Homework 2

Question 1: for each of the function pairs below, show whether $f(n)=O(g(n))$, or $f(n)=\Omega(g(n))$, or $f(n)=\Theta(g(n))$ by using limit approach.

↳ According to the limit asymptotic theorem;

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=L,$$

1. If $L=0$, then $f(n)\in O(g(n))$
2. If $L=C>0$, then $f(n)\in\Theta(g(n))$
3. If $L=\infty$, then $f(n)\in\Omega(g(n))$

** And some limit rules;

* If $\lim_{n\to\infty}f(n)=\infty$ and $\lim_{n\to\infty}g(n)=\infty$ then $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{f'(n)}{g'(n)}$
* In polinom, degree of denominator > degree of numerator then limit is zero
* Grow rate; $x^x > x! >$ exponential > polynomials > logarithms

So, let's look

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

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{n^2+7n}{n^3+7}=0,\ \text{so}\ f(n)=O(g(n))$$

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

$$\lim_{n\to\infty}\frac{12n+\log_2 n^2}{n^2+6n}=\lim_{n\to\infty}\frac{n\left(12+\cancel{\frac{\log_2 n^2}{n}}^{\,0}\right)}{n^2\left(1+\cancel{\frac{6}{n}}^{\,0}\right)}=\lim_{n\to\infty}\frac{12}{n}=0,\ \text{so}\ f(n)=O(g(n))$$

c) $f(n)=n.\log_2 3n$ and $g(n)=n+\log_2(8n^3)$

$$\lim_{n\to\infty}\frac{n.\log_2 3n}{n+\log_2(8n^3)}=\lim_{n\to\infty}\frac{\log_2 3n+\cancel{n}.\frac{3}{3\cancel{n}}.\log_2 e}{1+\frac{24n^2}{8n^3}.\log_2 e}=\frac{\log_2 3n+\log_2 e}{1+\frac{3}{n}.\log_2 e}=\infty,$$

so $f(n)=\Omega(g(n))$

d) $f(n)=n^n+5n$ and $g(n)=3.2^n$

grow fast

$$\lim_{n\to\infty}\frac{n^n+5n}{3.2^n}=\infty,\ \text{so}\ f(n)=\Omega(g(n))$$

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

$$\lim_{n\to\infty}\frac{\sqrt[3]{2n}}{\sqrt{3n}}=\lim_{n\to\infty}\frac{\sqrt[3]{2}.\sqrt[3]{n}}{\sqrt{3}.\sqrt{n}}=\frac{\sqrt[3]{2}}{\sqrt{3}}.\lim_{n\to\infty}\frac{n^{\frac{1}{3}}}{n^{\frac{1}{2}}}=\frac{\sqrt[3]{2}}{\sqrt{3}}\lim_{n\to\infty}\frac{1}{n^{\frac{1}{6}}}=0,$$

so $f(n)=O(g(n))$

Question 2: Analyze the worst-case time complexity of the following methods.

↳ Big O

↳ * If algorithm is not dependent on input size then it has constant time. Statements like comparisons, assignments is $O(1)$ time complexity.

* In for loop, running time is dependent running time of the statements in the for loop times the number of iterations.

* And, when finding time complexity,

1. find the fastest growing term. 2. take out the coefficient

So,

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

$T_a = n.O(1)$

$T_a = n.c$

$T_a = O(n)$ //

b)
```
static void methodB(){
    String[] myArray = new String[]{"CSE222", "CSE505", "HW2"};   → O(1)
    for(int i=0; i< myArray.length; i++)   → 3 times (fixed size)
        methodA(myArray);                  → MethodA ~ 3.O(1) = O(1)
}
```

* Here we know the size of array. So, array size is fixed which is 3.

$T_b = O(1) + 3.O(1) = O(1)$ //

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

→ If we incremented the value of i inside the loop, we could say $O(n)$ for the time complexity of code. BUT here, the value of i is not increased and therefore cannot exit the loop. It becomes an infinite loop. That's why we don't specify the time complexity of this code.

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

→ Here we check the value of the element inside the array while looping. If all the elements in the loop are less than the specified number, there will be as many iterations as the number of elements and the worst case time complexity will be $O(n)$. BUT here, in the code there is no check for i, which at worst mean that if all the numbers in the array are less than the specified number, the loop will continue, which will cause an error. So the codes gives an error. That's why we cannot specify the time complexity of this code.

Question 3: What is the difference between the time complexities of the following methods? Which one is more advantageous?

↳
```
static void withoutLoop(int[] myArray){
    int i=0
    System.out.println(myArray[i++]);
    System.out.println(myArray[i++]);
    System.out.println(myArray[i++]);
    System.out.println(myArray[i++]);
    System.out.println(myArray[i++]);
    /*
    assume that the System.out.println is
    called myArray.length times in total
    */
    System.out.println(myArray[i++]);
}
```

The time complexity of the code is $O(n)$. There doesn't seem to be a loop like a for loop here, but i is incremented each time and iterated as much as the number of elements in the array. So it depends on the input size.

```
static void witLoop(int[] myArray){
    for(int i=0; i<myArray.length; i++)      ⟶ n times
        System.out.println(myArray[i]);      ⟶ O(1)
}
```

$T_2 = n \cdot O(1)$

$T_2 = n \cdot c$

$T_2 = O(n)$

As a result, the time complexity of both methods is the same which is $O(n)$. If we look at which one is more advantageous except for the time complexities, the witLoop method is more advantageous in terms of such as readability, writability.

Scanned with CamScanner

Question 4 : Consider an array of n integers ($n \in \mathbb{Z}^+$). You do not have any information on whether the array is sorted or not, and you are supposed to check if the array contains a specific integer. Considering all possible inputs, can you solve this problem in constant time ? If so, write down the pseudo code of the algorithm and analyze its time complexity. If not, explain why ?

↳ Now, let's assume that our array is not sorted. And we're trying to find the number 8.

Array = {3, 5, 4, 2, 7, 1, 9, 6, 8}

start → 3, 5, 4, 2, 7, 1, 9, 6, 8
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
$3 \neq 8$ × × × × × × × ✓

We must look at all the elements in the array until we find the number we are looking for. In other words, we will start from the first element of the array and proceed in order. In the worst case scenario, the number we are looking for may be the last element of the array. So, if we call the number of elements of the array n, we will have to look at it n times, its time complexity will be $O(n)$.

Now, suppose to our array is sorted.

Array = {1, 2, 3, 4, 5, 6, 7, 8, 9}

1, 2, 3, 4, 5, 6, 7, 8, 9
× ↑ × ↑ ↑
$5<8$ $7<8$ $8=8$ ✓

This time, instead of looking at all the elements of the array, we first look at whether the number we are looking for is greater or less than the middle number of the array and stop looking at the remaining half of the array accordingly. And in this way, we reduce our iteration count by comparing it to the middle element each time. So, the time complexity of this one is slightly better than the other, which is $O(\log n)$.

But it is not possible to find the number we are looking for in constant time. Because in order to find the number we are looking for, we need to compare whether it is sorted or not. And we cannot do this in constant time.

Question 5: Consider two integer arrays A and B as follows:

$$A = [a_0, a_1, \dots, a_{n-1}]$$

$$B = [b_0, b_1, \dots, b_{m-1}]$$

where $n, m \in \mathbb{Z}^+$. Design a linear time algorithm to find the minimum value of $a_i \cdot b_j$ where $0 \le i < n$ and $0 \le j < m$. Explain your algorithm (along with the pseudo-code) and analyze its worst-case time complexity.

↳ To find the smallest value of $a_i * b_j$, we need to find the smallest element of both arrays.

So,

1. Assign first elements of both arrays to variables min1 and min2.
2. Compare min1 to other elements in the array A and assign smallest element to min1.
3. Compare min2 to other elements in the Array B and assign smallest element to min2.
4. Multiply min1 and min2

Pseudo Code

```
min1 = A[0]  ---> O(1)
min2 = B[0]  ---> O(1)
while i < A.length ---> n times
    if A[i] is smaller than min1
        min1 = A[i] ---> O(1)
while j < B.length ---> m times
    if B[j] is smaller than min2
        min2 = B[j] ---> O(1)
res = min1 * min2 ---> O(1)
```

Time complexity $= O(1) + n.O(1) + m.O(1) + O(1)$

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