Problem Set #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 these problems in. The goal is to be ready for the in class quiz that will cover the same or similar problems.**

Problem 1: Algorithms and Decision Trees

You are given 9 identical looking balls and told that one of them is slightly heavier than the others. Your task is to identify the defective ball. All you have is a balanced scale that can tell you which of the two sides is heavier or if the two sides weigh the same.

Give a decision tree lower bound showing that it is not possible to determine the defective ball in fewer than 2 weighings.

Solution

Any decision in this tree has three possible outcomes: the compared sets weigh the same; set A is heavier or set B is heavier. There are 9 possible outcomes (any one of the balls can be the heavy one. Any tree with branching factor 3 has at least 3^h leaves. So we set 3^h equal to 9 and solve for h. Take the log of both sides; $h = 2$. Thus to be able to reach any of the 9 possible outcomes, I must make at least two weighings.

Problem 2: Decision Tree Lower Bounds

Alan's toolbox is a mess. He has lots and lots of bolts of all different sizes. He keeps them in bins, and they are a little bit sorted within the bins. For example, we know that all of the bolts in the first bin are smaller than any of the rest of the bolts. We know that all of the bolts in the second bin are smaller than all of the rest of the bolts, except those in the first bin. And so on.

Let's think about a way to help him get organized. Let's say he has n bolts. They are divided into bins as described above such that each bin contains exactly k bolts (i.e., there are n/k bins), and the bolts in a given bin are all smaller than the bolts in the succeeding bin and larger than the bolts in the preceding bin.

(a) Show an $\Omega(n \log k)$ lower bound on the number of comparisons needed to put all n bolts in total order.

Solution

We construct a decision tree for this variant of the sorting problem and show that it has height at least $n \log k$. Each leaf of the tree corresponds to a permutation of the original sequence. Each subsequence has $k!$ permutations, and there are n/k subsequences, so there are $(k!)^{n/k}$ permutations of the whole sequence. Thus the decision tree has $(k!)^{n/k}$ leaves. A binary tree with $(k!)^{n/k}$ leaves has height at least $\log((k!)^{n/k})$.

$$\begin{aligned} \log((k!)^{n/k}) &= n/k \log(k!) \\ &= \Theta(n/k * k \log k) \\ &= \Theta(n \log k) \end{aligned}$$

Therefore the lower bound on the number of comparisons needed to solve this variant of the sorting problem is $\Omega(n \log k)$.

(b) Design an optimal algorithm for me to use to sort Alan's bolts.

Solution

Merge-sort each of the subsequences. And then just concatenate them. Each merge-sort takes $O(k \log k)$. We have to do n/k such merge-sorts. Concatenation could be constant or take as much as $O(n)$ depending on storage. Therefore, we get $n/k * O(k \log k) + O(n)$ in the worst case. This is $O(n \log k) + O(n) = O(n \log k)$.

Problem 3: 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 rung $n/4$; 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 4: 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 5: Priority Queues

One of your classmates claims to have developed a new data structure for priority queues (other than using a heap) that supports the standard priority queue operations INSERT, MAXIMUM, and EXTRACT-MAX all in constant ($O(1)$) time in the worst case. Prove that he is mistaken. (Hint: this is not a lengthy proof; think about the implications of such a claim on the efficiency of sorting.)

Solution

If this were true, it would mean that we could sort in $O(n)$ time. That is, we would insert n elements into his fancy priority queue (that takes $n*O(1) = O(n)$ time). Then we would call extract max n times to get them back out in sorted order (that also takes $n * O(1) = O(n)$ time). That would contradict our $O(n \log n)$ lower bound on sorting. If he were using something other than comparisons to determine priority, it is possible, but assume he uses comparisons and is lying.