Design and Analysis of Algorithms | CSCI 5454
Lecture 2 | 19 January 2011
Prof. Aaron Clauset

1 An aside: finding a duplicate

Before we finish our analysis of Quicksort, let's revisit the "finding a duplicate" puzzle we briefly discussed last time in class. Recall the following problem on the pre-class survey:

You're given a list of $n+1$ real-valued numbers $\{x_i\}$ (represented as 64-bit floating points). 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.

Last time, we discussed an efficient solution using a comparison-based sorting algorithm like Quicksort or Mergesort to first sort the $n+1$ values in the array A, and then use a simple loop to find the i for which `A[i]=A[i+1]` is true. Because sorting takes $O(n \log n)$ and the loop takes $O(n)$, this algorithm takes $O(n \log n)$ time and $O(1)$ additional memory.

One proposal from the class, which we did not go over in detail, was to use the naïve algorithm from Lecture 0 to find it. However, that algorithm was designed only to handle a problem in which the input array contains each integer value $x_i \in [10^0 \ldots 10^6]$ at least once, but only one value twice. This algorithm can indeed be adapted to this more general task, but it must be done carefully.

1.1 A naïve approach

The *histogram* method we discussed in class only works for integers, i.e., for $x_i \in \mathbb{N}$, because it relies on allocating a second array B with a size equal to the range of x_i.[1] Thus, we need to use a non-array-based method. As an exercise, however, let's consider a slight variation of the general problem that allows an array-based solution: suppose that the values $\{x_i\}$ are actually integers on the unbounded interval $x_i \in (-\infty, \infty)$.

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

[1] In principle, since the values are 64-bit floating points, we could circumvent this problem by noting that 64-bit floats can represent exactly 2^{64} unique values, and thus we could allocate an array B with 2^{64} elements, one for each possible value we might encounter in A. This is not a particularly efficient solution. Generalizing the precision of the values to $\log_2 n$-bit (why is this the right level of precision?), we encounter another problem: which value does the ith counter in B correspond to? Keeping track of this information efficiently means, effectively, using the approach in Section 1.2.

and keeping track of the largest and smallest values we observe, denoted $maxA$ and $minA$ respectively. This is an $\Theta(n)$ algorithm. 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.

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.

1.2 An efficient approach

A better version dispenses with using arrays to build the histogram and instead counts only the values actually observed, rather than all possible values on some interval. In this way, it avoids the problem identified above. That is, the values in A may be sparsely distributed over the range $[minA, maxA]$, and the running time above depends on just how sparse they are. We can use what's called a *sparse vector* data structure to do this cleverly.

Recall that a balanced binary tree (BBT) data structure, like an AVL tree or a red-black tree (Chapter 13), stores a set of tuples of the form `(key,value)`. For this problem, let the "keys" be the observed values x_i and the "values" be their counts in the input array A. Now, we run through the elements of A in order. For each element x_i, we perform a `find(x_i)`; if it returns a `NULL`, then we know that we have not encountered this value before; otherwise, we have found the duplicate. To remember it for the future, we add it to our list by performing an `insert(x_i)` operation.

Find and insert for balanced binary tree data structures both 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. This solution can easily be generalized to compute the entire histogram for the input A, by simply incrementing the "value" associated with a particular "key", whenever `find(x_i)` does not return a `NULL`. The contents of the tree can then be transferred into an array in $O(n \log n)$ time.

2 Quicksort, continued

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 $O(n \log n)$. (Why?) This time, we'll 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 structure of the input array. For the moment, assume that each of the $n!$ possible orderings of n elements is equally likely. If we are unlucky enough to be given a badly structured input, then we will choose $\Theta(n)$ bad pivots and pay a $\Theta(n)$ cost each time. Recall from our heuristic argument for 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, which does not change the asymptotic performance. Thus, to fall into the $\Theta(n^2)$ worst case, we need to make bad choices more than a constant fraction of the time, e.g., a $1 - o(1)$ fraction of the time. Put another way, we would need to choose a good pivot only a constant number of times, regardless of the size of the input. (Do you see why?)

If input sequences 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?) Unfortunately, we are using the adversarial model of algorithm analysis which means we can't be so optimistic. Can we change Quicksort to behave more like the average case on all inputs?

2.1 Randomized Quicksort

The answer is yes. By making use of a pseudo-random number generator to simulate random choices of the pivot element,[2] we can make a randomized version of Quicksort that performs well on almost all inputs. This is our first example of a *randomized algorithm*.[3]

[2]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.

[3]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.

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 = Randomized-Partition(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(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
   for (j=p; j<=r-1;j++) {
      if A[j]<=x {
         i++
         exchange(A[i],A[j])
   }}
   exchange(A[i+1],A[r])
   return i+1
},
```

where `random(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.

2.2 Analysis

Because we are using a source of random bits to randomize the decisions of Quicksort, the running time of Randomized Quicksort is 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, which covers introductory probability theory.)

2.2.1 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

$$E(X) = \sum_x x \Pr(X = x) \text{ where } \sum_x \Pr(X = x) = 1 \tag{1}$$

$$E(X) = \int_x x \Pr(x) \mathrm{d}x \text{ where } \int_x \Pr(x) \mathrm{d}x = 1 \tag{2}$$

where Eq. (1) applies to discrete variables and Eq. (2) applies to continuous variables. The right-hand side of the statement verifies that the function $\Pr(x)$ 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 $\Pr(X = 1 \text{ or } Y = 1) = \Pr(X = 1) + \Pr(Y = 1) = 1/6 + 1/6 = 1/3$.

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

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

2.2.2 Analyzing Randomized Quicksort

Armed with these tools, we can analyze the behavior of Randomized Quicksort. For convenience, some notation. Let the input array $A = [z_1, z_2, \ldots, z_n]$ and let A be already sorted, that is, z_i is the ith smallest element of A. Further, let the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ denote those elements between z_i and z_j, inclusive.

When does Randomized Quicksort compare z_i and z_j? At most, this comparison will occur once, and this will happen 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. That is,

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

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

By linearity of expectations, we can simply 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

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

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, which appear as 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

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

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

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

where we have used a change of variables $k = j - i$ in Eq. (8), an inequality in Eq. (9) and a bound on the harmonic series in Eq. (10) to arrive at our result in Eq. (11).

Thus, the expected running time of Randomized Quicksort is $O(n \log n)$. (Do you see why this is also a proof of the expected running time for Quicksort, if inputs are equally likely?)

2.3 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. 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 are in the worst-case, while if $k = n/2$ we are in the best-case. More generally, if $k =$ constant, we experience $\Theta(n^2)$ performance, while if $k = O(n)$ we experience $O(n \log n)$ performance. In Randomized Quicksort, all values of k are equally likely and thus the 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:

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

(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:

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

$$n\,T(n) - (n-1)T(n-1) = 2T(n-1) + 2c_2(n-1)$$

$$n\,T(n) = (n+1)T(n-1) + 2c_2(n-1)\ , \tag{14}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c_2(n-1)}{n(n+1)}\ . \tag{15}$$

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:

$$\begin{aligned} D(n) &= D(n-1) + O(1/n) \\ &= O(\log n)\ , \end{aligned}$$

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

2.4 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. The larger k, the closer the identified element is to the true median of a particular subproblem, and, the closer the pivot is to the element, the more balanced a division is made. This, in turn, further reduces the number of input sequences that yield a worst-case performance. (What happens if we make the logical jump and set $k = n$?)

3 Next time

1. Randomized data structures: skip lists and hash tables
2. Read Chapter 11 (Hash tables)
3. Office hours this week on Thursday, January 20 (2:00pm-3:30pm)