CS 124 Section #8	Randomized Algorithms	4/4/11

1 Definitions

A randomized algorithm is an algorithm that is able to make fair coin tosses during its execution. Unlike a deterministic algorithm, when you describe how a randomized algorithm works, you can say "choose x randomly" rather than having to specifically say which x you are choosing. As discussed in lecture, the two main types of randomized algorithms are:

1. **Las Vegas Algorithm:** Always produces the correct result, but run-time is dependent on the random choices made during the algorithm.
   - **Analysis:** Once we prove that the algorithm is correct, we also need to calculate that its *expected* run-time. This allows us to use inequalities like Markov's to say that there is a high chance the algorithm will be efficient.
   - **Example from Class:** Quicksort and Quickselect
2. **Monte Carlo Algorithm:** Always finishes in a set amount of time, but the result may or may not be correct.
   - **Analysis:** We want to find a lower bound on the probability that the algorithm produces the correct result. For example, if we show that a Monte Carlo algorithm X is correct at least $\frac{1}{2}$ of the time, then we can simply run X multiple times and our chances of getting an incorrect answer diminish exponentially.
   - **Example from Class:** Freivalds algorithm for matrix multiplication verification

2 Integer Multiplication Checking

In this problem, we are given three integers a, b and c and want to determine whether or not $a \cdot b = c$. Suppose that $0 \leq a, b < 10^{250,000}$ and $0 \leq c < 10^{500,000}$ so actually performing the multiplication would not be feasible. Suppose that we are given a, b and c as strings (and thus do not have to worry about integer overflow on our machines).

Describe a Monte Carlo randomized algorithm that determines whether or not $a \cdot b = c$ and analyze it's accuracy.

Exercise. *As a first step, consider I told you to check whether* $23898239 \cdot 19392981 = 83431298313$ *is true. How can you tell me immediately that the answer is FALSE?*

Solution.
We can take both numbers mod 10, and look at it's last digit. Something ending in 9 times something ending in 1 can't be something ending in 3.

Exercise. *Generalize your strategy from above to come up with an algorithm that tests whether* $a \cdot b = c$.

Solution.
Let p be a large prime number. We simply check whether $a(\text{mod } p) \cdot b(\text{mod } p) = c(\text{mod } p)$. If equality does not hold, then we know immediately that $a \cdot b \neq c$. However, we may get false positives where the equality holds when taken mod p, but equality does not actually hold between $a \cdot b$ and c.

Exercise. *Using the Prime Number Theorem, which says that there are $\Theta(n/\ln n)$ primes less than n, bound the failure probability of your algorithm.*

Solution.
We get a false positive if $a \cdot b - c$ is a multiple of p. Let $d = |a \cdot b - c|$. We know that $d < 10^{500{,}000}$ so therefore d can have at most $500,000$ distinct prime factors. Suppose we randomly chose a prime number below 10^{18}. Then, by the prime number theorem, there were about $10^{18}/\ln(10^{18}) \approx 2.4 \times 10^{16}$ choices. In the worst case, each of the up to 500,000 prime divisors of d are in this range, so at worst there is a $\frac{500{,}000}{2.4 \times 10^{16}} = 4.2 \times 10^{-12}$ that p divides d. This is an upper bound on the failure probability, so $1 - 4.2 \times 10^{-12}$ is a lower bound on the probability of success, which as you can see is very very high.

Exercise. *Describe how you would implement such an algorithm that takes a, b and c as strings and returns whether $a \cdot b = c$.*

Solution.
We cannot simply convert a, b and c into integers because our numbers are absurdly large. Therefore, we will need to work with the given strings and extract digits as we go. The process of finding a mod p more efficiently is as follows:

Start with the left-most digit and move towards the right. Initialize a variable *total* to the left-most digit and iteratively multiply *total* by 10, add the next digit and mod by p. This allows us to in time linear in the number of digits of a, b and c, calculate them mod p.

3 Hashing

Remember from class that a hash function is a mapping $h : \{0, \ldots, n-1\} \to \{0, \ldots, m-1\}$. In most applications of hashing, you'll typically see $m < n$. (Why?) Hashing-based data structures are useful since they ideally allow for constant time operations (lookup, adding, and deletion), although collisions can change that if, for example, m is too large or you use a poorly chosen hash function. Why would these conditions generally increase the number of collisions?

In the following problems, we will assume that our hash function h is a simple uniform hash. This means that h evenly distributes $\{0, 1, 2, \ldots n-1\}$ among the m buckets. More formally, this means that $P(h(x) = h(y)) = \frac{1}{m}$ for all $x, y \in \{0, 1, 2, \ldots n-1\}$.

Exercise. *Suppose we use a hash function h to hash n distinct keys into m buckets. Assuming simple uniform hashing, what is the expected number of collisions? More formally, for how many distinct keys k_1, k_2 do we have $h(k_1) = h(k_2)$?*

Solution.
Let the indicator variable X_{ij} be 1 if $h(k_i) = h(k_j)$ and 0 otherwise for all $i \neq j$. Then, let X be a random variable representing the total number of collisions. It can be calculated as the sum of all the X_{ij}'s:

$$X = \sum_{i<j} X_{ij}$$

Now, we take the expectation and use linearity of expectation to get:

$$E[X] = E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}] = \sum_{i<j} P(h(k_i) = h(k_j)) = \sum_{i<j} \frac{1}{m} = \frac{n(n-1)}{2m}$$

Exercise. *Use your probability skills for these hashing problems:*

1. *Suppose I hash n items into m buckets with a simple uniform hash. What's the expected number of buckets that have exactly one item? At least 2 items? k items?*
2. *How large must n be so that the probability of a collision is at least $\frac{1}{2}$? (Don't actually work it out - just say how you'd do it)*

Solution.

1. The probability that a bucket has k items is $\binom{n}{k}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n-k}$. For $k = 1$, there are, on average, $n\left(1-\frac{1}{m}\right)^{n-1}$ buckets with exactly one item. The number with at least 2 items is:

$$m - (\#\text{ with 1 item}) - (\#\text{ with 0 items}) = m - n\left(1-\frac{1}{m}\right)^{n-1} - m\left(1-\frac{1}{m}\right)^{n}$$

2. This is just the birthday problem.

There are several ways to deal with collisions while hashing, such as:

- Linear probing: If $f(x)$ already has an item, try $f(x)+1, f(x)+2$, etc. until you find an empty location (all taken mod m).
- Chaining: Each bucket holds a set of all items that are hashed to it. Simply add x to this set.
- Double hashing: Use two hash functions: $f(i,x) = f_1(x)+if_2(x)$. If $f(0,x)$ is taken, try $f(1,x), f(2,x), etc.$ until you find an empty location. This generalizes linear probing.
- Cuckoo hashing: Again, use two hash functions and place x in either $f_1(x)$ or $f_2(x)$. If there's a collision with object y, push y out to it's other location and keep repeating until there are no more collisions.

3.1 Bloom Filters

A Bloom Filter is a probabilistic data structure, used for set membership problems, that are more space efficient than conventional hashing schemes. There are m bits and k hash functions $f_1, \ldots, f_k$. When adding an element x to the set, set bits $f_1(x), \ldots, f_k(x)$ to 1. To check if x is already in the set, check if the corresponding bits are set to 1. Typically, the buckets are split up into k tables, with each hash function "addressing" a single table.

Exercise. *Can you delete an element from a Bloom filter?*

Solution.
No. Try it! If you set the ith bit to 0, you have effectively deleted every other element x for which $f_j(x) = i$ for some j.

Bloom filters are probabilistic structures since it's possible to get false positives, but never false negatives. That is, querying for membership of y may return true if y hasn't been added to the set but will never return false if it has.

Exercise. *What's the probability of a false positive? That is, what is the probability that an element is actually not in the set, but the k hashes all turn out to be 1? Suppose that n items have been inserted into our bloom filter so far.*

Solution.
In a single table of size $\frac{m}{k}$, the probability that the relevant bit is 0 is $\left(1 - \frac{1}{m/k}\right)^n \approx e^{-nk/m}$. This means that none of the n items previously inserted have touched that particular bit. Therefore, the probability that that particular bit is 1 is $1 - e^{-nk/m}$. If we want a false positive, then for each of the k tables, that particular bit must be set to 1, so our final aswer is $\left(1 - e^{-nk/m}\right)^k$.