Analysis of Code Segments

- Each of the following examples illustrates how to determine the Big-O run time of a segment of code or a function.
  - Each of these functions will be analyzed for their runtime in terms of the variable n.
  - Keep in mind that run-time may be dependent on more than one input variable.
This is a straight-forward function to analyze

- We only care about the simple ops in terms of n, remember any constant # of simple steps counts as 1.
- Let’s make a chart for the different values of (i,j), since for each change in i,j we do a constant amount of work.

```c
int func1(int n) {
    x = 0;
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            x++;
        }
    }
    return x;
}
```
So for each value of $i$, we do $n$ steps.

$n + n + n + \ldots + n$

$= n \times n$

$= O(n^2)$
In this situation, the first for loop runs $n$ times, so we do $n$ steps. 
After it finishes, we run the second for loop which also runs $n$ times.

Our total runtime is on the order of $n + n = 2n$.
  
  In order notation, we drop all leading constants, so our runtime is $O(n)$.
Since $n$ is changing, let $\text{origN}$ be the original value of the variable $n$ in the function.

- The 1<sup>st</sup> time through the loop, $n$ gets set to $\text{origN}/2$
- The 2<sup>nd</sup> time through the loop, $n$ gets set to $\text{origN}/4$
- The 3<sup>rd</sup> time through the loop, $n$ gets set to $\text{origN}/8$
- In general, after $k$ loops, $n$ get set to $\text{origN}/2^k$

So the algorithm ends when $\text{origN}/2^k = 1$ approximately.
int func3(int n) {
    while (n>0) {
        printf("%d", n%2);
        n = n/2;
    }
}

- So the algorithm ends when $\frac{\text{origN}}{2^k} = 1$ approximately
  - (where $k$ is the number of steps)
  - $\rightarrow \text{origN} = 2^k$
  - take log of both sides
  - $\rightarrow \log_2(\text{origN}) = \log_2(2^k)$
  - $\rightarrow \log_2(\text{origN}) = k$
  - So the runtime of this function is
  - $O(\lg n)$

**Note:**
When we use logs in run-time, we omit the base, since for all log functions with different bases greater than 1, they are all equivalent with respect to order notation.
Sidenote:
- We never use bases for logarithms in O-notation.
- This is because changing bases of logs just involves multiplying by a suitable constant.
  - and we don’t care about constant of proportionality for O-notation!

For example:
- If we have $\log_{10} n$ and we want it in terms of $\log_2 n$.
  - We know $\log_{10} n = \frac{\log_2 n}{\log_2 10}$
  - Where $1/\log_2 10 = 0.3010$
  - Then we get $\log_{10} n = 0.3010 \times \log_2 n$
In this function, i and j can increase, but they can never decrease.

- Furthermore, the code will stop when i gets to n.
- Thus, the statement i++ can never run more than n times and the statement j++ can never run more than n times.
- Thus, the most number of times these two critical statements can run is **2n**.
- It follows that the runtime of this segment of code is **O(n)**.
All we did in this example is reset \( j \) to 0 at the beginning of \( i \) loop iteration.

- Now, \( j \) can range from 0 to \( n \) for EACH value of \( i \)
- (similar to example #1),
- so the run-time is

\[ O(n^2) \]
The amount of times the inner loop runs is dependent on \( i \)

- The table shows how \( j \) changes with respect to \( i \)
- The # of times the inner loop runs is the sum:
  - \( 0 + 1 + 2 + 3 + \ldots + (n-1) \)
  - \( = (n-1)n/2 = 0.5n^2 - 0.5n \)
  - So the run time is? \( O(n^2) \)

Common Summation:
\[
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\]

What we have:
\[
\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}
\]
This runtime is in terms of \texttt{sizea} and \texttt{sizeb}.

Clearly, similar to Example #1, we simply multiply the \# of terms in the 1\textsuperscript{st} loop by the number of terms in the 2\textsuperscript{nd} loop.

Here, this is simply \texttt{sizea}*\texttt{sizeb}.

So the runtime is? \textbf{O(sizea*sizeb)}
This runtime is in terms of \textit{sizea} and \textit{sizeb}.

Clearly, similar to Example #1, we simply multiply the \# of terms in the 1\textsuperscript{st} loop by the number of terms in the 2\textsuperscript{nd} loop.

Here, this is simply \textit{sizea\times sizeb}.

So the runtime is? \textit{O(sizea\times sizeb)}
As previously discussed, a **single binary search** runs in $O(lg n)$

- where $n$ represents the number of items in the original list you’re searching.

In this particular case, the runtime is? $O(sizea*lg(sizeb))$

- since we run our binary search on $sizeb$ items exactly $sizea$ times.

```c
int f8(int a[], int sizea, int b[], int sizeb) {
    int i, j;

    for (i=0; i<sizea; i++) {
        if (binSearch(b, sizeb, a[i]))
            return 1;
    }
    return 0;
}
```
In this particular case, the runtime is \( O(sizea \times \log(sizeb)) \) since we run our binary search on \( sizeb \) items exactly \( sizea \) times.

**Notice:**

that the runtime for this algorithm changes greatly if we switch the order of the arrays. Consider the 2 following examples:

1) \( sizea = 1000000, sizeb = 10 \) \( sizea \times \log(sizeb) \approx 3320000 \)
2) \( sizea = 10, sizeb = 1000000 \) \( sizea \times \log(sizeb) \approx 300 \)