CS174 Lecture 3 John Canny

Randomized Quicksort and BSTs

A couple more needed results about permutations:

Q1: What's the probability that 4 comes before 5 in a random permutation? Tempting to say $1/2$, but why?

For every permutation where 4 is before 5, there is a matching permutation where 5 is before 4, obtained by swapping 4 and 5. That's a 1-1 correspondence between the two sets of permutations, so they must have the same size, and partition the set of all permutations into two halves. Since all permutations are equally likely, the total probability for the event (4 is before 5) equals the probability of the event (5 is before 4).

Q2: What's the probability that an element u comes before i values $u_1, \ldots, u_i$?

By similar argument, we partition the set of all permutations into subsets S_j according to the ordering of $u, u_1, \ldots, u_i$. Each subset in this partition has the same size (which is $n!/(i+1)!$, the number of permutations of the other elements times the number of choices for the positions of the u's in the whole permutation). Therefore we concentrate on the question of what fraction of subsets S_j have u before the other elements. If u comes first in a permutation of $u, u_1, \ldots, u_i$, then there are $i!$ ways to order the u_i's, while there are $(i+1)!$ permutations total. So the probability that u is first is

$$i!/(i+1)! = 1/(i+1)$$

And this is also the probability that u comes before $u_1, \ldots, u_i$ in the whole permutation.

Randomized Quicksort

Quicksort sorts an array A by partitioning it into subarrays using a pivot element, and recursively sorting the subarrays. In the randomized (Las Vegas) version, the pivot is chosen at random from the subarray. In the pseudo-code below, assume A initially contains n unsorted elements, and that `left=1` and `right=n`.

```
Algorithm Quicksort(A, left, right)
  Choose a random element p from [left,..,right]
  Compare elements with A[p] and
        partition into subarrays < A[p] and > A[p]
  Recursively sort the subarrays < A[p] and > A[p]
```

After partitioning, the array looks like this:

Since the partition element is chosen randomly, the two subarrays are not equally sized. In fact the size of a subarray is a random variable with the uniform distribution from $1, \ldots, n-1$. Nevertheless as we shall see, recursive random partitioning is still efficient, and leads to a logarithmic expected recursion depth.

Analysis of Randomized Quicksort

You may have seen analysis of randomized Quicksort before. If so, it is probably different from the method we use here, which is based on indicator random variables. Unlike "top-down" analyses that use the recursive structure of Quicksort, we will do a "bottom-up" analysis by counting the number of comparisons between array elements that it makes. It should be clear that the running time of both Quicksort and Partition are dominated by the number of comparisons.

Let $A_{(i)}$ denote the i^{th} smallest element in the array. So $A_{(i)}$ is different from $A[i]$ when the algorithm begins, but $A_{(i)} = A[i]$ after the elements have been sorted. Define an indicator random variable X_{ij} as follows:

$$X_{ij} = \begin{cases} 1 & \text{if } A_{(i)} \text{ and } A_{(j)} \text{ are compared during a run of Quicksort} \\ 0 & \text{otherwise} \end{cases}$$

Let's pause here to think about the probabilistic setup. The sample space now is executions of the Quicksort algorithm for a particular input. Each sample point is characterized by the set of random numbers from the random number generator during the entire execution of the algorithm. The variable X_{ij} has its domain on that sample space.

Now notice that as we have described Quicksort, no two elements will be compared twice. That's because one of the two elements being compared is always the current pivot. The first time two elements are compared, whichever one is the pivot is "used up" and doesnt appear in either of the subarrays. Thus it has no more comparisons with the elements in those subarrays.

Let the total number of comparisons in a run of Quicksort be the random variable X, defined as:

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

and we can once again invoke the linearity of expected value to derive the expected number of comparisons for randomized Quicksort:

$$\mathrm{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

Let p_{ij} be the probability that $A_{(i)}$ and $A_{(j)}$ are compared in an execution of Quicksort. Since X_{ij} is 0-1 valued, we have

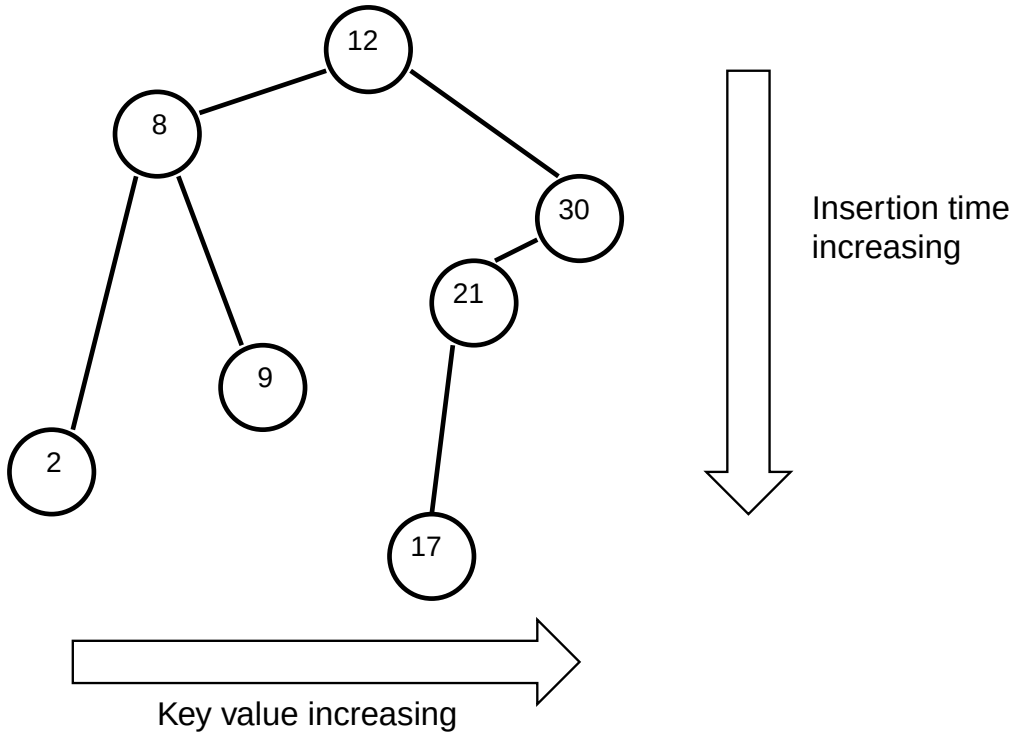
$$\mathrm{E}[X_{ij}] \;=\; 0 \times (1 - p_{ij}) + 1 \times p_{ij} \;=\; p_{ij}$$

So we can complete our analysis by computing p_{ij}, the probability that $A_{(i)}$ and $A_{(j)}$ are compared in a execution of Quicksort. The derivation is not obvious, and it will help to draw a connection between randomized Quicksort and randomized Binary Search Trees (BSTs).

Randomized Binary Search Trees

Suppose we take the n elements in the array A and build a binary search tree from them by inserting one element at a time. Assume there is no re-balancing of the tree. We can perform an inorder traversal of the resulting tree, and the elements will occur in ascending order of key value. Call this procedure "TreeSort".

The root of the BST will be the first element we add. Its left child will be the first element we add which is less than the root etc. The vertical order in the tree encodes the *time* of insertion. The left-to-right order encodes ascending key size. Every node was inserted later than all its ancestors, and earlier than all its descendents. The tree itself doesnt specify the insertion order completely, because it doesnt give the order between siblings. But any complete vertical ordering that keeps every node below its parents is a valid possible order for the sequence of insertions. See the figure below:



The insertions in this tree happened in this order: 12, 8, 30, 21, 9, 2, 17. Vertical position encodes time of insertion. Left-right position is determined by the key value.

The BST will be a randomized BST if we insert the elements in random order. Or equivalently, we could apply the random permutation algorithm from last time to "unsort" the elements, and then insert them one at a time. Every permutation of the elements specifies a unique vertical order. That vertical order, together with the relative size of the elements determines what the binary tree is. Suppose we want to analyze the running time to insert all the elements. Let X be the running time of randomized TreeSort. The running time is dominated by the number of comparisons between elements. Let $A_{(i)}$ denote the i^{th} smallest element in the set. Then define

$$X_{ij} = \begin{cases} 1 & \text{if } A_{(i)} \text{ and } A_{(j)} \text{ are compared during a run of TreeSort} \\ 0 & \text{otherwise} \end{cases}$$

And once again $\mathrm{E}[X_{ij}] = p_{ij}$ where p_{ij} is the probability that $A_{(i)}$ and $A_{(j)}$ are compared during an execution of TreeSort. Determining p_{ij} requires a powerful and non-obvious idea. This idea is useful in analysis of several other algorithms. It involves looking at both time ordering and key ordering of a set of elements. And it will bring us back to random permutations:

Notice that $A_{(i)}$ and $A_{(j)}$ are compared if and only if one element is an ancestor of the other in the BST. The converse is that they share a lowest common ancestor $A_{(k)}$ distinct from $A_{(i)}$ or $A_{(j)}$. Both elements will be compared with this ancestor, but not with each other.

Now our sample space is the set of initial random orderings of the elements. Each order is specified by a permutation π of $\{1, \ldots, n\}$. Assume $\pi(i)$ is the position of element $A_{(i)}$ in the sequence of insertions. If $A_{(i)}$ and $A_{(j)}$ have a distinct lowest common ancestor $A_{(k)}$, then they must lie in distinct subtrees with $A_{(k)}$ as their root. That means that $A_{(k)}$'s key value must lie between the keys of $A_{(i)}$ and $A_{(j)}$. Since our subscripts refer to the elements in sorted order, that means that $i < k < j$. Such an $A_{(k)}$ will be an ancestor of $A_{(i)}$ and $A_{(j)}$ in the tree if and only if it was inserted before them. That is, if and only if $\pi(k) < \pi(i)$ and $\pi(k) < \pi(j)$.

So we have reduced the determination of p_{ij} to a straightforward question about random permutations: Given $i < j$, what is the probability that there is no k such that $i < k < j$ and $\pi(k) < \min(\pi(i), \pi(j))$? If there is no such k, then either $\pi(i)$ or $\pi(j)$ is first (smallest) among $(\pi(i), \ldots, \pi(j))$. That is precisely the question we opened the lecture with. The probability that $\pi(i)$ is first among $(\pi(i), \ldots, \pi(j))$ is $1/(j - i + 1)$. Only one of $\pi(i)$ or $\pi(j)$ can be first so the total probability that one of them is first is

$$2/(j - i + 1) = p_{ij}$$

We can now complete the analysis of randomized TreeSort. Its expected running time is:

$$\mathrm{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 2/(j - i + 1)$$

making the change of variables $l = j - i + 1$, we obtain:

$$\mathrm{E}[X] = \sum_{i=1}^{n} \sum_{l=2}^{n-i+1} 2/l$$

to simplify the second sum, we simply extend it:

$$\mathrm{E}[X] \leq \sum_{i=1}^{n} \sum_{l=2}^{n} 2/l = 2 \sum_{i=1}^{n} H_n = 2nH_n$$

where H_n is the n^{th} harmonic number as before. Since H_n is well-approximated by $\ln n$, our running time bound is close to $2n \ln n$. So we have:

Theorem
The expected number of comparisons for randomized TreeSort of n elements is bounded by

$$2nH_n \approx 2n \ln n$$

and its expected running time is $O(n \log n)$.

Corollary
The expected number of comparisons to insert one element during randomized TreeSort is at most $2H_n \approx 2 \ln n$, and the expected time is $O(\log n)$.

If we maintained a perfectly balanced binary tree, the number of comparisons would be close to $\log_2 n$. Using the formula for change of base of logs ($\log_a(n) = \log_b(n) \log_a(b)$) we find that

$$2 \ln n = (2 \ln 2) \log_2 n \approx 1.386 \log_2 n$$

So a random BST has an expected number of comparisons for insertion (which is also the average depth of the tree) which is only 40% greater than a perfectly-balanced tree. Given the overhead for rebalancing trees which is often larger than 40%, randomized BSTs are quite attractive.

Back to Quicksort

You have probably been wondering about the connection between the BST and Quicksort. It is quite direct. As Quicksort runs, it defines a BST consisting of pivots. The first pivot chosen is the root. The pivot of the left subarray is its left child, and the pivot of the right subarray is the right child, and so on recursively. Convince yourself that this tree is a BST, that is, inorder traversal of it reads the elements in ascending order. This should help clarify why the analyses were so similar. With the pivot as the root of a subtree of the BST, its easy to see why elements in randomized Quicksort are only compared with their ancestors and descendents.

What's not so easy to see is that the random choices in Quicksort give the same outcomes as the random permutation of the elements in TreeSort. In fact, the random choices in Quicksort do *not* specify a unique time ordering of insertions. That's because we never specify in which order we choose the pivots in the two subtrees during Quicksort. But we can reconcile this lack of information by going the other way, showing that Quicksort's choices *could* have been derived from a permutation, but by throwing some information away.

So its enough to show that Quicksort's pivot choice would be the same if it had been based on a complete permutation $\pi(i)$ of all the elements. Let π be such a permutation, and construct a BST from π by inserting elements in the order specified by π. Pick any node $A_{(u)}$. Then the elements $A_{(v)}$ to the left of $A_{(u)}$ satisfy $v < u$. In the BST, the root of the left subtree will be the next element in the permutation from among elements in the left subtree. That is, the $A_{(w)}$ such that $w < u$ and $\pi(w)$ is minimized. In other words, the distribution of this node is the distribution of the first element in a restricted subset of elements in a random permutation. This is just the uniform distribution on the set of all candidate elements. That is precisely how Quicksort chose its pivot.

Thus the analyses for the running time for randomized Quicksort and Treesort are identical, and we have:

Theorem
The expected number of comparisons for randomized Quicksort of n elements is bounded by

$$2nH_n \approx 2n \ln n$$

and its expected running time is $O(n \log n)$.