CS 201 Data Structures Math Notes

JED YANG

1. Lower bound analysis for comparison-based sorting

A comparison-based sorting algorithm sorts by comparing two items at a time. One way to think about this is suppose we access information about the items only by using calls of the form `thisItem.compareTo(otherItem)`. Track the number of such calls.

Theorem 1. *Comparison-based sorting requires $\Omega(n \log n)$ comparisons to sort n items (in the worst case).*

Proof. Making k comparisons yields 2^k possible outcomes. We have $n!$ possible permutations of the n items, which must be distinguishable by the outcomes of the k comparisons. Therefore, $2^k \geq n!$. (Think of this as playing "20 questions" where I am secretly thinking of a permutation of n items, and you must determine that permutation by asking k yes/no questions of the type "is item 23 before item 201?") Now $n! \geq (n/2)^{n/2}$, so $k \geq \frac{n}{2}\log_2 \frac{n}{2} = \Omega(n \log n)$. □

2. Average analysis for quick sort

Theorem 2. *Sorting n items with quick sort uses $O(n \log n)$ comparisons on average.*

For simplicity, suppose we are sorting unique elements. Let $\mathbf{x} = (x_1, \ldots, x_n)$ be a permutation of $\{1, 2, \ldots, n\}$ chosen uniformly at random. Let $T(n)$ be the expected (average) number of comparisons when sorting $\mathbf{x}$.

Proof. Make $n-1$ comparisons of the pivot with everything else. There are i things smaller than the pivot and $n-i-1$ things bigger. Note that i is uniformly distributed. So we have

$$\begin{aligned}
T(n) &= (n-1) + \tfrac{1}{n}\sum_{i=0}^{n-1} T(i) + T(n-i-1) \\
&= n - 1 + \tfrac{2}{n}\sum T(i) \\
nT(n) &= n(n-1) + 2\sum T(i) && \text{cross multiply} \qquad (*) \\
nT(n) - (n-1)T(n-1) &= n(n-1) - (n-1)(n-2) + 2T(n-1) && \text{subtract (*) with (*)} \\
nT(n) &= (n+1)T(n-1) + 2(n-1) \\
\tfrac{T(n)}{n+1} &= \tfrac{T(n-1)}{n} + \tfrac{2(n-1)}{n(n+1)} \\
&\leq \tfrac{T(n-1)}{n} + \tfrac{2}{n+1} \\
&\leq \tfrac{T(n-2)}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\
&\vdots \\
&\leq \tfrac{T(1)}{2} + 2\left(\tfrac{1}{3} + \tfrac{1}{4} + \cdots + \tfrac{1}{n+1}\right) = O(\log n)
\end{aligned}$$

So $T(n) = O(n \log n)$, as desired. □

If we use a binary search tree (BST), we can get a one-line proof by using linearity of expectation:

Proof. The number of comparisons of quick sort on $\mathbf{x}$ is the number of comparisons when building a BST when we add items of $\mathbf{x}$ in order. For $i < j$, let X_{ij} be the indicator that x_j is compared to x_i (when adding x_j). Let $X = \sum_{i<j} X_{ij}$. Note that $(x_1, \ldots, x_i, x_j)$ is a permutation of those $i+1$ elements chosen uniformly at random. So x_j is next to x_i with probability $2/(i+1)$. So, by linearity of expectation, we have

$$\mathbb{E}[X] = \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i<j} \tfrac{2}{i+1} \stackrel{*}{=} O\left(\sum_{j} \log(j)\right) = O(n \log n),$$

where $*$ is because $\int_0^{j-1} \frac{1}{x+1}\,dx = \log j$. □

3. Amortized analysis for `add`ing to the end of an `ArrayList`

For `ArrayList`, what is the complexity of `add(E item)` where we add an item to the end? Technically it is $O(n)$ since we may need to double the size of the array, which involves copying the n items from the old array to the new array. However, there is a way around it. We can consider the cost per `add` operation averaged over all `add` operations. This is known as amortized analysis (cf. Wikipedia article).

Theorem 3. *The total cost of the first n* `ArrayList.add(E item)` *operations is $O(n)$, so, on average ("amortized"),* `add(E item)` *is $O(1)$.*

Proof. One way to think about this is as follows. Each time we `add`, we pay a cost of adding this item to the end of the array and two "tokens" that can be used to copy entries for free later (like savings in a bank).

When we are to double the array for the first time, say, from 10 to 20, we've already accumulated 20 tokens, so we use 10 of them to pay for array resizing. To make accounting easier, let us throw away the remaining 10 tokens—we will not need them.

When we want to double the array a second time, say, from 20 to 40, we would have collected 20 more tokens (2 for each of the 10 newly added items). That is exactly enough for us to pay for copying 20 items and left with 0 tokens.

The next time, say, from 40 to 80, we've added 20 more items, got 40 tokens, and can exactly pay for 40 copy operations.

In this way, we can see that the time it takes to add n items is at most the cost of adding n items at the end of an array (an $O(n)$ cost) and the cost of making $2n$ copies (also an $O(n)$ cost). If adding n items in total is $O(n)$, then adding one item on average is $O(1)$. □

Note that if, instead of doubling the array, we increase the size of the array by a constant (say 10) each time, then this analysis does not work. As such, it is better to double the array when resizing.

For this class, we will consider adding to the end of an array as an $O(1)$ operation.

4. Average analysis for binary search trees

Theorem 4. *A binary search tree with n nodes has height $O(\log n)$ on average.*

This is (way) too hard.