DSC 40B - Discussion 01

Problem 1.

In lecture, we analyzed dynamic arrays which had *geometric* growth rate. That is, when a resize occurred, the new size was a constant factor times the old. Now consider a *linear* growth rate, where the new size is the old size plus a constant. Show that the amortized time complexity of the append operation is $\Theta(n)$.

Solution:

Let the initial size of the array be β and the growth rate be c. Therefore, physical size of the array would grow as β, $\beta + c$, $\beta + 2c$, $\beta + 3c$, ...

The resizes occur when the logical size of the array equals the physical size of the array.

The first resize occurs when the size of the array is β.

The second resize occurs when the size of the array is $\beta + c$.

The third resize occurs when the size of the array is $\beta + 2c$ and so on.

We also know that resizing an array of size n takes time $\theta(n) = kn$ for some constant k.

We can visualize this using the following table:

Resize #	Size of the array	Time taken
1	β	$k(\beta)$
2	$\beta + c$	$k(\beta + c)$
3	$\beta + 2c$	$k(\beta + 2c)$
.	.	.
.	.	.
.	.	.
j	$\beta + (j-1)c$	$k(\beta + (j-1)c)$
.	.	.
.	.	.
.	.	.

Let x be the number of resizes required for n appends.

$$n = \beta + (x-1)c$$
$$x = \frac{n-\beta}{c} + 1 = \theta(n)$$

Let us now compute the total time taken for resizing.

$$\begin{aligned}
Total\ \ time &= \sum_{j=1}^{x} k(\beta + (j-1)c) \\
&= k\sum_{j=1}^{x} \beta + (j-1)c \\
&= k(\sum_{j=1}^{x} \beta + \sum_{j=1}^{x}(j-1)c) \\
&= k(\beta x + c\sum_{j=1}^{x}(j-1)) \\
&= k(\beta x + c\sum_{j=0}^{x-1} j) \\
&= k(\beta x + c\frac{(x-1)x}{2}) \\
&= k(\beta\theta(n) + c\frac{(\theta(n)-1)\theta(n)}{2}) \\
&= k\theta(n^2) \\
&= \theta(n^2)
\end{aligned}$$

The total time for n appends is the sum of time of growing appends and time of non-growing appends. As discussed in the lecture, the total time for non-growing appends is $\theta(n)$. Therefore, the total time for n appends is $\theta(n^2) + \theta(n) = \theta(n^2)$.

$$T_{amort}(n) = \frac{\theta(n^2)}{n} = \theta(n)$$

Problem 2.

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 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 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 the element 42 is not present in data.

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 : $\theta(n)$ when data is sorted in descending order.
> Worst case : $\theta(n^2)$ when data is sorted in ascending order or the largest element in data is at the last index.

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 numbers[0] is the median
Worst case : $\theta(n^2)$ when numbers[n-1] 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:
Best case : $\theta(n)$ when data[0] occurs more than n/2 times
Worst case : $\theta(n^2)$ when the mode occurs towards the end of the array.

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 : $\theta(n)$ when when numbers[0] is the median
Worst case : $\theta(n^2)$ when numbers[n-1] is the median