CI583: Data Structures and Operating Systems
Introduction to probabilistic and non-deterministic algorithms

Outline

Non-determinism

This lecture is about a class of algorithms which are quite unlike the ones we've discussed so far. These are ones that make use of probability and randomised elements.

Some of these algorithms use probability to compute approximate results, often to NP-hard problems, and the longer they are allowed to run the more accurate the results. Others use random choices (called a coin toss) to solve problems more elegantly than can be done in a deterministic fashion.

The dining philosophers

The dining philosophers is a standard problem in concurrency and illustrates the issues of deadlock and starvation.

$n = 5$ philosophers are sitting at table, each with a plate of spaghetti in front of them, and n forks are on the table.

The dining philosophers

Philosophers do two things: think, and eat. They can only eat when two forks are available – if one or more of the forks in front of them are being used, they need to wait.

If they wait too long, they will starve. They never speak to each other.

The dining philosophers

Any solution has to be deadlock free (if one or more philosophers is hungry, at least one gets to eat) and starvation free (every philosopher eats eventually). Why can't we just instruct each hungry philosopher to wait for the two forks in front of him to become free?

The dining philosophers

Any solution has to be deadlock free (if one or more philosophers is hungry, at least one gets to eat) and starvation free (every philosopher eats eventually). Why can't we just instruct each hungry philosopher to wait for the two forks in front of him to become free?

If the left-hand fork is free but the right-hand one is taken, then the left-hand fork might have gone by the time the other one is free. Also, two philosophers might try to pick up the same fork at the same time.

If each philosopher picks up his left-hand fork, for example, no-one will ever eat eat.

The dining philosophers

In this problem, the act of eating can be considered a *critical section* – no adjacent philosophers can eat at the same time. Also, forks are *critical resources* – philosophers must take it in turns to use them.

In this way, the problem models many real-world situations in CS where processes compete for shared resources.

The dining philosophers

The standard solution is to introduce a new participant, the waiter. The waiter shows philosophers out of the room when they have finished eating. When they get hungry again, they queue up at the door.

The waiter keeps track of the philosophers at the table, restricting it to $n - 1$. If this is true, then at least two philosophers have an empty place next to them and at least one can eat.

The dining philosophers

Using a waiter works, but it requires us to expend extra resources.
Can we solve the problem without doing that?

We can, if we allow the philosophers to toss coins!

The dining philosophers

Let each philosopher do the following:

```
Repeat:
  1 Think, until hungry
  2 Toss coin to choose a direction, R or L
  3 Wait until fork in chosen direction is free, then
      pick it up
  4 If other fork is not available:
    4.1 Put down fork
    4.2 Go to 2
  5 Otherwise, lift second fork
  6 Eat
  7 Put down both forks and go to 1
```

The dining philosophers

By virtue of a coin toss, this solution is deadlock free with probability 1 for an infinite execution of the solution.

In order to achieve deadlock, all philosophers would have to toss their coins at the same time and get the same result (all left or all right), not just once but *every time*. The likelihood of this is zero.

This solution does allow starvation though – we can only overcome this by allowing philosophers to communicate with their immediate neighbours.

Probabilistic primes

The first algorithms to use probability in a systematic way were several methods to find prime numbers developed in the 1970s.

Generating prime numbers is not just a hobby for mathematicians – the ability to find large primes is an essential aspect of some important algorithms, including many that deal with encryption.

However, deterministic methods for testing primality are inefficient and impractical for very large numbers.

Probabilistic primes

Solutions that uses probability turn out to be surprisingly simple and reliable. The simplest (not the most accurate) is the Fermat primality test.

The idea is to search at random for numbers which are witnesses to a potential prime's compositeness (i.e. begin non-prime). If we find a witness, we can stop straight away.

The problem is to find a definition of a witness that can be tested efficiently and by which we know that, for each composite number, n, more than half of the numbers up to $n - 1$ will be witnesses. If we don't find one, this definition of witness will enable us to stop searching quite early on, with good confidence that n is prime.

Probabilistic primes

Far fewer than half of the numbers between 1 and n will divide it evenly, so $n\%a == 0$ won't do.

Fermat's Little Theorem states that, if p is prime and $1 \leq a < p$ then

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

The reason that this suits our needs for a definition of witness depend on some fairly heavy number theory. However, if we want to test whether p is prime, we can generate random values of a and see if the equality holds. If not, a is a witness that p is definitely composite. Otherwise, it is probably prime.

Probabilistic primes

If we run this algorithm on, say, 200 random numbers between 1 and p then the chances that it will say *prime* when it should *composite* are less than 1 in 2^{200}.

There are a very small number of non-primes that will pass this test ("Fermat liars") but these are so rare that this algorithm is more than adequate for many applications. A variation on it is used by the Pretty Good Privacy (PGP) encryption tool.

Probabilistic primes

This is a Monte Carlo algorithm.

We have a procedure that will can say “no” when the answer is negative.

Otherwise, it will say “maybe yes”, and when it does so the answer will actually be “yes” **more than half of the time**.

If we run it many times and the answer is always “maybe yes”, probability tells us that “yes” is very (very) likely to be true.