Exercise 1: Complete these sentences..

1. Algorithms with time complexities such as n and 100n are called _____ algorithms. Ans: Linear
2. Algorithms with time complexities such as n^2 are called quadratic-time algorithms (True or False). Ans: True
3. Any quadratic-time algorithm is eventually more efficient than any linear-time algorithm (True or False). Ans: False
4. Functions such as $5n^2$ and $5n^2 + 100$ are called _____ functions. Ans: Quadratic Functions

Exercise 2: Order of Growth Classifications

Rank the functions below according to their growth - marking 1 for the slowest growing functions (or fastest algorithm) to 7 for the fastest growing function (or slowest algorithm).

T(N)	Growth function
n^2	6
480	2
2^n	7
logN	3
2^4	1
380N	5
1/2N	4

Same thing:

T(N)	Growth function
N logN	4
N^4	7
2^n	9
$\log_8 N$	2
$n\log_4 N$	6
$\log_2 N$	3
$n\log_6 N$	5
300	1
$6N^3$	8

*note: when speaking asymptotically, the base of logarithms is irrelevant. This is because of the identity loga b logb n = logan

Which kind of growth best characterizes each of the functions below?

T(n)	Constant	Linear	Polynomial	Exponential
1	Constant			
$2n^3$			Polynomial	
(4/3)n		Linear		
2^n				Exponential
$4n^2$			Polynomial	
5600	Constant			
2493n		Linear		
$3/2^n$				Exponential

Simplify the following functions into their closest Big O growth factors

Remember the rules to find the asymptotic behavior of the following functions:

- Drop lower-order terms
- Drop constant factors
- Keep the terms that grow the fastest.
- Use the smallest possible class of functions

Examples:
Say "2n is O(n)" instead of "2n is O(n2)"
Say "3n + 5 is O(n)" instead of "3n + 5 is O(3n)
f(n) = 5n + 12 => O(n)

Try these ones yourself:

1. f(n) = 5n + 12
   O(n)
2. f(n) = 109
   O(109)
3. f(n) = n^2+ 3n + 112
   O(n^2)
4. f(n) = n^3 + 1999n + 1337
   O(n^3)

Rules for classifying functions

For Loop: The running time of a for loop is at most the running time of the statements inside the loop times the number of loops

for (int I = 0; I < n; i++){
doSomething();
}
	O(n)

Nested loops: Do the analysis inside out. The total running time of a statement inside a group of nested loops is the running time of the statement times the product of the sizes of all the loops

for (int i = 0; i < n; i++){
For (int j = 0; j <n; j++){
doSomething();
}
}
	$O(n^2)$

Consecutive Statements / Functions
These simply add - the maximum is the one that counts.

for (int I = 0; I < n; i++){
doSomething();}
 	
for (int i = 0; i < n; i++){
For (int j = 0; j <n; j++){
doSomething();
} }
	O(n)
	
	$O(n^2)$
	
	T(N) = Total = $O(n) + O(n^2) = O(n^2)$

What is the complexity of the functions below?

```
int sum = 0;
for (int i = 1; i < N; i *= 2)
   for (int j = 0; j < N; j++)
      sum++;
```

$O(N^2)$

```
function isEven(value){
if (value % 2 == 0){
return true;
}
else
return false;
}
```

O(1)

```
arrayMax(A, n) {
currentMax = A[0]
For (i=1; i< A.length; i++){
If (A[i] > currentMax) then
currentMax = A[i]
}
```

O(A.length)

```
function areYouHere(arr1, arr2) {

   for (let i=0; i<arr1.length; i++) {
       const el1 = arr1[i];

       for (let j=0; j<arr2.length; j++) {
           const el2 = arr2[j];

           if (el1 === el2) return true;
       }

   }
   return false;
}
```

O(arr1.length*arr2.length)

```
function isPrime(n) {

   if (n < 2 || n % 1 != 0) {
       return false;
   }

   for (let i = 2; i < n; ++i) {

       if (n % i == 0) return false;
   }
   return true;
}
```

O(n)

```
function findRandomElement(arr) {
   return arr[Math.floor(Math.random() * arr.length)];

}
```

O(2)

```
function createPairs(arr) {

   for (let i = 0; i < arr.length; i++) {
       for(let j = i+1; j < arr.length; j++) {
           console.log(arr[i] + ", " +  arr[j] );
       }

   }
}
```

O(arr.length^2)

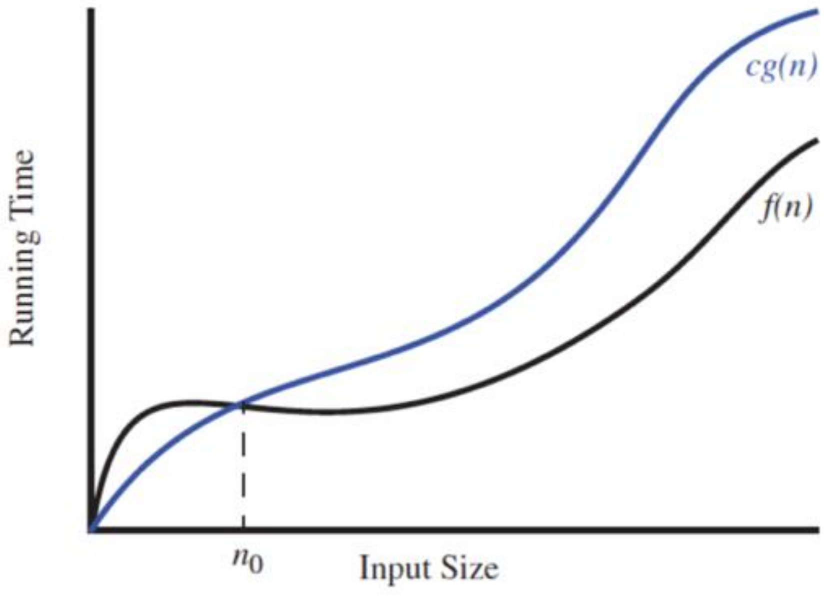
```
   public static int binarySearch(int[] a, int key)
   {
      int lo = 0, hi = a.length-1;
      while (lo <= hi)
      {
          int mid = lo + (hi - lo) / 2;
          if      (key < a[mid]) hi = mid - 1;
          else if (key > a[mid]) lo = mid + 1;
          else return mid;
     }
     return -1;
 }
```

O(a.length)

```
sum = 0;
for(inti=0;i<n;++i){
for(int j = 0; j < n*n; ++j) {
sum++;
}
}
```

O(n^3)

Big O Notation



Formal proof
Given two functions f(n) and g(n), we say that f (n) is O(g(n)) if and only if there are positive constants c and n0 such that $f(n) \leq cg(n)\ \forall\ n \geq n0$
****Note Constants C and n0 need to be independent (i.e. not dependent on N)**

Example
f(n) = 11n + 5 <= c*g(n)

Let's try C = 12

11n + 5 <= 12n

f(n) = 11n + 5 <= c*g(n)

11n + 5 <= 12n

5 <= 12n - 11n

5<= n

f(n) = 11n + 5 <= c*g(n)

11n + 5 <= 12n

5 <= 12n - 11n

5<= n

$f(n) \leq 12g(n)\ \forall\ n \geq 5$

We chose C = 12 but we could choose any real number >= 12 and any integer greater than or equal to 5 works for N

<u>**Try this one yourself**</u>

Show that 8*n* + 5 is *O*(*n*)

f(n) = 8*n* + 5

$f(n) \leq cg(n)\ \forall\ n \geq n0$

f(n) = 8n + 5 <= c*g(n)

Let's try C = 9

f(n) 8n + 5 <= c*g(n)

8n + 5 <= 9n

5 <= 9n – 8n

5 <= n

$f(n) \leq 9g(n)\ \forall\ n \geq 5$

Comparing two algorithms from different growth classes

We have two algorithms (ThreeSumA and ThreeSumB) that count the number of triples in a file of *N* integers that sums to 0 (ignoring integer overflow).

1. Estimate the Big O for each of these algorithms by looking at their code
   ThreeSumA: O(n^3)
   ThreeSumB: O(n^2)
2. Add simple java code to time the running time of both algorithms
3. Use the input files provided to run timing tests on both algorithms for each file, noting the results
4. Graph the results

*Note to run each file you need to first compile the java files to class files (use javac from the command line or your IDE) and then run each class with the different input files to time their performance. For example:

```
java ThreeSumA 1Kints.txt
java ThreeSumB 1Kints.txt
...and so on...and on.
```

Practical 2 ThreeSumFile

Files supplied:

- ThreeSumA.java
- ThreeSumB.java
- Input files of integers: 8ints, 1Kints, 4Kints, 8Kints, 16Kints, 32Kints

1. Use same code as last week for this:

import java.lang.*
This method returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC(coordinated universal time)
Final long elapsedTime = System.currentTimeMillis()

Sample code you can use to time how your algorithm performs with different size input:

```
public static void main(String[] args)
{
final long startTime = System.currentTimeMillis();
myFunction(N):
final long elapsedTime = System.currentTimeMillis() - startTime;
System.out.println("the time taken " + elapsedTime);
}
```

Converting to more readable time using something like below:

```
  // long minutes = (milliseconds / 1000) / 60;
        long minutes = TimeUnit.MILLISECONDS.toMinutes(milliseconds);
```

```java
        // long seconds = (milliseconds / 1000);
        long seconds = TimeUnit.MILLISECONDS.toSeconds(milliseconds);


```

2. We expect to see something like the outputs below for both programs

threeSumA	threeSumB
** java ThreeSumA 1Kints.txt * 70 * * java ThreeSumA 2Kints.txt * 528 * * java ThreeSumA 4Kints.txt * 4039 * * java ThreeSumA 8Kints.txt * 127867 * * * * * **	** java ThreeSumB 1Kints.txt * 70 * * java ThreeSumB 2Kints.txt * 528 * * java ThreeSumB 4Kints.txt * 4039 * * java ThreeSumB 8Kints.txt * 32074 * * java ThreeSumB 16Kints.txt * 255181 * * java ThreeSumB 32Kints.txt * 2052358*