1 Quicksort, continued

Recall that Quicksort is a divide and conquer algorithm that works by first choosing a "pivot" value q and then partitioning the input array A into a left part, which contains all the values in A that are less than q, and a right part, which contains all the values in A that are greater than q. It then (recursively) calls Quicksort on the left and right parts.

We have already proved that Quicksort is a correct algorithm, meaning that on any input, it returns an array A that is sorted. But, how long does it take to achieve this result? To determine this, we must analyze the behavior of the algorithm, and in doing this, it will be useful to remember this figure from Lecture 3.

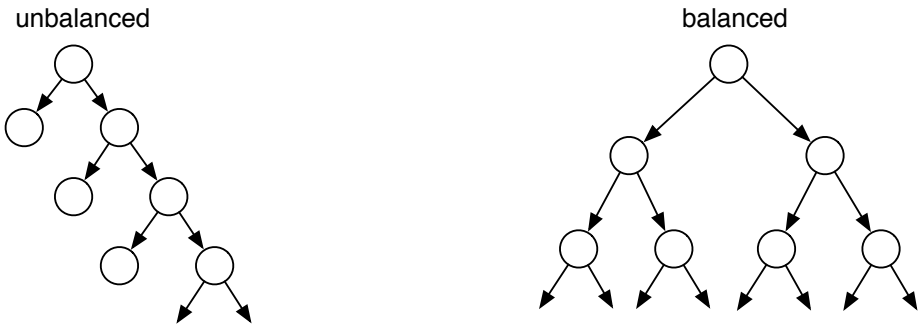

Figure 1: Examples of unbalanced and balanced binary trees. A fully unbalanced tree has depth $\Theta(n)$ while a full balanced tree has depth $\Theta(\log n)$.

And, recall that the general form of a recurrence relation is

$$T(n) = a\,T(g[n]) + f(n) \ , \tag{1}$$

where $T(n)$ is the running time, a is the number of recursive calls made, $g(n)$ is the size of the subproblem passed to each of these calls, and $f(n)$ is the cost of the current call.

1.1 Analysis

The loop inside `Partition` examines each element in its subarray and thus `Partition` takes $f(n) = \Theta(n)$ time for a subarray of length n. The total running time of Quicksort thus depends on whether the partitioning (the recursive step) is balanced or not, and this depends solely on which element in A is used as the pivot. To see why, we'll examine the worst- and best-case performances.

1.1.1 Worst case

The worst possible performance occurs when the Quicksort recursion tree is maximally *unbalanced* (see Figure 1). The most unbalanced partitions occur when we repeatedly divide A such that one piece has $O(1)$ elements while the other has $O(n)$ elements.

The recursion cost term (from Eq. (1)) for Quicksort is always $f(n) = \Theta(n)$, because this is the cost we incur for calling `Partition`. For the worst-case division, the function $g[n] = n - b$, for b constant. The mathematics for general b are largely the same as for the $b = 1$ case, but the $b - 1$ case is a little easier to understand. The recurrence relation for this worst-case performance of Quicksort is

$$T(n) = T(n-1) + \Theta(n) \ . \tag{2}$$

The form of Eq. (2) is characteristic of all "tail recursion" algorithms, which are logically equivalent to `for` loops. It can easily be solved using the "unrolling" method:

$$\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
T(n) &= T(n-2) + \Theta(n-1) + \Theta(n) \\
T(n) &= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
&\vdots \\
T(n) &= \Theta(1) + \cdots + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
T(n) &= \sum_{i=1}^{n} \Theta(i) \\
T(n) &= \Theta(n^2) \ .
\end{aligned}$$

Thus, the worst-case running time of Quicksort is $\Theta(n^2)$.

(As an at-home exercise, identify the input (an array containing values $x_1, x_2, \ldots x_n$) that produces this worst-case performance? What fraction of permutations of these values lead to this behavior?)

1.1.2 Best case, and the Master Theorem

The best case occurs when the partitions are proportionally sized, that is, when $g[n] = n/b$, with b constant. The mathematics for general b are largely the same as for the $b = 2$ case, in which the pivot is always chosen to be the median of the values in the subarray. In this case, the recurrence relation is

$$T(n) = 2T(n/2) + \Theta(n) \ . \tag{3}$$

That is, because we divide the list into 2 equal halves, the recursive call produces 2 subproblems each half the size as our current problem. To solve this recurrence relation, we will use the *master method* (CLRS Chapter 4.3), which can be applied to any recurrence relation in which $g[n] = n/b$ for some constant b.

Master theorem
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function and let $T(n)$ be defined on the non-negative integers by the recurrence,

$$T(n) = a\,T(n/b) + f(n) \ ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a\,f(n/b) \leq c\,f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. □

The master method does not require any thought — all of that was put into proving it — so it does not require much algebra. However, if you use it, you need to clearly state the case, and specify the conditions (including a, b, and ϵ).

Before applying it, let's consider what it's doing. Each of the three cases compares the asymptotic form of $f(n)$ with $n^{\log_b a}$; whichever of these functions is larger dominates the running time.[1] In Case 1, $n^{\log_b a}$ is larger so the running time is $\Theta(n^{\log_b a})$; in Case 2, the functions are are the same size, so we multiply by a $\log n$ factor and the solution is $\Theta(f(n) \log n)$; in Case 3, $f(n)$ is larger, so the solution is $\Theta(f(n))$.[2] [3]

Applying the master method to Eq. (3) is straightforward. Note that $a = b = 2$ and $f(n) = \Theta(n) = \Theta(n^{\log_2 2})$ which is Case 2, in which $n^{\log_b a}$ and $f(n)$ are the same size, so $T(n) = \Theta(n \log n)$.

(As an at-home exercise, try using the "unrolling" method to solve Eq. (3).)

[1] In fact, the function much be *polynomially* smaller, which is why there's a factor of n^{ϵ} in the theorem.

[2] Similar to Case 1, there is a *polynomially* larger condition here, plus a "regularity" condition in which $a\,f(n/b) \leq c, f(n)$, but most polynomially bounded functions we encounter will also satisfy the regularity condition.

[3] Another detail is that the Master Theorem does not actually cover all possible cases. There are classes of functions that are not covered by the three cases; if this is true, then the Master Theorem cannot be applied.

2 Average case, and Randomized Quicksort

Recall that in our analysis of the worst- and best-case behavior of Quicksort, we argued informally that so long as `Partition` chooses a good pivot element a constant fraction of the time, the average case will be $O(n \log n)$. We will now give a more rigorous treatment of the average case by studying the performance of a randomized variation of Quicksort. To begin, consider these questions:

How often should we expect the *worst case* to occur?
How often should we expect the *best case* to occur?

Because the pivot element is always the last element of each subproblem, i.e., the choice is deterministic, answers to these questions hinge on the precise ordering of the values $x_1, \ldots, x_n$ in the input array.

Recall that for n elements, there are $n!$ possible orderings. For the moment, assume that each of these is equally likely. If we are unlucky and the input array is badly ordered, then we will choose a bad pivot n times and pay $\Theta(n)$ cost each time. Recall from our informal argument about the average case that any constant fraction $1/k$ of such bad choices will inflate the depth of the recursion tree by a constant factor k. Being a factor of k deeper does not change the asymptotic performance because k is only a constant, independent of n.

Thus, to fall into the $\Theta(n^2)$ worst case, we need to choose a bad pivot more than a constant fraction of the time, e.g., we need to choose poorly a $1 - o(1)$ fraction of the time. The flip side of this argument is that in order to avoid the worst case performance, we need only choose a good pivot a constant number of times, regardless of the size of the input. (Do you see why?)

If input orderings are chosen uniformly at random, there is only a small chance of choosing one that will induce such a large number of bad pivots. (What is that chance?) By convention, however, we are concerned with worst-case scenarios (this is the so-called "adversarial model" of algorithm analysis), and thus we cannot be optimistic about the type of input the algorithm receives.

But, all is not lost: instead of hoping that inputs are random, we can explicitly add randomness into the operation of Quicksort in order to make it behave as if the input were random. This is a powerful algorithm design strategy: by using probability in the internal flow of the algorithm, we can convert an arbitrary input into a random input, thereby obtaining many of the nice properties of its average performance for nearly every input and defeating our adversary almost all of the time.

2.1 The algorithm

By making use of a pseudo-random number generator to simulate random choices of the pivot element,[4] we can make QuickSort behave as if whatever input it receives is actually an average case. This is our first example of a *randomized algorithm*.[5]

The pseudocode for Randomized Quicksort has the same structure as Quicksort, except that it calls a randomized version of the `Partition` procedure, which we'll call `RPartition` to distinguish it from its deterministic cousin. Here's the main procedure:

```
Randomized-Quicksort(A,p,r) {
   if (p<r){
      q = RPartition(A,p,r)
      Randomized-Quicksort(A,p,q-1)
      Randomized-Quicksort(A,q+1,r)
}}.
```

The `Randomized-Partition` function called above is where this version deviates from the deterministic Quicksort we saw last time:

```
RPartition(A,p,r) {
   i = random-int(p,r) // NEW: choose uniformly random integer on [p..r]
   swap(A[i],A[r])     // NEW: swap corresponding element with last element
   x = A[r]            // pivot is now a uniformly random element
   i = p-1             // code from deterministic Partition
   for (j=p; j<=r-1;j++) {
      if A[j]<=x {           // same as before
         i++                 //
         swap(A[i],A[j])     //
   }}
   swap(A[i+1],A[r])   //
   return i+1          //
},
```

[4] There is a philosophical debate about whether random numbers can truly be generated that we needn't have. So long as our source of random bits behaves as if its truly random, we can proceed as if it actually is. For the purposes of algorithm analysis, we assume that producing a single random real or integer number is an atomic operation, taking $\Theta(1)$ time.

[5] Randomized algorithms are a big part of modern algorithm theory; see *Randomized Algorithms* by Motwani and Raghavan. In general, a source of random bits—like a pseudo-random number generator—is used to make the behavior of the algorithm more independent of the input sequence, which constrains both the worst- and best-case behaviors. The analysis of randomized algorithms typically requires computing the likelihood of certain sequences of computation using probability theory. Bizarrely, some randomized algorithms can be *derandomized*—removing the source of random bits—even while preserving their overall performance.

where `random-int(i,j)` is a function that returns integers from the interval $[i, j]$ such that each integer is equally likely. Thus, we choose a uniformly random element from `A[p..r]` and make *it* the pivot by swapping it with the last element. The rest of the `RPartition` function does exactly the same steps as the deterministic `Partition` function.

3 On your own

1. Read Chapters 5 and 7