Question 1: Which does NOT make the Genome assembly computing problem difficult? (select ONE)
○ There may be multiple errors in the input strings.
○ The input is only a small view of the genome (100s of characters) but the output is 10,000s of characters.
○ The letters in the genome input are all 26 letters (A-Z), a vast number of possible combinations.
○ The output genome has to have all the errors removed.
○ None of the Above.

Question 2: What was wrong with early versions of the Google self-driving car at 4-way stops? (select ONE)
○ It couldn’t see the stop sign because its photo sensor was too small, so it drove right through!
○ It yielded to the driver on the left (instead of right), since that’s how they drive where the developers live!
○ It kept waiting for human drivers who would inch forward (looking for an advantage), so it’d be stuck there!
○ It had trouble stopping at the right place since the road markings were often obscured, so it would back up!
○ None of the Above.

Question 3: Prof Marti Hearst talked about what device/concept her HCI group helped invent? (select ONE)
○ Pinch and Zoom (or similar mobile app technology that lets you change the size of an image with 2 fingers)
○ Google Glass (or similar camera + computer + display eyewear technology to record your world while out)
○ Faceted Navigation (or similar web technology that lets you “prune” down what you want when searching)
○ Amazon Alexa (or similar technology that lets you interact with a computer through speech and audio)
○ None of the above

Question 4: What was one memorable moment from the Alumni Panel? (select ONE)
○ The alumnus from Amazon talked about the fun of programming drones to do aerial package delivery.
○ The alumnus from Apple talked about their new “spaceship” campus and getting lost once inside it!
○ The alumnus from Facebook talked about how fun it was watching Mark Zuckerberg testify before congress!
○ The alumnus from GradeScope talked about the fun working for a small startup located so close to Cal!
○ None of the above

Question 5: What is the Halting Problem? (select ONE)
○ It was used to prove that not all problems are decidable.
○ It was one of a family of NP-complete problems; solve one efficiently, and you can solve them all!
○ It was the problem that was recently solved (to great acclaim), proving that P=NP once and for all!
○ It was the occurrence of “forever” blocks in Snap! code that prevent programs from stopping, or “halting”.
○ None of the above

Question 6: What is 4616 divided by 128 written in Binary? (select ONE)

<table>
<thead>
<tr>
<th></th>
<th>02</th>
<th>12</th>
<th>102</th>
<th>112</th>
<th>1002</th>
<th>1012</th>
<th>1102</th>
<th>1112</th>
<th>None of these</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Question 7: We put the Fun in Functional Programming...** (8 = 2+2+2+2 pts)

SID: ____________

Consider the following two blocks and setup code:

![Diagram of code blocks](image)

**a)** We're now going to zoom in on pixels affected by calls to **Fun**; the sprite always starts in the lower left facing right, and the pen is in the center of the sprite. Your job is to shade in (completely!) all the pixels that will be colored in after calls to **Fun** with length set to 2, 4 and 8; don't worry about drawing the sprite at the end. **Clarification:** if the sprite were at (0,0) and moved 4 steps to the right, it would be at (4,0) and all pixels along the line from (0,0) through (4,0) would be shaded; 5 pixels in total.

![Grids for Fun(2), Fun(4), and Fun(8)](image)

**b)** Could **Fun** be rewritten iteratively? (select ONE)

- Yes, and in fact **any** recursive procedure can be written iteratively.
- Yes, and it's **fairly common** to be able to write recursive solutions iteratively, but not always.
- Yes, but it's **fairly rare** to be able to write recursive solutions iteratively. Most of the time you can't.
- No **because it's a command**. Had it been a reporter or a predicate, we would have been able to do it.
- No; it's **one of the rare cases** of recursive solutions that can't be written iteratively. Most of the time you can.
- No; it's **fairly common not** to be able to rewrite recursive solutions iteratively. Sometimes you can, though.
- No; you can **never rewrite recursive solutions in an iterative manner**.
Question 8: He knew something about the levels of gravitity and polarity... (12 pts = 3+2+2+2+3)
You have cards, numbered 1-N, which are placed in ANY order into a list; this is called a *shuffle*. We want to write a block to determine if a list is a valid shuffle (i.e., it has all the numbers 1-N [where N is the length of the list] in some order), but it might be buggy...

---

**a)** What is equivalent to the expression reported by `shuffle? helper`? (select ONE)
- ○ combine with or items of list of booleans
- ○ combine with and items of list of booleans
- ○ not combine with or items of list of booleans
- ○ not combine with and items of list of booleans

**b)** What does `shuffle?` report for different inputs (of length at least 2)? (select ONE per row)

<table>
<thead>
<tr>
<th>Input</th>
<th>shuffle? would report</th>
</tr>
</thead>
<tbody>
<tr>
<td>The numbers 1 through N, in reverse order</td>
<td>false ○ ○ true ○</td>
</tr>
<tr>
<td>Is a valid shuffle, but one of the numbers is replaced with -99</td>
<td>false ○ ○ true ○</td>
</tr>
<tr>
<td>A list of N ones</td>
<td>false ○ ○ true ○</td>
</tr>
</tbody>
</table>

**c)** Fix the code so it works correctly (select ALL that apply)
- □ Change false to true ○
- □ Remove the not
- □ Change #s 1-N contains to shuffle contains ○
- □ Change map over shuffle to map over #s 1-N ○
Question 9: What do tablets eat for dinner? iPad Thai... (28 = 2*9+10 pts)

Instead of a paper final exam for BJC, we decide to use tablet computers, with each of the N problems printed on separate digital “page” (screen). On each page, there’s a field for the answer and a submit button, which when clicked shows the next problem or a “You’re done!” screen if they’ve answered all the questions. All students start on page 1, and there’s no way to skip around or go back. When time is up, if the student is still working on a question, the system automatically clicks submit for them. The system records four pieces of information into a database (DB) each time submit is clicked; here’s an example that was recorded for a 2-question exam. Note that student 99999999 fell asleep during question 1, so the system submitted it for them at the end of the exam. All the times in “Seconds since the exam started” will always be in chronological order.

<table>
<thead>
<tr>
<th>Unique student ID</th>
<th>Seconds since the exam started</th>
<th>Question #</th>
<th>Auto-graded score for that question</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>60</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>11111111</td>
<td>100</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>11111111</td>
<td>110</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>12345678</td>
<td>260</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>99999999</td>
<td>600</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

a) Which of the following could be answered by analyzing only the data in DB after the exam is finished? (select ALL that apply)

☐ How many students saw the “You’re done!” screen after they finished the exam.
☐ The number of questions on the exam.
☐ The number of different students who took the exam.
☐ The highest overall score any student achieved.

b) Somehow the first column (Unique student ID) gets corrupted (lost). If we didn’t care about the actual student ID, but only that each student had a different ID that was consistent throughout the DB, when are we guaranteed to be able to recreate the first column using only the other values in DB? (select ALL that apply)

☐ When there is only one student
☐ When no student finishes the first question
☐ When exactly one student finishes the first question before time is up (i.e., clicked submit themselves)
☐ When no two students are ever on the same question at the same moment in time (except the first)

A separate grid below to put up to 6 lines of python into our code on the left AND tell us where to indent the lines. Write the code in the area on the left (as a backup). You might not need all 6 lines, and if a command is not used, leave its row blank.

```python
def SIDstoquestions(SIDs):
    D={}
    questions = []
    for SID in SIDs:
        if SID in D:
            D[SID] += 1
            if SID in questions:
                questions.append(D[SID])
        else:
            D[SID] = 1
            questions.append(D[SID])
        if SID in questions:
            questions.append(D[SID])
            questions.append(D[SIDs])
            questions.append(D)
            if SID in questions:
                questions.append(D[SIDs])
            if SID in questions:
                questions.append(D[SIDs])
            if SID in questions:
                questions.append(D[SIDs])
```

<table>
<thead>
<tr>
<th>Line number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


**Question 10: **Would you like another *pie* [1:3] ? (18 = 2*9 pts)

a) We recreated a script of playing in the interpreter, but we may have gotten some of it wrong.

For each response, indicate if it is correct, and if it’s not, what the right answer should be.

```python
>>> S = "1234"
>>> S[1:3]
"123" ✗ Correct! ✗"23" ✗"12" ✗ Error ✗ Infinite Loop

>>> S+S
"2468" ✗ Correct! ✗2468 ✗"12341234" ✗ Error ✗ Infinite Loop

>>> "3" in S
True ✗ Correct! ✗False ✗ Error ✗ Infinite Loop

>>> S[2]="0"
1034 ✗ Correct! ✗"1034" ✗1204 ✗"1204" ✗ Error ✗ Infinite Loop

>>> [x+1 for x in range(4) if x != 2]
[3,4,5] ✗ Correct! ✗[1,2,4] ✗[1,2,4,5] ✗[2,4,5] ✗ Error ✗ None of these

def mystery(word):
    if word[0] in "aeiou":
        return word+"ay"
    else:
        return mystery(word[len(word)-1] + word[0:len(word)-1])

>>> mystery("school")
"schoolay"
Correct! ✗"oolay" ✗"oolschay" ✗"loohcs" ✗"loohcsay" ✗"olschoay" ✗ Error ✗ Infinite Loop

>>> mystery("sky")
"skyay"
Correct! ✗"yay" ✗"yskay" ✗"yks" ✗"yksay" ✗ Error ✗ Infinite Loop

b) Assume for this problem that *mystery* was given a *word* with many characters, and returned successfully. What is its running time (as a function of the length of *word*)? You may also assume that slicing, concatenation and determining length are constant-time operations. (select ONE for each side)

<table>
<thead>
<tr>
<th></th>
<th>Constant</th>
<th>Logarithmic</th>
<th>Linear</th>
<th>Quadratic</th>
<th>Cubic</th>
<th>Exponential</th>
<th>Reasonable Time</th>
<th>Not Reasonable Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Question 11: Gambling is illegal at Bushwood, sir, and I never slice... (9 pts=6+3)
Playing golf is so frustrating; you mean to hit it straight, but you end up hitting it to the left sometimes and to the right sometimes. In the far wall of your backyard (facing you) you’d like to be able to measure where you hit it, so you put up posts 10 feet apart from -100 (a far left shot) through 0 (perfect!!) through 100 (a far right shot).

-100 -90 -80 ... -10 0 10 ... 80 90 100

a) You want to write code that will simulate your golf swing, which means randomly hitting one of these posts exactly (so there’s an equal chance of hitting -80 as there is of hitting 10). You don’t want your simulation ever to miss a post, so it should never return, say, 93. Only the #s: -100, -90, ..., 90, 100. Complete the code; unfortunately someone already typed a 1 into the 1st argument of pick random.

b) We want to simulate swinging the club 1,000 times, and we’d like to record every unique swing into a list called UNIQUE SWINGS. For example, if the first five swings of our simulation were 0 (great swing!), it should only add 0 once to the UNIQUE SWINGS list. We code it as shown above. What do you think about it? (select ONE)

○ It’s buggy, replace the add with
○ It’s buggy; remove the not
○ It’s buggy; it could add more than 1000 swings to the list.
○ It’s buggy; it could add the same number twice to the list.
○ Works great!