Release date: 2022/10/14 Deadline: 2022/10/28 23:00

# 1 Background

Consider the basic pattern matching problem discussed in class: given a string $p$ of length $m$ (a.k.a. the pattern) and a string $x$ of length $n$ (a.k.a the document), the task is to find all occurrences of $p$ in $x$. Assume $m \leq n$. We saw in class that a naïve algorithm for this takes $O(mn)$ time, whereas the smarter Knuth-Morris-Pratt algorithm does the job in $O(m+n)$ time. Throughout this document, if $y$ is a string, then we denote by $y[i..j]$ the substring of $y$ starting at index $i$ and ending at index $j$. Recall that the Knuth-Morris-Pratt algorithm computes the failure function $h : \{1, \ldots, m\} \longrightarrow \{1, \ldots, m\}$ associated with the pattern $p$, where $h(i)$ is the length of the longest proper prefix of $p[1..i]$ that is also a suffix of $p[1..i]$. This function is then used to process the document. Thus, the function $h$ remains in the working memory, occupying $\Theta(m \log m)$ bits of space. Another $\Theta(\log n)$ bits of space is needed to store the current index while scanning the document, bringing the overall space complexity to $\Theta(m \log m + \log n)$. The $\log n$ term is unavoidable because we have to store at least a constant number of indices, but can we cut down the $m \log m$ term?

Our task for this assignment is to design and implement an algorithm that has about the same time complexity as Knuth-Morris-Pratt, but uses only $O(\log m + \log n)$ working memory. Of course, this comes at a cost: the algorithm does report false positives with a tiny probability $\varepsilon$.

# 2 The Basic Idea

For the purpose of this assignment, assume that the document $x$ is a string over the uppercase Latin alphabet: $\{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$. Identify these characters with numbers as follows: A with 0, B with 1, ..., Z with 25. A string $y = y[0]y[1]\cdots y[n-1]$ over the set $\{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$ of length $n$ is the 26-ary representation of the number $f(y) = \sum_{i=0}^{n-1} 26^{n-i-1} \times y[i]$ (i.e. $y[0]$ is the most significant and $y[n-1]$ is the least significant), and the function $f$ is a bijection between strings and non-negative integers. The task of finding occurrences of $p$ in a document $x$ is the same as finding all indices $i$ such that $f(x[i..(i+m-1)]) = f(p)$. This observation does give a correct algorithm, but how better is it than the naïve and the Knuth-Morris-Pratt algorithms? Apart from the current index, we need to store the number $f(p)$ in our working memory, and this takes about $\log n + \log_2 26^m$ space, which is $\Theta(m + \log n)$ – not significantly better than Knuth-Morris-Pratt. For each $i$, computing $f(x[i..(i+m-1)])$ takes time $\Omega(m)$. Thus, the running time is $\Omega(mn)$ – no better than the naïve algorithm. We definitely need more ideas!

To get around the problem of $\Omega(m + \log n)$ space, we choose an appropriate prime number $q$, and store $f(p) \bmod q$ in our working memory instead of $f(p)$. Only $O(\log q)$ bits of working memory are sufficient for this. Then for each $i$, we compute $f(x[i..(i+m-1)]) \bmod q$, and report a match if and only if it equals $f(p) \bmod q$. This does introduce false positives, and we will see how to control the error probability by choosing $q$ carefully. Let us worry about the computation first. Note that computing $f(x[i..(i+m-1)]) \bmod q$ still takes $\Omega(m)$ time, resulting in an overall $\Omega(mn)$ running time. How do you get around this? Your first challenge is to design an algorithm whose output is the same as the algorithm mentioned above, but which runs in time $O(n \log_2 q)$, assuming that basic arithmetic operations on $b$-bit numbers take $\Theta(b)$ time. Your algorithm may only use $O(\log n + \log q)$ bits of working memory. Call this algorithm `modPatternMatch(q,p,x)`.

# 3 Controlling Error

How should we choose the prime number $q$ for `modPatternMatch(q,p,x)` so that we don't report too many false-positives (i.e. indices $i$ such that $f(x[i..(i+m-1)]) \neq f(p)$ but $(f(x[i..(i+m-1)]) \bmod q) = (f(p) \bmod q)$)? Clearly, choosing $q$ deterministically is not a good idea. A worst-case instance could have lots of occurrences of a pattern $p' \neq p$ such that $(f(p') \bmod q) = (f(p) \bmod q)$. To get around this, we choose $q$ to be a uniformly random prime which is at most an appropriately chosen number $N$. The number $N$ will depend on $m$, the length of the pattern, and $\varepsilon$, the upper bound on the error probability. Thus, we get the following algorithm.

`randPatternMatch(ε,p,x)`:

1. Compute `N` appropriately. (Your job is to figure out the details.)
2. `q` $\leftarrow$ `randPrime(N)`. (You are given an implementation of the function `randPrime(N)` which returns a uniformly random prime less than or equal to $N$. For analysis, we will ignore the running time of this function.)
3. Return `modPatternMatch(q,p,x)`.

As stated in the algorithm, your job is to figure out what $N$ you should use. You will find the following facts useful.

**Claim 1.** *The number of prime factors of a positive integer $d$ is at most $\log_2 d$.*

**Claim 2.** *Let $\pi(N)$ denote the number of primes that are less than or equal to $N$. Then for all $N > 1$,*

$$\pi(N) \geq \frac{N}{2\log_2 N}.$$

As an exercise, you may try proving the first claim. (Don't submit the proof.) The proof of the second claim is outside the scope of the course.

For full credit, `randPatternMatch(`$\varepsilon$`,p,x)` must run in time $O\big((m+n)\log\frac{m}{\varepsilon}\big)$ and use $O(k+\log n+\log(m/\varepsilon))$ working space (assuming $k$ is the space required to store $L$), ignoring the time taken and space used by the call to the function `randPrime` provided to you. It must return a sorted list $L$ of indices such that for each index $i$:

1. If $x[i..(i+m-1)] = p$, then $i \in L$ with probability one.
2. If $x[i..(i+m-1)] \neq p$, then $\Pr[i \in L] \leq \varepsilon$.

In other words, for each $i$, `randPatternMatch` must have *one-sided error* with probability at most $\varepsilon$.

# 4 Wildcards

Next, let us be a bit more ambitious. Suppose now that pattern $p$ is a string over the set $\{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\} \cup \{\text{'?'}\}$, where '?' is a wildcard character, and $p$ contains exactly one occurrence of the wildcard. The document $x$ is still a string over $\{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$. We say that the pattern $p$ matches the document $x$ at index $i$ if for all $j \in \{0, \ldots, m-1\}$: $p[j] = x[i+j]$ or $p[j] = \text{'?'}$. Given $p$ and $x$, the goal is to report all indices $i$ such that $p$ matches $x$ at index $i$. Consider the following algorithm.
`randPatternMatchWildcard(`$\varepsilon$`,p,x)`:

1. Compute `N` exactly like in the first step of `randPatternMatch`.
2. `q` $\leftarrow$ `randPrime(N)`.
3. Return `modPatternMatchWildcard(q,p,x)`.

This algorithm is analogous to `randPatternMatch`, except that instead of `modPatternMatch(q,p,x)`, it uses an algorithm `modPatternMatchWildcard(q,p,x)`. Your job is to modify the ideas behind the design of `modPatternMatch(q,p,x)` and come up with an implementation of `modPatternMatchWildcard(q,p,x)` so that the algorithm `randPatternMatchWildcard(`$\varepsilon$`,p,x)` runs in time $O\big((m+n)\log\frac{m}{\varepsilon}\big)$, uses $O(k+\log n+\log(m/\varepsilon))$ working space (again, assuming $k$ is the space for storing output and ignoring the time taken and space used by `randPrime`), and returns a sorted list $L$ of indices such that for each index $i$:

1. If $p$ matches $x$ at index $i$, then $i \in L$ with probability one.
2. If $p$ doesn't match $x$ at index $i$, then $\Pr[i \in L] \leq \varepsilon$.

## 5 A word of caution

Strings are immutable. If you somehow attempt to use the space in the given input string for your computation (eg. by creating a list of characters from it), you are using as much working memory as the length of the string. This is obviously unacceptable given our space constraints.

## 6 Submission Specifications

You are given a file named `a4.py`. The file contains the following functions.

1. `randPrime(N)`: this function is completely implemented and it returns a random prime number less than or equal to $N$. The time taken for this call and the space used is to be ignored in the analysis.

2. `findN(eps,m)`: this function is called by `randPatternMatch` and `randPatternMatchWildcard`. Given the pattern length `m` and the error bound `err`, this function should return an appropriate `N` so that the callers satisfy their respective error bounds. You are required to implement this.

3. `randPatternMatch(eps,p,x)`: this function is completely implemented, but it uses functions `findN` and `modPatternMatch(q,p,x)`, which you are required to implement. The requirements on the function `randPatternMatch` are given in Section 3.

4. `randPatternMatchWildcard(eps,p,x)`: this function is completely implemented, but it uses functions `findN` and `modPatternMatchWildcard(q,p,x)`, which you are required to implement. The requirements on the function `randPatternMatchWildcard` are given in Section 4.

5. `modPatternMatch(q,p,x)`: this function is required to return a sorted list $L$ of indices $i$ such that $(f(x[i..(i+m-1)]) \bmod q) = (f(p) \bmod q)$, where $m$ is the length of $p$. It must run in time $O((m+n)\log q)$, where $n$ is the length of $x$, and use $O(k + \log n + \log q)$ working memory, where $k$ is the space required for output list $L$. You are required to implement this.

6. `modPatternMatchWildcard(q,p,x)`: You are required to implement this function. It should also return a sorted list $L$ of indices $i$ such that $(f(x[i..(i+m-1)]) \bmod q) = (f(p) \bmod q)$, where $m$ is the length of $p$, except the definition of $f$ changes now. Figure out what the new definition of $f$ should be, so that `randPatternMatchWildcard` satisfies its requirements. The time and space bounds are the same as those of `modPatternMatch`.

Complete the implementations of the functions `modPatternMatch`, `modPatternMatchWildcard`, and `findN`, and submit a file with the same name – `a4.py`.

## 7 Evaluation

We will be using a mix of both auto-grading and manual grading to assess the correctness of all components of your code. As a requirement, please write a clean and self-explanatory code. You will lose marks if the TA fails to understand your code. Add comments to elaborate wherever necessary. You might be called for a viva on a case-by-case basis. We have primarily decided the following assignment (this may slightly change):

1. Correctness of `modPatternMatch` – autograder
2. Space and Time complexities of your solution – manual grading
3. Check the correctness of other parts – manual grading

*Note*: we will be assessing the **theoretical** time and space complexities of your code. You may ignore the differences that arise in practice due to Python's semantics.

## 8 Example Test Cases

```
>>> modPatternMatch(1000000007, 'CD', 'ABCDE')
[2]
>>> modPatternMatch(1000000007, 'AA', 'AAAAA')
[0, 1, 2, 3]
>>> modPatternMatchWildcard(1000000007, 'D?', 'ABCDE')
[3]
>>> modPatternMatch(2, 'AA', 'ACEGI')
[0, 1, 2, 3]
>>> modPatternMatchWildcard(1000000007, '?A', 'ABCDE')
[]
```