

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

Algorithm: Exhaustive search on every substring pair

Input: String A and B

Output: Longest common substring S

$S \leftarrow \emptyset$

```
for each substring  $a$  in string  $A$  do
    for each substring  $b$  in string  $B$  do
        if  $a = b$  and  $|a| > |S|$  then
             $S \leftarrow a$ 
        end
    end
end
return  $S$ 
```

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? Why does it work?

Hashing to the Rescue!

Problem 4. With respect to your solution to the previous problem, can you speed up the subroutine with a hash table? For the conflict resolution strategy of the hash table, use separate chaining. What is the subroutine's running time now? What would then be the running time for the entire binary search algorithm?

Problem 5. Is there any advantage to the earlier non-hash based solutions?

Problem 6. How might you reduce the space?

Can we go faster?

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

1. Reduce false positives that incur string comparisons
2. Speed up hashing for *substrings*

We shall explore these two avenues in the following sections.

Fingerprinting to the Rescue!

To speed up string comparisons, 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. The procedure for inserting a substring S into the hash table H using this new proposal is explicitly stated as follows:

Algorithm: Inserting substring into hash table with fingerprinting

Input: Substring S of length L , hash functions h_1 and h_2 , hash table H	
$address \leftarrow h_1(S)$	// $\Theta(L)$ time
$fingerprint \leftarrow h_2(S)$	// $\Theta(L)$ time
$bucket \leftarrow H[address]$	// $O(1)$ time
Insert the pair $(fingerprint, S)$ into $bucket$	// $O(L)$ time

Now, when we look up a substring in the table, we first check its fingerprint before checking the (potentially long) original substring:

Algorithm: Checking if substring exists in hash table with fingerprinting

Input: Substring S of length L , hash functions h_1 and h_2 , hash table H
Output: `true` if S is found in H , else `false`

```

address ←  $h_1(S)$  //  $\Theta(L)$  time
fingerprint ←  $h_2(S)$  //  $\Theta(L)$  time
bucket ←  $H[\text{address}]$  //  $O(1)$  time
for each ( $\text{fingerprint}', S'$ ) pair in bucket do // Some constant# iterations (SUHA)
    if fingerprint = fingerprint' then //  $O(1)$  time
        if  $S = S'$  then //  $L$  comparisons
            return true
        end
    end
end
return false

```

Realize that after obtaining the *bucket*, only in the worst case when both S and S' share the same fingerprint do we incur an additional $O(n)$ time for string comparisons in the bucket.

Problem 7. If we are looking up substring b in the hash table, what is the probability that any substring a of A of length L has the same fingerprint?

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. For this, we need a DS that implements the following ADT:

Operation	Behaviour
<code>Initialize(S)</code>	Takes a string S of length L .
<code>DeleteFirst</code>	Removes the first letter from the string.
<code>AddLast(c)</code>	Adds character c to the end of the string.
<code>Hash()</code>	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 we have the hash h of the current string S (which is equal to $s \bmod p$). How do we implement **DeleteFirst** and **AddLast**?

Problem 9. Give the entire final algorithm for determining if two strings A and B share a substring of length L .