# Big-O Lab

Recall that big-O measures the rate of growth of an algorithm's run time, relative to input size. In other words, as the size of input increases, how much more work does your code do? At this stage, these are the most typical cases:

1. $O(1)$ constant (runtime isn't affected by input size at all). Note that constant time doesn't have to be 1, just fixed. Any constant is $O(1)$, even very large constants. Therefore all basic statements that don't include loops or function calls are $O(1)$, and any number of basic statements lumped together is still $O(1)$.
2. $O(n)$ linear (runtime increases linearly with respected to the input size, or every additional input element increases the run time by a constant/fixed amount of work). This is your typical `for`/`while` loops, where the loop control is dependent on some variable $n$ and the loop body includes basic statements only.
3. $O(n^2)$ quadratic (runtime increases quadratically with respect to the input size). This is your typical doubly nested loops, where both the inner and outer loop controls are dependent on the same variable $n$ and the loop body includes basic statements only.

It is important to note that $n$ is just a variable name, and it can of course take on other names, depending on the actual code and the variable names used.

1) Write the big-O of the of the following code segments:

1.
```
for (int i=0; i<n; i++) {
  println(i);
}
```

2.
```
int i=0;
while (i<m) {
  println(i);
  i++;
}
```

3.
```
for (int i=0; i<n; i++) {
  for (int j=0; j<n; j++) {
    println(i+j);
  }
}
```

4.
```
for (int i=0; i<n; i++) {
  for (int j=m; j>0; j--) {
    println(i+j);
  }
}
```

5.
```
for (int i=0; i<n; i++) {
  for (int j=0; j<i; j++) {
    println(i+j);
  }
}
```

6.
```
for (int i=n; i>0; i/=2) {
  for (int j=0; j<n; j++) {
    println(i+j);
  }
}
```

7.
```
int i = 0;
while (i <= n) {
  for (int j=0; j<n; j++) {
    println(i+j);
  }
  i += 2;
}
```

2) Write the big-O of the following segments, assuming that the array `arr` is of length $n$.

1.
```
float sum = 0;
for (int i=0; i<arr.length; i++) {
  sum += arr[i];
}
```

2.
```
for (int i = 0; i < arr.length; i++) {
  for (int j = 0; j < arr.length; j++) {
    println(arr[i] + arr[j]);
  }
}
```

3.
```
boolean found = false;
for (int i = 0; i < arr.length; i++) {
  if (arr[i] == target) {
  found = true;
  break;
}
```

4.
```
println(arr[arr.length-1]);
```

3) Consider the following three functions. Suppose we pass the same $n \times m$ array `nums` to all three functions (assume $n \geq 4$ and $m \geq 4$). For `f2` and `f3`, how many times does the loop run? How much work is each function doing, in big-O terms?

```
void f1(int[][] arr) {

  arr[1][2] = 5;
  arr[3][3] = 10;

}
```

```
void f2(int[][] arr) {

  for (int j = 0; j < arr[0].length; j++) {
    arr[0][j] = 0;
  }

}
```

```
void f3(int[][] arr) {

  for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr[0].length; j++) {
      arr[i][j] = i + j;
    }
  }

}
```