15-451/651 Algorithms, Spring 2021
Recitation #4 Worksheet

Recap of this week's lectures:

- Picking a random prime, and the density of primes
- String equality testing and Karp-Rabin Fingerprinting

Streaming

Missing Numbers: Suppose I give you a stream of $n-1$ elements, which contains all the numbers from 1 thru n *except one of them*. (The numbers *do not* appear in sorted order.) Clearly you can figure out the missing number by storing all $n-1$ numbers and looking for the missing number. How can you output the missing number with only $O(\log n)$ space? What if there are two missing numbers: can you again use only $O(\log n)$ space?

Solution: For one missing number, you can store the sum of all numbers seen so far. Then finally subtract that from $\frac{n(n+1)}{2}$ to get the missing number. For two, you can store, e.g., the sum, and the sum of their squares. Then you'll know $a+b$ and a^2+b^2, and can solve for the answer.

You could also have stored the sum and product of the numbers seen, but that requires more space. The product of the numbers could be as large as $\Omega(n!)$ which requires $\Omega(n \log n)$ bits to store.

Fingerprinting

A String Matching Oracle: In this recitation we generalize the fingerprinting method described in lecture. Let $T = t_0, t_1, \ldots, t_{n-1}$, be a string over some alphabet $\Sigma = \{0, 1, \ldots, z-1\}$. Let $T_{i,j}$ denote the substring $t_i, t_{i+1}, \ldots, t_{j-1}$. This string is of length $j - i$. We want to preprocess T such that the following comparison of two substrings of T of length ℓ can be answered (with a low probability of a false positive) in constant time:

$$\text{Test if } T_{i,i+\ell} = T_{j,j+\ell}$$

First of all let's define the fingerprinting function. Let p be a prime, along with a base b (larger than the alphabet size). The Karp-Rabin fingerprint of T is

$$h(T) = (t_0 b^{n-1} + t_1 b^{n-2} + \cdots + t_{n-1} b^0) \bmod p$$

From now on we will omit the mod p from these expressions.

Now, to preprocess the string T, we will compute the following arrays for $0 \leq i \leq n$:
(*Don't forget we are omitting the* mod*s!*)

$$\begin{aligned} r[i] &= b^i \\ a[i] &= t_0 b^{i-1} + t_1 b^{i-2} + \cdots + t_{i-1} b^0 \end{aligned}$$

Give algorithms for computing these in time $O(n)$:

Solution:

$$r[i] = \begin{cases} 1 & \text{if } i = 0 \\ r[i-1] * b & \text{otherwise} \end{cases}$$

$$a[i] = \begin{cases} 0 & \text{if } i = 0 \\ a[i-1] * b + t_{i-1} & \text{otherwise} \end{cases}$$

Notice that

$$h(T_{i,j}) = a[j] - a[i] \cdot r[j-i]$$

Prove that this is the correct expression.

Solution:

$$\begin{array}{rcccc} a[j] & = & t_0 b^{j-1} + t_1 b^{j-2} + \cdots + t_{i-1} b^{j-i} & + & t_i b^{j-i-1} + \cdots + t_{j-1} b^0 \\ & = & a[i] \cdot b^{j-i} & + & h(T_{i,j}) \end{array}$$

So the end result is that we can test if $T_{i,i+\ell} = T_{j,j+\ell}$ by comparing $h(T_{i,i+\ell})$ with $h(T_{j,j+\ell})$. The probability of a false positive can be made as small as desired by picking a sufficiently large random prime p, as seen in lecture. (Here we are not concerned with bounding the false positive probability.)

Extension to string comparison: Suppose we want to know not just if $T_{i,i+\ell}$ equals $T_{j,j+\ell}$, but we want to know the result of comparing these two strings. (That is we want to know if the first is less, equal to, or greater than the second.)

Give an algorithm to do this that runs in $O(\log \ell)$ time:

Solution: Do a binary search to find the smallest k in $0 \le k \le \ell$ where $T_{i,i+k}$ and $T_{j,j+k}$ differ. Return the result of the comparison $t_{i+k} : t_{j+k}$.

Implementation notes:

- When implementing these algorithms it's important to understand the difference between the mathematical mod operator and the "`%`", or the "`mod`" operator that appears in many programming languages. Usually (e.g. in C, C++, Java, Ocaml,...) the value of "`a % b`" has the same sign as `a`. e.g. `(-3) % 5` is `-3`, but using the mathematical mod , as we are using in these notes, $(-3) \bmod 5 = 2$. You must take this into account, or your code will not work.

- The mod operator must be applied often enough so as to guarantee that no overflow occurs. For example if you're computing $\left(\sum_{i=0}^{1000} f_i * g_i\right) \bmod p$. Say you're working in 64-bit signed arithmetic. If the f_i and g_i are up to, say, 10^9, then this sum will probably overflow the available 64 bits. So what you have to do is take the mod after adding each $f_i * g_i$ term into the summation.

- The danger of false positives increases with the increased number of tests being done. So be aware that this must be considered in any deployment of these algorithms. One way to mitigate the effect of false positives is to compute the hash function two or more times using different primes in place of p, or different base values in place of b. Another way is to judiciously check tests that return "equal" to guarantee that the overall algorithm is computing the correct result. (I am not aware of how to do this efficiently in the case of the comparison algorithm described here.)