CSCI 356 Answers Homework 5: Bonus

Due: Wednesday, November 29, 11:59 PM

1 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
```

Answer 1(a)

```
x *= 2  # 1 step
```

$$T = 2$$

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

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

Answer 1(b)

```
x = 5       # 1 step
y = x + 1   # 1 step (add) + 1 step (assign)
```

$$T = 3$$

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

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

Answer 1(c)

```
x = 0                 # (1)  1 step
for i in range(n):    # (2)  n * (3)
    i *= 2            # (3)  1 step
```

$$T = 1 + n * 1 = 1 + n$$

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

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

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

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

Answer 1(d)

```
def g(y: int) -> None:
  return y * 2          # (1) 1 mult + 1 pop activation record

def f(x: int) -> None:
  for j in range(x):    # (2) n*(3) because x=n
    _ = g(x)            # (3) T_g + 1 + 1 push activation record

for i in range(n):      # (4) n * (5)
  f(n)                  # (5) T_f + 1 push activation record
```

Let T_g denote the time to execute function `g(y)`.

Let T_f denote the time to execute function `f(n)`.

$$\begin{aligned} T(n) &= n \cdot T_f(n) + 1 && (1)\\ T_f(n) &= n \cdot (T_g(n) + 2) && (2)\\ T_g(n) &= 2 && (3)\\ & && (4) \end{aligned}$$

Substituting Equation 3 into Equation 2 yields

$$T_f(n) = 4n$$

We then substitute this in to Equation 1 to get

$$T(n) = 4n^2 + 1$$

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

2 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 extra 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 snippet, 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 + 1 extra for amortized)

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

Answer 2(a)

```
x = []          # 1 step
x.append(5)     # amortized O(1)
```

$$T = 1 + \text{ amortized } O(1)$$

Amortized $O(1)$ is bigger than 1 so we can ignore the 1.

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

(b) (2 points)

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

Answer 2(b)

```
x = []               # (1)  1 step
for i in range(n):   # (2)  n * (3)
    x.append(i)      # (3)  amortized O(1)
```

$$T(n) = 1 + n \cdot \text{ amortized } O(1)$$

Amortized $O(1)$ means a cost was amortized over many operations to obtain an average of $O(1)$. In the case of `append`, the amortization is across n append operations. Amortized $O(1) = O(n)/n$. Thus, $n \cdot$ amortized $O(1) = O(n)$.

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

(c) (2 points)

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

Answer 2(c)

```
x = [randint(0, 1000) for x in range(n)]  # n * O(1)
```

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

(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)
```

Answer 2(d)

```
for _ in range(m):          # (1)  m * (2 to 4)
  i = randint(0, len(x)-1) # (2)  O(1)
  v = x[i]                  # (3)  O(1)
  x.remove(v)               # (4)  O(n)
```

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

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

(e) (3 points)

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

Answer 2(e)

```
x = []                          # (1) 1 step
for _ in range(n):              # (2) n * (3 to 4)
  i = randint(0, len(x)-1)      # (3) O(1) rand + O(1) len
  x.insert(i, randint(0, 100000)) # (4) O(1) + O(n) insert
```

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

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

(f) (3 points)

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

Answer 2(f)

```
n = len(x)                    # (1) O(1) len + 1 assign
for i in range(n):            # (2) n * (3 to 4)
   i = randint(0, len(x)-1)   # (3) O(1) len + O(1) randint + 1 assign
   x[i] = x[i] + 1            # (4) O(1) getitem + 1 + 1 + O(1) setitem
```

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

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

(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
```

2.1 Answer 2(g)

```
x = []                           # (1)  1 assign
i = 0                            # (2)  1 assign
for _ in range(n):               # (3)  n * (4 to 5)
    i += randint(0, 10)          # (4)  O(1) randint + 1 plusassign
    x.append(i)                  # (5)  amortized O(1)

print(f"inserted {n} ...")       # (6)  1 print
found = 0                        # (7)  1 assign
for _ in range(k):               # (8)  k * (9 to 12)
    y = randint(0, i)            # (9)  1 randint
    j = bisect_left(x, y)        # (10) O(log n) bisect + 1 assign
    if j < len(x) and x[j] == y: # (11) 1 less + O(1) len + 1 + 1
        found += 1               # (12) 1 plusassign
```

Let $T_1(n)$ refer to the time complexity for executing lines (1) to (5).

Let $T_2(n)$ refer to the time complexity for executing lines (6) to (12).

$$\begin{aligned} T(n) &= T_1(n) + T_2(n)1 \\ T_1(n) &= 1 + n \cdot (O(1) + 1 + \text{ amortized } O(1)) \\ T_1(n) &= O(n) \\ T_2(n) &= 1 + 1 + k \cdot (1 + O(\log n) + 1 + 1 + O(1) + 1 + 1) \\ T_2(n) &= O(k \log n) \end{aligned}$$

In Equation 6 we drop the "amortized" because the amortization of the n number of amortized $O(1)$ operations is amortized (spread) over the n operations resulting in a total worst case performance of $O(n)$.

$$\boxed{T(n) = O(n + k \log n)}$$

We cannot simplify it further because k can be any value $1 \leq k \leq n$. If k is a constant (i.e., not varying with n) then $O(k \log n)$ is $O(\log n)$. However, when $k = O(n)$, $O(k \log n)$ becomes $O(n \log n)$ which is greater than $O(n)$.

3 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 + 1 extra for "average")

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

Answer 3(a)

```
d = {}                      # (1) 1 assign
for i in range(n):          # (2) n * (3 to 4)
  j = randint(0, len(d)-1)  # (3) 1 rand + 1 len + 1 minus
  d[i] = j                  # (4) 1 assign + avg O(1) setitem
```

$$\begin{aligned} T(n) &= 1 + n \cdot (1 + 1 + 1 + 1 + \text{ average } O(1) \\ &= n \cdot (4 + \text{ average } O(1)) \end{aligned}$$

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

(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 + 1 extra for "average")

```
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]]   # OOPS!  Should be d[kv[i][0]]
```

Answer 3(b)

```
kv = [(randint(0, 10000), randint(0, 10000)) for _ in range(n] # (1)
d = dict(kv)           # (2) average O(len(kv)) = average O(n)
x = 0                  # (3) 1 assign
for i in range(m):     # (4) m * (5 to 6)
   i = randint(0, n-1) # (5) 1 minus + O(1) randint + 1 assign
   x += d[kv[i][0]]    # (6) 1 getitem + average O(1) setitem + 1
```

The first line requires $O(n)$ time to populate the key-value pairs. Let T_{for} be the time to execute the for loop in lines (4)-(6).

$$\begin{aligned} T(n) &= O(n) + \text{ average } O(n) + 1 + T_{for}(n) \\ T_{for}(n) &= m \cdot (1 + O(1) + 1 + 1 + \text{ average } O(1) + 1) \\ &= \text{ average } O(m) \\ T(n) &= O(n) + \text{ average } O(n) + \text{ average } O(m) \end{aligned}$$

For any given n, "average $O(n)$" could exceed $O(n)$, as such we can treat $O(n)$ as smaller than "average $O(n)$" and thus we drop the $O(n)$ but keep "average $O(n)$."

$$\boxed{T(n) = \text{ average } O(n+m)}$$

(c) Express time complexity in terms of n and m. (4 points + 1 extra for "average")

```
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]
```

Answer 3(c)

```
k = 0                      # (1) 1 step
kv = []                    # (2) 1 assign
for _ in range(n):         # (3) n * (4 to 6)
    k += randint(0, 10)    # (4) O(1) randint + 1 plusassign
    v = randint(0, 100000) # (5) O(1) randint + 1 assign
    kv.append( (k, v) )    # (6) amortized O(1) append
```

```
d = dict(kv)                # (7) average O(n) copy.
for i in range(m):          # (8) m * (9 to 11)
   k, v = kv.pop()          # (9) amortized O(1) pop + 1 assign
   del d[k]                 # (10) average O(1) delete.
```

Let $T_1(n)$ refer to the time complexity to execute steps (1)-(6).

Let $T_2(n)$ refer to the time complexity to execute steps (7)-(10).

$$\begin{aligned}
T(n) &= T_1(n) + T_2(n) && (6)\\
T_1(n) &= 2 + n \cdot (O(1) + 1 + O(1) + 1 + \text{ amortized } O(1)) && (7)\\
&= n \cdot \text{ amortized } O(1) && (8)\\
&= O(n) && (9)\\
T_2(n) &= \text{ average } O(n) + m \cdot (\text{ amortized } O(1) + 1 + \text{ average } O(1)) && (10)\\
&= \text{ average } O(n) + \text{ amortized } O(m) + \text{ average } O(m) && (11)
\end{aligned}$$

In Equation 9 the qualifier "amortized" is dropped, because the "amortized" part of the $O(1)$ operations has been amortized over the n operations resulting in a worst-case performance of $O(n)$. NOTE: The amortized $O(1)$ comes from taking $O(n)$ and dividing by n. Thus,

$$\begin{aligned}
\text{amortized } O(1) &= \frac{O(n)}{n}\\
n \cdot \text{ amortized } O(1) &= O(n)
\end{aligned}$$

$$\begin{aligned}
T(n) &= O(n) + \text{ average } O(n) + \text{ amortized } O(m) + \text{ average } O(m)\\
&= \text{ average } O(m+n) + \text{ amortized } O(m)
\end{aligned}$$

Because "average" is a weaker guarantee than "amortized" we drop amortized.

$$\boxed{T(n) = \text{ average } O(m+n)} \quad (12)$$

4 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()
```

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

  """

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

  def is_empty(self) -> bool:
      """
      :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 Queue empty.
      """
      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.

Answer 4(a)

```
q = Queue()                         # (1) O(1)
for i in range(n):                  # (2) n * (3)
    q.enqueue(randint(0, 1000))     # (3) 1 randint + O(n)
for i in range(n):                  # (4) n * (5)
    _ = q.dequeue()                 # (5) O(1)
```

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

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

(b) (3 points)

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

Answer 4(b)

```
q = Queue()                         # (1) O(1)
for i in range(n):                  # (2) n * (3 to 4)
    q.enqueue(randint(0, 1000))     # (3) 1 randint + O(1)
    _ = q.dequeue()                 # (4) O(1)
```

Note that in steps (3) and (4), the queue never grows beyond 1 element at any given time. The enqueue always occurs on an empty queue, as such there are no elements to shift right causing the enqueue to become an $O(n)$ operation.

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

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

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.py`

(c) (4 points)

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

```
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.

Answer 4(c)

```
q = SLLQueue()                  # (1) O(1)
for i in range(n):              # (2) n * (3)
    q.enqueue(randint(0, 1000)) # (3) O(1) randint + O(1)
for i in range(n):              # (4) n * (5)
    _ = q.dequeue()             # (5) O(1)
```

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

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

5 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 +1 extra for "amortized")

```
n = 512
def f(n):
    if n <= 1:
       return [n]
```

```
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("pop from 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.

```
        return f(n/2).extend(n)  # OOPS! should be append(n).

    print(f(n))
```

Answer 5(a)

```
n = 512                     # (1) 1 assign
def f(n):
    if n <= 1:              # (2) 1 compare
       return [n]           # (3) O(1) + 1 pop activation record.
    return f(n/2).append(n) # (4) T_f(n/2) + amortized O(1) + 1

print(f(n))                 # (5) T_f(n) + 1 push + 1 print.
```

$$\begin{aligned} T(n) &= 1 + T_f(n) + 1 \\ T_f(1) &= 1 + 1 + O(1) \\ T_f(1) &= O(1) \\ T_f(n) &= 1 + T_f(n/2) + \text{ amortized } O(1) \end{aligned}$$

$f(n)$ gets called $\log_2 n$ times before terminating.

$$T(n) = 2 + (\log_2 n) \cdot (1 + O(1) + \text{ amortized } O(1)$$

$$\boxed{T(n) = \text{ amortized } \log n}$$

Output:

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

(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))
```

Answer 5(b)

```
def smallest_factor(n, i=2):
    if i * i > n:                    # (1) 1 comp + 1 mult
        return n                     # (2) 1 pop act rec

    # If i is a factor of n, return i
    if n % i == 0:                   # (3) 1 mod + 1 comp
        return i                     # (4) 1 pop act rec

    return smallestFactor(n, i + 1) # (5) T_{sf}(n,i+1) + 1 push

# Example usage
print(smallest_factor(49))           # (6) T_{sf} + 1 print
print(smallest_factor(37))           # (7) T_{sf} + 1 print
```

$$\begin{aligned} T(n,2) &= 2 \cdot T_{sf}(n,2) + 2 \\ T_{sf}(n,2) &= 1+1+1+1+T_{sf}(n,3)+1 \\ T_{sf}(n,2) &= 5 + T_{sf}(n,3) \\ T_{sf}(n,i) &= 5 + T_{sf}(n,i+1) \\ T_{sf}(n,\sqrt{n}) &= 3 \end{aligned}$$

$T_{sf}(n,\sqrt{n})$ is the terminating condition.

With each recursion, i increases by 1 until it reaches the $\sqrt{n}$. Since starts at 2, $T_{sf}(n,2)$ becomes

$$\begin{aligned} T_{sf}(n,2) &= (\sqrt{n}-2)\cdot 5 + T_{sf}(n, sqrtn) \\ &= 5\sqrt{n} - 10 + 3 \\ T(n,2) &= 2 \cdot (5\sqrt{n} - 7) \end{aligned}$$

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

6 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.

Answer 6

For full credit either return an error message or raise an exception with an error message as in below:

```
def factorial(n):
    if n < 0:
        raise Error("n must be nonnegative")
    if n ==1 or n == 0:
        return 1
    return n * factorial(n-1)
```

7 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.

NOTE: Ignore the part using `sorted` because I wrote it incorrectly. The intension was that the second sentence would read "Write a code snippet to sort them using python sorted." Then the subsequent sentendce "What is the time complexity" makes sense.

Answer 7

```
x = sorted(x)
```

This has time complexity $O(n \log n)$.

The trick here is that the numbers are constrained to the range $[1, 100]$. As such we can create a table *cnts* with 101 elements so that the number of occurrences of any number in $i \in [1, 100]$ can be tallied by incrementing $cnts[i]$.

Here is a sort with linear time complexity:

```
def linear_sort(x: list) -> list:
    n = len(x)              # (1) O(1) + 1 assign
    result = [None] * n     # (2) O(n)
    cnts = [0] * 101        # (3) O(101) = O(1)
    for i in x:             # (4) n * (5)
        cnts[i] += 1        # (5) 1 plus + 1 setitem

    k = 0                   # (6) 1 assign
    for j in range(101):    # (7) 101 * (8 to 11)
        while cnts[j] > 0:  # (8) ? * (9 to 11)
            result[k] = j   # (9) 1 setitem
            k += 1          # (10) 1 increment
            cnts[j] -= 1    # (11) 1 minusequal + 1 setitem
    return result           # (12) O(1) pop activation record
```

Let T_{for} denote the time to execute lines (6) through (11). Let T_{while} denote the time to execute line (8) through (11).

$$\begin{aligned} T(n) &= O(1) + 1 + O(n) + O(101) + n \cdot (1 + 1) + T_{for}(n) + O(n) \\ &= O(1) + O(n) + 2n + T_{for}(n) \\ &= O(n) + T_{for}(n) \end{aligned}$$

The sum of all values in the `cnts` array is equal to n since we performed one increment in step (5) for each value in x. Each iteration through the `while` loop results in one value in the `cnts` array being decremented by 1. As such, the while loop will execute exactly n times across all iterations through the `for` loop. Thus, we can state

$$T_{while}(n) = n \cdot (1 + 1 + 1 + 1) = 4n$$

$$T_{for}(n) = 101 \cdot (1 + 1) + 4n = 202 + 4n \tag{13}$$

For large n, the $4n$ term dominates so

$$T(n) = O(n) + 202 + 4n$$

For large n, the 202 is ignorable.

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

8 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)

Answer 8(a)

```
common = []                     # (1) 1 assign
for x in arr1:                  # (2) n * (3 to 5)
    i = bisect_left(arr2, x)    # (3) O(log(m))
    if arr2[i] == x:            # (4) 1 comp + 1 getitem
        common.append(x)        # (5) amortized O(1)
```

$$T(n, m) = 1 + n \cdot \Big(O(\log m) + 2 + \text{ amortized } O(1) \Big)$$

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

(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 + 1 extra for "average")

Answer 8(b)

It would make more sense to implement this using a verb—set— rather than a `dict`, but the problem calls for using a dict.

```
common = []              # (1) 1 assign
d = {}                   # (2) 1 assign
for x in arr1:           # (3) n1 * (4)
    d[x] = 1             # (4) average O(1) setitem
for x in arr2:           # (5) n2  * (6 to 7)
    if x in d:           # (6) average O(1) \in
        common.append(x) # (7) amortized O(1)
```

$$T(n) = 1 + 1 + n \cdot \text{ average } O(1) + m \cdot (\text{ average } O(1) + \text{ amortized } O(1))$$

"average $O(1)$" is a looser constraint than "amortized $O(1)$" so we can ignore the "amortized $O(1)$" yielding

$$\boxed{T(n) = \text{ average } O(n+m)} \tag{14}$$

9 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)

Answer 9(a)

```
def kth_smallest(x: list, k: int) -> int:
    return sorted(x)[k]  # O(n log n) sorted + 1 getitem
```

$$\boxed{T(n) = O(n \log n)} \tag{15}$$

(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)

Answer 9(b)

```
from heapq import heapify, heappop

def kth_smallest(x:list, k: int) -> int:
    """k uses the convention of starting
       from 0.  Therefore k=0 refers to the smallest."""
    if k < 0:
        raise IndexError("k must be nonnegative.")
    if len(x) == 0:
        raise ValueError("x must be a non-zero length list.")
    heapify(x)              # (1) O(n)
    y = None                # (2) O(1)
    for i in range(k+1):    # (3) (k+1) * (4)
        y = heappop(x)      # (4) log(n)
    return y                # (5) 1 pop activation record
```

$$T(n,k) = O(n) + O(1) + (k+1)\log n + 1$$

$$\boxed{T(n,k) = O(n + k \log n)} \tag{16}$$