Worksheet 9: Summing Execution Times

In preparation: Read Chapter 4 to learn more about big-Oh notation.

Function	Common name	Running time
N!	Factorial	
2^n	Exponential	> century
N^d, d > 3	Polynomial	
N^3	Cubic	31.7 years
N^2	Quadratic	2.8 hours
N sqrt n		31.6 seconds
N log n		1.2 seconds
N	Linear	0.1 second
sqrt (n)	Root-n	$3.2 * 10^{-4}$ seconds
Log n	Logarithmic	$1.2 * 10^{-5}$ seconds
1	Constant	

The table at left, also found in Chapter 4, lists functions in order from most costly to least. The middle column is the common name for the function.

Suppose by careful measurement you have discovered that a program has the running time as shown at right. Describe the running time of each function using big-Oh notation.

$3n^3 + 2n + 7$	O(n ^{3})
(5 * n) * (3 + log n)	O(nlog n)
1 + 2 + 3 + … + n	O(n ^{2})
n + log n^2	O(n)
((n+1) log n) / 2	O(nlog n)
n^3 + n! + 3	O(n!)
$2^n + n^2$	O(2^n)
n (sqrt(n) + log n)	O(n sqrt(n)

Answer hint: By definition, one function that *dominates* another if as the input gets larger the dominating function will be always grow larger than the other one regardless of any constants involved. The **rule** is that when summing big-Oh values you throw away everything except the dominating function. Now, you will take help from the above table which shows the order from costly to least and find the big-Oh values following the rule. For example let's consider the last problem - n (sqrt(n) + log n) = (n sqrt(n) + n.log n) . From the above table, sqrt(n) grows larger than log n when the input size grows. So, sqrt (n) dominates log n. So the big-Oh value will be O (n sqrt (n)).

Worksheet 9: Summing Execution Times   Name:

Using the idea of dominating functions, give the big-Oh execution time for each of the following sequences of code. When elipses (…) are given you can assume that they describe only constant time operations.

<table>
<tr><td>

```
for (int i = n; i > 0; i = i / 2) {
  …
}
for (int j = 0; j * j < n; j++) …
```

</td><td>The first loop executes log n times. If we assume the statements are O(1), the total time for the for loop is log n * O(1), which is O(log n)
$j^2 < n$ or $j < sqrt(n)$, so the second loop executes sqrt(n). If we
assume the statements are O(1), the total time for the for loop is sqrt(n) * O(1), which is O(sqrt(n))
When we add up as sqrt n dominates log n overall complexity. O(sqrt n)</td></tr>
<tr><td>

```
for (int i = 0; i < n; i++) {
  for (int j = n; j > 0; j = j / 2) {
    …
  }
  for (int k = 0; k < n; k++) {
    …
  }
}
```

</td><td>The outer loop executes n times. Every time the outer loop executes, the first inner loop executes log n times. As a result, the statements in the first inner loop executes a total of n*log n times. If we assume the statements are O(1), the total time for the for first loop is n log n* O(1), which is O(n logn).

Also every time the outer loop executes the second inner loop executes n times. As a result, the statements in the second inner loop executes a total of n*n times. If we assume the statements are O(1), the total time for the for loop is n^2 * O(1), which is $O(n^2)$.

When we add up as n^2 dominates n*log n overall complexity. $O(n^2)$</td></tr>
<tr><td>

```
for (int i = 0; i < n; i++)
  …
for (int j = 0; j * j < n; j++)
  …
```

</td><td>The first loop executes n times. If we assume the statements are O(1), the total time for the for loop is n * O(1), which is O(n)
$j^2 < n$ or $j < sqrt(n)$, so the second loop executes sqrt(n) times. If we
assume the statements are O(1), the total time for the for loop is sqrt(n) * O(1),</td></tr>
</table>

	which is O(sqrt(n)) When we add up as n dominates sqrt(n) overall complexity. O(n)

for (int i = 0; i < n; i++)
…
for (int j = n; j > 0; j--)
…
	Each loop is independent and executes n times. Overall complexity , O(n)

for (int i = 1; i * i < n; i += 2)
…
for (int i = 1; i < n; i += 5)
…
	Two independent loops When summing up, n dominates sqrt(n) complexity is O(n)