ASSIGNMENT #2

SUBMITTED BY :

NAME : SAI KIRAN TADURI

NETID : sxt161730

(1) In the definition of Big-O, why is the "for $N >= n_0$" needed?

(A) The "for $N >= n_0$" is needed because the graphs of the algorithms can be abnormal or ambiguous but after n_0 the bigger function should always be above the smaller one. We only care what happens when N gets large, so we can ignore things that only happen at small N.

(2) Given, $f_1(N) = 2N$ $f_2(N) = 3N$.
By the definition of Big-O, we know that $T(N) = O(f(N))$ which means

that $T(N) <= cf(N)$ for some constant c and for $N >= n_0$.

Given $f_1(N) = f_2(N) = O(N)$

$\Rightarrow 2N = 3N = O(N)$

They are not $O(N)$ because $3N$ is larger than $2N$ for $N >= 1$, but

$f_1(N) <= c \cdot f(N)$

$2N <= 2 \cdot N$

Therefore, for $c = 2$,

$f_1(N) = O(N)$.

Similarly, for $f_2(N) = 3N = O(N)$

Therefore, both expressions have different coefficients which does not effect the Big-O. ∴ Coefficient are considered to be insignificant.

Therefore, $3N = 2N = O(N)$.

(3) (a) For $f_1(N) = 2N$ and $f_2(N) = 3N$

$f_1(5) = 10$ and $f_2(5) = 15$

$f_1(10) = 20$ and $f_2(10) = 30$

We know that,

$f(ax) = a f(x)$

as f is a linear function.

Therefore, when N was doubled, f_1 and f_2 are both doubled, as they are linear.

(b) For $f_1(N) = 2N * N$ and $f_2(N) = 3N * N$

$f_1(5) = 2 * 5 * 5 = 50$;

$f_2(5) = 3 * 5 * 5 = 75$;

$f_1(10) = 2 * 10 * 10 = 200$;

$f_2(10) = 3 * 10 * 10 = 300$;

Therefore when N was doubled, f_1 and f_2 grow at quadratic rate, they are both quadrupled.

(4) "Since the big O notation defines an upper bound of an algorithm, it bounds a function only from above.

We can think N as the "size" of the input, and f(N) as an estimate for the amount of time it takes the algorithm to output an answer given an input of that size.

(5) Which grows faster, 2^n or $n!$? Why?

$n!$ grows faster than 2^n.

$n! = n(n-1)(n-2) \ldots 2.1$

for $n = 4$,

$2^n = 16$.

$n = 8$,

$2^n = 256 = (16)^2$

Therefore,
whenever n doubles, the running time squares.

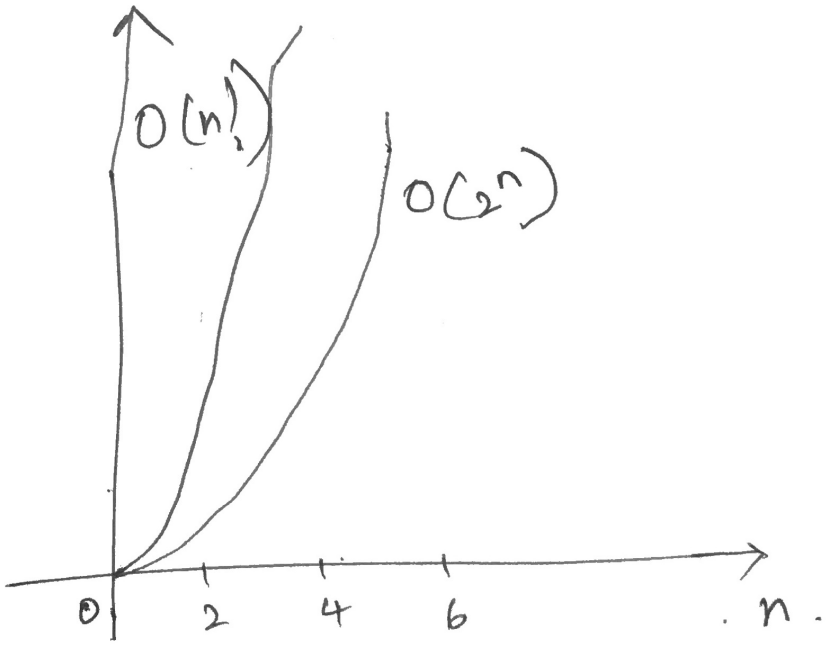
for $n = 4$,

$$n! = 24.$$

$n = 5$,

$$n! = 120 = 5 \times 4!$$

Therefore, whenever n increased by 1, the running time increased by n.



6 (a) $4n^5 + 3n^2 - 2 = O(n^5)$

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

(c) $(3/5) * n = O(n)$

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

(e) $(n(n+1)/2 + n)/2 = O(n^2)$

7 Let numItems = n;

1 for (int i=0; i<numItems; i++)

2. System.out.println(i+1);

Step 1 : for → int i=0 — 1 unit

for → i<numItems — n+1

for → i++ — n units

Step 2 : for, System.out.println(i+1) — n

Therefore,

1 + n+1 + n + n = 3n + 2, so the result is O(n)

(8) 1 for (int i=0 ; i<numItems ; i++)

2 for (int j=0; j<numItems; j++)

3 System.out.println((i+1) * (j+1));

Step 1: for,

int i=0 — 1

i<numItems — n+1

i++ — n

Step 2: for,

int j=0 — n

j<numItems — (n+1)n

j++ — n*n

Step 3: System.out.println((i+1) *(j+1)); — n^2

Total time = $1 + n+1 + n + n(n+1) + n^2 + n^2$

$= 3n^2 + 4n + 2$

$= O(n^2)$

(9)

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

Step 1:

```
int i=0;          1 unit
i <numItems+1;    n+2
i++;              n+1
```

Step 2:

```
int j=0;          n+1
j<2*numItems;     (n+1)[2(n)+1]
j++;              (n+1)[2(n)]
```

Step 3:

```
System.out.println((i+1) * (j+1));
                  (n+1)[2(n)]
```

Total time =

$1 + n+2 + n+1 + n+1 + n+1 + 2(n+1)^2 + 2(n+1)^2 + 2(n+1)^2$

(10)

Total time :

$$1 + n+2 + n+1 + n+1 + n+1 + 2n(n+1) + 2n(n+1) + 2n(n+1)$$

$$= 6n^2 + 10n + 6$$

$$= O(n^2)$$

(10)

```
if (num < numItems)                    1 unit
  for (int i=0; i<numItems; i++)       1+n+1+n
  {
     System.out.println(i);            n
  }
else
  System.out.println("too many");      1 unit.
```

For if-else statements, the test plus the larger of the two branches are considered in Big-O analysis.

In the above example, the if branch is larger so we consider only if branch for our analysis

Therefore, total time =

$$1 + 2n + 2 + n$$

$$= 3n + 3$$

$$= O(n).$$

(11)

```
int i = numItems;          1 unit
while (i > 0)              log n + 2
    i = i/2;               log n + 1
```

Consider numItems = 8;

i = 8.

while (i > 0)

1. while (i > 0) 8 > 0
   i = i/2; 4
2. while (i > 0) 4 > 0
   i = i/2; 2
3. while (i > 0) 2 > 0
   i = i/2; 1
4. while (i > 0) 1 > 0
   i = i/2; 0
5. while (i > 0) 0 ≯ 0

$\log_2 8 = 3$

Therefore,

Total time = $1 + \log n + 2 + \log n + 1$

$= 4 + 2\log n$

$= O(\log n)$

(12)

```
public static int div(int numItems)
{
    if (numItems == 0)        - 1 unit
        return 0;
    else
        return numItems % 2 + div(numItems/2);
}                             1 unit + 1 unit + T(n/2)
```

$T(0) = T(1) = 1$

$T(N) = T(n/2) + 3$ (3 for ft, add and % operation)

$T(2) = T(1) + 3 = 4$

$T(4) = T(2) + 3 = 7.$

$T(8) = T(4) + 3 = 10$

$T(16) = T(8) + 3 = 13$

$T(32) = T(16) + 3 = 16$

$T(64) = T(32) + 3 = 19$

$T(128) = T(64) + 3 = 22$

⋮ For large numbers $\approx \log n$

Each time the function call generates one more recursive call and each time numItems is halved by constant time so it is

$O(\log n)$.