DSC 40B - Homework 02

Due: Tuesday, October 15

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. Note: please do *not* use limits to prove your answer (though you can use them to check your work).

a) $f(n) = 5n^2 - 10n + 4$

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

We start with an upper bound:

$$\begin{aligned} f(n) &= 5n^2 - 10n + 4 \\ &\leq 5n^2 + 4 \\ &\leq 5n^2 + 4n^2 \qquad (n \geq 1) \\ &= 9n^2 \end{aligned}$$

Then for a lower bound:

$$\begin{aligned} f(n) &= 5n^2 - 10n + 4 \\ &\geq 5n^2 - 10n \end{aligned}$$

Our problem now is the $-10n$. 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 $-10n$ since that would actually make the quantity bigger. Instead, we recognize intuitively that n^2 will eventually be larger than $-10n$, so we can "break off" a piece of the $5n^2$ and use it to "balance" the $-10n$.

$$\begin{aligned} &= 4n^2 + n^2 - 10n \\ &= 4n^2 + (n^2 - 10n) \end{aligned}$$

Again, our intuition is that $n^2 - 10n$ 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 $n^2 - 10n \geq 0$ for n, finding that $n^2 \geq 10n \implies n \geq 10$. So when $n \geq 10$, the stuff in parens is positive and can be dropped:

$$\geq 4n^2 \qquad (n \geq 10)$$

Therefore, $4n^2 \leq f(n) \leq 9n^2$ provided that $n \geq 10$.

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

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, $f(n) \leq 3n^2$ 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: $n^2 \leq f(n) \leq 3n^2$ 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, $3n^2$, 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.

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:

$$\begin{aligned}
&T_1(n) = \Theta(n^4)\\
&T_2(n) = O(n^2 \log(n)) \text{ and } T_2(n) = \Omega(n)\\
&T_3(n) = \Omega(n^3)\\
&T_4(n) = O(n^6) \text{ and } T_4 = \Omega(n^2)\\
&T_5(n) = \Theta(n)\\
&T_6(n) = O(n^2) \text{ and } T_6(n) = \Omega(\log(n))\\
&T_7(n) = \Theta(\log(n))
\end{aligned}$$

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 $\Theta(n^4)$.

a) $T_1(n) + T_2(n) + T_5(n)$

Solution: $\Theta(n^4)$

b) $T_4(n) + T_6(n)$

Solution: $O(n^6)$ and $\Omega(n^2)$

Because $T_4(n) = \Omega(n^2)$, the sum has to take at least n^2 time.

c) $T_7(n) + T_3(n)$

Solution: $\Omega(n^3)$

d) $T_1(n) \cdot T_7(n) \ / \ T_5(n)$

Solution: $\Theta(n^3 \log(n))$

e) $T_1(n) + T_4(n)$

Solution: $O(n^6)$ and $\Omega(n^4)$.

Think of $T_1(n)$ as $O(n^4)$ and $\Omega(n^4)$. In other words, there’s a lower-bound of n^4. So the sum of the two functions can’t take less than n^4 time, and n^4 is a lower bound.

f) $T_2(n) + T_6(n)$

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

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)
```python
def first_repeated(data):
    """Finds the first repeated element in `data`.
    Returns `None` if no elements are repeated.
    `data` is an array of size n."""
    n = len(data)
    for i in range(n):
        for j in range(i + 1, n):
            if data[i] == data[j]:
                return data[i]
    return None
```

Solution: Best case: $\Theta(1)$, when the first and second elements of the array are the same.

Worst case: $\Theta(n^2)$, when there are no repeated elements.

You have to wonder... isn't there a faster approach?

b)
```python
def f_2(data):
    """`data` is an array of n numbers"""
    m = data[0]
    c = 1
    for ix in range(1, len(data)):
        if c == 0:
            m = data[ix]
            c = 1
        elif m == data[ix]:
            c += 1
            if c > len(data) // 2:
                return m
        else:
            c -= 1

    return m
```

Solution: Best case and worst case are both $\Theta(n)$. In the "best case", the loop exits early when `c > len(data) // 2`. But c grows by at most 1 on each iteration, so we'd need to go through at least $n/2$ iterations to exit, which is still $\Theta(n)$.

By the way, this algorithm is called the Boyer-Moore majority algorithm, and can be used to efficiently find a majority element of a list (as long as there is one). Though note that you can reason about the algorithm's time complexity without actually knowing what it computes.

c)
```
def kth_smallest(numbers, k):
    """Finds the k-th smallest 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 less 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 smallest.

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

d)
```
def index_of_kth_smallest(numbers, k):
    """`numbers` is an array of size n"""
    # the kth_smallest() from above
    element = kth_smallest(numbers)
    # the linear_search() from lecture
    return linear_search(numbers, element)
```

Solution: Best case: $\Theta(n)$. This occurs when the kth smallest element is the first element of the array, as then `kth_smallest` 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 smallest element is the last element of the array. In this situation, `kth_smallest` 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 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**2):
            print(i)
    else:
        for i in range(n):
            print(i)
```

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

We can think of the two cases here as being 1) k is randomly chosen to be greater than $n/10$, and 2) k is randomly chosen to be less than $n/10$. The probability of the first case is $9/10$, and the probability of the second case is $1/10$.

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

Therefore, the average time taken is:

$$\begin{aligned}\frac{9}{10}\Theta(n^2) + \frac{1}{10}\Theta(n)& \\ &= \Theta(n^2) + \Theta(n)\\ &= \Theta(n^2)\end{aligned}$$

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

    if k > 10: # <--- here is the difference!
        for i in range(n**2):
            print(i)
    else:
        for i in range(n):
            print(i)
```

Note: this code is *almost* the same as the code in the previous part, with one small difference that has been highlighted with a comment.

Solution: $\Theta(n)$.

Here we again randomly choose an integer k from 0 to n-1, but instead of checking if the integer is less than $n/10$, we check if k is greater than 10. If it is, we loop over `range(n**2)`, and if not we loop over loop over `range(n)`. The probability of the first case (k being greater than or equal to 10) is $(n - 10)/n$, and the probability of the second case (k being less than 10) is $10/n$. So, the average time taken is:

$$\begin{aligned}\frac{n-10}{n}\Theta(n^2) + \frac{10}{n}\Theta(n)& \\ &= \frac{n^3 - 10n^2}{n} + \frac{10n}{n}\\ &= n^2 - 10n + 10\\ &= \Theta(n^2 + 1)\\ &= \Theta(n^2)\end{aligned}$$

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 an array of n numbers, find the second largest number.

Solution: $\Omega(n)$.

Any algorithm must look through all n numbers to be sure it didn't miss the second largest number, and so must take $\Omega(n)$ time. Algorithm: loop through the numbers, keeping track of the maximum and the next largest number seen.

b) Given an array of n numbers, check to see if there are any duplicates.

Solution: $\Omega(n)$

Again, we have to look through all of the numbers in the worst case; if we don't, we can't be sure that it isn't a duplicate of a number we have looked at.

It turns out that the actual, *tight* theoretical lower bound for this problem is $\Omega(n \log n)$, under some assumptions.[a] But justifying that lower bound is not trivial, and this question asked you to state the best TLB that you could justify.

If you said that the TLB is $\Omega(n \log n)$ because you though of sorting the list and checking for consecutive equal elements, this is good thinking – but in order to claim that the TLB is therefore $\Omega(n \log n)$, you'd also need to justify the assumption that sorting (or something like it) is *necessary* of any optimal algorithm. That's not an easy claim to justify!

[a]See: `https://en.wikipedia.org/wiki/Element_distinctness_problem`

c) Given an array of n integers (with $n \geq 2$), check to see if there is a pair of elements that add to be an even number.

Solution: $\Omega(1)$

This can be done in constant time by checking only the first 2 numbers. If the first two are both even, or both odd, then the answer is “yes”, since the sum will then be even. If the first two numbers are of different polarity – one even and the other odd – then the answer is “no” if the size of the list is 2, otherwise it is “yes” since the third number in the array must be either even or odd; pairing it with one of the first two numbers with the same polarity gives an even sum.