15-451/651 Algorithm Design & Analysis

Spring 2023, Recitation #3

Objectives

- To see some applications and extensions of SegTrees
- To practice designing custom combination operations for SegTrees
- Learn about ℓ-wise independent hashing and practice proving it

Recitation Problems

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

2. **(ℓ-wise Independent Hashing)** A hash family $\mathcal{H}$ is *ℓ-wise independent* (a.k.a. ℓ-universal) if for all ℓ distinct keys $x_1, x_2, \ldots, x_\ell \in \mathcal{U}$ and every set of ℓ values $v_1, v_2, \ldots, v_\ell \in \{0, 1, \ldots, M-1\}$, we have that

$$\Pr_{h \in \mathcal{H}}[h(x_1) = v_1 \wedge h(x_2) = v_2 \wedge \cdots \wedge h(x_\ell) = v_\ell] = \frac{1}{M^\ell}$$

   Intuitively, this means that if you look at only up to ℓ keys, the hash family appears to hash them truly randomly.

   (a) Is this hash family from $U = \{a, b\}$ to $\{0, 1\}$ (i.e., $M = 2$) universal? 1-universal? pairwise independent?

	a	b
h_1	0	0
h_2	1	0

(b) Can you fill in the blanks in this hash family with values in $\{0, 1\}$ to make it pairwise independent? 3-wise independent?

	a	b	c
h_1	0	0	
h_2			1
h_3	0		
h_4	1	1	0

3. **(Extended Matrix Method)** In lecture we covered the *matrix method* of universal hashing binary vectors from the universe $\mathcal{U} = \{0, 1\}^u$ into a table of size $M = 2^m$ indexed by $\{0, 1\}^m$ where each hash function in the family is defined by a random matrix $A \in \{0, 1\}^{m \times u}$ and

$$h(x) = Ax \mod 2$$

(a) Prove that the matrix method is not 1-universal.

(b) Now suppose we extend the matrix method to be defined as

$$h(x) = Ax + b \mod 2$$

where $b \in \{0, 1\}^m$ is a random binary vector.

Prove that this extension of the matrix method is pairwise independent.

Further Review

1. **(Very independent)** Show that a k-wise independent hash family is also an ℓ-wise independent hash family for any $\ell \leq k$.

2. **(Which is the better hash?)** Which of the following is true? Prove it.

   (a) All universal hash families are 1-independent

   (b) All 1-independent hash families are universal

   (c) Neither of the above

3. **(Fooling an adversary)**

   (a) Suppose that an adversary knows the hash family $\mathcal{H}$ and controls the keys we hash, and the adversary wants to force a collision. In this problem part, suppose that $\mathcal{H}$ is universal. The following scenario takes place: we choose a hash function h randomly from $\mathcal{H}$, keeping it secret from the adversary, and then the adversary chooses a key x and learns the value $h(x)$. Can the adversary now force a collision? In other words, can it find a $y \neq x$ such that $h(x) = h(y)$ with probability greater than $\frac{1}{M}$?
   If so, write down a particular universal hash family in the same format as in part 3, and describe how an adversary can force a collision in this scenario. If not, prove that the adversary cannot force a collision with probability greater than $\frac{1}{M}$.

   (b) Answer the question from (a), but supposing that $\mathcal{H}$ is pairwise-independent, not just universal.

4. **(Target sums in SegTrees)** Consider the standard SegTree interface from lectures, where we support, in $O(\log n)$ time per operation:

   - **Assign**(i, x): Assign $a[i] \leftarrow x$,
   - **RangeSum**(i, j): Return $\sum_{i \leq k < j} a[k]$.

   Suppose we restrict the values $a[i]$ in the SegTree to be non-negative. We want to add an additional operation to our SegTrees:

   - **TargetSum**(v): Return the minimum index i such that **RangeSum**$(0, i) \geq v$, or n if no such i exists.

   That is, we want to find how many elements of the SegTree we would need to sum up in order to reach some target value.

(a) Give an $O(\log^2 n)$-time algorithm for this operation that just uses SegTrees as a block box (i.e., you just use standard SegTree operations and do not modify them in any way)

(b) Give a better $O(\log n)$-time algorithm. This will require using the SegTree's internal representation rather than using it as a black box.

5. **(Range updates *and* range queries??)** We saw in lecture that standard SegTrees support point updates and range queries. We also saw how this can be flipped to support range updates and point queries. Is it possible to support both range updates and range queries? There is in fact a general technique called lazy propagation that supports this for arbitrary operations, but its a bit complicated, so we're going to derive a simpler technique here.

   Our goal is to support the following interface over a dynamic array $a[0], \ldots, a[n-1]$

   - **PrefixAdd**(j, x): Add the value x to $a[0], a[1], ..., a[j-1]$
   - **PrefixSum**(j): Return the sum $\sum_{1 \le k < j} a[k]$

   (a) Suppose we had a data structure that supported the two operations given above. Describe how we could support the slightly more general desired interface:

   - **RangeAdd**(i, j, x): Add the value x to $a[i], ..., a[j-1]$
   - **RangeSum**(i, j): Return the sum $\sum_{i \le k < j} a[k]$

   We are going to support the desired interface by using not one, but *two* standard SegTrees as black boxes. We will make use of the following two important observations:

   (b) Suppose we perform an add operation **PrefixAdd**(j, x). For any subsequent query **PrefixSum**(j') such that $j' \ge j$, how does this add operation affect the result?

   (c) Similarly, for any subsequent query **PrefixSum**(j') such that $j' < j$, how does the add operation affect the result?

   Now let's put everything together!

   (d) Describe an algorithm that uses two standard SegTrees and implements the desired interface for **PrefixAdd**(j, x) and **PrefixSum**(j). The two standard SegTrees should correspond to the two observations made in Parts 5b and 5c.

   (e) Conclude that there exists a data structure supporting **RangeAdd** and **RangeSum** in $O(\log n)$ time.