15-451 Assignment 08

Karan Sikka
ksikka@cmu.edu
Collaborated with Dave Cummings and Sandeep Rao.
Recitation: A
November 14, 2014

1: Streaming Medians

(a)

The probability of getting majority heads is:

$$
\begin{aligned}
&Pr(heads >= k+1) \\
=&Pr(heads = k+1) + Pr(heads = k+2) + ... + Pr(heads = 2k+1) \\
=&\binom{2k+1}{k+1} p^{k+1}(1-p)^{k} + \binom{2k+1}{k+2} p^{k+2}(1-p)^{k-1} + ... + \binom{2k+1}{2k+1} p^{2k+1}(1-p)^{0} \\
=&\binom{2k+1}{k+1} \frac{(1-\varepsilon)}{2}^{k+1} \frac{(1+\varepsilon)}{2}^{k} + \binom{2k+1}{k+2} p^{k+2}(1-p)^{k-1} + ... + \binom{2k+1}{2k+1} p^{2k+1}(1-p)^{0}
\end{aligned}
$$

(b)

(c)

(d) To start off, allocate memory for an array of size l. While $t < l$, add the element to l in the position t. Note that so far all element are in l with equal probability: 1.

From now on, $l \leq t$.

In order for the elements in l to be a uniformly random subset of T, each must have a probability of l/t of being in l at time t.

At time t, we want to insert the element into l with probability l/t. However in order to do so, we must evict an element from l. Choose the element to evict uniformly, so that each element has probability $1/l$ of being evicted..

Note that prior to eviction at time $t-1$, all elements had a probability of $\frac{l}{t-1}$ (inductive hypothesis).

The probability that some element will be evicted is l/t. The probability that it's one element is $1/l$. The probability that an element will NOT be evicted at t is $1 - \frac{l}{t}\frac{1}{l} = 1 - \frac{1}{t} = \frac{t-1}{t}$

Mulitplying that to the old probability gives:

$\frac{l}{t-1}\frac{t-1}{t} = \frac{l}{t}$

2: Counting Substrings

Each edge of the suffix tree has a pair of indices (i, j) representing the starting and ending character of that edge.
The algorithm is simply to sum over all the edges, $j - i$.
You can do this by DFS in $O(n)$.

Proof of correctness:

1. Injection from substring -¿ highlighted path of characters starting at the root
Consider a substring in s. There exists a suffix starting with that substring.
Therefore we can identify the path in the suffix tree corresponding to that suffix,
From there we can highlight the characters of the edges on the path, starting at the root, which correspond to the substring.

Note that this highlighted portion of the suffix path is unique to the substring.
To see why, try adding the substring (optionally concatenate garbage to the end).
You'll see that while adding this to the suffix tree, the highlighted path is not modified.
The only modification to the suffix tree is an additional leaf node.

So any distinct substring maps uniquely to some highlighted path of characters in the suffix tree (starting at the root).

2. Injection highlighted path of characters starting at the root -¿ substring
Any highlighted path of characters starting at the root corresponds to a distinct substring.
Distinctness comes from the construction of the suffix tree. The highlighted path is obviously a substring because it's the prefix of some suffix.

3. Therefore there's a bijection.

The number of those character paths is $j - i$ for each edge since each character on the edge serves as the last character for exactly one distinct substring.

3: LDIS

(a)

Choosing a random prime: Let $|\Sigma|$ be the size of the alphabet. Note that it is upper-bounded by a constant.

Turn the string into a binary string by replacing each character with a binary string of p bits where $p = lg(|\Sigma|)$.

If the size of the original string was T, it is now t where $t = Tlg(|\Sigma|)$ which is in the order of T since log sigma is upper bounded by a constant.

Choose a random prime between 1 and $K = 5 * lg(\Sigma) * n * ln(lg(\Sigma)n)$.

You can do this by picking a random integer, checking if it's prime,
and trying again if not. This algorithm is expected $O(log(K))$ which is $O(log(p) + log(t) + log(log(p) + log(t))$ which is in $O(log(t))$

Modified Karp-Rabin: Now we will compute the Karp-Rabin hashes of each prefix and the string of the same length following it.
Starting at i = 1, compute h(s[0:i]), h(s[i:2i])
Increment i and compute the hashes again. Note that this takes constant time since $s[0 : i] \implies s[0 : i + 1]$ and $s[i : 2i] \implies s[i + 1 : 2i + 2]$ so to compute the hash we only have to do a constant time adjustment to the previously computed hash.

Repeat until $i > t/2$. Return the max i for which the two hash computed at the ith iteration are equal.

Proof that Pr[false positive] <1/2: Next we will show the probability of a false positive is less than 1/2.

Let the length of the string be n.
Suppose for any fixed locations i and $2i$ the probability of an incorrect match is upper-bounded by some δ.
Then by summing over $n/2$ locations the probability of any incorrect match is at most $n/2 * \delta$.
We want $n/2 * \delta < 1/2$ or equivalently $\delta < 1/n$.

We make an incorrect match when the hash a of one substring is equal to the hash b of another, modulo some random prime q.
Formally, this occurs when $q|a - b$.
A string like a or b is at most $n/2$ characters which is represented in binary with at most $lg(\Sigma) * n/2$ bits. Let $p = lg(\Sigma) * n/2$.

Then $a - b$ has at most p distinct prime divisors since it's a p-bit number and each prime divisor is at least 2.
In the case of a false answer, $q|a - b$, then q must have been one of the p prime divisors of $a - b$.

We want to choose a prime such that the chance that it's one of the p prime divisors is less than $1/n$, so that the total probability of failure is less than $n/2 * 1/n$ which is less than 1/2.
Equivalently we want to choose K large enough so that there are at least pn primes between 2 and K.

If there are $\pi(x)$ primes between one and x, then $\pi(x) \geq \frac{7}{8}\frac{n}{ln(n)}$. Thus we want to choose K such that $\pi(K) \geq \frac{7}{8}\frac{K}{ln(K)} > pt$.

Setting $K = 5 * lg(\Sigma) * n * ln(lg(\Sigma)n)$ acheives this result.

(b)

Construct a suffix tree for s in linear time.

We will preprocess the suffix tree to store at each node, the following value:
Let p be the pattern represented by the node.
Let L be the length of the pattern.
Store the index of the last occurence of p in the string, which starts on or before index L.

How we do the above is explained later.

Then, the algorithm proceeds as follows:

Do a DFS where you keep track of the length of the prefix at a node, called L, cache the index of the last occurence of P starting on or before index L+1.

Ask your children, what's your last occurence, and since your L+1 is strictly greater than my L+1, I'll filter out the invalid answers and still have the correct last occurence before L+1.

Traverse the suffix tree starting from the root, taking the edge which leads you to a node representing a pattern which is a prefix of the original string. At each node, keep track of the length of the prefix, and check whether the length of the value we computed earlier is equal to L. Keep a running max length of the prefix for which this condition is true.

TODO Correctness TODO details of how to do the precomputation