CSE222 Data Structures and Algorithms
Homework 2
Furkan Taşkın ~~200104004072~~

1) a) $f(n) = (n^2-3n)^2$ and $g(n) = 5n^3+n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(n^2-3n)^2}{5n^3+n} = \frac{n^4-6n^3+9n^2}{5n^3+n} = \frac{4n^3-18n^2+18n}{15n^2+1}$$

L'hospital rule $= \frac{12n^2-36n+18}{30n} = \frac{24n-36}{30} = \lim_{n\to\infty} = \frac{6n-12}{10} = \infty$

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

b) $f(n) = n^3$ and $g(n) = \log_2 n^4$

$$\lim_{n\to\infty} \frac{n^3}{\log_2 n^4} = \frac{n^3}{4\log_2 n} = \frac{3n^2}{\frac{4}{n \cdot \ln 2}} = \frac{3n^3 \cdot \ln 2}{4} = \infty \Rightarrow f(n) \in \Omega(g(n))$$

(it will go to infinity)

$\log_a(x) \to \frac{1}{x \cdot \ln(a)}$

c) $5n \cdot \log_2(4n)$ and $g(n) = n \cdot \log_2(5^n)$

$$\lim_{n\to\infty} \frac{5n \cdot \log_2(4n)}{n \cdot \log_2(5^n)} = \frac{5n \cdot (2+\log_2 n)}{n^2 \cdot \log_2 5} = \frac{5(2+\log_2 n)}{n \cdot \log_2 5} = \frac{10+5\log_2 n}{n \cdot \log_2 5}$$

L'hospital Rule. We can ignore 10 and $\log_2 5$ because it won't effect.

$\frac{\log_2 n}{n} =$ it will go 0

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

d) n^n and $g(n) = 10^n$

$\frac{n^n}{10^n}$ → $\ln(n^n/10^n) = n\ln(n) - n\cdot\ln 10 = n(\ln n - \ln 10)$

take natural logarithm

take derivative

$\lim_{n\to\infty} \ln(n) - \ln 10 + 1 = \infty$ $\quad f(n) \in \Omega(g(n))$

$1\cdot(\ln(n) - \ln 10) + \frac{1}{x}\cdot n = \ln(n) - \ln 10 + 1$

e) $f(n) = \sqrt[5]{2n}$ $\quad g(n) = n\cdot\sqrt[3]{n}$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{\sqrt[5]{2n}\cdot(2n)^{1/5}}{n\cdot n^{1/3}} \to \frac{16n^{6/5}}{n^{4/3}} = 16\cdot n^{\frac{6}{5}-\frac{4}{3}} = 16n^{\frac{18-20}{15}} = 16n^{-\frac{2}{15}}$

$\lim_{n\to\infty} = \frac{16}{n^{\frac{2}{15}}} = 0 \Rightarrow f(n) \in O(g(n))$

2) a) Since the array has a length of n, loop runs n times. Within the loop assigns an empty string to each element of the array. It takes constant time. The loop executes n times and each iteration performs a constant time operation. Worst-case time complexity of the method is O(n).

str_array[0] = ""
str_array[1] = ""
str_array[n] = ""

Loop puts empty string on every element of an array n times

b) First for loop iterates n times. Inside the loop methodA is called which iterates the same loop n times. So there is nested for loop. So, complexity of nested loop is $O(n^2)$. methodB also includes an another for loop iterates every element of the array and prints it. It has time complexity of O(n). So worst-case time complexity is $O(n^2) + O(n)$ but we take the dominant term which is $O(n^2)$.

c) There are nested for loops each iterating n times. Inside loops methodB is called with same array, MethodB has $O(n^2)$ time complexity itself. This means total time complexity becomes $n^2 \cdot n^2 = O(n^4)$

Suppose n=2.

Outer loop (i=0)
- inner loop (j=0)
  - methodB called
- Inner loop (j=1)
  - methodB called

Outer loop (i=1)
- Inner loop (j=0)
  - methodB called
- Inner loop (j=1)
  - methodB called

With an array of size 2, methodC calls methodB a total of four times. MethodB itself has a nested loop with $O(n^2)$ complexity.

---

d) The code goes to infinity therefore we can not talk about worst-case time complexity. Big O notation is primarily used to analyze algorithms that terminate after a finite number of steps. There is no relationship between input size and number of steps in this case.

Since in str_array[i--] = "a"; i is decremented and in the loop body i is incremented it creates an infinite loop.

e) Worst-case scenario would be if an array doesn't contain any empty strings. Then loop iterates entire array which has length of n, so worst-case time complexity is O(n).

str_array[0] = " ";
str_array[1] = " ";
⋮
str_array[n] = " ";

It checks until end of the array if empty string is not in anywhere of str_array which is worst-case scenario.

3). Ascending Order

```
//Assumes the array is already sorted in ascending order.
FUNCTION calcMaxDiff(array, arraySize)
    maxDifference = array[arraySize-1] - array[0]
    RETURN maxDifference
```

calcMaxDiff function takes an array and it's size as input. It assumes given array is in ascending order. Logic is based on first element is the smallest and last element is the largest. If we substract first element from last element we get the maximum difference on array. Worst-case time complexity of this algorithm is O(1). Because number of operations does not increase with the size of the input array. It will do the same operation if array has 7 or 70.000 element.

Array is not sorted.

```
FUNCTION calcMaxDiffUnsorted(array, arraySize)
    maxDifference = array[1] - array[0]
    minElement = array[0]

    FOR i FROM 1 TO arraySize - 1 DO
        IF array[i] - minElement > maxDifference THEN
            maxDifference = array[i] - minElement
        IF array[i] < minElement THEN
            minElement = array[i]
    RETURN maxDifference
```

First initialize maxDifference with the difference between the first 2 elements and sets minElement to the first element. Inside the loop code checks if the difference between the current element (array[i]) and the smallest element seen so far (minElement) is greater than the current maxDifference. If so, maxDifference is updated. Also minElement is keep tracked. Worst-case time complexity is O(n). One loop iterates through the array. Inside the loop constant time operations are done.