Notes: Randomized Algorithms

CS 3130/ECE3530: Probability and Statistics for Engineers

December 11, 2014

Average Runtime Complexity

Probability theory can help us understand the average-case behavior of our algorithms. This is set up as an expectation over all possible inputs that the algorithm could take. The following is a very simple example of the basic principle.

Say you have a list of n numbers in some unknown order, and you want to search these numbers for a particular value a, which you know appears only once somewhere in the list. A deterministic search algorithm would be to loop through the list until you find the value, taking worst-case $O(n)$ time to complete. But in the best-case scenario the search could find the value in the first element of the list. Perhaps a better question is: what is the average running time of the search? This is an expectation problem!

Let T denote the time it takes to check one element in the list. Let X denote the random variable of how long it takes to find a in the list. Note that X can take the values $T, 2T, 3T, \ldots, nT$, i.e., X returns kT when the search succeeds on the kth element of the list. We'll assume that the list is randomly ordered, so that each position is equally likely to contain the a value. What is the expected value of X, i.e., what is the average running time of a simple search?

$$E[X] = \sum_{k=1}^{n} (kT) P(X = kT) = \sum_{k=1}^{n} kT \frac{1}{n} = \frac{T}{n} \sum_{k=1}^{n} k = \frac{T}{n} \frac{n(n+1)}{2} = T \frac{n+1}{2}$$

Monte Carlo Algorithms

Continuing with the search example, let's now say that we can only afford to look at $m < n$ elements of the list. This might be because the time to compare a single element of the list is very expensive, which might occur for example in an application like searching a large database for a person's face. One possibility is to just search through the first m elements of the list each time. However, in repeated use of this algorithm we will bias towards only finding elements that appear at the beginning of the list. A better strategy would be to pick random locations in the list to test.

Assuming we pick uniformly random locations **without replacement**, what is the probability that we will find the desired entry? If S denotes the event that we are successful, then

$$P(S) = \frac{m}{n}$$

What if we instead pick locations **with replacement**? This might happen if our list is not static, and we can't be sure that we aren't repeating the selection of the same element. In this case, it is easier to first think about the probability of the event of failure, F. The probability of failing on each attempt is $(n-1)/n$. Because these events are independent, we get

$$P(F) = \left(\frac{n-1}{n} \right)^m.$$

Using the complement rule, the probability of success is

$$P(S) = 1 - P(F) = 1 - \left(\frac{n-1}{n}\right)^m.$$

Miller-Rabin Primality Testing

RSA encryption is widely used in everyday secure internet traffic. To generate the RSA keys, it is necessary to first generate two very large pseudorandom prime numbers. Recall that a prime number is a number that is evenly divisible only by one and itself. But how do we test if a very large integer m is prime? We could try out all integers between 2 and $\sqrt{m}$ and see if they evenly divide m. But the number of tests here grows exponentially in the number of bits in m (and RSA encryption now typically deals with 1024 bit numbers). Another option is the Miller-Rabin test, which is a randomized algorithm to test the primality of an integer. In the spirit of Monte Carlo algorithms, it may not give a perfect answer, but it is very efficient, and we can make the probability of an incorrect answer as small as we desire.

The Miller-Rabin test is based on Fermat's Little Theorem, which states:

If m is a prime number and $1 \leq a < m$ is an integer, then

$$a^{m-1} \equiv 1 \bmod m.$$

In other words, this says that a^{m-1} will always have remainder 1 when divided by m, when m is prime. This gives a quick test to see if an integer is composite (not prime). Choose a random a less than m and see if the above test gives some remainder other than 1. If so, we can immediately label m as a composite number. In this case, a is called a "witness" that m is composite. However, we may find some a values that pass this test even if m is composite. In this case, we call a a "liar" – its trying to make us think m is prime when it is not. It turns out that there aren't that many liars. For *any* composite number, there are always less than 25% of the a integers less than it that are liars. So, each test we make has only a 25% chance of failure. If we repeat this test with k randomly selected a values, our probability of success is

$$P(S) = 1 - P(F) \geq 1 - \frac{1}{4^k}.$$

We can make this probability very close to one with pretty reasonable sizes for k. Also, this is a lower bound, the percentage of "liars" on average is actually much, much less.

The only difficult part of the test above is how to compute the $a^{m-1} \bmod m$. Just taking the power a^{m-1} will cause an overflow because m is typically large. The following trick is useful. Compute $k = \log_2(m-1)$, which is the number of bits in the binary expansion of $m-1$. Create the sequence:

$$\begin{aligned}
b_0 &= a \bmod m \\
b_1 &= a^2 \bmod m = b_0^2 \bmod m \\
b_2 &= a^4 \bmod m = b_1^2 \bmod m \\
&\vdots \\
b_k &= a^{2^k} \bmod m = b_{k-1}^2 \bmod m
\end{aligned}$$

Notice each b_i only requires squaring the previous result and taking the remainder modulo m. This way you prevent overflow. Now in the binary expansion of $m-1$, record which bits are a 1, and call these positions $i_1, i_2, \ldots, i_j$. Your final answer is

$$a^{m-1} \bmod m = (b_{i_1} \times b_{i_2} \times \cdots \times b_{i_j}) \bmod m.$$

Note: if you have a long string of b_{i_j} values to multiply, you will want to keep a cumulative multiplication of them as you compute them, and mod them after each new term you multiply (rather than waiting to mod the entire product at the end). This is needed to prevent overflow.

Example: $m = 11, a = 6$
In binary: $m - 1 = 10 = 1010_2$

$$\begin{aligned}
b_0 &= 6\\
b_1 &= 6^2 \bmod 11 = 3\\
b_2 &= 3^2 \bmod 11 = 9\\
b_3 &= 9^2 \bmod 11 = 4
\end{aligned}$$

Final answer $6^{10} \bmod 11 = (b_1 \times b_3) \bmod 11 = 12 \bmod 11 = 1$. So, 11 passes the Miller-Rabin test for being prime.