```
1)public static void two(int n)
   {
      if(n > 0)    //exit condition
      {
         System.out.println("n: " +n);     //potentially n operations
         two(n - 1);     //n-1 times
         two(n - 1);     //n-1 times
      }
      else if (n < 0)    //exit condition
       {
          two(n + 1);    //potentially n times
          two(n + 1);    //potentially n times
          System.out.println("n: " + n);  //potentially n times
       }
   }
```

GRF is 5n.

O(n)

```
2) public void three(int n)

{
  int i, j, k;     //3
  for (i = n/2; i > 0; i = i/2)      //log n
      for (j = 0; j < n; j++) // n^2
            for (k = 0; k < n; k++) //  (n-1)^3
                  System.out.println("i: " + i + " j: " + j+" k: " + k);
//n-1 times

}  // end three
```

GRF = n^3+n^2+n-1+logN

O(n^3)

```
3) public static void four(int n)

{
    if (n > 1)   //termination condition   //exit condition
    {
       System.out.println(n);   //1
       four(n-1);     //n-1 times
    }
    for (int i = 0; i < n; i++)   //n times
      System.out.println(i);      //n times
}
```

GRF = 2n+n-1+1

O(N)

Explanation:

In order to get your growth rate function, we count how many times a particular line of code is executed, given an arbitrary input size n.

Once we have the GRF, we can then approximate the time complexity by dropping the constant terms (terms that are added or subtracted) and lower order terms (if you have n^3 and n^2, n^2 is dropped because it’s a lower order than n^3). Whatever you have left, is your big-oh notation time complexity.

For all of my examples, I first looked for the GRF, then dropped the constant terms and lower order terms, in order to get the big-oh notation time complexity.