CSCI 356 Homework 5: Bonus

Due: Wednesday, November 29, 11:59 PM

WARNING. Students may not work together. Students may discuss the problems with each other, but do not give any other student your solutions.

This is an optional homework. It replaces your lowest score on a homework.

1 Grading

This homework is out of a 100 points, but it will be scaled such that it has equal weight as a homework and then it will replace the lowest homework's grade. This homework is out of 100 simply because there are many parts and it is preferable to avoid scoring with fractions of a point.

Problem Number	Points Received	Max
1		10
2		20
3		10
4		10
5		10
6		10
7		10
8		10
9		10
Total		**100**

Table 1: Student Score Table

2 Submission

This homework contains a mix of time complexity analysis and a couple of short answer code writing questions.

Feel free to handwrite answers and scan in the answers or create a text file that contains the answers. You can copy code from this file hw5.pdf or its tex version which can be found in the source code repository at `csci-356/hw5.tex`.

Submit the answers as a single file to blackboard named `hw5_lastname_firstname.ext` where `ext` depends on the type of file submitted. A scanned document should be packaged into a `.pdf`. A text document should be a `.txt` file.

3 Answer review

I will go over the answers to the bonus homework on the last day of class. This is why the homework is due on Wednesday. There will be no extensions for this homework.

4 Preparation for the Final

The final will be a written exam. It will ask a number of time complexity questions and a few short programming questions using data structures that were covered in the class. The problems will be similar to those appearing in this homework.

5 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 (¡, ¿, ¡=, ¿=, ==)
- the act of callling a function (creating and pushing an activation record)
- returning from a function (popping the activation record)

Note that calling a function takes 1 step, but we must add to the time complexity the number of steps within the function.

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.

The definition for big-O is as follows:

> A function $f(n)$ is said to be $O(g(n))$ if and only if there exist postive constants C and n_0 such that
>
> $$f(n) \leq C \cdot g(n)$$
>
> for all $n \geq n_0$.

Similarly big-Ω is defined by

> A function $f(n)$ is said to be $O(g(n))$ if and only if there exist postive constants C and n_0 such that
>
> $$f(n) \geq C \cdot g(n)$$
>
> for all $n \geq n_0$.

6 Example Acceptable Answers

6.1 Example 1

When answering with regard to time complexity be sure to specify which group of lines are repeated and how many times. You are given the code snippet:

```
i = 12
i += 1
```

First label each step, and if it is a single step or known number of steps then write the number of steps for each line.

```
i = 12       # (1)   1 step
i += 1       # (2)   1 step
```

Group lines of code into blocks that contain lines that always execute one after another without looping or branching. In the above code there are no loops (i.e., no `while` or `for` statement) and there are no `if..then..` statements, so the code executes line (1) and then line (2). Define a time complexity variable T for each such code block. In this case we only have one code block, so I define T to mean the time complexity of executing lines (1) and (2).

The time complexity T is thus given by

$$T = 1 \text{ step } + 1 \text{ step } = 2 \text{ steps} \tag{1}$$

By the definition of big-O notation,

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

if and only if there exists C and n_0 such that $C > 0$ and $T \leq Cg(n)$ for all $n \geq n_0$.

If we choose $g(n) = 1$ then we see whether we can define a C and n_0 that satisfies the definition of big-O in order declare that T is $O(g(n)) = O(1)$.

For the follwoing inequality

$$T = 2 < C \cdot 1 \tag{3}$$

is true for any $C \geq 2$ and because 2 is not a fucntion of n any n_0 will satisfy the definition. Therefore,

$$\boxed{T = O(1)} \tag{4}$$

■

The above is a detailed justification for $T = O(1)$, and it would receive full credit, but it goes well beyond what is necessary to receive full credit.

A brief answer that would receive full credit would be the following:

```
i = 12       # (1)   1 step
i += 1       # (2)   1 step
```

let T denote the number of steps to execute lines (1) and (2).

$$\boxed{T = 1 + 1 = O(1)} \tag{5}$$

6.2 Example 2

Analyze the following code snippet to determine its time complexity. Provide the entire analysis.

```
def f(x: list):
    k = 5
    k -= 1
    k *= 2

f(x)
```

6.2.1 Descriptive answer

In answering the question above, I first annotate the code snippet as follows:

```
def f(x: list):          # x contains n items
    k = 5                # (1)  1 step
    k -= 1               # (2)  1 step
    k *= 2               # (3)  1 step
    # (4) returning from f(x) is 1 step to pop activation record

f(x)       # (5) call to f(x) is 1 step to push activation record
```

We add steps in a code block together to obtain the total number of steps executed in a code block. Since there are no for loops or while loops, each line is executed exactly once. Thus the number of steps to execute lines (1)-(5) is $1 + 1 + 1 + 1 + 1 = 5$.

Let $T(n)$ denote the number of steps as a function of some input n. Therefore for the code snippet above, $T(n) = 5$.

By the definition of big-O, $T(n) = O(g(n))$ if there exists a C and n_0 such that $C \cdot g(n) \geq T(n)$ for all $n > n_0$. We want the slowest growing function g(n) that can be justified given our understanding of the problem. Let's hypothesize $g(n) = n$.

$$
\begin{aligned}
g(n) &= n \\
T(n) &= 5 \leq C \cdot n \text{ if } C = 5 \text{ and } n_0 \text{ is any number.} \\
T(n) &= O(n)
\end{aligned}
$$

It is correct that $T(n) = O(n)$, but does there exist a tighter bound that can be justified given a reasonable understanding of the code. To get full credit, you must show a tighter bound. Let's hypothesize $g(n) = 1$.

$$
\begin{aligned}
g(n) &= 1 \\
T(n) &= 5 \leq C \cdot 1 \text{ if } C = 5 \text{ and } n_0 \text{ is any number.} \\
T(n) &= O(1)
\end{aligned}
$$

I know of no tighter bound than $O(1)$ so this is my final answer.

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

6.2.2 Acceptable answer

This descriptive answer goes far beyond what one needs to say to receive full credit. The follwing answer is acceptable.

```
def f(x: list):          # x contains n items
    k = 5                # (1)  1 step
    k -= 1               # (2)  1 step
    k *= 2               # (3)  1 step
    # (4) 1 step to pop activation record

f(x)       # (5) 1 step to push f(x) activation record
```

Let T denote the number of steps to execute the above code snippet.

$$T = 1 \text{ step call } + 3 \text{ steps body } + 1 \text{ return}$$

$$\boxed{T = O(1)}$$

6.3 Example 3

Analyze the following code snippet to determine its time complexity. Provide the entire analysis.

```
k = 5
for i in range(n):
    for j in range(m):
        k += 2
```

6.3.1 Descriptive answer

We first label all of the lines and provide some description of the number of steps each line takes.

```
k = 5                   # (1)  1 step
for i in range(n):      # (2)  n * (lines (3) and (4))
    for j in range(n):  # (3)  n * (line (4))
        k += 2          # (4)  1 step
```

Each `for` loop executes its body a number of times. The first for loop executes the code block spanning lines (3) and (4) n times. The inner for loop executes the code block spanning line (4) n times.

The above code has one step that takes place outside of for loops, i.e., the assignment `k=5`. We add that to the number of steps

Let $T(n)$ denote the number of steps to execute lines (1)-(4).

Let $T_1(n)$ denote the number of steps to execute the `for` loop with a body that spans lines (3) (4).

Let $T_2(n)$ denote the number of steps to execute the `for` loop with a body that spans line (4).

$$\begin{aligned} T(n) &= 1 \text{ step for line (1)} \\ &+ T_1(n) \text{ for lines (2) to (4)} \\ T(n) &= 1 + T_1(n) \end{aligned}$$

The definition of T_1 includes the time to execute the inner loop.

$$\begin{aligned} T(n) &= 1 + n \cdot T_2 \\ T_2(n) &= n \cdot 1 \\ T(n) &= 1 + n^2 \end{aligned}$$

Using big-O notation, we hypothesize that $T(n) = O(g(n))$ where $g(n) = n^2$. To prove the hypothesis that $T(n) = O(n^2)$ we need to find a C and n_0 for which $T(n) \leq C \cdot g(n)$ for all $n \geq n_0$.

$$T(n) = 1 + n^2 \stackrel{?}{\leq} Cn^2$$

Rearranging the inequality above yields

$$\begin{aligned} 1 + n^2 &\leq Cn^2 \\ C &\geq \frac{1 + n^2}{n^2} \\ C &\geq 1 + \frac{1}{n^2} \end{aligned}$$

Because $\frac{1}{n^2} \leq 1$ for all $n \geq 1$,

$$\begin{aligned} C &= 2 \\ n_0 &= 1 \end{aligned}$$

satisfy the definition of big-O notation for $g(n) = n^2$ in that

$$T(n) = 1 + n^2 \leq 2 \cdot n^2 \text{ for } n \geq 1$$

Therefore,

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

6.3.2 Acceptable answer

The answer in the prior section is more detailed than required to receive full credit. The following is sufficient.

```
k = 5                   # (1)  1 step
for i in range(n):      # (2)  n * (lines (3) and (4))
    for j in range(n):  # (3)  n * (line (4))
        k += 2          # (4)  1 step
```

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

$$T(n) = 1 + n \cdot n \cdot 1$$

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

The following are no longer example questions. Answer each of the following in a manner meeting the example questions and answers on prior pages.

7 Problem 1

(10 points divided 2, 2, 3, 3 bewteen (a) through (d) respectively)

Analyze the time complexity of each of the following code snippets. Use the example problems as guide as to what constitutes a sufficient answer.

(a)
```
x *= 2
```

(b)
```
x = 5
y = x + 1
```

(c)
```
x = 0
for i in range(n):
    i *= 2
```

(d)
```
def g(y: int) -> None:
  return y * 2

def f(x: int) -> None:
  for j in range(x):
    _ = g(x)     # Following convention that _ means "ignore this variable."

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

8 Problem 2

(20 points divided 2, 2, 2, 3, 3, 3, 5 bewteen (a)-(g) respectively)

Now we introduce Python's built-in list. For full credit, when specifying big-O notation include the word "amortized" when the cost is known to have an amortized cost that exceeds the $g(n)$ when stating that a function $f(n) = O(g(n))$. For example, appending to a Python list is amortized $O(1)$. When not considering amortized costs, the actual worst case performance for an append to a python list is $O(n)$ because the need to copy all n items when the allocated array holding the n items is full.

When a problem has n or m, but these variables have no assigned value within the code snipet, assume that they have been set before this code snippet.

Assume the following has been executed near the top of the file containing these code snippets:

```
from random import randint
from bisect import bisect_left
```

(a) (2 points)

```
x = []
x.append(5)
```

(b) (2 points)

```
x = []
for i in range(n):
    x.append(i)
```

(c) (2 points)

```
x = [randint(0, 1000) for x in range(n)]
```

(d) (3 points) express the time complexity as a function of n and m. x is an array containing n integers. $m < n$. (3 points)

```
for _ in range(m):
  i = randint(0, len(x)-1)
  v = x[i]
  x.remove(v)
```

(e) (3 points)

```
x = []
for _ in range(n):
  i = randint(0, len(x)-1)
  x.insert(i, randint(0, 100000))
```

(f) (3 points)

```
n = len(x)
for i in range(n):
   i = randint(0, len(x)-1)
   x[i] = x[i] + 1
```

(g) n and k are integers. $k < n$. Express the time complexity as a function of k and n. (5 points)

```
x = []
i = 0
for _ in range(n):
    i += randint(0, 10)
    x.append(i)

print(f"inserted {n} items into list x in increasing order.  Max value is {i}.")
```

```
found = 0
for _ in range(k):
    y = randint(0, i)
    j = bisect_left(x, y)
    if j < len(x) and x[j] == y:
        found += 1
```

9 Problem 3

(10 points divided 3, 3, 4)

Now we add dicts. Analyze the following code snippets. Use same assumptions as in problem 3. dicts

(a) (3 points)

```
d = {}
for i in range(n):
  j = randint(0, len(d)-1)
  d[i] = j
```

(b) In this problem the first line contains a list comprehension that creates a list of n key-value pairs. This step takes $O(n)$. Express the time complexity in terms of n and m. (3 points)

```
kv = [(randint(0, 10000), randint(0, 10000)) for _ in range(n]
d = dict(kv)
x = 0
for i in range(m):
   i = randint(0, n-1)
   x += d[kv[i]]
```

(c) Express time complexity in terms of n and m. (4 points)

```
k = 0
kv = []
for _ in range(n):
    k += randint(0, 10)
    v = randint(0, 100000)
    kv.append( (k, v) )

d = dict(kv)
for i in range(m):
   k, v = kv.pop()
   del d[k]
```

10 Problem 4

(10 points)

Now we consider queues.

We implement a queue using a list as shown in Figure 1. What is the time complexity of the following code:

(a) (3 points)

```
q = Queue()
for i in range(n):
    q.enqueue(randint(0, 1000))
for i in range(n):
    _ = q.dequeue()
```

(b) (3 points)

```
q = Queue()
for i in range(n):
    q.enqueue(randint(0, 1000)
    _ = q.dequeue()
```

Now consider a Queue implemented usinq a singly-linked list as shown in Figures 2 and 3. These implementations can also be found in the repository at `csci-356/lecture15to17/sll_queue.py` and `csci-356/lecture15to17/singly_linked_list.`

(c) (4 points)

```
q = SLLQueue()
for i in range(n):
    q.enqueue(randint(0, 1000))
for i in range(n):
    _ = q.dequeue()
```

11 Problem 5

(10 points divided 5, 5).

Now we consider recursion.

(a) What is the final output and what is the time complexity as a function of n. Although n is assigned the constant 512, express the time complexity as if n is not a constant. n is set to 512 simply to allow you to specify the final output. (5 points)

```
class Queue:
  """
  A Queue implemented by wrapping a Python list.

  """

  def __init__(self):
      self.items = []

  def is_empty(self) -> bool:   # renamed to be compliant with PEP 8 conventions.
      """
      :return: whether the Queue is empty.
      """
      return self.items == []

  def enqueue(self, item) -> None:
      """
      Enqueue the passed item to the end of the queue.

      :param item: item to be enqueued.
      :return: None
      """
      self.items.insert(0,item)

  def dequeue(self) -> object:
      """
      Remove the first item in the queue.
      :return: the item removed from the front of the queue.
      :raises IndexError: if the Queue is empty when dequeue is called.
      """
      return self.items.pop()

  def __len__(self):
      """
      :return: the number of items in the Queue.
      """
      return len(self.items)
```

Figure 1: Implementation of a Queue provided for HW3 p4.

```
from singly_linked_list import SinglyLinkedList

class SLLQueue:
    """Implementation of a Queue based on Python's list."""

    def __init__(self):
        self._items = SinglyLinkedList()

    def is_empty(self):
        return len(self._items) == 0

    def enqueue(self, item):
        # O(n) operation.
        self._items.push_back(item)

    def dequeue(self):
        return self._items.pop_front()

    def __len__(self):
        return len(self._items)
```

Figure 2: Queue implementation based on a singly-linked list. This was covered somewhere in lectures 15 to 17, and can be found in the directory csci-356/lecture15to17.

```
class SinglyLinkedList:
  """Singly-linked list."""
  class _Node:
      def __init__(self, item: object):
          self._next = None
          self._item = item

  def __init__(self):
      """Constructor of the SinglyLinkedList."""
      self._front = None
      self._end = None
      self._n = 0

  def push_back(self, x) -> None:
      """Pushes the item x to the rear of the linked list."""
      if self._front is None:
          assert self._end is None
          node = SinglyLinkedList._Node(x)
          self._front = self._end = node
          self._n = 1
      else:
          node = SinglyLinkedList._Node(x)
          self._end._next = node
          self._end = node
          self._n += 1

  def __len__(self) -> int:
      return self._n

  def pop_front(self) -> object:
      if self._n == 0:
          raise IndexError("Cannot pop from an empty linked list.")

      node = self._front
      self._front = node._next
      if self._front is None:
          self._end = None

      item = noded._item
      self._n -= 1
      del node
      return item
```

Figure 3: Implementation of singly-linked list. This also appears in the repository in csci-356/lecture15to17. I omitted some of the comments to fit the implementation on a single page.

```
n = 512
def f(n):
    if n <= 1:
       return [n]
    return f(n/2).extend(n)

print(f(n))
```

(b) What is the final output and what is the time complexity of `smallest_factor()` as a function of n.

```
def smallest_factor(n, i=2):
    if i * i > n:
        return n

    # If i is a factor of n, return i
    if n % i == 0:
        return i

    return smallestFactor(n, i + 1)

# Example usage
print(smallest_factor(49))
print(smallest_factor(37))
```

12 Problem 6

(10 points)

Write a function factorial(n) that uses recursion to calculate the factorial of a given positive integer n. The factorial of a number n is the product of all positive integers less than or equal to n. It is denoted by n! and is defined as:

$$n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1$$

By definition, $0! = 1$

Your function should:

Check if n is a non-negative integer. If not, the function should return an error message or handle the case appropriately. Implement the base case: if n is 0, return 1.

13 Problem 7

(10 points)

Now we consider sorting. Given a n numbers that all reside in the $[1, 100]$ where $n >> 100$. Write a code snippet to sort them in $O(n)$ time using python

`sorted`. What is the time complexity? Now write a sort that can sort these in $O(n)$ time. Hint: the fact that the range is bounded and much less than n matters.

14 Problem 8

(10 points divided 5 and 5)

Given two sorted Python lists `arr1` and `arr2` of lengths n and m respectively, write a function to find all the elements that are common in both lists. Implement two solutions:

(a) using python binary search (see python's `bisect` module). For each element in arr1, use binary search to check if it exists in arr2. What is the time complexity as a function of n and m? (5 points)

(b) using a python dict (i.e., a hash table) Create a hash table from one array, then iterate through the other array to check for common elements. What is the time complexity as a function of n and m? (5 points)

15 Problem 9

(10 points divided 5 and 5)

Given a list of n integers. We wish to find the k^{th} smallest.

(a) Write a function to find the the kth smallest value using python's `sorted`. What is its time complexity as a function of n and k? (5 points)

(b) Write a function to find the kth smallest value using a binary min heap. What is its time complexity as a function of n and k? (5 points)