Homework 2 solutions

Problem 1 Give the Big-O performance of the following code fragment:

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

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

Let T_3 denote the time to execute step (3). Let T_2 denote the time to execute steps (2) and (3). Let T_1 denote the time to execute steps (1), (2), and (3). Since T_1 contains the time to execute all of the steps, $T = T_1$.

T_3 takes at least two steps. The addition and the assignment. In machine langauge, this may take additional steps to read and write the result from memory. We cannot know exactly how many machine language instructions this code may compile down to, because that will depend on the specifics of the CPU architecture. We can however know that the number of instructions will be bounded by some constant. It will never take 1000 instructions to implement step (3) on any existing processor. As such, we can say there exists a constant C_3 number of steps for which

$$T_3(n) \leq C_3 \tag{1}$$

The definition of big O states

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

if and only if there exists some C and n_0 such that $f(n) \leq C \cdot g(n)$ for all $n > n_0$.

$$T_3(n) = O(1) \tag{3}$$

because $T_3(n) \leq C_3 \cdot 1$ regardless of n, because we choose to set $C = C_3$.

Because step (2) repeats step (3) n times,

$$T_2(n) = n \cdot T_3(n) \tag{4}$$

Because step (1) repeats step (2) and (3) n times,

$$T(n) = T_1(n) = n \cdot T_2(n) = n^2 T_3(n) = C \cdot n^2 \tag{5}$$

From the definition of big-O,

$$\boxed{T(n) = O(n^2)} \tag{6}$$

NOTE: We can also say $T(n) = O(n^3)$ and $T(n) = O(2^n)$, because big-O notation establishes an upper bound. However, when we ask for the big-O of a particular code fragment, we want the tightest upper bound. You will not get full credit if you said that the time complexity of the code fragment is $O(n^3)$.

Problem 2 Give the Big-O performance of the following code fragment:

```
for i in range(n):    # (1)  n * (step (2))
    k = 2 + 2         # (2)  O(1)
```

The time complexity of the code above becomes

$$T(n) = n \cdot O(1) \tag{7}$$

so

$$\boxed{T(n) = O(n)} \tag{8}$$

Problem 3 Give the Big-O performance of the following code fragment:

```
i = n             # (1)  O(1)
while i > 0:      # (2)  log_2(n) * (steps (3) and (4))
    k = 2 + 2     # (3)  O(1)
    i = i // 2    # (4)  O(1)
```

Let $T(n)$ denote th enumber of steps to execute the code fragement above.

Because step (4) divides i by 2 on each iteration, this causes the i reach 0 in $log_2(n) + 1$ divisions. Thus the time complexity of the code above in big-O notation becomes

$$T(n) = O(1) + (log_2(n) + 1) \cdot (O(1) + O(1)) \tag{9}$$

Because $log_2(n)$ grows with n while the $O(1)$ terms do not, the $log_2(n)$ becomes the dominant term as n grows. Thus Forumula 9 becomes

$$T(n) = O(log_2(n)) \tag{10}$$

It so happens that we can convert between bases by multiplying by a constant. Remember back to your math, the log base change rule states

$$log_b(a) = \frac{log_c(a)}{log_c(b)} \tag{11}$$

As such,

$$log_k(n) = \frac{log_2(n)}{log_2(k)} \tag{12}$$

Note that $\log_2(k)$ is a constant. We can thus define a constant $C = \frac{1}{log_2(k)}$ such that Equation 11 becomes

$$log_k(n) = Clog_2(n) \tag{13}$$

This means

$$O(log_2(n)) = O(log_3(n)) = O(log_4(n)) = ... \tag{14}$$

Since I can do this for any k, we often drop the base and state

$$O(log_2(n)) = O(log(n)) \tag{15}$$

so

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

Problem 4 Give the Big-O performance of the follwoing code fragment:

```
for i in range(n):          # (1)  n * (steps (2) through (4))
   for j in range(n):       # (2)  n * (steps (3) through (4))
       for k in range(n):   # (3)  n * (step (4))
           k = 2 + 2        # (4)  O(1)
```

Due to the triple nesting of for loops the code block above has time complexity give by

$$n \cdot n \cdot n \cdot O(1) = O(n^3) \tag{17}$$

Problem 5: (2 points) The Bag class from lectures 6 and 7 can be found in the class repository at

`https://git.cs.olemiss.edu/harrison/csci-356`

in

`lectures6and7/bag/bag.py`

Add an iterator class to Bag. The iterator class must pass the unit tests committed in the repository in the directory `hw2/bag/test_bag.py`. You will receive *partial credit if you do not write unit tests for your iterator class, or if the code lacks comments or type hints.* Write to test more conditions than the tests given in `test_bag.py`. The tests should cover edge conditions like the iterator should work as expected on an empty list.

Answer 5

See hw2/bag/bag.py

These should pass the tests in hw2/test_bag.py as well as the additional tests in hw2/test_bag_extended.py.

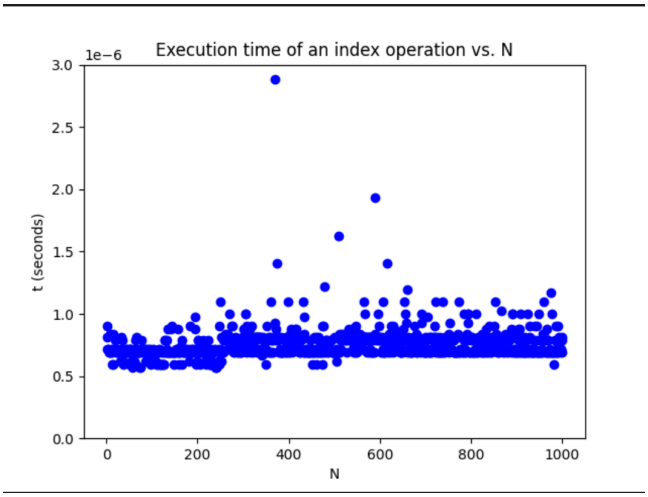
The additional tests just test a couple edge cases.

Problem 6: (2 points) Write a program that verifies that the list index operator is O(1). The program must plot the run time of the list index operator as a function of n using matplotlib.

Answer 6

See p6.py

Possible plot:



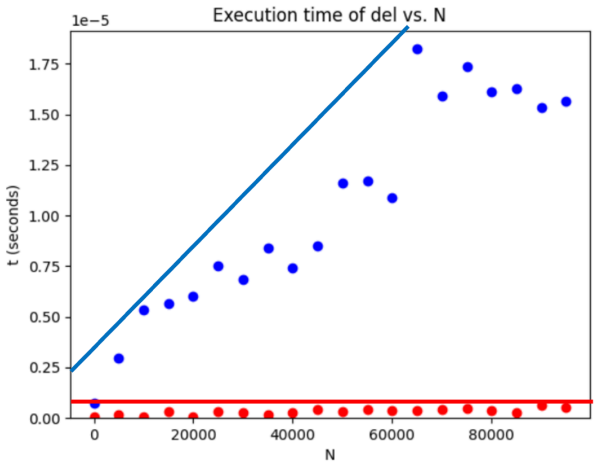
In this plot, there doesn't appear to be any correlation between n and the time to execute an index operation. This would be the case if the operation takes $O(1)$ time.

Problem 7: (2 points) Write a program that compares the performance of the `del` operator on lists and dictionaries. The main program should plot the run time of each on the same plot as a function of n. Also plot functions that bound the time complexity and print out what you think is the time complexity of

`del` operators for lists and dictionaries using big-O notation. When measuring performance on the list `del` operator, be sure to delete items at random locations from the list.

Answer 7

See p7.py



Red denotes average execution time per delete from a dict containing n items. Blue denotes the same thing but for lists.

From this it looks like the time complexity of deletion from a dict is O(1), but deletion from a list is O(n).