UCF

# INTRO TO ALGORITHM ANALYSIS

COP 3502

# 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.

# Analysis of Code Segments: EX 1

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

1 operation

| $i$ | $j$ |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 1 | 3 |
| … | … |
| 1 | n |
| 2 | 1 |
| 2 | 2 |
| 2 | 3 |
| … | … |
| 2 | n |
| … | … |
| n | 1 |
| … | … |
| n | n |

- 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.

# Analysis of Code Segments: EX 1

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

1 operation

| $i$ | $j$ |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 1 | 3 |
| … | … |
| 1 | n |
| 2 | 1 |
| 2 | 2 |
| 2 | 3 |
| … | … |
| 2 | n |
| … | … |
| n | 1 |
| … | … |
| n | n |

- So for each value of ***i***, we do ***n*** steps.
- $n + n + n + \ldots + n$
- $= n * n$
- $= O(n^2)$

# Analysis of Code Segments: EX 2

```
int func2(int n) {
  x = 0;
  for (i = 1; i <= n; i++)
     x++;                1 operation
  for (i = 1; i<=n; i++)
     x++;                1 operation
  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*** = 2 ***n***.
  - In order notation, we drop all leading constants, so our runtime is
  - **O(*n* )**

# Analysis of Code Segments: EX 3

```
int func3(int n) {
  while (n>0) {
    printf("%d", n%2);     ← 1 operation
    n = n/2;               ← 1 operation
  }
}
```

- Since n is changing, let ***origN*** be the original value of the variable ***n*** in the function.
  - The **1st** time through the loop, ***n*** gets set to **origN/2**
  - The **2nd** time through the loop, ***n*** gets set to **origN/4**
  - The **3rd** time through the loop, ***n*** gets set to **origN/8**
  - In general, **after k loops**, ***n*** get set to **origN/$2^k$**
- So the algorithm ends when **origN/$2^k$ = 1** approximately

# Analysis of Code Segments: EX 3

```
int func3(int n) {
   while (n>0) {
      printf("%d", n%2);      ← 1 operation
      n = n/2;                ← 1 operation
   }
}
```

- So the algorithm ends when **origN/$2^k$ = 1** approximately
  - (where ***k*** is the number of steps)
  - → origN = $2^k$
  - take log of both sides
  - → $\log_2(\text{origN}) = \log_2(2^k)$
  - → $\log_2(\text{origN}) = k$
  - 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**

# Logarithms

- **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 $\log_{10}n = \log_2 n/\log_2 10$
    - Where $1/\log_2 10 = 0.3010$
    - Then we get $\log_{10}n = 0.3010 \times \log_2 n$

# Analysis of Code Segments: EX 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 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)**

# Analysis of Code Segments: EX 5

```
int func5(int** array, int n) {
    int i=0, j=0;
    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
  - $O(n^2)$

# Analysis of Code Segments: EX 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;
}
```

| i | j | value |
|---|---|---|
| 0 | 1,2,3,…,n-1 | n-1 |
| 1 | 2,3,4,…,n-1 | n-2 |
| 2 | 3,4,5,…,n-1 | n-3 |
| … | | |
| n-1 | nothing | 0 |

- The amount of times the inner loop runs is dependent on ***i***
  - The table shows how ***j*** changes w/respect to ***i***
  - The # of times the inner loop runs is the sum:
  - **0 + 1 + 2 + 3 + … + (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:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

# Analysis of Code Segments: EX 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 multiply the # of terms in the 1st loop by the number of terms in the 2nd loop.
- Here, this is simply ***sizea*sizeb***.
- So the runtime is? O(sizea*sizeb)

# Analysis of Code Segments: EX 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 multiply the # of terms in the 1st loop by the number of terms in the 2nd loop.
- Here, this is simply ***sizea*sizeb***.
- So the runtime is? O(sizea*sizeb)

# Analysis of Code Segments: EX 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 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.

# Analysis of Code Segments: EX 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;
}
```

- 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 2 following examples:

1) ***sizea*** = 1000000, ***sizeb*** = 10 sizea*lg(sizeb) ~ 3320000
2) ***sizea*** = 10, ***sizeb*** = 1000000 sizea*lg(sizeb) ~ 300