15-451/651 Algorithm Design & Analysis

Spring 2023, Recitation #4

Objectives

- Practice designing and reasoning about algorithms in the streaming model
- To understand *fingerprinting* as an algorithm design tool

Recitation Problems

1. **(Plausible Majority Elements)** The ε-heavy-hitters algorithm guarantees that it will output all elements $e \in \Sigma$ such that $\text{count}_t(e) > \varepsilon t$ if they exist, but makes no guarantees whatsoever about the sorts of false positives it may emit. Suppose for example that we want to find the majority element (aka $\varepsilon = \frac{1}{2}$).

   (a) Show that it is possible for the ε-heavy-hitters algorithm to output an element which appears only once in an arbitrarily large stream.

   (b) Come up with a one-pass streaming algorithm that guarantees it will output the majority element if there is one, but will never output an element that appears less than a quarter of the time (an implausible majority element).

   (c) What is your algorithm's asymptotic space complexity as a function of t and $|\Sigma|$?

   (d) Generalize your algorithm to guarantee it will output all elements e such that $\text{count}_t(e) > \varepsilon_1 t$ but never output an element e' such that $\text{count}_t(e') < \varepsilon_2 t$ for $\varepsilon_1 > \varepsilon_2$. What is its asymptotic space complexity?

2. **(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 $0 \le i \le n$: (*Don't forget we are omitting the* mod*s!*)

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

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

(b) Find an expression for $h(T_{i,j})$.

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.)

3. **(Optional: Palindrome Counting)** Given a text $T \in \Sigma^n$ represented as an array of characters, devise an algorithm to count the number of substrings of T that are palindromes in $O(n \log n)$ time with error rate at most some given $\epsilon > 0$.

(a) Find a way to check in $O(1)$ if a given substring $T_{i,j}$ is a palindrome (with high probability). You can use $O(n)$ preprocessing time.

(b) Now, find the number of palindromic substrings of T. (Hint: If $T_{i,j}$ is a palindrome, then what other substrings do we know are palindromes?)

Fun fact: This problem can be solved deterministically (without hashing) in $O(n)$ using Manacher's Algorithm.

Further Review

1. **(General analysis of Karp-Rabin)**

   (a) In lecture we analyzed the complexity of the Karp-Rabin algorithm for $\Sigma = \{0, 1\}$, and showed that to achieve 1% error, this required a random $O(\log m + \log n)$-bit prime. Generalize this for any Σ. How many bits should p be to retain 1% error?

   (b) Now suppose we want an error rate of δ, how large should our prime be?

2. **(Keeping a sample)** Consider a data stream of items $\langle x_1, x_2, ... \rangle$, where item $x_i \in U = \{0, 1, ..., u - 1\}$ arrives at time i. We would like to design an algorithm to maintain a uniformly randomly sampled item from the stream. More specifically, want our algorithm to maintain a variable X such that at any given time t, for all i

   $$\Pr[X = x_i] = \frac{1}{t}.$$

   That is, each item seen so far has an equal probability of being the sampled one.

   (a) Give an efficient streaming algorithm for this problem

   (b) Prove that this algorithm has the desired property

   (c) How many bits of space does the algorithm require?

3. **(Keeping a bigger sample)** We would like to generalize our stream sampling algorithm (Problem 2) to maintain a random sample of k elements from the stream. That is, it will maintain k variables $X_1, X_2, ..., X_k$ such that at time t, each of the $\binom{t}{k}$ possible subsets are equally likely. The algorithm will work as follows. Initially pick the first k elements. Then, at each time step $t > k$, decide to replace one of the X_j with x_t.

   (a) With that probability should the algorithm replace one of the X_j with x_t? Which element should it replace?

   (b) Prove that this algorithm has the desired property

4. **(Frequent entries)** In this problem we have an array of n integers $(a_0, a_1, \ldots, a_{n-1})$ in the range $\{0, 1, \ldots n^2 - 1\}$. We will assume that arithmetic on numbers of $O(\log n)$ bits can be done in $O(1)$ time, and that accessing the array element a_i also takes $O(1)$ time. Space is measured in bits, and we do not count the space to store the input.

   The goal of this problem is to output all integers that occur in the array exactly $\sqrt{n}$ times (assume that $\sqrt{n}$ is an intege). You are allowed exactly two passes through the array.

(a) We might find it useful to use a universal hash family in this problem. What would we want the domain and range of our hash functions to be?

(b) Using the hash family from the previous problem assume that you can pick a hash function from it in expected $O(1)$ time and that each hash function can be stored in $O(\log n)$ bits. Show how to output integers that occur exactly $\sqrt{n}$ times using expected $O(n)$ time and $O(n \log n)$ bits of memory.

(c) Now show how to improve your memory to only $O(\sqrt{n} \log n)$ bits, using a deterministic algorithm. We will allow your running time to be as large as $O(n^{3/2})$ for this part of the problem.