Randomized Quicksort and QuickSelect\(^1\)

Analysis of Randomized Quicksort

We give here two methods for analyzing randomized quicksort. The first is more intuitive – we covered it in class. The second is a neat tricky way using the power of linearity of expectation; however it is less intuitive – this is covered in Chapter 7 in the “CLRS” book.

Method 1

For simplicity, let us assume no two elements in the array are equal — when we are done with the analysis, it will be easy to look back and see that allowing equal keys could only improve performance. We now prove the following theorem.

**Theorem 1** The expected number of comparisons made by randomized quicksort on an array of size \(n\) is at most \(2n \ln n\).

**Proof:** First of all, when we pick the pivot, we perform \(n - 1\) comparisons (comparing all other elements to it) in order to split the array. Now, depending on the pivot, we might split the array into a LESS of size 0 and a GREATER of size \(n - 1\), or into a LESS of size 1 and a GREATER of size \(n - 2\), and so on, up to a LESS of size \(n - 1\) and a GREATER of size 0. All of these are equally likely with probability \(1/n\) each. Therefore, we can write a recurrence for the expected number of comparisons \(T(n)\) as follows:

\[
T(n) = (n - 1) + \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n - i - 1)).
\]  

Formally, we are using the expression for Expectation given in (??), where the \(n\) different possible splits are the events \(A_i\).\(^2\) We can rewrite equation (1) by regrouping and getting rid of \(T(0)\):

\[
T(n) = (n - 1) + \frac{2}{n} \sum_{i=1}^{n-1} T(i)
\]  

\(^1\)These lecture notes are due to Avrim Blum.

\(^2\)In addition, we are using Linearity of Expectation to say that the expected time *given* one of these events can be written as the sum of two expectations.
Now, we can solve this by the “guess and prove inductively” method. In order to do this, we first need a good guess. Intuitively, most pivots should split their array “roughly” in the middle, which suggests a guess of the form $cn \ln n$ for some constant $c$. Once we’ve made our guess, we will need to evaluate the resulting summation. One of the easiest ways of doing this is to upper-bound the sum by an integral. In particular if $f(x)$ is an increasing function, then

$$
\sum_{i=1}^{n-1} f(i) \leq \int_1^n f(x)dx,
$$

which we can see by drawing a graph of $f$ and recalling that an integral represents the “area under the curve”. In our case, we will be using the fact that $\int (cx \ln x)dx = (c/2)x^2 \ln x - cx^2/4$.

So, let’s now do the analysis. We are guessing that $T(i) \leq ci \ln i$ for $i \leq n - 1$. This guess works for the base case $T(1) = 0$ (if there is only one element, then there are no comparisons). Arguing by induction we have:

$$
T(n) \leq (n - 1) + \frac{2}{n} \sum_{i=1}^{n-1} (ci \ln i) \\
\leq (n - 1) + \frac{2}{n} \int_1^n (cx \ln x)dx \\
\leq (n - 1) + \frac{2}{n} ((c/2)n^2 \ln n - cn^2/4 + c/4) \\
\leq cn \ln n, \text{ for } c = 2. \quad \blacksquare
$$

In terms of the number of comparisons it makes, Randomized Quicksort is equivalent to randomly shuffling the input and then handing it off to Basic Quicksort. So, we have also proven that Basic Quicksort has $O(n \log n)$ average-case running time.

**Method 2**

Here is a neat alternative way to analyze randomized quicksort that is very similar to how we analyzed the card-shuffling example.

**Alternative proof (Theorem 1):** As before, let’s assume no two elements in the array are equal since it is the worst case and will make our notation simpler. The trick will be to write the quantity we care about (the total number of comparisons) as a sum of simpler random variables, and then just analyze the simpler ones.

Define random variable $X_{ij}$ to be 1 if the algorithm does compare the $i$th smallest and $j$th smallest elements in the course of sorting, and 0 if it does not. Let $X$ denote the total number of comparisons made by the algorithm. Since we never compare the same pair of
elements twice, we have

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

and therefore,

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

Let us consider one of these \( X_{ij} \)'s for \( i < j \). Denote the \( i \)th smallest element in the array by \( e_i \) and the \( j \)th smallest element by \( e_j \), and conceptually imagine lining up the elements in sorted order. If the pivot we choose is between \( e_i \) and \( e_j \) then these two end up in different buckets and we will never compare them to each other. If the pivot we choose is either \( e_i \) or \( e_j \) then we do compare them. If the pivot is less than \( e_i \) or greater than \( e_j \) then both \( e_i \) and \( e_j \) end up in the same bucket and we have to pick another pivot. So, we can think of this like a dart game: we throw a dart at random into the array: if we hit \( e_i \) or \( e_j \) then \( X_{ij} \) becomes 1, if we hit between \( e_i \) and \( e_j \) then \( X_{ij} \) becomes 0, and otherwise we throw another dart. At each step, the probability that \( X_{ij} = 1 \) conditioned on the event that the game ends in that step is exactly \( 2/(j-i+1) \). Therefore, overall, the probability that \( X_{ij} = 1 \) is \( 2/(j-i+1) \).

In other words, for a given element \( i \), it is compared to element \( i+1 \) with probability 1, to element \( i+2 \) with probability \( 2/3 \), to element \( i+3 \) with probability \( 2/4 \), to element \( i+4 \) with probability \( 2/5 \) and so on. So, we have:

\[ E[X] = \sum_{i=1}^{n} 2 \left( \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n-i+1} \right). \]

The quantity \( 1+1/2+1/3+\ldots+1/n \), denoted \( H_n \), is called the “\( n \)th harmonic number” and is in the range \([\ln n, 1 + \ln n]\) (this can be seen by considering the integral of \( f(x) = 1/x \)). Therefore,

\[ E[X] < 2n(H_n - 1) \leq 2n \ln n. \]

**Selection: finding the median in linear time**

**Overview**

Given an unsorted array, how quickly can one find the median element? Can one do it more quickly than by sorting? This was an open question for some time, solved affirmatively in 1972 by (Manuel) Blum, Floyd, Pratt, Rivest, and Tarjan. We will describe two linear-time algorithms for this problem: one randomized (in this lecture) and one deterministic (in the next lecture). More generally, we solve the problem of finding the \( k \)th smallest out of an unsorted array of \( n \) elements.
1 The problem and a randomized solution

A related problem to sorting is the problem of finding the \( k \)th smallest element in an unsorted array. (Let’s say all elements are distinct to avoid the question of what we mean by the \( k \)th smallest when we have equalities). One way to solve this problem is to sort and then output the \( k \)th element. Is there something faster – a linear-time algorithm? The answer is yes. We will explore both a simple randomized solution and a more complicated deterministic one.

The idea for the randomized algorithm is to notice that in Randomized-Quicksort, after the partitioning step we can tell which subarray has the item we are looking for, just by looking at their sizes. So, we only need to recursively examine one subarray, not two. For instance, if we are looking for the 87th-smallest element in our array, and after partitioning the “LESS” subarray (of elements less than the pivot) has size 200, then we just need to find the 87th smallest element in LESS. On the other hand, if the “LESS” subarray has size 40, then we just need to find the \( 87 - 40 = 46 \)th smallest element in GREATER. (And if the “LESS” subarray has size exactly 86 then we just return the pivot). One might at first think that allowing the algorithm to only recurse on one subarray rather than two would just cut down time by a factor of 2. However, since this is occurring recursively, it compounds the savings and we end up with \( \Theta(n) \) rather than \( \Theta(n \log n) \) time. This algorithm is often called Randomized-Select, or QuickSelect.

**QuickSelect:** Given array \( A \) of size \( n \) and integer \( k \leq n \),

1. Pick a pivot element \( p \) at random from \( A \).
2. Split \( A \) into subarrays LESS and GREATER by comparing each element to \( p \) as in Quicksort. While we are at it, count the number \( L \) of elements going in to LESS.
3. (a) If \( L = k - 1 \), then output \( p \).
   (b) If \( L > k - 1 \), output QuickSelect(LESS, \( k \)).
   (c) If \( L < k - 1 \), output QuickSelect(GREATER, \( k - L - 1 \))

**Theorem 2** The expected number of comparisons for QuickSelect is \( O(n) \).

Before giving a formal proof, let’s first get some intuition. If we split a candy bar at random into two pieces, then the expected size of the larger piece is \( 3/4 \) of the bar. If the size of the larger subarray after our partition was always \( 3/4 \) of the array, then we would have a recurrence \( T(n) \leq (n - 1) + T(3n/4) \) which solves to \( T(n) < 4n \). Now, this is not quite the case for our algorithm because \( 3n/4 \) is only the *expected* size of the larger piece. That is, if \( i \) is the size of the larger piece, our expected cost to go is really \( E[T(i)] \) rather than \( T(E[i]) \). However, because the answer is linear in \( n \), the average of the \( T(i) \)'s turns out to be the same as \( T(\text{average of the } i\text{'}s) \). Let’s now see this a bit more formally.
Proof (Theorem 2): Let $T(n, k)$ denote the expected time to find the $k$th smallest in an array of size $n$, and let $T(n) = \max_k T(n, k)$. We will show that $T(n) < 4n$.

First of all, it takes $n-1$ comparisons to split into the array into two pieces in Step 2. These pieces are equally likely to have size $0$ and $n-1$, or $1$ and $n-2$, or $2$ and $n-3$, and so on up to $n-1$ and $0$. The piece we recurse on will depend on $k$, but since we are only giving an upper bound, we can imagine that we always recurse on the larger piece. Therefore we have:

$$T(n) \leq (n-1) + \frac{2}{n} \sum_{i=n/2}^{n-1} T(i)$$

$$= (n-1) + \text{avg}[T(n/2), \ldots, T(n-1)].$$

We can solve this using the “guess and check” method based on our intuition above. Assume inductively that $T(i) \leq 4i$ for $i < n$. Then,

$$T(n) \leq (n-1) + \text{avg}[4(n/2), 4(n/2 + 1), \ldots, 4(n-1)]$$

$$\leq (n-1) + 4(3n/4)$$

$$< 4n,$$

and we have verified our guess. ■