
DSC 40B - Quiz 02

Name:

Student ID:

About the quizzes:

- Quizzes in DSC 40B are *optional* and graded pass/fail.
- A score of 70% or higher earns a “pass” and 1.5 credits toward your final grade.
- If you don’t pass, no credits are earned, but it doesn’t hurt your grade.
- You have 30 minutes to complete the quiz.
- Unfortunately, we can’t answer clarifying questions during the quiz. If you think a question has a bug or is unclear, please let us know in a private post on Campuswire after the quiz, and we’ll take it into account when grading.

Problem 1.

Let

$$f(n) = \frac{n^4 - 3n^2 + 7}{\sqrt{n} + 5} \cdot \frac{\log n}{n^2 + 10n}.$$

What is $f(n)$ in asymptotic notation, in simplest terms?

- ☐ $\Theta(n^{1.5})$
 ☒ $\Theta(n^{1.5} \log n)$
 ☐ $\Theta(n^2 \log n)$
 ☐ $\Theta(n^{2.5} \log n)$
 ☐ $\Theta(n^4 \log n)$

Solution: $\Theta(n^{1.5} \log n)$.

Keep only the dominant term in each numerator and denominator. The first fraction is $\Theta(n^4/\sqrt{n}) = \Theta(n^{3.5})$, and the second is $\Theta\left(\frac{\log n}{n^2}\right)$. Multiplying them gives

$$\Theta\left(\frac{n^{3.5} \cdot \log n}{n^2}\right) = \Theta(n^{1.5} \log n).$$

This problem was similar to Problem #048 in the Quiz 02 practice problems.

Problem 2.

Let $f(n) = 3n^2 \log n + 50n$. True or False: $f(n) = O(n^3)$.

- ☒ True
☐ False

Solution: True.

$f(n) = \Theta(n^2 \log n)$, since $n^2 \log n$ grows faster than n . Because $\log n$ grows more slowly than n , $n^2 \log n$ grows more slowly than $n^2 \cdot n = n^3$. So n^3 is an upper bound on f (just not a tight one). Remember that O only says that f grows *no faster* than n^3 .

If you responded with False, you may have been thinking of O as a *tight* bound. Remember, O is just an *upper* bound, and n^3 does grow faster than $f(n)$, so it is a valid upper bound.

This problem was similar to Problem #032 in the Quiz 02 practice problems.

Problem 3.

Consider the function

$$f(n) = \begin{cases} n^3, & \text{if } n \text{ is even;} \\ n^2, & \text{if } n \text{ is odd.} \end{cases}$$

True or False: $f(n) = \Theta(n^3)$.

- ☐ True
☒ False

Solution: False.

$f(n) \leq n^3$ for every n , so $f(n) = O(n^3)$. But f is not $\Omega(n^3)$: for that, we'd need a positive constant c with $f(n) \geq cn^3$ for all large n . Whatever c we pick, on odd values of n we have $f(n) = n^2$, and $n^2 < cn^3$ once $n > 1/c$. So f keeps dipping below cn^3 no matter how large n gets.

The tightest bounds we can give are $f(n) = O(n^3)$ and $f(n) = \Omega(n^2)$.

If you responded with True, you may have looked only at the larger of the two cases. For $f(n) = \Theta(n^3)$, f has to be bounded above *and* below by constant multiples of n^3 for all large n , and on odd n , $f(n) = n^2$ falls below every cn^3 . In other words, to be $\Theta(n^3)$, f has to be sandwiched between two copies of n^3 ; the upper part of the sandwich is fine, but we can never find a c so that cn^3 works as the lower part of the sandwich.

This problem was similar to Problem #054 in the Quiz 02 practice problems.

Problem 4.

Suppose `bar` and `baz` are two functions. `bar` takes $\Theta(n^3)$ time on an input of size n , while `baz` takes $\Theta(n)$ time on an input of size n . What is the time complexity of `foo`, below?

```
def foo(n):  
    if n < 10_000:  
        bar(n)  
    else:  
        baz(n**2)
```

☐ $\Theta(1)$ ☐ $\Theta(n)$ ☒ $\Theta(n^2)$ ☐ $\Theta(n^3)$ ☐ $\Theta(n^6)$

Solution: $\Theta(n^2)$.

Asymptotic time complexity describes the time taken when n is large, so it doesn't matter what happens when $n < 10,000$. When $n \geq 10,000$, `foo` calls `baz` on an input of size n^2 . Since `baz` takes linear time in the size of its input, this takes $\Theta(n^2)$ time.

If you responded with $\Theta(n^3)$, you may have used the time taken by `bar`. But `bar` is only called when $n < 10,000$, and asymptotic time complexity only describes what happens when n is large. In fact, you could change `bar`'s time complexity to be *anything you want* and it wouldn't affect the answer.

If you responded with $\Theta(n)$, you may have forgotten that `baz` is called on an input of size n^2 , not n . Since `baz` takes linear time in the size of its input, that's $\Theta(n^2)$ time.

This problem was similar to Problem #043 in the Quiz 02 practice problems.

Problem 5.

Suppose $f_1(n)$ is both $O(n^2)$ and $\Omega(n)$, and suppose $f_2(n) = \Theta(n^3)$. What are the tightest possible bounds on the product $f_1(n) \cdot f_2(n)$?

☐ $\Theta(n^4)$ ☐ $\Theta(n^5)$ ☒ $O(n^5)$ and $\Omega(n^4)$ ☐ $O(n^6)$ and $\Omega(n^3)$

Solution: $O(n^5)$ and $\Omega(n^4)$.

For the upper bound, f_1 is at most a constant times n^2 and f_2 is at most a constant times n^3 (for large n), so their product is at most a constant times n^5 . For the lower bound, f_1 is at least a constant times n and f_2 is at least a constant times n^3 , so the product is at least a constant times n^4 .

We can't say Θ of anything, because both bounds can happen: if $f_1(n) = n^2$, the product is $\Theta(n^5)$, and if $f_1(n) = n$, it is $\Theta(n^4)$.

This problem was similar to Problem #061 in the Quiz 02 practice problems.

Problem 6.

Suppose Algorithm A takes $\Theta(n^2)$ time, while Algorithm B takes $\Theta(n^3)$ time. True or False: on every input of size $n = 1,000$, Algorithm A takes less time than Algorithm B.

- ☐ True
☒ False

Solution: False.

Asymptotic notation hides constants, and it only describes how the time grows when n is large. For example, Algorithm A could take $10,000n^2$ seconds while Algorithm B takes n^3 seconds. On an input of size 1,000, A takes 10^{10} seconds, while B takes only 10^9 seconds. A is only guaranteed to be faster *eventually*, once n is large enough.

This problem was similar to Problem #039 in the Quiz 02 practice problems.

Problem 7.

What is the **best case** time complexity of the following function?

```
def foo(arr):  
    """`arr` is a list containing n numbers."""  
    n = len(arr)  
    for i in range(n):  
        if arr[i] == max(arr):  
            for j in range(n):  
                print(arr[i] - arr[j])  
            return
```

- ☐ $\Theta(1)$ ☐ $\Theta(\log n)$ ☒ $\Theta(n)$ ☐ $\Theta(n \log n)$ ☐ $\Theta(n^2)$ ☐ $\Theta(n^3)$

Solution: $\Theta(n)$.

The best case is when the maximum is the first element of the array. Then, on the very first iteration of the outer loop, we compute `max(arr)`, which takes $\Theta(n)$ time, find that it equals `arr[0]`, run the inner loop ($\Theta(n)$ time), and return. The total is $\Theta(n) + \Theta(n) = \Theta(n)$.

Note that the best case is not $\Theta(1)$: even when we return on the first iteration, we still have to call `max` once and run the inner loop once.

If you responded with $\Theta(1)$, you may have counted the early `return` but not the work done before it: the call to `max(arr)` and the inner loop both take $\Theta(n)$ time.

If you responded with $\Theta(n^2)$, you may have multiplied the $\Theta(n)$ time of `max(arr)` by the $\Theta(n)$ time of the inner loop. In the best case, they each run once, one after the other, so their times add. $\Theta(n^2)$ would be the time complexity of the *worst* case.

This problem was similar to Problem #084 in the Quiz 02 practice problems.

Problem 8.

What is the **worst case** time complexity of the same function from the previous problem, reproduced below?

```
def foo(arr):  
    """`arr` is a list containing  $n$  numbers."""  
    n = len(arr)  
    for i in range(n):  
        if arr[i] == max(arr):  
            for j in range(n):  
                print(arr[i] - arr[j])  
            return
```

☐ $\Theta(1)$ ☐ $\Theta(\log n)$ ☐ $\Theta(n)$ ☐ $\Theta(n \log n)$ ☒ $\Theta(n^2)$ ☐ $\Theta(n^3)$

Solution: $\Theta(n^2)$.

The worst case is when the (only) maximum is the last element of the array. Then the outer loop runs all n times, and on each iteration it calls `max(arr)`, which takes $\Theta(n)$ time. That's $n \cdot \Theta(n) = \Theta(n^2)$ time. On the last iteration, the inner loop runs once, taking another $\Theta(n)$, for a total of $\Theta(n^2) + \Theta(n) = \Theta(n^2)$.

Even though the inner loop is nested inside the outer loop, it runs at most once, since the function returns right after it. The $\Theta(n^2)$ comes from calling `max` inside the loop.

If you responded with $\Theta(n^3)$, you may have multiplied the $\Theta(n)$ iterations of the outer loop by the $\Theta(n)$ time of `max(arr)` and the $\Theta(n)$ iterations of the inner loop. Remember, we only multiply for independent nested loops, where the inner code runs on each iteration of the outer loop. In this case, the innermost nested loop runs *at most once*, so we add its time instead of multiplying. Moreover, we wouldn't multiply the $\Theta(n)$ time of `max(arr)` by the $\Theta(n)$ iterations of the inner loop – although in a sense the `max(arr)` is iterating over the array, the inner loop is not being executed during `max(arr)`'s loop.

If you responded with $\Theta(n)$, you may have treated `max(arr)` as taking $\Theta(1)$ time. It takes $\Theta(n)$ time, and it's called on every iteration of the outer loop.

This problem was similar to Problem #085 in the Quiz 02 practice problems.

Problem 9.

Suppose you are given an array of n integers that is **sorted** in increasing order. You are tasked with determining whether every element of the array is the same (that is, all of the entries have the same value).

What is the **tight** theoretical lower bound for the worst case time complexity of this problem?

☒ $\Theta(1)$ ☐ $\Theta(\log n)$ ☐ $\Theta(n)$ ☐ $\Theta(n \log n)$

Solution: $\Theta(1)$.

First, observe that there's an algorithm that solves this problem in $\Theta(1)$ time.

Because the array is sorted, every element lies between the first element and the last one. So every element is the same exactly when the first and last elements are equal, and we can answer by checking whether `arr[0] == arr[-1]`. This takes $\Theta(1)$ time in the worst case.

No algorithm can do better than $\Theta(1)$, and this algorithm achieves it, so $\Theta(1)$ is a *tight* lower bound.

You might wonder: what's the tight lower bound if the array is not sorted? First, we have a lower bound of $\Omega(n)$: any algorithm must look at every element in the array to be sure that they are all the same (if it doesn't look at an element, that element could be different from the others and the algorithm

could be incorrect).

Is that a tight lower bound? It is, and to show that, we can come up with an algorithm that solves the problem in $\Theta(n)$ time. The idea is to look at the first element of the array and store it – if all of the entries are to have the same value, they must all be equal to this first element. Then we start looping through the array, checking each element to see if it is equal to the first element. If we find one that is different we can return early, but in the worst case we will have to check each of the other $n - 1$ elements, and so the worst case time complexity is $\Theta(n)$. Since there's an algorithm that has the same time complexity as the lower bound, we conclude that $\Theta(n)$ is a tight lower bound for the worst case time complexity of this problem when the array is not sorted.

If you responded with $\Theta(n)$, you may have found the tight lower bound for an *unsorted* array, where every element must be checked. Because the array is sorted, checking the first and last elements is enough.

This problem was similar to Problem #090 in the Quiz 02 practice problems.

Problem 10.

What is the **expected** time complexity of the following function? You may assume that `math.sqrt` takes $\Theta(1)$ time.

```
import math
import random
def foo(n):
    # draw a number uniformly at random from 0, 1, ..., n-1 in  $\Theta(1)$  time
    x = random.randrange(n)
    if x < math.sqrt(n):
        for i in range(n**2):
            print(i)
    else:
        print('Lucky!')
```

☐ $\Theta(1)$ ☐ $\Theta(\sqrt{n})$ ☐ $\Theta(n)$ ☒ $\Theta(n\sqrt{n})$ ☐ $\Theta(n^2)$

Solution: $\Theta(n\sqrt{n})$.

There are two cases. The loop runs when x is one of $0, 1, \dots, \lceil \sqrt{n} \rceil - 1$, which happens with probability about $\sqrt{n}/n$. In that case, the function takes $\Theta(n^2)$ time. Otherwise, it takes $\Theta(1)$ time. So the expected time is

$$\frac{\sqrt{n}}{n} \cdot \Theta(n^2) + \left(1 - \frac{\sqrt{n}}{n}\right) \cdot \Theta(1) = \Theta(n\sqrt{n}) + \Theta(1) = \Theta(n\sqrt{n}).$$

If you responded with $\Theta(n^2)$, you may have forgotten to multiply the probability of the first case ($\sqrt{n}/n = 1/\sqrt{n}$) by the time it takes in that case ($\Theta(n^2)$).

If you responded with $\Theta(n)$, you may have used $1/n$ as the probability of the first case. The loop runs whenever x is less than $\sqrt{n}$, which happens for about $\sqrt{n}$ of the n possible values of x , so the probability is $\sqrt{n}/n$.

This problem was similar to Problem #098 in the Quiz 02 practice problems.