DSC 40B - Homework 02

Due: Tuesday, January 19

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 on Tuesday at 11:59 p.m.

Problem 1. *(Eligible for Redemption)*

Write each of the following functions in Θ-notation in simplest terms. You do not need to show your work or provide justification.

Hint: your answer should have the form $\Theta(n^\alpha)$, $\Theta((\log_\mu(n))^\beta)$, $\Theta(\gamma^{\delta n})$, or $\Theta(1)$, where $\alpha, \beta, \gamma, \delta, \mu$ are constants that depend on the problem. While you *can* find the answer using more formal means (such as limits), you should be able to find the answers rather quickly using informal reasoning. You can always use the formal approach to check your answer, if you wish.

a) $f(n) = \sqrt{n^3 + 2n^2 + 500}$

Solution: $f(n) = \Theta(n^{3/2})$

b) $f(n) = 20 \times 2^{\log_2(n^3+n)}$

Solution: $f(n) = \Theta(n^3)$

c) $f(n) = 2^{10} + 10^2 + \log_{10} 1000$

Solution: $f(n) = \Theta(1)$.

d) $f(n) = \log_2(3^{n+2} + 5n^3 + 1)$

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

3^{n+2} grows a lot faster than the other terms inside of the logarithm, so we can ignore them. Therefore we ask: what is $\log_2(3^{n+2})$ simplified in Θ-notation?

If you look up a table of logarithm properties, you'll find the identity $\log_a b^n = n \log_a b$. Therefore $\log_2(3^{n+2}) = (n+2)\log_2 3 = \Theta(n)$.

In general, $\log_a b^{g(n)} = \Theta(g(n))$, as long as a and b are constants greater than one independent of n.

e) $f(n) = 2^{10n} + 10^{2n}$

Solution: $f(n) = \Theta(2^{10n})$

Informally, 2^{10n} grows a lot faster than 10^{2n}, so this is dominated by 2^{10n}. Note that $\Theta(2^{10n})$ is *not* the same as $\Theta(2^n)$; the former grows a lot faster. This is a case where constants *do* matter.

If you want to see this more formally, the easiest way might be to use limits. We'll show that

$$\lim_{n\to\infty} \frac{2^{10n} + 10^{2n}}{2^{10n}} = 1$$

which proves that $f(n) = \theta(2^{10n})$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{2^{10n} + 10^{2n}}{2^{10n}} &= \lim_{n\to\infty} \left[\frac{2^{10n}}{2^{10n}} + \frac{10^{2n}}{2^{10n}}\right] \\
&= \lim_{n\to\infty} \frac{2^{10n}}{2^{10n}} + \lim_{n\to\infty} \frac{10^{2n}}{2^{10n}} \\
&= 1 + \lim_{n\to\infty} \frac{10^{2n}}{2^{10n}}
\end{aligned}$$

This limit is easier to find if we work with the same base for each exponential. We can write $10 = 2^{\log_2 10} \approx 2^{3.3}$.

$$\begin{aligned}
&= 1 + \lim_{n\to\infty} \frac{(2^{\log_2 10})^n}{2^{10n}} \\
&= 1 + \lim_{n\to\infty} \frac{2^{(\log_2 10)n}}{2^{10n}}
\end{aligned}$$

Recall that $2^a/2^b = 2^{a-b}$, so:

$$= 1 + \lim_{n\to\infty} 2^{n(\log_2 10 - 10)}$$

$\log_2 10 < 10$, so the exponent is negative and the limit goes to zero:

$$= 1 + 0$$

f) $f(n) = \log_2(n^2 + 3n + 5) \times \log_2(3n^3)$

Solution: $\Theta((\log_2 n)^2)$. The base doesn't actually matter, though, so we typically write $\Theta((\log n)^2)$.

Problem 2. *(Eligible for Redemption)*

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 f_1(data):
    """`data` is a two-dimensional array of size n*n"""
    n = len(data)
    for i in range(n):
        for j in range(n):
            if data[i, j] == 42:
                return (i,j)
```

Solution: Best case: $\Theta(1)$, when `data[0,0] == 42`.

Worst case: $\Theta(n^2)$, when `42` is not in the array.

b)
```
def f_2(data):
    """`data` is an array of n numbers"""
    n = len(numbers)
    swapped = True
    while swapped:
        swapped = False
        for i in range(1,n):
            if numbers[i-1] < numbers[i]:
                # next line swaps elements in Theta(1) time
                numbers[i], numbers[i-1] = numbers[i-1], numbers[i]
                swapped = True
```

Solution: Best case: $\Theta(n)$. This occurs when the input list is sorted in descending order.

Worst case: $\Theta(n^2)$. This occurs when the input list is sorted in ascending order.

c)
```
def median(numbers):
    """computes the median. `numbers` is an array of n numbers"""
    n = len(numbers)
    for x in numbers:
        less = 0
        more = 0
        for y in numbers:
            if y <= x:
                less += 1
            if y >= x:
                more += 1
        if less >= n/2 and more >= n/2:
            return x
```

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

Worst case: $\Theta(n^2)$, when the first $n-1$ elements are not medians, and the last element of the array is the median.

d)
```
def mode(data):
    """computes the mode. `data` is an array of n numbers."""
    mode = None
    largest_frequency = 0
    for x in data:
        count = 0
        for y in data:
            if x == y:
                count += 1
        if count > largest_frequency:
            largest_frequency = count
            mode = x
        if count > n/2:
            return mode
    return mode
```

Solution: The best case is $\Theta(n)$, while the worst case is $\Theta(n^2)$.

In the worst case, the mode is at the very end of the list, as in `[5, 1, 2, 7, 9, 9]`. The outer loop iterates n times, and the inner loop does $\Theta(n)$ work per iteration for a total of $\Theta(n^2)$.

In the best case, the mode is the first element of the list, as in, for instance `[1, 5, 8, 2, 1, 1, 1]`. The mode will be discovered on the very first iteration of the outer loop, but the inner loop will still take $\Theta(n)$ time, for a total of $\Theta(n)$.

e)
```
def index_of_median(numbers):
    """`numbers` is an array of size n"""
    # the median() from part c
    m = median(numbers)
    # the linear_search() from lecture
    return linear_search(numbers, m)
```

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

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

Problem 3. *(Eligible for Redemption)*

For each problem below, state a *tight* theoretical lower bound. Provide justification for this lower bound, and sketch an *optimal* algorithm; that is, an algorithm which has the lower bound as its worst case time complexity.

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. Algorithm: loop through the array, checking each element against the target.

a) Given an array of n numbers, find the maximum.

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

b) Given a *sorted* array of n numbers, find a median.

Solution: $\Omega(1)$. Any algorithm must take at least constant time (no justification necessary). Algorithm: return the element at the middle of the sorted array in constant time.

c) Given an array of the heights of n people, determine the height of tallest person you can make by stacking two of them.

Solution: $\Omega(n)$. Any algorithm must look through all of the heights, taking $\Omega(n)$ time. Algorithm: loop through the heights, keeping track of the largest and second largest in linear time. Return this pair.

Problem 4. *(Eligible for Redemption)*

In discussion, we learned how to express the asymptotic time complexity of algorithms which depend on *multiple* sizes. This problem will give more practice doing this.

State the asymptotic time complexity of each of the operations below using multivariate Θ notation. Your answer should include every size variable used in the problem statement, such as number of points, dimensionality, etc. You do not need to show your work.

Example: compute the distance between a point in $\mathbb{R}^d$ and n other points in $\mathbb{R}^d$.

Example Solution: $\Theta(nd)$.

a) Given a set of n points in $\mathbb{R}^d$, compute the distance between each pair of points.

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

There are $\Theta(n^2)$ pairs, and computing the distance between a single pair of points takes $\Theta(d)$ time.

b) Given a set of b blue points and a set of r red points in $\mathbb{R}^d$, find the smallest distance between a blue point and a red point using the brute force approach.

Solution: $\Theta(brd)$.

With a nested loop, we examine each pair where one point is red and the other is blue; there are br such pairs. We calculate the distance for each pair in $\Theta(d)$ time.

c) Given a set of n points in $\mathbb{R}^d$, 10% of them red and 90% of them blue, compute the smallest distances between a blue point and a red point.

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

d) Given a set $\mathcal{D}$ of n data points in $\mathbb{R}^d$ and a new point $\vec{x}$ in $\mathbb{R}^d$, find the point in $\mathcal{D}$ which is kth closest to $\vec{x}$ using the `kth_smallest` function below:

```python
def kth_smallest(numbers, k):
    """Given an array of numbers, find index of kth smallest."""
    numbers = np.copy(numbers)
    for _ in range(k):
        smallest_index = None
        smallest_value = float('inf')
        for i, x in enumerate(numbers):
            if x < smallest_value:
                smallest_index = i
                smallest_value = x
        numbers[smallest_index] = float('inf')
    return smallest_index, smallest_value
```

Solution: $\Theta(nd + nk)$.

For each point in the data set $\mathcal{D}$, we calculate its distance to $\vec{x}$ in time $\Theta(d)$. Doing this for all n points takes $\Theta(nd)$ time. Finding the kth smallest takes $\Theta(nk)$ time. These two steps together take $\Theta(nd + nk)$ time.

Problem 5.

In each of the problems below compute the average case time complexity 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)
```python
def median(numbers):
    """computes the median. `numbers` is an array of n numbers"""
    np.random.shuffle(numbers) # randomly permute `numbers` in linear time
    n = len(numbers)
    for x in numbers:
        less = 0
        more = 0
        for y in numbers:
            if y <= x:
                less += 1
            if y >= x:
                more += 1
        if less >= n/2 and more >= n/2:
            return x
```

Tip: `np.random.shuffle` randomly permutes an array in linear time. In this part, you may assume that the array has a unique median.

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

The first case is that the first element of `numbers` after shuffling is a median. The second case is that the second element is a median. And so on.

The probability of each case is $1/n$, since each is equally-likely.

Each iteration of the inner loop takes constant time; call this constant c. The inner loop runs n times per iteration of the outer loop.

In the first case – when the first element is a median – the outer loop executes once, so the inner loop iterates n times, taking time, say, cn.

In the second case – when the second element is the median – the outer loop executes two times, so the inner loop iterates $2n$ times taking total time $2 \cdot cn$.

In general, the algorithm takes time $cn \cdot k$ when the median is in position k. If we use $T(k)$ to denote the time taken on case k, we have $T(k) = cn \cdot k$.

The expected time is therefore:

$$\begin{aligned}\sum_{k=1}^{n} P(k)\cdot T(k) &= \sum_{k=1}^{n} \frac{1}{n}\cdot cn\cdot k\\ &= n\frac{c}{n}\sum_{k=1}^{n} k\\ &= c\cdot\frac{n(n+1)}{2}\\ &= \Theta(n^2)\end{aligned}$$

b)
```
def f(numbers):
    """search by randomly guessing. `numbers` is an array of n numbers"""
    n = len(numbers)
    # randomly choose a number between 0 and n-1 in constant time
    guess = np.random.randint(n)

    if numbers[guess] == max(numbers):
        for i in range(n**2):
            print(i * numbers[guess])
```

In this part, you may assume that the numbers are distinct. Be careful: does every line take constant time to execute once?

Solution: $\Theta(n)$.

We can break this into two cases: one where the guess is the largest number in the array, and one where it is not.

In the case where `numbers[guess]` is the largest number, the for-loop will execute, taking $\Theta(n^2)$ time. This probability of this case occurring is $1/n$.

On the other hand, if the guess is not the largest number, the for-loop will not execute. The time complexity will be dominated by `max`, which takes linear time. The probability of this case is $1 - 1/n$.

Therefore, the expected time is:

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