray from one end to the pivot. Thus, if we picked $pivot = A[i]$ which is the n_1-th element in the sorted array, we either recurse on $(n_1 - 1)$ elements smaller than *pivot* or $n - n_1 - 1$ elements larger than *pivot*. Of course, in the worst-case, we recurse on $\max\{n_1 - 1, n - 1 - n_1\}$. Denoting c as the constant hidden in the big-O notation for `Partition`, and setting it large enough to account also for the constant number of operations we do per one recursive call and for $f(1)$ (which only takes a constant number of operations) we have that $f(n) = c \cdot n + f\left(\max\{n_1 - 1, n - n_1 - 1\}\right)$.

This leaves the question of what is n_1. Since *pivot* is chosen uniformly at random than $\Pr[n_1 = j] = \frac{1}{n}$

for any j. Thus

$$\begin{aligned} f(n) &\le c\cdot n + \sum_{j=1}^{n} \tfrac{1}{n} f(\max\{j-1, n-j-1\}) \\ &\le c\cdot n + \tfrac{1}{n}\sum_{j=1}^{n/2} f(n-j-1) + \tfrac{1}{n}\sum_{j=n/2+1}^{n} f(j-1) \qquad \text{Since } (j-1 > n-j-1 \ \Leftrightarrow 2j > n) \\ &= c\cdot n + \tfrac{1}{n}\sum_{j=n/2+1}^{n} f(j-1) + \tfrac{1}{n}\sum_{j=n/2+1}^{n} f(j-1) \\ &= c\cdot n + \tfrac{2}{n}\sum_{j=n/2}^{n-1} f(j) \ \blacksquare \end{aligned}$$

(f) (2 pts) Prove that $\mathbf{E}[T(n)] = \Theta(n)$.
Answer. We prove by induction that $f(n) \le 10cn$.
Base case: for $n = 1$ we have $f(1) \le c < 10c \cdot n$.
Induction step: Assuming $f(i) \le 10c \cdot i$ for every $i < n$ we show that $f(n) \le 10c \cdot n$. Indeed:

$$\begin{aligned} f(n) &\le c\cdot n + \tfrac{2}{n}\sum_{i=n/2}^{n-1} f(i) \le c\cdot n + \tfrac{20c}{n}\sum_{i=n/2}^{n-1} i \\ &\le c\cdot n + \tfrac{20c}{n}\left(\tfrac{1}{2}\cdot\tfrac{n}{2}\cdot(n+\tfrac{n}{2})\right) = c\cdot n + \tfrac{20c}{n}\cdot\tfrac{3n^2}{8} \le c\left(1+\tfrac{60}{8}\right)n \le 10c\cdot n \end{aligned}$$

Therefore, $\mathbf{E}[T(n)] \in O(n)$. In contrast, since we must run `Partition` at least once then $T(n) \in \Omega(n)$ (regardless of the pivot chosen), so $\mathbf{E}[T(n)] \in \Theta(n)$. ■

And here is a deterministic version of `RSelect` (you may assume n is divisible by 5 or 10)

```
DSelect(A, p, r, k)          ** precondition: p − 1 + k ≤ r
if (r − p < 5) then
   Sort(A[p, ..., r]) and return A[k].
                             ** Sorting can be done using even InsertionSort since any
                             ** (reasonable) sorting algorithm on ≤ 5 elements takes O(1) time
Let B be an array of length n/5.
for (i from 1 to n/5)
   B[i] ← DSelect(A, (5(i − 1) + 1), 5i, 3)
key ← DSelect(B, 1, n/5, ⌊½ · n/5⌋)
for (j from 1 to n)
   if (A[j] = key) then
      i ← j
exchange A[i] ↔ A[r]         ** From here on the code is like RSelect
s ← Partition(A, p, r)
index ← s − p + 1
if (index = k) then
   return A[s]
if (index > k) then
   return DSelect(A, p, (s − 1), k)
if (index < k) then
   return DSelect(A, (s + 1), r, k − index)
```

(g) (12 pts) Prove that the $key(= A[i])$ chosen by the recursive call of `DSelect` on B satisfies

$$\#\{x \in A : x \leq key\} \geq \tfrac{3}{10}n \quad \text{, and similarly} \quad \#\{x \in A : x \geq key\} \geq \tfrac{3}{10}n$$

You may assume n is divisible by 10 and that all elements in A are unique.

Answer.

What is key? It is the median of B. Thus $key \geq$ half of the elements in B, which, by definition, holds $n/5$ elements all of which were originally in A. Hence we already know $key \geq \frac{n}{10}$ elements in A. In other words, denoting

$$J = \{j : B[j] < key\} = \{j : median(A[5j-4, .., 5j]) \leq key\}$$

we know $|I| \geq \frac{n}{10}$.

However, each element in B is taken from a unique 5-tuple in A, and moreover, it is the *median* of that 5-tuple. Thus, for every $j \; in J$ with $B[j] < key$ there exists 2 additional elements in $A[5j-4, .., 5j]$ that are $\leq B[j]$. Combining the two arguments, we have that for every $j \in J$ the chosen $keyt$ is greater then 3 elements: $B[j]$ and the two other elements in $A[5j-4, .., 5j]$ smaller than $B[j]$. Thus $\#\{x \in A : x \leq key\} \geq 3|J| = \frac{3}{10}n$.

The argument regarding the set of elements $\geq key$ is symmetric. ■

(h) (10 pts) Write down the recursion describing the runtime $T(n)$ of `DSelect` and prove that $T(n) = \Theta(n)$. Hint: You may rely on previous homework without reproving them.

Answer. We denote $T(n)$ as the runtime of `DSelect` on A of size n. Clearly $T(n) \in \Omega(n)$ as regardless of the pivot we run `Partition` that takes $\Omega(n)$ time.

Populating B takes $O(n)$ time, since we sort and find the median of each 5-tuple in $O(1)$ time. Recursing thus takes $T(n/5)$-time. Finding the j for which $A[j] = key$ takes $O(n)$ time as we traverse all elements in A again. `Partition` takes $O(n)$ time and then we recurse on an instance of size $\leq \frac{7}{10}n$ which takes at most $T(\frac{7}{10}n)$. It follows that there exists a constant c s.t.

$$T(n) = c \cdot n + T(\tfrac{n}{5}) + T(\tfrac{7n}{10})$$

Homework #2 solves this recursion (in the special case of $c = 3$ but that will only effect the constants in the solution) and shows that $T(n) = O(n)$. Combining the lower- and upper-bound we get $T(n) \in \Theta(n)$. ■

Problem 2. (10 pts) Lower bound on Ramsey numbers:

A clique on k nodes is the complete graph on these n nodes: between any two nodes there's an edge. Thus, a clique on k nodes has $\binom{k}{2}$ edges.

(a) (2 pts) Suppose we color the edges of a clique on k nodes randomly, independently and uniformly in red or blue. Show that the probability that the clique is monochromatic — namely, all edges are colored by the same color — is $2^{-\frac{k^2-k-2}{2}}$.

Answer. Given a collection of $\binom{n}{2}$ edges, each colored independently from the rest in red with probability $\frac{1}{2}$ and in blue with probability $\frac{1}{2}$, the probability that all of them are colored red is $\left(\frac{1}{2}\right)^{\binom{k}{2}}$; and the probability that all of them is colored blue is (the same) $\left(\frac{1}{2}\right)^{\binom{k}{2}}$. Hence, the probability of all $\binom{k}{2}$ edges being colored the same color is

$$2 \cdot 2^{-\binom{k}{2}} = 2^{1-\frac{k(k-1)}{2}} = 2^{\frac{-k^2+k+2}{2}}$$

(b) (1 pts) Argue by induction that for any $k \geq 3$ we have $\frac{2^{\frac{k}{2}+1}}{k!} < 1$.

Answer. We prove the required by induction on k. For the base case of $k = 3$ we have $2^{1+\frac{3}{2}}3! = \frac{2^{\frac{5}{2}}}{6} = \frac{\sqrt{32}}{\sqrt{36}} < 1$. For the induction step, we fix any $k \geq 3$ and assume the claim holds for k, and so $\frac{2^{\frac{k+1}{2}+1}}{(k+1)!} = \frac{\sqrt{2}}{k+1} \cdot \frac{2^{\frac{k}{2}+1}}{k!} \overset{\text{IH}}{<} \frac{\sqrt{2}}{4} \cdot 1 < 1$.

(c) (7 pts) Suppose we color the edges of a clique on n nodes randomly, independently and uniformly in red or blue. Show that the expected number of monochromatic cliques of size k we create is strictly smaller than 1 given that $k \geq 2\log(n) \geq 3$.

Hint: Make sure you define the suitable Bernoulli random variables. You may also use the naïve bound: $\binom{n}{k} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k!} \leq \frac{n^k}{k!}$.

Answer. For any subset S of size k of the n nodes, let X_S be the indicator random variable whose value is 1 if all edges between the nodes of S are colored in the same color. Since all the edges between any pair of vertices in S are colored randomly, independently and uniformly red or blue, then for a given S we have $\mathbf{E}[X_S] = 1 \cdot \Pr[S \text{ is monochromatic}] = 2^{-\frac{k^2-k-2}{2}}$.

There are $\binom{n}{k} \leq \frac{n^k}{k!}$ such possible subsets. Therefore, because of the linearity of expection we have that the expected number of monochromatic cliques on k-vertices we create is

$$\begin{aligned}
\mathbf{E}\left[\sum_S X_S\right] &= \sum_S \mathbf{E}[X_S] = \binom{n}{k} \cdot 2^{-\frac{k^2-k-2}{2}} \\
&\leq \frac{1}{k!} \cdot n^k \cdot 2^{-\frac{k^2-k-2}{2}} = \frac{1}{k!} \cdot 2^{k\log(n) - \frac{k^2-k-2}{2}} \overset{\log(n) \leq \frac{k}{2}}{\leq} \frac{2^{k \cdot \frac{k}{2} - \frac{k^2-k-2}{2}}}{k!} = \frac{2^{\frac{k}{2}+1}}{k!} \\
&\overset{k \geq 3}{<} 1
\end{aligned}$$

Explanation: The Ramsey number $R(k, l)$ is the smallest natural n such that any coloring of the edges of the complete graph over n nodes in red or blue must produce either a monochromatic red clique of size k or a monochromatic blue clique of size l. We argue that you have just proven that $R(k, k) > \lfloor 2^{\frac{k}{2}} \rfloor$.

Note that by definition, the number of monochromatic k-cliques is an integer. If the expected number of monochromatic k-cliques is smaller than 1, then *there has to be at least one coloring of the edges of a $2^{k/2}$-clique that creates no red k-clique and no blue k-clique!* (This is because expectation is averaging — for each coloring of the edges in red and blue we count the number of monochromatic k-cliques we create and then average those numbers. Since the average of those integers is smaller than 1 then one of these integers must be smaller than 1 and therefore 0.)

This is a classic result by Erdős (1947), which started the probabilistic method: providing examples without actually specifying them, but rather just pointing out to the fact that such examples must exist – using probability.

Problem 3. (30 pts)
(a) (15 pts) Your TAs love cereals — for the prize in the box. They have purchased n boxes of cereal, one of which contains the big (and heavy) prize. Instead of opening and emptying all n boxes, they'd like to use your help.
Design an algorithm that uses the scales in their lab — on which you can weigh any set of boxes against any other set of boxes and see which one is heavier — and minimizes the number of weighs required until the one heavier box that contains the prize is found.

Answer. The algorithm partitions n into 3 equi-size disjoint subsets (note that always there are two sets of the same number of elements) and weighs two sets of the same size against one another. If one of them is heavier, it recurses on that set; otherwise, it recurses on the set of elements left out. This way we maintain the invariant that we always recurse on a subset of boxes that contains the heavy prize. Thus, $T(n) = 1 + T(\lceil \frac{n}{3} \rceil)$. Also, whenever $n = 1$ we don't make any weighs, just output the singleton box we have as it must contain the heavy prize. So $T(1) = 0$.

We argue that $T(n) \leq \lceil \log_3 n \rceil$, and prove the claim by induction.
Base case: We can easily verify that $T(1) = 0 = \log_3 1$, $T(2) = T(3) = 1 = \lceil \log_3 2 \rceil = \log_3 3$.
Induction step: Assuming the claim holds for any $1 \leq i < n$ we show it also holds for n. We thus have that

$$T(n) \quad = \quad 1 + T(\lceil \tfrac{n}{3} \rceil) = 1 + \lceil \log_3 \left(\lceil \tfrac{n}{3} \rceil \right) \rceil$$

We split the analysis to cases.

- If $n = 3k$ for some integer k, then $n/3 = k$ thus $T(n) = 1 + \lceil \log_3(k) \rceil \leq \lceil log_3(k) + 1 \rceil = \lceil \log_3(3k) \rceil = \lceil \log_3(n) \rceil$.
- If $n = 3k + 1$ for some integer k then $\lceil \frac{n}{3} \rceil = k + 1$ thus $T(n) = 1 + \lceil \log_3(k + 1) \rceil \leq \lceil \log_3(3k + 3) \rceil$. It is left to show that $\lceil \log_3(3k + 3) \rceil = \lceil \log_3(3k + 1) \rceil$; namely, that there exists some i for which both $3^i < 3k + 1 \leq 3^{i+1}$ and $3^i < 3k + 3 \leq 3^{i+1}$. Set i to be the value for which $3^i < 3k + 1 \leq 3^{i+1}$ holds. Clearly, $3k + 3 > 3k + 1 > 3^i$. Assume for the sake of contradiction that $3k + 3 > 3^{i+1}$. Then we get $k + 1 > 3^i$, thus $k \geq 3^i$ (we are dealing with integers) so $3k + 1 > 3k \geq 3^{i+1}$. Contradiction.
- If $n = 3k + 2$ for some integer k then $\lceil \frac{n}{3} \rceil = k + 1$ thus $T(n) = 1 + \lceil \log_3(k + 1) \rceil \leq \lceil \log_3(3k + 3) \rceil$. We already know that in this case $\lceil \log_3(3k + 3) \rceil = \lceil \log_3(3k + 1) \rceil$ and since obviously $\lceil \log_3(3k + 1) \rceil \leq \lceil \log_3(3k + 2) \rceil \leq \lceil \log_3(3k + 3) \rceil$ then $\lceil \log_3(3k + 2) \rceil = \lceil \log_3(3k + 3) \rceil = \lceil \log_3(n) \rceil$.

Your answer need not get into this level of specification, but we do require the upper bound $\lceil \log_3(n) \rceil$. We will accept solutions that assumes n is a power of 3.

(b) (15 pts) Prove that your algorithm is optimal. That is, if the algorithm that you design makes in the worst-case w weighs, then show that any other algorithm that is guaranteed to find the prize-box must also make w weighs in the worst-case.
Answer. We claim that any algorithm that finds the one cereal box that is heavier than any other box must make $\lceil \log_3(n) \rceil$ weighs.

To prove the claim, look at the decision tree of any algorithm. At any node we split the set into 3 disjoint subsets: the boxes on the right side of the scale, the boxes on the left side of the scale, and the boxes left out of the weighing. The weighing reveals which of the 3 subsets contains the prize-box, hence each node has 3 children. The leafs of the decision tree point to the prize-box, so there are n leafs. It follows that the longest path root→leaf must contain $\lceil \log_3(n) \rceil$ nodes, since otherwise all root→leaf paths contain $< \log_3(n)$ nodes, allowing for $< n$ leafs in a complete tri-nary tree.

We therefore deduce that for any weighing-based algorithm there must exist some input on which it makes at least $\lceil \log_3(n) \rceil$ weighs. Thus, our above-mentioned algorithm is optimal. ■

One can also prove the same argument using contradiction: ASOC that there was some algorithm making w weighs *in the worst-case*, where $w < \log_3(n)$, and is guaranteed to find the heavy prize-box. Each weighing results in one of three situations: the RHS of the scale is heavier than the LHS, the LHS is heavier than the RHS, or both sides are equal. Therefore, for a deterministic algorithm, the w queries along their answer fully determine the output of the algorithm. In other words, the output of the algorithm is determined by the 3^w possible sequences of query-answers. Since $w < \log_3(n)$ then $3^w < n$, and so — by the pigeonhole principal — there must exists two distinct inputs (the prize is in the ith box, the prize is in the jth box for some $i \neq j$) on which that algorithm must see the exact same sequence of query-answer and thus map them to the same output. Therefore, the algorithm must err on at least one of these two inputs. We infer that $w \geq \log_3(n)$. Since w is an integer it follows that $w \geq \lceil \log_3(n) \rceil$. ■

Note: Ideally, your upper- and lower-bound will match *exactly* (or up to ± 1). If you can't get them to match exactly, have your bounds be asymptotically the same.

Problem 4. (25 pts) Various Sorting Problems.

(a) (10 pts) Propose an efficient algorithm for the following problem. The input is composed of a heap A of size n, and an additional element x. The algorithm's objective is to return (say, print to the screen) all elements in A that are greater than or equal to x (note that x is not necessarily in A). The running time of your algorithm should be $O(k)$ where k is the number of elements reported by the algorithm. Prove the correctness and runtime of the algorithm.

Answer. The algorithm will be a recursive traversal of the tree (heap). At each node, since we know that the largest element of that subtree is at the root, by comparing the root with x we can determine whether we must explore that subtree or not. If the root of that subtree is smaller than x then all the elements of that subtree are smaller than x. Otherwise, we will print the root and recursively check each of the subtrees rooted at the children of the current node. Here is the psuedocode (we call it with $i = 1$):

```
Report-All(A, i, x)
    ** printss all the elements ≥ x in the heap rooted at A[i].
if (A[i] ≥ x) then
    Print(A[i])
    lc ← leftchild(i)
    rc ← rightchild(i)
    if (lc ≤ heapsize(A)) then
        Report-All(A, lc, x)
    end if
    if (rc ≤ heapsize(A)) then
        Report-All(A, rc, x)
    end if
end if
```

The correctness of the algorithm is easy. Assume for the sake of contradiction that there exists some element $A[j] \geq x$ which we have not printed. This can happen only if `Report-All` was never invoked on j. Thus, along the path from j to the root of the heap there was some j' for which $A[j'] < x$. But this implies $A[j'] < A[j]$, which means A isn't a heap — in the subheap rooted at j', the root *isn't* the largest element. Contradiction.

To argue that we take $O(k)$ time to print, we denote a function $T(k)$ which is the runtime of the algorithm on a rooted heap with exactly k elements greater than x. Clearly, $T(0) = c$ for some constant. Moreover, $T(k) = c' + T(k_1) + T(k_2)$ where c' is some other constant, and k_1 and k_2 denote the number of elements greater than x is the left and in the right sub-heaps of the root. Clearly, $k_1, k_2 \geq 0$ and $k_1 + k_2 \leq k - 1$. Denoting $C = \max\{c, c'\}$, we can prove using induction that $T(k) \leq C \cdot (2k + 1)$ which results in $T(k) \in O(k)$.

Base case: For $k = 0$ have $T(0) = c \leq C = C(2 \times 0 + 1)$.

Induction step: Fix some k. Assuming the claim holds for any $0 \leq i < k$ we show it holds for k. Indeed:

$$\begin{aligned} T(k) \quad &= c' + T(k_1) + T(k_2) \\ &\leq C + C \cdot (2k_1 + 1) + C \cdot (2k_2 + 1) \\ &= C \cdot (2(k_1 + k_2) + 3) = C \cdot (2(k - 1) + 3) = C \cdot (2k + 1) \quad \blacksquare \end{aligned}$$

(b) (15 pts) Given a BST T, let us add a field *size* that keeps track of the number of nodes in T.

1. (2pt) How would you alter `Insert()` to update the size of the tree (and all its subtrees) when we add a new node to T, in a way that doesn't increase the runtime of `Insert()`?

   Answer. Here is one way — and there are others. The new tree is created with $size = 0$ and we invoke the following code.

```
NewInsert(T, key) ** Inserts into T a new leaf holding key
Insert(T, key) ** standard BST-insert
T ← Find(T, key)
while (T ≠nil) do
    T.size ← T.size + 1
    T ← T.root.parent
```

Note that if standard `Insert()` places the new leaf at level l, this means that it took $O(l)$-time to insert the new node; but then finding the leaf holding *key* will also take $O(l)$-time, and the `while`-loop will also iterate $O(l)$ times. Therefore, if the original insertion takes $O(l)$ — then so is the insertion.

Other technique could be to insert the new tree with $size = 1$ and change the code of `Insert()` so that upon a recursive call (whether we turn left or right) we afterward increment $T.size$. This increments in turn the *size* field of all ancestors of T.

2. (3pt) How would you alter `Delete()` to update the size of the tree (and all its subtrees) when we remove a node from T, in a way that doesn't increase the runtime of `Delete()`?

   Answer. Here we need to alter the `DeleteTree()` code. Once we invoke a deletion that actually removes a node from the BST, we have to travel from that node through all of its ancestors, up to the root, and decrement their *size*-fields. Thus, before the code calls an actual `delete` T (after it connects T's parent with T's single-child), we have to invoke the following lines:

```
T' ← T
while (T' ≠nil) do
    T'.size ← T'.size − 1
    T' ← T'.root.parent
delete T
```

   Note that yet again, if we ended up deleting a node on the l-th layer, then the deletion (including with the initial `Find()`) costs $O(l)$, so climbing from that node through all of its ancestors and up to the root — to decrement each ancestor's *size* field — takes $O(l)$ in total.

3. (10pt) Write a code (that uses the *size* field) for the quantile problem: given a BST T and a value v, `Quantile(`T, v`)` should return the number of elements in T which are $\leq v$. Argue the correctness of your code, and justify its runtime.

   Answer. Our code is as follows:

```
Quantile(T, v)
if (T =nil) then
    return 0
else if (T.root.key > v) then
    return Quantile(T.root.left, v)
else ** T.root.key ≤ v
    return T.root.left.size + 1+Quantile(T.root.right, v)
```

   We argue correctness of the code using induction on the tree's height.

   Base case: height $= -1$ (the empty tree), in which case no element in T is upper bounded by v, so the answer is 0.

You can also argue the base case for a tree of height 0 (a single node) — both right and left subtree's are empty so we either return 0 (if the root is greater than v) or return 1 (if the root is $\leq v$) — as we should.

Induction step. Fix $h \geq 0$. Assuming correctness to all tress of height $\leq h$ we show correctness for all trees of height $h+1$. Fix T to be any tree of height $h+1$.
Case 1: $T.root > v$. Therefore, any key stored in the tree which is $\leq v$ must also be $< T.root$ and thus belong to T's left subtree. Therefore, all the keys that are $\leq v$ are stored in the left subtree, whose height is $< h+1$, and so `Quantile(`$T.root.left$`)` returns the number of such keys.
Case 2: $T.root \leq v$. Therefore, any key in the left subtree of the root is $\leq T.root \leq v$ and so the set of keys that are in T which are $\leq v$ can be partitioned into three subsets: (i) the root (a singleton set), (ii) all the keys in the left subtree, (iii) all the keys in the right subtree of the root which are $\leq v$. The size of the first set is 1; the size of the second set is stored in the field $T.root.left.size$, and by IH `Quantile(`$T.root.right, v$`)` returns the size of the third set as the height of the right-subtree of the root is $< h+1$. Therefore, we return the correct size of the set of all keys in T which are $\leq v$. ■

Not surprisingly, since at every level of the recursion we do $O(1)$ work and after $O(h)$ iteration we must reach the base case of the recursion, our code runs in $O(h)$-time. We can also see is using the recursion relation $T(-1) = 0$ and $T(h) = T(h-1) + 1$, which solves to $h + 1 \in O(h)$.

Note how similar are the `Select()` problem and the `Quantile()` problem (except for the fact you were asked to solve the `Quantile()` problem only on BST, but we could define a similar problem for an array of numbers: `Quantile(`A, v`)`= #elements in A which are $\leq v$.)