Analysis of Code Segments

Each of the following examples illustrates how to determine the Big-Oh 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.

Example #1

```
int func1(int n) {
  x = 0;
  for (i = 1; i <= n; i++) {
    for (j = 1; j <=n; j++) {
      x++;
    }
  }
  return x;
}
```

This is one of the more straight-forward functions to analyze. We only care about the number of simple operations in terms of n, and remember that any constant number of simple steps just counts as 1. Let's make a chart of the different values of the ordered pair (i,j):

i	j	
1	1	
1	2	
…		
1	n	(n steps)
2	1	
2	2	
…		
n	1	
…		
n	n	

Note that for each pair of values (i,j) we do a constant amount of work.

For each value of i, we do n steps.

n + n + n + … + n (adding n times)

n*n = O(n^2)

<u>Example #2</u>

```
int func2(int n) {
  x = 0;
  for (i = 1; i <= n; i++)
      x++;
  for (i = 1; i<=n; i++)
      x++;
  return x;
}
```

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).

Example #3

```
int func3(int n) {
  while (n>0) {
    printf("%d", n%2);
    n = n/2;
  }
}
```

For the clarity of our analysis, let origN be the original value of the variable n in the function.

The first time through the loop, n gets set to origN/2.
The second time through the loop, n gets set to origN/4.
The third time through the loop, n gets set to origN/8.
In general, after k loops, n get set to origN/2^k.

The algorithm ends when origN/2^k = 1, approximately.
This is when origN = 2^k.
By the definition of logs, we have k = $\log_2$origN.
It follows that 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.

Example #4

```
int func4(int** array, int n)  {

    int i=0, j=0;
    while (i < n) {
        while (j < n && array[i][j] == 1)
             j++;
        i++;
    }
    return j;
}
```

In this function, i and j can increase, but they can never decrease. Furthermore, the code will 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).

Example #5

```
int func5(int** array, int n)  {

    int i=0, j;
    while (i < n) {
        j=0;
        while (j < n && array[i][j] == 1)
             j++;
        i++;
    }
    return j;
}
```

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 $O(n^2)$.

Example #6

```
int func6(int array[], int n) {

  int i,j, sum=0;
  for (i=0; i<n; i++) {
    for (j=i+1; j<n; j++)
      if (array[i] > array[j])
        sum++;
  }
  return sum;
}
```

Here is a chart for how many times the inner loop runs, depending on the value of i:

i=0,	**j=1,2,3,4,..,n-1**	**(n-1 values)**
i=1,	**j=2,3,4,5,…,n-1**	**(n-2 values)**
i=2,	**j=3,4,5,…n-1**	**(n-3 values)**
…		
i=n-1,	**j=nothing**	**(0 values)**

Thus, the number of times the inner loop runs is the sum

0 + 1 + 2 + … + (n-1).

This sum is (n – 1)n/2 = .5n^2 - .5n. Using the rules of order notation, this is O(n^2), by dropping both constants and all terms except the most significant.

Example #7

```
int f7(int a[], int sizea, int b[], int sizeb) {

    int i, j;
    for (i=0; i<sizea; i++)
        for (j=0; j<sizeb; j++)
            if (a[i] == b[j])
                return 1;
    return 0;
}
```

This runtime is in terms of sizea and sizeb. Clearly, similar to Example #1, we simply multiplty the number of terms in the first loop by the number of terms in the second loop. Here, this is simply sizea*sizeb.

Example #8

```
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;
}
```

As previously discussed, a single binary search runs in O(lg n) where n represents the number of items within which you were searching.

In this particular case, the runtime is O(sizea*lg(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 two following examples:

sizea=1000000, sizeb=10
sizea=10, sizeb=1000000

Note that sizea*lg(sizeb) ~ 3320000 for the first case and
and sizea*lg(sizeb) ~ 300 for the latter case

Example #9: Tracing (uses similar skills to order analysis)

Find the value of x in terms of n after the following code segment below has executed. You may assume that n is a positive even integer.

```
x = 0;
for (i = 1; i <= n*(8*n+8); i++) {
    for (j = n/2; j <=n; j++) {
          x = x + (n – j);
    }
}
```

First notice that all we are doing is repeatedly adding numbers into x. Furthermore, since the inner loop is NOT dependant on the value of i, we are adding the same value into x for each iteration of the outer loop. Thus, we must first figure out how much is being added into to x each time the entire inner loop runs.

Iteration	value of j	value of n-j
1	n/2	n/2
2	n/2+1	n/2 - 1
3	n/2 +2	n/2 - 2
...		
n/2+1	n	0

Thus, we must add all the values in the right-hand column to figure out the value that gets added to x for a complete run of the inner loop.

$0+1+2+...+n/2 = n/2*(n/2+1)/2 = (n^2 + 2n)/8$

Now, we see that we add this value into x exactly n(8n+8) number of times. Repeated addition is multiplication, so the value of x after the loops are done will be

$(n^2 + 2n)/8*n(8n+8) = n(n+2)/8*n*8*(n+1)$

$= n^2(n+1)(n+2)$

Practice Problem

Assuming that n is a positive even integer, what will be the value of x right after this segment of code is run?

```
x = 0;
for (i = 1; i <= n/2; i++) {
    for (j = 1; j <=n; j++) {
       if (j > i)
          x++;
    }
}
```