# Computer Systems Organization
# RK Lab: Approximiate document matching
# Due: Sept 23 11:59PM

## 1 Introduction

In many scenarios, we would like to know how "similar" two documents are to each other. For example, 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 as part of the search result. As another example, professors can detect plagiarism by checking how similar students' handins are. We refer to the 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 discussion group.

## 2 Handout Instructions

Start by pointing your brower to:
http://cs.nyu.edu/~jchen/fa14/labs/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. You will also work with bloom.c.

## 3 Approximate Matching

Checking whether a document is an exact duplicate of another is straightforward [1]. By contrast, it is trickier to determine similarity. 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

## 3 Approximate Matching

Checking whether a document is an exact duplicate of another is straightforward [1]. By contrast, it is trickier to determine similarity. Let us f rst start with an ineff cient 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. For each substring matched, the algorithm increments a counter $ctr$. Since there are total $m - k + 1$ substrings to match, the algorithm calculates the fraction of matched substrings, i.e. $\frac{ctr}{m-k+1}$, as the approximate match score. The more similar f le X is to Y, the higher its f nal score. In particular, if f le X is identical to Y, the f nal score would be $1.0$.

The naive algorithm works, but is slow. In particular, the naive way of checking whether a string appears as a substring in Y is to check if that string matches a substring of length $k$ at every position $0, 1, 2, ..., (n - k)$ in Y. Thus, each substring matching takes $O(k * n)$ times, and since there are a total of $m - k + 1$ substrings of X to be matched in Y, the total runtime would be $O(k * m * n)$. This runtime is pretty bad and we will improve it greatly in this lab step by step.

### 3.1 Simple Approximate Matching

As the f rst optimization, we observe that it is not necessary to do substring matching for all $m - k + 1$ substrings of X. Rather, we simply “chop” X (conceptually) into $\lfloor \frac{m}{k} \rfloor$ chunks and tries 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 $O(m * n)$ [2]. We refer to this version as the simple algorithm.

In addition to speed improvement, we also make the simple algorithm more robust to spurious differences between documents. In particular, we “normalize” 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”.

**Your job:** Implement the simple approximate matching algorithm. The `rkmatch.c` f le 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 simple algorithm, you need to complete the procedures `simple_match` and `normalize`.

[1] Make sure you know how to do this.

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

The `main` procedure f rst invokes `normalize` to normalize the content of f les X and Y. It then considers each chunk of X in turn and invokes `simple_match` to f nd a match in f le Y. The invocation passes in—as part of the procedure arguments—the pointer to the chunk (`const char *ps`) as well as the pointer to the content of Y (`const char *ts`). 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 0`

## 3.2 Rabin-Karp Approximate Matching

Our next optimization comes from using the brilliant Rabin-Karp substring matching algorithm (RK for short), invented in the eighties by two famous computer scientists, Michael Rabin and Richard Karp [3].

RK checks if a given query string P appears as a substring in Y. 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, $hash(Y[0...k-1]), hash(Y[1...k]), ..., hash(Y[n-k...n-1])$. A hash function turns any arbitary string into a b-bit hash value with the property that collision (two different strings with the same hash value) is unlikely. Therefore, by comparing $hash(P)$ with each of the $n-k+1$ hashes from Y, we can check if P appears as a substring in Y. There are many nice hash functions out there (such as MD5, SHA-1), but RK's magical ingredient is its "rolling" hash function. Specif cally, given $hash(Y[i...i+k-1])$, it takes only constant time instead of $O(k)$ time to compute $hash(Y[i+1...i+k])$.

Now we explain how the rolling hash is computed. 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 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 $hash(P[0...k-1]) = 256^{k-1} * P[0] + 256^{k-2} * P[1] + ... + 256 * P[k-2] + P[k-1]$. Now let's see how to do a rolling calculation of the values for every substring of Y. Let $y_0 = hash(Y[0...k-1]$ and $y_i = hash(Y[i...i+k-1])$. We can compute $y_{i+1}$ from $y_i$ in constant time, by observing that

$$y_{i+1} = 256 * (y_i - 256^{k-1} * Y[i]) + Y[i+k]$$

Note that we have to remember the value of $256^{k-1}$ in a variable instead of re-computing it for $y_i$ each time.

Now we've seen how rolling hash works. The only f y in the ointment is that these radix-256 hash values are too huge to work with eff ciently and conveniently. Therefore, we perform all the computation in modulo $q$, where $q$ is chosen to be a large[4] prime[5]. 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 $O(n)$ on average 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 $O(\frac{m}{k} * n)$.

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

[4]Why choosing a large modulus?

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

**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_match`, our `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 1`

## 3.3 RK Approximate Matching with a Bloom Filter

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 f lter.

Our enhanced algorithm uses a Bloom f lter which is simply 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 in Y into the "f lter". 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 $1, 5$ 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 f lter. 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 f lter 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. Thus, to conf rm 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 f lter, our enhanced algorithm has a total runtime of $O(m + n)$, signif cantly faster than our previous two versions!

**Your job:** First implement the Bloom f lter functions by implementing the`bloom_init`, `bloom_query`, and `bloom_add` in the source f le `bloom.c`.

To help you implement `bloom_add` and `bloom_query`, we provide you with a particular hash function implementation for Bloom f lter, `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 def ned in f le `bloom.c`.

After you are done with the bloom f lter implementation, test its correctness with our test script, using the command `./rktest.py 2` The test script invokes `bloom_test` (see its implementation in `bloom_test.c`) to test your bloom f lter implementation and it compares your output to the correct answer.

Next, implement the RK algorithm with a Bloom f lter 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 f lter'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 f lter 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 f lter 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_match` or `rabin_karp_match`, returning 1 or 0 is no longer suff cient; `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 `$./rktest.py 2` to test your Bloom f lter implementation in `bloom.c`. You can then test the batch Rabin-Karp method with `$./rktest.py 3`. To run the full suite of tests including the Bloom f lter tests, run the test program without arguments, `$./rktest.py`

## 4 Evaluation and Hand-in

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

**30** Correctness points, each version of the algorithm carries 10 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
```
Email your file to:
`jchen@cs.nyu.edu`