CSE 241 Algorithms and Data Structures **Lab 2**

Hashing for Biosequence Comparison

Assigned: 10/8/2012 *Due Date: 10/29/2012*

1 Background

The goal of this lab is to implement hashing as part of a tool for comparing genomic DNA sequences. The approach to biosequence comparison that we'll use here is an important part of such well-known tools as FASTA (Pearson & Lipman 1988) and BLAST (Altschul et al. 1990, 1997; Altschul & Gish 1996).

A DNA sequence is a string of characters, called *bases*, from the alphabet $\{a, c, g, t\}$. Genomic DNA encodes a large collection of *features*, including:

- *genes* – the instructions for building proteins;
- *regulatory sites* – sequence markers recognized by cellular machinery that can increase or decrease the rate at which a given gene is used to make protein;
- *repeats* – junk left behind by *transposable elements*, pieces of DNA that can autonomously copy themselves and move around in the genome. Transposable elements proliferate, then die, leaving behind many inactive copies of themselves in the genome as repeats.

All DNA is subject to *mutations* that alter its sequence over time. However, functional sequence features like genes and regulatory sites are more resistant to mutation than DNA that doesn't code for anything (because natural selection usually kills off organisms with too many mutations in these features). We can therefore find these features by comparing DNA sequences from two different organisms and seeing which parts of the sequences have remained similar to each other since their lineages split from their last common ancestor.

Abstractly, we are given two long strings s_1 and s_2 of characters, and we want to find short substrings (parts of the features) common to s_1 and s_2. In general, the common substrings might not be exactly the same because even functional sequences mutate over time. However, we often find that s_1 and s_2 still exhibit perfectly matching substrings of at least 10 to 15 characters in the neighborhood of their shared features. By detecting these exact substring matches, we can find the most likely locations of the shared features in s_1 and s_2 and can then apply more sensitive but more expensive similarity search algorithms only at those locations.

Naively, we could find k-mers (i.e. substrings of length k) common to s_1 and s_2 by comparing every k-mer from one sequence to every k-mer from the other. Such an approach would take time $\Theta(|s_1| \cdot |s_2|)$, which is unacceptable because interesting DNA sequences range from thousands to billions of characters in length (your own genome is about three billion characters long). Fortunately, there is a much better approach.

Call s_1 the **corpus string** and s_2 the **pattern string**, and assume we are searching for common substrings of length k. We first construct a table T of every k-mer in the pattern string, remembering

where in the pattern it occurs. Then, for each k-mer in the corpus string, we check whether it occurs in the table T; if so, we have found a match between pattern and corpus. If T supports constant-time insertions and lookups (e.g., if it is a hash table), we can process the entire pattern and corpus in time

$$\Theta(|s_1| + |s_2| + M),$$

where M is the number of common substrings actually found. In general, this time cost is much lower than the cost of the naive algorithm. Fast substring matching based on hashing therefore forms the core of many of today's high-speed biosequence matching algorithms.

To make things slightly more interesting, we will also allow the user to specify a **mask string** that contains "uninteresting" DNA. For example, if we're looking for matching genes, we might not be interested in any common substrings that are part of known repeats. Any k-mer appearing in the mask string is removed from the table before searching for matches in the corpus.

2 What To Do

The following sections are marked to indicate roughly what percentage of the total credit for the lab may be gained by completing each section. **Please start early so that you can get help if you need it!**

2.1 Part 1: Implement the Table (70%)

I have provided you with code that implements most of a sequence matching tool. My code reads sequences from files, takes a particular *match length* (e.g., 15 characters) from the command line, then performs the substring matching computation described above. However, there is an important piece missing: a `StringTable` class that implements the hash table used by the matching algorithm. Your goal is to implement this missing class.

Your hash table will hold objects of type `Record` that store the association between a k-mer string (the key) and a list of positions in the pattern where that string occurs. The `StringTable` class exports four public methods with the following semantics:

1. `StringTable(int maxSize)`: this constructor creates a new, empty hash table big enough to hold at least `maxSize` Records.

2. `boolean insert(Record r)`: insert a new Record into the table. If the insertion succeeds, return `true`; if the table is full or the Record's key is already present in the table, return `false`.

3. `void remove(Record r)`: remove a Record from the table. You'll have to find the Record in order to delete it, so unless you keep some extra information in the Record structure itself, you'll have to do a search of the hash table before you can remove it.

4. `Record find(String key)`: find a Record with the given key in the table. If it exists, return it; otherwise, return `null`.

The interface above is for the Java implementation. For the C++ version, things are slightly different: the methods take and return *pointers* to Records, and the constructor also takes the substring match length to store in the table so that I can pass raw character arrays instead of `string` objects as keys. You may assume that you don't need to copy Records to store them – the calling program won't delete or modify a Record unless it first removes it from the table.

In addition to the public interface, the table will need some internal methods that implement its hash functions. I have provided one of these methods: `int toHashKey(String s)` maps a String s to an integer value that you can pass to your hash functions. You **must** implement the actual hash functions yourself using *multiplicative hashing*.

Your implementation must resolve collisions by open addressing with double hashing. You should aim for a maximum load factor $\alpha \approx \frac{1}{4}$. Naturally, you may not use the Java built-in `HashSet` or `HashMap` classes (or the equivalent C++ STL classes) to implement your table! Also, to prevent arithmetic overflow bugs, **you must compute slot sequences in a way that avoids multiplication**, as we discussed briefly in class.

What to Turn In

To complete this part, you must ensure that your SVN repository contains all of the following:

- Your code, in particular your implementation of the `StringTable` class and any modifications you make to the `Record` class.
- A correctly edited *control file* that specifies the implementation language you used and indicates that the lab is ready for grading. The file `lab2/control.txt` is the control file for this lab; see that file and the Labs page on the course website for editing instructions.
- At least three *small* test cases (input files of your own design) designed to illustrate the correctness of your implementation.

  Put your test cases in the `lab2/test-cases` subdirectory of the repository and name them `case1.txt`, `case2.txt`, and `case3.txt`. Please explain the design of these test cases in your `README` file (see below).
- A brief document describing your implementation (you need not recapitulate the theory of hashing with open addressing), your choice of hash functions, how you chose your test cases, and anything else interesting you noticed. If your implementation is buggy and you were unable to fix the bugs, please tell us what you think is wrong and give us a test case that shows the error. You'll lose fewer points for a wrong answer if you indicate that you know it's wrong (and, if possible, why it's wrong).

  Put this document in the `lab2` subdirectory of your repository. It should be named `README`, with a file extension matching its format. The file may be plain text (`README.txt`), Word (`README.doc` or `README.docx`), or PDF (`README.pdf`).

We will test your implementation's correctness after it is turned in by running a suite of test cases of our own design on the code in your repository. You may not have access to all the test cases that we use for validation.

Command Syntax and Reading From Files

The provided Java driver program has the following command syntax:

```
java Lab2 <match length> <corpus file> <pattern file> [ <mask file> ]
```

The corpus, pattern, and mask files contain their respective sequences, while the match length specifies the length of substrings to match between corpus and pattern. Note that the mask file is an optional argument. The C++ driver's command line is similar.

As for the previous lab, if you cannot pass arguments to your program on the command line, just hard-code the arguments in `main()` and recompile.

3 Part 2: Improve the Table (30%)

Implement the following improvements to your basic hash table:

- (10%) Extend each slot of your hash table to store the integer result of applying the `toHashValue` procedure to the key of the Record in that slot. Use this extra value to speed up searches by avoiding a full comparison of two strings when their hash values differ.
- (20%) Implement the doubling procedure described in Homework 2 to allow your hash table to grow dynamically as records are added. The doubling code should be a separate method of your hash table that is called internally when you detect that the table is full. To get credit for this modification, **you must ignore the `maxSize` argument to the constructor**. Your hash table should instead be initialized to hold only two records. You must implement the doubling yourself; do not use the `ArrayList` class. Note that doubling requires you to rehash and reinsert every Record in the table!

If your improved implementation works, you can use it *instead of* the original one to satisfy all the turn-in requirements of Part 1. Document how you did the extensions in your `README`.