Probabilistic Algorithms

Reading from Special Topics textbook.

Chapter 5, pp. 131-144

Probabilistic Algorithms

- The algorithms that we have considered so far are all **deterministic**.
- They leave nothing to chance. Running a deterministic algorithm time after time with the same input will produce identical results each time.
- On the other hand, a **probabilistic** algorithm contains steps that make random choices by invoking a random (or pseudorandom) number generator.
- Thus, they are subject to the laws of chance. In particular, a probabilistic algorithm can perform differently for two runs with the same input.

Four Main Categories of Probabilistic Algorithms

- Randomizations of deterministic algorithms
- Monte Carlo algorithms
- Las Vegas algorithms
- Numerical probabilistic algorithms

Description of Types of Probabilistic Algorithms

1. **Randomization of a deterministic algorithm** results by replacing certain steps that made canonical choices by steps that make these choices in some random fashion. Randomization is done to break the connection between a particular input and worst-case behavior, and thereby homogenize the expected behavior of inputs to the algorithm.

2. **Monte Carlo algorithm**s often produce solutions very quickly, but only guarantee correctness with high probability.

3. **A Las Vegas algorithm** never outputs an incorrect solution but has some probability of reporting a failure to produce a solution.

4. **Numerical probabilistic algorithms** were among the first examples of introducing randomness into the design of algorithms. A classic example is the estimation of π obtained by throwing darts at a square and recording how many darts land inside a circle inscribed in the square.

Advantage of Probabilistic Algorithms

- In practice, obtaining solutions with high probability is almost as satisfactory as the foolproof guarantee provided by a deterministic algorithm.
- For many important problems, such as prime testing, which we'll discuss in this lecture, the most efficient algorithms currently known for their solutions are probabilistic.

Expected Number of Basic Operations

- Because running the algorithm twice with the same input I may result in a different number of basic operations being performed, $\tau(I)$ is no longer well defined.
- Instead, what is relevant is the expected number, $\tau_{\text{exp}}(I)$, of basic operations performed by the algorithm for input I with respect to the random choices made by the algorithm.
- As with any expectation, if we run the algorithm many times with fixed input I, then we can expect that the algorithm performs $\tau_{\text{exp}}(I)$ basic operations on average. If the algorithm performs many random choices, then even for a single run we can expect that the number of basic operations performed is very close to $\tau_{\text{exp}}(I)$.

Expected Complexities

Analogous to the functions *B*(*n*), *W*(*n*), and *A*(*n*) for the best-case, worst-case, and average complexities of a deterministic algorithm, we now have B_{exp}(n), $W_{\text{exp}}(n)$, $A_{\text{exp}}(n)$ for the expected best-case, worst-case, and average complexities of a probabilistic algorithm. These are defined by

$$B_{\exp}(n) = \min\{\tau_{\exp}(I) \mid I \in \mathscr{I}_{\mathrm{n}}\}$$

$$W_{\exp}(n) = \max\{\tau_{\exp}(I) \mid I \in \mathscr{I}_{\mathrm{n}}\}$$

$$A_{\exp}(n) = E(\tau_{\exp})$$

Randomized Quicksort

There are two versions:

1. Before each recursive call the an element $L[i]$ from the sublist list $L[low{:}high]$ is **chosen at random** and made the pivot element, i.e., $L[low]$ and $L[i]$ are swapped.

2. Before calling *Quicksort* we first randomize the list. This is called **stochastic preconditioning**. In particular, given an input list $L[0{:}n - 1]$ to *Quicksort,* we first call a procedure ***Permute*** with input list $L[0{:}n - 1]$ that randomly permutes the list elements.

PSN. Design the procedure *Permute* using the function *Random,* where *Random*(i, j), returns a random number (index) between i and j.

Sherwood algorithms

Quicksort has $\Theta(n\log n)$ average complexity, but $\Theta(n^2)$ worst-case complexity. A randomization of quicksort does not eliminate the possibility that an input performs $\Omega(n^2)$ basic operations, but it breaks the connection between an input and $\Omega(n^2)$ behavior.

More precisely, the randomizations of quicksort presented in this section will have $\tau_{\text{exp}}(I) = A(n)$ for any *any* input of size n, where $A(n)$ is the average behavior of ordinary quicksort.

In particular, the randomized versions of quicksort satisfy the ultimate homogenization condition:

$$B_{\text{exp}}(n) = A_{\text{exp}}(n) = W_{\text{exp}}(n).$$

Randomized algorithms satisfying this ultimate homogenization condition have been dubbed *Sherwood algorithms* in honor of Robin Hood.

Monte Carlo probabilistic algorithms

A Monte Carlo algorithm is a probabilistic algorithm that has a certain probability of returning the correct answer whatever input is considered.

The most useful class of Monte Carlo algorithms are those that have a probability of returning the correct answer greater than some fixed positive constant for any input.

More precisely, for a (fixed) real number, p, $0 < p < 1$, a ***p-correct*** *Monte Carlo* algorithm is a probabilistic algorithm that returns the correct answer with probability not less than p *no matter what input is considered*

False-Biased Monte Carlo Algorithm

- A Monte Carlo algorithm for a decision problem is ***false-biased*** if it is always correct when it returns the value **.false.** and only has some (hopefully small) probability of making a mistake when returning the value **.true.**.
- A similar definition holds for *true-biased* Monte Carlo algorithms.

Ramping up Correctness

Given a false-biased Monte Carlo algorithm *MC*, consider the following algorithm *MCRepeat*. In a particular application, the generic name *MC* will be replaced by the specific name of the Monte Carlo algorithm.

function *MCRepeat*(*k*)
Input: *k* (a positive integer)
Output: **.false.** if *MC* returns **.false.** for any invocation, **.true.** otherwise
    for *i* ← 1 **to** *k* **do**
        if *MC* returns **.false. then**
            return(**.false.**)
        endif
    endfor
    return (**.true.**)
end *MCRepeat*

MCRepeat can ramp up the correctness of a *p*-correct Monte Carlo algorithm *MC* to a number as close to unity as desired.

Ramping up Correctness cont'd

Proposition 5.3.1 Suppose that we have a p-correct false-biased Monte Carlo algorithm *MC*. Then the algorithm *MCRepeat*(k) is a $(1 – (1 – p)^k)$-correct false-biased Monte Carlo algorithm.

For example, if $p = \frac{1}{2}$, then the ramped-up Monte Carlo algorithm is incorrect with probability at less that or equal to $\left(\frac{1}{2}\right)^k$, which is infinitesimally small, even for relatively small k, say $k = 100$.

Prime Testing

PSN. Give a deterministic algorithm for testing whether a positive integer n is prime.

IsPrime not efficient for large integers

IsPrime is better than *IsPrimeNaive*, but is still very inefficient for large primes and would take zillions and zillions of years to terminate from *n* has hundreds of digits and such larger integers are which needed for cryptography applications.

We now describe a probabilistic Monte Carlo algorithm discovered by Miller and Rabin. It utilizes Fermat's Little Theorem.

Theorem (Fermat). Let *a* and *n* be positive integers, where n is prime and a is not divisible by *n*. Then,

$$a^{n-1} \equiv 1 \pmod{n}.$$

Pierre de Fermat
(1601-1665)

Fermat Test

- Based on Fermat's Little Theorem, we have a primality test, which we will refer to as the Fermat test
- In particular, we have the following false-biased Monte Carlo algorithm: choose a base a at random from $\{2, \ldots, n-1\}$ and return **.true.** (that is, the number is prime) if and only if $a^{n-1} \equiv 1 \pmod{n}$.
- Most composite numbers n fail the Fermat test for many integers a between 2 and $n - 1$. Thus, for such numbers, the Monte Carlo algorithm has a high probability of being correct.
- Unfortunately, there exist composite integers n for which $a^{n-1} \equiv 1 \pmod{n}$ for most $a < n$.
- In fact, there are composite numbers for which $a^n \equiv 1 \pmod{n}$ for **all** a, $1 < a < n$, that are relatively prime to n. They are called *Carmichael numbers*, named after the mathematician Robert Carmichael.
- Thus, if we choose a Carmichael numbers as an input to our algorithm, it has almost no chance of being correct.

Miller-Rabin Test

To obtain a stronger test, we look for a stronger condition satisfied by prime numbers. We observe that (for *n* odd)

$$a^{n-1} - 1 \equiv (a^{(n-1)/2} - 1)\,(a^{(n-1)/2} + 1).$$

Thus, we have by Fermat's little theorem that

$$(a^{(n-1)/2} - 1)\,(a^{(n-1)/2} + 1) \equiv 0 \pmod{n}.$$

Also, since *n* is prime, it follows that one of the two terms must be divisible by *n*, so that

$$a^{(n-1)/2} \equiv \pm 1 \pmod{n}.$$

If $a^{(n-1)/2} \equiv 1$ an (n – 1)/ 2 is even, then by the same reasoning it follows that $a^{(n-1)/4} \equiv \pm 1$. Similarly if $a^{(n-1)/4} \equiv 1$ and (n – 1)/4 is even, it follows that $a^{(n-1)/8} \equiv \pm 1$, and so forth. Thus, we have either

$$a^{n-1} \equiv a^{(n-1)/2} \equiv \ldots \equiv a^{(n-1)/2^{j-1}} \equiv 1,\ a^{(n-1)/2^{j}} \equiv -1 \pmod{n},$$

for some *j* or

$$a^{n-1} \equiv a^{(n-1)/2} \equiv \ldots \equiv a^{(n-1)/2^{k-1}} \equiv a^{(n-1)/2^{k}} \equiv 1 \pmod{n},$$

where 2^k is the largest power of 2 that divides $n - 1$.

Miller-Rabin Test cont'd

- Equivalently, letting m denote the largest odd number that divides $n - 1$, it follows that either

$$a^m \equiv 1 \pmod{n} \text{ or } a^{(n-1)/2^j} \equiv -1 \pmod{n}$$

for some j, $0 \leq j \leq k$, where 2^k is the largest power of 2 that divides $n - 1$.

- Testing whether a number n satisfies these conditions is called the **Miller-Rabin test**.

Miller-Rabin Test is a False-Biased .75 correct

It turns out that an odd composite number n will pass the Miller-Rabin test for at most 25% of all possible bases a, $2 \leq a \leq n - 1$. Thus, the Miller-Rabin test yields a false-biased .75-correct Monte Carlo algorithm for primality testing.

Pseudocode for the Miller-Rabin Primality Test

function *MillerRabinPrimalityTest*(n)
Input: n (an odd positive integer)
Output: **.true.** or **.false.** (always correct when **.false.** is returned, and correct at least 75% when .true. is returned)
$a \leftarrow Random(2, n-2)$

$k = n - 1$

// Fermat's Little Theorem test

if (a^k mod n) $\neq$ 1 **then return .false.**

// other tests based on using $a^k - 1 = (a^{k/2} - 1)(a^{k/2} + 1)$

while (k **mod** 2) $=$ 0 **do** // while k is even

$k = k/2$

if (a^k **mod** n) $== n - 1$ **then return .true.**

else if (a^k **mod** n) $\neq$ 1 **then return .false.**

end while

return .true.

end *MillerRabinPrimalityTest*

Why do plants hate math?

It gives them square roots.