15-451/651 Algorithm Design & Analysis, Spring 2024

Recitation #3

Objectives

- Understand the technique of *fingerprinting* and apply it to solve string problems
- Practice amortized analysis using the aggregate and potential methods

Recitation Problems

1. **(A String Matching Oracle)** In this recitation we generalize the fingerprinting method described in lecture. Let $T = t_0, t_1, \ldots, t_{n-1}$, be a string over some alphabet $\Sigma = \{0, 1, \ldots, z-1\}$. Let $T_{i,j}$ denote the substring $t_i, t_{i+1}, \ldots, t_{j-1}$. 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

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

   From now on we will omit the mod p from these expressions.

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

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

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

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

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, as seen in lecture. (Here we are not concerned with bounding the false positive probability.)

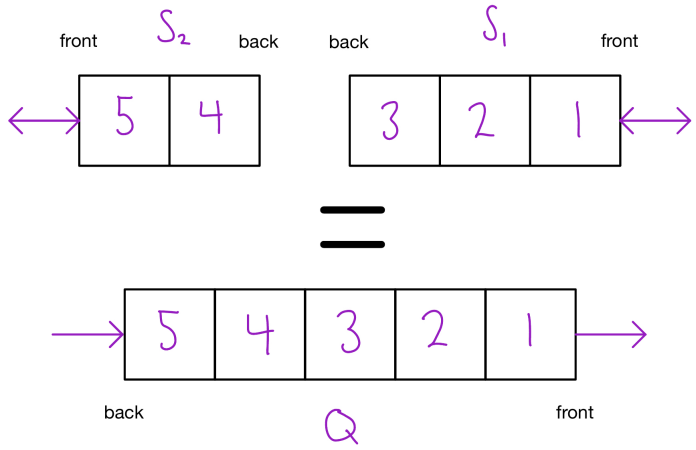
2. **(Deque)** There is a classic method to construct a first-in-first-out queue with $O(1)$ amortized cost operations `pushBack` and `popFront` from two last-in-first-out stacks with cost 1 operations `pushFront`, `popFront`, and `size` as follows:

   Let stacks S_1 and S_2 represent the front/head and back/tail of the queue Q respectively. Then implement the operations by moving all elements from S_2 to S_1 whenever we would need to remove but it is empty:

```
pushBack(x) {
   S2.pushFront(x)
}

popFront() {
   if (S1.size() == 0) {
      for i in range(S2.size()) {
         S1.pushFront(S2.popFront())
      }
   }
   return S1.popFront()
}
```

   Under this implementation the amortized bounds follow from setting $\Phi(S_1, S_2) = 2|S_2|$ and correctness follows from the invariant that the elements of Q are always equal to the elements of S_1 appended to the elements of S_2 in reverse.



   One natural extension of this is to implement a deque, which in addition to the standard queue functionality, also provides `pushFront` and `popBack`, allowing users to insert and remove from either side as they please.

   (a) Suppose you provide symmetric implementations of those methods as follows:

```
pushFront(x) {
   S1.pushFront(x)
}

popBack() {
   if (S2.size() == 0) {
      for i in range(S1.size()) {
         S2.pushFront(S1.popFront())
      }
   }
   return S2.popFront()
}
```

Using the aggregate method, give a sequence of n operations under which each operation has $\Omega(n)$ amortized cost.

(b) Now suppose you had access to a third stack S_3 to use for temporary processing. Come up with a way to implement the deque operations that maintains the invariant but avoids the expensive case above.

(c) Define a potential function $\Phi(S_1, S_2)$ and use it to prove that the operations you de-

fined have $O(1)$ amortized cost.

(d) Suppose that you start with an empty deque and then perform n operations (push or pop from either the front of the back). Bound the total actual cost of these operations.

3. **(Tree of Arrays)** You have a tree with differently-sized sorted arrays at the leaves. You want to update the values of the rest of the nodes in the trees such that they represent the sorted list of all the elements in their subtree (assume all elements are unique). Give an $O(n \log^2(n))$ algorithm *in the comparison model* which does this, where n is the total number of elements in all the arrays.

4. **(Palindrome Counting (Optional))** Given a text $T \in \Sigma^n$ represented as an array of characters, devise an algorithm to count the number of substrings of T that are palindromes in $O(n \log n)$ time with error rate at most some given $\epsilon > 0$.

   (a) Find a way to check in $O(1)$ if a given substring $T_{i,j}$ is a palindrome (with high probability). You can use $O(n)$ preprocessing time.

   (b) Now, find the number of palindromic substrings of T. (Hint: If $T_{i,j}$ is a palindrome, then what other substrings do we know are palindromes?)

Fun fact: This problem can be solved deterministically (without hashing) in $O(n)$ using Manacher's Algorithm.