

Lab 2

February 4, 2003

Due Date: February 11

Get started right away, especially if you did not submit Lab 1 on time. Remember that this lab is due on one week.

1 Background (courtesy of Dr. Jeremy Buhler)

The goal of this lab is to implement hashing as part of a tool for finding matching substrings in two text strings or files. In particular, we'll look at the application 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).

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 exactly 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 Your Assignment

Part 1: Implement a Hash Table using Open Addressing (40 points)

We have provided you with code that implements most of a sequence matching tool. The provided code reads sequences from files, asks the user for the desired *match length* (e.g. 15 characters), then performs the substring matching computation described above. However, there's an important piece missing: a **StringDictionary** class that implements the hash table used by the matching algorithm. Your goal is to implement this missing class.

Your hash table will associate each key with an 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 associated data). The **StringDictionary** class exports seven public methods with the following semantics. These semantics were selected to match that of the Java Dictionary class.

StringDictionary(int maxSize): this constructor creates a new, empty hash table big enough to hold at least **maxSize** Records. You can assume for the purposes of this lab that there will be no more than **maxSize** items held in the StringDictionary.

Record put(String key, Record r): Associates the specified Record **r** with the specified key in the dictionary. If the dictionary already contains a Record for this key, the old Record is replaced by **r**. This function returns the previous value associated with specified key, or null if there was no mapping for key. A null return can also indicate that the map previously associated null with the specified key.

boolean containsKey(String key): Returns true if the given key is contained in the dictionary. If the key is not in use, false is returned.

Record get(String key): Returns the Record associated with the specified key. Returns null if the dictionary does not contain any item with the given key. A return value of null does not necessarily indicate that the dictionary does not contain the key; it's also possible that the map explicitly maps the key to null. The **containsKey** method may be used to distinguish between these two cases.

Record remove(String key): Removes key from the dictionary, if it is present. Returns the Record associated with the key if it was in use, or null if the key was not in the dictionary. (A null return can also indicate that the Record previously associated with the specified key was null.)

int size(): Returns the number of key-Record mappings in the dictionary.

You may assume that you don't need to copy Records to store them – the calling program is responsible for properly maintaining the records, the dictionary's job is just to store them and return them as appropriate.

In addition to the public interface, the table will need some internal methods that implement its hash functions. I have provided three of these methods: `int toHashKey(String s)` maps a String `s` to an integer value that you can pass to the hash functions. Your implementation *must* resolve collisions by open addressing with double hashing. You should aim for a maximum load factor $\alpha \approx \frac{1}{3}$. You will also want to add a private class that encapsulates the key and Record into one object.

Naturally, you may not use the Java `HashSet` or `HashMap` classes to implement your table! I've also provided the primary hash function, `int baseHash(int hashKey)`, and the secondary hash function, `int stepHash(int hashKey)`. In order for the secondary hash function to reach all slots (which is a property you want), the hash table size is required to be a power of 2. Do not change any of these provided functions.

The interface above is for the Java implementation. For the C++ version, things are just slightly different: the methods take and return *pointers* to Records, and the constructor also takes the substring match length to store in an instance variable, `matchLength` so that you can pass raw character arrays instead of String objects as keys, and `NULL` is used instead of `null`. Also, a C++ `equals` method is provided that returns true if and only if the first `matchLength` characters of two provided character arrays are equal.

NOTE: The provided driver to perform the matching algorithm does not test all aspects of the required methods. As just one example, `put` is never called with a key already in use. It is your job to be sure that your code provides the functionality described above. The TAs will be checking this in your code and deductions will be made if there are errors even if you obtained the correct output for the 4 test cases. Those test cases should find any logical errors in how you implement open addressing but you need to be sure to carefully read the interface required and adhere to it.

Part 2: Empirically Study Number of Probes (10 points)

Each time you examine an entry in the hash table within the `contains` method, that is called a *probe*. This includes the table entry that ends the search (i.e. the one with the key if is in the hash table or an empty slot if it is not there). In order to see how the performance of your hash table varies with the load factor, here you modify your implementation to compute the average, min, and max probes used by the `contains` method calls made from within `findMatches` of the driver.

One way to compute these statistics would be to add an instance variable, and public accessor, that counts the number of probes made by the last call of `contains`. Then you can compute the average, min, and max values within `findMatches`. Another option, it to pass a flag to `contains` indicating which calls to include when gathering statistics and then have the `StringDictionary` compute the needed statistics and provide a public accessor to get them. Any way you want to do this is fine.

You will need to compute the average, maximum and minimum number of probes for 10 different load factors roughly between $\frac{1}{10}$ and $\frac{1}{2}$ on Data Set 3 with a match length of 18. See the below section on “What to Submit” for details on what you will need to submit.

Command Syntax and Reading From Files

The provided Java driver program has the following command syntax:

```
java Lab2 <transcript file> <corpus file> <pattern file> [ < mask file> ]
```

The corpus, pattern, and mask files contain their respective sequences, while the transcript file stores a record of the program's output. Note that the mask file is an optional argument. Also,

since I know some students do not know how to pass command line arguments, if no command line arguments are provided then the main program will ask the user to input which data set (1,2,3 or 4) should be used and then it will use the appropriate files (with the provided names) and also save a transcript file. More details is included in the comments in `Lab2.java`.

The C++ version of the driver omits the transcript file argument, so you'll have to capture its output some other way such as using the `transcript` program provided within Unix.

A Note on Compatibility

The provided Java code has been tested with JDK 1.2 and higher, but it may not compile on JDK 1.1.x. You should have no problem using the code on the CEC Windows machines or the "hotel" Solaris servers (`hilton`, `ritz`, etc.), but it will probably break if you use the ancient JDK installed on CEC's Sparc 5 workstations. You may be able to make the code work on older JDK's by replacing all uses of the `ArrayList` data type with the older `Vector` type.

What to Submit

Please staple the following items and write your name on each page. You should have a cover sheet (with all portions filled in including your name and signature) stapled to the front of what you submit. If there are too many sheets to staple them all together, please use a binder clip or other mechanism to keep the parts from getting separated.

- Turn in your `StringDictionary` class. If you modify the `Record` classes in any way, turn those in as well along with the code for any new classes created. (Include any new/revised header files for those using C++).
- On the course web page (under labs) are four sets of DNA sequences on which to run your program; two of these sets include the correct outputs within the provided README file. **For the first test set, use a table size of 128.** For the others your program should select a table size to give a load factor of about $\frac{1}{3}$ for the first part, and vary this for the second part. Run your program on all four test sets. Use a match length of 3 for Data Set 1, a match length of 15 for Data Set 2, a match length of 18 for Data Set 3, and a match length of 65 for Data Set 4. You'll find more information about the Data sets in the provided README file.
- 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).
- If you did Part II, report your results by creating a graph where the x -axis is the load factor (measured based on the number of key-Record mappings actually in the dictionary when `findMatches` is called) and the y -axis is the average number of probes. Report your results from running the program with about 10 different load factors roughly between $\frac{1}{10}$ and $\frac{1}{2}$ on Data Set 3 with a match length of 18. Also, give a table (or add two additional lines to your graph) for the minimum and maximum number of probes for each data point. Finally write a brief discussion. Some examples of questions to address are: How does the average number of collisions relate with the load factor? Is this what you expected? If not, try to explain why.