(25 pts) 1. **(Permutations and Union-Find.)**

   (a) (15 pts) Recall the tree-based union-find data structure from Lecture #7 (Section 4 of the notes). Suppose you are given a sequence of operations, where first all the makeset operations happen, then all the union operations occur, then all finds happen. There are $n$ makesets and $m$ unions and finds. Show that the data structure from class (no changes allowed) incurs a total cost of $O(m + n)$ on such a request sequence.

   (b) (Rest of the points) Now you’re given as input $\pi = (\pi_1, \pi_2...\pi_n)$, a permutation of $1...n$. Let $G_\pi$ be an undirected graph constructed from $\pi$ as follows: It has $n$ vertices, and there’s an undirected edge between $i$ and $j$ where $i < j$ and $\pi_i > \pi_j$. (In other words, there’s an edge for each inversion in the permutation.) E.g., if the permutation was $\pi = (2, 3, 1, 4)$, then $G_\pi$ is the following:

   ![Graph](image)

   Your goal, should you choose to accept it, is to develop an algorithm for computing the connected components of $G_\pi$. I.e., the output of the algorithm on the above input could be $\{1, 2, 3\}$, $\{4\}$

   For full credit your algorithm should run in $O(n)$ time. You may use the result of part (a) if it helps.$^1$

(25 pts) 2. **(Quick Quacks.)**

   (a) We want to maintain a max-stack, which is a data structure that supports the following operations.

   - `push` (number $x$), pushes $x$ on the stack
   - `pop`, pops the top element off the stack
   - `return-max`, returns the maximum number among the elements still on the stack. (Does not push or pop anything.)

   $^1$You need not use the result of part (a), or indeed the union-find data structure if you don’t want to. We know of solutions both ways.
How would you implement a max-stack, while maintaining constant time per operation (worst-case).
Clarifications: You are allowed to allocate infinitely large arrays, etc. On an empty stack, return NULL on a pop or return-max. (Similar clarifications apply to the following parts.)

(b) We want to now maintain a max-queue, which is what you’d expect given the previous definition. It supports the following operations.
- enqueue(number x), adds x to the end of the queue
- dequeue, removes the element at the front of the queue
- return-max, returns the maximum number among the elements still in the queue. (Does not enqueue or dequeue anything.)

Show how to use two max-stacks to implement a max-queue. This max-queue should take \( O(1) \) amortized time per operation. That is, a sequence of \( n \) operations (consisting of some number of enqueue, dequeue and return-max operations in any order) should take total time \( O(n) \).

(c) Consider the streaming setting, where you are making one pass over the stream \( a_1, a_2, \ldots, a_n \), with each \( a_i \) being a number. You are given a number \( r \geq 0 \). For each time \( t \), when you see \( a_t \), you want to output the maximum value among the past \( r \) elements \( a_{t-r+1}, a_{t-r+2}, \ldots, a_t \). E.g., if \( r = 4 \) and the input stream is

\[
3, 17, 3, 9, 2, 0, 7, 2, 8, 9, 1, 8, 4, 5, 9
\]

then the output stream should be

\[
3, 17, 17, 17, 9, 9, 7, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 9
\]

Give an algorithm that makes one pass over a stream of \( n \) numbers to produce the desired output stream, and the total time taken is \( O(n) \). You are allowed enough space to store \( O(r) \) elements. (Note that taking time \( O(rn) \) would be trivial by just storing the most recent \( r \) elements and recomputing the max from scratch each time – you want to do better than that.)

(25 pts) 3. (Let’s Just Hash it Out.)

We saw (in Recitation #3) that \( H \) is \( \ell \)-universal over range \( m \) if for every fixed sequence of \( \ell \) distinct keys \( \langle x_1, x_2, \ldots, x_\ell \rangle \), if we choose a hash function \( h \) at random from \( H \), the sequence \( \langle h(x_1), h(x_2), \ldots, h(x_\ell) \rangle \) is equally likely to be any of the \( m^\ell \) sequences of length \( \ell \) with elements drawn from \( \{0, 1, \ldots, m - 1\} \). It’s easy to see that if \( H \) is 2-universal then it is universal. (Check for yourself, or see recitation notes!)

Consider a universe \( U \) of strings \( s = s_1, s_2, \ldots, s_n \) of length \( n \) from an alphabet of size \( k \). (Each character is an integer in \( \{0, 1, \ldots, k - 1\} \).) Hence \( |U| = k^n \). Assume that \( m = 2^b \). E.g., if \( k = 3 \) then we’re hashing down from \( U = [3]^n \) to \( [m] = [2^b] \).

An interesting universal family \( \mathcal{G} \) (of functions from \( U \) to \( \{0, \ldots, m - 1\} \)) can be obtained as follows. First, generate a 2-dimensional table \( T \) of \( b \)-bit random numbers; recall that \( b = \lg(m) \). The first index of \( T_{i,j} \) is in the range \( [1, n] \) and the second index is in the range \( [0, k - 1] \). Now define the hash function \( g_T() \) as follows:

\[
g_T(s) = \bigoplus_{i=1}^{n} T_{i,s_i}
\]
where “$$\bigoplus$$” represents the bitwise-xor function (recall, each $$T_{i,j}$$ is a $$b$$-bit string). The output of $$g_T(s)$$ is a $$b$$-bit string which is then interpreted as a number in $$\{0, \ldots, m-1\}$$. Note that since each choice of the table $$T$$ gives a hash function $$g_T$$, and $$T$$ is specified by $$n \cdot k \cdot b$$ bits, the family $$G$$ consists of $$2^{nkb}$$ functions.

(a) Prove that $$G$$ is not 4-universal.

Hint: To show that $$G$$ is not 4-universal, you should exhibit 4 distinct keys $$\langle x_1, x_2, x_3, x_4 \rangle$$ such that if you were told the values of $$g_T(x_1), g_T(x_2),$$ and $$g_T(x_3)$$, you could infer the value of $$g_T(x_4)$$ uniquely (without knowing anything else about $$T$$). This will mean that not all 4-tuples of hash-values are equally likely, since the first 3 entries in the tuple $$\langle g_T(x_1), g_T(x_2), g_T(x_3), g_T(x_4) \rangle$$ determined the 4th entry. You can do this using $$n = 2$$ and $$k = 2$$.

(b) Prove that $$G$$ is 3-universal.

(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\(^2\).

- It is possible preprocessor 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 $$B$$s 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 $$53 \cdot (256)^2 + 48 \cdot (256)^1 + 49 \cdot (256)^0 \mod p$$, 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.

\(^2\)Although there are other methods to solve this problem, we expect you to do it in a manner consistent with the description here.
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.

Then the output might be:

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

And if the input is:

1000000000
122334445555666778

Then the output could be:

trying modulus 1492891781
start=6 length=4
(bonus) B3. **The Density of Primes. (1 bonus point.)**

The prime density function, \( \pi(n) \) is defined to be the number of primes less than or equal to \( n \). In Lecture we talked about an interesting result from number theory, the Prime Number Theorem, stating that \( \lim_{n \to \infty} \frac{\pi(n)}{n/\ln n} = 1 \). Here you prove that \( \pi(n) = \Theta(n/\ln n) \) . While this is weaker, it suffices for all our applications — and you’ll be able to prove it for yourself! Let \( N := \binom{2n}{n} \) for some positive integer \( n \).

(a) (Easy) Show that \( \frac{2^{2n}}{2^n} \leq N \leq 2^{2n} \). You should not use Stirling’s approximation for this. Also recall that \( N \) is itself an integer.

(b) (Slightly Harder) Show that the product of all the primes in \( (n, 2n] \) is at most \( N \). Infer that \( \pi(2n) - \pi(n) \leq 2n \). Hence, \( \pi(n) \leq O\left(\frac{n}{\log n}\right) \).

(c) (Slightly Harder Yet) By considering how many times a prime \( p \in [1, n] \) appears in the prime factorizations for \( n! \) and \( (2n)! \), show that the number of times prime \( p \) divides \( N \) is at most \( \log_p (2n) \). Hence infer that \( \pi(2n) \geq \frac{2n}{\log_2 (2n)} - 1 \).