

---

## DSC 40B - Homework 02

Due: Monday, October 10

---

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 p.m.

### Problem 1.

State the growth of the function below using  $\Theta$  notation, and prove your answer by finding constants which satisfy the definition of  $\Theta$  notation.

$$f(n) = \frac{n^3 - n^2 + n + 1000}{(n-1)(n+2)}$$

**Solution:** First we find the upper bound of the function,

$$f(n) = \frac{n^3 - n^2 + n + 1000}{(n-1)(n-2)} \tag{1}$$

$$\leq \frac{2n^3}{n^2 + (n^2 - 2)} \tag{2}$$

$$\leq 2n \text{ where } n \geq \sqrt{2} \tag{3}$$

Then for the lower bound it is,

$$f(n) = \frac{n^3 - n^2 + n + 1000}{(n-1)(n-2)} \tag{4}$$

$$\geq \frac{n^3}{2n^2} \tag{5}$$

$$= \frac{1}{2}n \tag{6}$$

Therefore we get the final inequality to be,

$$\frac{1}{2}n \leq f(n) \leq 2n$$

Which yields us the time complexity,

$$f(n) = \Theta(n)$$

### Problem 2.

Suppose  $T_1(n), \dots, T_6(n)$  are functions describing the runtime of six algorithms. Furthermore, suppose we

have the following bounds on each function:

$$\begin{aligned}T_1(n) &= \Theta(n^3) \\T_2(n) &= O(n \log n) \\T_3(n) &= \Omega(\log n) \\T_4(n) &= O(n^4) \text{ and } T_4 = \Omega(n^2) \\T_5(n) &= \Theta(n) \\T_6(n) &= \Theta(n \log n) \\T_7(n) &= O(n^{1.5} \log n) \text{ and } T_7 = \Omega(n \log n)\end{aligned}$$

What are the best bounds that can be placed on the following functions?

For this problem, you do not need to show work.

Hint: watch the supplemental lecture at <https://youtu.be/tmR-bIN2qw4>.

**Example:**  $T_1(n) + T_2(n)$ .

**Solution:**  $T_1(n) + T_2(n)$  is  $\Theta(n^3)$ .

a)  $T_1(n) + T_5(n)$

**Solution:**

$$T_1(n) + T_5(n) = \Theta(n^3) + \Theta(n) = \Theta(n^3)$$

b)  $T_2(n) + T_6(n)$

**Solution:** First we get a upper and lower bound,

$$O = T_2(n) + T_6(n) = O(n \log n) + \Theta(n \log n) = O(n \log n)$$

and

$$\Omega = T_2(n) + T_6(n) = O(n \log n) + \Theta(n \log n) = \Omega(n \log n)$$

Therefore we can get  $\Theta$  to be  $\Theta(n \log n)$ .

c)  $T_4(n) + T_5(n)$

**Solution:** With the info given we can deduce an upper and a lower bound

$$O = O(n^4 + n) = O(n^4)$$

$$\Omega = \Omega(n^2 + n) = \Omega(n^2)$$

d)  $T_7(n) + T_4(n)$

**Solution:** With the info given we can get both a upper and a lower bound

$$O = O(n^{1.5} \log n + n^4) = O(n^4)$$

$$\Omega = \Omega(n \log n + n^2) = \Omega(n^2)$$

e)  $T_3(n) + T_1(n)$

**Solution:**

$$O(\infty + n^3) = \text{Undefined}$$

We cannot get an upper bound for this problem as  $T_3(n)$  only has a lower bound, therefore we cannot make any inferences about the upper bound. Therefore we cannot get an upper bound with the information given.

$$\Omega(\log n + n^3) = \Omega(n^3) \quad (7)$$

f)  $T_1(n) \times T_4(n)$

**Solution:** We can get a upper and a lower bound,

$$O = O(n^3 \times n^4) = O(n^7)$$

$$\Omega = \Omega(n^3 \times n^2) = \Omega(n^5)$$

### Problem 3.

In each of the problems below compute the average case time complexity (or expected time) of the given code. State your answer using asymptotic notation. Show your work for this problem by stating what the different cases are, the probability of each case, and how long each case takes. Also show the calculation of the expected time.

a) `def foo(n):`

```
    # randomly choose a number between 0 and n-1 in constant time
    k = np.random.randint(n)

    if k > n/10:
        for i in range(n):
            print(i)
    else:
        print('Never mind...')
```

**Solution:** Throughout the whole question, case 1 will refer to the case where  $k > \frac{n}{10}$  and case 2 refers to the else statement.

First we get the probability of each case happening,  $\begin{cases} P(\text{Case 1}) = \frac{n-10}{n} \\ P(\text{Case 2}) = \frac{10}{n} \end{cases}$  Now we need to

get the time complexity of each case,  $\begin{cases} T(\text{Case 1}) = \Theta(n) \\ T(\text{Case 2}) = \Theta(1) \end{cases}$  Using all that we can solve for the expected time complexity of the function,

$$\sum P(i) \cdot T(i) = \frac{n-10}{n} \Theta(n) + \frac{10}{n} \Theta(1) = \Theta(n)$$

b) `def bogomax(numbers):`

```
    """Find the largest number by random guess and check."""
    while True:
        # randomly choose an element with uniform probability in constant time
        guess = random.choice(numbers)

        # check whether it is the largest
```

```

for number in numbers:
    guess_is_a_maximum = True
    if number > guess:
        guess_is_a_maximum = False

if guess_is_a_maximum:
    return guess

```

In this part, you may assume that the numbers are distinct, and that  $n$  is the size of `numbers`.

Hint: if  $0 < b < 1$ , then  $\sum_{p=1}^{\infty} p \cdot b^{p-1} = \frac{1}{(1-b)^2}$ .

**Solution:** First we have to notice that for this particular function, it can go on infinitely theoretically. Therefore we have to find the case in which the function ends to find the average time complexity. We can get the probability of that to happen to be,

$$P(i) = \left(1 - \frac{1}{n}\right)^{k-1} \cdot \left(\frac{1}{n}\right)$$

Where  $k$  is the iteration in which the guess is found. ANd we also need to get the time complexity for one iteration which we can get as,

$$T(i) = \Theta(n)$$

Then using that the the formula we get,

$$\sum_{i=1}^{\infty} \left(1 - \frac{1}{n}\right)^{k-1} \cdot \left(\frac{1}{n}\right) \cdot cni = c \sum_{i=1}^{\infty} i \left(1 - \frac{1}{n}\right)^{k-1} \quad (8)$$

$$= c \times \frac{1}{\left(1 - 1 + \frac{1}{n}\right)^2} \quad (9)$$

$$= cn^2 \quad (10)$$

Therefore we get that the average time complexity of the function is  $\Theta(n^2)$

### c) Extra Credit (3 points)

**Note:** this part is extra credit, and is totally optional! We suggest skipping it for now and coming back to it if you have time.

The above version of `bogomax` can be improved by immediately returning when a maximum is found, as shown in the following code:

```

import random

def bogomax(numbers):
    """Find the largest number by random guess and check."""
    while True:
        # randomly choose an element with uniform probability in constant time
        guess = random.choice(numbers)

        # check whether it is the largest
        for number in numbers:
            if number > guess:
                break
        else:
            # yes, python has for-else blocks.

```

```
# if we get here, the for-loop completed without breaking
return guess
```

This makes the code more performant, but harder to analyze.

Analyze the average case time complexity of this code rigorously. You may assume that the elements of **numbers** are unique and that the maximum is equally-likely to be anywhere in the list.

**Solution:** With this version of bogomax, we can again have the same assumption that the function can theoretically go on forever, and the probability of getting the max value on any guess is still,

$$P(i) = \left(1 - \frac{1}{n}\right)^{k-1} \cdot \left(\frac{1}{n}\right)$$

The difference however, is that the for loop breaks if and when the max is found. We can deduce that the new for loop is essentially like a linear search. Therefore we get,

$$T(i) = \Theta(n)$$

Therefore we get the average case to still be,

$$\sum_{i=1}^{\infty} \left(1 - \frac{1}{n}\right)^{k-1} \cdot \left(\frac{1}{n}\right) \cdot cni = c \sum_{i=1}^{\infty} i \left(1 - \frac{1}{n}\right)^{k-1} \quad (11)$$

$$= c \times \frac{1}{\left(1 - 1 + \frac{1}{n}\right)^2} \quad (12)$$

$$= cn^2 \quad (13)$$

So the average case for the function is still  $\Theta(n^2)$  although it is easy to see that this change does make the code faster, but in the consideration of big  $\Theta$  it is not any faster and still  $\Theta(n^2)$

#### Problem 4.

Provide a tight theoretical lower bound for the problems given below. Provide justification for your answer.

- a) Given a list of size  $n$  containing **Trues** and **Falses**, determine whether **True** or **False** is more common (or if there is a tie).

**Solution:** The tight theoretical lower bound will have to be  $\Omega(n)$  this is because in order to be sure the exact count of each element we have to iterate through every single element in the list at least once to ensure we have the right most common value.

- b) Given a list of  $n$  numbers, all assumed to be integers between 1 and 100, sort them.

**Solution:** If we assume the best case scenario where the list is already sorted, we can say that the tight lower bound is  $\Omega(n)$  as the algorithm would still have to iterate through every single element in order to verify that the list is indeed already sorted. However, if we assume that the list is not sorted, then the tight lower bound would be  $\Omega(n \log n)$  this is because the algorithm would have to iterate through every value at least once, and if there is an element in the wrong spot, would have to move it to the right spot which should be  $\log n$  time. Therefore giving us  $\Omega(n \log n)$ .

- c) Given an  $\sqrt{n} \times n$  array whose rows are sorted (but whose columns may not be), find the largest overall entry in the array.

For example, the array could look like:

$$\begin{pmatrix} -2 & 4 & 7 & 8 & 10 & 12 & 20 & 21 & 50 \\ -30 & -20 & -10 & 0 & 1 & 2 & 3 & 21 & 23 \\ -10 & -2 & 0 & 2 & 4 & 6 & 30 & 31 & 35 \end{pmatrix}$$

This is an  $\sqrt{n} \times n$  array, with  $n = 9$  (there are 3 rows and 9 columns). Each row is sorted, but the columns aren't.

**Solution:** The tight lower bound is  $\Omega(n)$ , this is due to the special case where the rows are already sorted so one must only check the last column of the matrix to find the largest value in the array. As no matter what the largest value of each row is at the last column making the only possible place for the largest value to be to be in the last column only.