Name: _____

This test has 17 questions across 17 pages worth a total of 140 points, and is to be completed in 120 minutes. Point totals are labeled next to each question. The exam is closed book, and no calculators or other electronic devices are permitted.

Write the statement out below in the blank provided and sign your name. You may do this before the exam begins.

"I have neither given nor received any assistance in the taking of this exam."

_____

_____

Signature: _____

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- Use your time wisely. If you are having too much trouble on a question, skip it and return to it later. **Work through the ones you are comfortable with first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.**
- If you are given skeleton code to fill in, we will not grade your answer if you alter the skeleton code or write outside the solution box.
- For questions with *square checkboxes*, you may select *more than one* choice.
- For questions with *circles*, you may *only select one* choice.
- Relax, and try to have fun with it! Think of this exam as a checkpoint along your JamCoders journey. You got this!

Staff use only.

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Total

1. ***(4 points)*** Consider the following code.

```
if a or b:
    if a:
        print("Robert")
    else:
        print("Anushka")
else:
    if b:
        print("Frank")
    else:
        print("Mallika")
```

Fill in the boxes next to all answers for which the code will print **only** `Anushka`.

- ☐ `a = True, b = True`
- ☐ `a = True, b = False`
- ☐ `a = False, b = True`
- ☐ `a = False, b = False`

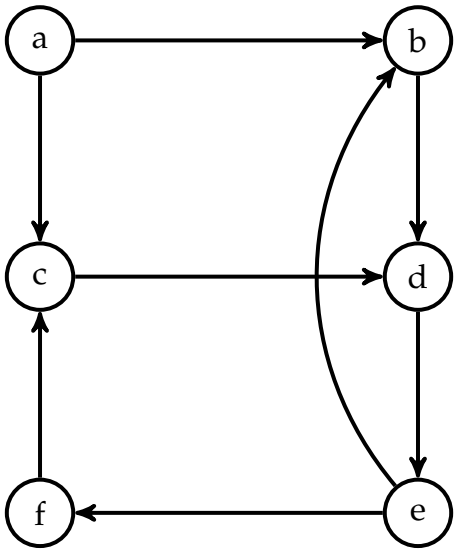
2. ***(6 points)*** What will be printed after running the following code? Write your answers on the dashed lines.

```
a = "Cheer"
b = "leader"
c = a + b * 2
print(c)        # Printed: ____________________________________
b = c[1:4]
print(b)        # Printed: ____________________________________
a = a + b
print(a)        # Printed: ____________________________________
```

3. ***(6 points)*** Complete the following code to return the minimum element in a non-empty list of integers.

```
def find_min(L):

    min_so_far = ________________

    for ______________ in ____________________________:

        if ________________________________________:

            min_so_far = ________________________________________

    return ________________________________________
```

4. For the next question, consider the following ***directed*** graph.



a. ***(5 points)*** Which of the following are cycles in the graph?

- ☐ $a \to b \to d \to e \to a$
- ☐ $b \to d \to e \to b$
- ☐ $c \to d \to e \to f \to c$
- ☐ $a \to c \to d \to b \to a$
- ☐ $d \to e \to f \to d$
- ☐ $e \to b \to a \to c \to e$

b. ***(4 points)*** How many **neighbors** does node `e` have?

c. ***(4 points)*** Which node in the graph has ***no incoming edges***?

5. Debugging

Each of the following programs contains a bug. For each one:

- Describe what goes wrong when the program is run.
- Write the line number that caused the problem.
- Briefly describe how to fix the bug.

a. ***(6 points)*** This program should print the numbers `0` through `4`.

```
1  i = 0
2  while i < 5:
3      print(i)
4  i += 1
```

Problem: _____ Line number: _____

Describe the Fix:

b. ***(6 points)*** This program should print every word in the list in uppercase.

```
1  words = ["cat", "dog", "bird"]
2  for i in range(len(words)):
3      print(i.upper())
```

Problem: _____ Line number: _____

Describe the Fix:

c. ***(6 points)*** This program should print the square of the number `5`.

```
1  def square(x):
2      return x * x
3
4  square(5)
5  print(result)
```

Problem: _____ Line number: _____

Describe the Fix:

6. ***(8 points)*** Adrianna invented a new number sequence. The n-th number in the sequence, g_n, is defined as follows:

$$g_0 = 6$$
$$g_1 = 7$$
$$g_n = 12 \cdot g_{n-1} - 7 \cdot g_{n-2} + 4$$

For example, the first few numbers in the sequence are $g_0 = 6, g_1 = 7, g_2 = 46$.

She has started to implement a function that calculates the n-th Adrianna number, but didn't have time to write the whole thing. Finish the code below.

```
def adrianna_num(n):

    if _________________________________:

        ____________________________________________________

    if _________________________________:

        ____________________________________________________

    return ________ * adrianna_num(n - 1) - 7 * ____________________ + 4
```

7. ***(8 points)*** What will be printed after running the following code? If there would be an error, write `error`. Write your answers on the dashed lines.

```
wildcats = {
    "Troy": "Sharpay",
    "cast": ["Troy", "Gabriella", "Sharpay", "Chad"],
    "callback_time": 4
}

wildcats["callback_time"] *= 2

print(wildcats["callback_time"]) # Printed: ____________________________

wildcats["Troy"] = "Gabriella"

print(wildcats["Troy"])          # Printed: ____________________________

wildcats["Chad"] = "Taylor"

print(wildcats["Chad"])          # Printed: ____________________________

print(wildcats["cast"][2])       # Printed: ____________________________

print(wildcats["cast"][10])      # Printed: ____________________________
```

8. ***(5 points)*** What is printed after running the following code? Write your answers on the dashed lines.

```
def magic(lst):
    if len(lst) == 1:
        return 2 * lst[0]
    return lst[0] + lst[-1] + magic(lst[:-1])

print(magic([4]))            # Printed: _______________________________

print(magic([1, 3, 5, 7]))   # Printed: _______________________________
```

9. ***(8 points)*** Sarika is vegetarian and keeps track of what she eats at Rex each day. Write a function `countVeggies` that accepts a list of strings representing dish names (`dishes`) and returns a ***dictionary*** containing the number of times each dish appears.

Each dish name should be a key in the dictionary, and its value should be the number of times that dish appears in the list.

Examples:

- `countVeggies(["tofu stir-fry", "veggie chunks", "tofu stir-fry"])` returns the dictionary: {`"tofu stir-fry": 2, "veggie chunks": 1`}
- `countVeggies(["curry chickpeas", "curry chickpeas", "curry chickpeas"])` returns the dictionary: {`"curry chickpeas": 3`}
- `countVeggies([])` returns the empty dictionary: {}

Complete the function below.

```
def countVeggies(dishes):

    counts = ______________________________

    for dish in ____________________________:

        if ________________________________:

            _________________________ = _________________________

        else:

            _________________________ = _________________________

    return ________________________________________
```

10. Orr and Andrey have "invented" their own custom logical operators:

```
def xORR(a, b):
    return (a and not b) or (not a and b)

def ANDrey(a, b):
    return a and b
```

`xORR(a, b)` returns `True` exactly when ***exactly one*** of `a` and `b` is `True`. `ANDrey(a, b)` returns `True` only when ***both*** `a` and `b` are `True`.

a. ***(4 points)*** What does `ANDrey(xORR(True, True), True)` evaluate to?

b. ***(8 points)*** A list `L` contains `True`, `False`, and `None` values. Fill in the recursive function `xorr_chain(L)` below, which ignores every `None` in `L` and applies `xORR` to all the remaining values, one after another, left to right.

For example, `xorr_chain([True, None, False, None, True])` first ignores both `None`s, leaving `[True, False, True]`, then computes `xORR(True, xORR(False, True))`, which is `xORR(True, True)`, which is `False`.

```
def xorr_chain(L):

    if ________________________________________________:

        return ________________________________________

    if ________________________________________________:

        return ____________________________________________________

    else:

        return ____________________________________________________
```

11. ***(8 points)*** Daniel and Naila run a cupcake stand, and they want their menu board sorted alphabetically by flavor name, using ***selection sort***.

Write a function `sort_cupcakes(flavors)` that takes a list of cupcake flavor name strings and returns the list sorted in alphabetical order, using selection sort.

You may assume every flavor name in `flavors` is entirely **lowercase**. Think of alphabetical order the same way you think of numerical order: if one flavor name comes earlier in the alphabet than another, treat it as the ***smaller*** of the two.

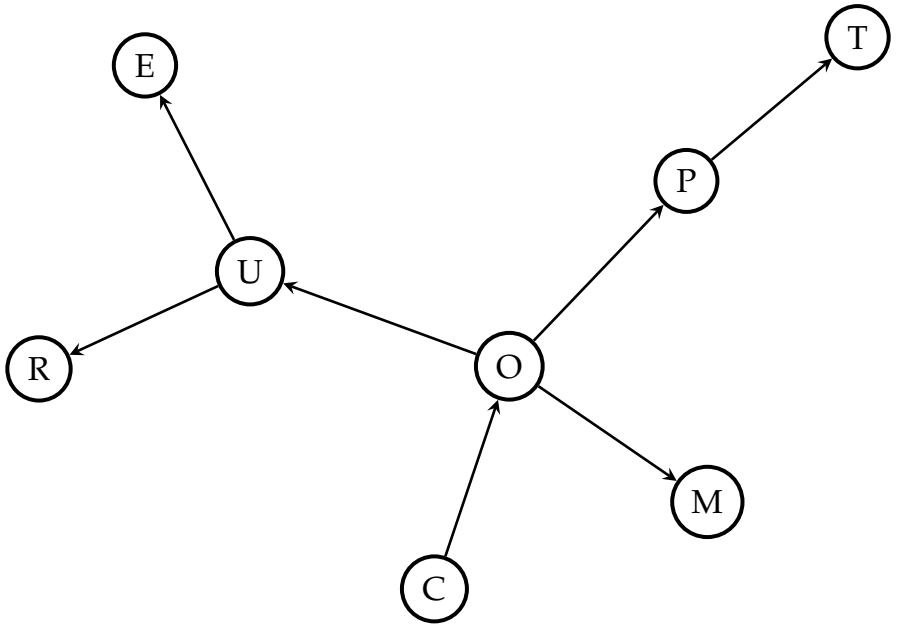
Examples:

- `sort_cupcakes(["vanilla", "red velvet", "chocolate", "lemon"])` returns `["chocolate", "lemon", "red velvet", "vanilla"]`.
- `sort_cupcakes(["strawberry", "chocolate", "banana"])` returns `["banana", "chocolate", "strawberry"]`.
- `sort_cupcakes([])` returns `[]`.

Complete the code below.

```
def sort_cupcakes(flavors):

    for cur_pos in ____________________________:

        s_index = ____________________________

        for next_pos in ____________________________:

            if ____________________________:

                s_index = ____________________________

        temp = ____________________________

        flavors[____________________________] = ____________________________

        flavors[____________________________] = ____________________________

    return flavors
```

12. Recall that Depth First-Search (DFS) and Breadth First-Search (BFS) are algorithms that traverse ("visit") nodes in a graph. Consider the following **directed** graph.



a. ***(5 points)*** In what order are the nodes visited during a *depth-first search* on the graph, starting from node *C*? Break ties alphabetically.

b. ***(5 points)*** In what order are the nodes visited during a *breadth-first search* on the graph, starting from node *C*? Break ties alphabetically.

13. Runtime Analysis

a. ***(6 points)*** Consider the following three functions:

$$f(n) = 8n^8 + 15n^5 + 2n + 9,$$
$$g(n) = n \log n + 5n,$$
$$h(n) = 4^n + n^3.$$

Rank these functions in order of asymptotic growth rate by filling in the blanks using f, g, and h. Use each exactly once.

As n gets large, _____(n) grows larger than _____(n), which grows larger than _____(n).

b. The following code defines two functions, `f` and `g`.

```
def f(n):
    for i in range(n):
        for j in range(1000000):
            print("work")

def g(n):
    for i in range(n):
        f(n)
```

Determine the ***tightest*** asymptotic (Big-O) runtime for each function.

i. ***(4 points)*** The runtime of `f` is:

- ○ $O(1)$
- ○ $O(n)$
- ○ $O(n \log n)$
- ○ $O(n^2)$
- ○ $O(n^3)$

ii. ***(4 points)*** The runtime of `g` is:

- ○ $O(n)$
- ○ $O(n^2)$
- ○ $O(n^2 \log n)$
- ○ $O(n^3)$
- ○ $O(2^n)$

14. ***(10 points)*** Sarika, Andrey, and Kanav play GeoGuessr together every single day, and they keep a list `scores` of their combined daily score, ordered from their very first day to their most recent. Since they've gotten steadily better with practice, `scores` is sorted in ***non-decreasing*** order (later scores are always greater than or equal to earlier ones, though some days may tie).

They want to know: what is the **first day** (i.e., the smallest index) on which their combined score was ***at least*** some `target` score? Since `scores` has thousands of entries, they want to answer this using **binary search** rather than scanning the whole list.

Write a function `first_day_reaching(scores, target)` that returns the index of the ***first*** score in `scores` that is greater than or equal to `target`, or `-1` if no score ever reaches `target`. Your solution should run in $O(\log n)$ time, where $n =$ `len(scores)`.

Examples:

- `first_day_reaching([1200, 1500, 1500, 1800, 2100, 2100, 2400], 1800)` returns `3`
- `first_day_reaching([1200, 1500, 1500, 1800, 2100, 2100, 2400], 2200)` returns `6`
- `first_day_reaching([1200, 1500, 1500, 1800, 2100, 2100, 2400], 3000)` returns `-1`

Hint: Regular binary search stops as soon as it finds *any* matching value. Here, once you find a day whose score meets the target, that *might not* be the **first** such day.

15. ***(10 points)*** A ***unigram language model*** assigns each word a probability based only on how often that word appears in the entire dataset. Each word is treated independently, so its probability does not depend on the word before it. The unigram probability of a word is its total number of appearances divided by the total number of words in the list.

Write a function `next_word_probabilities(words, previous_word, wordCounts)` that takes:

- a list of strings `words`,
- a string `previous_word`, and
- a dictionary `wordCounts` mapping each word to its total number of appearances in `words`.

Return a dictionary whose keys are **only** the words that appear ***immediately after*** `previous_word` somewhere in `words`. For each such word, its value should be its ***unigram probability***.

If no word ever appears immediately after `previous_word`, return an empty dictionary.

Example:

```
words = ["tung", "larp", "fairs", "tung", "tung",
         "rizz", "fairs", "67", "ball", "knowledge"]

wordCounts = {
    "tung": 3, "larp": 1, "fairs": 2, "rizz": 1,
    "67": 1, "ball": 1, "knowledge": 1
}

next_word_probabilities(words, "fairs", wordCounts)
# Returns: {"tung": 0.3, "67": 0.1}
```

This result contains `"tung"` and `67` because they are the only words that appear immediately after `"fairs"`. Their unigram probabilities are `0.3` and `0.1`, respectively.

Write your solution below.

16. ***(10 points) Optional*** **Challenge Problem:** *Finish other problems before attempting.*

Given an $m \times n$ 2D binary grid `grid`, where each cell contains either `'1'` or `'0'`, write a function named `num_coder_islands(grid)` that returns the number of ***Coder's Islands***.

A cell containing `"1"` represents a **coder station**, while a cell containing `"0"` represents the **ocean**.

A ***Coder's Island*** is a group of *one or more* adjacent coder stations connected **horizontally** or **vertically**. Diagonal connections do not count. You may assume that all four edges of the grid are surrounded by ocean.

Example 1:

```
grid = [
    ["1","1","1","1","0"],
    ["1","1","0","1","0"],
    ["1","1","0","0","0"],
    ["0","0","0","0","0"]
]
```

There is exactly one Coder's Island, so the function should return `1`.

Example 2:

```
grid = [
    ["1","1","0","0","0"],
    ["1","1","0","0","0"],
    ["0","0","1","0","0"],
    ["0","0","0","1","1"]
]
```

There are three Coder's Islands, so the function should return `3`.

17. ***(10 points) Optional*** **Challenge Problem:** *Finish other problems before attempting.*

Bezawit is building the ultimate playlist. She only listens to songs from five artists, and (for reasons known only to her) every song by a given artist is ***exactly*** the same length of time. For example, all of Burna Boy's songs are exactly 1 minute long. This information is given to you as a dictionary:

```
durations = {
    "Burna Boy": 1,
    "KATSEYE": 2,
    "Rophnan": 3,
    "Bad Bunny": 4,
    "Drake": 5
}
```

She wants her playlist to last **exactly** `n` minutes, with songs played back to back and no gaps. Two playlists are considered different if the ***order*** of artists is different (even if the total set of artists used is the same).

For example, if `n = 3`, there are 4 possible playlists: `[Burna Boy, Burna Boy, Burna Boy]`, `[Burna Boy, KATSEYE]`, `[KATSEYE, Burna Boy]`, and `[Rophnan]`.

Write a function `count_playlists(n, durations)` that returns the number of different playlists Bezawit could build that last exactly `n` minutes.

Hint: Think about the ***last*** song in the playlist. ***Loop over the dictionary*** rather than writing a separate case for each artist by hand.

You will only receive credit if your solution runs in $O(n)$ time. A correct, but slow (exponential-time) solution will receive ***no credit***, even if it produces the right output. Use *memoization*!