cs124 section

abhi shelat

March 5, 1997

Ammortized Analysis

There are three methods in this section. I find the aggregate method the most convincing, and have given two examples of its use. The accounting and potential methods might be useful later in the course.

1.1 Aggregate method

The point here is to bound the worst-case time for a series of n operations, and then amortize this cost over the n operations to get an idea of how much a single op would cost. The idea goes way beyond that. We can use this to count the number of elements in a complicated set, or bound the probability on some occurence. The idea is to group things together in a novel way that is perhaps easier than the analyzing it in the conventional way.

An example:

Problem 1 *18.1-3 (page 360)*

A sequence of n operations is performed on a data structure. The ith operation costs :

$$c_i = \begin{cases} i & i = 2^j \\ 1 & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation.

1.2 Accounting method

In the aggregate method, all operations are assumed to have the same cost. In the accounting method, we keep tidy account of how much an operation costs, and charge exactly that much, keeping track of "credits" as we go along. We just need to guarantee that we are always *in the black*. How? We charge just enough for each operation.... Lets see how this works with the problem we solved above:

1.3 potential method

The book is better here... I have no insight to add.

Examples

Here are a few real examples of amortized analysis.

2.1 Sorting

You have written a super duper sort, but you are not sure that it is working correctly. How can you test the output of your sort to make sure that it is right? Well, you can check that the values are all increasing in linear time, but that does not guarentee that everything is correct. What else might go wrong?

M. Blum has devised an ingenous method for determining whether two sets of numbers are permutations of one another. It is a randomized algorithm, and we use the aggregate method to compute the probability that it *lies* to us! We can use this trick to check the sort. Oh, it runs in $O(n)$ time of course!

The insight is to define the function :

$$f(x) = (x - a_1)(x - a_2) \dots (x - a_n)$$

Lets take it from there:

2.2 Universal Hashing

This example also uses the aggregate method, but pertains to hashing, the second topic of todays section.

Suppose we have a static dictionary of elements that need to be placed into a data structure that allows for fast queries (imagine a dictionary). This result proposes a construction which guarentees $O(1)$ lookups, in fact, only 5 memory accesses and a few mult/divs are necessary.

First, let M be the universe of objects, and let $S \subset M$ be the static set of elements. Let $m = |M|, s = |S|$, and assume that we can allocate a hash table with r buckets. Let $p = m + 1$ be a prime (the result holds even if $m + 1$ is not a prime, we just need a prime that is larger than m).

Consider the family of hash functions:

$$h_k(x) = (kx \mod p) \mod r$$

where $k = \{1, 2, \ldots, m\}$.

Although this family of hash functions is not universal, it is a damn good one in terms of speed and performance. To argue its merits, lets analyze how it acts in the worst case. Define $B_i(k, r, S) = \{x \in S | h_k(x) = i\}$. That is, B_i is the set of elements that hash to bucket i. Let $b_i(k, r, S) = |B_i(k, r, S)|$.

Lets count the total number of $(k, \{x, y\})$ pairs that satisfy the following conditions:

1. $x, y \in S$ and $x \neq y$.
2. $h_k(x) = h_k(y)$

That is, we count the total number of tuples that hash to the same place; call this value C. We can do this using the techniques of amortization.

First, fix an $\{x, y\}, x \neq y$ pair. This pairs total *contribution* to C will be equal to the number of ks that satisfy condition (2), i.e,:

$$(kx \mod p) - (ky \mod p) = 0 \mod r$$

$$k((x - y) \mod p) = \pm 0, \pm r, \pm 2r, \ldots, \pm \lfloor \frac{(p-1)}{r} \rfloor r$$

But remember that p is a prime, so that Z_p is a field! This means that every element in Z_p has a multiplicative inverse (well, except 0). This is equivalent to saying that equations of the form

$$k(x - y) \mod p = jr$$

have unique solutions. Hence, the total contribution to C given by the pair $\{x, y\}$ is less than $\frac{2(p-1)}{r}$.

Now we just loop over every possible $\{x, y\}$ pair in S. Hence, we end up with the result:

$$C = \binom{s}{2}\left(\frac{2(p-1)}{r}\right) \leq \frac{s^2(p-1)}{r}$$

So where do we go from here?