Midterm CSCI 356 Answers

Spring 2023

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
```

Short Answer 1

The short answer is sufficient to obtain full credit, although more detail makes it easier to give partial credit.

```
x = 0                      # (1)   1 step
for i in range(0, n, 3):   # (2)   execute (3) n // 3 + 1 times.
   x = x + i               # (3)   2 steps
```

Let $T(n)$ denote the total number of steps in the code snippet above.

Let $T_1(n)$ denote the number of steps in line (1).

Let $T_2(n)$ denote the number of steps in lines (2) and (3). T_2 includes both lines because (2) and (3) form a "for" loop.

Let $T_3(n)$ denote the number of steps in line (3).

$$
\begin{aligned}
T_3(n) &= 2 = O(1) \\
T_2(n) &\approx n/3 \cdot T_3(n) = n/3 \cdot O(1) = O(n) \\
T_1(n) &= 1 = O(1)
\end{aligned}
$$

$$T(n) = T_1(n) + T_2(n) = O(1) + O(n) = O(n) \tag{1}$$

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

For some of the subsequent problems, I only provide the detailed answer since the detailed answer provides more clarity. Full credit was given for anwsers much closer to this **Short Answer**.

Detailed Answer 1

```
x = 0                      # (1)   1 step
for i in range(0, n, 3):   # (2)   execute (3) n // 3 + 1 times.
   x = x + i               # (3)   2 steps
```

Let $T(n)$ denote the total number of steps in the code snippet above.

Let $T_1(n)$ denote the number of steps in line (1).

Let $T_2(n)$ denote the number of steps in lines (2) and (3). T_2 includes both lines because (2) and (3) form a "for" loop.

Let $T_3(n)$ denote the number of steps in line (3).

$$T(n) = T_1(n) + T_2(n) \tag{3}$$

$T_3(n)$ and $T_1(n)$ are

$$T_3(n) = 2$$
$$T_1(n) = 1$$

However, $T_2(n)$ is a little more complicated. $T_2(n)$ will cause line (3) to execute multiple times depending on n as

n	i	times (3) executed
1	0	1
2	0	1
3	0	1
4	0, 3	2
5	0, 3	2
6	0, 3	2
7	0, 3, 6	3
8	0, 3, 6	3
9	0, 3, 6	3
10	0, 3, 6, 9	4
...	...	...

$$T_2(n) = \left(\left\lfloor \frac{n-1}{3} \right\rfloor + 1 \right) \cdot T_3(n)$$
$$T_2(n) \leq \left(\frac{n-1}{3} + 1 \right) \cdot T_3(n)$$
$$T_2(n) \leq 2\left(\frac{n-1}{3} + 1 \right)$$

Substituting T_1 and T_2 into Equation 3 yields the inequality

$$T(n) \leq 1 + 2\left(\frac{n+2}{3}\right) \tag{4}$$

Thus, Inequality 4 can be rewritten as

$$T(n) \leq \frac{2}{3}n + \frac{7}{3} \tag{5}$$

From the definition of Big-O notation, we say that $f(n) = O(g(n))$ if there exists a C and an n_0 for which for all $n \geq n_0$, $f(n) \leq C \cdot g(n)$. Let's try $g(n) = n$.

Does there exist a C and n_0 such that the following is true?

$$Cn \geq \frac{2}{3}n + \frac{7}{3} \tag{6}$$

Let $C = 1$ and find n_0 for which Cn is greater for all $n \geq n_0$.

$$\begin{aligned} 1 \cdot n &\geq \frac{2}{3}n + \frac{7}{3} \\ n &\geq 7 \end{aligned}$$

Thus our n_0 is 7 for $C = 1$. There thus exists a C and n_0 for which Inequality 6 holds true. Thus,

$$\boxed{T(n) = O(n)}$$

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)
```

Answer 2

```
def f(x):        # (1) 1 step for call and return
    x *= 2       # (2) 1 step
    return x     # (3)

def h(x):        # (4) 1 step for call and return
    x = f(x)     # (5) executes (1)-(3)
    x = f(x)     # (6) executes (1)-(3) again.

h(x)             # (7) executes (4)-(6)
```

Note that the code contains no reference to n. As such the performance should not vary with n, and we would expect $O(1)$ time complexity. If you stated this, this was enough to get full credit.

We can however break it out a bit further:

Let $T(n)$ denote the number of steps to execute the above code snippet.

Let $T_h(n)$ denote the number of steps to execute function `h(x)`.

Let $T_f(n)$ denote the number of steps to execute function `f(x)`.

Lines (1)-(6) define two function which are not executed until line (7). Thus,

$$T(n) = T_h(n) \tag{7}$$

The function `h(x)` executes `f(x)` twice so,

$$\begin{aligned} T_f(n) &= 2 \\ T_h(n) &= 1 + T_f(n) + T_f(n) = 1 + 2 \cdot T_f(n) \\ T(n) &= T_h(n) = 1 + 2 \cdot 2 = 5 \end{aligned}$$

According to the definition of big-O notation, does there exists a C and n_0 such that $Cg(n) \geq T(n)$ for all $n \geq n_0$? If we let $C = 5$ and $g(n) = 1$ then the question becomes

$$5 \cdot 1 \geq 5 \tag{8}$$

which is true for all n. Thus

$$\boxed{T(n) = O(1)} \tag{9}$$

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
```

Answer 3

```
i = 0             # (1)  1 step
j = 1             # (2)  1 step
while i * i < n:  # (3)  some number of times lines (4) and (5)
    j *= 2        # (4)  1 step
    i += 1        # (5)  1 step
```

Let $T_3(n)$ denote the time in steps to execute lines (4) and (5).

Let $T_2(n)$ denote the time in steps to execute lines (3)-(5).

Let $T_1(n)$ denote the time in steps to execute lines (1)-(2).

Let $T(n)$ denote the time to execute the above code fragment.

Because the entire code fragment spans lines (1)-(5).

$$T(n) = T_1(n) + T_2(n) \tag{10}$$

$$T_1(n) = 2$$
$$T_3(n) = 2$$

Without changing the semantics, Line (3) can be rewritten as

```
while i < sqrt(n):     # (3')
    ...
```

This means that lines (4) and (5) are executed $\lfloor\sqrt{n}\rfloor$ times. Thus,

$$T_2(n) = \lfloor\sqrt{n}\rfloor \cdot T_3(n) \tag{11}$$

So Equation 10 becomes

$$T(n) = 2 + 2\lfloor\sqrt{n}\rfloor \tag{12}$$

If we pick $C > 2$ then $C\sqrt{n}$ outpaces $T(n)$ given in Equation 12, so

$$\boxed{T(n) = O(\sqrt{n})} \tag{13}$$

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
```

Short Answer 4

```
x = 0                       # (1)    1 step
for i in range(n):          # (2)    n * step (3) and (4).
   for j in range(i, n):    # (3)    (n-i) * step (4).
       x += i * j           # (4)    2 steps
```

Let $T(n)$ denote the number of steps executed by the code fragment above.

Let T_1 denote the number of steps to execute line (1).

Let T_2 denote the number of steps to execute lines (2)-(4).

Let T_3 denote the number of steps to execute lines (3) and (4).

Let T_4 denote the number of steps to execute line (4).

$$T(n) = T_1(n) + T_2(n) \tag{14}$$
$$T_1(n) = 1 \tag{15}$$
$$T_2(n) = n \cdot T_3(n) \tag{16}$$
$$T_4(n) = 2 \tag{17}$$
$$\tag{18}$$

$T_3(n)$ is a bit more complicated.

$$T_3(n) = (n - i) \cdot T_4(n) = 2(n - i) \tag{19}$$

Some students stated

$$2(n - i) \leq 2n \tag{20}$$

If we substitute that into Equation 19 we get

$$T_3(n) \leq 2n \tag{21}$$

We can then substitute Inequality 21 into our definition for $T_2(n)$ to get

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

We can then substitute Equation 15 and the above into Equation 14 to get

$$T(n) \leq 1 + 2n^2 \tag{22}$$

Thus

$$\boxed{T(n) = O(n^2)} \tag{23}$$

I gave credit for this, because it is a correct upper bound that happens to be as tight as you can get, even though demonstrating there is no tighter upper bound requires taking into account the $(n - i)$.

Detailed Answer 4

For the detailed answer we take into account $(n - i)$.

$$T(n) = T_1(n) + T_2(n) \tag{24}$$
$$T_1(n) = 1 \tag{25}$$
$$T_2(n) = \sum_{i=0}^{n-1} T_3(i, n) \tag{26}$$
$$T_3(n) = 2(n - i) \tag{27}$$

Substituting Equation 27 into Equation 26 yields

$$T_2(n) = \sum_{i=0}^{n-1} 2(n - i) = 2 \sum_{i=0}^{n-1} (n - i) \tag{28}$$

The summation may not be familiar with many of you, but I bet most people are familiar with

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{29}$$

We can use this if we massage the summation in Equation 29 to

$$T_2(n) = 2\left[\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i\right] \tag{30}$$

$$= 2\left[n \cdot n - \sum_{i=1}^{n-1} i\right] \tag{31}$$

$$= 2\left[n^2 - \frac{(n-1)n}{2}\right] \tag{32}$$

$$= 2\left[n^2 - \frac{n^2}{2} + \frac{n}{2}\right] \tag{33}$$

$$= 2\left[\frac{n^2}{2} + \frac{n}{2}\right] \tag{34}$$

$$= n^2 + n \tag{35}$$

If we substitute the above into Equation 24 we get

$$T(n) = n^2 + n + 1 \tag{36}$$

To satisfy the definition of big-O we need a C and n_0 such that $T(n) \leq Cn^2$ for all $n \geq n_0$. If we set $C = 2$ we can solve for an n that satisfies

$$n^2 + n + 1 \leq 2n^2 \tag{37}$$

$$n \geq 1 + \frac{1}{n} \tag{38}$$

Since $1/n < 1$ for all $n > 1$ we can set our $n_0 = 2$.

Thus,

$$\boxed{T(n) = O(n^2)} \tag{39}$$

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))
```

Answer 5

```
def f(n):
    x = []                            # (1)  1 step
    for _ in range(n):                # (2)  n * steps in line (3)
        x.append(randint(0,1000))     # (3)  O(1) steps

    for _ in range(n):                # (4)  n * steps in line (5)
                                      # (5)  O(n) worst case
        x.insert(randint(0, len(x)), randint(0, 1000))
```

Let $T(n)$ denote the time in steps to execute all steps in `f(n)`.

Let $T_1(n)$ denote the number of steps to run line (1).

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

Let $T_3(n)$ denote the number of steps to execute line (3).

Let $T_4(n)$ denote the number of stps to execute lines (4) and (5).

Let $T_5(n)$ denote the number of steps to execute line (5).

$$
\begin{aligned}
T(n) &= T_1(n) + T_2(n) + T_4(n) \\
T_1(n) &= 1 \\
T_2(n) &= n \cdot T_3(n) \\
T_3(n) &= O(1) \\
T_4(n) &= n \cdot T_5(n) \\
T_5(n) &= O(n)
\end{aligned}
$$

$T_5(n)$ is $O(n)$ worst-case because an insertion may occur near the beginning of the list. If an insertion occurs at the beginning of the list then every element

must be copied one position to the right.

Combining all fo the components into $T(n)$ yields

$$\begin{aligned} T(n) &= 1 + n \cdot T_3(n) + n \cdot T_5(n) \\ T(n) &= 1 + O(n) + n \cdot O(n) \\ T(n) &= 1 + O(n) + O(n^2) \end{aligned}$$

The $O(n^2)$ term dominates so this becomes

$$\boxed{T(n) = O(n^2)} \tag{40}$$

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]
```

Answer 6

```
def f(n):
    keys = []                     # (1) O(1)
    for i in range(n):            # (2) n * steps in line (3)
        keys.append(i)            # (3) O(1)
    shuffle(keys)                 # (4) O(n)
    d = {}                        # (5) O(1)
    for k in keys:                # (6) n * steps in line (7)
        d[k] = randint(0, 1000)   # (7) O(1)
    shuffle(keys)                 # (8) O(n)
    for k in keys:                # (9) n * steps in line (10)
        del d[k]                  # (10) O(1)
```

Let $T_i(n)$ for $i \in U$ where $U = 1, 3, 4, 5, 7, 8, 10$ denotes the time in steps to execute line i .

Let $T_i(n)$ for $i \in V$ where $V = 2, 6, 9$ denotes the time in steps to execute line (i) and line $(i + 1)$.

$$T(n) = T_1(n) + T_2(n) + T_4(n) + T_5(n) + T_6(n) + T_8(n) + T_9(n)$$
$$T(n) = O(1) + n \cdot O(1) + O(n) + O(1) + n\dot{O}(1) + O(n) + n \cdot O(1)$$
$$T(n) = O(1) + O(n) + O(n) + O(1) + O(n) + O(n) + O(n)$$

$O(n)$ dominates over $O(1)$ and $O(n) + O(n) = O(n)$ so the above becomes

$$\boxed{T(n) = O(n)} \tag{41}$$

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
```

Answer on the next page.

Answer 7

```
def high_low(sorted_list, target):
    low = 0                             # (1)  1 step
    high = len(sorted_list) - 1         # (2)  2 steps

    while low <= high:                  # (3)
        mid = (low + high) // 2         # (4)  3 steps
        mid_value = sorted_list[mid]    # (5)  O(1)

        if mid_value == target:         # (6)  1 step
            return mid                  # (7)  1 step
        elif mid_value < target:        # (8)  1 step
            low = mid + 1               # (9)  2 steps
        else:                           # (10) 1 step
            high = mid - 1              # (11) 2 steps
    return None                         # (12) 1 step

def find_multi(sorted_list: list, targets: list):
    found = []                          # (13) 1 step
    for tgt in targets:                 # (14) m * steps in (15)-(17)
        i = high_low(sorted_list, tgt)  # (15) steps in high_low.
        if i is not None:               # (16) 1 step
            found.append(tgt)           # (17) O(1)
    return found                        # (18) 1 step
```

Let $T(n, m)$ be the time in steps to execute `find_multi` passed a `sorted_list` with n items and a `targets` list containing m items.

Let $T_h(n)$ be the time in steps to execute `high_low`.

Let T_i denote the time to execute line (i).

$$
\begin{aligned}
T(n,m) &= T_{13}(n) + m \cdot T_{14}(n) + T_{18}(n) \\
T(n,m) &= 1 + T_{14}(n) + 1 \\
T_{14}(n) &= m \cdot \Big(T_{15}(n) + T_{16}(n) + T_{17}(n) \Big) \\
T_{15}(n) &= T_h(n) \\
T_{14}(n) &= m \cdot \Big(T_h(n) + 1 + O(1) \Big) \\
T_{14}(n) &= m T_h(n) + O(m)
\end{aligned}
$$

Combining the above yields

$$T(n,m) = mT_h(n) + O(m) \tag{42}$$

We step into `high_low` to determine the time cost of T_h.

$$\begin{aligned} T_h(n) &= T_1(n) + T_2(n) + T_3(n) + T_{12}(n) \\ T_h(n) &= 1 + 2 + T_3(n) + 1 \\ T_h(n) &= T_3(n) + 4 \end{aligned}$$

Lines (4)-(11) are all $O(1)$. Because $O(1) = O(1) + O(1)$, we can combine the time complexity of all of these lines to $O(1)$.

$$T_3(n) = k \cdot O(1) \tag{43}$$

where k denotes the number of times the while loop repeeats.

Since the algorithm divides the space by two with each iteration, this results in logarithmic growth in n. So $k = \log n$.

$$T_3(n) = \log n \cdot O(1) \tag{44}$$

and $T_h(n)$ becomes

$$T_h(n) = \log n \cdot O(1) + 4 = O(\log n) \tag{45}$$

and substituting $T_h(n)$ into Equation 42 yields

$$\begin{aligned} T(n,m) &= m \cdot O(\log n) + O(m) \\ T(n,m) &= O(m \log n) + O(m) \end{aligned}$$

Because $O(m \log n)$ grows faster than $O(m)$ we drop the $O(m)$ yielding

$$\boxed{T(n,m) = O(m \log n)} \tag{46}$$

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):
        ...
```

“‘

Answer 8

```
class ReverseIterator:

    def __init__(self, x: list):
        self._x = x
        self._i = len(x)

    def __iter__(self):
        return self

    def __next__(self):
        self._i -= 1
        if self._i < 0:
            raise StopIteration
        return self._x[self._i]
```

See the code in `reverse_iterator.py` and the tests in `test_reverse_iterator.py`.

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

Answer 8(b)

```
x = [1, 4, 3]
rit = ReverseIterator(x)
for i in rit:
    print(i)
```

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

Answer 8(c)

```
def __next__(self):
    self._i -= 1             # (1)  O(1)
    if self._i < 0:          # (2)  O(1)
        raise StopIteration  # (3)  O(1)
    return self._x[self._i]  # (4)  O(1)
```

Let $T(n)$ denote the number of steps required to iterate over all elements in a list using ReverseIterator. For example,

```
x = [1, 4, 3]                # (5)  O(n)
rit = ReverseIterator(x)     # (6)  O(1)
for i in rit:                # (7)  len(x) * steps in (8)
    ...   # do something     # (8)  assume O(1)
```

The time complexity of a single call to `__next__` is $O(1)$.

Each pass through the `for` loop results in a call to `__next__`. Thus `__next__` is called `len(x)` times. Consider a list with n items, i.e., `n=len(x)`. Then the time complexity becomes

$$T(n) = n \cdot O(1) \tag{47}$$

Thus,

$$\boxed{T(n) = O(n)} \tag{48}$$