Computer Systems Organization, Spring 2011
RK Lab: Approximiate document matching
Due: Sep 23 11:59PM

1 Introduction

In many scenarios, we would like to know how "similar" two documents are are to each other. For example, many pages on the web are similar but not entirely identical to each other. Search engines like Google or Bing need to group similar web pages together such that only one among a group of similar documents is displayed in the search result. We refer to this process of measuring document similarity as *approximate matching*.

In this lab, you will write a program to approximately match one input file against another file. The goal is to get your hands dirty in programming using C, e.g. manipulating arrays, pointers, number and character representation, bit operations etc.

This is an individual project. All handins are electronic. Clarifications and corrections will be posted on the course Web page or discussion group.

2 Handout Instructions

Start by pointing your brower to
`http://news.cs.nyu.edu/~jinyang/fa13/restricted/handouts/rklab-handout.tar`
and copying `rklab-handout.tar` to your home directory on the virtual machine. Then type the following command:

```
unix> tar xvf rklab-handout.tar
```

This will cause a number of files to be unpacked. The only file most of you will be modifying is `rkmatch.c`. If you are an honors student and/or choose to do the optional Bloom filter section, you will also work with `bloom.c`.

3 Exact Matching

We start by solving the simpler problem of checking whether a document is an exact duplicate of another. However, we will do something slightly more sophisticated by matching documents' raw contents character-by-character. In particular, we "normalize" (i.e. clean up) a document by doing the following:

1. Convert all upper case letters to lower case ones.
2. Convert different white space characters (e.g. carriage return, tab,...) into the space character,
3. Shrink any sequence of two or more space characters to exactly one space.
4. Remove any whitespace at the beginning and end of the text.

As an example, if the original content of X is "␣I␣am␣␣A$\backslash n$Dog" where ␣ is the space character and $\backslash n$ is the new line character, the normalized content of X should be "i am a dog".

We consider document X and Y to be an exact match if their corresponding normalized contents are identical.

Your job: Implement the perfect matching algorithm. The `rkmatch.c` file already contains a code skeleton with a number of helper procedures. Read `rkmatch.c` and make sure you understand the basic structure of the program. To implement the exact matching algorithm, you need to complete the procedures `exact_match` and `normalize`.

In `rkmatch.c`, the `main` procedure first invokes `normalize` to normalize the content of files X and Y. It then invokes `exact_match`. The invocation passes in—as part of the procedure arguments—the pointer to X's content (`const char *qdoc`), X's length, the pointer to Y's content (`const char *doc`), and Y's length. If X and Y are an exact match, the procedure returns 1, otherwise, it returns 0.

Testing: Run the given tester program `$./rktest.py -t exact`

4 Approximate Matching

Determining similarity is a much trickier business than performing exact matching. Let us first start with an inefficient but working algorithm that measures how well document X (of size m) approximately-matches another document Y (of size n). The algorithm considers every substring of length k in X and checks if that substring appears in document Y. For example, if the content of X is "abcd" and $k = 2$, then the algorithm tries to match 3 substrings ("ab", "bc", "cd") in Y. The algorithm counts the number of matched substrings (ctr). Since there are total $m - k + 1$ substrings to match, the fraction of matched substrings is $\frac{ctr}{m-k+1}$. We can use the calculated fraction as the approximate match score. Intuitively, the more similar file X is to Y, the higher its final score. In particular, if file X is identical to Y, the final score would be 1.0.

Our approximate algorithm relies on a subroutine to determine whether a shorter string (called the query string) appears as a substring in some other longer string (called the destination string). In our case, the query string is a k- character-long substring from X and the destination string is the normalized content of Y. The naive way to check for substring match to compare the query string with *every* substring of length

(k) in Y starting at position $0, 1, 2, ..., (n-k)$. Thus, each query string takes $O(k*n)$ time. Since there are a total of $m-k+1$ substrings of X to be matched in Y, the total runtime of our approximate matching algorithm would be $O(k*m*n)$. This runtime is pretty bad and we will improve it greatly in subsequent steps.

4.1 Simple Approximate Matching

As the first optimization, we observe that it is not necessary to do substring matching for all $m-k+1$ substrings of X. Rather, we can "chop" X into $\lfloor \frac{m}{k} \rfloor$ non-overlapping substrings (called chunks) and try to match each chunk in Y. For example, if the content of X is "abcd" and $k=2$, the optimized algorithm only matches 2 chunks ("ab", "cd") instead of 3 substrings as in the original naive algorithm. Doing so cuts down the runtime by a factor of k to $O(m*n)$ [1]. We refer to this version as the simple algorithm.

Your job: Implement the simple approximate matching algorithm. You only need to add your code to `rkmatch.c`. Specifically, you need to complete the procedures `simple_substr_match`.

In `rkmatch.c`, the `main` procedure considers each chunk of X in turn and invokes `simple_substr_match` to find a match in file Y. The invocation passes in—as part of the procedure arguments—the pointer to the chunk, chunk's length (k), the pointer to Y's content and Y's length. If a match is found, the procedure returns 1, otherwise, it returns 0. (If a chunk of X appears many times in Y, the return value should still be 1.)

Testing: Run the given tester program `$./rktest.py -t simple`

4.2 Rabin-Karp Approximate Matching

Our next optimization comes from using Rabin-Karp substring matching algorithm (RK for short), invented in the eighties by two famous computer scientists, Michael Rabin and Richard Karp [2].

RK checks if a given query string P appears as a substring in Y. RK uses the idea of hashing: A hash function turns a string of arbitary length into a b-bit hash value with the property that collision (two different strings having the same hash value) is unlikely. At a high level, RK works by computing a hash for the query string, $hash(P)$, as well as a hash for each of the $n-k+1$ substrings of length k in Y, $hash(Y[0...k-1]), hash(Y[1...k]), ..., hash(Y[n-k...n-1])$, where $Y[0...k-1]$ is the first substring of Y and so on. By comparing $hash(P)$ with each of the $n-k+1$ hashes from Y, we can determine if P appears as a substring in Y. There are many nice hash functions out there (such as MD5, SHA-1), but unfortunately, none of them would make RK any faster since it takes $O(k)$ time to compute each of the $n-k+1$ hashes from Y.

RK's magical ingredient is its invention of a "rolling" hash function. Specifically, given $hash(Y[i...i+k-1])$, RK takes only constant time instead of $O(k)$ time to compute $hash(Y[i+1...i+k])$.

We first describe how RK hashes a string. Let's treat each character as a digit in radix-d notation. We choose radix $d=256$ since each character in the C language is represented by a single byte and we can conveniently use the byte value of the character as its digit. For example, the string 'ab' corresponds to two

[1] The simple algorithm does not produce the exact same score as the naive algorithm in all scenarios, but the resulting score is close enough for practical purposes.

[2] If you do not know who Rabin and Karp are, it is time to look them up in Wikipedia.

digits with one being 97 (the ASCII value of 'a'), and the other being 98 (the ASCII value of 'b'). The decimal value of 'ab' in radix-256 can be calculated as $256 * 97 + 98 = 24930$. The hash of a string P in RK is $hash(P[0...k-1]) = 256^{k-1} * P[0] + 256^{k-2} * P[1] + ... + 256 * P[k-2] + P[k-1]$.

We now see how to do a rolling calculation of the hash values for consecutive substrings of Y. Let $y_i = hash(Y[i...i+k-1])$. We can compute y_{i+1} from y_i in constant time, by observing that

$$\begin{aligned} y_{i+1} &= 256^{k-1} * Y[i+1] + 256^{k-2} * Y[i+2] + ... + Y[i+k] \\ &= 256 * (256^{k-2} * Y[i+1] + ...Y[i+k-1]) + Y[i+k] \\ &= 256 * ((256^{k-1} * Y[i] + 256^{k-2} * Y[i+1] + ...Y[i+k-1]) - 256^{k-1} * Y[i]) + Y[i+k] \\ &= 256 * (y_i - 256^{k-1} * Y[i]) + Y[i+k] \end{aligned}$$

In order to achieve constant time calculation, we have to remember the value of 256^{k-1} in a variable instead of re-computing it each time.

Now we've seen how rolling hash works. The only fly in the ointment is that these radix-256 hash values are too huge to work with efficiently and conveniently. Therefore, we perform all the computation in modulo q, where q is chosen to be a large[3] prime[4]. Hash collisions are infrequent, but still possible. Therefore once we detect some $y_i = hash(P)$, we should compare the actual strings $Y[i...i+k-1]$ and $P[0...k-1]$ to see if they are indeed identical.

Since RK speeds up substring matching to $O(n)$ (assuming hash collusion is unlikely) instead of $O(n * k)$ as in the simple algorithm. However, we still need to run RK $\lfloor \frac{m}{k} \rfloor$ times for each of the $\lfloor \frac{m}{k} \rfloor$ chunks of X to be matched in Y. Thus, our approximate matching algorithm using RK has an overall runtime of $O(\frac{m}{k} * n)$.

Your job: Implement the RK substring matching algorithm by completing the `rabin_karp_match` procedure. When calculating the hash values, you should use the given modulo arithmatic functions, `madd`, `mdel`, `mmul`.

As with `simple_substr_match`, the `main` procedure will invoke `rabin_karp_match` for each chunk of X to be matched. `rabin_karp_match` has the same interface as `simple_match` and should return 1 if the chunk appears as a substring in Y or 0 if otherwise.

Testing: Run the given tester program `$ ./rktest.py -t rk`

4.3 RK Approximate Matching with a Bloom Filter (optional for non-honors)

Our RK-based approximate matching algorithm has a runtime of $O(\frac{m}{k} * n)$. Now we will boost its speed further by using a Bloom filter.

A Bloom filter is a bitmap of h bits initialized to zeros in the beginning. We insert all $\lfloor \frac{m}{k} \rfloor$ RK hash values of X that we want to match into the "filter".

To insert a value v into the bitmap, we use f hash functions to map v into f positions in the bitmap and set each position to be one. For example, starting with a 10-bit bitmap and $f = 2$, if v is mapped to positions

[3] Why choosing a large modulus?

[4] Why choosing a prime? Those who take Algebra in college will know better, but for the rest of us mortals, it suffices to say using primes makes hash collisions less likely than using non-primes

$1, 5$ and v' is mapped to $3, 9$, the bitmap after inserting v, v' would be 0101010001. After we have inserted all $\lfloor \frac{m}{k} \rfloor$ hash values of X, we proceed to calculate every y_i in Y and check whether it *may match* any of the elements in the filter. This is done by mapping each y_i into f positions using the same f hash functions and checking whether *all* f positions contain value 1. If so, y_i is a *probable* match. We say the y_i's match is probable because Bloom filter incurs false positives in that y_i may be considered to equal to some of the $\lfloor \frac{m}{k} \rfloor$ hash values even though it is not.[5] Thus, to confirm that y_i is a real match, we check whether $Y[i...i+k-1]$ is indeed identical to any of the $X[0...k-1]$,$X[k...2k-1]$... strings.

Using a Bloom filter, our enhanced algorithm has a total runtime of $O(m+n)$ (assuming there are not too many false positives), significantly faster than our previous two versions of approximate matching!

Your job: First implement the Bloom filter functions by implementing the`bloom_init`, `bloom_query`, and `bloom_add` in the source file `bloom.c`.

To help you implement `bloom_add` and `bloom_query`, we provide you with a particular hash function implementation for Bloom filter, `int hash_i(int i, long long x)`. This function will hash a given Rabin-Karp hash value `x` into an `int` value according to the `i`-th hash function. The number of hash functions that you should use is given by `BLOOM_HASH_NUM`, a global constant defined in file `bloom.c`.

After you are done with the bloom filter implementation, test its correctness with our test script, using the command `./rktest.py -t bloom` The test script invokes `bloom_test` (see its implementation in `bloom_test.c`) to test your bloom filter implementation and it compares your output to the correct answer.

Next, implement the RK algorithm with a Bloom filter by completing the `rabin_karp_batchmatch` procedure in `rkmatch.c`. In the template code, we tell you the size of the bitmap to use (in bits) as a procedure argument (`bsz`). In the `rabin_karp_batchmatch` procedure, you should invoke the auxilary function `bloom_init` to allocate and initialize a byte array (i.e. character array) that will pack `bsz` bits for use as the bloom filter's bitmap. You should then compute all $\lfloor \frac{m}{k} \rfloor$ RK hash values corresponding to X's chunks and insert them into the Bloom filter using `bloom_add`. Subsequently, you scan Y's content and compute a RK value for each of its $n-k+1$ substrings and query its presence in the Bloom filter using `bloom_query`. If `bloom_query` indicates a potential match, you proceed to check that the corresponding substring of Y indeed matches one of the $\lfloor \frac{m}{k} \rfloor$ chunks of X.

Keep in mind that unlike `simple_substr_match` or `rabin_karp_match`, returning 1 or 0 is no longer sufficient; `rabin_karp_batchmatch` is invoked only once and therefore must return the total number of chunks of X that have appeared in Y.

Testing: Run the given tester program `$./rktest.py -t bloom` to test your Bloom filter implementation in `bloom.c`. You can then test the batch Rabin-Karp method with `$./rktest.py -t rkbatch`. To run the full suite of tests including the Bloom filter tests, run the test program without arguments, `$./rktest.py`

5 Evaluation and Hand-in

Your score will be computed out of a maximum of 35 points based on the following distribution:

[5] It is an important property that Bloom filter never incurs false negatives.

30 Correctness points, each version of the algorithm carries 10 points. For non-honors students, the 3rd version of the algorithm carries 10 bonus points.

5 Style points. We reserve 5 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive.

Hand-in instructions: Run the following command to generate `handin.tar`:
`make handin`
Submit your `handin.tar` file at the following URL:
`http://news.cs.nyu.edu/cgi-bin/fa13/submit-rklab.pl`