

15-451/651 Algorithm Design & Analysis, Fall 2026

Recitation #3 (Solutions)

Objectives

- Understand the technique of *fingerprinting* and apply it to solve string problems
- Understand the SegTree data structure and its applications
- See how to define and use custom operators to make SegTrees solve even more problems

Recitation Problems

1. (**A String Matching Oracle**) In this recitation we generalize the fingerprinting method described in lecture. Let $T = t_0, t_1, \dots, t_{n-1}$, be a string over some alphabet $\Sigma = \{0, 1, \dots, z-1\}$. Let $T_{i,j}$ denote the substring $t_i, t_{i+1}, \dots, t_{j-1}$. This string is of length $j-i$. We want to preprocess T such that the following comparison of two substrings of T of length ℓ can be answered (with a low probability of a false positive) in constant time:

$$\text{Test if } T_{i,i+\ell} = T_{j,j+\ell}$$

First of all let's define the fingerprinting function. Let p be a prime, along with a base b (larger than the alphabet size). The Karp-Rabin fingerprint of T is

$$h(T) = (t_0 b^{n-1} + t_1 b^{n-2} + \dots + t_{n-1} b^0) \bmod p$$

We are essentially interpreting the characters as digits in an integer in base b instead of base 2 like in lecture. From now on we will omit the $\bmod p$ from these expressions.

Now, to preprocess the string T , we will compute the following arrays for $0 \leq i \leq n$:
(Don't forget we are omitting the mods!)

$$\begin{aligned} r[i] &= b^i \\ a[i] &= t_0 b^{i-1} + t_1 b^{i-2} + \dots + t_{i-1} b^0 \end{aligned}$$

- (a) Give algorithms for computing these in time $O(n)$:

We can use the following recurrences:

$$\begin{aligned} r[i] &= \begin{cases} 1 & \text{if } i = 0 \\ r[i-1] * b & \text{otherwise} \end{cases} \\ a[i] &= \begin{cases} 0 & \text{if } i = 0 \\ a[i-1] * b + t_{i-1} & \text{otherwise} \end{cases} \end{aligned}$$

- (b) Find an expression for $h(T_{i,j})$.

Expression is this:

$$h(T_{i,j}) = a[j] - a[i] \cdot r[j-i]$$

This is why it works:

$$\begin{aligned} a[j] &= t_0 b^{j-1} + t_1 b^{j-2} + \dots + t_{i-1} b^{j-i} & + & & t_i b^{j-i-1} + \dots + t_{j-1} b^0 \\ &= & a[i] \cdot b^{j-i} & + & h(T_{i,j}) \end{aligned}$$

- (c) Explain how to select p such that for $T_{i,i+\ell} \neq T_{j,j+\ell}$, we have

$$\Pr[h(T_{i,i+\ell}) = h(T_{j,j+\ell})] \leq 1/n.$$

Let $F_{i,i+\ell}$ and $F_{j,j+\ell}$ be the “un-modded” base- b encodings of $T_{i,i+\ell}$ and $T_{j,j+\ell}$, respectively (i.e. the Karp-Rabin fingerprints before applying modulus p).

The two hashes collide when

$$F_{i,i+\ell} = F_{j,j+\ell} \pmod{p} \iff p \text{ divides } D = |F_{i,i+\ell} - F_{j,j+\ell}|.$$

But this difference D does not exceed b^n . Therefore, the maximum number of prime divisors D can have is $\log_2(b^n) = n \log_2(b)$.

Recall that for $M = \lceil 2 \cdot s \log_2(s) \rceil$, there exist at least s primes within $\{1, 2, \dots, M\}$. Hence, we set $s = n^2 \log_2(b)$ and $M = \lceil 2 \cdot n^2 \log_2(b) \log_2(n^2 \log_2(b)) \rceil$. Then, the probability that we select a prime from $\{1, 2, \dots, M\}$ that divides D (i.e. a prime that causes a hash collision) is at most

$$\frac{n \log_2(b)}{n^2 \log_2(b)} = \frac{1}{n}.$$

- (d) Argue that this choice of p allows for constant-time modular arithmetic under the word-RAM model.

The size of each integer operand in modular arithmetic is no larger than its modulus p . Therefore, each operand uses no more than $\log_2(M)$ bits. Note that

$$\begin{aligned} M &= \lceil 2 \cdot n^2 \log_2(b) \log_2(n^2 \log_2(b)) \rceil \\ &\leq 2 \cdot n^2 b \cdot \log_2(n^2 b) \\ &\leq 2 \cdot n^2 b \cdot n^2 b \\ &= 2 \cdot n^4 \cdot b^2. \end{aligned}$$

Therefore the size of each operand is at most

$$\log_2(M) \leq \log_2(2 \cdot n^4 \cdot b^2) = 1 + 4 \log_2(n) + 2 \log_2(b).$$

Since b is constant and each word is at least $w \geq \log n$ bits under the word-RAM model, each operand only uses a constant number of words, and modular arithmetic between operands of constant size take constant time.

So the end result is that we can test if $T_{i,i+\ell} = T_{j,j+\ell}$ by comparing $h(T_{i,i+\ell})$ with $h(T_{j,j+\ell})$. The probability of a false positive can be made as small as desired by picking a sufficiently large random prime p .

2. **(Abby's Favorite Problem)** Suppose we start with some array of integers A . Given a sequence of query intervals in the form $[l_1, r_1), [l_2, r_2), \dots, [l_m, r_m)$, return the maximum element and how many times it appears in $A[l_i], \dots, A[r_i - 1]$ in $O(\log n)$ time for each query $[l_i, r_i)$. You can use $O(n)$ time for preprocessing.

Recall that a SegTree supports 2 operations given an array $a[0], a[1], \dots, a[n - 1]$, and an associative combining operation $\oplus$:

- **Assign**(i, x): Execute the assignment $a[i] \leftarrow x$
- **RangeSum**(i, j): Return $\bigoplus_{i \leq k < j} a[k]$

Note that a Segtree doesn't only work with **RangeSum**. Technically, it can work with **Range(any associative combining function)**. All you need to do is decide what value is stored at each node in the SegTree and how to combine two nodes in the SegTree.

We implement this with a SegTree as follows:

- At each node, we would like to store the following data:
type node = max * num occurrences of max
- Each leaf node in the segtree corresponds to an element $A[i]$. Store $(A[i], 1)$ at the corresponding leaf node
- We can define a custom combining function as follows:

```
def combine((m1, c1), (m2, c2)):
    if m1 > m2:
        return (m1, c1)
    else if m2 > m1:
        return (m2, c2)
    else:
        return (m1, c1 + c2)
```

We can then call **RangeSum**(l_i, r_i) while replacing Sum with our combine function for each query $[l_i, r_i)$ to solve our problem

3. **(Crossing intervals)** Suppose we have a list of n intervals $I_i = [a_i, b_i]$ where $0 \leq a_i, b_i < 2n$, such that the endpoints of all of the intervals are distinct (no two intervals ever share an endpoint at either end). A pair of intervals I_i and I_j are *crossing* if they overlap but one does not strictly contain the other. We want to devise an algorithm to count the number of pairs of crossing intervals.



- (a) Give a simple $O(n^2)$ algorithm for the problem

Since we have $O(n^2)$ time we can loop over every pair of intervals and just check if they are crossing:

```

answer := 0
for i in 0 to n - 1 {
  for j in 0 to i - 1 {
    if  $(a_i, b_i)$  and  $(a_j, b_j)$  are crossing then
      answer += 1
  }
}
return answer

```

- (b) Come up with a more efficient $O(n \log n)$ algorithm by making use of a SegTree

For a fixed interval $[a_i, b_i]$, it intersects with any interval $[a_j, b_j]$ that started before it (that is, $a_j < a_i$) and whose right endpoint is inside it (that is $a_i < b_j < b_i$). So, let's go through the intervals sorted by their left endpoint (a) and keep a SegTree that tracks how many right endpoints are in any given range. To count the number of intersecting intervals, since every interval previously considered has $a_j < a_i$ (we are looping in order of a), we just need to count the number of right endpoints in (a_i, b_i) , which is achieved with a **RangeSum** (a_i, b_i) . Then, we add the current interval's right endpoint to the SegTree with **Assign** $(b_i, 1)$.

```

result : int = 0
endpoints : SegTree = array<int>(2*n, 0)
for  $(a_i, b_i)$  in sorted( $((a_i, b_i))$ , key=a) do {
  result = result + endpoints.RangeSum( $a_i, b_i$ )
  endpoints.Assign( $b_i, 1$ )
}
return result

```

Sorting the points takes $O(n \log n)$ with any good sorting algorithm, then we perform $4n$ SegTree operations which take $O(\log n)$ time each, so in total our algorithm runs in $O(n \log n)$ time.