Introduction

The most difficult part of our discussion of time complexity is the computation of $T(n)$, the timing function for a code fragment. Once the timing function is known, it is quite easy to characterize the growth of the algorithm in some simpler and more recognizable function. In this lecture, we will explore several examples to illustrate our concept of time complexity applied to computer algorithms.

Example 1 Find the time complexity of the code fragment in Figure 1.

```
void max(int a, int b)
{   if (a>b) {
        return a;
    } else {
        return b;
    }
}
```

Figure 1. Program for Example 1.

Solution: Complete the following steps.

1. Create a three column table putting line numbers in the first column, statements of the program in the second column, and the cost of each statement in the third column.
2. Sum the cost of the individual statements to find the cost of the program.

Line	Statement	Cost
1	**void** *max*(**int** *a*, **int** *b*)	0
2	{ **if** (*a*>*b*) {	1
3	**return** *a*;	0
4	} **else** {	0
5	**return** *b*;	0
6	}	0
7	}	0
	Total	1

Explanation:

1. The statement of line 1 was assigned a cost of zero (0) because the function header is translated to the subprogram prolog and by Time Complexity Rule 1 subprogram prologs are said to cost nothing.
2. The statement on line 2 was assigned a cost of one (1) because of the relational operation.
3. The return-statement on line 3 is translated to a subprogram epilog and, therefore, is assigned a cost of zero (0).
4. There are no operations on line 4 resulting in a cost of zero (0).
5. The return-statement one line 5 is translated to a subprogram epilog and, therefore, is assigned a cost of zero (0).
6. There are no operations on line 6 resulting in a cost of zero (0).
7. The closing curly brace on line 7 is translated to the subprogram epilog that has no cost.

The total is one (1).

Example 2

Find the time complexity of function *findmax* in Figure 2 that finds the maximum value in the array of integers given by the two parameters *A* and *n*.

```
int findmax(int A[], int n)
{   int max=A[0];
    for (int i=1;i<n;i++) {
        if (A[i]>max) max=A[i];
    }
    return max;
}
```

Figure 2. Program for Example 2.

Solution: Complete the following steps.

1. Create a three column table putting line numbers in the first column, statements of the program in the second column, and the cost of each statement in the third column.
2. Sum the cost of the individual statements to find the cost of the program.

Line	Statement	Cost
1	**int** *findmax*(**int** *A*[], **int** *n*)	0
2	{ **int** *max*=*A*[0];	1
3	**int** *i*=1;	1
4	**while** (*i*<*n*) {	$k = \sum_{i=1}^{n-1} 1 = n - 1$
5	**if** (*A*[*i*]>*max*) *max*=*A*[*i*];	$2k$
6	*i*++;	k
7	}	1
8	**return** *max*;	0
9	}	0
	Total	$T(n) = 4k + 3$
	Total	$T(n) = 4(n-1) + 3$
	Total	$T(n) = 4n - 1$

Explanation:

1. The statement of line 1 was assigned a cost of zero (0) because the function header is translated to the subprogram prolog and by Time Complexity Rule 1 subprogram prologs are said to cost nothing.
2. Only the assignment operation is counted on line 2. The index operation **[]** is assigned a cost of zero (0).
3. The assignment operation is assigned a cost of one (1) on line 3.

   We transformed the original statement on line 3 in Figure 1 to three statement is the table above. The for-statement was altered to the initialization on line 3, the while-test on line 4, and the increment on line 6.
4. The test on line 4 is true $n - 1$ times and false 1 time. We account for the cost of the times the test is executed when it is true on line 4 and the one time when it is false on line 7.

Explanation continued.

5. There are two operations on line 5, the relational operation and the assignment. The relational operation is executed every time the loop body is executed, $n - 1$ times. The assignment is only executed when the relational operation is true. However, for the purpose of computing time complexity, we assume that the assignment statement is executed every time the loop body is executed, $n - 1$ times.
6. The loop variable *i* is incremented $n - 1$ times on line 6.
7. We account for the cost of the one time when the loop test on line 4 is false on line 7.
8. The return-statement on line 8 is translated to a subprogram epilog and, therefore, is assigned a cost of zero (0).
9. The closing curly brace on line 9 is translated to the subprogram epilog that has no cost.

The total is $T(n) = 4n - 1$. This expression can be interpreted to mean that roughly four operations are required to find the maximum value for every element in the array.

Example 3

Find the time complexity of function *findsum* in Figure 3 that exercises a nested loop.

```
int findsum(int n)
{   int sum=0;
    for (int a=n;a>0;a--) {
        for (int b=n;b>a;b--) {
            sum++;
        }
    }
    return sum;
}
```

Figure 3. Program for Example 3.

Solution: Complete the following steps.

1. Simplify for-statements
2. Create a three column table putting line numbers in the first column, statements of the program in the second column, and the cost of each statement in the third column.
3. Sum the cost of the individual statements to find the cost of the program.

Line	Statement	Cost
1	**int** *findsum*(**int** *n*)	0
2	{ **int** *sum*=**0;**	1
3	**int** *a*=*n*;	1
4	**while** (*a*>**0**) {	$j = \sum_{a=1}^{n} 1 = n$
5	**int** *b*=*n*;	j
6	**while** (*b*>*a*) {	$k = \sum_{a=1}^{n} \sum_{b=a+1}^{n} 1$
7	*sum*++;	k
8	*b*--;	k
9	}//**end b**	j
10	*a*--;	j
11	}//**end a**	1
12	**return** *sum*;	0
13	}	
Total		$T(n) = 3k + 4j + 3$
k		$k = \sum_{a=1}^{n} \sum_{b=a+1}^{n} 1$
k		$k = \sum_{a=1}^{n} (n - a) = \sum_{a=1}^{n} n - \sum_{a=1}^{n} a$
k		$k = \sum_{a=1}^{n} n - \sum_{a=1}^{n} a = n^2 - \frac{n(n+1)}{2} = n^2 - \frac{1}{2}n^2 - \frac{1}{2}n = \frac{1}{2}n^2 - \frac{1}{2}n$
Total		$\boldsymbol{T(n) = 3\left[\frac{1}{2}n^2 - \frac{1}{2}n\right] + 4[n] + 3}$
Total		$\boldsymbol{T(n) = \frac{3}{2}n^2 + \frac{5}{2}n + 3}$

Explanation:

1. The statement of line 1 was assigned a cost of zero (0) because the function header is translated to the subprogram prolog and by Time Complexity Rule 1 subprogram prologs are said to cost nothing.
2. Only the assignment operation is counted on line 2. The index operation **[]** is assigned a cost of zero (0).
3. The assignment operation is assigned a cost of one (1) on line 3.

   We transformed the original statement on line 3 in Figure 1 to three statement is the table above. The for-statement was altered to the initialization on line 3, the while-test on line 4, and the increment on line 6.
4. The test on line 4 is true $n-1$ times and false 1 time. We account for the cost of the times the test is executed when it is true on line 4 and the one time when it is false on line 7.

Example 4

Find the time complexity of function *CountIterations* in Figure 4 that finds the number of iterations executed by the loop.

```
void CountIterations(int n)
{    while (n>1) n=n/2;
}
```

Figure 4. Program for Example 4.

Solution: Complete the following steps.

1. Simplify for-statements
2. Create a three column table putting line numbers in the first column, statements of the program in the second column, and the cost of each statement in the third column.
3. Sum the cost of the individual statements to find the cost of the program.

Line	Statement	Cost
1	**void** *CountInterations*(**int** *n*)	0
2	**{ while(***n***>1) {**	k
3	*n*=*n*/**2;**	$2k$
4	**}**	1
5	**}**	0
Total	$T(n) = 3k + 2$	

k $$n_{i+1} = \frac{n_i}{2}, 0 < n_k \leq 1$$

k $n_0 = n_p$ **where** p **signifies parameter. The initial value of** *n*, n_0 **is the value of** *n* **passed as the parameter of the function.**

k $$n_2 = \frac{n_1}{2} = \frac{n_0/2}{2} = \frac{n_0}{4} = \frac{n_0}{2^2}$$

k $$n_k = \frac{n_0}{2^k}$$

Also, we know that in the k^{th} **iteration,** $n \leq 1$**, and the loop terminates.**

k **Assume** $n_k = \frac{n_0}{2^k} = 1$**, then** $n_0 = 2^k$ **and** $k = log_2 n_0$

k **Now** k **is an integer value and** $log_2 n_0$ **does not always produce an integer value. Therefore,** $k = \lceil log_2 n_0 \rceil$ **or** $k = \lfloor log_2 n_0 \rfloor$**. The question**

is which is it? Let us try a few values of n and see if we can find a pattern.

k	n	k	$\lceil log_2 n \rceil$	$\lfloor log_2 n \rfloor$
	0	**0**	**undefined**	**undefined**
	1	**0**	**0**	**0**
	2	**1**	**1**	**1**
	3	**1**	**2**	**1**
	4	**2**	**2**	**2**
	5	**2**	**3**	**2**
	6	**2**	**3**	**2**
	7	**2**	**3**	**2**
	8	**3**	**3**	**3**
	9	**3**	**4**	**3**
	10	**3**	**4**	**3**

k It appears that $= \lfloor log_2 n_0 \rfloor$ and

Total $$T(n) = 3\lfloor log_2 n \rfloor + 2$$

Example 5

Find the time complexity of function *ExponentialCount* in Figure 5 that finds the maximum value in the array of integer given by the two parameters *A* and *n*.

```
void ExponentialCount(int n)
{   int sum=1;
    for (int a=0;a<n;a++) sum=sum*2;
    while (sum>1) sum--;
}
```

Figure 5. Program for Example 5.

Solution: Complete the following steps.

1. Simplify for-statements
2. Create a three column table putting line numbers in the first column, statements of the program in the second column, and the cost of each statement in the third column.
3. Sum the cost of the individual statements to find the cost of the program.

Line	Statement	Cost
1	**void** *ExponentialCount*(**int** *n*)	0
2	{ **int** *sum*=**1;**	1
3	**int** *a*=**0;**	1
4	**while** (*a*<*n*) {	$j = \sum_{a=0}^{n-1} 1 = n$
5	*sum*=*sum***2;**	$2j$
6	*a*++**;**	j
7	}	1
8	**while** (*sum*>**0**) {	k
9	*sum*--**;**	k
10	}	1
11	}	0

Total $$T(n) = 2k + 4j + 4$$

Total $T(n) = 2k + 4n + 4$

k

1. **The sum-loop iterates over the values sum down to 1. We can say that the sum-loop executes sum times.**
2. **Since the sum-loop executes sum times, we can say that k and sum have the same value.**
3. $sum_0 = 1, sum_{i+1} = 2sum_i$
4. $sum_1 = 2 \times sum_0 = 2 \times 1 = 2^1$
5. $sum_2 = 2 \times sum_1 = 2 \times 2 = 2^2$
6. $sum_3 = 2 \times sum_2 = 2 \times 4 = 2^3$
7. $sum_i = 2^i$
8. $sum_n = 2^n$
9. $sum = k = 2^n$

Total $T(n) = 2 \cdot 2^n + 4n + 4$