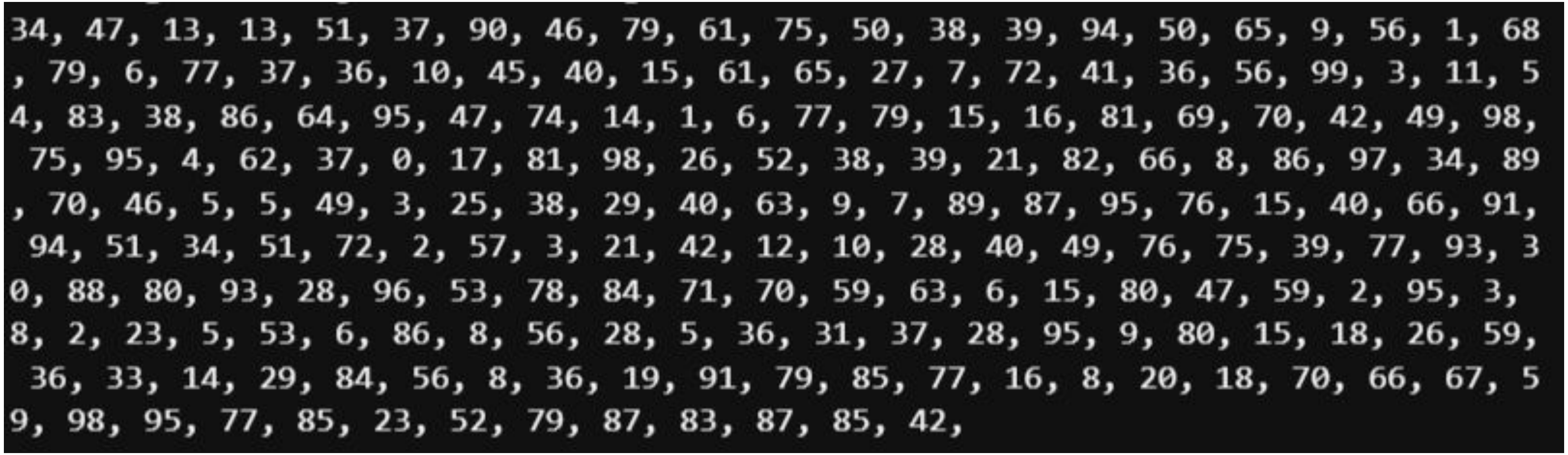
Sorting

Mr. Poole
Java

Observe this set of numbers. What can we tell?



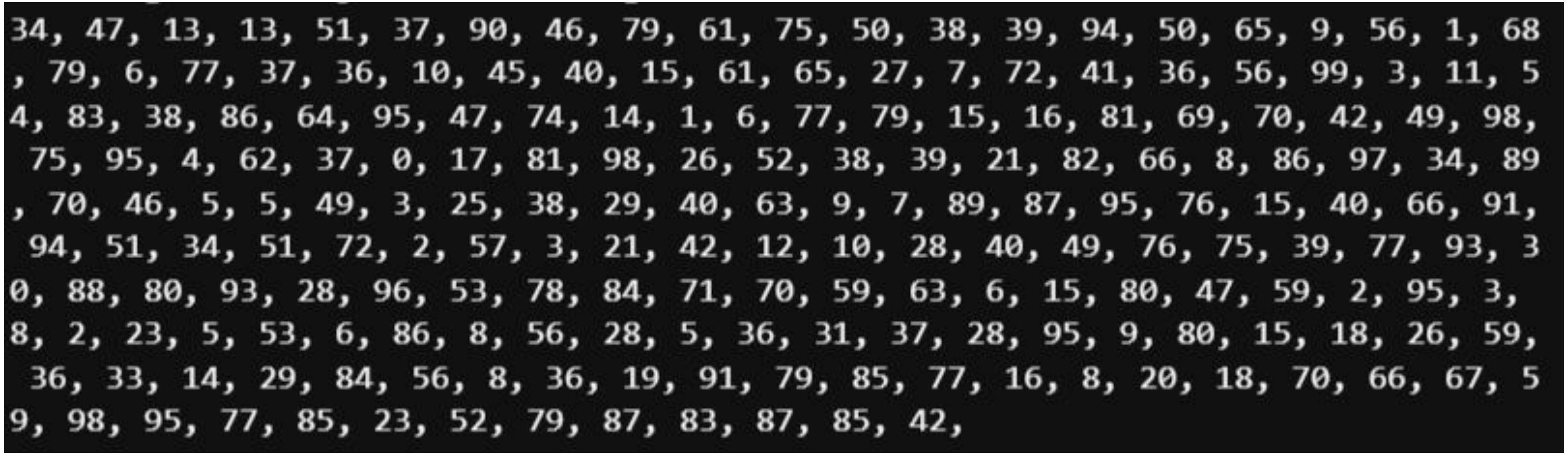
```
34, 47, 13, 13, 51, 37, 90, 46, 79, 61, 75, 50, 38, 39, 94, 50, 65, 9, 56, 1, 68
, 79, 6, 77, 37, 36, 10, 45, 40, 15, 61, 65, 27, 7, 72, 41, 36, 56, 99, 3, 11, 5
4, 83, 38, 86, 64, 95, 47, 74, 14, 1, 6, 77, 79, 15, 16, 81, 69, 70, 42, 49, 98,
 75, 95, 4, 62, 37, 0, 17, 81, 98, 26, 52, 38, 39, 21, 82, 66, 8, 86, 97, 34, 89
, 70, 46, 5, 5, 49, 3, 25, 38, 29, 40, 63, 9, 7, 89, 87, 95, 76, 15, 40, 66, 91,
 94, 51, 34, 51, 72, 2, 57, 3, 21, 42, 12, 10, 28, 40, 49, 76, 75, 39, 77, 93, 3
0, 88, 80, 93, 28, 96, 53, 78, 84, 71, 70, 59, 63, 6, 15, 80, 47, 59, 2, 95, 3,
8, 2, 23, 5, 53, 6, 86, 8, 56, 28, 5, 36, 31, 37, 28, 95, 9, 80, 15, 18, 26, 59,
 36, 33, 14, 29, 84, 56, 8, 36, 19, 91, 79, 85, 77, 16, 8, 20, 18, 70, 66, 67, 5
9, 98, 95, 77, 85, 23, 52, 79, 87, 83, 87, 85, 42,
```

Nothing!

It’s hard to read!

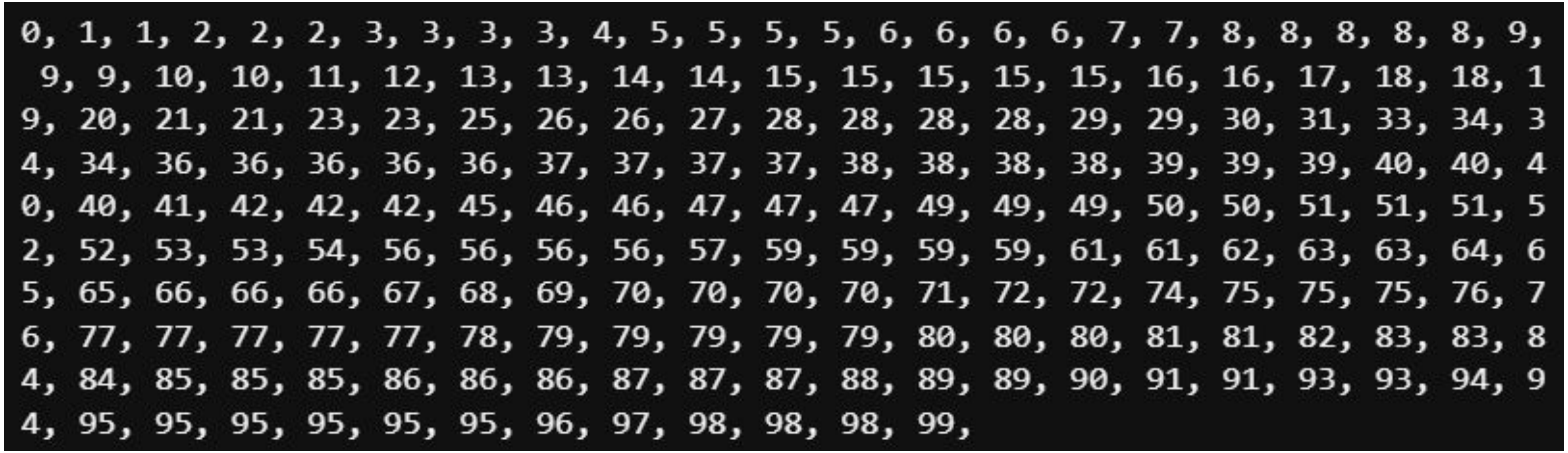
Searching - Look for how many 57’s there are.

Must search through all numbers to check if correct.



```
34, 47, 13, 13, 51, 37, 90, 46, 79, 61, 75, 50, 38, 39, 94, 50, 65, 9, 56, 1, 68
, 79, 6, 77, 37, 36, 10, 45, 40, 15, 61, 65, 27, 7, 72, 41, 36, 56, 99, 3, 11, 5
4, 83, 38, 86, 64, 95, 47, 74, 14, 1, 6, 77, 79, 15, 16, 81, 69, 70, 42, 49, 98,
 75, 95, 4, 62, 37, 0, 17, 81, 98, 26, 52, 38, 39, 21, 82, 66, 8, 86, 97, 34, 89
, 70, 46, 5, 5, 49, 3, 25, 38, 29, 40, 63, 9, 7, 89, 87, 95, 76, 15, 40, 66, 91,
 94, 51, 34, 51, 72, 2, 57, 3, 21, 42, 12, 10, 28, 40, 49, 76, 75, 39, 77, 93, 3
0, 88, 80, 93, 28, 96, 53, 78, 84, 71, 70, 59, 63, 6, 15, 80, 47, 59, 2, 95, 3,
8, 2, 23, 5, 53, 6, 86, 8, 56, 28, 5, 36, 31, 37, 28, 95, 9, 80, 15, 18, 26, 59,
 36, 33, 14, 29, 84, 56, 8, 36, 19, 91, 79, 85, 77, 16, 8, 20, 18, 70, 66, 67, 5
9, 98, 95, 77, 85, 23, 52, 79, 87, 83, 87, 85, 42,
```

Sorting and Searching

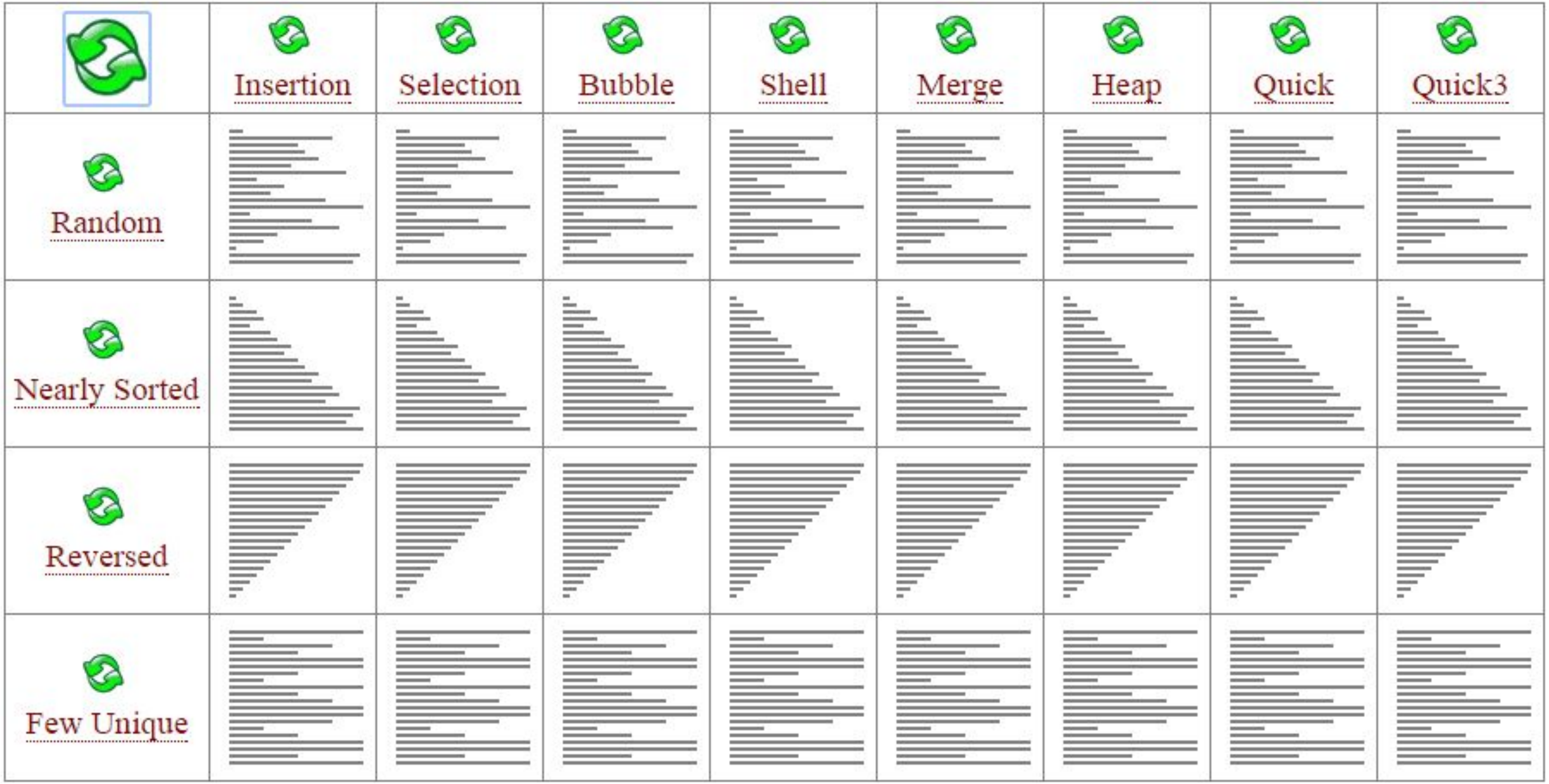


```
0, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 5, 5, 5, 5, 6, 6, 6, 6, 7, 7, 8, 8, 8, 8, 8, 9,
 9, 9, 10, 10, 11, 12, 13, 13, 14, 14, 15, 15, 15, 15, 15, 16, 16, 17, 18, 18, 1
9, 20, 21, 21, 23, 23, 25, 26, 26, 27, 28, 28, 28, 28, 29, 29, 30, 31, 33, 34, 3
4, 34, 36, 36, 36, 36, 36, 37, 37, 37, 37, 38, 38, 38, 38, 39, 39, 39, 40, 40, 4
0, 40, 41, 42, 42, 42, 45, 46, 46, 47, 47, 47, 49, 49, 49, 50, 50, 51, 51, 51, 5
2, 52, 53, 53, 54, 56, 56, 56, 56, 57, 59, 59, 59, 59, 61, 61, 62, 63, 63, 64, 6
5, 65, 66, 66, 66, 67, 68, 69, 70, 70, 70, 70, 71, 72, 72, 74, 75, 75, 75, 76, 7
6, 77, 77, 77, 77, 77, 78, 79, 79, 79, 79, 79, 80, 80, 80, 81, 81, 82, 83, 83, 8
4, 84, 85, 85, 85, 86, 86, 86, 87, 87, 87, 88, 89, 89, 90, 91, 91, 93, 93, 94, 9
4, 95, 95, 95, 95, 95, 95, 96, 97, 98, 98, 98, 99,
```

Sort FIRST!

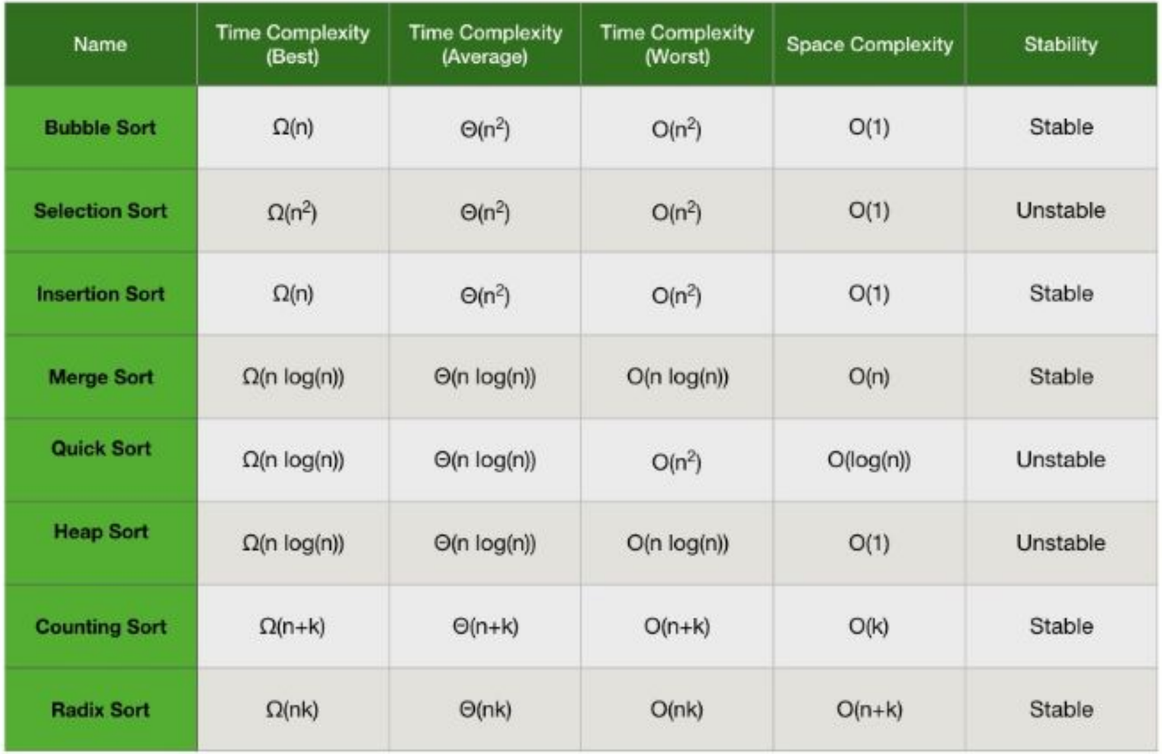
Then search, cuts down how much we have to look through

Sorting Algorithms



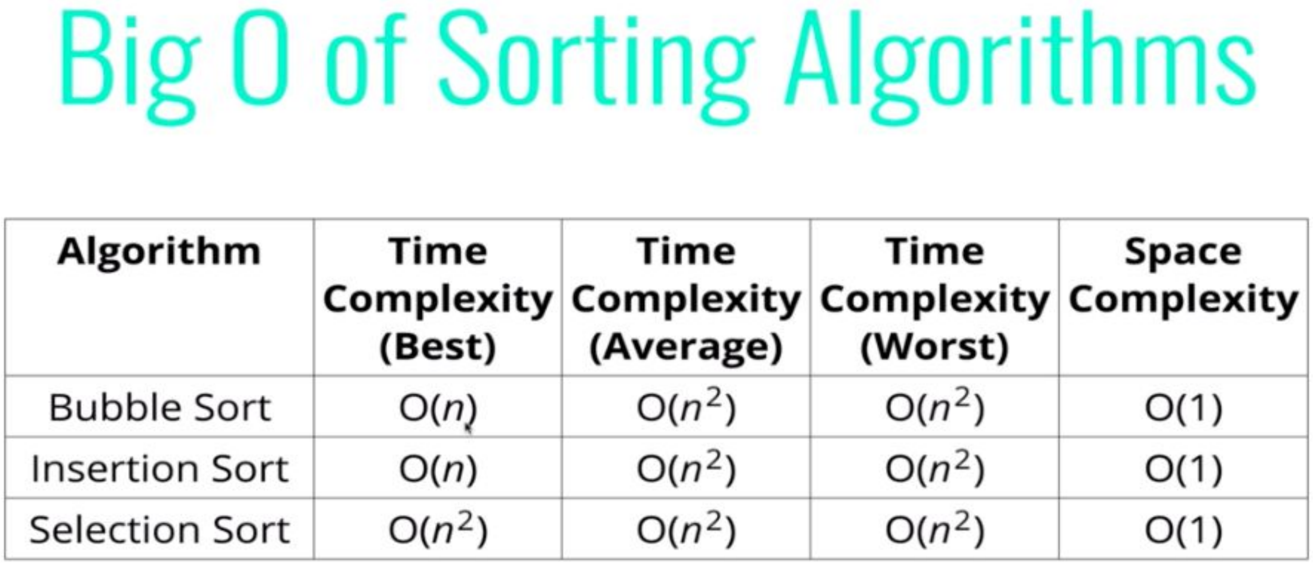
	Insertion	Selection	Bubble	Shell	Merge	Heap	Quick	Quick3
Random								
Nearly Sorted								
Reversed								
Few Unique								

Sorting Complexities



Name	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stability
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Stable
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Unstable
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Stable
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$	Stable
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$	Unstable
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$	Unstable
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$	Stable
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$	Stable

Quadratic Algorithms



Big O of Sorting Algorithms

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

Big O notation
Pigeons vs the Internet

Big O notation

Internet Transfer	**Pigeon Transfer**
1 GB -----> 30 min	1 GB -----> 60 min
2 GB -----> 60 min	2 GB -----> 60 min
3 GB -----> 90 min	3 GB -----> 60 min
1000 GB -----> 500 hours	1000 GB -----> 60 min

Linear < Constant

Big O notation - O(?)

```
main (String argos[]){
   print(a);
   print(b);
   print(c);
}
```

O(?)

```
while(x > 0){
   print(x);
   x++;
}
```

O(?)

Big O notation - O(?)

```
main (String argos[]){
    print(a);
    print(b);
    print(c);
}
```

O(1) - This happens constant times

```
while(n > 0){
    print(n);
    n++;
}
```

O(n) - This happens n times.

Big O - Rule 1: Different Steps get added

```
function test (){
    printArray(array[a]);
    printArray(array[b]);
}
```

Array of size “a” and “b” would be two different run times.

So we add them together to get **O(a+b)**

Big O - Rule 2: Drop Constants

```
function test (){
    for (array [x])
        findMinimum(array)
    for (array [x])
        findMaximum(array)
}
```

O(?)

```
function test (){
    for (array [x])
        findMinimum(array)
        findMaximum(array)
}
```

O(?)

Big O - Rule 2: Drop Constants

```
function test (){
    for (array [n])
        findMinimum(array)
    for (array [n])
        findMaximum(array)
}
```

O(2n)

```
function test (){
    for (array [n])
        findMinimum(array)
        findMaximum(array)
}
```

O(n)

We get to remove the 2 because in the process of running say
200 million times, there isn’t much of a time difference between 200 and 400

Big O - Rule 3: Different inputs get different variables

```
function test (){
   for (array [a])
      print(array)
   for (array [b])
      print(array)
}
```

O(?)

Big O - Rule 3: Different inputs get different variables

```
function test (){
    for (array [a])
        print(array)
    for (array [b])
        print(array)
}       O(a+b)
```

Since a and b are different variables, one may be vastly larger than the next. This means we must represent them differently.

Big O - Rule 4: Drop non-dominant terms

```
function test (){
    for (array [n])
        print(array)          O(?)
    for (array [n])
        for (array [n])           O(?)
            print(array)
}
```

O(?)

Big O - Rule 4: Drop non-dominant terms

```
function test (){
    for (array [n])
        print(array)
    for (array [n])
        for (array [n])
            print(array)
}
```

$O(n)$

$O(n^2)$

Overall - $O(n + n^2)$

But overall we can write the equation

$O(n^2) <= O(n^2 + n) <= O(n^2 + n^2)$

$O(n^2 + n^2)$ also equals $O(\underline{2}n^2)$ but from rule 2 we drop constants

So overall we drop n since it isn’t the dominant term. This equals $O(n^2)$