**CS3230 Design & Analysis of Algorithms** **March 7, 2021**

# Programming Assignment 1 Solutions

*Prepared by: Tan Likai*

## 1 Partial Task

Since there are no wildcards, an implementation of the Karp-Rabin algorithm will be sufficient to solve the task. Some students used the Knuth-Morris-Pratt (KMP) algorithm to solve the task in a deterministic manner in $O(m)$ time. This solution is also accepted.

## 2 Full Task

### 2.1 Simple Approach

For this task, the technique of a rolling hash can be generalized to allow for wildcards. Recall that the division hash of a string $S[1..n]$ over the alphabet $\{0,1\}$ is defined as:

$$h_p(S) = h_p(int(S)) = \left(\sum_{i=1}^{n} S[i] \cdot 2^{n-i}\right) \mod p$$

In our case, the characters at positions $i_1, i_2, ..., i_k$ are replaced with wildcards. Suppose the "actual" values at these positions are $w_1, w_2, ..., w_k$ respectively. Then the hash of the string $S$ is

$$h_p(S) = \left(w_1 2^{n-i_1} + w_2 2^{n-i_2} + \cdots + w_k 2^{n-i_k} + \sum_{i \neq i_1, i_2, ..., i_k} S[i] \cdot 2^{n-i}\right) \mod p$$

This gives us a simple $O(m2^K)$ algorithm:

1. Compute the rolling hash as per normal, except that wildcards are ignored. Essentially, compute $X = \sum_{i \neq i_1, i_2, ..., i_k} S[i] \cdot 2^{n-i}$ for each substring $S = M[x..x+n-1]$ of $M[1..m]$.
2. Keep track of the positions of the wildcards in the current string. One efficient method is to use a sliding-window approach, and store the positions in a deque.
3. If the current number of wildcards $k$ is at most $K$, for each of the $2^K$ possible assignments for $w_1, w_2, ..., w_k$, compute the sum $Y = w_1 2^{n-i_1} + w_2 2^{n-i_2} + \cdots + w_k 2^{n-i_k}$ (if the wildcard indices are stored relative to $M$, adjust the equation accordingly).
4. If there exists an assignment such that $h_p(N[1..n]) = X + Y$, increment the number of matches found by one. (The search should terminate after finding a match.)

The second step may take $\Theta(K2^K)$ if done inefficiently. To achieve $O(2^K)$ time, compute the values of $Y$ using either recursion, bitmasks, Gray codes or by the amortized analysis for a binary counter. There is no penalty for inefficiencies in this step.

There is an important caveat that the error probability is much higher (in fact it is amplified by a factor of order $2^k$). However, taking a larger prime $p$ will compensate for this increased error probability.

This approach is sufficient to obtain full credit for the assignment. Note that there are variants of the division hash, but they all function in a similar manner.

### 2.2 Better Approach

However, the approach above does not make use of the fact that we actually do know the correct assignments for each of the wildcards; positions $i_1, i_2, ..., i_k$ must be assigned the characters $N[i_1], N[i_2], ..., N[i_k]$. Then it is sufficient to check that assignment of values. To be precise, for each substring $S[1..n]$ of $M$, compute

$$h_p(S) = \left( N[i_1]2^{n-i_1} + N[i_2]2^{n-i_2} + \cdots + N[i_k]2^{n-i_k} + \sum_{i \neq i_1, i_2, \ldots, i_k} S[i] \cdot 2^{n-i} \right) \mod p$$

The computation takes $O(K)$ per substring. This gives a $O(mK)$ algorithm. The error probability is no worse than the error probability of the original Karp-Rabin algorithm.

## 3 Comments

- Many students asked about the process of choosing $p$. In the lecture, the reason $p$ is randomly chosen is to ensure good success probability even with inputs generated by an adversary (with full knowledge of the algorithm). For this programming assignment, this is not the case, and the inputs can be assumed to be effectively random. This is why we allow alternative methods, including hard-coding a large prime.

- Some students used `BigInteger` in Java to store hashes. While this will reduce the error probability, the class itself is very slow. Recall the Word-RAM model mentioned in Lecture 6; operations like addition, multiplication and modulo cannot be assumed to run in constant time when the operands are large.

- A common mistake in computation is to compute $2^{n-i}$ directly using an exponentiation function. This will result in integer overflows. To avoid overflows, compute the result either iteratively or recursively (see the Wikipedia page on "Modular exponentiation").

- Finally, there is a simple deterministic algorithm that achieves $O(m2^K)$ time using the KMP algorithm by filling the wildcards with all 0-1 strings of length $K$ in a cyclic manner. There is also a much more advanced approach using the Fast Fourier Transform which counts wildcard matches in $O(m \log n)$ time (detailed in the paper "Simple Deterministic Wildcard Matching" by Clifford et al at https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.5266&rep=rep1&type=pdf). Currently, we are not aware of a deterministic algorithm that always beats $O(mK)$ time.