15-451/651 Algorithms, Spring 2018

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

(25 pts) 1. **Streaming with Nearly Optimal Space**

We are given a stream of insertions of length m with items drawn from $[n] = \{1, 2, \ldots, n\}$. If there is a majority item $i^* \in [n]$, that is, an item for which its number of occurrences f_{i^*} satisfies $f_{i^*} > m/2$, then we want to return i^* and an estimate $\hat{f_{i^*}}$ with $|\hat{f_{i^*}} - f_{i^*}| \leq \epsilon m$, for a given $\epsilon > 0$. If there is no majority item, we can return anything.

We can solve this using the "Misra-Gries" algorithm in class which maintains $O(1/\epsilon)$ counters and $O(1/\epsilon)$ identities, and so $O((1/\epsilon)(\log m + \log n))$ bits of space. For simplicity, let's assume $m = \Theta(n)$ and that the algorithm knows m before reading the stream. The Misra-Gries algorithm then uses $O((1/\epsilon)\log n)$ bits of space.

You will now show how to use only $O((1/\epsilon)\log(1/\epsilon) + \log n)$ bits of space.

(a) (15 points) Suppose for each insertion in the stream, we keep it with probability p, and otherwise discard it. Let f_i' be the number of occurrences of item i among the stream updates we keep. If $p = \min(1, 400/(m\epsilon^2))$, show with probability at least $99/100$, simultaneously for all $i \in [n]$, $|f_i'/p - f_i| \leq (\epsilon/2)m$.

> **Hint:** Chebyshev's inequality may help, which says for a random variable X with expectation $\mathbf{E}[X]$ and variance $\mathbf{Var}[X]$, for $\lambda > 0$, $\mathbf{Pr}[|X - \mathbf{E}[X]| \geq \lambda] \leq \frac{\mathbf{Var}[X]}{\lambda^2}$.

By simultaneously, more precisely, we want you to show

$$\mathbf{Pr}[\forall i, |f_i'/p - f_i| \leq (\epsilon/2)m] \geq 99/100.$$

(b) (10 points) Give a (1-pass) streaming algorithm to find the majority item $i^* \in \{1, 2, \ldots, n\}$, if it exists, and output $\hat{f_{i^*}}$ with $|\hat{f_{i^*}} - f_{i^*}| \leq \epsilon m$. Your overall algorithm should succeed with at least $9/10$ probability and use $O((1/\epsilon)\log(1/\epsilon) + \log n)$ bits of space.

> You may assume that $p = 2^{-j}$ for some j, and so you can sample stream insertions independently by generating $j = O(\log n)$ uniform bits for each stream insertion, and discarding these random bits after deciding to keep the insertion or not. You can also assume that hash functions $h : [M] \to [N]$ can be drawn from a universal family, stored, and evaluated, using $O(\log(MN))$ bits of space, for any M and N.
> **Hint:** Think about running some of the streaming algorithms we saw in class, possibly on the substream of insertions we keep from part (a).

Don't worry too much about the constant factors, such as the probability 99/100 in the first part or 9/10 in this part. They are both adjustable by adjusting constants. Just try to get some constant probabilities for both parts.

Solution: Note that f_i' is a binomial variable with $n = f_i$ and p. Thus, $\mathbf{E}[f_i'] = f_i p$ and $\mathbf{Var}[f_i'] = f_i \cdot p \cdot (1-p)$. Now we plug into Chebyshev's inequality, giving us that:

$$\begin{aligned}
\mathbf{Pr}[f_i'/p - f_i \geq (\epsilon/2)m|] &= \mathbf{Pr}[|f_i' - f_i p| \geq (\epsilon/2)mp] \\
&= \mathbf{Pr}[f_i' - \mathbf{E}[f_i']| \geq (\epsilon/2)mp] \\
&\leq \frac{Var[f_i']}{((\epsilon/2)mp)^2} \\
&\leq \frac{f_i \cdot p \cdot (1-p)}{(\epsilon mp/2)^2} \\
&\leq \frac{f_i \cdot p}{(\epsilon mp/2)^2} \\
&\leq \frac{4 \cdot f_i}{\epsilon^2 \cdot m^2 \cdot p} \\
&\leq \frac{f_i}{100m}
\end{aligned}$$

Now, to finish the problem we note that the probability that the inequality holds for all i is one minus the probability that it is false for any i.

$$\begin{aligned}
\mathbf{Pr}[\forall i, |f_i'/p - f_i| \leq (\epsilon/2)m] &= 1 - \mathbf{Pr}[\bigcup_i (|f_i'/p - f_i| \geq (\epsilon/2)m)] \\
&\leq 1 - \sum_i \mathbf{Pr}[|f_i'/p - f_i| \geq (\epsilon/2)m] \\
&\leq 1 - \sum_i \frac{f_i}{100m} \\
&\leq 1 - \frac{1}{100} \\
&= \frac{99}{100}
\end{aligned}$$

Solution: We first describe the algorithm:

(a) run the Misra-Gries (Heavy hitter) with one counter/memory cell on every element.

(b) Concurrently keep a hash-table which uses a universal hash function $h : [n] \to [40/\epsilon]$. Increment $h(i)$ whenever i is seen **in the sub-stream from part a**. (CountMin)

(c) Return $h(i^*)/p$, where i^* is the majority element that Misra-Gries found.

Correctness: We claim that the result returned by the algorithm is the right majority element and within ϵm of f_i^* with probability $9/10$.

From class, we know that i^* will be correct, so $\mathbf{E}[h(i^*)] \leq f_i' + m'\epsilon/40$, where m' is the size of the sub-stream from part a. By Markov's inequality, $m' \leq 20mp$ with probability $19/20$.

So with probability $19/20$, $\mathbf{E}[h(i^*)] \leq f'_{i*} + mp\epsilon/2$, with additional probability $99/100$, the bound from part a is satisfied, so $\mathbf{E}[h(i^*)/p] \leq f'_{i^*} + m\epsilon/2 \leq f_i + m\epsilon$. Finally, to bound the probability that $h(i^*) - f_i \leq m\epsilon$, we can run the algorithm with $\epsilon' = \epsilon/100$ instead, and then use Markov's inequality to give a probability of $99/100$ that we are within bounds.

Similar to part a, the probability that this fails is less than the probability that either our sub-stream is too large ($1/20$ chance), or that the sub-stream from part a didn't meet the bound from part a ($2/100$ chance). In total the chance that we succeed is greater than $(1 - 1/10) = 9/10$.

Space: The total space used up is $\log n$ for step a, and $40/\epsilon$ counters of size at most $\log(20mp) = \log(800/\epsilon^2) = O(\log(1/\epsilon))$ (the size of the sub-stream), in addition to the space for the hash functions, which is $\log(40/\epsilon) + \log(n)$. In total we used $O(\frac{1}{\epsilon}\log 1/\epsilon + \log n)$ space.

(25 pts) 2. **A Colorful Tree**

Let $G = (V, E)$ be an undirected graph with the vertex set V and edge set E, where each edge has a positive weight w_e. Let $|V| = n$. A K-partition of G is a coloring of the vertices with two colors (red/blue) such that K nodes are colored red, and the remaining $n - K$ nodes are colored blue.

The input is a graph and an integer K, the goal of the K-PARTITION problem is to find a K-partition of G where the total weight of split edges (edges with one endpoint red and the other blue) is minimized. In general the problem is NP-hard.

However, if the graph is a tree, the K-PARTITION problem becomes easier. Design a polynomial-time algorithm to solve this tree case. Prove the correctness of your algorithm and analyze its running time. For full credit, your algorithm should take $O(n^3)$ time.

~~You can get up to 18 points if you~~ You will get full credit if you solve it for the special case when the input graph G is a binary tree. Formally, a binary tree (for our purposes) is a tree rooted at some node r, and each node has at most two children.

Solution: We will write a dynamic program to solve this problem. Lets first make two simplifications, namely we will solve the problem on the binary tree, and we will just focus on finding the cost of the minimum K-partition.

Define $C(v, c, K)$ = cost of min K-partition of tree rooted at v where v is colored c.

Our base case is as follows: $C(v, 1, R) = C(v, 0, B) = 0$ when v is a leaf, and every other entry is initialized to ∞

$$
\begin{aligned}
C(v, blue, K) = \min_{a,b \in \{red,blue\}^2} \{ & \min_{j \leq K} \{C(l, a, j) + C(r, b, K - j) + \\
& w_{vl}(1 - \delta(blue, a)) + w_{vr}(1 - \delta(blue, b))\}\} \\
C(v, red, K) = \min_{a,b \in \{red,blue\}^2} \{ & \min_{j \leq K-1} \{C(l, a, j) + C(r, b, K - 1 - j) + \\
& w_{vl}(1 - \delta(red, a)) + w_{vr}(1 - \delta(red, b))\}\}
\end{aligned}
$$

Here $\delta(x, y)$ is the Kronecker delta (1 if $x = y$ and 0 otherwise).

To summarize the DP, if a vertex v can have K red children and wants to be colored red, it will have $K-1$ remaining red vertices under it, which will be split between its two children in some way. Similarly, if v wants to be blue, it will have K remaining red vertices for its children. It will give each of its children arbitrary colors and some number of red vertices, and then take the min cost over all of the choices it gave to its children.

The final answer we return is $\min(C(r, red, K), C(r, blue, K))$. The total number of vertices in the DAG representing this DP is $4nK = O(n^2)$, and at each vertex we do $O(n)$ work taking a min over $4K$ values. Thus the total work for the DP is $O(n^3)$.

To get the actual assignment, we can go back into the DP table and look at which sub-calls were taken in the min and keep building up the assignment.

Solution: Now consider the case when the tree is not a binary tree. For every parent node, we assign the children an random index i, referring to $C_{v,i}$ as the i^{th} child of parent v. There are only n of these because $C_{v,i}$ refers to a unique vertex in the graph.

(a) $F_1(C_{v,i}, c, K)$ = The min cost of partitioning the leftmost i children of v, where the root is color c and there are K red children in the leftmost i children.

(b) $F_2(v, c, K)$ = The min cost of partitioning the sub-trees of v such that there are K red children under it and v has color c.

We see that the answer to the problem is $\min(F_2(r, red, K), F_2(r, blue, K))$ as before, and that we can recursively define the two as follows:

$$F_1(C_{v,i}, c, K) = \min_{c'}\{\min_{j \le K}\{F_1(C_{v,i-1}, c, j) + F_2(C_{v,i}, c', K-j) + \delta(c, c') \cdot w(v, C_{v,i})\}\}$$

$$F_2(v, c, K) = F_1(C_{v,d_v-1}, c, K)$$

The base cases are the same as before, $F_2(v, red, 1) = F_2(v, blue, 0) = 0$ if v is a leaf, and $F_1(C_{v,0}, c, K) = \min_{c'}\{F_2(C_{v,0}, c', K) + \delta(c, c')w(v, C_{v,0})\}$. All other entries are ∞.

Computing a term of F_1 takes $O(n)$ time, and there are $2nK$ elements in the table. F_2 takes $O(1)$ time to fill in, and there are $2nK$ elements in the table, so in total the algorithm takes $O(n^3)$ time to complete as before.

(25 pts) 3. **On The Road**

You and your friends decide to go on a road trip to Miami immediately after the midterm. To plan the trip, you have laid out a map of the U.S., and marked places you think might be interesting to visit along the way. However, the requirements are:

(a) Each stop on the trip must be strictly closer to Miami than the previous stop.

(b) The total length of the trip can be no longer than D.

You and your friends want to visit the most places subject to these conditions. As a first step, you create a directed graph with n nodes (one for each location of interest) and an edge from i to j if there is a road from i to j and j is closer to Miami than i. Let d_{ij} be the length of edge (i, j) in this graph.

Give an $O(mn)$-time algorithm to solve your optimization problem. Specifically, given a directed acyclic graph (DAG) G with n nodes, m edges and with lengths on the edges, and given a start node s, a destination node t, and a distance bound D, your algorithm should find the path in G from s to t that visits the most intermediate nodes, subject to having total length $\leq D$.

(Note that in general graphs, this problem is NP-complete: in particular, a solution to this problem would allow one to solve the traveling salesman problem. However, the case that G is a DAG is much easier.)

Solution: There are two natural ways to solve this problem. Perhaps the easiest is to do a variation on Bellman-Ford. Let $L[v][i]$ be the length of the shortest path from s to v that uses exactly i edges (or infinity if no such path exists). We can fill in this matrix row-by-row (i.e., for $i = 0, 1, 2, \ldots, n-1$) as follows. We start with $L[s][0] = 0$ and $L[v][0] = \infty$ for all $v \neq s$. Now, inductively assuming we have already filled in the ith row of L, we can fill in the $(i+1)$st row using:

$$L[v][i+1] = \min_{edges\ (w,v)} \left(L[w][i] + d_{wv}\right).$$

That is, the shortest path from s to v that uses $i+1$ edges is the minimum, over all edges (w, v) into v, of the shortest path from s to w that uses i edges plus the length of the edge (w, v). Furthermore, when doing the above calculation, we can have a second matrix M that stores in $M[v][i+1]$ the name of the node w that produced the minimum.

Now, since G is a DAG, the number of distinct intermediate nodes on any path is exactly equal to the number of edges minus 1 (if G wasn't a DAG, this might not be true since the path could have cycles). So, we just find the largest value i such that $L[t][i] \leq D$ and then use the M matrix to output the actual path.

The above computation takes time $O(m)$ per row (since every edge is examined once) and there are n rows, so the total time is $O(mn)$.

A second way to solve this problem is to first perform a topological sorting on the vertices, and then to work backwards from t to s, or forwards from s to t. A topological sorting can be computed in linear time using DFS by outputting nodes in the order that they are exited from DFS. (This works because if v is exited before u then there cannot possibly be a path from v to u—think about why.) Working forwards, we store at each vertex v an array A_v such that $A_v[i]$ is the length of the shortest path from s to v that uses exactly i edges. We start out filling in A_s (and deleting all vertices that precede s in the topological ordering) and then work forwards in the ordering using the same equation as the previous method. Essentially, this is just reordering the two loops in the algorithm, and the rest of the analysis is the same. Here, we use the DAG property to make sure that all values needed in computing A_v have already been calculated, since for any edge (w, v), we know w comes earlier in the topological ordering.

(25 pts) 4. **(Palindromes in Las Vegas)** As you know, 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.

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) B2. **A Set of Pills.**

There are n red pills and n blue pills. The weights of all pills, blue and red, are distinct. Moreover, if the weights are denoted by $b_1, b_2, \ldots, b_n$ for the blue pills and $r_1, \ldots, r_n$ for the red pills, then you are guaranteed the "interleaving property" that

$$b_1 < r_1 < b_2 < r_2 < \ldots < b_n < r_n.$$

You have a balance scale that allows you to put a red pill on one side and a blue pill on the other side. In one weighing, it tells you if the red one is heavier, or the blue one is heavier. You are not allowed to weigh two pills of the same color.

You want to find the median weight red pill (i.e., the red pill $b_{\lfloor n/2 \rfloor}$) but only using blue-red comparisons. Clearly, if you do all the n^2 comparisons, you can find this pill.

Give a randomized algorithm where the expected number of weightings is $O(n(\log n)^c)$ for some constant c independent of n. Ideally $c = 0$, but you will get the bonus point for any constant $c \leq 4$, say. If you can do this in a deterministic way, that's even better.

(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.