Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Kazaam

You were at Trader Joe's and you heard a song that was an absolute banger. Sadly, you didn't know what song it was.[1] To help with this, you want to build an app called *Kazaam* that can listen to a small section of a song's lyrics and figure out what song it is.

More specifically, given a string of song lyrics L and a query lyric string Q, we want to find if Q appears as a contiguous substring of L. As an example, if $L =$ `'tung tung tung tung sahur'`, the answer would be `True` if $Q =$ `'ahur'`, but `False` if $Q =$ `'tungtung'`.

For this problem, we assume that all our strings are made up of ASCII characters, which map each character to a number between 0 and 127 inclusive. The Python function `ord(c)` gives us the ASCII value of character `c`.

A simple brute force algorithm solves our problem in $O(|L||Q|)$ time: there are $|L|-|Q|+1$ contiguous substrings of L that have length $|Q|$:

$$\mathcal{L} = \{L_i = (l_i, \ldots, l_{|Q|+i-1}) \mid i \in \{0, \ldots, |L| - |Q|\}\}.$$

So for each substring $L_i \in \mathcal{L}$, check whether $L_i = Q$, character-by-character in $O(|Q|)$ time. If we could determine whether $L_i = Q$ in $O(1)$ time instead of $O(|Q|)$ time, then the above algorithm would run much faster, in $O(|L|)$ time. We can use hashing to our advantage!

(a) We start by creating a hash function for our strings. Given some string S, we want to create a function $R(S)$ that turns our **string into a number**, such that $R(S_1) = R(S_2)$ if and only if $S_1 = S_2$.

Since we're using ASCII characters, we can actually just represent our string as a $|S|$-digit base-128 number. For example, say there are 8 characters in our string, and the characters of S are represented as $(s_0, s_1, s_2, \ldots)$. We can compute

$$R(S) = 128^7 s_0 + 128^6 s_1 + 128^5 s_2 + \cdots .$$

In this case s_0 is known as the most-significant digit, which means that it gets assigned the highest power of 128.

Describe an algorithm to compute $R(S)$ using $O(|S|)$ arithmetic operations, i.e., `(-, +, *, //, %)`. Note that computing an exponent here is the expensive part: for instance, 128^8 will take 8 operations. For parts (a), (b), and (c), assume that one arithmetic operation can operate on integers of arbitrary size.

(b) Now we have a way to turn a string into a number using $R(S)$. Consider L_i, which is the i^{th} substring of our song lyrics L of size $|Q|$. Given $R(L_i)$, describe an algorithm to compute $R(L_{i+1})$ using $O(1)$ arithmetic operations.

You can assume you have access to a value $f = 128^{|Q|}$ precomputed.

(c) Describe an algorithm to check if a query string Q appears inside a song lyrics string L using at most $O(|L|)$ arithmetic operations.

[1] `https://i.imgur.com/AnUCXDj.png`

The above algorithms pose a problem: if the string S is large, then $R(S)$ is roughly $\Omega(128^{|S|})$, likely much too large to fit in a single register on your CPU, so we won't be able to perform arithmetic operations on such numbers in $O(1)$ time. Instead of computing $R(S)$ directly, let's instead hash it down to a smaller range, specifically

$$R'(S) = R(S) \bmod p,$$

where p is a randomly chosen large prime number that fits into a machine word so it can be operated on in a CPU register in $O(1)$ time.

If $Q = L_i$, then certainly $R'(Q) = R'(L_i)$, and we will be able to identify the match quickly! However, sometimes $R'(Q) = R'(L_i)$ when $Q \neq L_i$, and we will get a false match. Assume that p can be chosen sufficiently randomly such that false matches are unlikely to occur, i.e., the probability that $R'(Q) = R'(L_i)$ when $Q \neq L_i$ is at most $\frac{1}{|Q|}$, for any $L_i \in \mathcal{L}$.

(d) Given $R'(L_i)$ and the value $f' = 128^{|Q|} \bmod p$, describe an $O(1)$-time algorithm to compute $R'(L_{i+1})$.

(e) Describe an expected $O(|L|)$-time algorithm to solve the same problem in part (c) using our new R'.

Solution:

(a) The general idea is to reuse the powers of 128 from the last iteration. Let's generalize $R(S)$ to

$$T(S,k) = \sum_{i=0}^{k} s_i \cdot 128^{k-i}.$$

Then $T(S,0) = s_0$, $T(S,|S|-1) = R(S)$, and

$$T(S,k) = 128 \cdot T(S,k-1) + s_k.$$

So to compute $R(S)$, compute each $T(S,k)$ iteratively from $T(S,k-1)$ for k from 1 to $|S|-1$. Each step uses 2 arithmetic operations, for a total of $2(|S|-1) = O(|S|)$ operations.

An alternative correct solution is to precompute the powers 128^i from 0 to $|S|-1$ and then evaluate the sum directly.

(b) Observe that

$$R(L_i) = l_i \cdot 128^{|Q|-1} + \sum_{j=1}^{|Q|-1} l_{i+j} \cdot 128^{|Q|-1-j}$$

and

$$R(L_{i+1}) = l_{i+|Q|} + 128 \sum_{j=1}^{|Q|-1} l_{i+j} \cdot 128^{|Q|-1-j}.$$

Therefore,

$$R(L_{i+1}) = 128 \cdot R(L_i) - f \cdot l_i + l_{i+|Q|}.$$

This uses a constant number of arithmetic operations, so the update takes $O(1)$ arithmetic operations.

(c) First compute f using $O(|Q|)$ multiplications. Compute $R(Q)$ and $R(L_0)$, each using $O(|Q|)$ arithmetic operations via part (a). Then scan L from left to right. For each i from 0 to $|L|-|Q|$, check whether $R(Q) = R(L_i)$. If they are equal, return `True`; otherwise, compute $R(L_{i+1})$ from $R(L_i)$ and f using part (b), then continue.

This algorithm is correct because $R(Q) = R(L_i)$ if and only if $Q = L_i$, and we check every length-$|Q|$ substring $L_i \in \mathcal{L}$. The preprocessing takes $O(|Q|)$ arithmetic operations, and the scan performs $O(1)$ work for each of the $|L|-|Q|+1$ substrings, for a total of $O(|L|)$ arithmetic operations.

For the modular version:

(d) This part is the same as part (b), except that the computation should be taken modulo p:

$$R'(L_{i+1}) = \left(128 \cdot R'(L_i) - f' \cdot l_i + l_{i+|Q|}\right) \bmod p.$$

Since all numbers in this computation fit within a constant number of machine words, each arithmetic operation takes worst-case $O(1)$ time, and there are only constantly many operations.

(e) This part is the same as part (c), except that we compute and compare $R'(Q)$ with $R'(L_i)$ values instead of $R(Q)$ and $R(L_i)$. We can compute f' in $O(|Q|)$ time by performing multiplications modulo p, and we can similarly compute $R'(Q)$ and $R'(L_0)$ in $O(|Q|)$ time.

When comparing $R'(Q)$ to $R'(L_i)$, if $R'(Q) \neq R'(L_i)$, then we know $Q \neq L_i$ and can continue. However, when $R'(Q) = R'(L_i)$, it is possible that $Q \neq L_i$, so we spend $O(|Q|)$ time to compare each character of Q against each character of L_i. If the two match, return `True`; otherwise, continue checking L_{i+1}.

The time spent checking each L_i is $O(1)$ when $R'(Q) \neq R'(L_i)$, and $O(|Q|)$ when $R'(Q) = R'(L_i)$. By the assumption in the problem, a false match occurs with probability at most $1/|Q|$ when $Q \neq L_i$, so the expected time spent checking any particular L_i is $O(1)$. Thus the algorithm runs in expected $O(|L|)$ time.

2 TikTok Audit

TikTok's servers are supposed to store exactly one copy of every video ID in $[n] = \{1, 2, \ldots, n\}$. We want to audit the stream using much less space than storing all the IDs. Assume every video ID in the stream is an integer in $[n]$, and that the stream can only be read once from left to right.

(a) Suppose the stream has length $n-1$ and contains $n-1$ distinct video IDs from $[n]$. In other words, exactly one video ID from $[n]$ is missing. Give a one-pass streaming algorithm that outputs the missing ID using $O(\log n)$ bits of memory.

(b) Suppose the stream has length n, but exactly one video ID m from $[n]$ is missing, and exactly one video ID d from $[n]$ appears twice. Every other video ID appears exactly once. Give a one-pass streaming algorithm that outputs both m and d using $O(\log n)$ bits of memory.

Solution:

(a) Maintain a running sum S of all video IDs seen in the stream. At the end, output

$$\frac{n(n+1)}{2} - S.$$

This is correct because $\frac{n(n+1)}{2}$ is the sum of all IDs in $[n]$, and the stream contains every ID except the missing one. The sum is at most $O(n^2)$, so storing it takes $O(\log n)$ bits.

(b) Maintain two running sums:

$$S = \sum_{i=1}^{n} x_i \qquad \text{and} \qquad T = \sum_{i=1}^{n} x_i^2,$$

where $x_1, \ldots, x_n$ are the IDs in the stream. Let

$$S_0 = \frac{n(n+1)}{2} \qquad \text{and} \qquad T_0 = \frac{n(n+1)(2n+1)}{6}$$

be the corresponding sums if every video ID in $[n]$ appeared exactly once.

Since the stream has one duplicate d and one missing ID m, we have

$$S - S_0 = d - m$$

and

$$T - T_0 = d^2 - m^2 = (d-m)(d+m).$$

Let $A = S - S_0$ and $B = T - T_0$. Since $d \neq m$, we have $A \neq 0$, so we can compute

$$d + m = \frac{B}{A}.$$

Now we know both $d - m$ and $d + m$, so

$$d = \frac{(d+m) + (d-m)}{2} \qquad \text{and} \qquad m = \frac{(d+m) - (d-m)}{2}.$$

Thus the algorithm outputs

$$d = \frac{B/A + A}{2} \qquad \text{and} \qquad m = \frac{B/A - A}{2}.$$

The largest value we store is $T = O(n^3)$, which still takes only $O(\log n)$ bits. Therefore the algorithm is one-pass and uses $O(\log n)$ bits of memory.