Ethem Mergaaslan

200104004016

CSE 222 Hw2

Q 1)

a) $f(n) = n^2 + 7n$, $g(n) = n^3 + 7$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n^2 + 7n}{n^3 + 7} = 0$$ (converges to zero since $n^2 < n^3$)

so, $f(n) \in O(g(n))$

b) $f(n) = 12n + \log_2(n^2)$, $g(n) = n^2 + 6n$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{12n + \log_2 n^2}{n^2 + 6n} = 0$$ (converges to zero since $n < n^2$)

so, $f(n) \in O(g(n))$

c) $f(n) = n \cdot \log_2(3n)$, $g(n) = n + \log_2(8n^3)$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n \cdot \log_2(3n)}{n + \log_2(8n^3)}$$?

d) $f(n) = n^n + 5n$, $g(n) = 3 \cdot 2^n$

$$\lim \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n^n + 5n}{3 \cdot 2^n} = \infty$$ (since $n^n > 2^n$ for $n \to \infty$)

so, $f(n) \in \Omega(g(n))$

e) $f(n) = \sqrt[3]{2n}$, $g(n) = \sqrt{3n}$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{\sqrt[3]{2n}}{\sqrt{3n}} = 0$$ (since $\sqrt[3]{2n} < \sqrt{3n}$ for $n \to \infty$)

so, $f(n) \in O(g(n))$

Q2)

a)
```
static void methodA (String names[]) {
    for (int i = 0; i < names.length; i++)      → O(n)
        System.out.println(names[i]);           → O(1)
}
```

The time complexity for method A is $O(n)$.

b)
```
static void methodB() {
    String[] myArray = new String[] {"CSE222", "CSE505", "HW2"};   → O(1)
    for (int i=0; i < myArray.length; i++)      → O(n)
        methodA(myArray);                       → O(n)
}
```

Since an $O(n)$ complexity method is called in a for loop for n times, the time complexity is:

$O(n \cdot n) = O(n^2)$

c)
```
static void methodC(int numbers[]) {
    int i=0;                                    → O(1)
    while (i < numbers.length)                  → READ BELOW
        System.out.println(numbers[i]);         → O(1)
}
```

This piece of code contains an infinite loop, therefore we cannot determine an algorithmic time complexity for it.

d)

```
static void methodD(int numbers[]){
    int i=0;                                    → O(1)
    while (numbers[i] <4)                       → O(n)
        System.out.println(numbers[i+1]);       → O(1)
}
```

The while loop is executed until an element of the array is less than 4. The best case scenario is O(1) and the worst case scenario is iterating through the whole array, O(n).

The time complexity is O(n)

(Q.3)

The time complexity of the first algorithm may seem like O(1), but it is not.

Since the number of lines are dependent to the size of the array, the complexity of that algorithm is O(n)

The time complexity of the second algorithm is dependent of the size of the for loop only, since the lines inside the for loop are constant time lines (println).

So, the time complexity is the same for the algorithms. But, evaluating elements like readability, error proneness, being easily modifiable etc., the second algorithm (for loop) is more advantageous than the first algorithm.

Q4)

No, it is not possible.

The best case is O(1) and the worst case is O(n). Since we don't know if the array is sorted, we would perform a linear search on the array, and it depends on the size of the array.

So constant time is not possible.

Pseudo code for linear search

```
for i in size
    if array[i] = theNumber
        return true
return false || If the loop ends without finding the integer,
            || it returns false.
```

Note: The best case is that the element we are looking for in the array is the first element of the array, O(1)

The worst case is that the element we are looking for in the array is the last element of the array, O(n)

(Q 5)

```
int minOfA = A[0];          ---> O(1)
int minOfB = B[0];          ---> O(1)

for i in range A.length     ---> O(n)
    if A[i] < minOfA        ---> O(1)
        minOfA = A[i];      ---> O(1)

for j in range B.length     ---> O(m)
    if B[j] < minOfB        ---> O(1)
        minOfB = B[j];      ---> O(1)

return minOfA * minOfB;     ---> O(1)
```

The first two lines are for initializing the "min" variables for both arrays, assigning them the first elements of the arrays as a temporary value.

The following for loops are for iterating through the arrays seperately and finding the lowest values inside each array.

Then the product of the two minimum values are returned.

The time complexity:

Other than the constant lines, we have two for loops with time complexities $O(n)$ and $O(m)$.

In this case, the time complexity is:

$$O(n) + O(m) = O(n + m)$$