20 Quicksort

Quicksort sorts a list of integers as follows:

Quicksort ( List R )

\[
\text{if } |R| \leq 1 \text{ then} \\
\quad \text{return } R \\
\text{else} \\
\quad \text{select pivot } a \text{ in } R \\
\quad \text{partition } R \text{ into} \\
\quad \quad L = \text{elements less than } a \\
\quad \quad M = \text{elements equal to } a \\
\quad \quad U = \text{elements greater than } a \\
\quad \text{return List( Quicksort(L), M, Quicksort(U) )}
\]

For now, we’ll select the first element of \( R \) as the pivot \( a \).

20.1 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!

20.1.1 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.
20.1.2 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 $\ell_n \equiv (1, 2, \ldots, n)$. 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 $U = (2, 3, \ldots, n)$. 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 $\ell_n$, then

$$t(n) = n - 1 + t(n - 1)$$

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, we perform $\frac{n(n-1)}{2}$ comparisons for this $R$.

Since the upper and lower bounds are the same, we know that $T_{wc}(n)$ must be exactly $n(n-1)/2$ for quicksort. Therefore, $T_{wc}(n) \in \Theta(n^2)$.

20.2 Average case analysis

Let’s see if Quicksort does better on average than it does in the worst case.

1. Our sample space $S_n$ will be all the permutations of the list $(1, 2, \ldots, n)$ since we don’t care what the actual values of the elements are, just how they’re ordered.

   **Exercise.** Why is it a reasonable assumption that there are no repeated elements?

2. Our probability distribution will be the uniform one; that is, we’ll assume all permutations are equally likely and therefore have probability $1/n!$.

3. Let the random variable $t_n : S_n \to \mathbb{N}$ 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

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

(14)

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 $A_{ni} \subset S_n$ be such that $A_{ni} = \{ \text{All permutations of } (1, 2, \ldots, n) \text{ such that } i \text{ is the first element} \}$. $A_{ni}$ occurs with probability $1/n$ because each element is equally likely to be the first. If this is the case, then elements $1, 2, \ldots, i-1$ go into $L$ and elements $i+1, i+2, \ldots, n$ 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 $T_{\text{avg}}(n)$ is the average case complexity of a list of length $n$, then $t_{\text{avg}}(A_{ni})$—the average number of comparisons needed to sort a list in $A_{ni}$—is

$$t_{\text{avg}}(A_{ni}) = n - 1 + T_{\text{avg}}(i - 1) + T_{\text{avg}}(n - i),$$

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 $A_{ni}$'s. We can partition the sum over $S_n$ into a sum of sums over each $A_{ni}$:

$$T_{\text{avg}}(n) = \sum_{i=1}^{n} \sum_{x \in A_{ni}} \Pr(x) t_{n}(x).$$

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

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

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

$$T_{\text{avg}}(n) = \sum_{i=1}^{n} \Pr(A_{ni}) t_{\text{avg}}(A_{ni})$$

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

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

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

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

$$T_{\text{avg}}(n) = n - 1 + \frac{2}{n} \sum_{j=1}^{n-1} T_{\text{avg}}(j) \quad (15)$$

$$T_{\text{avg}}(n - 1) = n - 2 + \frac{2}{n - 1} \sum_{j=1}^{n-2} T_{\text{avg}}(j). \quad (16)$$

Our method will be to eliminate the denominators from the summations and subtract. We get the following equation:

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

Then, we simplify:

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

66
or
\[
\frac{T_{\text{avg}}(n)}{n + 1} = \frac{T_{\text{avg}}(n - 1)}{n} + \frac{2(n - 1)}{n(n + 1)}.
\]

If we let \( B(n) \equiv \frac{T_{\text{avg}}(n)}{n+1} \) then
\[
B(n) = B(n - 1) + \frac{2(n - 1)}{n(n + 1)},
\]
where \( B(0) = T_{\text{avg}}(0)/1 = 0 \). Then,
\[
B(n) = \sum_{i=1}^{n} \frac{2(i - 1)}{i(i + 1)} = 2 \sum_{i=1}^{n} \frac{1}{i(i + 1)} - 2 \sum_{i=1}^{n} \frac{1}{i(i + 1)} = 2 \sum_{i=1}^{n+1} \frac{1}{i} - 1 - 2 \sum_{i=1}^{n} \frac{1}{i(i + 1)}.
\]

[CLRS Appendix A] shows that \( \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n) \). Clearly, the first term of \( B(n) \) dominates the second and third terms, so \( B(n) \in \Theta(\log(n + 1)) = \Theta(\log n) \). Since \( T_{\text{avg}}(n) = (n + 1)B(n) \), \( T_{\text{avg}}(n) \in \Theta(n \log n) \).

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

### 20.3 Randomized Quicksort

In this section we will be discussing a randomized version of Quicksort. Although randomized Quicksort may appear related to the average case analysis of non-randomized Quicksort, they are not the same. You should be certain that you understand the previous sections on average case analysis before proceeding.

We have seen that Quicksort does well on the average input, but we have also seen that there are some particular inputs on which it does badly. If our input is typically 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. In other words, where Quicksort always chooses the same element as the pivot, RQuicksort chooses a random element as the pivot.

For a given input \( R \), 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 \( p = (p_1, p_2, \ldots, p_n) \) 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
\( t_R : P_n \rightarrow \mathbb{N} \), the running time of \( R\text{Quick} \)sort on list \( R \) given some sequence of pivot choices. The expected running time of \( R\text{Quick} \)sort on input \( R \) is defined as

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

Note that \( E[t_R] \) is the expected running time for \( R\text{Quick} \)sort on a given input whereas \( T_{\text{avg}}(n) \) is the expected running time for an algorithm over all possible inputs.

Despite this fact, in this case, \( T_{\text{avg}}(n) \) and \( E[t_R] \) for any input \( R \) happen to have the same value. This is because choosing a random element of \( R \) is equivalent to choosing the first element of a random permutation of \( (1, 2, \ldots, n) \). 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 \( R\text{Quick} \)sort.

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 \( P_n(x) \).

Instead of relying on unknown distribution of inputs, randomize an algorithm by picking random element as pivot. This way, random behaviour of an algorithm on any fixed input is equivalent to fixed behaviour of the same algorithm on a uniformly random input. In other words, expected worst case complexity of \( R\text{Quick} \)sort is \( \Theta(n \log n) \). In general, randomized algorithms are good when there are many good choices but it is difficult to find one choice that is guaranteed to be good.