Critical Thinking 3:

Asymptotic Analysis Exercises

Alejandro Ricciardi

Colorado State University Global

CSC400: Data Structures and Algorithms

Professor: Hubert Pensado

September 1, 2024

Critical Thinking 3:

Asymptotic Analysis Exercises

This documentation is part of the Critical Thinking 3 Assignment from CSC400: Data Structures and Algorithms at Colorado State University Global. It consists of a series of exercises designed to demonstrate the principles of asymptotic analysis. Asymptotic analysis uses the Big-Oh notation.

"In computer science, the Big-Oh notation is used to describe the time complexity or space complexity of algorithms (Geeks for Geeks, 2024). Mathematically, it defines the upper bound of an algorithm's growth rate, known as the asymptotic upper bound, and is denoted as $f(n)$ is $O(g(n))$ or $f(n) \in O(g(n))$, pronounced $f(n)$ is Big-Oh of $g(n)$. The term "asymptotic" refers to the behavior of the function as its input size n approaches infinity. In the context of computer science, it describes the worst-case scenario for time complexity or space complexity. For example, an algorithm with $O(n^2)$ time complexity will grow much faster than one with $O(n)$ as the input size increases, with n representing the number of primitive operations. Primitive operations are low-level instructions with a constant execution time." (Ricciardi, 2024. Post)

Definition of Big-Oh:
Let *f(n)* and *g(n)* be functions mapping positive integers to positive real numbers.
We say that $f(n) \in O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n) \quad , \quad for\ n \geq n_0$$

(Carrano & Henry, 2018)

The Assignment Direction:
Complete the following exercises. For each exercise, show your work and all the steps taken to determine the Big-Oh for each problem. Partial points cannot be awarded without showing work.

Exercise 1
What is the Big-Oh of the following computation?

```
int sum = 0;
for (int counter = n; counter > 0; counter = counter - 2)
      sum = sum + counter;
```

Exercise 2
Suppose your implementation of a particular algorithm appears in Java as follows:

```
for (int pass = 1; pass <= n; pass++)
{
       for(int index  = 0; index < n; index++)
       {
```

```
                for(int count = 1; count < 10; count++)
                {
                        . . .

                } //end for
        } // end for
} //end for
```

The algorithm involves an array of "n" items. The previous code shows only the repetition in the algorithm, but it does not show the computations that occur within the loops.Those computations, however, are independent of "n." What is the order of the algorithm?

Exercise 3
Consider two programs, A and B. Program A requires 1000 x n^2 operations and Program B requires 2n operations.
For which values of n will Program A execute faster than Program B?

Exercise 4
Consider an array of length "n" containing unique integers in random order and in the range 1 to n + 1.
For example, an array of length 5 would contain 5 unique integers selected randomly from the integers 1 through 6. Thus the array might contain 3 6 5 1 4. Of the integers 1 through 6, notice that 2 was not selected and is not in the array.
Write Java code that finds the integer that does not appear in such an array.
Explain the Big-Oh in your code.

Submit your completed assignment as a Word doc or Notepad.

⚠️ **My notes:**

- Each exercise starts on a new page.
- $g(n)$ is the number of primitive operations.
- The summation properties of a constant

$$\sum_{i=1}^{n} c = (n - 1 + 1) \cdot c = cn \quad => \quad \sum_{i=0}^{n} c = cn$$

$$\sum_{i=1}^{n-1} c = ((n-1) - 1 + 1) \cdot c = c(n-1) \quad => \quad \sum_{i=1}^{n-1} c = c(n-1)$$

$$\sum_{i=0}^{n-1} c = ((n-1) - 0 + 1) \cdot c = cn \quad => \quad \sum_{i=0}^{n-1} c = cn$$

Exercise 1

What is the Big-Oh of the following computation?

```
int sum = 0;
for (int counter = n; counter > 0; counter = counter - 2)
      print(counter);
      sum = sum + counter;
```

In this exercise $g(n)$ is the number of primitive operations.

Step 1: Understanding the code

- The 'sum' variable initializes to '0'.
- The 'counter' loop iterates from 'n' to '1' inclusive. Note that all the variables and constants in the loop are integers.
    - The 'counter' variable initializes to 'n'.
    - The 'counter' variable is compared to '0', the loop iterates as long as 'counter' is greater than '0'.
    - The 'counter' variable is post-decremented by '2', this means that the 'counter' is compared to '0' before being decremented.
    - The 'sum' variable is incremented by the value of the 'counter' variable in each iteration of the for-loop until the 'counter' is less than or equal to '0'.

If 'n = 10'

- 1st iteration 'counter = 10' and 'sum = 0 + 10 = 10'
- 2nd iteration 'counter = 8' and 'sum = 10 + 8 = 18'
- 3rd iteration 'counter = 6' and 'sum = 18 + 6 = 24'
- 4th iteration 'counter = 4' and 'sum = 24 + 4 = 28'
- 5th iteration 'counter = 2' and 'sum = 28 + 2 = 30'

The number of iterations is $\frac{n}{2}$

If 'n = 11'

- 1st iteration 'counter = 11' and 'sum = 0 + 11 = 11'
- 2nd iteration 'counter = 9' and 'sum = 11 + 9 = 20'
- 3rd iteration 'counter = 7' and 'sum = 20 + 7 = 27'
- 4th iteration 'counter = 5' and 'sum = 27 + 5 = 32'
- 5th iteration 'counter = 3' and 'sum = 32 + 3 = 35'
- 6th iteration 'counter = 1' and 'sum = 32 + 1 = 36'

The number of iterations is $\frac{n+1}{2}$

Step 2: Counting the number of iterations

The for-loop will iterate as long as the 'counter > 0', the 'counter is initialized to 'n', and it is decremented by '2'.

If k is the number of iterations, then the last iteration would occur when $n - 2k \leq 0$. Solving for k, $k \geq \frac{n}{2}$. Using the examples above of 'n = 10' and 'n = 11', we can say that $k = \frac{n}{2}$ when n is even and $k = \frac{n+1}{2}$ when n is odd.

Justification:

An odd number is an integer of the form $n = 2c + 1$, where c is an integer.

Then $k = \frac{2c+1+1}{2} = \frac{2}{2}c + \frac{2}{2} = c + 1$.

An even number is an integer of the form $n = 2c$, where c is an integer.

Then $k = \frac{2c}{2} = \frac{2}{2}c = c$.

If $c = 5$

Then $n_{even} = 2 * 5 = 10$ and $k_{even} = \frac{n_{even}}{2} = 5 = c$

And $n_{odd} = 2 * 5 + 1 = 11$ and $k_{odd} = \frac{11+1}{2} = \frac{12}{2} = 6 = c + 1$

Step 3: Number of primitive operations $g(n)$ per instruction

- The 'sum' variable initializes to 0, it is executed only once, $g_1(n) = 1$.
- The 'counter' variable initializes to *n* by the for-loop statement this is executed only once, $g_2(n) = 1$.
- The 'counter' variable is compared to '0', this is executed in each iteration of the loop, and the number of iterations is $k_{even} = \frac{n_{even}}{2}$, then $g_3(n_{even}) = \frac{n_{even}}{2}$; and $k_{odd} = \frac{n_{odd}+1}{2}$, then $g_3(n_{odd}) = \frac{n_{odd}+1}{2}$
- The 'counter' is post-decremented by '2', this is done after each iteration of the loop and the comparison of 'counter' to '0'returns true, the number of iterations is $k_{even} = \frac{n_{even}}{2}$, then $g_4(n_{even}) = \frac{n_{even}}{2}$; and $k_{odd} = \frac{n_{odd}+1}{2}$, then $g_4(n_{odd}) = \frac{n_{odd}+1}{2}$
- The 'sum' variable is incremented by the value of the 'counter' variable in each iteration of the for-loop until the 'counter' is less than or equal to '0', then the number of times this instruction will be executed is the number of iterations $k_{even} = \frac{n_{even}}{2}$, then $g_5(n_{even}) = \frac{n_{even}}{2}$; and $k_{odd} = \frac{n_{odd}+1}{2}$, then $g_5(n_{odd}) = \frac{n_{odd}+1}{2}$

Step 4: The total of primitive operations

If n is even, $k = \frac{n}{2}$

- $g(n) = g_1(n) + g_2(n) + g_3(n) + g_4(n) + g_5(n)$
- $g(n) = 1 + 1 + k + k + k = 1 + 1 + \frac{n}{2} + \frac{n}{2} + \frac{n}{2}$
- $g(n) = 2 + \frac{3}{2}n$

If n is odd, $k = \frac{n-1}{2}$

- $g(n) = g_1(n) + g_2(n) + g_3(n) + g_4(n) + g_5(n)$
- $g(n) = 1 + 1 + k + k + k = 1 + 1 + \frac{n+1}{2} + \frac{n+1}{2} + \frac{n+1}{2} = \frac{2}{2} + \frac{2}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{n}{2} + \frac{n}{2} + \frac{n}{2}$
- $g(n) = \frac{7}{2} + \frac{3}{2}n$

Step 5: The Big-Oh notation

By the definition of the Big-Oh notation, $O\big(g(n)\big)$.

If n is even, with $g(n) = 2 + \frac{3}{2}n$ then the asymptotic complexity is $O(n)$

Justification:

$2 + \frac{3}{2}n \leq cn$, for $c = \frac{4}{2} + \frac{3}{2} = \frac{7}{2}$, when $n \geq n_0 = 2$ and n is even.

If n is odd, with $g(n) = \frac{7}{2} + \frac{3}{2}n$ then the asymptotic complexity is $O(n)$

Justification:

$2 + \frac{3}{2}n \leq cn$, for $c = \frac{4}{2} + \frac{3}{2} = \frac{7}{2}$, when $n \geq n_0 = 1$ and n is odd.

When n is even Big-Oh is $O(n)$ and when n is even Big-Oh $O(n)$, thus:

The Big-Oh of the computation is $O(n)$

In conclusion, the asymptotic analysis shows that the complexity of the algorithm is directly proportional to the size of the input n, indicating a linear growth rate. *O(n)* represents a linear type of complexity.

Exercise 2

Suppose your implementation of a particular algorithm appears in Java as follows:

```
for (int pass = 1; pass <= n; pass++)
{
	for(int index  = 0; index < n; index++)
	{
		for(int count = 1; count < 10; count++)
		{
			. . .

		} //end for
	} // end for
} //end for
```

The algorithm involves an array of "n" items. The previous code shows only the repetition in the algorithm, but it does not show the computations that occur within the loops. Those computations, however, are independent of "n." What is the order of the algorithm?

Note that the order of an algorithm refers to its "time complexity" or Big-Oh notation. Below is the time complexity analysis of the code above.

Step 1: Understanding the code

- The outer loop (‘pass’) itenerates from ‘1’ to ‘n’ inclusive. Note that all the variables and constants in the loop are integers.
    - The ‘pass’ variable initializes to ‘1’.
    - The ‘pass’ variable is compared to ‘n’, the loop iterates as long as ‘pass’ is less than or equal to ‘n’.
    - The ‘pass’ variable is post-incremented by ‘1’, this means that the ‘pass' variable is compared to ‘n’ before being incremented.
- The middle loop (‘index’) itenerates from ‘0’ to ‘n’ exclusive. Note that all the variables and constants in the loop are integers.
    - The ‘index’ variable initializes to ‘0’.
    - The ‘index’ variable is compared to ‘n’, the loop iterates as long as ‘index’ is less than ‘n’.
    - The ‘index’ variable is post-incremented by ‘1’, this means that the ‘pass' variable is compared to ‘n’ before being incremented.
- The inner loop (‘count’) itenerates from ‘1’ to ‘10’ exclusive. Note that all the variables and constants in the loop are integers.
    - The ‘count’ variable initializes to ‘1’.
    - The ‘count’ variable is compared to ‘10’, the loop iterates as long as ‘count’ is less than ‘10’.
    - The ‘count’ variable is post-incremented by ‘1’, this means that the ‘count' variable is compared to ‘10’ before being incremented.

Step 2: Counting the number of iterations

- The outer loop will iterate as long as 'pass' is less than or equal to 'n'.
  and it is post-incremented by '1', starting at 'pass = 1'.
  If k is the number of iterations:

$$k = \sum_{pass=1}^{n} 1 = 1 \cdot n = n$$

  Thus, the loop iterates $\boldsymbol{n}$ **times**.
- The outer middle will iterate as long as the 'index' is strictly less than 'n'.
  and it is post-incremented by '1', starting at 'index = 0'.
  If k is the number of iterations:

$$k = \sum_{index=0}^{n-1} 1 = 1 \cdot n = n$$

  Thus, the loop iterates n times. However, since the loop is nested within the outer loop, it iterates a total of $n \cdot n = \boldsymbol{n^2}$ **times.**
- The inner loop will iterate as long as 'count' is strictly less than '10'.
  and it is post-incremented by '1', starting at 'pass = 1'.
  If k is the number of iterations:

$$k = \sum_{count=1}^{10-1} 1 = 1 \cdot (10 - 1) = 9$$

  Thus, the loop iterates 9 times. However, since the loop is nested within the middle loop, it iterates a total of $9 \cdot n^2 = \boldsymbol{9n^2}$ **times.**

Step 3: Number of primitive operations $g(n)$ per instruction

- The number of primitive operations for the outer loop is:
  - The 'pass' variable initializes to '1' by the for-loop statement this is executed only once, then $g_1(n) = 1$.
  - The 'pass' variable is compared to 'n', this is executed in each iteration of the loop, and the number of iterations is $k = n$, then $g_2(n) = n$.
  - The 'pass' is post-incremented by '1', this is done after each iteration of the loop and the comparison of 'pass' to 'n' returns true, the number of iterations is $k = n$, then $g_3(n) = n$.

  The total of primitive operations for the outer loop is:

$$g_{outer\ loop}(n) = g_1(n) + g_2(n) + g_3(n) = 1 + n + n = 1 + 2n$$

$$g_{outer\ loop}(n) = 1 + 2n$$

- The number of primitive operations for the middle loop is:
  - The 'index' variable initializes to '0' by the for-loop statement this is executed only once. However, since the middle loop is nested within the outer loop, the 'index' variable is initialized n times, then $g_1(n) = 1 \cdot n = n$.

- The 'index' variable is compared to 'n', this is executed in each iteration of the loop, and the number of iterations is $k = n^2$, then $g_2(n) = n^2$.
- The 'index' is post-incremented by '1', this is done after each iteration of the loop and the comparison of 'index' to 'n' returns true, and the number of iterations is $k = n^2$, then $g_3(n) = n^2$.

The total of primitive operations for the outer loop is:

$$g_{middle\ loop}(n) = g_1(n) + g_2(n) + g_3(n) = +n + n^2 + n^2 = n + 2n^2$$

$$g_{middle\ loop}(n) = n + 2n^2$$

- The number of primitive operations for the inner loop is:
  - The 'count' variable initializes to '1' by the for-loop statement this is executed only once. However, since the inner loop is nested within the middle loop, the 'count' variable is initialized n^2 times, then $g_1(n) = 1 \cdot n^2 = n^2$.
  - The 'index' variable is compared to '10', this is executed in each iteration of the loop, and the number of iterations is $k = 9n^2$, then $g_3(n) = 9n^2$
  - The 'index' is post-incremented by '1', this is done after each iteration of the loop and the comparison of 'index' to '10' returns true, and the number of iterations is $k = 9n^2$, then $g_3(n) = 9n^2$.

  The total of primitive operations for the outer loop is:

$$g_{inner\ loop}(n) = g_1(n) + g_2(n) + g_3(n) = n^2 + 9n^2 + 9n^2 = 19n^2$$

$$g_{inner\ loop}(n) = 19n^2$$

Step 4: The total of primitive operations done by the loops

- $g_{loops}(n) = g_{outer\ loop}(n) + g_{middle\ loop}(n) + g_{inner\ loop}(n)$
- $g_{loops}(n) = 1 + 2n + n + 2n^2 + 19n^2$
- $g_{loops}(n) = 21n^2 + 3n + 1$

Step 5: The Big-Oh notation

Since the array computation in the inner loop, the only place where computations are done can be considered a constant said we called 'c', then the number of times that computation will be is executed is $g_{inner\ comp} = 9cn^2$.

With $g_{inner\ comp}(n) = 9cn^2$ then the asymptotic complexity is $O_{inner\ comp}(n^2)$.

Justification:

$9cn^2$ with c being a constant $\leq (9c)n^2 = cn$. By the definition of the Big-Oh notation, $O_{inner\ comp}(n^2)$.

In other words, we can ignore the array computations in terms of time complexity analysis because the computations within the inner loop are independent of "n", meaning that the computations in the inner loop, in the context of the time complexity, are constant and are ignored by the Big-Oh nation.

The time complexity of the loops is $O_{loops}(n^2)$:
with $g_{loops}(n) = 21n^2 + 3n + 1$ then the asymptotic complexity is $O_{loops}(n^2)$
Justification:
$21n^2 + 3n + 1 \leq (21 + 3 + 1)n^2 = cn$, for $c = 25$, when when $n \geq n_0 = 1$. By the definition of the Big-Oh notation.

The overall complexity of the algorithm is $O(n^2)$
Justification:
With $O_{loop}(n^2)$, $g_{loop}(n) = n^2$ and $O_{inner\ comp}(n^2)$, $g_{inner\ comp}(n) = n^2$
$n^2 + n^2 = 2n^2 = cn^2$, for c=2. By the definition of the Big-Oh notation, the overall complexity of the algorithm is $O(n^2)$.

Therefore

The order of the algorithm is $O(n^2)$

In conclusion, the computations within the inner loop are independent of the input 'n', meaning that the computation in inner loop, in the context of the time complexity is constant. Since this is the only place where the actual computations occur, we can ignore it in terms of time complexity analysis. The order of the algorithm is $O(n^2)$ indicating that time complexity is quadratic. $O(n^2)$ is a quadric type time complexity.

Exercise 3

Consider two programs, A and B. Program A requires 1000 x n^2 operations and Program B requires 2n operations.
For which values of n will Program A execute faster than Program B?

The number of operations for program A is $g_A(n) = 1000n^2$ where n is the number of inputs.
The number of operations for program B is $g_B(n) = 2n$ where n is the number of inputs.

When comparing algorithms a primitive operation corresponds to 1 unit of time, $g_A(n)$ and $g_B(n)$ can be defined as a growth rate where the out is the duration and the input is the number of operations 'n'. Therefore to find the 'n' values where Program A executes faster than Program B, we need to identify where $g_A(n) < g_B(n)$.

$$g_A(n) < g_B(n)$$
$$1000n^2 < 2n$$
$$\frac{n^2}{n} < \frac{2}{1000}$$
$$n < 0.002$$

The values of 'n' where Program A executes faster than Program B are between -∞ and 0.002 excluded, (-∞,0.002). However, operations can be defined only as whole numbers or integers, 'n' the number of operations can be a decimal, it can be defined only as an integer Therefore, Therefore, Program A will never run faster than Program B.

Another method that provides a more accurate representation of the relationship is to use the time complexity of the algorithms to compare the algorithms.
The time complexity of Program A is $O_A(n^2)$
Justification:
$1000n^2 \leq (1000)n^2 = cn$, for $c = 1000$, when $n \geq n_0 = 1$
The time complexity of Program B is $O_A(n)$
Justification:
$2n \leq (2)n = cn$, for $c = 2$, when $n \geq n_0 = 1$

To find the value of 'n' where Program A executes faster than Program B, we need to identify where $O_A(n^2) < O_B(n)$.

$$n^2 < n \quad => \quad \frac{n^2}{n} < 1 \quad => \quad n < 1$$

this is not possible when 'n' is an integer and $n \geq n_0 = 1$

No values of 'n' exist where Program A executes faster than Program B.

Exercise 4

Consider an array of length "n" containing unique integers in random order and in the range 1 to n + 1.
For example, an array of length 5 would contain 5 unique integers selected randomly from the integers 1 through 6. Thus the array might contain 3 6 5 1 4. Of the integers 1 through 6, notice that 2 was not selected and is not in the array.
Write Java code that finds the integer that does not appear in such an array.
Explain the Big-Oh in your code.

Note that the expected sum of integers in array length from 1 to $n + 1$ is:

$$expectedSum = \sum_{i=0}^{n=1} i = \frac{(n+1)(n+2)}{2}$$

The Java code:

```java
/**
* Finds the missing number
*
* @param array The array of unique integers with one integer missing.
* @param n     The length of the array
* @return The missing integer
*/
public static int missingNum(int[] array, int n) {
	// The expected sum of integers from 1 to n+1
	int expectedSum = (n + 1) * (n + 2) / 2;
	int actualSum = 0;

	// The actual sum of the numbers in the array
	for (int i = 0; i < n; i++) {
		actualSum += array[i];
	}
	return expectedSum - actualSum;
}
```

Step 1: Understanding the code

- The ‘expectedSum’ variable initializes to ‘(n + 1) * (n + 2) / 2’.
- The ‘actualSum’ variable initializes to ‘0’.
- The ‘array’ loop iterates through from the ‘0’ to the ‘n’ exclusive. Note that all the variables and constants in the loop are integers.
    - The ‘i’ variable initializes to ‘0’.
    - The ‘i’ variable is compared to ‘n’, the loop iterates as long as ‘i’ is lesse than ‘n’.
    - The ‘i’ variable is incremented by ‘1’, this means that the ‘i’ is compared to ‘n’ before being incremented.
    - The ‘atualSum’ variable is incremented by the value of the ‘arrau[i]’ variable in each iteration of the for-loop until the ‘i’ is equal to or more than ‘n’.
- The function returns ‘expectedSum – actualSum’

Step 2: Counting the number of iterations in the array loop

- The array loop will iterate as long as 'i' is less 'n'.
  and it is post-incremented by '1', starting at 'i = 0'.
  If k is the number of iterations:

$$k = \sum_{i=0}^{n-1} 1 = 1 \cdot n = n$$

  Thus, the loop iterates n times.

Step 3: Number of primitive operations $g(n)$ per instruction

- The 'expectedSum' variable initializes to '0', it is executed only once, $g_1(n) = 1$.
- The 'actualSum' variable initializes to ''(n + 1) * (n + 2) / 2', it is executed only once, $g_2(n) = 1$.
- The number of primitive operations for the array loop is:
  - The 'i' variable initializes to '0' by the for-loop statement this is executed only once, $g_3(n) = 1$.
  - The 'i' variable is compared to 'n', this is executed in each iteration of the loop, and the number of iterations is $k = n$, then $g_4(n) = n$.
  - The 'i' is post-incremented by '1', this is done after each iteration of the loop and the comparison of 'pass' to 'n' returns true, the number of iterations is $k = n$, then $g_5(n) = n$.
  - The 'actualSum' is incremented with the value of 'array[i]' variable in each iteration of the for-loop until the 'i' is equal to or more than 'n'. The number of times this instruction will be executed is the number of iterations *k = n*, then $g_6(n) = n$

  The total of primitive operations for the outer loop is:

$$g_{oop}(n) = g_3(n) + g_4(n) + g_5(n) + g_5(n) = 1 + n + n + n = 1 + 3n$$

$$g_{loop}(n) = 1 + 3n$$

- The function returns 'expectedSum – actualSum', this instruction is performed only once, the $g_7(n) = 1$

Step 4: The total of primitive operations

- $g(n) = g_1(n) + g_2(n) + g_{loop}(n) + g_7(n)$
- $g(n) = 1 + 1 + 1 + 3n + 1 = 4 + 3n$
- $g(n) = 4 + 3n$

Step 5: The Big-Oh notation

The Big-Oh notation is *O(n).*

Justification:

$4 + 3n \leq (4 + 3)n = cn$, for $c = 7$, when $n \geq n_0 = 1$. By the definition of the Big-Oh notation, the overall complexity of the code is $O(n)$.

The Big-Oh notation of my code is $O(n)$.

The Big-Oh notation $O(n)$ in my code indicates that the complexity of the code grows in a 1-to-1 proportional relationship with the size of the input, ‘n’, which is the size of the ‘actualArray’ array. In other words, the complexity of the code grows linearly with the size of the array ‘n’, this is known as linear complexity. For example, in terms of time complexity, as the size of the array gets larger, the time it takes to run the code also increases proportionally in 1-to-1 relationship. This is because the duration that it takes to iterate through each element in the loop is the most time-consuming part of the code and all other primitive operations are ignored by the Big-Oh notation, due to their minimal impact on the overall complexity of the code.

References

Carrano, F. M., & Henry, T. M. (2018, January 31). Algorithms: Algorithm analysis. *Data structures and abstractions with Java* (5th Edition). Pearson.

Ricciardi, A. (2024, August 29). Discussion forums module 3 [Post]. CSC400: Data Structures and Algorithms. CSU Global Computer Science Department.