DSC 40B - Homework 02

Due: Wednesday, April 16

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.

For each of the following functions, express its rate of growth using Θ-notation, and prove your answer by finding constants which satisfy the definition of Θ-notation. Make sure to show your work by writing out the chain of inequalities to prove each bound, like we did in lecture. Note: please do *not* use limits to prove your answer (though you can use them to check your work).

a) $f(n) = 3n^2 - 8n + 4$

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

We start with an upper bound:

$$\begin{aligned} f(n) &= 3n^2 - 8n + 4 \\ &\leq 3n^2 + 4 \\ &\leq 3n^2 + 4n^2 \qquad (n \geq 1) \\ &= 7n^2 \end{aligned}$$

Then for a lower bound:

$$\begin{aligned} f(n) &= 3n^2 - 8n + 4 \\ &\geq 3n^2 - 8n \end{aligned}$$

Our problem now is the $-8n$. Remember that we can do anything that makes the quantity *smaller* when trying to find a lower bound, so we can't simply drop the $-8n$ since that would actually make the quantity bigger. Instead, we recognize intuitively that n^2 will eventually be larger than $-8n$, so we can "break off" a piece of the $3n^2$ and use it to "balance" the $-8n$.

$$\begin{aligned} &= 2n^2 + n^2 - 8n \\ &= 2n^2 + (n^2 - 8n) \end{aligned}$$

Again, our intuition is that $n^2 - 8n$ will eventually be positive when n is large enough, and so we'll be able to drop it to get a lower bound. When is this? We solve $n^2 - 8n \geq 0$ for n, finding that $n^2 \geq 8n \implies n \geq 8$. So when $n \geq 8$, the stuff in parens is positive and can be dropped:

$$\geq 2n^2 \qquad (n \geq 8)$$

Therefore, $2n^2 \leq f(n) \leq 7n^2$ provided that $n \geq 8$.

b) $f(n) = \dfrac{n^2 + 2n - 5}{n - 10}$

Solution: $f(n) = \Theta(n)$.

We begin with an upper bound.

$$\begin{aligned} f(n) &= \frac{n^2 + 2n - 5}{n - 10} \\ &\leq \frac{n^2 + 2n}{n - 10} \\ &= \frac{n(n+2)}{n - 10} \end{aligned}$$

Now, $(n+2)/(n-10) \leq 2$ for all $n \geq 22$, hence:

$$\leq 2n, \qquad (n \geq 22)$$

Now for a lower bound:

$$\begin{aligned} f(n) &= \frac{n^2 + 2n - 5}{n - 10} \\ &\geq \frac{n^2 - 5}{n - 10} \end{aligned}$$

To get a lower bound, we make this quantity smaller. We can do so by making the denominator bigger by dropping the -10:

$$\begin{aligned} &\geq \frac{n^2 - 5}{n} \\ &\geq \frac{\frac{1}{2}n^2 + \frac{1}{2}n^2 - 5}{n} \end{aligned}$$

Now, $\frac{1}{2}n^2 - 5 \geq 0$ for all $n \geq 4$. Hence:

$$\begin{aligned} &\geq \frac{\frac{1}{2}n^2 + 0}{n}, \qquad (n \geq 4) \\ &= n/2 \end{aligned}$$

So in total, $\frac{n}{2} \leq f(n) \leq 2n$ for all $n \geq 22$.

c) $f(n) = \begin{cases} n^2, & n \text{ is odd,} \\ 4n^2, & n \text{ is even} \end{cases}$

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

Our goal is to find a function that upper bounds f both when n is even and when it is odd, and a separate function that lower bounds f both when n is even and when it is odd.

In this case, $f(n) \leq 4n^2$ for all $n \geq 1$, whether n is even or odd. Likewise, $f(n) \geq n^2$ for all $n \geq 1$, even or odd. Hence: $n^2 \leq f(n) \leq 4n^2$ for all $n \geq 1$.

You might be tempted to say that the first part, n^2, is $\Theta(n^2)$, and the second part, $4n^2$, is also $\Theta(n^2)$, so the whole function is $\Theta(n^2)$. This *is* a valid argument, but we haven't proven that it's true – though you can prove it is valid using the techniques that we learned in lecture, but it's a

little too "high level" for this problem. Here, we want to go all the way back to the definition.

d) State the growth of the function below using Θ notation in as simplest of terms possible, and prove your answer by finding constants which satisfy the definition of Θ notation.

E.g., if $f(n)$ were $3n^2 + 5$, we would write $f(n) = \Theta(n^2)$ and not $\Theta(3n^2)$.

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

Solution: $\Theta(n)$.

Recall that we write $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2, N such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

for all $n \geq N$.

In this case, we will prove the inequality for $g(n) = n$. There are infinitely-many choices of c_1, c_2, and N that will satisfy the inequality; we will prove such one:

Upper Bound : $f(n) \leq c_2 \cdot g(n)$

$$\begin{aligned}
\frac{n^3 - n^2 + n + 1000}{(n-1)(n+2)} &\leq \frac{1002n^3}{n^2 + n - 2} \\
&\leq \frac{1002n^3}{n^2 + (n-2)} (n \geq ?) \\
&\leq \frac{1002n^3}{n^2} (n \geq 2) \\
&\leq 1002n
\end{aligned}$$

Lower Bound : $c_1 \cdot g(n) \leq f(n)$

$$\begin{aligned}
\frac{n^3 - n^2 + n + 1000}{(n-1)(n+2)} &\geq \frac{n^3 - n^2}{n^2 + n} \\
&\geq \frac{0.5n^3 + (0.5n^3 - n^2)}{n^2 + n} (n \geq ?) \\
&\geq \frac{0.5n^3}{n^2 + n} (n \geq 2) \\
&\geq \frac{0.5n^3}{n^2 + n^2} \\
&\geq \frac{1}{4} n
\end{aligned}$$

We have $n \geq 2$ for both the upper and lower bound. Hence, $N = 2$.
Therefore $f(n) = \theta(n)$ with constants $N = 2, c_1 = 0.25, c_2 = 1002$

Problem 2.

Suppose $T_1(n), \ldots, T_6(n)$ are functions describing the runtime of six algorithms. Furthermore, suppose we have the following bounds on each function:

$$T_1(n) = \Theta(n^{2.5})$$
$$T_2(n) = O(n \log n)$$
$$T_3(n) = \Omega(\log n)$$
$$T_4(n) = O(n^4) \text{ and } T_4 = \Omega(n^2)$$
$$T_5(n) = \Theta(n)$$
$$T_6(n) = \Theta(n \log n)$$
$$T_7(n) = O(n^{1.5} \log n) \text{ and } T_7 = \Omega(n \log n)$$

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 1: $T_1(n) + T_2(n)$.

Solution: $T_1(n) + T_2(n)$ is $\Theta(n^{2.5})$.

Example 2: $f(n) = 2 \cdot T_4(n)$.

Solution: $f(n) = 2 \cdot T_4(n)$ is $O(n^4)$ and $\Omega(n^2)$.

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

Solution: $\Theta(n^{2.5})$

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

Solution: $\Theta(n \log n)$

c) $T_2(n) + T_4(n)$

Solution: $O(n^4)$ and $\Omega(n^2)$

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

Solution: $O(n^4)$ and $\Omega(n^2)$

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

Solution: $\Omega(n^{2.5})$

Note that we can't give an upper bound here, because we don't have an upper bound on $T_3(n)$. For example, it could be that $T_3(n) = n^{10}$ or n^{100} or n^{1000}; we just do not have enough information

to know.

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

Solution: $O(n^{6.5})$ and $\Omega(n^{4.5})$

g) $T_6(n) + T_4(n)/T_5(n)$

Solution: $O(n^3)$ and $\Omega(n \log n)$

Problem 3.

In each of the problems below state the best case and worst case time complexities of the given piece of code using asymptotic notation. Note that some algorithms may have the same best case and worst case time complexities. If the best and worst case complexities *are* different, identify which inputs result in the best case and worst case. You do not otherwise need to show your work for this problem.

Example Algorithm: `linear_search` as given in lecture.

Example Solution: Best case: $\Theta(1)$, when the target is the first element of the array. Worst case: $\Theta(n)$, when the target is not in the array.

a)
```
def kth_largest(numbers, k):
    """Finds the k-th largest element in the array.
    `numbers` is an array of n numbers."""
    n = len(numbers)
    for i in range(n):
        count = 0 # Count how many numbers are larger than numbers[i]
        for j in range(n):
            if numbers[j] > numbers[i]:
                count += 1
        if count == k - 1:
            return numbers[i]
```

Solution: Best case: $\Theta(n)$, when the first element of the array is the kth largest.

Worst case: $\Theta(n^2)$, we need to count the number of elements larger than each element. If the last element is kth largest, then we need to run entire nested loops.

b)
```
def index_of_kth_largest(numbers, k):
    """`numbers` is an array of size n"""
    # the kth_largest() from above
    element = kth_largest(numbers, k)
    # the linear_search() from lecture 3 slide 22
    return linear_search(numbers, element)
```

Solution: Best case: $\Theta(n)$. This occurs when the kth largest element is the first element of the array, as then `kth_largest` takes $\Theta(n)$ and `linear_search` takes $\Theta(1)$, for a total of $\Theta(n)$.

Worst case: $\Theta(n^2)$. This occurs when the kth largest element is the last element of the array. In this situation, `kth_largest` takes $\Theta(n^2)$ time and `linear_search` takes $\Theta(n)$ time, for a total of $\Theta(n^2)$ time.

Problem 4.

In each of the problems below compute the 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 double_coin_toss(n):
    coin_1 = np.random.rand() > 0.5
    coin_2 = np.random.rand() > 0.5
    if coin_1 and coin_2:
        i = n
        while i > 0:
            print("We got two heads!")
            i -= 1
    else:
        for i in range(n ** 2):
            print("We didn't get two heads.")
```

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

We can think of the two cases here as being 1) We got 2 heads, and 2) We did not get 2 heads. The probability of the first case is $1/4$ because the probability of getting each individual head is $1/2$ and we multiply $1/2 * 1/2$ for 2 coins, giving $1/4$. The probability of the second case is $3/4$ which comes from subtracting the probability of the first case from 1 giving $1 - 1/4 = 3/4$.

In the first case, linear time is taken since we must loop over `n`. In the second case, quadratic time is taken since we must loop over `range(n**2)`.

Therefore, the average time taken is:

$$\begin{aligned} &\frac{1}{4}\Theta(n) + \frac{3}{4}\Theta(n^2) \\ &\qquad = \Theta(n) + \Theta(n^2) \\ &\qquad = \Theta(n^2) \end{aligned}$$

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

    if k < np.sqrt(n):
        for i in range(n**2):
            print(i)
    else:
        print('Never mind...')
```

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

We can think of the two cases here as being 1) k is randomly chosen to be less than $\sqrt{n}$, and 2) k is randomly chosen to be greater. The probability of the first case is $\sqrt{n}/n = 1/\sqrt{n}$, and the probability of the second case is $(n - \sqrt{n})/n = 1 - 1/\sqrt{n}$.

In the first case, quadratic time is taken since we must loop over `range(n**2)`. In the second

case, constant time is taken since we print `"Never mind..."` and exit.

Therefore, the average time taken is:

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

c)
```
def baz(n):
    # randomly choose a number between 0 and n-1 in constant time
    x = np.random.randint(n)

    for i in range(x):
        for j in range(i):
            print("Hello!")
```

Hint: In class, we saw that the sum of the first n integers is $\frac{n(n+1)}{2}$. It turns out that the sum of the first n integers squared (so, $1^2 + 2^2 + 3^2 + \ldots + n^2$) is $\frac{n(n+1)(2n+1)}{6}$. You can (and should) use this fact, but make sure to point it out when you do.

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

Here, `x` is a random integer from 0 to n-1. We can think of each possible value of `x` as a different case. That is, in Case 0, we get 0 for `x`, in Case 1, we get 1 for `x`, and so on up to Case $n-1$, where we get $n-1$ for `x`.

Each case is equally likely, and there are n of them, so the probability of each case is $1/n$.

Next, we calculate the time taken in each case by counting the number of executions of the inner loop body. Consider Case k (in which `x` turns out to be k). In this case, the nested loop becomes:

```
for i in range(k):
    for j in range(i):
        print("Hello!")
```

We have analyzed these sorts of dependent nested loops before in lecture, and we know that the number of executions of the inner loop body is equal to the sum $1 + 2 + 3 + \ldots + (k-1)$. This is an arithmetic sum, and it has a formula: $\frac{k(k-1)}{2}$. So, the number of executions in Case k is $k(k-1)/2$, and therefore the time is proportional to this.

Recalling that the expected time complexity has the formula:

$$T_{\text{expected}}(n) = \sum_{cases} P(\text{case } k) \cdot T(\text{case } k)$$

We fill in the probabilities and times:

$$
\begin{aligned}
&= \sum_{k=0}^{n-1} \frac{1}{n} \cdot \frac{k(k-1)}{2} \\
&= \frac{1}{2n} \sum_{k=0}^{n-1} k(k-1) \\
&= \frac{1}{2n} \sum_{k=0}^{n-1} k^2 - k \\
&= \frac{1}{2n} \left(\sum_{k=0}^{n-1} k^2 - \sum_{k=0}^{n-1} k \right)
\end{aligned}
$$

We recognize the second sum as an arithmetic sum, and we recognize the first sum from the hint. Therefore:

$$
\begin{aligned}
&= \frac{1}{2n} \left(\frac{n(n-1)(2n-1)}{6} - \frac{n(n-1)}{2} \right) \\
&= \frac{1}{2n} \left(\Theta(n^3) - \Theta(n^2) \right) \\
&= \Theta(n^2)
\end{aligned}
$$

The hardest part of this problem might have been figuring out $T(\text{case } k)$; that is, in counting the number of exeuptions of the inner loop body in Case k. If you got stuck here, or wanted to check your guess, remember that you can try it out in code:

```
def count_execs_for_case(k):
    number_of_execs = 0

    for i in range(k):
        for j in range(i):
            print("Hello!")
            number_of_execs += 1

    return number_of_execs

print(count_execs_for_case(9))
```

This would print `36`, which matches the prediction of our formula when $k = 9$, since $9(9-1)/2 = 36$.

Problem 5.

For each problem below, state the largest theoretical lower bound that you can think of and justify (that is, your bound should be "tight"). Provide justification for this lower bound. You do not need to find an algorithm that satisfies this lower bound.

Example: Given an array of size n and a target t, determine the index of t in the array.

Example Solution: $\Omega(n)$, because in the worst case any algorithm must look through all n numbers to verify that the target is not one of them, taking $\Omega(n)$ time.

a) Given a list of size n containing `True`s and `False`s, determine whether `True` or `False` is more common (or if there is a tie).

Solution: $\Omega(n)$.

In the worst case, we need to read the whole list, as the "winner" will be determined by the very last entry.

Theoretical lower bound concerns the worst case, but note that even the best case will require us to read at least half of the entries here. For example, if $n = 100$ and the first 51 entries are `True`, we can conclude that `True` is the winner after reading those 51. But if even one of them is `False`, it may be that `False` is the winner if it makes up all of the remaining entries.

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

Solution: $\Omega(n)$.

At the minimum, we have to read all of the numbers, which takes $\Theta(n)$ time.

You may have heard that "sorting takes $\Theta(n \log n)$ time", but we have to be careful. First, this applies only to *comparison sorts*, where we sort by comparing elements of the input to one another. If we don't have any assumptions on the numbers, we have to do a comparison sort. But since we've assumed that they are between 0 and 100, we can actually sort them without ever comparing any two numbers. Here's the algorithm:

1. Initialize a list, `counts`, with 100 entries, all zero to start.
2. Loop through the input list. For each number, x, increment `counts[x]` by one. For example, if we see 7, increment `counts[7]` by one. In short, this loop counts how many times we see each number.
3. Finally, loop through the range $0, 1, \ldots, 100$, and print the number x a number of times equal to `counts[x]`. That is, if `counts[7]` is 3, print 3 sevens.

Because we're looping through the input list once, and it has n elements, we've sorted the list in $\Theta(n)$ time.

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: $\Omega(\sqrt{n})$

The largest element of the array has to be in the last column, so we simply look through all of the entries in that column. There are $\sqrt{n}$ such entries, so we take $\Theta(\sqrt{n})$ time.

Note that we don't have to read all of the entries of the array – we can actually ignore almost all of them.