Homework #4

You should try to solve these problems by yourself. I recommend that you start early and get help in office hours if needed. If you find it helpful to discuss problems with other students, go for it. **You do not need to turn in these problems. The goal is to be ready for the in class quiz that will cover the same or similar problems.**

Problem 1: Glass Jars

You have been asked to do some testing of a model of new glass jars to determine the maximum height at which they can be dropped without breaking. The setup for your experiment is as follows. You have a ladder with n rungs. You need to find the highest rung from which you can drop one of the jars without it breaking. We'll call this the *highest safe rung*.

Intuitively, you might try a binary search. First, drop the jar from the middle rung and see if it breaks. If it does, try run $n/4$; if not, try rung $3n/4$. But this process can potentially break a lot of jars. If your primary goal were to break as few jars as possible, you might start at rung 1. If the jar doesn't break, you move on to rung 2. You're guaranteed to break only one jar, but you may have to do a lot of dropping if n is very large.

To summarize, you have to trade off the number of broken jars for the number of drops. To understand this tradeoff better, consider how to run the experiment given a budget of $k \geq 1$ jars. Your goal is to determine the correct answer (i.e., the *highest safe rung*) using at most k jars.

(a) Suppose your budget is $k = 2$ jars. Describe an approach for finding the *highest safe rung* that requires at most $f(n)$ drops for some function $f(n)$ that grows slower than linearly. (In other words, it must be true that $\lim_{n\to\infty} f(n)/n = 0$.) This means you cannot just start at the bottom rung and work up.

Solution

Suppose (for simplicity) that n is a perfect square. We drop the first jar from heights that are multiples of $\sqrt{n}$ (i.e., from $\sqrt{n}, 2\sqrt{n}, 3\sqrt{n}$, etc.) until it breaks. If we drop it from the top rung and it survives, then we're done. Otherwise, suppose it breaks when dropped from height $j\sqrt{n}$. Then we know the highest safe rung is between $(j-1)\sqrt{n}$ and $j\sqrt{n}$, so we drop the second jar from run $1 + (j-1)\sqrt{n}$ on upward, going up by one rung each time. In this way, we drop each of the two jars at most $\sqrt{n}$ times, for a total of at most $2\sqrt{n}$. If n is not a perfect square, then we drop the first jar from heights that are multiples of $\lfloor\sqrt{n}\rfloor$ and the apply the above rule for the second jar. In this way we drop the first jar at most $2\sqrt{n}$ times (quite an overestimate if n is reasonably large) and the second jar at most $\sqrt{n}$ times, still obtaining a bound of $O(\sqrt{n})$ on the number of drops.

(b) Now suppose your budget is $k > 2$ jars, for some given k. Describe an approach for finding the *highest safe rung* using at most k jars. If $f_k(n)$ is the number of times you need to drop a jar according to your strategy, then the functions $f_1, f_2, f_3, \ldots$ should have the property that each grows asymptotically slower than the previous one, i.e., that it is true that $\lim_{n\to\infty} f_k(n)/f_{k-1}(n) = 0$ for each k.

Solution

We claim by induction that $f_k(n) \leq 2kn^{1/k}$. We begin by dropping the first jar from heights that are mutliples of $\lfloor n^{(k-1)/k} \rfloor$. In this way, we drop the first jar at most $2n/n^{(k-1)/k} = 2n^{1/k}$ times, and thus narrow the set of possible rungs down to an interval of length at most $n^{(k-1)/k}$. Wen then apply the strategy for $k-1$ jars recursively. By induction, this uses at most $2(k-1)(n^{(k-1)/k})^{1/(k-1)} = 2(k-1)n^1/k$ drops. Adding in the $\leq 2n^{1/k}$ drops made using the first jar, we get a bound of $2kn^{1/k}$, completing the induction step.

Problem 2: Heaps

(a) Devise an algorithm for finding the k smallest elements of an unsorted set of n integers in $O(n + k \log n)$.

Solution

Build a min heap (this takes $O(n)$ time). Then call extract-min k times to get the k smallest elements. Each call to extract min takes $O(\log n)$ time; we do it k times. In total, this is $O(n + k \log n)$ time.

(b) Give an $O(n \log k)$ time algorithm that merges k sorted lists with a total of n elements into one sorted list.

Solution

Create a min-heap containing pairs (value, listID), keyed on values. The value is the value of each element, and the listID is which of the k lists the element came from. Call extract-min on this heap to get the smallest element of all of the lists. Put it in your sorted list. Get the next smallest element from the list with the listID of the element you just extracted and put it in your min heap (if listID is empty, just skip this step). Call heapify. Repeat for all n elements. The running time to create the min heap is $O(k)$. The time for each cycle of extract-min and heapify with its replacement takes $O(\log k)$ (since there are at most k elements in the min heap at any time. We do this n times, for a total of $O(n \log k)$.

Problem 3: Heaps

Give an efficient algorithm to find all keys in a min heap that are smaller than a provided value X. The provided value *does not* have to be a key in the min heap. Evaluate the time complexity of your algorithm.

Solution

Start from the root of the heap. If the value of the root's key is smaller than X, the print the root's key and call the procedure recursively on the left child and on the right child. If the value of the root node is greater than X, then all of its children's keys will be greater than X, so we can safely not print the root's key AND not make any recursive calls.
This algorithm has a time complexity of $O(n)$, where n is the number of elements in the heap. This bound is reached in the worst case, in which every element of the heap has a key smaller than X, so every key will be printed.

Problem 4: Graph Representations

(a) What is the time complexity of Breadth First Search with an adjacency list representation? What about adjacency matrix representation? Why?

Solution

Adjacency list is $O(|V| + |E|)$ because all vertices are visited, and each edge is evaluated for the search. Adjacency matrix is $O(|V|^2)$ because you must go down the entire matrix column (or rather half the column) to evaluate all outward edges from that vertex. Note that in highly connected graphs $O(|E|)$ converges to $O(|V|^2)$ thus the representations converge to the same time complexity as connectivity increases. This shows that that adjacency lists are typically better for searching sparse graphs.

(b) What is the time complexity of Depth First Search with an adjacency list representation? What about adjacency matrix representation? Why?

Solution

The answer is the same as above. Depth First and Breadth First both require evaluating the graph exhaustively.