1) BIG O Definition:

....

$T(N) = O(f(N))$ if there are positive constants c and $n0$ such that $T(N) \le cf(N)$ when $N \ge n0$. ......(1)

The idea of the above definition is to establish a relative order among functions. Given two functions, there are usually points where one function is smaller than the other function, so it does not make sense to claim, for instance, $f(N) < g(N)$ for all N. Thus, we compare their relative rates of growth.
For example, let $f(x) = 6x^4 - 7x^3 + 5$, and suppose we wish to simplify this function, using O notation, to describe its growth rate as x approaches infinity. The growth of $F(x)$ mainly depends on the term with highest degree i.e x^4. We can write $f(x)=O(x^4)$.
Let us prove this for all $X \ge 1$.

$$|6x^4 - 7x^3 + 5| \le 6x^4 + |7x^3| + 5$$
$$\le 6x^4 + 7x^4 + 5x^4$$
$$= 18x^4$$

so

$|6x^4 - 7x^3 + 5| \le 18x^4$ for all $x \ge 1$. But there may be some values of $x<1$ such that $|6x^4 - 7x^3 + 5| > 18x^4$.

Here in the above example $T(N)=6N^4-7N^3+5$, $f(N)=N^4$, $c=18$ and $n0=1$.

---

2) Generally Running time calculations of algorithms mainly consider the behaviour of functions for very large values and not for small values. And for large values of N, constants will not show any considerable impact on the behaviour of the functions. Hence the constants are ignored.

Let us compare the values of the given functions $f1(N)=2N$ and $f2(N)=3N$

N	f1(N)=2N	f2(N)=3N
1	2	3
2	4	6
10	20	30
100	200	300
1000	2000	3000
10000	20000	30000
100000	200000	300000
1000000	2000000	3000000

From the above observations we can say that the values of F1 and F2 increase linearly with repect to N. So inorder to simplify the problem we ignore leading constants and we just consider the behaviour of the function. Big O is just concentrating on the long-term growth rate of functions, rather than their absolute magnitudes. As the impact is just because of the highest degree of N, but not by the constants.
As we are considering the behaviour the graphs of both $f1(N)=2N$ and $f2(N)=3N$ are linear. Similar to the graph of $f(N)=N$, just the slope differs.
Hence in Big O notation $f1(N)=2N$ and $f2(N)=3N$ are considered as $O(N)$. As the behaviour is just Linear.

---

3)a)

Given f1(N)=2N and f2(N)=3N

f1(5)=2*5=10 and f2(5)=3*5=15

f1(10)=2*10=20 and f2(10)=3*10=30

By doubling the value of N, the value of both f1(N) and f2(n) are doubled i.e 10 to 20 and 15 to 30 in this case. Since Big-O(N).

3)b)

Given f1(N)=2N*N and f2(N)=3N*N

f1(5)=2*5*5=50 and f2(5)=3*5*5=75

f1(10)=2*10*10=200 and f2(10)=3*10*10=300

By doubling the value of N in each case, the value of both f1 and f2 are quadrupled. i.e multiplied by 4. Since Big-O(N^2).

---

4) The running time of algorithms will generally be exponential(2^n,5^n), logarithmic(logN, NlogN), polynomial(Linear, Quadratic, cubic) etc.

The above functions are mathematical functions which can be represented as mathematical graphs. So inorder to represent these functions we require a mathematical tool to analyse these algorithms. Hence we use Big-O notation which is a mathematical tool.

---

5)

n! grows much faster than 2^n for all n>=4.

Let us compare the values of both the functions

n	n!	2^n
1	1	2
2	2	4
3	6	8
4	24	16
5	120	32
6	720	64

As the value of n increases we can observe drastic increase in n! when compared to 2^n.

The reason behind this drastic increase is, in 2^n we are multiplying '2' n times while in n! we are multiplying the numbers succesively from 1 to n.
So, if the value of n is large the value of n! grows faster when compared to 2^n.

----

6)

a) 4n^5 + 3n^2 -2 = Big-O(n^5) i.e Polynomial of degree 5

b) 5^n - n^2 + 19 = Big-O(5^n) i.e Exponential (As exponential > Quadratic in relative order)

c) (3/5)*n = Big-O(n) i.e Linear

d) 3n * log(n) +11 = Big-O(nlogn)

e) [n(n+1)/2 + n] / 2 = Big-O(n^2) i.e Quadratic

----

7) Given code

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

From the rules of algorithm analysis, we know that the running time of for loop is Big-O(numItems).

explanation:

```
for (int i=0; i<numItems; i++){        //[1 + (numItems+1) + numItems] units
                                       // 1 for initialization + (numItems+1) for comparison + numItems for increment.

        System.out.println(i+1);       // [2*numItems] units

}
```

= 1+2numItems+1+2numItems = 4numItems+2 = Big-O(numItems)

----

8) Given code

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

From the rule-1 we know that the running time of Nested-for loop is Big-O(numItems^2).

explanation:

```
for (int i=0; i<numItems; i++){         //1 + (numItems+1) + numItems = 2numItems+2


         for (int j=0; j<numItems; j++){        // [1 + (numItems + 1) + numItems]numItems = (2numItems+2) * (numItems) = 2numItems^2 + 2numItems.


             System.out.println( (i+1) * (j+1) );    // 4 * (2numItems+2) * (numItems) = 8numItems^2 + 8numItems

        }
}
= 10numItems^2 + 12numItems + 2 = Big-O(itemNums^2).
```

----

9) Given code

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

From the rule-2 we know that the running time of Nested-for loop is Big-O(numItems^2).

explanation:

```
for (int i=0; i<numItems+1; i++){ 1 + (numItems + 1 + 1) + numItems + 1 = 2numItems + 4


              for (int j=0; j<2*numItems; j++){       [1 + (2*numItems + 1) + 2*numItems]*(numItems + 1) = (4numItems + 2) * (numItems + 1) = 4numItems^2 + 6numItems + 2
```

```
                System.out.println( (i+1) * (j+1) );    4*(4numItems^2 +
6numItems + 2)= 16numItems^2 + 24numItems + 8.

            }
      }
= 20numItems^2 + 32numItems + 14 = Big-O(numItems^2).
```


----

10) Given code,

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

From the rule-4, we know that for if-else statements then test condition plus the larger of the two branches. So the running time would be Big-O(n).

```
      explanation:
      ...........

      if ( num < numItems ){            // 1


                for (int i=0; i<numItems; i++){         // 2numItems + 2

            System.out.println(i);      // numItems


            }
      }
      else{

            System.out.println("too many");        // 1

      }
```

= 3n + 3 = Big-O(n). Condition plus only the largest part of if-else is considered.

----

11) Given code,

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

From the rules of algorithm analysis the running time Big-O(log(numItems)).

explanation:
....

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

   i = i / 2;               // log(numItems)

}
```

=2log(numItems) + 2 = Big-O(log(numItems))

----

12) Given code,

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

From the rules of algorithm analysis the running time of the above code is Big-O(log(numItems)).

explanation:
....

```
public static int div(int numItems){

      if (numItems == 0){    // 1unit*log(numItems)

            return 0;
      }                                                   // and for
numItems=0 running time is 2.
      else{

      return numItems%2 + div(numItems/2);    // 4log(numItems)

      }
}
```

= 5log(numItems) + 2 = Big-O(log(numItems))

