Midterm Practice Problems

The midterm exam is coming up a week from today, so to help you prepare and practice, here's a collection of random questions on assorted topics from throughout the quarter. Feel free to work through these to review the course topics and sharpen your skills!

Balanced Trees

Red/black trees are an isometry of 2-3-4 trees (B-trees of order two). It's possible to encode other types of B-trees as binary search trees by using more colors.

A B-tree of order four is a B-tree where non-root nodes have between three and seven keys and where the root node has between one and seven keys. A *red/black/blue tree* is a BST where each node is colored red, black, or blue according to a set of color rules. Just as the color rules for red/black trees enforce the isometry with B-trees of order two, the color rules for red/black/blue trees enforce the isometry with B-trees of order four.

Your job in this problem is to develop color rules for red/black/blue trees that enforce an isometry with B-trees of order four. Specifically, your color rules should have the following properties:

- Any red/black/blue tree obeying the color rules encodes a B-tree of order four.
- Any B-tree of order four can be encoded as a red/black/blue tree obeying the color rules.

Splay Trees

Let $S = \{ x_1, x_2, \ldots, x_n \}$ be a set of keys in a splay tree where $x_1 < x_2 < \ldots < x_n$. Prove that if you perform a sequence of lookups of $x_1, x_2, \ldots, x_i$, in that order, then the resulting splay tree will have the following structure:

- The root will be x_i.
- The right subtree consists of an arbitrary key containing all keys greater than x_i.
- The left subtree will be a degenerate tree consisting of a left spine holding keys $x_1, \ldots, x_{i-1}$.

Aho-Corasick String Matching

Suppose that you have a set of pattern strings $P_1, \ldots, P_k$ of total length n where no pattern string is a substring of any other. Building an Aho-Corasick automaton for these strings takes time $O(n)$.

Prove that the time required to find all matches of these patterns in a string of length m using the matching automaton is $O(m)$.

String Searching

The normal string matching problem is too restrictive in many genomics applications where, due either to sequencing errors or random mutations, a pattern string P might not exactly match anywhere in T even though it *almost* matches. The ***k-approximate matching problem*** is the following: given a string T of length m, a string P of length n, and a value $k \geq 0$, determine whether there is a substring of T of length n that matches P in all but at most k places. For example, given the strings

T = `thatthatisisthatthatisnotisnotisthatitititis` P = `matisse`

then P can be placed at the following location relative to T such that it matches in all but 3 places:

```
thatthatisisthatthatisnotisnotisthatitititis
      matisse
```

However, there is no way to place P such that it matches in all but 2 places.

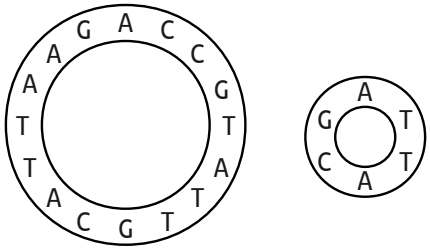
Design an $O(mk + n)$-time algorithm for solving the k-approximate matching problem and prove that your algorithm is correct. In the common case where $k \ll n$, this can be significantly faster than the naïve algorithm. As a hint, this problem is mostly about picking the right subroutine.

There are several different definitions of multivariable big-O notation. For the purposes of this problem, use this definition from our Mathematical Terms and Identities Handout:

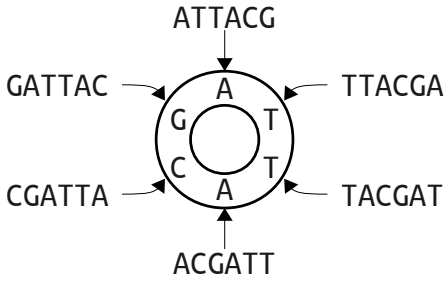
> When multiple variables are involved in an expression, big-O notation generalizes as follows: we say that $f(x_1, \ldots, x_n) = O(g(x_1, \ldots, x_n))$ if there are constants N and c such that for any $x_1 \geq N$, $x_2 \geq N$, …, $x_n \geq N$, we have $f(x_1, \ldots, x_n) \leq c \cdot g(x_1, \ldots, x_n)$.

Suffix Trees and Suffix Arrays

A *plasmid* is a ring of DNA. For example, here are some small plasmids:



We can represent a plasmid as a string by starting at an arbitrary location in the plasmid and listing the characters in clockwise order. Because plasmids don't have a definitive start or end point, there can be many different representations of the same plasmid. For example, below is a plasmid and all six strings that represent it:



In many biological applications, for simplicity, it helps to choose a single string as a "canonical" representation of a plasmid. One simple way to do this is to choose the lexicographically smallest string that represents that plasmid. For example, the above plasmid would be represented as `ACGATT`.

Design an algorithm that, given a string T representing a plasmid, outputs the lexicographically smallest string representing that plasmid. Your algorithm should run in time $O(m)$, where m is the length of the string T. Then, prove that your algorithm is correct.

As a hint, build a suffix tree or suffix array, but not for the input string T.

Count Sketches

In the count sketch data structure, each row consists of an array w of counters. We then choose two hash functions $h : \mathcal{U} \to [w]$ and $s : \mathcal{U} \to \{+1, -1\}$, where h sends each element into a slot in the table and s determines whether we increment or decrement the counter when processing the element. To ***increment***(x), we add $s(x)$ to count$[h(x)]$. To ***estimate***(x), we return count$[h(x)] \cdot s(x)$.

You can interpret the hash function s as assigning a *direction* to each element $x \in \mathcal{U}$, where +1 means "up" and -1 means "down." The ***estimate***(x) procedure then works by returning the net number of steps in the direction indicated by $s(x)$ the counter at position $h(x)$ has taken.

We can generalize this so that each counter is a 2D point as follows. Change s so that it now maps from $\mathcal{U}$ to the set {up, down, left, right} and replace each counter in the array count with a point in 2D space. To ***increment***(x), we compute $h(x)$ to determine which slot x belongs in, then adjust the point in that slot by moving it one step in the direction given by s(x). To ***estimate***(x), we compute $h(x)$ and look at the point it corresponds to. We then return the net number of steps in the direction given by $s(x)$ that point has taken from the origin. For example, if x is assigned the direction "left," then we return the net number of steps to the left of the origin that the corresponding point has taken.

Compare this modified version of the count sketch to the original version of the count sketch in terms of time, space, accuracy, and confidence. Justify your answer.

Hashing and Sketching

In this problem, you'll analyze a randomized data structure called a *group tester* that identifies frequent elements in a data stream. Let $\mathcal{U} = \{ x_1, \ldots, x_n \}$ be a set of n elements. Suppose that we have a stream of elements drawn from $\mathcal{U}$. Let $\boldsymbol{a}$ be the frequency vector for the stream, where $\boldsymbol{a}_i$ is the frequency of x_i.

For any positive integer k, a *k-heavy-hitter* in the data stream is an element x_i where $\boldsymbol{a}_i > \|\boldsymbol{a}\|_1 / (k + 1)$. All data streams have at most k distinct k-heavy-hitters, though some may have fewer than k.

A *group tester* is a randomized data structure for finding k-heavy-hitters. The group tester is parameterized over two values – an integer k and a probability δ – and supports the following operations:

- *gt.**increment***(x), which increments the frequency of element x, and
- *gt.**find-heavy-hitters***(), which returns a list of elements that, with probability at least $1 - \delta$, contains all of the k-heavy-hitters.

Like the count sketch and count-min sketch, the group tester consists of multiple independent rows that produce independent estimates that are then aggregated together. In the first two parts of this problem, you'll analyze individual rows of the group tester. In the third, you'll analyze the overall data structure.

Each row in a group tester consists of an array of $2k$ buckets. We'll associate with each row a hash function $h : \mathcal{U} \to [2k]$ chosen from a family $\mathcal{H}$ of pairwise-independent hash functions.

For each j in the range $0 \leq j < 2k$, define $S_j \subseteq \mathcal{U}$ to be the set of all elements in $\mathcal{U}$ that hash to slot j in the table. Formally:

$$S_j = \{ x \in \mathcal{U} \mid h(x) = j \}$$

Next, define N_j to be the total frequency of all elements that hash to slot j. Formally:

$$N_j = \sum_{x_i \in S_j} \boldsymbol{a}_i$$

(i) Heavy Hitter Distributions

Let x_i be a k-heavy-hitter. Prove that if $x_i \in S_j$, then

$$\mathrm{E}[N_j - \boldsymbol{a}_i] < \frac{\|\boldsymbol{a}\|_1}{2(k+1)}$$

In other words, the expected total frequency of all the elements that hash to slot $h(x_i)$, excluding x_i, is less than $\|\boldsymbol{a}\|_1 / 2(k + 1)$.

You may find the following fact useful: a family of hash functions $\mathcal{H}$ from $\mathcal{U}$ to $[m]$ is pairwise independent iff for any distinct $x, y \in \mathcal{U}$ and for any $r, s \in [m]$, the following holds:

$$\Pr_{h \in \mathcal{H}}[h(x) = r \mid h(y) = t] = \frac{1}{m}$$

As a hint, first try proving that

$$\mathrm{E}[N_j - \boldsymbol{a}_i] = \frac{1}{2k} \sum_{s \neq i} \boldsymbol{a}_s$$

and simplifying the right-hand side to get the desired result.

Using Markov's inequality on your result from part (i), we can prove that if x_i is a k-heavy-hitter that hashes to bucket j, then

$$\Pr\left[N_j - \boldsymbol{a}_i \leq \frac{\|\boldsymbol{a}\|_1}{k+1} \right] \geq 1/2$$

(ii) Majority Elements

An element x_i in a data stream is called a *majority element* if strictly more than half the elements in the data stream are copies of x_i. Prove that if x_i is a k-heavy-hitter, then with probability at least ½ it will be a majority element in the bucket it hashes to.

You can take the following as a given:

> There is a space-efficient data structure that supports ***increment***(x), which increments the frequency of element x, and and ***find-majority***(), which returns a majority element if one exists and returns nothing otherwise.

Each row in a group tester consists of $2k$ buckets, where each bucket is a majority element data structure. Within a single row of the group tester, we perform ***increment***(x) by going to the $h(x)$th majority element data structure and calling ***increment***(x). Similarly, within a single row of the group tester, we evaluate ***find-heavy-hitters***() by calling ***find-majority***() on each of the majority element data structures and returning the set of all elements reported this way.

The overall group tester data structure then consists of d rows, each with its own hash function. The overall ***increment***(x) operation calls ***incrememt***(x) on each row, and the overall ***find-heavy-hitters***() operation returns the union of the sets returned by ***find-heavy-hitters*** in each row.

(iii) Amplifying the Probability

Prove that in a group tester with d rows, the probability that the reported set of k-heavy-hitters actually includes each of the k-heavy-hitters from the data stream is at least $1 - k / 2^d$. This means that if we pick $d = \lg (k / \delta)$, then with probability at least $1 - \delta$ the data structure will find each of the k-heavy-hitters.

It's okay if the reported set also contains some elements that aren't k-heavy-hitters; you don't need to worry about this.