Due: May 4th (Saturday), 6pm, submit to courseworks.

You may use "standard" arguments without a formal proof—either cite from class or add one sentence showing the main idea.

Problem 1: Independence properties of Tabulation Hashing

Recall the definition of Tabulation hashing: Each w-bit key $x = x_1 \dots x_w$ is partitioned into c blocks of $B = (w/c)$ bits (characters) each $x'_1 \dots x'_c$, and hashed into the XOR of c totally random hash tables : $T(x) := T_1[x'_1] \oplus \dots T_c[x'_c]$, where each T_i is a random table $\in_R [m]^{2^B}$.

1. Prove that this hash function is 3-wise independent, that is: for any three keys x, y, z, the tuple $(T(x), T(y), T(z))$ is equally likely to get mapped to any value in $[m]^3$.
2. Prove that Tabulation hashing is not 4-wise independent: show that there are key values x, y, w, z such that their hashes $T(x), T(y), T(w), T(z)$ are not independent.

Problem 2: Cuckoo Hashing

Prove that Pr[Insert(x) to Cuckoo hash traverses a k-length simple path] $\geq 2^{-O(k)}$, (when we hash n keys to 2 completely random tables g, h each of size $m = (1+\epsilon)n$). This is tight by the argument proved in class (we showed the probability of this event is $\leq 2^{-\Omega(k)}$).

Also show that the probability that cuckoo hashing fails (i.e. it makes $\Omega(\log n)$ "hops" when inserting some key) is $\Omega(1/n)$ (hint: what is the simplest case in which such event happens? Obviously we have excluded graphs with long chains, what other cases are there?).

Problem 3: Longest Non-Overlapping Substring

Devise a linear-time algorithm that for a given string S finds the longest substring x such that there are two positions i, j: $S_i S_{i+1} \dots S_{i+|x|-1} = S_j S_{j+1} \dots S_{j+|x|-1} = x$ and $i + |x| - 1 < j$, i.e. substring x occurs at least twice in S and these two occurences don't overlap.

Problem 4: Longest Common Substring in linear time

The LCS problem is, given 2 strings a, b, to find the longest (contiguous) common substring of T_1, T_2. (e.g., if $T_1 =$ "*alibabalc*" and $T_2 =$ "*abac*", then $LCS(T_1, T_2) =$ "*aba*"). Use suffix trees to solve this algorithmic problem in $O(|T_1| + |T_2|)$ time.

Problem 5: Least Common Ancestor

Consider the (static) LCA problem: Given a binary tree T on n nodes, preprocess it so that for every pair of vertices $u, v \in T$, we can quickly compute $LCA(u, v)$, that is, the node v of minimum depth in T, which is an ancestor of both u, v.

1. Consider the following "Range Minimum" problem RMQ_n: Preprocess an array of length n so that for any 2 entries $i, j \in [n]$, the DS must output the minimum element $x \in A[i] \dots A[j]$. Show that the LCA problem reduces to RMQ_n, i.e,., that any (s, t)-DS for RMQ induces a similar DS for LCA).

   (Hint: *Euler Tour* of a graph may help here).

2. Show how to solve RMQ_n with $O(n \lg n)$ words of space and constant $t = O(1)$ query time.

   (Hint: Consider storing, for every entry in the array, the minimum element at distance 2^i from that entry, for every i. How much space does this take? And why does it suffice for RMQ queries?).

3. For extra credit, show how to reduce the space to linear $O(n)$. To do this, use Indirection + the important fact that the the resulting RMQ problem has all consecutive entries of the array $A[i]$ differ from the next/previous entry $A[i+1]$ by at most ± 1.

4. As an application, show that the LCP problem discussed in class (Given a text T and 2 indexes i, j, determine the longest common prefix $LCP(T[i:], T[j:])$) can be solved in $O(n)$ space and $O(1)$ time.

Problem 6: Finding Blobs

Consider the following dynamic problem: given a changing forest T with n vertices, each either black or white, you are asked to compute sizes of the "blobs" around some vertices under edge addition/removal and color changes. A "blob" around vertex v is the largest connected component containing v that consists of vertices of the same color.

More specifically, your algorithm should support the following types of queries:

1. Change(v) — change color of vertex v,

2. BlobSize(v) — return the size of the "blob" around vertex v,

3. Add(u, v) — add an edge between u and v,

4. Remove(u, v) — remove an edge between u and v.

It is promised that at every point of time T is a forest. Design an algorithm for that problem with linear space, give bounds on its running time. You will get partial credit if your algorithm supports only Change and BlobSize queries.