CS109B Project 1: Decoding the Genome

Due Monday, May 8, 1995

Some Biochemistry Background

A fundamental problem in discovering the human genome sequence involves breaking a piece of chromosome into many overlapping pieces (30,000 is a typical number) and finding which of the pieces have a certain "marker" sequence. If we take a solution made from copies of one or more pieces, we can test, by a chemical process, whether at least one of these pieces contains the marker. Typically, 10 of the 30,000 pieces will have the marker.

In practice, those 10 pieces are found by an algorithm that begins by mixing samples of the 30,000 pieces into 30 groups of 1000 each. One test for the marker in each of the 30 groups will then eliminate all but the (up to) 10 groups that test positive; i.e., we have in 30 tests eliminated 20,000 pieces from the problem. The algorithm then proceeds by testing each of the positive groups of 1000. They are broken into 30 overlapping subsets of 100 each, corresponding to the 30 planes of a $10 \times 10 \times 10$ cube.

The reason 30 keeps cropping up is that this is all done by a robot that can make 30 tests at once. However, the algorithm used by this robot is far from optimal.

Optimizing the Test Sequence

Let us suppose that we have complete freedom to form solutions with any subset of the pieces and test that solution for the presence of the marker. Our goal will be to identify the 10 pieces with the marker, making as few tests as possible. Note that it is not the computation that is expensive, but the biochemical steps that cost money. Thus, we have a funny sort of "running time" problem, where the computation needed to select a set of pieces to test is "free" but the chemical test has a cost of 1.

There is a generally good heuristic that says when we make a test we want the outcome to be "yes" or "no" with equal probability. Thus, we get the most "information" out of the test. There is a whole theory (called "information theory," appropriately enough) that formally justifies this view. Without delving into this theory, we can at least observe intuitively that in the opposite extreme, where one outcome has probability 1 and the other 0, the test is useless and we never get anywhere with such tests. Thus, the first test should be to take a solution of m out of the 30,000 pieces, where the probability that one or more of the 10 pieces with markers are among the m is 1/2. Here, m is approximately 1500; do you see why?

To simplify the testing of your code, let us replace 30,000 by n. We shall give you problems with various values of n, including some much smaller than 30,000.

To start you off, the probability space in which we are working consists of all choices of 10 marked pieces out of n. Thus, the number of possible outcomes is $\binom{n}{10}$. That is a large number, approximately $n^{10}/10!$. However, if you can use your tests optimally, one test will cut the number of remaining possibilities in half, so in $\log_2(n^{10}/10!)$ tests you get down to one possibility; i.e., you have solved the problem. Since $\log_2(10!)$ is about 22, the optimal number of tests is about $10\log_2 n - 22$. That's not too many, even for $n = 30{,}000$, (it's about 130) and much more than the robot uses.

Interfacing With a Simulated Test Situation

We shall provide you (in the class directory, `/usr/class/cs109b`) with modules called `Genome`i for various i. Each is in a file `genome`i and simulates a real-world problem in which a different set of 10 integers are "marked." You access a module by including its file, with `use("genome1")`, e.g, and then opening it (like `Array` as discussed in the ML notes for 4/16), e.g.:

```
open Genome1;
```

Each module contains the following four values and functions you can use:

1. `maxint : int` is the total number of samples, of which 10 are marked. The samples will always be numbered from 1 up to `maxint`. In the "real" problem, `maxint` is 30,000, but we shall give you some smaller values (and larger ones!) as well.

2. `initialize : unit -> unit` initializes the count of tests to 0. You should call `initialize()` once before doing any tests but not call it again.

3. `test : int list -> boolean` takes a list of integers L and returns `true` if one or more of the integers on list L is "marked," or `false` if not.

4. `readout : unit -> int` returns the number of times you have called `test`. You should call `readout()` once, after you have determined the 10 marked integers between 1 and `maxint`. The output of your program should be the list of these 10 integers and the result of `readout()`, with suitable prose around the numbers, e.g., "The following 10 samples are marked: $\cdots$, and we made 130 tests to solve the problem."

What to Hand In

Please put a copy of your function, which you should call `findMarked()`, in the dropoff subdirectory of the class directory, i.e., in `/usr/class/cs109b/dropoff`. We shall test it on some `Genome`i modules different from the ones distributed for your debugging use. Also, please hand in a short writeup with:

1. A listing of your code.

2. The names of all team members.

3. The name of the file in `dropoff` where your code can be found.

4. Perhaps one page describing the principal ideas behind your algorithm. What strategy are you using to decide what to test next?

5. Perhaps one page describing the "running time" of your algorithm. As mentioned above, we are not as interested in the running time of your code, as we are in the worst and/or average case number of tests it makes as a function of n, the number of samples. Of course, if your code runs too slowly, it will not finish on large examples, and you will lose credit. It is reasonable to expect your program to solve problems with $n = 30{,}000$ in a few seconds of running time, and no more than a minute on the wall clock.