**CS2040S: Data Structures and Algorithms**

# Recitation 6

*Goals:*

- Clever application of hashing to string matching
- Binary search speedup
- Rolling hash functions

## Longest Common Substring

This week, we will look at the problem of finding the longest common substring in two long texts. One real-world application of this is in genome sequence comparisons: if you have two long DNA strings, you can measure their similarity by the longest common substring.

For the purpose of this problem, assume you have two long strings $A$ and $B$, both of length $n$. Obviously strings can have different lengths in the generalized case, but for simplicity sake we will just round them both up to the longer length.

A string $S$ is a *substring* of $A$ if there is some range $[i, j]$ where $S = A[i \dots j]$. We want to find the longest string $S$ such that is a substring belonging to both $A$ and $B$.

In this recitation, we are not going to ignore the fact that strings can be huge and therefore cannot be compared or read or written in $O(1)$ time:

- Comparing two strings of length $L$ will cost $L$ time!
- Hashing a string of length $L$ will cost $\Theta(L)$ time
- Copying a string of length $L$ will cost $L$ time
- And so on ...

## Naive Attempt

```
   Input: Strings A and B
   Output: Longest common substring S
1  S ← ∅
2  for each substring a in A do
3  |  for each substring b in B do
4  |  |  if a = b and |a| > |S| then
5  |  |  |  S ← a
6  |  |  end
7  |  end
8  end
9  return S
```

**Algorithm:** Exhaustive search on every substring pair

**Problem 1.** What is the running time of this algorithm?

**Problem 2.** This is obviously a bit silly, because if two strings are not the same length, there is no chance they can be identical. Can you improve this a little bit by only comparing strings of the same length? What is the running time of your improved algorithm?

## Binary Search to the Rescue!

**Problem 3.** In our naive strategy, we sequentially and exhaustively explore all possible substring lengths from 1 to $n$. Clearly that seems a little inefficient! How might you apply binary search to speed up the exploration time? What subroutine do you need for comparisons (*Hint:* Treat it as a helper function)? Why does it work?

Let us refer to this subroutine required by our LCS binary search as `subroutineBS`. For the rest of this recitation, we shall look at how to systematically improve its performance.

## Hashing to the Rescue!

**Problem 4.** Can you speed up `subroutineBS` with a hash table? For the conflict resolution strategy of the hash table, use separate chaining. What is `subroutineBS`'s running time now? What would then be the running time for the LCS binary search algorithm?

**Problem 5.** Is there any merit to our earlier non-hash based LCS solutions? What is the main disadvantage of using a hash table here?

**Problem 6.** How might you reduce the space requirements of the hash table?

## Can we go *even* faster?

If we want our `subroutineBS` with hashing to be faster, there are only two potential areas:

1. Reduce string comparisons on *false positives* within each bucket
2. Speed up hash computations on *substrings*

We shall explore these two avenues in the following sections.

## Fingerprinting to the Rescue!

To reduce string comparisons on false positives within each bucket, let us utilize fingerprinting technique again. We shall do so carefully using two hash functions:

Hash $h_1$: Our main hash function mapping a string to a bucket# in the range $[1, n]$ (i.e. table address with chaining)

Hash $h_2$: Our special *fingerprint* hash function mapping a string to an integer in the range $[1, n^2]$

Notice that each of the fingerprints is of size $\log(n^2) = 2 \log n$. We are not going to use this for a hash table though, since that would require too much table addressing space. Instead, we will be storing the fingerprint *along with* the substring in the hash table. When we look up a substring in the table, we first check its fingerprint before checking the (potentially long) original substring. The fingerprint therefore act as a layer of guard against unnecessary string comparisons with false positive substrings in each bucket: only in the worst case when both $a$ and $b$ share the same fingerprint do we incur an additional $O(n)$ time for string comparisons in the bucket.

Explicitly, our `subroutineBS` with fingerprinting is now specified as follows;

```
   Input: Strings A and B, length L, hash functions h_1 and h_2
   Output: true if there is a common substring of length L in A and B, else false
 1 Initialize hash table H which uses hash function h_1
 2 for each L-length substring a in A do                          // n − L + 1 iterations
 3 |   address ← h_1(a)                                                    // Θ(L) time
 4 |   fingerprint ← h_2(a)                                               // Θ(L) time
 5 |   bucket ← H[address]                                                // O(1) time
 6 |   Insert the pair (fingerprint, a) into bucket                        // Θ(L) time
 7 end
 8 for each L-length substring b in B do                          // n − L + 1 iterations
 9 |   address' ← h_1(b)                                                   // Θ(L) time
10 |   fingerprint' ← h_2(b)                                              // Θ(L) time
11 |   bucket ← H[address']                                               // O(1) time
12 |   for each (fingerprint, a) pair in bucket do // Some constant# iterations (SUHA)
13 |   |   if fingerprint = fingerprint' then                             // O(1) time
14 |   |   |   if b = a then                                          // L comparisons
15 |   |   |   |   return true
16 |   |   |   end
17 |   |   end
18 |   end
19 end
20 return false
```

**Algorithm:** `subroutineBS` with fingerprinting

**Problem 7.** In a single execution of `subroutineBS`, what is the expected time spent on string comparisons? Assume SUHA in your analysis. If substring hashing be done in $O(1)$ time, what will be the overall complexity of the LCS binary search algorithm now?

## Rolling Hash

Now we shall look at how we might hash a substring faster. We know that computing the hash of a random string with length $L$ takes $\Theta(L)$ and there's no way to get around that. However, we are not just hashing any random string here—we are hashing substrings! To obtain all the substrings with length $L$, we are implicitly using a window of size $L$ and sliding it across the entire string from start to end, one character at a time. Each step of the sliding window therefore captures a substring of length $L$. Realize that at every single step of the sliding window, we are only removing the frontmost character and appending a new rearmost character to the current substring. The *rolling hash function* is an algorithm which exploits this important property so as to avoid having to rehash the entire substring every step of the way.

A rolling hash is a DS that implements the following ADT:

| Operation | Behaviour |
|---|---|
| `Initialize(`$S$`)` | Takes a string $S$ of length $L$. |
| `DeleteFirst` | Removes the first letter from the string. |
| `AddLast(`$c$`)` | Adds character $c$ to the end of the string. |
| `Hash()` | Returns the hash of the current string. |

Suppose each character occupies a byte, then we can think of the entire string as an $8L$ bit string. If we wanted to store the entire string as one large binary number $S$ of $8L$ bits, then we could:

1. Implement `DeleteFirst` by computing $S - (c \ll 8(L-1))$ where byte $c$ is the first character in the string
2. Implement `AddLast(`$c$`)` by computing $(S \ll 8) + c$

Of course, we do not want to store the entire string: we just want a hash of it. Let us use the hash $h(S) = S \bmod p$ where $p$ is some large prime number.

**Problem 8.** Suppose that given the current substring $S$, we have its hash $k = S \bmod p$. How should we implement `DeleteFirst` and `AddLast`?

**Problem 9.** Give the entire `subroutineBS` algorithm using rolling hash technique.