15-451/651 Algorithms, Spring 2018

Homework #3 **Due: February 22, 2018**

All problems on written HWs are to be done individually, no collaboration is allowed. (And there is one bonus problem.) Otherwise same rules as before.

(25 pts) 1. **(Some Streaming Specialities)**

Suppose we are given a positive integer K, and we want to estimate the number of distinct elements in a stream $a_1, a_2, \ldots, a_n$. In particular, we want to find out if the number of distinct elements is at most K ("low"), or at least $2K$ ("high"). Here is an algorithm:

- Fix a 2-universal hash family $H : U \to \{0, 1, 2, \ldots, 4K-1\}$, and pick a hash function h from it.
- If for some $t \in \{1, 2, \ldots, n\}$, we have $h(a_t) = 0$, say "High"; else say "Low". (So, we don't need to keep a hash table to run this algorithm, just a flag that tells us if anything has hashed to 0.)

(i) Show that if the number of distinct elements is at most K, then

$$\mathbf{Pr}[\text{we say "High"}] \leq 1/4.$$

(ii) Show that if the number of distinct elements is at least $2K$, then

$$\mathbf{Pr}[\text{we say "High"}] \geq 3/8.$$

(Hint: Show this for the case when the number of distinct elements is $2K$. How does the probability behave when this number increases.)

(iii) Now to boost the success probability, we consider m copies of the above data structure, with independently chosen hash functions. Some of these might say "High" and some say "Low". If the fraction of Highs is larger than $\frac{1}{2}(\frac{1}{4} + \frac{3}{8}) = \frac{5}{16}$ then we say "High", else we say "Low". Prove that the probability that this boosted data structure makes a mistake is at most $2e^{-m/128}$. (Let's call this data structure $\textsc{HiLo}(K)$.)

(iv) Run one copy of $\textsc{HiLo}(K)$ for each value of $K = 1, 2, 4, 8, \ldots, U/2$. As an estimate for $\|x\|_0$, output the smallest value K^* for which $\textsc{HiLo}(K^*)$ says "Low". Show that if $m = c(\ln \ln U + \log 1/\eta)$ for some large enough constant c, all copies of $\textsc{HiLo}$ will answer correctly with probability at least $1 - \eta$. In this case observe that our estimate is correct to within a factor of 4. Also, ignoring the space to represent your hash function, how many bits of space do you need to store your entire data structure? Try to optimize this.

One can get a $(1 + O(\varepsilon))$-approximate estimate by looking at powers of $(1 + \varepsilon)$ instead of powers of 2, we just kept it to powers of 2 to make it cleaner.

Hints: For the first two parts, recall the Boole-Bonferroni inequalities which state that for any set of events A_i we have

$$\sum_i \mathbf{Pr}(A_i) - \sum_{i<j} \mathbf{Pr}(A_i \cap A_j) \leq \mathbf{Pr}\left[\bigcup_i A_i\right] \leq \sum_i \mathbf{Pr}(A_i).$$

For the third, use Hoeffding's inequality, which states that for m independent 0-1 valued random variables $X_1, \ldots, X_m$ with $p_i = \mathbf{E}[X_i]$, we have

$$\mathbf{Pr}\left[\left|\frac{1}{m}\sum_{i=1}^{m} X_i - \frac{1}{m}\sum_{i=1}^{m} p_i\right| \geq \epsilon\right] \leq 2e^{-2m\epsilon^2}.$$

(25 pts) 2. **(Rolle thee Olde Diee)**

Roll-the-Die is played by two players A and B. The board has positions $0, 1, \ldots, n$. The game state is defined by the position of A and the position of B on the board. The players take turns rolling a standard 6-sided die and moving toward zero by that many steps. The first player to reach (or pass) position 0 wins.

(a) Give an $O(n^2)$ algorithm that computes the probability P_{ij} that the first player wins, given that the first player starts at i and the second starts at j, with $i, j \in \{0, 1, \ldots, n\}$.

(b) Now change the problem slightly: if a 6 is rolled, then the player does not move. Solve part (a) with these changed rules.

(25 pts) 3. **(Palindrome Deconstruction)**

As you know a palindrome is a string which is invariant under reversal. For example "madamimadam" is a palindrome. As are "x" and "yy" and "xyyx" and "pqp", or indeed, the empty string.

Given a string S you can reduce it to nothing by repeatedly deleting palindromic substrings. For example "xabayx" can be "deconstructed" by removing the palindrome "aba" leaving "xyx". Now we finish the job by removing "xyx". So two palindrome removals are sufficient. You could also do it in six operations by removing one character at a time. (A single character is always a palindrome.)

Give an $O(n^3)$ algorithm to compute the minimum number of palindrome removals necessary to deconstruct a string S of length n.

(25 pts) 4. **(Palindromes in Las Vegas)**

As you know from #3, a palindrome is a string that is the same as its reversal. In this problem you will write a program that takes as input a string S of length n and outputs longest (contiguous) substring of S that is a palindrome. In case of ties, the one starting earliest in S is preferred. It's easy to do this in $O(n^2)$ time. However by use of Karp-Rabin fingerprinting and binary search, this can be reduced to expected $O(n \log n)$ time[1].

- It is possible preprocess a string S so that computing the fingerprint of a range $S[i, j)$ (meaning the substring $S_i, S_{i+1}, \ldots, S_{j-1}$) can be done in $O(1)$ time. You should use this technique.
- You will be given a bound B. The prime modulus p your algorithm uses for fingerprinting should a random prime the range $[B, 2B]$. Your program should compute this by repeatedly generating random numbers in the range and testing them. You don't need complicated primality testing algorithms: for the range of Bs we use, you can use a brute-force search that will take about $O(\sqrt{B})$ time.
- For the fingerprinting, you should interpret the string S as a sequence of characters in base 256. So `h("501")` would be $(53 * (256)^2 + 48 * (256)^1 + 49 * (256)^0) \bmod p$, since 53 is the ASCII value for the character `"5"`, etc.
- Finally, your algorithm should protect itself from the fact that fingerprinting has false positives, and might deem two strings to be equal when they are not. To do this, recall Recitation #3 about Las Vegas and Monte Carlo randomized algorithms.

  When your algorithm purports to have found the best palindrome, do a brute-force test to make sure it IS a palindrome. If it is not a palindrome, then start the whole process over with an independent random prime modulus p. In this way you will convert a Monte Carlo algorithm (which might give the wrong answer) to a Las Vegas one (which has randomized running time, but is guaranteed to give the correct answer.) Your program's running time will be limited to 10 seconds.

Input The input consists of two lines. The first contains B. $3 \leq B \leq 10^9$. The second line contains a string S. It will contain only regular printable characters, and no spaces, tabs or newlines. (There will be a newline after S, but that is not part of S.) $1 \leq |S| \leq 10^6$. The parameters will be chosen such R (the expected number of moduli tried) satisfies $R \times |S| \leq 1.1 \times 10^6$.

Output For each choice of prime modulus, output one line of the form `trying modulus 17`. This list of moduli are followed by a line containing the starting index (0-based) and the length of the palindromic substring found. The palindrome specified by the last line is unique. The lines prior to that are random and may differ from run to run. See the samples below.

Samples For example if the input is:

```
200
'Twas-brillig,-and-the-slithy-toves-did-gyre-and-gimble-in-the-wabe.
```

[1] Although there are other methods to solve this problem, we expect you to do it in a manner consistent with the description here. You will not receive any points for other approaches.

Then the output might be:

```
trying modulus 353
trying modulus 367
trying modulus 239
start=35 length=5
```

And if the input is:

```
1000000000
12233344445555666778
```

Then the output could be:

```
trying modulus 1492891781
start=6 length=4
```

(1 bonus) B3. **Sands of Time (or, Time is Money)**

There are n sand-timers located in a long straight corridor at positions $x_1, x_2, \ldots, x_n$ (measured in yards). Initially, at time zero, each sand-timer contains m units of sand. The sand trickles from the top chamber to the bottom one at a rate of 1 unit per second for each timer. You start at position 0 on the line and you can walk at a rate of 1 yard/second (in either direction, and reversing whenever you want). When you come to a timer, you get all the sand in the top chamber, which you can then exchange for an equal amount of money.

Give an algorithm ($O(n^3)$) to compute the maximum amount of sand you can possibly collect. Example: If three timers each with 15 units of sand are located at positions -3, 1, and 6, then the most sand you can collect is 25 units. First get 14 units of sand at position 1, then move to position -3 and collect 10 units, then move to position 6 and collect the remaining 1 unit.