Many Campers Sort Piles | Instructor: Padraic Bartlett

Lecture 3: Randomness and Sorting

Week 4 | *Mathcamp 2012*

Randomization is a tool we've already used in our sorting algorithms; for example, bogosort works[1] by randomly shuffling the elements of a list until it accidentally puts all of the elements in order! However, this was kind of a "dumb" example of using randomness in algorithms; in practice, bogosort is an absolutely horrible way to sort a list.

However, as it turns out, this was not the fault of randomness! In fact, randomization is an incredibly useful tool in designing algorithms; throughout this talk, we will study several examples of how randomization can give us fast and stable ways to perform otherwise tricky tasks.

1 A Motivating Example

Consider the following problem.

Problem. Take a list L of n elements, of which half are the symbol 'a' and the other half are the symbol 'b'. We want to quickly find a copy of the symbol a.

There are many obvious deterministic (i.e. nonrandom) algorithms for finding such an element: a fairly obvious one is to simply start at the first element in your list, check if it's an a, and continue down the list until you get to an a. This algorithm will need to perform $\frac{n}{2}$-many checks in the worst case (where it sees the list "$bb \dots bbaa \dots aa$.")

However, we never really should need $n/2$-many steps to find a a; if half of the elements are a, it feels like we should be able to find one relatively quickly! However, if we have any deterministic algorithm for searching our list — i.e. some order $i_1, i_2, \dots i_n$ for looking at the elements of our list L — it's possible that the first $n/2$ elements we look at are all b's. This is relevant in the case that our list is being created by a malevolent user, who knows our algorithm and is deliberately creating "difficult" lists for us to sort! In this situation, we're in trouble: no matter how we make our deterministic algorithm, they can always make a "bad" list.

How can we get around this? In other words, how can we make an algorithm that will typically run quickly, regardless of whether the person giving us our list is "deliberately" trying to give us a bad list?

The solution (as you may have guessed, based on the title of this lecture) is randomness! Specifically, consider the following, very simple, algorithm: repeatedly choose elements at random from our list until we get an a. After k steps, the probability that we've found an a is

$$1 - (1/2)^k;$$

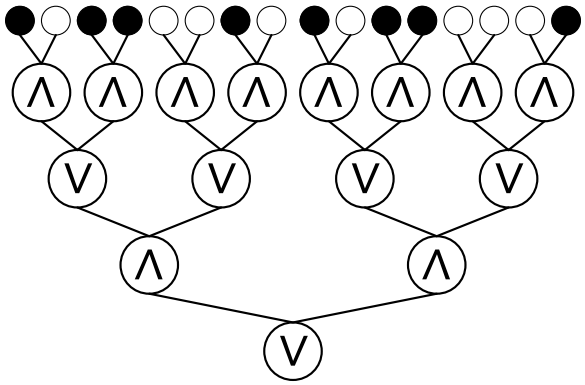
[1] Well, worked is a strong word. It tried.

for values of k around 100, this is so stupidly small that it's far more likely that a software error occurs due to electromagnetical interference than because our algorithm didn't find an a.

This works! Regardless of whether our list is being provided by a malicious adversary or not, this process will find an a extremely quickly. In general, this idea — that randomization allows us to not worry about "malicious" users deliberately creating "bad" lists — is an incredibly useful one in designing practical algorithms.

2 How to Defeat Death via Randomness

As a second application of the technique of randomness, consider the following scenario: one day, Death finds you and challenges you to a game. Death, knowing you're a Mathcamp student, opts to not challenge you to the traditional game of chess, and instead presents you with a following complete binary tree with 4^n leaves:



On each of the 4^n leaves, Death has placed a vial full of either deadly poison (black) or an elixir of infinite youth (white.) The game is as follows:

- Initially, a token is placed at the root node of the tree.
- You and Death then take turns moving the token. On a player's turn, you must move the token from its current location to one of its two children[2] nodes further along the tree.
- You move first: the game ends after $2n$ moves, when the token reaches a final leaf, and you drink whichever elixir the token lands on.

You get to look at the tree before deciding if you want to play or not. Given any such game tree, how can you determine whether or not you should play this game?

One method for determining the outcome of this game is suggested by the labellings of the nodes above. In particular, label the root node "or," the two children node of this root node "and," the children of this node "or" again, and keep alternating until every vertex in this graph is labeled other than the leaf nodes. Then think of the leaf nodes as a bunch

[2]The children of a node in a tree are all of the vertices linked to that node that are further away from the root.

of variables that are either true (white) or false (black): in this interpretation, our tree is just a giant chain of true-false statements linked together with and's and or's. Therefore, it evaluates to some value! Suppose that it's true.

On your first turn, you're starting at an or-statement and looking at two possible child nodes. Because your entire logical statement evaluates to true, one of your two child nodes (because you're at an or) must be true. Go there.

Death then is staring at an "and" choice. Because we chose this to be true, no matter what he chooses, he arrives at a true statement. It is again your turn: again, pick whichever node of the two available to you evaluates to true. Repeat this process! By construction, every node we get to evaluates to true: so, when we reach the end of the tree, we will arrive at a true (i.e. white, i.e. infinite life) leaf.

Otherwise, if this tree evaluates to false, Death can adopt a similar strategy to force you to choose a poison-containing vial. (In this case, you should probably decline, or perhaps instead challenge Death to a game of tic-tac-toe.)

So: this is a method for determining whether this is a good game to play. How long does it take us to make this evaluation? Well: one method we can do is to simply start from the leaf nodes, and condense them pair-by-pair into either true (if they're both true) or false (if one is false, because the second-to-last node is an "and." Repeat this process: this will involve visiting all 4^n nodes, and therefore take 4^n many steps.

Death, having taken an algorithms course of his own, knows this, and therefore challenges you to decide whether to play this game or not in $O(3^n)$ moves instead! Can you still succeed?

Using randomness, it turns out that you (typically) can! Consider the following algorithm (call it treeTest), that will determine if this game is worth playing with an expected number of steps that is roughly $O(3^n)$:

1. Start from the first vertex.

2. Randomly choose one of the two possible children of the vertex we're at. Run treeTest on the tree made by taking that vertex and all of its children-nodes. If we're on a or-node and this run of treeTest comes up true, we're done! Return true!

3. Similarly, if we're at an and-node and this run comes up false, we're done! Return false.

4. Otherwise, we don't yet know where we are. Run treeTest on the tree made by taking the other vertex along with its children, and return whatever truth value that run gives.

When we run this algorithm, we always look at one of our child nodes (chosen randomly) and look at the other only when it's necessary. Intuitively, this means that we roughly look at any given node about $\frac{3}{4}$ of the time: half of the time we pick it, and the other half of the time we maybe have a 50% chance of picking it. (You can make this rigorous, and I'm asking you to do so on the HW!)

3 Quicksort!

Finally, we close by discussing how randomness is used in the implementation of quicksort. Recall, from yesterday, the definition of quicksort: given a list $L = (l_1, \ldots l_n)$,

- Pick an element l_k out of our list. Call this a **pivot** element.
- Take the rest of our list, and split it into two lists: the list $L_<$ of elements less than our pivot, and the list $L_\geq$ of elements that are greater than or equal to our pivot.
- Quicksort the two lists $L_<$ and $L_\geq$.

The difficulty with quicksort implementations is in that word "pick" at the first step. Typically, as Nic discussed yesterday, quicksort takes something like $O(n \log(n))$ steps to sort our list; however, as Nic also mentioned yesterday, there are occasional lists where it takes n^2-many steps to sort the list.

In particular, if we have any deterministic method for picking out elements, like (say) "always choose the first element of our list for the pivot," it's possible that the list we start from will make us take n runs to sort our list. For example, suppose that we started with the list $(1, 2, \ldots n)$:

- At the start, we choose the element 1 as a pivot. We then split our list into two lists, $L_<$ and $L_\geq$: the first list, by definition, is empty, while the second list is $(2, 3, \ldots n)$.
- We now quicksort these two lists. The empty set is trivially sorted! To quicksort the second list, we choose its first element (2) as a pivot, and split it into two lists $L_< = \emptyset$ and $L_\geq = (3, 4, \ldots n)$.
- ... repeat this process!

This clearly takes n passes, and therefore n^2 comparisons, to sort our list. Furthermore, there's no deterministic algorithm that can avoid this problem: given any deterministic algorithm that will pick the $k(L)$-th element out of a list L of n elements, we can put the smallest element of our list at the location $k(L)$, the second-smallest element at the location corresponding to where k would pick an element out of the list $L \setminus$ (smallest element), and so on/so forth. So, we're in the same spot that we were in at the start of lecture: if we have a malevolent user submitting intentionally awful lists, our sorting algorithm may take forever!

Again, the answer to this problem is to use randomness! In particular, suppose that we choose this pivot element randomly. What is the expected number of passes we will have to make through any list?

Well, one way to get a rough upper bound is the following: given a list L, call a pivot element p "good" if it's from the middle-half of the elements of L after sorting, and call it "bad" otherwise. The probability that a random pivot is "good" is clearly $1/2$. Furthermore, if we pick such a good pivot, it will cut our list of n elements into two pieces, one of size at most $3n/4$ and the other of size at least $n/4$; otherwise, if we pick a bad pivot, it's at the least the same as not doing anything (i.e. it leaves us with pretty much the same list.)

Therefore, if we set $T(n)$ to be the expected amount of steps that it will take to random-quicksort a list with n elements, we have just shown that

$$T(n) \leq n + \frac{1}{2}\left(T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right)\right) + \frac{1}{2}T(n).$$

This is because each choice of pivot takes n comparisons to determine how to split our sets, and we've either chosen a good pivot (in which case we've split our sets into two pieces) or a bad pivot (in which case we've not split our sets at all.) Solving for $T(n)$ gives us

$$T(n) \leq 2n + T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right).$$

Using this, we can solve for an upper bound on $T(n)$: in particular, if we guessed (based on yesterday's discussion) that $T(n)$ was something like $an \cdot \log(n)$, we could prove by induction that $T(n) \leq an \cdot \log(n)$. If $a \geq 2$, this is certainly true for $n = 2$, as it takes us at most 2 steps to quicksort a list of two elements (pick a pivot, the other element is either less than or greater than it.)

So: if we inductively assume that it holds for all values $\leq n$, we get

$$\begin{aligned}
T(n) \leq & 2n + \frac{3an}{4}\log\left(\frac{3n}{4}\right) + \frac{an}{4}\log\left(\frac{n}{4}\right) \\
= & 2n + \frac{an}{4}\log\left(\left(\frac{3n}{4}\right)^3\right) + \frac{an}{4}\log\left(\frac{n}{4}\right) \\
= & 2n + \frac{an}{4}\log\left(\left(\frac{3n}{4}\right)^3 \cdot \frac{n}{4}\right) \\
= & 2n + \frac{an}{4}\log\left(\frac{3^3 n^4}{4^4}\right) \\
= & 2n + \frac{an}{4}\log\left(\frac{3^3}{4^4}\right) + \frac{an}{4}\log\left(n^4\right) \\
= & \left(2 + \frac{a}{4}\log\left(\frac{3^3}{4^4}\right)\right)n + an\log(n) \\
\leq & \left(2 + \frac{a}{4} \cdot (-.97)\right)n + an\log(n)
\end{aligned}$$

So, if $a \geq 9$, say, then $1 + \frac{a}{4} \cdot (-.97)$ is less than 0, and therefore we have that this entire quantity is $\leq an\log(n)$, which is what we wanted to prove. So this quicksort has an expected runtime that is $O(n\log(n))$: i.e. up to a constant, it runs like $n\log(n)$! And it's immune to someone intentionally creating "bad" lists, because it's randomized: no matter what list it's fed, it will expect to finish after $\leq 9n\log(n)$ steps.