CMPSCI 187: Programming With Data Structures

Lecture #16: Applications of Queues
14 October 2011

Applications of Queues

- Substitution Ciphers
- L&C's Program For a Key Cipher
- Queues to Process Streams
- L&C's Simulation of Cinema Cashiers
- Randomness in Simulation
- Java's Random Number Generators

Substitution Ciphers

- Julius Caesar used a simple cryptosystem to send secret messages to his generals. Each letter was encoded as the third letter preceding, so D became A, E became B, and so on, with the last letters wrapping around. (So that in our alphabet A would become X, B would become Y, and C would become Z.)

- A **substitution cipher** is a system where individual letters are encoded, and the ciphertext letters occur in the same order as the plaintext letters. We can generalize Caesar's "-3" or "+23" cipher to add a chosen number to each letter, where the addition is taken "mod 26" (for the key k, we map x to (x+k)%26).

- This is easy to encode and decode by hand, but not very secure. Even if your enemy does not know the key, they can quickly determine which of the 26 possible keys you are using. And even an arbitrary mapping of letters to letters is subject to **frequency analysis** -- the most common ciphertext letter is likely to be the translation of E, the most common pair is TH, and so forth.

Key Ciphers

- A better method is to use multiple translation systems, and rotate among them every letter. A convenient way to do this is to adopt a short word as a **key**. Let's look at encoding the message "ET TU BRUTE" with the key "SPQR".

- The first, fifth, and ninth letters get encoded with key S, the second and sixth with key P, the third and seventh with key Q, and the fourth and eighth with key R. "Encoding E with key S" means converting both E and S to numbers (using A=0, B=1,..., Z=25, so that E=4 and S=18), adding the numbers mod 26 (getting 4+18=22), and converting the result back to a letter (W).

- Deciphering means finding the key letter for each ciphertext letter and subtracting it to get the plaintext letter.

```
Plaintext:    ET TU BRUTE
Repeated Key: SP QR SPQRS
Ciphertext:   WH IK TFJJW
```

More on Key Ciphers

- A similar system more in use today takes the **bitwise XOR** of the plaintext and key letters to be the ciphertext letter.

- A key cipher with a short key can be attacked by the same means as a single substitution cipher, guessing the key length m and looking at m different problems, each using only a single letter. The longer the key, the more ciphertext is needed to let this attack work.

- The ultimate key cipher is called a **one-time pad** and has a key *the same length as the message*, chosen randomly and known to both the sender and receiver. This is provably unbreakable but presents logistical problems -- how do you transmit the key?

- A more practical variant is to generate a **pseudorandom** key string.

L&C's Program for a Key Cipher

- L&C's example program in Section 5.2 encrypts and then decrypts a fixed message (declared as the initial value of a string variable) using such a key cipher with a key of length 7. Their key "letters" are small ints, which they add or subtract to the characters (viewed as numbers) to get different characters. They don't bother with the wraparound, since adding or subtracting a small number from a letter in ASCII leaves you in the area of ASCII that has printable characters.

- They load their seven numbers into two queues of Integers, and then use one to encode and one to decode. When they use a key letter, they replace it at the tail of the queue, so they just rotate among the key letters.

- Of course they could have avoided the queues by keeping a count of letters, and every time looking at array element key [count % 7]. But using the queue is simple and clear.

Queues to Process Streams

- Encrypting and decrypting with a key cipher are examples of **stream processing**. The real world offers many situations where a huge amount of data comes at us, and we must react to each small bit of it as we see it.

- This is different from the way we usually think in intro Java, where a program has a fixed input, which we can look at all at once, and a fixed output.

- Some algorithms convert streams to streams, like our cipher that took plaintext and keytext and gave us ciphertext. L&C used a queue to make the fixed key act like a stream.

- We can establish some control of how we deal with the data by putting it in a queue as it arrives, and taking it out of the queue at our own pace. But this is potentially limited by the size of the queue.

L&C's Simulation of Cinema Cashiers

- Their assumptions: A new customer arrives every 15 seconds, a cashier takes 120 seconds to process a customer, customers wait in a real queue for an available cashier.

- With eight or more cashiers, customers never wait -- with fewer than eight the line will back up. How badly does this happen with each number of cashiers? Knowing this, the management could decide how few to hire and still have an acceptable waiting time. (Note that the waiting time will also depend on the number of customers -- L&C pick 100.)

- We could measure the average waiting time (L&C's choice), the maximum waiting time, or the average or maximum number of customers in the queue.

- Let's look a little more closely at their implementation.

The `TicketCounter` Class

- Single program tests varying numbers of cashiers with the same assumptions each time. Departure time is arrival + 120 if cashier is free, otherwise time cashier is done + 120.

```
public class TicketCounter {
   public static void main (String [ ] args) {
      // constants: process time, max # cashiers, # customers
      Customer customer;
      QueueADT<Customer> cq = new LinkedQueue<Customer> ( );
      int [ ] cashierTime = new int[MAX_CASHIERS];
      int totalTime, averageTime, departs;

      for (int cashiers=0; cashiers < MAX_CASHIERS; count++) {
         // set up array, load cq with Customer (count*15)
         while (!cq.isEmpty( )) {
            for (int count=0; count < cashiers; count++) {
               // assign next customer to next cashier, find
               // their resulting departure time, record it
```

Randomness in Simulation

- When we simulate a real world situation, we can't predict the exact behavior of the actors, but often we have an idea of their range of possible behaviors. We can use a **random number generator** to get one of the many possible scenarios and work out what happens in it. The better our model of the **distribution** of behaviors, the more plausible our simulation.

- This means, of course, that the result of our simulation depends in part on the random numbers generated. But we can use the **Monte Carlo method**, running lots of independent simulations, to get an idea of the distribution of possible results according to the assumptions of our simulation.

- This technique is widely used in sports analysis, such as the website coolstandings.com (which said in early September that the Red Sox had a 99% chance of making the playoffs). Nate Silver, now of the *New York Times*, is the most famous practitioner of this technique in political analysis.

Java's Random Number Generators

- Java has a class called `Math.Random`, which provides a variety of pseudorandom generators. How might we use these in the cinema cashier simulation?

- One method there generates real numbers (doubles) that are uniformly distributed between 0 and 1. Rather than have a customer arrive every 15 seconds like clockwork, we could have them arrive after 10 + 10*rand seconds, so they would be uniform between 10 and 20.

- Since the number of seconds is an integer, we could "throw a ten-sided die" and add the result to 10 to get the number of seconds.

- The more likely realistic choice is a Gaussian (Normal) Distribution. We could get a series of arrival times where they average 15, and 95% are between 10 and 20, by using 15 + 2.5*gaussian. `Math.Random` has a method for this.