

CSC 378 Lecture 2

May 24, 2001

For another reference, see chapter 7 of the textbook (CLRS).

1 Quicksort

Quicksort sorts a list of integers as follows:

```
Quicksort ( List R )

  if |R| <= 1 then
    return R

  else
    select pivot a in R
    partition R into
      L = elements less than a
      M = elements equal to a
      U = elements greater than a
    return List( Quicksort(L), M, Quicksort(U) )
```

For now, we'll select the first element of R as the pivot a .

1.1 Worst-case analysis

We'll get an upper bound and then a lower bound on $T_{wc}(n)$, the worst-case running time of Quicksort for inputs of size n . As in the ListSearch example, we'll measure running time in terms of the number of comparisons performed. It will turn out that the upper and lower bounds are the same!

Upper bound: Quicksort works by comparing elements of R with the pivot a . Each element of R gets to be the pivot at most once, and it then gets compared to elements which have not yet been used as the pivot. So, we never compare the same pair of elements twice. Hence we perform at most $\binom{n}{2} = \frac{n(n-1)}{2}$ comparisons.

Lower bound: To get a lower bound, we guess the worst input for Quicksort and observe how many comparisons are needed to sort it. The quantity $T_{wc}(n)$ must, by definition, be at least this number. Let's let R be the already sorted list $\ell_n \stackrel{d}{=} (1, 2, \dots, n)$. We start by choosing 1 as the pivot, then comparing the rest of the $n - 1$ elements with 1. Everything is greater than 1 so it all ends up in $U = (2, 3, \dots, n)$. Now we have to run Quicksort on U , which is just as bad as R

(since it's already sorted) except that it is smaller by one element. So, if $t(n)$ is the number of comparisons needed for ℓ_n , then

$$t(n) = n - 1 + t(n - 1) \tag{1}$$

for all $n > 1$, and $t(1) = 0$. If we plug in $n - 1$ for n , then we get $t(n - 1) = n - 2 + t(n - 2)$. We can substitute this quantity for $t(n - 1)$ in (1) to get $t(n) = n - 1 + n - 2 + t(n - 2)$. Next we can substitute for $t(n - 2)$ in terms of $t(n - 3)$ and continue until we get to $t(1)$. So,

$$t(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Hence Quicksort takes $\frac{n(n-1)}{2}$ comparisons to sort the input $R = \ell_n$.

Since the upper lower bounds are the same, we know that $T_{wc}(n)$ must be exactly $n(n - 1)/2$.

1.2 Average-case analysis

Let's see if Quicksort does better on average than it does in the worst case. Our sample space S_n will be all the permutations of the list $(1, 2, \dots, n)$ since we don't care what the actual values of the elements are, just how they're ordered. Our probability distribution will be the uniform one; that is, we'll assume all permutations are equally likely and therefore have probability $1/n!$. Let the random variable $t_n : S_n \rightarrow \mathbf{N}$ be the number of comparisons needed to sort a given list in S_n . Recall that the definition of $T_{avg}(n)$, the average-case complexity of a list of length n , is

$$T_{avg}(n) \stackrel{d}{=} E[t_n] \stackrel{d}{=} \sum_{x \in S_n} \Pr(x) t_n(x). \tag{2}$$

We don't want to have to consider each individual input in order to compute $T_{avg}(n)$, so let's group them together into categories. Let the event $A_i \subset S_n$ be the lists where i is the first element. This happens with probability $1/n$ because each element is equally likely to be the first. If this is the case, then elements $1, 2, \dots, i - 1$ go into L and elements $i + 1, i + 2, \dots, n$ go into U . All the orderings of L and U are equally likely since all the orderings of the original list were equally likely. So, if $T_{avg}(n)$ is the average-case complexity of a list of length n , then $t_{avg}(A_i)$ —the average number of comparisons needed to sort a list in A_i —is

$$t_{avg}(A_i) = n - 1 + T_{avg}(i - 1) + T_{avg}(n - i),$$

where the three terms on the right are for (i) partitioning into L and U , (ii) sorting L , and (iii) sorting U .

Let's try to rewrite (2) in terms of A_i 's. We can partition the sum over S_n into a sum of sums over each A_i :

$$T_{avg}(n) = \sum_{i=1}^n \sum_{x \in A_i} \Pr(x) t_n(x).$$

Since $t_{avg}(A_i)$ is the average time that $x \in A_i$ takes, we can write

$$T_{avg}(n) = \sum_{i=1}^n \left(\sum_{x \in A_i} \Pr(x) \right) t_{avg}(A_i).$$

The sum $\sum_{x \in A_i} \Pr(x)$ is just the definition of $\Pr(A_i)$, so

$$T_{avg}(n) = \sum_{i=1}^n \Pr(A_i) t_{avg}(A_i).$$

This is equal to

$$\sum_{i=1}^n \frac{1}{n} (n-1 + T_{avg}(i-1) + T_{avg}(n-i)) = n-1 + \frac{2}{n} \sum_{j=1}^{n-1} T_{avg}(j).$$

In addition, we know that $T_{avg}(0) = T_{avg}(1) = 0$.

This seems like a difficult recurrence to solve, but observe the similarities between the following two equations:

$$T_{avg}(n) = n-1 + \frac{2}{n} \sum_{j=1}^{n-1} T_{avg}(j) \quad (3)$$

$$T_{avg}(n-1) = n-2 + \frac{2}{n-1} \sum_{j=1}^{n-2} T_{avg}(j). \quad (4)$$

If we clear the denominators of the sums, we can cancel most of the terms by subtracting the bottom equation from the top:

$$nT_{avg}(n) - (n-1)T_{avg}(n-1) = n(n-1) - (n-1)(n-2) + 2T_{avg}(n-1).$$

Simplifying:

$$nT_{avg}(n) = (n+1)T_{avg}(n-1) + 2(n-1)$$

or

$$\frac{T_{avg}(n)}{n+1} = \frac{T_{avg}(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$

We can rewrite $\frac{T_{avg}(n)}{n+1}$ as $B(n)$ so that

$$B(n) = B(n-1) + \frac{2(n-1)}{n(n+1)},$$

where $B(0) = T_{avg}(0)/1 = 0$. Then,

$$B(n) = \sum_{i=1}^n \frac{2(i-1)}{i(i+1)} \quad (5)$$

$$= 2 \sum_{i=1}^n \frac{1}{i+1} - 2 \sum_{i=1}^n \frac{1}{i(i+1)} \quad (6)$$

$$= 2 \sum_{i=1}^{n+1} \frac{1}{i} - 1 - 2 \sum_{i=1}^n \frac{1}{i(i+1)}. \quad (7)$$

[CLR, p. 50] shows $\sum_{i=1}^n 1/i = \Theta(\log n)$. Clearly, the second and third terms of $B(n)$ are smaller, so $B(n) = \Theta(\log(n+1)) = \Theta(\log n)$. Since $T_{avg}(n) = (n+1)B(n)$, $T_{avg}(n) = \Theta(n \log n)$.

If you solve the recurrence more carefully, you will find that all of the constants which are eliminated by the Θ notation are small values. This fact is one of the reasons that Quicksort is actually quick in practise (compared with other sorting algorithms that have complexity $\Theta(n \log n)$).

1.3 Randomized Quicksort

Note: This section discusses a complexity measure which may seem similar to T_{avg} , average case complexity. They are not the same! If you do not understand the definition of average case complexity, *do not* continue to this section.

We've seen that Quicksort does pretty well on the average input, but we've also seen that there are some particular inputs on which it does badly. If our input is usually sorted or close to sorted then Quicksort is not a good solution.

One way to fix this situation is to pick a random element as the pivot instead of the first element. We'll call this algorithm RQuicksort. Note that this is a different algorithm from Quicksort; Quicksort is *deterministic*, while RQuicksort is *randomized* (that is, it makes random choices).

For a given input S , we start by picking a random index p_1 as the pivot. Then, when we recurse on L and U , we pick random indices p_2 and p_3 , respectively, as the pivots, etc. Let $p = (p_1, p_2, \dots, p_n)$ be the sequence of random pivot choices in one execution of RQuicksort on a particular input list R . The possible p 's constitute a sample space P_n , and we'll assume that the probability distribution on the space is uniform. We can then define the random variable $t_R : P_n \rightarrow \mathbf{N}$, the running time of RQuicksort on list R given some sequence of pivot choices. The expected running time of RQuicksort on input R is defined as

$$E[t_R] = \sum_{p \in P_n} \Pr(p) t_R(p).$$

Do not confuse this with $T_{avg}(n)$, the average-case running time of Quicksort, which is the expected running time over all possible input lists!

On the other hand, in this case, $T_{avg}(n)$ and $E[t_R]$ happen to have the same value for any input R . This is because choosing a random element of R is equivalent to choosing the first element of a random permutation of $(1, 2, \dots, n)$. So $E[t_R] = \Theta(n \log n)$ for any R . This is good because there is no particular input which will definitely be bad for RQuicksort.

In general, the expected running time of a randomized algorithm A may vary depending upon the input. As usual, let S_n be the possible inputs to A of size n . Let $P_n(x)$ be the sample space of random choices that A can make on input x . We can define the *expected worst-case complexity* of A as

$$\max_{x \in S_n} \{E[t_x]\},$$

where the expectation is over $P_n(x)$.