CS61B: Lecture 38
Monday, April 29, 2013

RANDOMIZED ANALYSIS
===================
_Randomized_algorithms_ are algorithms that make decisions based on rolls of the dice. The random numbers actually help to keep the running time low. Examples are quicksort, quickselect, and hash tables with random hash codes.

Randomized analysis, like amortized analysis, is a mathematically rigorous way of saying, "The average running time of this operation is fast, even though the worst-case running time is slow." Unlike amortized analysis, the "average" is taken over an infinite number of runs of the program. A randomized algorithm will sometimes run more slowly than the average, but the probability that it will run _asymptotically_ slower is extremely low.

Randomized analysis requires a little bit of probability theory.

Expectation
-----------
Suppose a method x() flips a coin. If the coin comes up heads, x() takes one second to execute. If it comes up tails, x() takes three seconds.

Let X be the exact running time of one call to x(). With probability 0.5, X is 1, and with probability 0.5, X is 3. For obvious reasons, X is called a _random_variable_.

The _expected_ value of X is the average value X assumes in an infinite sequence of coin flips,

```
  E[X] = 0.5 * 1 + 0.5 * 3 = 2 seconds expected time.
```

Suppose we run the code sequence

```
  x();     // takes time X
  x();     // takes time Y
```

and let Y be the running time of the _second_ call. The total running time is T = X + Y. (Y and T are also random variables.) What is the expected total running time E[T]?

The main idea from probability we need is called _linearity_of_expectation_, which says that expected running times sum linearly.

```
  E[X + Y] = E[X] + E[Y]
           = 2 + 2
           = 4 seconds expected time.
```

The interesting thing is that linearity of expectation holds true whether or not X and Y are _independent_. Independence means that the first coin flip has no effect on the outcome of the second. If X and Y are independent, the code will take four seconds on average. But what if they're not? Suppose the second coin flip always matches the first--we always get two heads, or two tails. Then the code still takes four seconds on average. If the second coin flip is always the opposite of the first--we always get one head and one tail--the code still takes four seconds on average.

So if we determine the expected running time of each individual operation, we can determine the expected running time of a whole program by adding up the expected costs of all the operations.

Hash Tables
-----------
The implementations of hash tables we have studied don't use random numbers, but we can model the effects of collisions on running time by pretending we have a random hash code.

A _random_hash_code_ maps each possible key to a number that's chosen randomly. This does _not_ mean we roll dice every time we hash a key. A hash table can only work if a key maps to the same bucket every time. Each key hashes to a randomly chosen bucket in the table, but a key's random hash code never changes.

Unfortunately, it's hard to choose a hash code randomly from all possible hash codes, because you need to remember a random number for each key, and that would seem to require another hash table. However, random hash codes are a good _model_ for how a good hash code will perform. The model isn't perfect, and it doesn't apply to bad hash codes, but for a hash code that proves effective in experiments, it's a good rough guess. Moreover, there is a sneaky number-theoretical trick called _universal_hashing_ that generates random hash codes. These random hash codes are chosen from a relatively small set of possibilities, yet they perform just as well as if they were chosen from the set of all possible hash codes. (If you're interested, you can read about it in the textbook "Algorithms" by Cormen, Leiserson, Rivest, and Stein.)

Assume our hash table uses chaining and does not allow duplicate keys. If an entry is inserted whose key matches an existing entry, the old entry is replaced.

Suppose we perform the operation find(k), and the key k hashes to a bucket b. Bucket b contains at most one entry with key k, so the cost of the search is one dollar, plus an additional dollar for every entry stored in bucket b whose key is not k. (Recall from last lecture that a _dollar_ is a unit of time chosen large enough to make this statement true.)

Suppose there are n keys in the table besides k. Let V1, V2, ..., Vn be random variables such that for each key ki, the variable Vi = 1 if key ki hashes to bucket b, and Vi is zero otherwise. Then the cost of find(k) is

```
  T = 1 + V1 + V2 + ... + Vn.
```

The expected cost of find(k) is (by linearity of expectation)

```
  E[T] = 1 + E[V1] + E[V2] + ... + E[Vn].
```

What is E[Vi]? Since there are N buckets, and the hash code is random, each key has a 1/N probability of hashing to bucket b. So E[Vi] = 1/N, and

```
  E[T] = 1 + n/N,
```

which is one plus the load factor! If we keep the load factor n/N below some constant c as n grows, find operations cost expected O(1) time.

The same analysis applies to insert and remove operations. All three hash table operations take O(1) expected amortized time. (The word "amortized" accounts for table resizing, as discussed last lecture.)

Observe that the running times of hash table operations are _not_ independent. If key k1 and key k2 both hash to the same bucket, it increases the running time of both find(k1) and find(k2). Linearity of expectation is important because it implies that we can add the expected costs of individual operations, and obtain the expected total cost of all the operations an algorithm performs.

## Quicksort

Recall that mergesort sorts n items in O(n log n) time because the recursion tree has 1 + ceiling(log_2 n) levels, and each level involves O(n) time spent merging lists. Quicksort also spends linear time at each level (partitioning the lists), but it is trickier to analyze because the recursion tree is not perfectly balanced, and some keys survive to deeper levels than others.

To analyze quicksort, let's analyze the expected depth one input key k will reach in the tree. (In effect, we're measuring a vertical slice of the recursion tree instead of a horizontal slice.) Assume no two keys are equal, since that is the slowest case.

Quicksort chooses a random pivot. The pivot is equally likely to be the smallest key, the second smallest, the third smallest, ..., or the largest. For each case, the probability is 1/n. Since we want a roughly balanced partition, let's say that the least floor(n/4) keys and the greatest floor(n/4) keys are "bad" pivots, and the other keys are "good" pivots. Since there are at most n/2 bad pivots, the probability of choosing a bad pivot is <= 0.5.

If we choose a good pivot, we'll have a 1/4-3/4 split or better, and our chosen key k will go into a subset containing at most three quarters of the keys, which is sorted recursively. If we choose a bad pivot, k might go into a subset with nearly all the other keys.

Let D(n) be a random variable equal to the deepest depth at which key k appears when we sort n keys. D(n) varies from run to run, but we can reason about its expected value. Since we choose a bad key no more than half the time,

$$E[D(n)] \le 1 + 0.5\, E[D(n)] + 0.5\, E[D(3n / 4)].$$

Multiplying by two and subtracting E[D(n)] from both sides gives

$$E[D(n)] \le 2 + E[D(3n / 4)].$$

This inequality is called a _recurrence_, and you'll learn how to solve them in CS 170. (No, recurrences won't be on the CS 61B final exam.) The base cases for this recurrence are D(0) = 0 and D(1) = 0. It's easy to check by substitution that a solution is

$$E[D(n)] \le 2 \log_{4/3} n.$$

So any arbitrary key k appears in expected O(log n) levels of the recursion tree, and causes O(log n) partitioning work. By linearity of expectation, we can sum the expected O(log n) work for each of the n keys, and we find that quicksort runs in expected O(n log n) time.

## Quickselect

For quickselect, we can analyze the expected running time more directly. Suppose we run quickselect on n keys. Let P(n) be a random variable equal to the total number of keys partitioned, summed over all the partitioning steps. Then the running time is in Theta(P(n)).

Quickselect is like quicksort, but when we choose a good pivot, at least one quarter of the keys are discarded. We choose a good pivot at least half the time, so

$$E[P(n)] \le n + 0.5\, E[P(n)] + 0.5\, E[P(3n / 4)],$$

which is solved by E[P(n)] <= 8n. Therefore, the expected running time of quickselect on n keys is in O(n).

## Amortized Time vs. Expected Time

There's a subtle but important difference between amortized running time and expected running time.

Quicksort with random pivots takes O(n log n) expected running time, but its worst-case running time is in Theta(n^2). This means that there is a small possibility that quicksort will cost Omega(n^2) dollars, but the probability of that happening approaches zero as n approaches infinity.

A splay tree operation takes O(log n) amortized time, but the worst-case running time for a splay tree operation is in Theta(n). Splay trees are not randomized, and the "probability" of an Omega(n)-time splay tree operation is not a meaningful concept. If you take an empty splay tree, insert the items 1...n in order, then run find(1), the find operation _will_ cost n dollars. But a sequence of n splay tree operations, starting from an empty tree, _never_ costs more than O(n log n) actual running time. Ever.

Hash tables are an interesting case, because they use both amortization and randomization. Resizing takes Theta(n) time. With a random hash code, there is a tiny probability that every item will hash to the same bucket, so the worst-case running time of an operation is Theta(n)--even without resizing.

To account for resizing, we use amortized analysis. To account for collisions, we use randomized analysis. So when we say that hash table operations run in O(1) time, we mean they run in O(1) _expected_, _amortized_ time.

| | |
|---|---|
| Splay trees | O(log n) amortized time / operation * |
| Disjoint sets (tree-based) | O(alpha(f + u, u)) amortized time / find op ** |
| Quicksort | O(n log n) expected time *** |
| Quickselect | Theta(n) expected time **** |
| Hash tables | Theta(1) expected amortized time / op ***** |

If you take CS 170, you will learn an amortized analysis of disjoint sets there. Unfortunately, the analyses of both disjoint sets and splay trees are complicated. Goodrich & Tamassia give the amortized analysis of splay trees, but you're not required to read or understand it for this class.

*     Worst-case time is in Theta(n), worst-case amortized time is in Theta(log n), best-case time is in Theta(1).

**    For find operations, worst-case time is in Theta(log u), worst-case amortized time is in Theta(alpha(f + u, u)), best-case time is in Theta(1). All union operations take Theta(1) time.

***   Worst-case time is in Theta(n^2)--if we get worst-case input AND worst-case random numbers. "Worst-case expected" time is in Theta(n log n)--meaning when the _input_ is worst-case, but we take the average over all possible sequences of random numbers. Recall that quicksort can be implemented so that keys equal to the pivot go into a separate list, in which case the best-case time is in Theta(n), because the best-case input is one where all the keys are equal. If quicksort is implemented so that keys equal to the pivot are divided between lists I1 and I2, as is the norm for array-based quicksort, then the best-case time is in Theta(n log n).

****  Worst-case time is in Theta(n^2)--if we get worst-case input AND worst-case random numbers. Worst-case expected time, best-case time, and best-case expected time are in Theta(n).

***** Worst-case time is in Theta(n), expected worst-case time is in Theta(n) (worst case is when table is resized), amortized worst-case time is in Theta(n) (worst case is when every item is in one bucket), worst-case expected amortized time is in Theta(1), best-case time is in Theta(1). Confused yet?