Midterm CSCI 356

Spring 2023

Name: ______________________________

Email: ______________________________

Problem	Points	Max
1		10
2		10
3		10
4		10
5		10
6		10
7		20
8		20
Total		100

Do not progress to the next page before being told to do so.

NOTES REGARDING TIME COMPLEXITY ANALYSIS

The time-complexity using Big-O notation for a code fragment establishes an upper bound on worst-case performance. When solving for the time complexity of a code fragment, always answer with the tightest possible time complexity you can justify. If your time complexity is greater than the tightest that can be justified given a good understanding of the problem, but your time complexity is a correct upper bound, you will get partial credit.

Assume all of the following operations take exactly 1 step.

- mathematical operators: division, multiplication, addition, and subtraction operators (/, *, +, -)
- assignment (e.g., x = 5)
- relational operators (<, >, <=, >=, ==)

We do not know the exact number of steps taken by operations on lists or dicts. For such operations just use the Big-O notation for the number of steps those operations take.

Operations on dicts and lists may take greater than $O(1)$ steps depending on the operation.

The time complexity of a function call depends on the code within the function. A function that does nothing but return takes exactly 1 step to call and return, add to this the time in steps to execute the body of the function.

Example Acceptable Answer

When answering with regard to time complexity be sure to specify which group of lines are repeated and how many times. For example,

```
for i in range(n):      # (1)  n * (lines (2) and (3))
    for j in range(n):  # (2)  n * (line (3))
        k = 2 + 2       # (3)  2 steps
```

Clearly define the time in steps for each block of code. The following would be a complete answer:

Let $T(n)$ denote the execution time measured in steps of the code fragment above.

Let $T_3(n)$ denote the time in steps to execute line (3) as a function of n.

Let $T_2(n)$ denote the time in steps to execute lines (2) and (3).

Let $T_1(n)$ denote the time in steps to execute lines (1), (2), and (3). Since this encompasses all lines $T(n) = T_1(n)$ is the total number of steps to complete the code fragment above.

Let's start by analyzing the code inside the loops, i.e., line (3).

```
k = 2 + 2
```

As per the notes on page 2, the above is exactly 2 steps.

$$
\begin{aligned}
&T_3(n) = 2 \text{ steps} \\
&T_2(n) = n \cdot T_3(n) = 2n \text{ steps} \\
&T_1(n) = n \cdot T_2(n) = n \cdot 2n = 2n^2 \text{ steps}
\end{aligned}
$$

Because T_1 covers all steps,

$$T(n) = 2n^2 \tag{1}$$

The definition of big-O states that function f(n) = O(g(n)) provided there exists a C and n_0 such that for all $n > n_0$, $f(n) \leq Cg(n)$. Therefore,

$$T(n) = 2n^2 \leq Cn^2 \text{ for } C >= 2. \tag{2}$$

Thus,

$$\boxed{T(n) = O(n^2)} \tag{3}$$

STOP

Do not progress to the next page before being told to do so.

Problem 1 (10 points)

What is the tightest time complexity you can justify for the following code fragment? Consult the "Example Acceptable Answer" on page 3.

```
x = 0
for i in range(0, n, 3):   # the 3 denotes skip 3 so i = 0, 3, ...
   x = x + i
```

Problem 2 (10 points)

What is the tightest time complexity you can justify for the following code fragment? Consult the "Example Acceptable Answer" on page 3.

```
def f(x):
    x *= 2
    return x

def h(x):
    x = f(x)
    x = f(x)

h(x)
```

Problem 3 (10 points)

What is the tightest time complexity you can justify for the following code fragment. Consult the "Example Acceptable Answer" on page 3.

```
i = 0
j = 1
while i * i < n: # n is set before this code fragment
    j *= 2
    i += 1
```

Problem 4 (10 points)

What is the tightest time complexity you can justify for the following code fragment? Consult the "Example Acceptable Answer" on page 3.

```
x = 0
for i in range(n):
   for j in range(i, n):
       x += i * j
```

Problem 5 (10 points)

What is the tightest time complexity you can justify for calling `f(n)` as a function of n? Consult the "Example Acceptable Answer" on page 3. NOTE: Calling `randint()` once takes $O(1)$ time. Ignore the cost of importing.

```
from random import randint

def f(n):
    x = []
    for _ in range(n):
        x.append(randint(0,1000))

    for _ in range(n):
        x.insert(randint(0, len(x)), randint(0, 1000))
```

Problem 6 (10 points)

What is the tightest time complexity you can justify for calling `f(n)` as a function of n? the following code fragment. Consult the "Example Acceptable Answer" on page 3. NOTE: Calling `randint()` takes $O(1)$ time. Ignore the cost of importing. `shuffle()` takes $O(n)$ time.

```
from random import randint, shuffle

def f(n):
    keys = []
    for i in range(n):
        keys.append(i)
    shuffle(keys)  # shuffle takes O(n). It randomly reorders the list.
    d = {}
    for k in keys:
        d[k] = randint(0, 1000)
    shuffle(keys)
    for k in keys:
        del d[k]
```

Problem 7 (20 points)

The following is based on the `high_low` example we went over in class.

What is the tightest time complexity you can justify for calling find_multi given a `sorted_list` of length n and a list of `targets` of length m. The resulting time complexity will be a function of both n and m. Consult the "Example Acceptable Answer" on page 3.

```
def high_low(sorted_list, target):
    """
    find whether the target value is in the sorted list using binary
    search.
    """

    low = 0
    high = len(sorted_list) - 1

    while low <= high:
        mid = (low + high) // 2
        mid_value = sorted_list[mid]

        if mid_value == target:
            return mid
        elif mid_value < target:
            low = mid + 1
        else:
            high = mid - 1
    return None

def find_multi(sorted_list: list, targets: list):
    """
    This returns the subset of the targets that were
    found in the passed `sorted list`.  The targets
    are not in necessarily in order.
    """

    found = []
    for tgt in targets:
        i = high_low(sorted_list, tgt)
        if i is not None:
            found.append(tgt)

    return found
```

Problem 7 (cont.)

More space for problem 7.

Problem 8 (20 points)

Write an iterator class that that traverses a list in reverse order, i.e., provide the body of the `__init__` and `__next__` methods.

```
class ReverseIterator:

    def __init__(self, x: list):
        ...

    def __iter__(self):
        return self

    def __next__(self):
        ...
```

(b) Write a code fragment showing the iterator being used by a for loop to iterate over an example list.

Problem 8 (cont.)

(c) What is the time complexity of iterating over all elements in the list in reverse order?