Assignment 2. • PRD190001 •

Q1. In the definition of Big-O why is the for $N \geq n_0$ needed?

→ $N \geq n_0$ is needed as n_0 is a threshold value, after which a bigger function will always be above the smaller one, but before n_0 it can be ambiguous.

Q2. If $f1(N) = 2N$ & $f2(N) = 3N$. why are they both $O(N)$ since $3N$ is larger than $2N$ for $N \geq 1$?

→ According to the definition of Big-O

$f(n) = O(g(n))$ if there exists a positive integer n_0 and a postive constant c, such that

$f(n) \leq c \cdot g(n) \; \forall n \geq n_0$

So in $f1(N) = 2N$; $2N \leq c \cdot g(n)$ when $c \geq 2$

$f2(N) = 3N$; $3N \leq c \cdot g(n)$ when $c \geq 3$

∴ constants do not matter in the world of Big-O and both the functions are $O(N)$

Q3 a) $f1(N) = 2N$ & $f2(N) = 3N$

calculate $f1(5)$ & $f2(5)$

$f1(10)$ & $f2(10)$

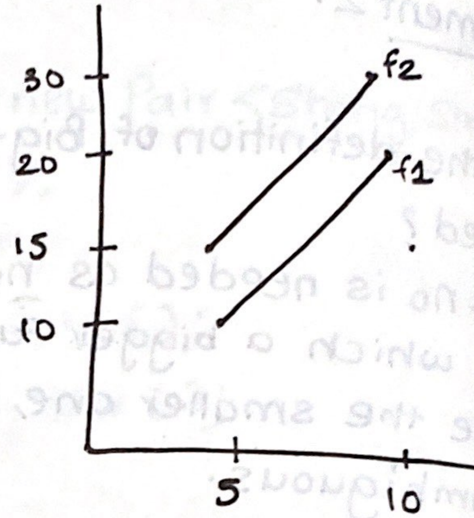
when N was doubled in each case what happened to the result. Explain why?

→ Answer on next page.

→ $f1(5) = 10$ $f2(5) = 15$

$f1(10) = 20$ $f2(10) = 30$

When n was doubled, so was the result due to the linearity.



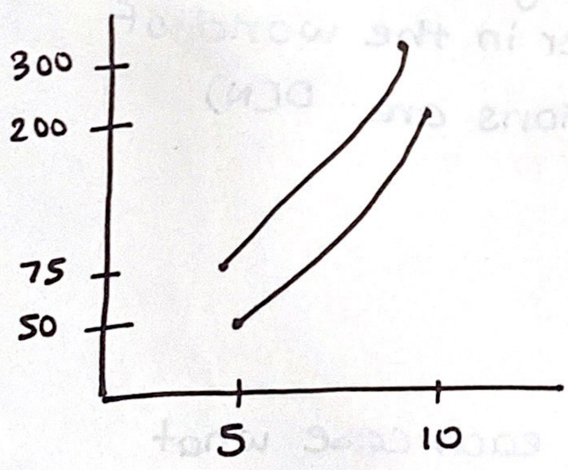
Q3b) For $f1(N) = 2N*N$ & $f2(N) = 3N*N$:

Calculate $f1(5)$ & $f2(5)$ Then $f1(10)$ & $f2(10)$

When N was doubled in each case, what happened to the result?

→ $f1(5) = 2*5*5 = 50$ $f2(5) = 3\times5\times5 = 75$

$f1(10) = 2*10*10 = 200$ $f2(10) = 3\times10\times10 = 300$



The results are quadrupled as the function is quadratic.

Q4) Since Big-O notations is a mathematical tool for functions like f(N) or g(N). How is it applicable to algorithm analysis.

→ The Big O function gives the worst case analysis ie an upper bound of a function. Since algorithms can be defined as functions with input, and the size of the input being N. Big-O will be a great tool to estimate an algorithms upper bound. Thus f(N) or g(N) are estimates of time as input sizes change.

Q5. which grows faster 2^n or n! ? Explain why?

→ n! will grow faster as when n is huge, n! is multiplied by number close to it ie. $n \times (n-1)(n-2) \times (n-3) \dots 1$ but 2^n is $2 \times 2 \times 2 \times 2 \dots n$. Thus n! increases at a factor of n and n! will grow faster. Though factorial might start slow but as numbers get big, it will be way more than 2^n.

eg:-

$10! = 3628800$

$2^{10} = 1024$

Q6. Give the Big O notation of following expressions?

a) $4n^5 + 3n^2 - 2 \rightarrow O(n^5)$

b) $5^n - n^2 + 19 \rightarrow O(5^n)$

c) $(3/5) * n \rightarrow O(n)$

d) $3n * \log(n) + 11 \rightarrow O(n \log n)$

e) $[n(n+1)/2 + n]/2 \rightarrow O(n^2)$

Q7 What is the Big-O running time for this code? Explain your answer.

```
for (int i=0; i<numItems; i++)
    System.out.println(i+1)
```

→ O(n)

For loop is the number of iterations multiplied by the statement inside.

Thus O(1) * O(numItems)

= O(numItems)

hence O(n)

Q8

```
for (int i=0; i<numItems; i++)
    for (int j=0; j<numItems; j++)
        System.out.println((i+1) x (j+1))
```

→ $O(n^2)$. Outer loop is until numItems x Inner loop until numItems x constant. Thus $O(\text{numItems}^2) = O(n^2)$

Q9)
```
for (int i=0 ; i<numItems + 1 ; i++)
        for(int j=0; j<2* numItems; j++)
                System.out.println((i+1) * (j+1))
```

→ Again, for loop is the number of iterations multiplied by the statement inside.
Outer loop goes → numItems time
Inner loop goes → 2 × numItems
& last statement is constant

Thus

→ $O(numItems) \times O(2 \times numItems) \times O(1)$

→ $O(2 \times numItems^2)$

→ $O(numItems^2)$

→ $\underline{\underline{O(n^2)}}$

Q10)
```
if (num < numItems)
        for (int i=0; i<numItems; i++)
        {
            System.out.println(i);
        }
    else
        System.out.println("too many");
```

→ O(n). As for conditionals we consider the running time of maximum condition & the forloop inside if statement is executed numItems time. Thus $O(numItems) * O(1) + \underset{\text{if condition}}{O(1)}$

$= O(numItems) = \underline{\underline{O(n)}}$

Q11) int i = numItems;

```
while (i>0)
    i = i/2;
```

→ O(logn) As it takes constant time to divide i by 2 in each iteration. The problem size is reduced by fraction in some constant time. Thus it becomes

$O(1) + O(\log n) = \underline{O(\log n)}$

Q12)

```
public static int div (int numItems)
{
    if (numItems == 0)
        return 0;
    else
        return numItems % 2 + div(numItems/2)
}
```

→ In conditionals we evaluate running time of the longer taking condition so that the else statement here.

Every time the recursion calls itself it divides (numItems/2) thus at every call the problem size is reduced by a fraction in constant time.

This is $\underline{O(\log n)}$