DSC 40B - Homework 02

Due: Wednesday, January 25

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:

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

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: $\Theta(n^3)$

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

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

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

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: $\Omega(n^3)$

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: $O(n^7)$ and $\Omega(n^5)$

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

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

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, linear time is taken since we must loop over `range(n)`. In the second case, constant time is taken since we print `"Never mind..."` and exit.

Therefore, the average time taken is:

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

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.

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

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.