Addis Coder 2023 Quiz 4

Problem 1

What is the **output** of `bool(0.0)`?

A) `True`

B) `False`

C) `0.0`

D) `1.0`

Problem 2

How do you get the **last** element of a list `lst`?

A) `lst[-1]`

B) `lst(-1)`

C) `lst[len(lst)]`

D) `lst(len[lst])`

Problem 3

What happens when this function is called with `printNumbers(5)`? (**Hint: look at the syntax carefully!**)

```
def printNumbers(number):
    for i in range(len(number)):
        print(i)

printNumbers(5)
```

A) All numbers from 1 to 5 are printed.

B) All numbers from 0 to 4 are printed.

C) Infinite loop. The code runs but does not terminate.

D) Error. The code does not run at all.

Problem 4

What will the following recursive function return for `f(5)` ?

```
def f(n):
    if n == 0:
        return 0
    else:
        return n + f(n-1)
```

A) 10

B) 15

C) 5

D) 25

Problem 5

Which statement about binary search is **false**?

A) It is more efficient than linear search for large datasets.

B) The list must be sorted before performing a binary search.

C) Binary search starts searching from the middle of the list.

D) Binary search is the fastest way of finding an item in a dictionary.

Problem 6

In dynamic programming, the **memoization table** is used to:

A) Count the number of operations taken.

B) Remember results we have seen.

C) Calculate if two graph components are connected using DFS

D) Traumatize the students

Problem 7

Which of the following has the **highest growth rate** as n becomes very large?

A) 2^n

B) n^2

C) $n \log n$

D) n

Problem 8

Which of the following list comprehensions generates `[0, 1, 4, 9, 16]` ?

A) `[x * x for x in range(6)]`

B) `[x ** 2 for x in range(5)]`

C) `[x for x in range(1, 6) if x ** 2]`

D) `[x for x in range(5) ** 2]`

Problem 9

What will be printed after running the following code? Write your answers for each case.

```python
a = "Hello"
b = "AddisCoder"
a = a + a
print(a)       # Printed: ________________________
b = b[4:]
print(b)       # Printed: ________________________
a = a + b
print(a)       # Printed: ________________________
```

Problem 10

a) Write a function `lucky(n)` that takes an integer `n` and does the following:

- If `n` is less than 1 or greater than 100, print `"Invalid input"`
- If `n` is contains the digit 7 **once** (eg `7`, `17`, `87`, `70`, `75` etc), print `"LUCKY!"`
- If `n` has 7 in both digits (ie `77`), print `"DOUBLE LUCKY!"`
- In all other cases, print the number `n` itself

Example:

`lucky(17)` prints `LUCKY!`

`lucky(51)` prints `51`

`lucky(7)` prints `LUCKY!`

`lucky(0)` prints `Invalid input`

In [ ]:

```

```

b) Write a program that runs `lucky()` on numbers from 50 to 100 inclusive.

In [ ]:

```

```

Problem 11

What is the **runtime complexity** of the following functions? Assume that `n` is a positive integer.

In [ ]:
```
def fun1(n):
    i = 0
    while i < n:
        print(i)
        i += 1
```

In [ ]:

In [ ]:
```
def fun2(n):
    i = 0
    while 2**i < n:
        print(i)
        i += 1
```

In [ ]:

In [ ]:
```
def fun3(n):
    i = 0
    while i < 10000:
        print(i)
        i += 1
```

In [ ]:

In [ ]:
```
def fun4(n):
    i = 1
    while i < n:
        print(i)
        i *= 2
```

In [ ]:

In [ ]:
```
def fun5(n):
    i = 0
    while i < n**2:
        print(i)
        i += 1
```

In [ ]:

Problem 12

Consider the following function (**Hint: read the code carefully**):

```
def find(lst, elem):
    for i in range(len(lst)):
        if lst[i] == elem:
            return i
        else:
            return -1
```

What is the **output** of `print(find([1,2,3,4], 2))`?

In [ ]:

Problem 13

a) **Write a function** `is_sorted(lst)` which returns `True` if `lst` **is sorted** (from small to large), and `False` otherwise. Your function should run in $O(n)$ where `n` is the length of the list.

Hint: You should **NOT implement** and **NOT use** any sorting algorithms.

Example:

`is_sorted([5,1,2,4,3])` returns `False`

`is_sorted([1,1,2,2,3,3,10,100])` returns `True`

In [ ]:

Problem 14

Compare these two implementations of `factorial()` for positive numbers:

---

1. Non-memoized version

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)
```

---

2. Memoized version

```
def memo_factorial(n, mem):
    if n == 1:
        return 1
    if mem[n] != -1:
        return mem[n]
    else:
        answer = n * memo_factorial(n-1, mem)
        mem[n] = answer
        return answer

def factorial_main(n):
    mem = [-1]*(n+1)
    return memo_factorial(n, mem)
```

---

a) **How many times** is `memo_factorial()` called when computing `factorial_main(x)` for an integer $x > 0$?

In [ ]:

b) **How many times** is `factorial()` called when computing `factorial(x)` for an integer $x > 0$?

In [ ]:

c) Is the memoized version **faster** than the non-memo version?

In [ ]:

Problem 15

Consider the function `f` defined recursively as the following, and `m` and `n` are positive integers

`f(m, n) = f(m-1, n) + f(m, n-1)` and

`f(0, n) = f(m, 0) = 1` for $m, n \geq 0$.

Implement this function in python, and **add memoization** to speed up its computation.

In [ ]:

```

```

Problem 16

Look at this program:

```
cache = [-1, -1, -1, -1, -1, -1, -1, -1]
def calculate(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif cache[n] > -1:
        return cache[n]
    else:
        answer = calculate(n-1) + calculate(n-2)
        cache[n] = answer
        return answer
calculate(5)
```

What is the value of `cache` after this program runs? (**Hint:** Draw the *call tree* on paper. Which numbers were `calculate` called on?)

In [ ]:

```

```

Problem 17

The **snake graph** on `n` nodes is a graph where node `i` is connected to node `i+1` for $0 \leq i \leq n-2$. Example for `n=4`:

Write a function `snake_graph(n)` which returns the snake graph on `n` nodes in as an **adjacency dictionary**.

Example: `snake_graph(4)` should return `{0: [1], 1: [2], 2: [3], 3: []}`

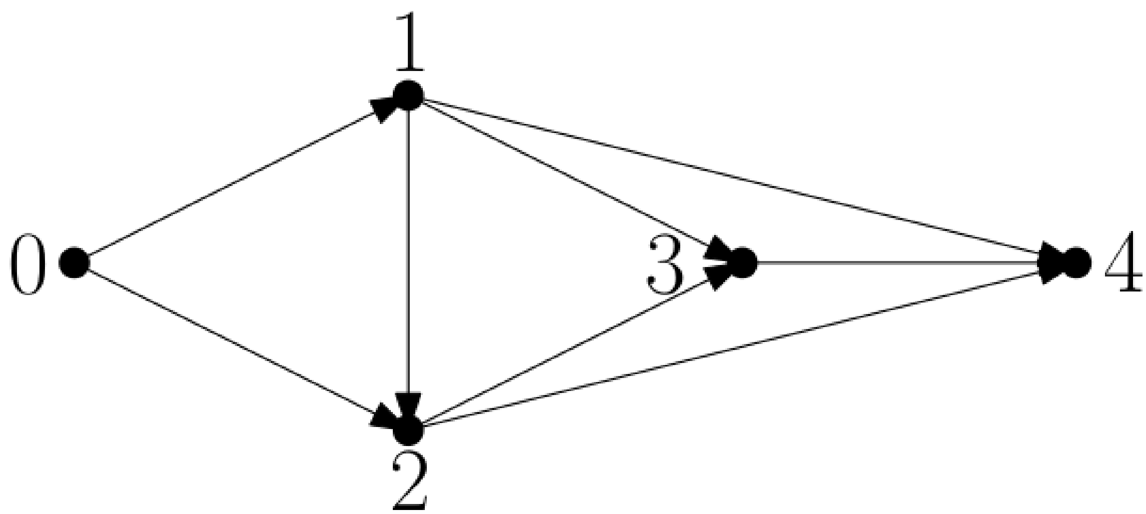
In [ ]:

```

```

Problem 18

Write a function `count_paths(G, a, b)` that returns the **number of paths** from `a` to `b` in the graph `G`. `G` is given as an **adjacency dictionary**, and has **no cycles** (which means there are no paths with repeated nodes).



Example: For the graph above, `count_paths(G, 0, 4)` should return 6, because there are 6 paths from 0 to 4:

- 0 -> 1 -> 2 -> 3 -> 4
- 0 -> 1 -> 2 -> 4
- 0 -> 1 -> 3 -> 4
- 0 -> 1 -> 4
- 0 -> 2 -> 3 -> 4
- 0 -> 2 -> 4

Hint: To get the `count_paths(G, a, b)` (the number of paths from `a` to `b`), use **recursion** assuming you know `count_paths(G, x, b)` (the number of paths from `x` to `b`) **for all neighbors** `x` of `a`!

For example, from `0`, you have to go to either `1` or `2` first, so `count_paths(G, 0, 4) = count_paths(G, 1, 4) + count_paths(G, 2, 4)`.

In [ ]:

```

```

Problem 19

Write a function `consecutive_as(s)` that takes a string `s` and determines the length of the **longest sequence of consecutive lowercase `"a"` 's** within the string.

Example:

- `consecutive_as("aabctqaaapwaaaabbbrpq")` should return `4`.
- `consecutive_as("bcdefg")` should return `0`.

In [ ]:

```

```

b) What is the time complexity of your solution if `s` is of length `n`?

In [ ]:

```

```

Problem 20

You're given a list `lst` of length `n`, which contains the integers from `0` to `n` inclusive on both ends, with **one number missing**.

a) **Write a function** `missing(lst)` which finds the **missing number**.

Example:

- `missing([4, 0, 1, 5, 2]`) should return `3`
- `missing([1, 0])` should return `2`

In [ ]:

```

```

b) What is the time complexity of your solution if `lst` is of length `n`?

In [ ]:

```

```