ECE220: Computer Systems & Programming **Fall 2020 ZJUI**
Lab 10 **Tuesday 24 November**

Using Randomness to Find Order

In today's lab, you must write some simple array-based code to determine whether two documents in English contain common text. The idea is taken from one of the first successful web search engines, Alta Vista, and was applied to determine to what extent documents on the web matched one another, enabling humans to avoid reading multiple documents with the same content. Although the code given to you uses several concepts you have yet to learn, most of the processing (and all of the code that you need to write) is based on the use of arrays to represent information. MP9 also uses arrays to represent all information, although with somewhat more complex data structures.

Begin by checking out the **`lab10`** subdirectory in your Subversion repository. The directory contains a copy of this document (**`lab10.pdf`**), a C header file **`lab10.h`**, a C source file **`lab10main.c`** that provides most of the code, a **`Makefile`**, and a C source file, **`lab10.c`**, with which you can start this lab. You can immediately "**`make`**" the executable, but without your code, all documents will match one another.

The Key Idea Behind Page Matching

The researchers at DEC SRC (Systems Research Center) invented the idea of comparing content by using pseudo-random numbers. In computing, a *hash* is a function that produces a seemingly random output. For example, given an 24-bit color image, one could XOR all of the RGB pixel data together to produce a hash. The resulting 24-bit number is then somewhat random: given two pictures, we wouldn't expect the function outputs to match, nor could we easily predict the value of the result given the picture. However, given the same picture, the result is always the same.

Now imagine that we have a document and a hash function that produces a 32-bit integer from five words (strings). Take every sequence of five words in the document and calculate the hash. What do we have? A bunch of unrelated pseudo-random numbers! But now take the smallest of those numbers. It's the hash of some five-word sequence in the document.

Now imagine that someone takes the document and modifies it, but leaves most of the content untouched. Repeat the hashing process on the new document. Do we obtain a different minimum hash value? Possibly. If the five-word sequence that generated the hash value was removed, the minimum value is likely to be different. The minimum is also different if the new content happens to generate a sequence that produces a smaller hash value. The probability of the first event depends on how much of the document was changed. The probability of the second event depends on how much new text was added. Both events are related to differences between the two documents.

Finally, instead of one hash function, use 10 (as we will in our lab), or 100, or 1,000. The more (independent) hash functions, the better one can estimate overlap between the two documents!

The Task

You must implement the function **`calculate_hashes`**, which is invoked on every sequence of five words from every file examined by the program. The signature is as follows:

```
void calculate_hashes (int32_t hashes[], char* words[]);
```

The **`hashes`** array contains **`NUM_HASHES`** integers for each file, and the **`words`** array contains **`NUM_WORDS`** NUL-terminated ASCII strings. (For this lab, **`NUM_HASHES`** has the value 10, and **`NUM_WORDS`** is 5, but you can change those definitions in the header file.) Using each hash function (details below), your code should calculate the hash value for the given sequence of words and replace the corresponding value in the **`hashes`** array if and only if the calculated hash value is smaller than the value already in the **`hashes`** array.

The hash functions are defined as follows. For hash function N, start with the value 0 and go over every character in each word in order, starting with the first word and covering the characters from the start of the string until the end of the string (not including the NUL), then moving on with the second word, third word, and so on. For each character, update the hash value as shown below:

$$\text{hash}_{\text{new}} = (\text{hash}_{\text{old}} << (N + 2)) \text{ ^ character}$$

where "<<" and "^" are C's left shift and bitwise XOR operators, respectively. As you might guess, it's not critical that you obtain exactly the same functions as I used in my program, only that the results are reasonably independent and random.

You can compare files by executing the program and listing the files to be compared (up to 10) as arguments. For example,

```
./lab10 lab10.c lab10.h Makefile
```

Try copying files, making minor edits, combining files, and so forth, to see how well the comparison tool works.