

# More Algorithm Analysis



Computer Science Department  
University of Central Florida

*COP 3502 – Computer Science I*



# Big-O Notation

---

- What is Big O?
  - Big O comes from Big-O Notation
    - In C.S., we want to know how efficient an algorithm is...how “fast” it is
    - More specifically...we want to know **how the performance of an algorithm responds to changes in problem size**
    - The goal is to provide a *qualitative* insight on the # of operations for a problem size of  $n$  elements.
    - And this total # of operations can be described with a mathematical expression in terms of  $n$ .
      - This expression is known as Big-O



# More Algorithm Analysis

---

- Examples of Analyzing Code:
  - We now go over many examples of code fragments
  - Each of these functions will be analyzed for their runtime in terms of the variable  $n$
  - Utilizing the idea of Big-O,
    - determine the Big-O running time of each



# More Algorithm Analysis

---

- Example 1:

- Determine the Big O running time of the following code fragment:

```
for (k = 1; k <= n/2; k++) {  
    sum = sum + 5;  
}  
for (j = 1; j <= n*n; j++) {  
    delta = delta + 1;  
}
```



# More Algorithm Analysis

## ■ Example 1:

### ■ So look at what's going on in the code:

- We care about the total number of REPETITIVE operations.

- Remember, we said we care about the running time for LARGE values of  $n$
- So in a for loop with  $n$  as part of the comparison value determining when to stop 

```
for (k=1; k<=n/2; k++)
```
- Whatever is INSIDE that loop will be executed a LOT of times
- So we examine the code within this loop and see how many operations we find
  - When we say operations, we're referring to mathematical operations such as  $+$ ,  $-$ ,  $*$ ,  $/$ , etc.



# More Algorithm Analysis

## ■ Example 1:

- So look at what's going on in the code:
  - The number of operations executed by these loops is the sum of the individual loop operations.
  - We have 2 loops,
    - The first loop runs  $n/2$  times
    - Each iteration of the first loop results in one operation
      - The + operation in: `sum = sum + 5;`
    - So there are  $n/2$  operations in the first loop
    - The second loop runs  $n^2$  times
    - Each iteration of the second loop results in one operation
      - The + operation in: `delta = delta + 1;`
    - So there are  $n^2$  operations in the second loop.



# More Algorithm Analysis

- Example 1:
  - So look at what's going on in the code:
    - The number of operations executed by these loops is the sum of the individual loop operations.
    - The first loop has  $n/2$  operations
    - The second loop has  $n^2$  operations
    - They are NOT nested loops.
      - One loop executes AFTER the other completely finishes
    - So we simply ADD their operations
    - The total number of operations would be  $n/2 + n^2$
    - In Big-O terms, we can express the number of operations as  $O(n^2)$



# More Algorithm Analysis

- Example 2:
  - Determine the Big O running time of the following code fragment:

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





# More Algorithm Analysis

---

- Example 2:
  - So look at what's going on in the code:
    - We care about the total number of REPETITIVE operations
    - We have two loops
      - AND they are NESTED loops
    - The outer loop runs  $n$  times
      - From  $i = 1$  up through  $n$
      - How many operations are performed at each iteration?
        - Answer is coming...
    - The inner loop runs  $n$  times
      - From  $j = 1$  up through  $n$
      - And only one operation ( $x++$ ) is performed at each iteration



# More Algorithm Analysis

- Example 2:
  - So look at what's going on in the code:
    - Let's look at a couple of iterations of the OUTER loop:
      - When  $i = 1$ , what happens?
        - The inner loop runs  $n$  times
        - Resulting in  $n$  operations from the inner loop
      - Then,  $i$  gets incremented and it becomes equal to 2
      - When  $i = 2$ , what happens?
        - Again, the inner loop runs  $n$  times
        - Again resulting in  $n$  operations from the inner loop
    - We notice the following:
      - For EACH iteration of the OUTER loop,
      - The INNER loop runs  $n$  times
        - Resulting in  $n$  operations



# More Algorithm Analysis

---

- Example 2:
  - So look at what's going on in the code:
    - And how many times does the outer loop run?
      - n times
    - So the outer loop runs n times
    - And for each of those n times, the inner loop also runs n times
      - Resulting in n operations
    - So we have n operations per iteration of OUTER loop
    - And outer loop runs n times
    - Finally, we have  $n \cdot n$  as the number of operations
    - We approximate the running time as  $O(n^2)$



# More Algorithm Analysis

---

- Example 3:
  - Determine the Big O running time of the following code fragment:

```
int func3(int n) {
    int i, x = 0;
    for (i = 1; i <= n; i++)
        x++;
    for (i = 1; i<=n; i++)
        x++;
    return x;
}
```



# More Algorithm Analysis

---

- Example 3:
  - So look at what's going on in the code:
    - We care about the total number of REPETITIVE operations
    - We have two loops
      - They are NOT nested loops
    - The first loop runs  $n$  times
      - From  $i = 1$  up through  $n$
      - only one operation ( $x++$ ) is performed at each iteration
    - How many times does the second loop run?
      - Notice that  $i$  is indeed reset to 1 at the beginning of the loop
      - Thus, the second loop runs  $n$  times, from  $i = 1$  up through  $n$
      - And only one operation ( $x++$ ) is performed at each iteration



# More Algorithm Analysis

---

- Example 3:
  - So look at what's going on in the code:
    - Again, the loops are NOT nested
    - So they execute sequentially (one after the other)
  - Therefore:
    - Our total runtime is on the order of  $n+n$
    - Which of course equals  $2n$
  
- Now, in Big O notation
  - We approximate the running time as  $O(n)$



# More Algorithm Analysis

---

- Example 4:
  - Determine the Big O running time of the following code fragment:

```
int func4(int n) {  
    while (n > 0) {  
        printf("%d", n%2);  
        n = n/2;  
    }  
}
```



# More Algorithm Analysis

- Example 4:
  - So look at what's going on in the code:
    - We have one while loop
      - You can't just look at this loop and say it iterates  $n$  times or  $n/2$  times
      - Rather, it continues to execute as long as  $n$  is greater than 0
      - The question is: **how many iterations will that be?**
    - Within the while loop
      - The last line of code divides the input,  $n$ , by 2
      - So  $n$  is halved at each iteration of the while loop
    - If you remember, we said this ends up running in  $\log n$  time
    - Now let's look at how this works





# More Algorithm Analysis

- Example 4:
  - So look at what's going on in the code:
    - For the ease of the analysis, we make a new variable
      - originalN:
        - originalN refers to the value originally stored in the input, n
        - So if n started at 100, originalN will be equal to 100
    - The first time through the loop
      - n gets set to  $\text{originalN}/2$ 
        - If the original n was 100, after one iteration n would be  $100/2$
    - The second time through the loop
      - n gets set to  $\text{originalN}/4$
    - The third time through the loop
      - n gets set to  $\text{originalN}/8$

**Notice:**

After **three** iterations, n gets set to  $\text{originalN}/2^3$



# More Algorithm Analysis

- Example 4:
  - So look at what's going on in the code:
    - In general, after  $k$  iterations
      - $n$  gets set to  $\text{originalN}/2^k$
    - The algorithm ends when  $\text{originalN}/2^k = 1$ , approximately
    - We now solve for  $k$
    - Why?
      - Because we want to find the **total # of iterations**
    - Multiplying both sides by  $2^k$ , we get  $\text{originalN} = 2^k$
    - Now, using the definition of logs, we solve for  $k$ 
      - $k = \log \text{originalN}$
    - So we approximate the running time as  $O(\log n)$



# Brief Interlude: Human Stupidity

---





# More Algorithm Analysis

- Example 5:
  - Determine the Big O running time of the following code fragment:

```
int func5(int** array, int n) {
    int i = 0, j = 0;
    while (i < n) {
        while (j < n && array[i][j] == 1)
            j++;
        i++;
    }
    return j;
}
```



# More Algorithm Analysis

- Example 5:
  - So look at what's going on in the code:
    - At first glance, we see two NESTED loops
    - This can often indicate an  $O(n^2)$  algorithm
      - But we need to look closer to confirm
    - Focus on what's going on with  $i$  and  $j$

```
int func5(int** array, int n) {
    int i = 0, j = 0;
    while (i < n) {
        while (j < n && array[i][j] == 1)
            j++;
        i++;
    }
}
```



# More Algorithm Analysis

- Example 5:
  - So look at what's going on in the code:
    - Focus on what's going on with i and j
      - i and j clearly increase (from the j++ and i++)
      - BUT, they never decrease
      - AND, neither ever gets reset to 0

```
int func5(int** array, int n) {
    int i = 0, j = 0;
    while (i < n) {
        while (j < n && array[i][j] == 1)
            j++;
        i++;
    }
}
```



# More Algorithm Analysis

## ■ Example 5:

- So look at what's going on in the code:
  - And the OUTER while loop ends once i gets to n
  - So, what does this mean?
    - The statement i++ can never run more than n times
    - And the statement j++ can never run more than n times

```
int func5(int** array, int n) {
    int i = 0, j = 0;
    while (i < n) {
        while (j < n && array[i][j] == 1)
            j++;
        i++;
    }
}
```



# More Algorithm Analysis

## ■ Example 5:

- So look at what's going on in the code:
  - The MOST number of times these two statements can run (combined) is  $2n$  times
  - So we approximate the running time as  $O(n)$

```
int func5(int** array, int n) {
    int i = 0, j = 0;
    while (i < n) {
        while (j < n && array[i][j] == 1)
            j++;
        i++;
    }
}
```





# More Algorithm Analysis

## ■ Example 6:

- Determine the Big O running time of the following code fragment:
  - What's the one big difference here???

```
int func6(int** array, int n) {
    int i = 0, j;
    while (i < n) {
        j = 0;
        while (j < n && array[i][j] == 1)
            j++;
        i++;
    }
    return j;
}
```



# More Algorithm Analysis

## ■ Example 6:

- So look at what's going on in the code:
  - The difference is that we RESET `j` to 0 at the beginning of the OUTER while loop

```
int func6(int** array, int n) {
    int i = 0, j;
    while (i < n) {
        j = 0;
        while (j < n && array[i][j] == 1)
            j++;
        i++;
    }
    return j;
}
```



# More Algorithm Analysis

---

- Example 6:
  - So look at what's going on in the code:
    - The difference is that we RESET  $j$  to 0 at the beginning of the OUTER while loop
    - How does that change things?
      - Now  $j$  can iterate from 0 to  $n$  for EACH iteration of the OUTER while loop
        - For each value of  $i$
        - This is similar to the 2<sup>nd</sup> example shown
    - So we approximate the running time as  $O(n^2)$



# More Algorithm Analysis

## ■ Example 7:

- Determine the Big O running time of the following code fragment:

```
int func7(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;
}
```



# More Algorithm Analysis

---

- Example 7:
  - So look at what's going on in the code:
    - First notice that the runtime here is NOT in terms of  $n$
    - It will be in terms of  $\text{sizeA}$  and  $\text{sizeB}$
    - And this is also just like Example 2
    - The outer loop runs  $\text{sizeA}$  times
    - For EACH of those times,
      - The inner loop runs  $\text{sizeB}$  times
    - So this algorithm runs  $\text{sizeA} * \text{sizeB}$  times
    - We approximate the running time as  $O(\text{sizeA} * \text{sizeB})$



# More Algorithm Analysis

- Example 8:
  - Determine the Big O running time of the following code fragment:

```
int func8(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;
}
```



# More Algorithm Analysis

---

## ■ Example 8:

### ■ So look at what's going on in the code:

- Note: we see that we are calling the function `binSearch`
- As discussed previously, a single binary search runs in  $O(\log n)$  time
  - where  $n$  represents the number of items within which you are searching

### ■ Examining the for loop:

- The for loop will execute `sizeA` times
- For EACH iteration of this loop
  - a binary search will be run
- We approximate the running time as  $O(\text{sizeA} * \log(\text{sizeB}))$



# More Algorithm Analysis

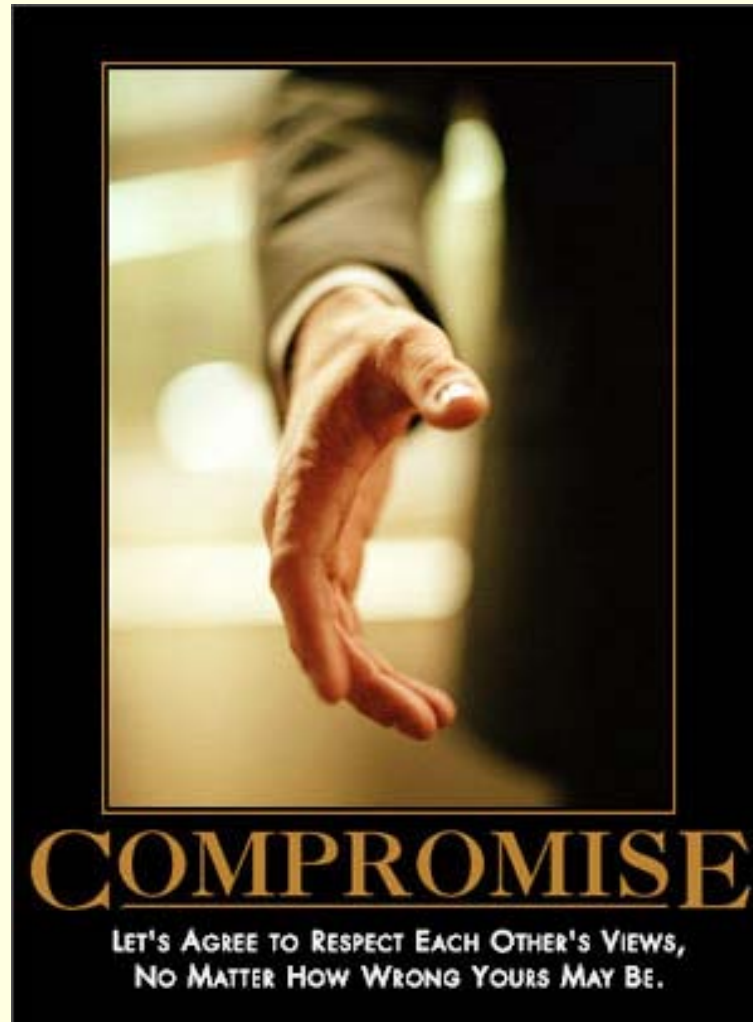
---

**WASN'T  
THAT  
SWEET!**





# Daily Demotivator



# More Algorithm Analysis



Computer Science Department  
University of Central Florida

*COP 3502 – Computer Science I*