CSCI 270 Homework #2 Solutions

1. Write your name, student ID Number, and which lecture you attend (morning or afternoon). Multi-page submissions must be stapled.

2. Aaron is trapped on an island with hungry velociraptors. He is currently at a safe location s, and he wants to get to the boat t which will get him home. There are many paths through the island, represented as a directed graph. Each edge e has a probability p_e that a traveler along this edge will be eaten by a raptor. If we select a path P from s to t, the probability he arrives safely is $p_{safety} = \Pi_{e \in P}(1 - p_e)$. Give an efficient algorithm which maximizes the probability he arrives safely. *Note: Please do not feed your instructor to hungry dinosaurs.*
   Solution:
   At each node, store the maximum probability you can reach that node without getting eaten. Initially set R(s)=1, and all other nodes set to 0 (since you have not found a path to them yet).

   Maintain an explored set of nodes, wherein you have settled upon the maximum probability and will not change this value in the future.

   Find the unexplored node x with highest $R(x)$ value, and add it to the explored set. Iterate over all outgoing edges (x, y) from x, and update $R(y)$ to be $\max(R(y), R(x) \cdot (1 - p_{(x,y)}))$. Repeat this step until you explore t. At this point, $R(t)$ is the maximum probability possible to escape, so return this value.

   Note that this is merely a reskin of Dijkstra's algorithm: you have a slightly different method of choosing "distance" from s, but the template of the algorithm is the same. If you use a MaxHeap to store the R values, this algorithm works in $\Theta(m \log n)$ time.

   As a more concise solution, note that maximizing $\Pi_{e \in P}(1-p_e)$ is the same thing as maximizing $\sum_{e \in P} \log(1 - p_e)$, which is the same thing as minimizing $\sum_{e \in P} - \log(1 - p_e)$. So you could change all edge weights to be $-\log(1-p_e)$ and then run Dijkstra's algorithm on the resulting graph.

3. Suppose that I pick an integer x between 1 and n. Your job is to identify x using the fewest number of "yes/no" questions possible. You may select any subset of the numbers between 1 and n, and ask if x is within that subset.

   (a) Explain how to identify x using the smallest number of queries.
   Solution:
   Do a binary search. Ask if the number is $\leq \frac{n}{2}$, use the answer to divide the problem space in half, and repeat until you have identified the number.

   (b) Analyze the number of queries needed for your solution to part a, by setting up and solving a recurrence relation.
   Solution:
   $f(n) = f(\frac{n}{2}) + c$.
   $a = 1, b = 2, g(n) = c, n^{\frac{\log a}{\log b}} = 1$, so by Master Theorem, the runtime is $\Theta(\log n)$.

   (c) Suppose that I am a particularly sadistic instructor, and that I am allowed to lie exactly once. You can still solve this problem by simply asking each question twice (and a third

time when you detect the lie). Analyze the number of queries needed in this strategy.
Solution:
We are simply multiply the total number of queries by 2, and adding 1 to the result. So the total number of queries is still $\Theta(\log n)$.

(d) Assuming I am still allowed to lie once, explain how we can eliminate $\frac{1}{4}$ of the possible numbers using only two queries.
Solution:
Question 1 would be "is the number $\leq \frac{n}{2}$?"
Question 2 would be "is the number between $\frac{n}{4}$ and $\frac{3n}{4}$?
If the answer to both questions is yes, both questions have reinforced that it is not $> \frac{3n}{4}$, so eliminate that quarter.
If the answer to both questions is no, both questions have reinforced that it is not between $\frac{n}{4}$ and $\frac{n}{2}$, so eliminate that quarter.
If only the first question is answered yes, both questions have reinforced that it is not $> \frac{n}{2}$ and $\leq \frac{3n}{4}$, so eliminate that quarter.
Finally, if only the second question is answered yes, both questions have reinforced that it is not $< \frac{n}{4}$, so eliminate that quarter.

(e) Analyze the number of queries needed using your solution to part d, by setting up and solving a recurrence relation. Which solution (part c or part d) is better overall?
Solution:
The recurrence would be $f(n) = f(\frac{3n}{4}) + c$.
$a = 1, b = \frac{4}{3}, g(n) = c, n^{\frac{\log a}{\log b}} = 1$, so by Master Theorem, the runtime is $\Theta(\log n)$. Therefore, the difference between these two algorithms is a constant factor, and neither is better using the framework of this class.
As a more complete answer, part c asks 2 questions to reduce the problem in half, and part d asks 2 questions to reduce the problem by $\frac{1}{4}$, so it seems clear that the solution to part c will have better constants and thus be better overall, even though the asymptotic analysis is identical.

4. Suppose we have an array A allocated in memory, but we do not know the size of this array (and there is no member variable or function of A which tells us the size of the array). We can query the contents of $A[i]$ in constant time, but if i is greater than the bounds of the array, we will simply get back the unhelpful error: "index out of bounds." If we want to identify the size of the array, we could query the contents of each index until we find the first error, but this would take $\theta(n)$ time, where n is the (currently unknown) size of the array. Give a more efficient algorithm to determine the size of the array, and analyze the running time in terms of n.
Solution:
Query index $A[2^i]$ for $i = 0, 1, 2, ...$ until you find the first out-of-bounds index k. This step takes $\Theta(\log n)$ time.
You now have a possible range of $A[\frac{k}{2}..k]$, so search for the first out-of-bounds index using binary search.
If the current index is out-of-bounds, search the left-half of the array. If the current index returns a value, search the right-half of the array.
As a base case, when you have a single value $A[x]$ left to search, if $A[x]$ is out-of-bounds, then

return $x - 1$, otherwise return x.
The recurrence relation for the above step is $f(n) = f(\frac{n}{2}) + c$, which by Master Theorem has a solution of $\Theta(\log n)$. Therefore, the total runtime for this algorithm is $\Theta(\log n)$.

5. Suppose we are given all the elements we wish to insert into an initially empty MinHeap up front. If we insert one at a time, it takes $O(n \log n)$ time to build the MinHeap. Suppose we instead build a "pre-heap" of depth d by forming a complete binary tree with each node in an arbitrary position. We must "heapify" this pre-heap so that each node has value $\leq$ its children.

   (a) Suppose we take the subtrees of size ≤ 3 rooted at depth $d - 1$. Show how to heapify **all** of these subtrees in $O(n)$ total time.
   Solution:
   Each subtree of size ≤ 3 can be heapified in constant time by finding the smallest element and moving it to the "root". Since there are $\Theta(n)$ different subtrees, all of them can be heapified in $\Theta(n)$ time.

   (b) Assuming that all subtrees rooted at depth j are valid MinHeaps, show how to heapify **a single** subtree rooted at depth $j - 1$ in $O(d - j)$ total time.
   Solution:
   The subtree rooted at the left child, and the subtree rooted at the right child, are both heaps. The root may be in the wrong place however. Use the PercolateDown function to move the root to the correct place. That is, compare the root to its two children: if it is smaller than both, return. Otherwise, swap the root with the smaller child. The smaller child is in the correct place, but the root may need to move further downward. In the worst case, the root will percolate all the way to the bottom of the tree, which has depth $\Theta(d - j)$. Therefore, this is how many comparisons will be needed, and the total runtime is $\Theta(d - j)$.

   (c) Analyze the running time to build a heap, using the inductive design process outlined in parts a and b. *You may find the following summation useful:* $\sum_{i=1}^{n} \frac{i}{2^i} = O(1)$.
   Solution:
   There will be a total of $\log n$ levels.
   The top level has a single node which will take $c \log n$ time to heapify, assuming the subtrees are MinHeaps.
   The second level has two nodes, each will take $c((\log n) - 1)$ time to heapify.
   Each successive level, the number of nodes double, but the time to heapify each subtree decreases by 1 comparison.
   Looking at it from the other direction, the bottom level will have $\frac{n}{2}$ trees to heapify, but it takes c time for each one, the previous level will have $\frac{n}{4}$ trees to heapify, but it takes $2c$ time for each one, etc.
   The total work can thus be summed up as $\sum_{i=1}^{\log n} \frac{n}{2^i} \cdot ic = cn \sum_{i=1}^{\log n} \frac{i}{2^i}$. Using the hint, the sum is merely a constant, so the total runtime is $\Theta(n)$. That is, we can create a heap in LINEAR time!