INSTRUCTIONS

Your assignment is to answer the questions on the following pages. There is a total of 36 questions. To help you not to miss a question, I have highlighted all questions in yellow.

Although there is a lot of text, most of it is background information so that you understand the question. **Please read all the information (and the video) associated with each question before answering the question.**

Write your answers in a Word document like this:

PART A: 1 [insert answer]

The above example is how you would provide your answer for the first question in Part A. Do so for each answer in every part.

Your only deliverable is that Word document.

Deadline: 24 hours

PART A

Recall that C programs are first compiled into a lower-level language called "assembly" before that assembly is assembled into machine code that a computer can execute. There are a number of different types of assembly languages, but they generally share similar properties: assembly languages have a limited set of instructions for performing basic operations like putting data in a variable (otherwise known in this context as a "register"), moving data from one location in memory to another, etc.

Consider a simplified assembly language wherein there are four registers (i.e., locations to store values) called `r1`, `r2`, `r3`, and `r4`. This assembly language supports the following instructions, wherein `R`, `Rx`, `Ry`, and `Rz` represent (any of those) registers, `V` represents a literal value (an integer or a string), and `L` represents a line number:

- `PRINT R` prints the value in register `R`.
- `INPUT R` prompts the user for input and stores it in register `R`.
    - You may assume that if the input looks like an integer (i.e., it consists of only digits), it will be stored as an integer; otherwise, it will be stored as a string.
- `SET R V` stores the value `V` in register `R`.
    - For example, `SET r1 50` would store the value `50` in register `r1`.
- `ADD Rz Rx Ry` adds the value stored in register `Rx` to the value stored in register `Ry` and stores the result in register `Rz`.
- `JUMPEQ Rx Ry L` checks if the values stored at registers `Rx` and `Ry` are equal to one another. If so, the program jumps to line `L`. Otherwise, the program continues to the next instruction.
- `JUMPLT Rx Ry L` checks if the values stored at register `Rx` is less than the value stored at register `Ry`. If so, then the program jumps to line `L`. Otherwise, the program continues to the next instruction.
- `EXIT` exits the program.

Every line of code in this assembly language consists of a line number followed by a single instruction. No parentheses, curly braces, semicolons, or any other syntax other than the above instructions!

For example, here is a program that prompts the user for two numbers and prints whether they are equal or not:

```
 1  SET r1 "x: "
 2  PRINT r1
 3  INPUT r2
 4  SET r1 "y: "
 5  PRINT r1
 6  INPUT r3
 7  JUMPEQ r2 r3 11
 8  SET r1 "x is not equal to y"
 9  PRINT r1
10  EXIT
11  SET r1 "x is equal to y"
12  PRINT r1
```

```
13  EXIT
```

1. In English, explain how the program above works, making clear why it is correct, as by explaining the role of each line, from `1` through `13`.

2. Rewrite the program above in such a way that, instead of just printing out `x is not equal to y` when the two numbers are not equal, it instead prints either `x is less than y` or `x is greater than y`, depending on which number is greater. The program should still print `x is equal to y` if the two numbers are equal.

3. Write a program in this assembly language that “coughs” (i.e., prints `cough`) some number of times. Your program should first prompt the user for a number and then print `cough` exactly that many times. You may assume the user will input a non-negative number.

The assembly language you just used to write these programs is a simplified version of the assembly language your computer might use when compiling a C program.

When `clang` compiles your C program, it first compiles your C program into an assembly language called “x86-64” and then assembles assembly into machine code. It turns out we can actually stop `clang` midway through that process so as to take a look at the assembly code corresponding to our program.

Copy the program below into a file called `compare.c`

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int x = get_int("x: ");
    int y = get_int("y: ");

    if (x < y)
    {
        printf("x is less than y\n");
    }
    else
    {
        printf("x is not less than y\n");
    }
}
```

In your terminal, run `clang -S compare.c`. The `-S` flag tells `clang` to output the assembly code for the program. After you run the command, you should see a file called `compare.s` containing the assembly. Open that file and take a look!

Odds are it looks pretty complicated! No need to understand all the details but notice that most lines contain some instruction followed by one or more arguments for that instruction. The `movl` instruction, for example, moves data from one location to another.

4. Unlike our own assembly language above, x86-64 has an instruction for calling a function from inside of a program. Based on the assembly code in `compare.s`, what is the name of the instruction for calling a function? How do you know?

5. Based on the assembly code in `compare.s`, what is the name of the x86-64 instruction via which the program decides what to print? And how does that instruction decide what to print?

PART B

Watch the video below on how bitcoin, a "cryptocurrency," works, up through 00:21:54.

https://youtu.be/bBC-nXj3Ng4

1. What, in your own words, is a blockchain? How does bitcoin use a blockchain?

2. What, in your own words, is a cryptographic hash function?

3. Consider the hash function, below, which calls `ord`. Assume that `s` is a string of length 1 or greater.

   ```
   def hash(s):
        return ord(s[0]) % 50
   ```

   Why is this implementation of `hash` unsuitable as a cryptographic hash function?

4. Although a block in an actual blockchain represents multiple transactions, assume for simplicity that a block represents only one and that each block thus has, among other fields:

   - a unique identifier for the transaction itself,
   - a unique identifier for the transaction's sender,
   - a unique identifier for the transaction's recipient,
   - the transaction's amount,
   - a digital signature from the sender, and
   - a proof of work.

   Suppose we have a type `blockchain` defined as

   ```
   typedef block *blockchain;
   ```

where a `blockchain` is a pointer to a value of type `block`.

Implement a type called `block` in a manner consistent with the video and specifications above. Assume that `digest` is a type with which you can represent SHA256 digests.

PART C

Recall that ASCII is just one way to represent characters.

1. How many total characters can be represented in ASCII, if each character is represented using 7 bits?

If we want to represent more characters than ASCII allows, we can use Unicode, which uses more bits than ASCII to represent some characters. One implementation of Unicode, UTF-8, uses "variable-width encoding" to represent characters: characters can be represented by either one, two, three, or four bytes.

Read up on UTF-8 at fileformat.info/info/unicode/utf8.htm.

2. When reading, as via `fread`, a text file encoded as UTF-8, how can you determine how many bytes a character will take?

3. If you're reading a file encoded as UTF-8, and you read a byte, how can you determine if that byte is a continuation of an existing character, rather than the beginning of a new character?

PART D

Recall that a "regular expression," otherwise known as a "regex," is a pattern designed to match some string. For instance, a regex like `y` would match any string containing a `y`, a regex like `yes` would match any string containing `yes`, a regex like `^yes` would match any string starting with `yes`, and a regex like `^yes$` would match only the string `yes`.

Read up on regular expressions via Python's Regular Expression HOWTO, focusing on these sections specifically:

- Introduction
- Simple Patterns
- More Pattern Power
- Modifying Strings

1. Describe, in a sentence, the set of strings that a regex like `^yes+$` would match.

2. Describe, in a sentence, the set of strings that a regex like `^yes\s*$` would match.

3. Consider the Python program below.

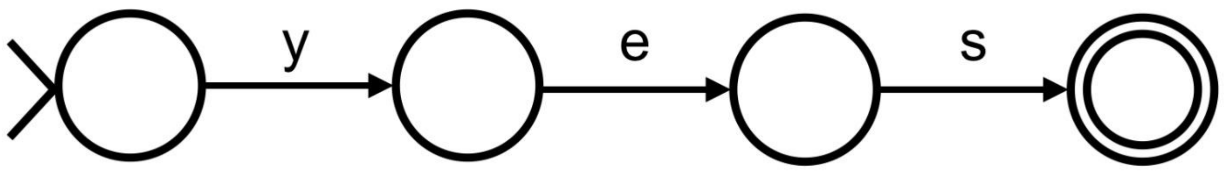
```python
import re

words = input("Say something!\n")
p = re.compile("my name is (.*)", re.IGNORECASE)
matches = p.search(words)
if matches:
    print(f"Hey, {matches[1]}.")
else:
    print("Hey, you.")
```

Given input like `My name is Earl`, this program politely outputs `Hey, Earl`. Given input like `My name is Earl Hickey`, though, this program outputs `Hey, Earl Hickey`, which feels a bit formal. Propose how to modify the argument to `re.compile` in such a way (without hardcoding `Earl`) that this program would instead output `Hey, Earl` in both cases.

DFAs

It turns out that regular expressions can be implemented in software with “deterministic finite automata” (DFAs), otherwise known as “state machines.” DFAs can even be implemented with diagrams as well, wherein each node (i.e., circle) represents a “state” and each edge (i.e., line) represents a “transition.” For instance, below is a DFA that implements a regex like `^yes$`:



A DFA is a machine (or algorithm, really) in the sense that it takes some input and produces some output. Its input is a string (like `y` or `yes`). Its output is a decision: to “accept” or to “reject” that string. A DFA accepts a string if that string matches a pattern, the very regex that the DFA implements. A DFA rejects a string if that string doesn’t match the pattern.

How does a DFA decide whether its input matches a pattern? Well, when a DFA is “turned on,” so to speak, it’s considered to be in its “start” state, as is indicated by a circle with a **>**. It then looks at the first character of its input. If there is an edge labeled with that same character from the start state to some other state, the DFA transitions from its start state to the other, thereby consuming the character. The DFA then looks at the second character of its output. If there is an edge labeled with that same character from its current state to another, it transitions again, consuming the character. If, upon consuming every character of its input, the DFA finds itself in an “accept” state, as is indicated by a circle within a circle, it must be the case that the string matched the pattern, and so the DFA accepts it (and turns off). If, though, at any point, the DFA finds itself in a “reject” state, as is indicated with a circle without a circle within it, having consumed all of its input or unable to transition from one state to another because there’s no edge labeled the same as the character currently being read, it must be the case that the input didn’t match the pattern, and so the DFA rejects it (and turns off).

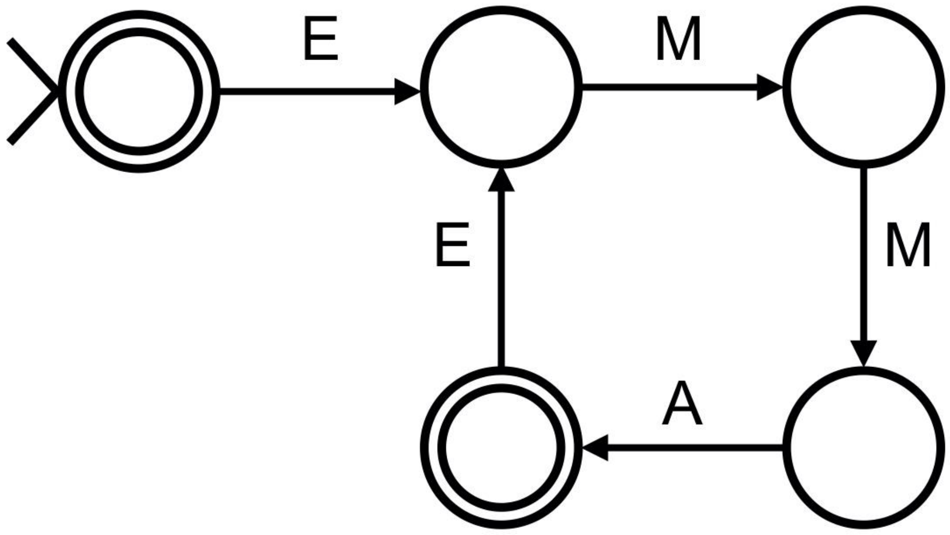
Consider, for instance, how the DFA above might process an input like `y`, which doesn’t match `^yes$`.

i. The DFA starts in its start state.
ii. The DFA reads the first (and only) character of its input, `y`. Because there is an edge labeled `y`, the DFA transitions from its start state to its second state, thereby consuming that `y`.
iii. Because the DFA has consumed all of its input but is in a reject state, the DFA rejects `y`, just as it should, because `y` doesn’t match `^yes$`.

Now consider how the DFA above might process an input like `yes`, which does match `^yes$`:

i. The DFA starts in its start state.
ii. The DFA reads the first character of its input, `y`. Because there is an edge labeled `y`, the DFA transitions from its start state to its second state, thereby consuming that `y`.
iii. The DFA reads the second character of its input, `e`. Because there is an edge labeled `e`, the DFA transitions from its second state to its third state, thereby consuming that `e`.
iv. The DFA reads the third (and final) character of its input, `s`. Because there is an edge labeled `s`, the DFA transitions from its third state to its fourth state, thereby consuming that `s`.
v. Because the DFA has consumed all its input and is in an accept state, the DFA accepts `yes`, just as it should, because `yes` matches `^yes$`.

Consider the DFA below.



4. What regex does this DFA implement?

5. What are three strings that this DFA would accept?

6. What is one string that this DFA would not accept?

PART E

Some social networks, including Twitter and Instagram, allow one user (a “follower”) to follow another user (a “followee”). Let’s consider how such networks might represent data about their users. A social network might, for instance, have a SQL database with tables like the below.

```
CREATE TABLE users (
    id INTEGER,
    username TEXT UNIQUE,
    name TEXT,
    PRIMARY KEY(id)
);

CREATE TABLE followers (
    follower_id INTEGER,
    followee_id INTEGER,
    FOREIGN KEY (follower_id) REFERENCES users(id),
    FOREIGN KEY (followee_id) REFERENCES users(id)
);
```

Assume that Ileana is registered for this social network with a username of `ileana`, that Reese is registered with a username of `reese`, and that Max is registered with a username of `max`.

1. Suppose that Max starts following Ileana. What (single) SQL statement should be executed?

2. With what (single) SQL query could you select the usernames of every user that follows Reese and whom Reese also follows back?

3. Suppose that Reese is looking for additional users to follow. One suggestion that a social network might provide is to suggest users who are “two degrees of separation” away: users who are followed by users whom Reese already follows. With what (single) SQL query could you select the usernames of users who are followed by users whom Reese follows?

Social networks like Twitter and Instagram support asymmetric relationships: Alice might follow Bob, but that does not mean that Bob also follows Alice. Other social networks, like Facebook, support symmetric relationships as well: in order for Alice and Bob to be “friends,” one of them must send a “friend request” to the other, which the other must then accept.

4. With what SQL statement could you create a `friendships` table that allows for the representation of friendships and friend requests? Assume that `users` exists as above.

5. Suppose that Max sends a friend request to Ileana. What (single) SQL statement should be executed?

6. Suppose that Ileana accepts Max's friend request. What (single) single SQL query should be executed?

7. Suppose that Reese deletes his account on this social network. What SQL statements should be executed in order to remove all traces of Reese, his friendships, and his followees?

PART F

Recall that in a previous assignment you wrote code to analyze sequences of DNA. Those sequences of DNA were stored in text files that might resemble the below.

```
AAAAAAATTTTTAAAGGGGCCCCCCAAACCCC
```

Some of those text files, though, were quite large. And we gave you quite a few such files!

It turns out we could have compressed those files by representing those sequences differently. Rather than store one character per nucleotide (`A`, `C`, `G`, or `T`), we could have stored one character (plus a number) per "run" of nucleotides, whereby a run is a sequence of identical values. For instance,

- `A` could instead be encoded as `A1`,
- `TTT` could instead be encoded as `T3`, and
- `CCCCCCCCCC` could instead be encoded as `C10`.

This type of compression is known as "run-length encoding," wherein runs of values are encoded by storing the number of repetitions.

Our original sequence of 32 nucleotides, `AAAAAAATTTTTAAAGGGGCCCCCCAAACCCC`, could thus be encoded `A7T5A3G4C6A3C4` with only 14 characters, thereby decreasing the length of our sequence (and number of bytes) by more than half!

1. Consider the DNA sequence below.

   ```
   TTTAAAACCGAAA
   ```

   How could that sequence instead be encoded using run-length encoding?

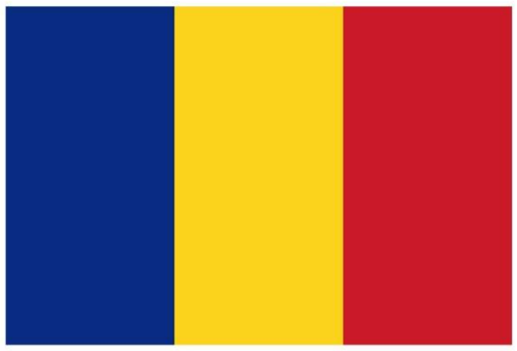
While run-length encoding will result in smaller file sizes for some DNA sequences, for other sequences this algorithm might actually increase the size of the file.

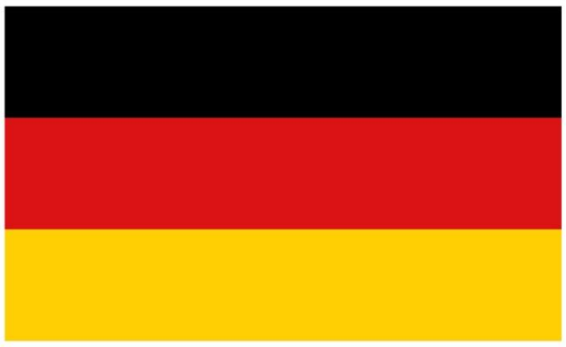
2. For what types of sequences would this algorithm likely increase the file size, rather than decrease it? Include in your answer an example of a DNA sequence for which the “compressed” version actually requires more characters than the original.

It turns out that run-length encoding can be used to compress images as well. Recall that (24-bit) BMP files can be thought of as a sequence of rows, where each row consists of pixels, and each pixel is represented using 1 byte for an amount of red, 1 byte for an amount of green, and 1 byte for an amount of blue.

Consider what might happen if, instead of storing every pixel, we were to compress images using run-length encoding, where runs of the same pixel within a row were stored as a pixel, followed by a count.

3. Imagine what might happen if we compressed BMPs of the flags of Romania and Germany, below, and compressed each using run-length encoding for each row of pixels. Assuming both flags have the same resolution (i.e., rows and columns of pixels), which image could be compressed more (i.e., take up less space once compressed)? Why?


Flag of Romania


Flag of Germany

Let’s now revisit our encoding of DNA and consider a different method altogether for storing a DNA sequence using fewer bytes. Recall that Problem Set 6’s files were stored using ASCII, where each character in the text file took up 1 byte (8 bits) of space. According to ASCII,

- `A` is represented as `01000001` (which is 65 in decimal),
- `C` is represented as `01000011` (which is 67 in decimal),
- `G` is represented as `01000111` (which is 71 in decimal), and
- `T` is represented as `01010100` (which is 84 in decimal).

An ASCII encoding is useful if our text files will many different characters: uppercase letters, lowercase letters, numbers, punctuation, and/or the like. But if the only characters we need our file to store are `A`, `C`, `G`, and `T`, then we could use fewer bits to represent each nucleotide.

Consider the following encoding instead:

- `A` is represented as `0`
- `C` is represented as `1`
- `G` is represented as `10`
- `T` is represented as `11`

Thus, the sequence `AGCC` would be represented as `01011`. Only 5 bits, instead of the original 32!

4. Using this new encoding, how would you represent the sequence `CCCAGTTA` in binary?
5. Using this new encoding, how would you represent the sequence `TGCATCCA` in binary?
6. What problem does this encoding thus have? Propose another encoding for DNA nucleotides that fixes that problem, while still using fewer bits than storing nucleodides as ASCII characters.

PART G

Consider the implementation of `sort` , below, which takes a `list` as input and returns a sorted version thereof. It calls `issorted`, also below, to determine whether those numbers are sorted. It also imports a module, `random`, and calls `random.shuffle` therein. We've included the bottommost line so that you can try out the code in your IDE, but assume that `numbers` could be any `list` of size *n*.

```
def issorted(numbers):
    if sorted(numbers) == numbers:
        return True
    else:
        return False


def sort(numbers):
    import random
    while not issorted(numbers):
        random.shuffle(numbers)
    return numbers


print(sort([6, 3, 8, 5, 2, 7, 4, 1]))
```

1. According to Python's documentation, what's the running time (on average) of `sorted` and, in turn, `issorted`?

2. Why can the running time of `issorted`, however implemented, not be in *O*(1)?

3. In the worst case, what might the running time of `sort` itself be?

Consider the re-implementation of `sort`, below, which imports a module, `itertools`, and calls `itertools.permutations` therein. As before, assume that `numbers` could be any `list` of size *n*. And assume that this implementation of `sort` calls an implementation of `issorted` in *O*(*n*).

```
def sort(numbers):
    import itertools
    for permutation in itertools.permutations(numbers):
        if issorted(permutation):
            return permutation
```

4. What's a lower bound on the running time of this implementation of `sort`?

5. What's an upper bound on the running time of this implementation of `sort`?