Finance & Exam

PHYLLIS ZHANG

##

Finance Hints

- Recall that when you execute `db.execute("SQL QUERY")`, a *list of dictionaries* is returned. If you would like a specific dictionary from the list, you can index using `[i]` where `i` is the index of the list that you'd like. When you index with `[i]`, you'll then get a dictionary. Recall that to access the value of a particular key in a dictionary, you can simply do `["key"]`.
    - As an example...

      ```
      pokemon = db.execute("SELECT name, type FROM pokemon WHERE id = ?", 258)
      ```

        - This gives me a list of dictionaries, containing all pokemon with `id = 258`. There is only one pokemon with `id = 258` though, and it's mudkip!
        - Then, pokemon would be a list of one dictionary, namely:

          ```
          pokemon = [
              {
                  "name": "Mudkip",
                  "type": "Water"
              }
          ]
          ```

        - Assume we'd like to get the name of this particular pokemon. We would then do

          ```
          name_of_pokemon = pokemon[0]["name"]
          ```

        - `pokemon[0]` gives us the first dictionary, which is

          ```
          pokemon[0] = {
                  "name": "Mudkip",
                  "type": "Water"
          }
          ```

        - Then, we'd like the value for the key "name", so we do `pokemon[0]["name"]`

- If you'd like to *update your table* using some math, you can write

  ```
  db.execute("UPDATE table_name SET item = item - ? WHERE condition",
  decrement_value)
  ```

    - This decreases the value of `item` by `decrement_value`.
    - You might find this useful in `buy()` and in `sell()`.

- When you create another table, you may consider storing the user's transactions, so you can easily display their transactions in the *history* tab. Take a look at how history should look to determine which columns should go into this new table.
    - Double click `finance.db` to open up `phpliteadmin`. Create your table here!!! Don't type it all into the terminal :(
- Inside `sell.html`, you might find yourself wanting to use a `form-group` that allows for `option` tags. In this case, you should hardcode the values for the `option` tags by just writing `<option value = "{{ stuff }}">`. Note that when `request.form.get` looks for the name (which will be the name of your `form-group`), it will look for that particular `value` attribute of the selected `option`.
- To ensure the user is logged in and you are accessing the correct user, make sure that when you query the database, you have a condition where you say `WHERE user_id = ?` and the `?` placeholder is then substituted by `session['user_id']`.
- The *lookup* function returns a *dictionary*. For example, suppose I would like to look up the stock information of "CGC".
    - ```
      info = lookup("CGC")
      name = info["name"]
      price = info["price"]
      ```
    - Note that in this piece of code, the lookup function returned a dictionary with keys `name` and `price`. I was then able to access the name and price of my CGC stock via `info["name"]` and `info["price"]`.
    - ```
      print(name) # would output "Canopy Growth Corporation"
      print(price) # would output the stock price of CGC
      ```
- You should additionally make sure that the stock the user types in is valid. If it's invalid, then `info = lookup("INVALID_SYMBOL")` will be some variant of `None`. Thus, you can write
    - ```
      if not info:
          # this means that info is some variant of `None`
      ```

Debugging Tips

- If you're getting a `500: Internal Server` error, go back to the terminal and check where your error is.
- If you're getting a "cannot multiply this type with another type" or some variant, be careful with how you're indexing your lists of dictionaries from your SQL query. If you're unsure, think through what each of the types of your variables are. This is particularly applicable in `buy()` and `sell()`.
- Check the URL of your page. If you see `wownumbersletters:8080/`, this means you're at the index page. Use this to make sure you're being redirected to the correct places!
      ```

- If you've just completed register, and you see an apology saying `TODO`, check if your URL is `wownumbersletters:8080/`. If it is, it means your `register` page is working correctly and you've been redirected to the index page!

CS50 Test

Boolean Logic

- Remember how we talked about `AND` and `OR`? Well, it turns out there are a lot more boolean functions, like `XOR` and `NAND`. And then you can make more and more of these boolean functions via only these `AND` and `OR` functions. In fact, you can create any boolean function in the world with just `NAND` functions, where `NAND` stands for `NOT AND`! The best way to approach these problems is to try smaller examples and then create a chart. Here are a few tables to get started:

X	Y	AND(X, Y)
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	OR(X, Y)
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	XOR(X, Y)
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	NAND(X, Y)
0	0	1
0	1	1
1	0	1
1	1	0

- Definitely think through these problems carefully, if they exist.

Adjacency Matrices, "Directed Graphs," and Regular Expressions

- These questions are actually *Deterministic Finite Automata* questions in disguise (aka CS121 material), but they're easy to get! Take a look at last year's regular expression question for inspiration.
- Adjacency Matrices show you where edges are in a graph. For example, if I had the table, or matrix, below...

	A	**B**	**C**
A	0	1	1
B	0	0	1
C	0	0	0

  - Make sure you understand the rules on the question. In this case, if I specify that the rows on the left have edges directed towards the columns up top, I would then have a graph where there is an edge from A to B, from A to C, and from B to C. Note that this is due to the placement of the 1s in the matrix.
  - Adjacency Matrices can also be represented as Adjacency Lists. An identical adjacency list would just be:

```
A: [B, C]
B: [C]
C: []
```

  - Every node has a list of vertices that it points to. This should remind you of... nodes pointing to linked lists!

- Given a regular expression problem, you should definitely check your regex. There are lots of websites to check your regex, but here's one.
- Given a directed graph, please draw it out! You can turn this into a regular expression or adjacency matrix far more easily if you draw it out.

Prefix Freeness

- Prefix Free encodings are very simple! Essentially, if you encode a DNA sequence `A` as `1`, `C` as `0`, `G` as `10`, and T as `11`, given `11`, you don't know if your sequence is `AA` or `T`, meaning that this is *not* prefix free.
- For an easy fix to create a prefix free solution, simply make every letter fixed width! If every encoding was exactly `2` bits, then we can't be confused – we just read `2` bits at a time!

Big O Notation

- Run through the algorithm yourself. How many for loops are there? Are they nested?
  - If the for loops are nested, you multiply them.

    - If the for loops are not nested, you add. Recall that `O(n + n) = O(2n) = O(n)`.
- Recall that the best case is shown with an omega and not an O.
- For common functions in data structures, here's a Big O Cheatsheet to help you out!
- If you're trying to prove that something is `O(n^2)` for example, you just need to prove that the time it takes is less than `n^2`. If you find that it's linear, then you would say `O(n)`, right? But remember that something that is `O(n)` is also `O(n^2)` since we're talking about the worst case. This only works in the *worst case* situation.
- We spent a lot of time analyzing certain cases. For example, if a binary search tree is badly, badly balanced, we'll get a linked list. If a hash table has a very, very poor hash function, leading to a ton of collisions, we'll just get a few very long linked lists. Consider these cases!
    - We can fix a badly, badly balanced binary search tree by balancing it! If this shows up, you may want to look at `AVL trees` or `Red-Black Trees`, which are trees with self-balancing algorithms incorporated.

Coding Problems

- This exam is open internet, so odds are, you'll be able to find snippets of code on the internet. Cite your sources! If you can't figure out a coding problem, you might be able to find an algorithm online to help you. Always start with small cases and see if you can identify a simple pattern that follows. Run the program in your IDE to check that it works as intended, and make sure to *try EDGE CASES*.

TCP/IP

- Check out my notes on TCP/IP that I uploaded to my Github!

Compiling, Assembly Code, Hexadecimal

- Memory is a large array of 8-bit wide bytes. Thus, we have random access, the ability to jump to different addresses.
- Each location in memory has an address.
- A 32-bit system is able to process addresses up to 32 bits in length, and a 64-bit system is able to process addresses up to 64 bits in length.
- When referring to memory addresses, we use *hexadecimal* notation, which is the base 16 system. While a binary 32-bit address might look something like this: `00101001 11010110 00101110 01010111`, a hexadecimal 32-bit address would instead look something like this: `0x29D62E57`.
    - Four binary digits can be expressed with a single hex digit.
    - The `0x` is included in the front to denote that this value is in hexadecimal.
- If you see these strange fellas: `eax`, `ecx`, ..., and `ebp`, they are *registers*, very small units close to the memory that store data.

- The remainder of the discussion on registers and on assembly code is under my week 1/2 Github notes.

Heaps, PriorityQueues, and other Data Structures

- Your best bet in this case is to watch some youtube videos and draw along to truly understand how the data structure is able to retain data. I particularly like this `min heap` visualization, where you can insert numbers and see how they're added and deleted. Pay special attention! Note that this is an example of a *min* heap as well; there's something else for *max* heaps!

Algorithms: Depth First Search, Breadth First Search, Two Pointers, Recursion, Sliding Window Etc.

- Most likely you won't know that the problems are these types of problems. If you're at a complete loss at how to approach an algorithm though, here's a brief overview of each of these, and you can see if it's applicable to what you're trying to do.
- *DFS and BFS:* these allow you to search in a graph for certain nodes by implementing stacks and queues (recall that stacks and queues are abstract data types and can thus be implemented via linked lists or even just arrays!)
- *Two Pointers:* Sometimes, it might be helpful to have a left pointer and a right pointer. The left pointer starts at the beginning of, let's say, an array, and the right pointer starts at the end of the array. You want to somehow meet in the middle, satisfying some sort of constraint. Then, you can move your left and right pointers however you wish!
- *Recursion:* If you need to keep calculating smaller cases for a calculation, use recursion. You can even store your intermediary variables in an array, and we call that *dynamic programming*. If your answer for `n` depends on `n-1`, you can either keep calling recursively or create an array, and `arr[n]` can be calculated from `arr[n-1]`. Look up *dynamic programming* if interested!
- *Sliding Window:* You have a bunch of trees in front of you, one after the other. Each has `x` ducks on it, for different `x`s. Bert and Ernie are evil and have told you that you must get `10` ducks on `3` consecutive trees, and no more than `10` or else all the ducks die. You use a sliding window technique where you just keep track of the sum of the first `3` trees, then you move forward one (add the fourth tree, subtract the first tree) to check the sum of the 2nd through 4th trees and you keep going until the sum is `10`. You never get `10?` Welp :(. Promise the sliding window technique is applicable to more than this dumb example, haha.

Closing

- Start early and budget enough time to finish. Good luck! The test requires you to apply your knowledge in interesting ways – you got this :). And remember, if you don't know, at least write *something* down!