DSC 40B - Homework 02

Due: Monday, October 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.

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

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

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:

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

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 $\Theta(n^2)$.

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

Solution: $\Theta(n^3)$

b) $T_4(n) + T_5(n)$

Solution: $\Theta(n^3)$

c) $T_7(n) + T_4(n)$

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

d) $T_3(n) + T_1(n)$

Solution: $\Omega(n^2)$

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.

e) $T_2(n) + T_6(n)$

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

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

Solution: $O(n^4)$ and $\Omega(n^3)$

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 < 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 (that is, $\Theta(n^2)$ 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:

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

b)
```
def bogosearch(numbers, target):
    """search by randomly guessing. `numbers` is an array of n numbers"""
```

```
n = len(numbers)

while True:
    # randomly choose a number between 0 and n-1 in constant time
    guess = np.random.randint(n)
    if numbers[guess] == target:
        return guess
```

In this part, you may assume that the numbers are distinct and that the target is in the array.

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

Solution: $\Theta(n)$.

Maybe the best way to divide this into cases is to consider Case 1 to be when we guess the location of the target on the very first try; Case 2 is when we guess the location of the target incorrectly on the first try, but get it right on the second; Case 3 is when we get the first two guesses wrong but get the third guess correct, and so on. In general, Case α is when we made incorrect guesses on the first $\alpha - 1$ iterations, but a correct guess on the αth iteration, therefore returning.

The probability of Case α is the probability of guessing wrong $\alpha - 1$ times followed by guessing correctly. Since these guesses are all statistically independent, and since the probability of any one guess being wrong is $(n-1)/n$ and the probability of it being right is $1/n$, we find that the probability of Case α is:

$$\underbrace{\left(\frac{n-1}{n}\right)\left(\frac{n-1}{n}\right)\cdots\left(\frac{n-1}{n}\right)}_{\alpha-1 \text{ times}} \cdot \frac{1}{n} = \left(\frac{n-1}{n}\right)^{\alpha-1} \frac{1}{n} = \left(1 - \frac{1}{n}\right)^{\alpha-1} \frac{1}{n}$$

The time taken on any iteration is constant; let's call it c. In case α we make α iterations, and so we take $c\alpha$ time in total.

There are actually infinitely many cases here, since it is possible that we are consistently unlucky and never guess the right entry. Therefore, our summation will actually be an infinite series.

Therefore the expected time over all possible cases is:

$$\sum_{\alpha=1}^{\infty} P(\text{case } \alpha) \cdot T(\text{case } \alpha) = \sum_{\alpha=1}^{\infty} \left(1 - \frac{1}{n}\right)^{\alpha-1} \frac{1}{n} \cdot c\alpha = \frac{c}{n} \sum_{\alpha=1}^{\infty} \alpha \left(1 - \frac{1}{n}\right)^{\alpha-1}$$

We can now use the hint. Letting $b = 1 - 1/n$, we have:

$$= \frac{c}{n} \sum_{\alpha=1}^{\infty} \alpha b^{\alpha-1}$$
$$= \frac{c}{n} \cdot \frac{1}{(1-b)^2}$$

Now substituting back in for $b = 1 - 1/n$:

$$\begin{aligned} &= \frac{c}{n} \cdot \frac{1}{[1-(1-1/n)]^2} \\ &= \frac{c}{n} \cdot \frac{1}{1/n^2} \\ &= \frac{c}{n} \cdot n^2 \\ &= cn \end{aligned}$$

So the average case time complexity is $\Theta(n)$.

Problem 4.

For each problem below, state the largest theoretical lower bound that you can think of and justify. 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.