Individual Assessment 0: Concept Refresher

This first individual assessment is designed as a refresher of the concepts that you’ve seen in the pre-requisite courses leading into CS166 (namely, CS103, CS107, CS109, and CS161). We hope that you find these problems interesting in their own right. Once you’ve finished, you should be in tip-top coding and theory shape and ready to take on the new concepts from this quarter.

Before starting, 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, “Assignment Policies,” for more information about our submission and collaboration policies; and

This is an individual assessment, and, as the name suggests, must be completed individually. Specifically, you’re not allowed to work with a partner, and you should not discuss these problems with other students in CS166. However, the course staff are happy to answer clarifying questions on Piazza (if you do, please post the question privately) or in our office hours.

Due Tuesday, April 14th at 2:30PM Pacific time.
Section One: Mathematical Prerequisites

Problem One: Fibonacci Fun!

The Fibonacci numbers are a famous sequence defined as

\[ F_0 = 0 \quad F_1 = 1 \quad F_{n+2} = F_n + F_{n+1}. \]

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

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

There’s a close connection between the Fibonacci numbers and the quantity \( \phi = \frac{1 + \sqrt{5}}{2} \), the golden ratio. In case you’re wondering where that number comes from, the golden ratio is the positive root of the quadratic equation \( x^2 = 1 + x \).

We’d like you to prove some results about the Fibonacci numbers. In what follows, please do not use any properties of Fibonacci numbers other than what’s given in the definition above. The purpose of this problem is to make sure you’re comfortable reasoning about terms from first principles.

i. Using the formal definition of the Fibonacci sequence and big-O notation, prove that \( F_n = O(\phi^n) \).

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.

Our proofwriting style expectations are along the lines of what you’d see in CS161. Write in complete sentences rather than bullet points, use mathematical notation when appropriate but not as a stand-in for plain English, etc. Remember that an actual person is going to be reading your proof, so be nice to them by writing a lucid, clear argument that respects the intelligence of the reader but doesn’t ask them to do the heavy lifting for you.

ii. Along the lines of part (i) of this problem, using the formal definition of big-Ω notation, prove that \( F_n = \Omega(\phi^n) \).

You’ve just proved that \( F_n = \Theta(\phi^n) \), which is not immediately obvious! Fibonacci numbers show up in lots of algorithms and data structures, and what you’ve just proved will definitely make an appearance later this quarter.
Problem Two: Probability and Concentration Inequalities

The analysis of randomized data structures sometimes involves working with sums of random variables. Our goal will often be to get a tight bound on those sums, usually to show that some runtime is likely to be low or that some estimate is likely to be good. If we only know two pieces of information about those random variables (what their expected values are and that they’re nonnegative), we can get some information about how their sums behave.

i. 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 be independent of one another.) Prove that $\Pr[\sum_{i=1}^{n} X_i \geq 2n] \leq \frac{1}{2}$. You may want to use Markov’s inequality.

Sometimes you’ll find that the sort of bound you get from an analysis like part (i) isn’t strong enough to prove what you need to prove. In those cases, you might want to start looking more at the spread of each individual random variable. If, for example, you know the variances of those variables are small, you might be able to get a tighter bound.

ii. Let $X_1, X_2, \ldots, X_n$ be a collection of $n$ nonnegative random variables. As in part (i), you know that $E[X_i] = 1$ for each variable $X_i$. But now suppose you know two other facts. First, you know that each variable has unit variance (the fancy way of saying $\text{Var}[X_i] = 1$ for each variable $X_i$). Second, while you don’t know for certain whether these variables are independent of one another, you know that they’re pairwise uncorrelated. That is, you know that $X_i$ and $X_j$ are uncorrelated random variables for any $i \neq j$. Under these assumptions, prove that $\Pr[\sum_{i=1}^{n} X_i \geq 2n] \leq \frac{1}{n}$. You may want to use Chebyshev’s inequality.

The analysis in part (ii) only works if the variables are pairwise uncorrelated.

iii. Pick a natural number $n > 0$ 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
- $\Pr[\sum_{i=1}^{n} X_i \geq 2n] > \frac{1}{n}$.

Then, prove that your random variables have these properties.

As you saw in this problem, learning more about the distribution of random variables makes it easier to provide tighter bounds on their sums, and correlations across those variables makes it harder. This is a good intuition to have for later in the quarter, where we’ll be discussing how different assumptions on hash functions lead to different analyses of data structures.
Section Two: Algorithmic Prerequisites

Problem Three: Binary Search Trees

Binary search trees are versatile and flexible data structures, and we’ll explore their many properties over the quarter. In the course of doing so we’ll 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/a0
```

and implement the functions in the bst.c source file. (That folder just houses the raw files; there’s no git repository there, so use `cp -r` rather than `git clone` to copy the files.) 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 credit on this part of the assignment, your code should compile with no warnings (e.g. compiled with `-Wall -Werror`) and should run cleanly under `valgrind` (that is, you should have no memory errors or memory leaks). We’ll run your code on the `myth` machines, so we recommend you test there before submitting.

i. Implement a function

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

that inserts the specified value into the given 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 should not attempt to balance the tree.

ii. Implement a function

```c
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

```c
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

```c
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

```c
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 tree height.
Problem Four: Finding Cities

What city are you closest to right now? What are the ten closest cities to you? The fifty closest cities? Even if you know your area pretty well, you might not be able to answer these questions purely off the top of your head.

Let’s imagine that you have a list of all the cities in your area. Each city 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 a list of \(n\) cities and some number \(k\), returns a list of the \(k\) cities that are closest to you, sorted by increasing order of distance. Your algorithm should run in time \(O(n + k \log k)\). Then prove your algorithm is correct and meets the required time bounds.

Some specific details and edge cases to watch for:

- You can assume, for simplicity, that no two cities are at the same distance from you.
- 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. 😃

As a hint, think about the algorithms you studied in CS161 and see if any of them would make for good subroutines.

To make things easier for the grader, we recommend doing the following when writing up your solution:

1. Start off by giving a quick, two-sentence, high-level description of your approach. This makes it easier for the grader to contextualize what it is that you’re trying to do.

2. Next, go into more detail. Describe how your algorithm works, one step at a time. Please don’t write actual code unless it would be nearly impossible to describe the algorithm in plain English. (Trust us – from experience, reading code is often much harder than reading prose!)

3. Write a quick correctness proof. Tell us what, specifically, you’re going to prove, then go prove it. Our proof expectations are similar to those for CS161 – write in complete sentences, use mathematical notation when appropriate but not as a stand-in for plain English, etc.

4. Write a runtime analysis. Go at whatever level of detail seems most appropriate.

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 is designed as a warm-up, we’ll accept either a deterministic algorithm with a worst-case runtime of \(O(n + k \log k)\) or a randomized algorithm with an expected runtime of \(O(n + k \log k)\), though in the future we’ll tend to be a bit stricter about avoiding randomness.