1 Randomized Quicksort

Last time, we did the worst- and best-case analyses of Quicksort, and we argued informally that so long as `Partition` chooses a good pivot element a constant fraction of the time, the average case will be $O(n \log n)$. (Do you remember?) This time, we will more rigorously treat 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.

1.1 The algorithm

By making use of a pseudo-random number generator to simulate random choices of the pivot element,[1] 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*.[2]

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 {            //
         i++                  //
         swap(A[i],A[j]) //
   }}
   swap(A[i+1],A[r])     //
   return i+1                 //
},
```

[1] 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.

[2] 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.

1.2 Analysis

Because we are using a source of random bits to randomize the decisions of Quicksort, the running time of Randomized Quicksort is itself a *random variable*. We're interested in the average or *expected* running time—the expected value of this random variable—and to analyze that, we'll need to use a few tools from probability theory. (See Appendix C.2-3 in CLRS, which covers introductory probability theory.)

1.2.1 A few tools from probability theory

A *random variable* is a variable X that takes several values, each with some probability. For example, the outcome of rolling a die, like a pair of 6-sided dice as in some casino games, is often modeled as a random variable. The *expected value* of a random variable is defined as

$$E(X) = \sum_x x \Pr(X = x) \qquad \text{given that } \sum_x \Pr(X = x) = 1 \text{ and } x \text{ is discrete} \tag{1}$$

$$E(X) = \int_x x \Pr(x) \mathrm{d}x \qquad \text{given that } \int_x \Pr(x) \mathrm{d}x = 1 \text{ and } x \text{ is continuous.} \tag{2}$$

The right-hand sides of these statements verify that the function $\Pr(x)$ in each case is a *probability distribution*. For example, the expected value of a single roll of one 6-sided die is

$$\begin{aligned} E(X) &= 1\left(\frac{1}{6}\right) + 2\left(\frac{1}{6}\right) + 3\left(\frac{1}{6}\right) + 4\left(\frac{1}{6}\right) + 5\left(\frac{1}{6}\right) + 6\left(\frac{1}{6}\right) \\ &= \frac{1}{6}\sum_{i=1}^{6} i = \frac{1}{6} \cdot \frac{6(6+1)}{2} = \frac{21}{6} = 3.5 \ . \end{aligned}$$

In this example, outcomes are *independent*, which means that the outcome of one roll does not change the likelihood of outcomes on any subsequent roll. Mathematically, we say that if $X = x$ and $Y = y$, x and y are independent if and only if ("iff"):

$$\Pr(X = x, Y = y) = \Pr(X = x) \Pr(Y = y) \ . \tag{3}$$

Continuing the dice example, the probability of getting "snake eyes", that is, both dice show a 1, is $\Pr(X = 1, Y = 1) = \Pr(X = 1) \cdot \Pr(Y = 1) = 1/6 \cdot 1/6 = 1/36$. That is, the probability of getting $X = 1$ *and* $Y = 1$ is the product of the probabilities of the individual events.

Similarly, the probability of getting $X = 1$ *or* $Y = 1$ is the sum of the individual probabilities. For example, given two dice, the probability of getting at least one of the dice to show a 1 is $\Pr(X = 1 \text{ or } Y = 1) = \Pr(X = 1) + \Pr(Y = 1) = 1/6 + 1/6 = 1/3$.[3]

An *indicator random variable* is a special kind of random variable, which use can use to count the number of times some event A occurs:

$$I(A) = \begin{cases} 1 & \text{if the event } A \text{ occurs} \\ 0 & \text{otherwise .} \end{cases}$$

For instance, we could use such a structure to count the number of times a 6-sided die comes up 1.

If X and Y are two random variables, then the property of *linearity of expectations* states that

$$E(X + Y) = E(X) + E(Y) \ , \tag{4}$$

which holds even if X and Y are not independent. More generally, if $\{X_i\}$ is a set of random variables, then by linearity of expectations, the following is true:

$$E\left(\sum_i X_i\right) = \sum_i E(X_i) \ . \tag{5}$$

1.2.2 Analyzing Randomized Quicksort

With these tools, we can now analyze Randomized Quicksort. Let the input array be denoted $A = [z_1, z_2, \ldots, z_n]$ and let A be already sorted, that is, z_i is the ith smallest element of A. Further, let the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}$ denote those elements between z_i and z_j, inclusive. Note that A being sorted has no impact the running time of Randomized Quicksort.

When does Randomized Quicksort compare z_i and z_j? This comparison will occur either one or zero times, and it occurs only when (i) a subproblem of Randomized Quicksort contains Z_{ij} *and* (ii) either z_i or z_j is chosen as the pivot element. Otherwise, z_i and z_j are never compared. (Do you see why?) Let X denote the running time of Randomized Quicksort; X is then exactly equal to the number of times z_i and z_j are compared, for all i, j, i.e., X counts the number of comparison operations.[4] That is,

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ ,$$

[3] For a fun excursion into combinatorics and probability, try computing the probability of getting a *yahtzee*, i.e., all dice showing the same value, for 5 dice where you may re-roll each die at most 2 times after your initial roll. Hint: the probability is several thousand times larger than $6 \times (1/6)^5 \approx 0.00077$.

[4] The total number of atomic operations is proportional to the number of comparisons, and thus, for asymptotic analysis, it is sufficient to count only the comparisons.

where we define $X_{ij} = I\{z_i \text{ is compared to } z_j\}$.

By linearity of expectations, we can simplify this expression:

$$\begin{aligned} E(X) &= E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j) \ . \end{aligned} \tag{6}$$

This is more helpful, but Eq. (6) is not something we can express yet in big-O notation. To do that, we need to further refine what $\Pr(z_i \text{ is compared to } z_j)$ means. Recall that

$$\Pr(z_i \text{ is compared to } z_j) = \Pr(\text{either } z_i \text{ or } z_j \text{ are the first pivots chosen in } Z_{ij}) \ .$$

To see that this is true, recall two facts:

1. if no element in Z_{ij} has yet been chosen, no two elements in Z_{ij} have yet been compared, and thus all of Z_{ij} are in the same subproblem of Randomized Quicksort; and
2. if some element in Z_{ij} other than z_i or z_j is chosen first, then z_i and z_j will be split into separate lists, and thus will never be compared.

To complete our analysis, we need to identify $\Pr(z_i \text{ is compared to } z_j)$ in terms of i and j, the summation indices in Eq. (6). Recall that the choice of z_i is independent of z_j and that we choose the pivot uniformly from the set Z_{ij}, which has $j - i + 1$ elements. Thus, we can write

$$\begin{aligned} &\Pr(z_i \text{ is compared to } z_j) \\ &\quad = \Pr(\text{either } z_i \text{ or } z_j \text{ is first pivot chosen in } Z_{ij}) \\ &\quad = \Pr(z_i \text{ is first pivot chosen in } Z_{ij}) + \Pr(z_j \text{ is first pivot chosen in } Z_{ij}) \\ &\quad = \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &\quad = \frac{2}{j-i+1} \ . \end{aligned} \tag{7}$$

Substituting this result into Eq. (6), we can now complete our analysis

$$\begin{aligned}
E(X) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j) \ . \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} && \text{change of variables } k = j-i && (8) \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} && \text{lower bound} && (9) \\
&= \sum_{i=1}^{n-1} O(\log n) && \text{harmonic series bound} && (10) \\
&= O(n \log n) \ . && && (11)
\end{aligned}$$

Thus, the expected running time of Randomized Quicksort is $O(n \log n)$. This result is also a proof of the expected running time for (deterministic) Quicksort, when inputs are equally likely.

1.3 An alternative analysis, and a trick

There are several other ways to analyze the performance of Randomized Quicksort. Here is one that doesn't use any of the probabilistic tools we used above, but instead uses some tricks from recurrence relations. Let's consider the specific structure of the splits that `Partition` induces. Recall that generally speaking, the running time is given by the recurrence

$$T(n) = T(k-1) + T(n-k) + \Theta(n) \ , \tag{12}$$

where the value $k-1$ is the number of elements on the left side of the recursion tree, i.e., k tells us how good (balanced) the split is. If $k = 1$ then we have the worst case and if $k = n/2$ we have the best-case. More generally, if $k = O(1)$ the running time is $\Theta(n^2)$ and if $k = O(n)$ the running time is $O(n \log n)$. In Randomized Quicksort, all values of k are equally likely and thus the average running time is given by averaging Eq. (12) over all k:

$$\begin{aligned}
T(n) &= \frac{1}{n} \sum_{k=1}^{n} \left(T(k-1) + T(n-k) + \Theta(n) \right) \\
&= c_1 n + \frac{1}{n} \sum_{k=1}^{n} T(k-1) + \frac{1}{n} \sum_{k=1}^{n} T(n-k) \ ,
\end{aligned}$$

where we have made the asymptotic substitution $\Theta(n) = c_1 n$, for some constant c_1. We can simplify this by substituting $i = k-1$ in the first summation and $i = n-k$ in the second, and then multiplying through by n:

$$T(n) = c_1 n + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$
$$n\,T(n) = c_2 n^2 + 2\sum_{i=0}^{n-1} T(i) \ . \tag{13}$$

(Do you see why the summation bounds change the way they do?) Now we'll use a technique you can use to solve some recurrence relations: subtract a recurrence for a problem of size $n-1$ from the recurrence for a problem of size n. Using Eq. (13), this yields:

$$n\,T(n) - (n-1)T(n-1) = \left(c_2 n^2 + 2\sum_{i=0}^{n-1} T(i)\right) - \left(c_2(n-1)^2 + 2\sum_{i=0}^{n-2} T(i)\right)$$
$$n\,T(n) - (n-1)T(n-1) = 2T(n-1) + c_2(2n-1)$$
$$n\,T(n) = (n+1)T(n-1) + c_2(2n-1) \ , \tag{14}$$
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{c_2(2n-1)}{n(n+1)} \ . \tag{15}$$

Where we divided both sides of Eq. (14) by $n(n+1)$ to produce Eq. (15).

Now, one last change of variables: let $T(n)/(n+1) = D(n)$, and thus $T(n-1)/n = D(n-1)$. This yields:

$$D(n) = D(n-1) + O(1/n)$$
$$= O(\log n) \ ,$$

where the last line follows from a bound on the harmonic series. Thus, reversing the substitution for $D(n)$, we see that $T(n) = O(n \log n)$.

1.4 Alternative Quicksorts

There are other ways to change Quicksort to avoid the worst-case behavior, all of which focus on choosing good pivot elements; some of these are discussed in the textbook.

The most popular of these alternatives is to use a `Median-of-k` algorithm to choose the pivot element. In this version, the `Partition` algorithm selects the pivot as the *median* of k elements, where typically $k = O(1)$ ($k = 3$ is a common choice). The larger k, the closer the identified element

is to the true median of a particular subproblem. The closer the pivot is to the true median, the more balanced a division is made. This strategy further reduces the number of input sequences that yield a worst-case performance. (What happens if we set $k = n$ in this algorithm?)

2 Next time

1. Randomized data structures: skip lists and hash tables
2. Read Chapter 11 (Hash tables)

An aside: thinking algorithmically to find a duplicate

As a small aside, consider the problem of finding a duplicate value in an array of otherwise unique values:

You're given a list of $n+1$ real-valued numbers $\{x_i\}$, stored in an array A. Exactly n of the values of unique, but one is a duplicate. Describe an efficient algorithm to find it and give its asymptotic running time.

To again illustrate how to think carefully about a project like this, we'll consider two solutions, one naïve and one efficient.

A naïve approach

Let's assume that the values in the list $\{x_i\}$ are integers on the unbounded interval $x_i \in (-\infty, \infty)$, and let's apply the *histogram* method, in which we allocate a second array B to store the histogram, or the frequency f_i of the ith integer.

In order to allocate B, we must first identify the range that the x_i values span since B will contain $m = \max_i A[i] - \min_i A[i] + 1$ elements. We could do this by running over each element in A and keeping track of the largest and smallest values we observe, denoted $maxA$ and $minA$ respectively. This step of our algorithm takes $\Theta(n)$ time. Once we've allocated B, we then loop over the elements of A a second time and increment the corresponding count variable in B, i.e, `B[minA+A[i]]++` (why do we index into B in this way?). When this is finished, the elements of B are a histogram of the values in A. Finally, we loop over B looking for any element with `B[i]>1`; alternatively, if we are confident there is only a single duplicate, before we increment a count variable, we could first check if it has previously been incremented; if so, we have found the duplicate and we can terminate. (Does this second strategy change the worst-case running time? Where would the adversary place the duplicate in A in order to guarantee the worst case?)

How long does this algorithm take? How much memory does it take? Remember that we must assume the adversarial model of algorithm analysis—what's the worst kind of input A that the adversary could give us? The answer is that this algorithm takes time $O(maxA - minA)$ and that there is no bound on how large $maxA - minA$ could be. Thus, the adversary could choose the an input in which n of the values of consecutive integers starting at 1, but with $maxA = n^n$ or worse. Because the running time depends mainly on the time to allocate B, there is no limit to how slow this algorithm could be.

An efficient approach

A better solution is to dispense with the histogram solution completely and instead use a sorting approach. As we discussed in class, we could use an efficient sorting algorithm like QuickSort or MergeSort to reorder the $\{x_i\}$ in ascending order. This step takes $O(n \log n)$ time. To find the duplicate, we simply examine each element in turn and compare it with the preceding element; if this comparison is ever true, we return either of the two elements. That is, for each i, we ask `A[i]==A[i+1]` and we return `A[i]` if the comparison is true.

This algorithm can be naturally generalized to solve the problem of finding the *mode* of the list of numbers, i.e., the most frequent value, not just a single duplicate. Again, sort the input array. Now, store the first element `s=A[1]`, initialize a counter $c_s = 1$ (which will count the frequency of the value stored in s), initialize a value `t=NULL` and a second counter $c_t = -\infty$. The idea is to iterate through the list using c_s to measure the frequency of the value stored in s; because the array is sorted, all of these values much occur together. When we encounter the first value that is different from s, we check to see whether the $c_t < c_s$, i.e., if the value stored in s occurs more frequently than the value stored in t (our old "most common" value). Is it does, then we store the new candidate for most common value ($t \leftarrow s$ and $c_t \leftarrow c_s$), increment i and set $s = A[i]$ and $c_s = 1$ and proceed.

At the end of the loop, we check if $c_t < c_s$; if so, we return s as the mode; if not, we return t. (Can you prove that this algorithm is correct? That is, that it returns the correct answer on all inputs? What is the worst case input?)

A general solution

The sorting approach solves the problem of finding the modal value, but it does not solve the more general problem of finding the frequencies of all the values. Fortunately, from a computational point of view, finding the frequencies of *all* the values is *not* more expensive than finding the frequency of only one value. Formally, I claim that the time required to find the most frequent value and the time required to find the frequencies of all values (the histogram) are asymptotically equal, and both take $O(n \log n)$ time. How can this be?

It should be clear that if I can construct the histogram in $O(n \log n)$ time, I can return the kth most frequent value by simply sorting the values by their frequencies and then returning the kth value. Because sorting takes $O(n \log n)$ time, it is no more expensive than the construction itself, which makes this step "free" with respect to the total asymptotic running time. (In fact, I can do a constant number of any operations with that running time for "free" and not change the asymptotics.) Thus, it only remains to show that I can construct the histogram in $O(n \log n)$ time.

The naïve histogram method suffers from allocating an array that spans the interval $[minA, maxA]$, which could be expensive because $maxA$ could be extremely large. In our general solution, we will dispense with the second array B and instead count only the values that actually occur. In this way, we can avoid the problem identified above, which is that the values of $\{x_i\}$ may be sparsely distributed over the range $[minA, maxA]$, and the running time above depends on just how sparse they are. If we are sloppy, a clever adversary could make our algorithm take an infinite amount of time by simply adding a single value y to the list where y is infinitely large. To do this, we will use a self-balancing binary search tree data structure to implement a *sparse vector*.

Recall that a self-balancing binary search tree, like an AVL tree or a red-black tree (Chapter 13), stores a set of tuples of the form `(key,value)`. For our purposes, let the "keys" be the observed values x_i and the "values" be their frequencies in A. To begin, we initialize our sparse vector to be empty and then begin iterating over the elements of A, in order. For each element x_i, we perform a `find(x_i)` operation on our sparse vector; if it returns a `NULL`, then we know that we have not encountered this value before and then call `insert(x_i,1)` to add it to the histogram; otherwise, it returns a value f, which is the frequency of x_i and we then call `delete(x_i)` followed by `insert(x_i,f+1)` in order to increment its frequency.

Find, insert and delete operations for self-balancing binary search trees all take $O(\log n)$ time, and thus for n values, this algorithm takes $O(n \log n)$ time to complete and $O(n)$ additional memory. To read out the histogram in sorted order, we then do an in-order traversal (a.k.a., depth-first search) through the tree structure and store the results in a new auxiliary array B. Thus, constructing the histogram can be done in $O(n \log n)$ time.