Problem Set 0: Concept Refresher

This first problem set of the quarter is designed as a refresher of the concepts that you've seen in the prerequisite courses leading into CS166 (namely, CS103, CS107, CS109, and CS161). We hope that you find these problems interesting in their own right. By the time you're done, you should be in algorithmic and coding shape for the rest of the quarter!

Before starting this problem set, we recommend reading over

- Handout #02, "Mathematical Terms and Identities," for a recap of some concepts and equations you've likely seen in the past;
- Handout #03, "Problem Set Policies," for more information about our submission and collaboration policies; and
- Handout #04, "Computer Science and the Stanford Honor Code," for details about the Honor Code and how it applies in CS166.

As always, feel free to get in touch with us if you have any questions. We're happy to help out.

Due Tuesday, April 10th at 2:30PM.

Section One: Mathematical Prerequisites

Problem One: Fibonacci Fun! (3 Points)

The Fibonacci numbers are a famous sequence defined as

$$F_0 = 0 \qquad F_1 = 1 \qquad F_{n+2} = F_n + F_{n+1}.$$

For example, the first few terms of the Fibonacci sequence are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

The Fibonacci numbers show up in surprising places in the analysis of algorithms and data structures.

One nice property of the Fibonacci numbers is that $F_n = \Theta(\varphi^n)$, where φ is the ***golden ratio***. The golden ratio is $\varphi = \frac{1+\sqrt{5}}{2}$ and is the positive root of the equation $x^2 = 1 + x$.

i. Using the formal definition of big-O notation, prove that $F_n = O(\varphi^n)$. To do so, find explicit choices of the constants c and n_0 for the definition of big-O notation, then use induction to prove that those choices are correct.

ii. Using the formal definition of big-Ω notation, prove that $F_n = \Omega(\varphi^n)$. As before, find explicit choices of c and n_0 needed by the definition of big-Ω notation, then use induction to prove that those choices are correct.

Problem Two: Probability and Concentration Inequalities (4 Points)

Let $X_1, X_2, \ldots, X_n$ be a collection of n nonnegative random variables such that $E[X_i] = 1$ for each variable X_i. (Note that these random variables might not all be independent of one another.)

i. Prove that $\Pr\left[\sum_{i=1}^{n} X_i \geq 2n\right] \leq \frac{1}{2}$. You may want to use Markov's inequality.

Now, suppose you learn that $\text{Var}[X_i] = 1$ for each variable X_i and these variables are all pairwise independent (that is, X_i and X_j are independent for any $i \neq j$).

ii. Prove that $\Pr\left[\sum_{i=1}^{n} X_i \geq 2n\right] \leq \frac{1}{n}$. You may want to use Chebyshev's inequality.

What happens, though, if the conditions from part (ii) of this problem are weakened so that the variables are not necessarily pairwise independent?

iii. Pick a natural number n and define a collection of random variables $X_1, X_2, \ldots, X_n$ such that

- each X_i is nonnegative,
- $E[X_i] = 1$ for each variable X_i,
- $\text{Var}[X_i] = 1$ for each variable X_i, but
- $\Pr\left[\sum_{i=1}^{n} X_i \geq 2n\right] > \frac{1}{n}$.

This shows that the pairwise independence requirement is essential for part (ii) of this problem to work. Once you've done this, go back to your proof from part (ii) and make sure you can point out the specific spot where the math breaks down once you've removed that requirement.

Section Two: Algorithmic Prerequisites

Problem Three: Binary Search Trees (4 Points)

Binary search trees are one of the most versatile and flexible data structures in existence and we'll explore their many properties over the course of the quarter. In the course of doing so will get to see some really, really cool ideas from Theoryland. Before we do that, though, we want to make sure everyone's had a chance to refresh some of the core concepts from BSTs.

To complete this part of the assignment, download the starter files from

```
/usr/class/cs166/assignments/ps0
```

and implement the functions in the `bst.c` source file. Test your implementation extensively. You may want to use our provided test harness as a starting point. Feel free to add your own tests on top of ours.

To receive full credit on this part of the assignment, your code must compile with no warnings (e.g. compiled with `-Wall -Werror` turned on) and should run cleanly under `valgrind` (that is, you should have no memory errors or memory leaks). We will test your code on the `myth` machines, so we recommend you test there before submitting.

i. Implement a function

```
void insert_into(struct Node** root, int value);
```

that inserts the specified value into the specified BST if it doesn't already exist. Your algorithm should run in time $O(h)$, where h is the height of the tree. (You can assume that `root` is not `NULL`, though `*root` can be `NULL` when the BST you're inserting into is empty.) You do not – and, in fact, should not – make any attempt to balance the tree.

ii. Implement a function

```
void free_tree(struct Node* root);
```

that deallocates all memory associated with the specified tree. Your function should run in time $\Theta(n)$, where n is the number of nodes in the tree.

iii. Implement a function

```
size_t size_of(const struct Node* root);
```

that returns the number of nodes in the specified tree. Your function should run in time $\Theta(n)$, where n is the number of nodes in the tree.

iv. Implement a function

```
int* contents_of(const struct Node* root);
```

that returns a pointer to a dynamically-allocated array containing the elements of that BST in sorted order. Your function should run in time $\Theta(n)$, where n is the number of nodes in the tree.

v. Implement a function

```
const struct Node* second_min_in(const struct Node* root);
```

that returns a pointer to the second-smallest element of the given BST, or `NULL` if the tree doesn't have at least two elements. Your function should run in time $O(h)$, where h is the height of the tree.

Problem Four: Event Planning (4 Points)

You're trying to figure out what Fun and Exciting Things you'd like to do over the weekend. You download a list of all the local events going on in your area. Each event is tagged with its location, which you can imagine is a point in the 2D plane. (We'll pretend that the world is flat, at least in a small neighborhood around your location. Thanks, multivariable calculus.) You also have your own (x, y) location.

Design a algorithm that, given some number k, returns a list of the k events that are closest to you. Your algorithm should run in time $O(n)$, where n is the number of nearby events (notice that this runtime bound is independent of k.) Then prove your algorithm is correct and meets the required time bounds.

Some specific details and edge cases to watch for:

- Your algorithm can produce the list of the k nearest events in whatever order you'd like.
- If there are multiple events tied for the same distance, you can break ties arbitrarily.
- By "distance," we mean Euclidean distance. We're already assuming the world is flat, so while we're at it seems pretty reasonable to also ignore things like roads and speed limits. ☺

A note on this problem, and other problems going forward: when measuring runtime in the context of algorithms and data structures, it's important to distinguish between ***deterministic*** and ***randomized*** algorithms. There's a lot of research into how to take *randomized* algorithms with a nice *expected* runtime and convert them into *deterministic* algorithms with a nice *worst-case* runtime. Since this problem set is designed as a warm-up, we'll accept either a deterministic algorithm with a worst-case runtime of $O(n)$ or a randomized algorithm with an expected runtime of $O(n)$, though in the future we'll tend to be a bit stricter about avoiding randomness.

As a hint, think about using a fast selection algorithm.