DSC 40B - Midterm Review

The questions below are indicative of what you will see on the midterm. However, note that the midterm itself will be longer. The exam will contain around 25 questions, with each taking an average of 2 to 3 minutes. The time limit for the exam will be 90 minutes.

Problem 1.

What is the time complexity of this piece of code?

```
def factorial(n):
    r = 1
    i = n
    while i > 0:
        r = r*i
        i -= 1
    return r
```

- ☐ $\theta(n^2)$
- ☐ $\theta(n^3)$
- ☐ $\theta(nlogn)$
- ☐ $\theta(n)$

Solution: $\theta(n)$

Problem 2.

What is the time complexity of this piece of code?

```
def foo(n):
    for i in range(5, 10):
        for j in range(i):
            for k in range(n):
                print(i, j, k)
```

- ☐ $\theta(n^2)$
- ☐ $\theta(n^3)$
- ☐ $\theta(nlogn)$
- ☐ $\theta(n)$

Solution: $\theta(n)$
The two outer loops run for only a constant number of iterations and the inner most loop runs for $\theta(n)$ iterations. Therefore, the total time complexity is $\theta(n)$.

Problem 3.

This piece of code returns the number of "pairs" of the form (x, -x) in a collection of numbers. What is the time complexity of this piece of code if the input is a Python list of size n?

```
def count_pairs(numbers):
    count = 0
    for x in numbers:
        if -x in numbers:
            count += 1/2
    return count
```

- ☐ $\theta(n^2)$
- ☐ $\theta(n^3)$
- ☐ $\theta(nlogn)$
- ☐ $\theta(n)$

Solution: $\theta(n^2)$
Checking if an element is in the list using the Python **in** operator takes time linear in the size of the list. Therefore, the if statement in the code takes $\theta(n)$ time. The for loop executes for $\theta(n)$ iterations. Hence, the total time complexity is $\theta(n^2)$.

Problem 4.

The below code shows the iterative version of binary search.

```
def binary_search(arr, t, start, stop):
    while start < stop:
        middle = start + (stop -start) // 2
        if(arr[middle] == t):
            return middle
        if(arr[middle]<t):
            start = middle + 1
        else:
            stop = middle
```

Let n = stop - start. What is the worst-case time complexity of this version of binary search?

- ☐ $\theta(n^2)$
- ☐ $\theta(logn)$
- ☐ $\theta(nlogn)$
- ☐ $\theta(n)$

Solution: $\theta(logn)$

Problem 5.

Here is a recursive algorithm for computing the factorial of n:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

What is the recurrence relation describing this function's run time?

Solution: T(n) = 1 + T(n-1)

Problem 6.

Suppose $f_1(n) = \Theta(n^3)$ and $f_2(n) = \Omega(n)$. Which is true about the upper bound of $f_1 + f_2$?

- ☐ $f_1(n) + f_2(n) = O(n)$
- ☐ It cannot be determined
- ☐ $f_1(n) + f_2(n) = O(n^3)$

Solution: It cannot be determined

Problem 7.

Suppose $f_1(n) = \Omega(n^2)$ and $O(n^5)$ and $f_2(n) = \Omega(n^3)$ and $O(n^6)$. Which is true about $f_1 + f_2$?

- ☐ It is $O(n^5)$ and $\Omega(n^2)$
- ☐ It is $O(n^6)$ and $\Omega(n^3)$
- ☐ It is $O(n^6)$ and $\Omega(n^2)$
- ☐ It is $O(n^5)$ and $\Omega(n^3)$

Solution: It is $O(n^6)$ and $\Omega(n^3)$

Problem 8.

If $f(n) = O(n^2)$, then $f(n) = \Omega(n^2)$

- ☐ True
- ☐ False

Solution: False
Assume f(n) = n. In this case, $f(n) = O(n^2)$, but $f(n) \neq \Omega(n^2)$

Problem 9.

If $f(n) = O(n^5)$, and $g(n) = O(n^2)$ then $f(n)/g(n) = O(n^3)$

- ☐ True
- ☐ False

Solution: False
Assume $f(n) = n^5$ and $g(n) = 1$. In this case, $f(n) = O(n^5)$, and $g(n) = O(n^2)$.
But $f(n)/g(n) = n^5 \neq O(n^3)$.

Problem 10.

The best case and worst case time complexity of merge sort is $\theta(nlogn)$

- ☐ True
- ☐ False

Solution: True

Problem 11.

The recursive calls made by mergesort are always on arrays of strictly smaller size than the input array.

- ☐ True
- ☐ False

Solution: True

Problem 12.

Consider the modified mergesort given below:

```
def mergesort(arr):
    if len(arr)>1 :
        middle=math.floor(len(arr)/2)
        left=arr[:middle]
        right=arr[middle:]
        for i in range(len(arr)):
            for j in range(len(arr)):
                print("Mergesort")
        mergesort(left)
        mergesort(right)
        merge(left, right, arr)
```

What is the time complexity of this modified mergesort?

- ☐ $\theta(n^2)$
- ☐ $\theta(n^3)$
- ☐ $\theta(nlogn)$
- ☐ $\theta(n)$

Solution: $\theta(n^2)$
The recurrence relation for the modified mergesort is $T(n) = n^2 + 2T(n/2)$. Let's try to solve the recurrence by unrolling the loop.

$$\begin{aligned}
T(n) &= 2T(n/2) + n^2 \\
&= 2[2T(n/4) + (\frac{n}{2})^2] + n^2 \\
&= 4T(n/4) + 2(\frac{n^2}{2^2}) + n^2 \\
&= 4T(n/4) + n^2(1 + \frac{1}{2}) \\
&= 4[2T(n/8) + (\frac{n}{4})^2] + n^2(1 + \frac{1}{2}) \\
&= 8T(n/8) + 4(\frac{n^2}{4^2}) + n^2(1 + \frac{1}{2}) \\
&= 8T(n/8) + n^2(1 + \frac{1}{2} + \frac{1}{4})
\end{aligned}$$

At k-th step, we have $T(n) = 2^k T(n/2^k) + n^2(1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^{k-1}})$

Base case is T(1).

$$n = 2^k$$
$$=> logn = k$$

Substituting the value of k, we get

$$T(n) = 2^{logn}T(1) + n^2(1 + 1/2 + ...)$$
$$= \theta(n^2)$$

Problem 13.

Suppose you are given n numbers in a Python list in sorted order. Describe an efficient algorithm for checking to see if there is any number in the list which occurs 42 times. Do not use a dictionary/hash map.

Solution: Iterate over the elements in the list at indices from 0 to len(list) - 42 using a for loop. For each index i, check if the element at i+41 is equal to the element at i. If yes, return the element.

Problem 14.

Consider this version of quicksort given below. It is essentially the same as that given in lecture, except that 1) it always uses the last element of the array as the pivot, and 2) it has a `print` statement inserted at a crucial place.

```
def quicksort(arr, start, stop):
    """Sort arr[start:stop] in-place."""
    if stop - start > 1:
        pivot_ix = partition(arr, start, stop, stop-1)
        quicksort(arr, start, pivot_ix)
        quicksort(arr, pivot_ix+1, stop)


def partition(arr, start, stop, pivot_ix):
    def swap(ix_1, ix_2):
        arr[ix_1], arr[ix_2] = arr[ix_2], arr[ix_1]

    pivot = arr[pivot_ix]
    swap(pivot_ix, stop-1)
    middle_barrier = start
    for end_barrier in range(start, stop - 1):
        if arr[end_barrier] < pivot:
            print('hello')
            swap(middle_barrier, end_barrier)
            middle_barrier += 1
        # else:
            # do nothing
    swap(middle_barrier, stop-1)
    return middle_barrier
```

Suppose `arr` is an array of length n with entries `[1, 2, 3, ..., n]`, where n is some large integer. If `quicksort(arr, 0, n)` is run, exactly how many times will `"hello"` be printed to the screen? Your answer should be an expression involving n, and should not involve $\sum$ or .... Show your work.

Solution: Suppose we make a call to `quicksort` on a sorted array of size k. The pivot is set to the last (and largest) element in the array, therefore the condition of the `if`-statement in `partition` always evaluates to true, printing `"hello"` $k-1$ times.

When the array is sorted, the root call to `quicksort` spawns two recursive calls; one on an array of size $n-1$ and the other on an array of size zero. The first recursive call spawns two calls; one on an array of size $n-2$ and the other on an array of size zero. And so forth. Only the recursive calls to arrays of size > 1 result in a call to `partition`. Therefore `partition` is called on arrays of size $n-1$, $n-2$, $n-3$, ..., 3, 2, and the total number of printed `"hello"`s is:

$$(n-2)+(n-3)+\ldots+3+2+1=\frac{(n-1)n}{2}.$$