ICS 240
Introduction to Data Structures

Jessica Maistrovich

Metropolitan State University

Complexity Analysis

Introduction to Runtime Analysis

- An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.

input → algorithm → output

- **Goal:** Given two algorithms, which one is better?
- **Criteria?**
  - Faster? – Smaller? (less memory)
  - Easier to implement? – Easier to maintain?
  - Worst case? – Average case?
- The study of algorithm efficiency is called **complexity analysis**
- In ICS 240, we will focus primarily on which algorithm is **faster**
- Runtime analysis of an algorithm is a method that is used to estimate the running time of an algorithm

How Long Will My Algorithm Take to Solve the Problem: **Approach 1**

- There are different approaches to **measure** how long an algorithm takes
- Implement the algorithm, execute it, and use a stop watch to measure the time
- Issues with this method:
  - Time consuming: You need to implement the algorithm first, wasting time if you discover the algorithm is not efficient
  - There are external factors that affect time taken by the algorithm (e.g., hardware, compiler, libraries)
    - the algorithm may take different time if run on a different computer
  - What about **input data** sets that were not tested?
    - The algorithm may take different time if you change the input data or size of data

How Long Will My Algorithm Take to Solve the Problem (continued): **Approach 2**

- Count the operations the algorithm performs
  - Count the number of operations that are performed for a given input size (e.g., number of items stored in an array)
    - uses a high-level description of the algorithm instead of an implementation in a programming language
  - Characterize running time as a function of the input size (commonly represented as variable **N**)
    - Characterize how the number of operations changes as N changes
  - This method of evaluation is independent of the actual input or implementation environment

Goal

Characterize how algorithm performance changes as the input size (**N**) changes

Two algorithms

Find the maximum element *in a bag of integers*

```
1.    static int findMax(int[] arr, int N) {
2.        int currentMax = arr[0];
3.        for  (int i= 0; i < N; i++)
4.              if (currentMax < arr[i] )
5.                          currentMax = arr[i];
6.        return currentMax;
7.    }
```

- **Counting operations:**
- **Line 1:** method signature: no time
- **Lines 2 and 3:** 1 operation each(assignment) for a total of 2 operations
- **Line 3:** loop iterates N times – so we will multiply N by the count of the loop body
- **Lines 4 and 5:** 4 operations (<, =, <, ++)
- **Line 6:** 1 operation (return)
- **Total:** $2 + (N)*4 + 1 = 4N + 3$

Two algorithms (continued)

```
Find the maximum element in an ordered list of
integers
1.     static int findMax(int[] arr, int N) {
2.             return arr[N-1];
3.     }
```

- **Counting operations:**
- **Line 1:** method signature: no time
- **Line 2:** 1 operation (return)
- **Total:** always 1 (independent of number of elements in the list)

Which of the two algorithms leads to a better approach to finding max?

Approach	Number of Operations
Bag (unordered)	4**N** + 3
Ordered List	1

Big-O Notation

Introducing Big-O Notation

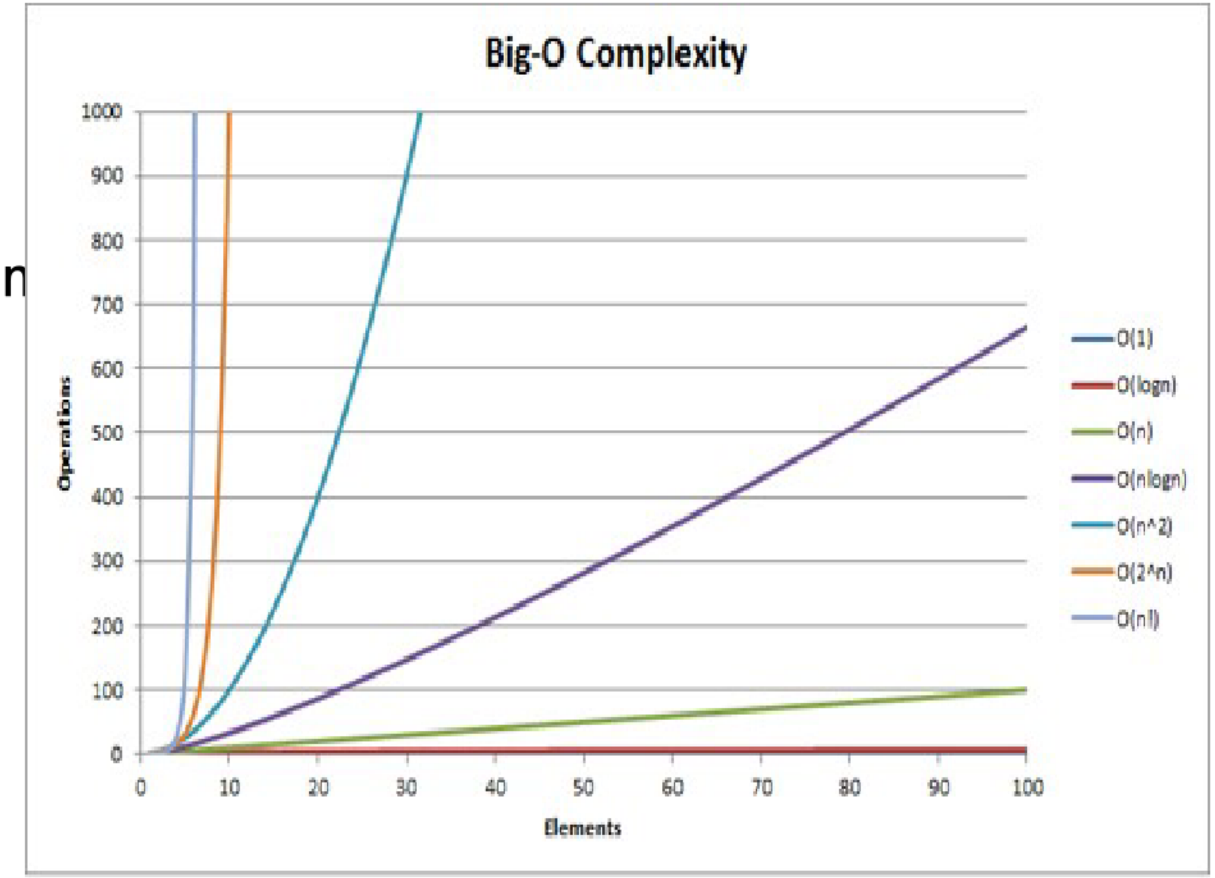
- **Big-O** notation is used in Computer Science to describe the performance or complexity of an algorithm
- Big-O classes of algorithms:
  - **O(1)** – constant time algorithm
    - The performance of the algorithm does not depend on input size
  - **O(N)** -- linear time algorithm
    - The performance of the algorithm changes linearly (in direct proportion) as the input size grows
  - **O(N^2)** – quadratic time algorithm
    - The performance of the algorithm changes as the square of the input size
- If two algorithms perform the same task with different big-O times, then with sufficiently large input, the algorithm with the better big-O analysis will perform faster
- Big-O notation **approximates** performance

Common Big-O Functions

grows more slowly

- Constant function $O(1)$
- The logarithm function $O(\log n)$
- The linear function $O(n)$
- The $n \log n$ function $O(n \log n)$
- The quadratic function $O(n^2)$
- Cubic and other polynomial functions
- The exponential function $O(2^n)$
- The factorial function $O(n!)$

grows faster



Asymptotic Growth Rates by Numbers

c	log n	n	n log n	n^2	n^3	2^n	n!
1000	0	1	0	1	1	2	1
1000	1	2	2	4	8	4	2
1000	2	4	8	16	64	16	24
1000	3	8	24	64	512	256	40320
1000	4	16	64	256	4096	65536	2.092E+13
1000	5	32	160	1024	32768	4294967296	2.631E+35
1000	6	64	384	4096	262144	1.84467E+19	1.269E+89
1000	7	128	896	16384	2097152	3.40282E+38	3.86E+215
1000	8	256	2048	65536	16777216	1.15792E+77	#NUM!

Big-O Notation Simplification

- Drop constants:
  - 1,000,000 = 1 * 1,000,000 = O(1)
  - A million is big-O of 1
  - We do not worry about constants because they do not change as the input size change

- Keep only the dominant term (fastest growing term)
  - 3n + 5 is O(n)
    - 3n + 5 = O(3n) – ignore 5 because 3n is the dominant term
    - O(3n) = O(n) – drop constants

Examples

- $f(n) = 4n^2$
  - $f(n)$ is $O(n^2)$
- $f(n) = 50n^3 + 20n + 4$
  - We say that $f(n)$ is $O(n^3)$
  - It is also true to say $f(n)$ is $O(n^3+n)$
    - However, this is not useful, as n^3 exceeds by far n, for large values
  - It is also true to say $f(n)$ is $O(n^5)$
    - However, we should be as close as possible to $f(n)$

Consider the function $g(n) = 100 + n^2 + 2^n$. Which of the following is true:

- $g(n) = O(1)$
- $g(n) = O(n^2)$
- $g(n) = O(2^n)$

What is the order of complexity of this code snippet?

$2*N^3 + 200*N + 52$

https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/220exercises1.pdf

Try it!

Assume that each of the expressions below gives the processing time $T(n)$ spent by an algorithm for solving a problem of size n. Select the dominant term(s) having the steepest increase in n and specify the lowest Big-Oh complexity of each algorithm.

Expression	Dominant term(s)	$O(\ldots)$
$5 + 0.001n^3 + 0.025n$		
$500n + 100n^{1.5} + 50n \log_{10} n$		
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$		
$n^2 \log_2 n + n(\log_2 n)^2$		
$n \log_3 n + n \log_2 n$		
$3 \log_8 n + \log_2 \log_2 \log_2 n$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$2n + n^{0.5} + 0.5n^{1.25}$		
$0.01n \log_2 n + n(\log_2 n)^2$		
$100n \log_3 n + n^3 + 100n$		
$0.003 \log_4 n + \log_2 \log_2 n$		

https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/220exercises1.pdf

Answers to Try it:

Expression	Dominant term(s)	$O(\dots)$
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n\log_{10} n$	$100n^{1.5}$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2 \log_2 n + n(\log_2 n)^2$	$n^2 \log_2 n$	$O(n^2 \log n)$
$n \log_3 n + n \log_2 n$	$n \log_3 n,\ n \log_2 n$	$O(n \log n)$
$3 \log_8 n + \log_2 \log_2 \log_2 n$	$3 \log_8 n$	$O(\log n)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01n \log_2 n + n(\log_2 n)^2$	$n(\log_2 n)^2$	$O(n(\log n)^2)$
$100n \log_3 n + n^3 + 100n$	n^3	$O(n^3)$
$0.003 \log_4 n + \log_2 \log_2 n$	$0.003 \log_4 n$	$O(\log n)$

What is the order of complexity of this code snippet?

```
for (int i=0; i < N-3; i++) {
     System.out.println (N);
}
```

What is the order of complexity of this code snippet?

```
if (N % 2 == 0)    {
    for (int i=0; i < N-3; i++) {
         System.out.println (N);
    }
} else
    System.out.println(2);
```

Example 1

- Assume you are given an array, `originalArr`
- Compute another array, `averageArr`, such that, for each value of `i`, `averageArr[i]` is the average of all elements in positions `0` to `i` in `originalArr`

originalArr	averageArr	
10	10	
20	15	(10+20)/2
30	20	(10+20+30)/3
40	25	(10+20+30+40)/4
50	30	(10+20+30+40+50)/5
60	35	(10+20+30+40+50+60)/6

Example 1 Algorithms

- Usually, there is more than one way to solve a given problem

Solution 1

```
int sum;
for (int i=0 ; i < numItems; i++) {
    sum = 0;
    for (int j = 0; j <=i; j++){
        sum = sum + originalArr[j];
        averageArr[i] = sum / (i+1);
    }
}
```

Solution 2

```
int sum = 0;
for (int i=0 ; i < numItems; i++) {
    sum = sum + originalArr[i];
    averageArr[i] = sum;
}
for (int i=0 ; i < numItems; i++)
    averageArr[i] = averageArr[i] / (i+1);
```

Which solution is faster? Why?

Example 1

Usually, there is more than one way to solve a given problem.

Solution 1

```
int sum;
for (int i=0 ; i < numItems; i++){
    sum = 0;
    for (int j = 0; j <=i; j++){
        sum = sum + originalArr[j];
        averageArr[i] = sum / (i+1);
    }
}
```

$O(n^2)$

Solution 2

```
int sum = 0;
int[] averageArr = new
int[originalArr.length];
for (int i=0 ; i < numItems; i++){
    sum = sum + originalArr[i];
    averageArr[i] = sum;
}
for (int i=0 ; i < numItems; i++)
    averageArr[i] = averageArr[i] / (i+1);
```

$O(n)$

What is the order of complexity of this code snippet?

```
for (int j = N; j > 0; j = j/10  )    {

        System.out.println ( j );

}
```

What is the order of complexity of this code snippet?

```
for (int k=1; k < N+3; k++) {

        for (int m = N; m > 0; m--) {

            System.out.println(m*k);

        }

    }
```

What is the order of complexity of this code snippet?

```
for (int i = 1; i < 14; i++) {

        for (int j=N; j > 1; j--)   {

            if (i%2 == 0)  {

                System.out.println(i*j);
            }
        }

}
```

Try it!

https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/220exercises1.pdf

1. Work out the computational complexity of the following piece of code:

```
for( int i = n;  i > 0;  i /= 2 ) {
  for( int j = 1;  j < n;  j *= 2 ) {
    for( int k = 0;  k < n;  k += 2 ) {
         ... // constant number of operations
    }
  }
}
```

2. Work out the computational complexity of the following piece of code.

```
for ( i=1; i < n; i *= 2 ) {
  for ( j = n; j > 0; j /= 2 ) {
    for ( k = j; k < n; k += 2 ) {
      sum += (i + j * k );
    }
  }
}
```

3. Work out the computational complexity of the following piece of code assuming that $n = 2^m$:

```
for( int i = n;  i > 0;  i-- ) {
  for( int j = 1;  j < n;  j *= 2 ) {
    for( int k = 0;  k < j;  k++ ) {
         ... // constant number C of operations
    }
  }
}
```

https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/220exercises1.pdf

Try it!

4. Work out the computational complexity (in the "Big-Oh" sense) of the following piece of code and explain how you derived it using the basic features of the "Big-Oh" notation:

```
for( int bound = 1; bound <= n; bound *= 2 ) {
   for( int i = 0; i < bound; i++ ) {
      for( int j = 0; j < n; j += 2 ) {
         ... // constant number of operations
      }
      for( int j = 1; j < n; j *= 2 ) {
         ... // constant number of operations
      }
   }
}
```

https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/220exercises1.pdf

Answers to Try it:

1. In the outer `for`-loop, the variable `i` keeps halving so it goes round $\log_2 n$ times. For each `i`, next loop goes round also $\log_2 n$ times, because of doubling the variable `j`. The innermost loop by `k` goes round $\frac{n}{2}$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O\left(n(\log n)^2\right)$.

2. Running time of the inner, middle, and outer loop is proportional to n, $\log n$, and $\log n$, respectively. Thus the overall Big-Oh complexity is $O(n(\log n)^2)$.

   More detailed optional analysis gives the same value. Let $n = 2^k$. Then the outer loop is executed k times, the middle loop is executed $k+1$ times, and for each value $j = 2^k, 2^{k-1}, \ldots, 2, 1$, the inner loop has different execution times:

j	Inner iterations
2^k	1
2^{k-1}	$(2^k - 2^{k-1})\frac{1}{2}$
2^{k-2}	$(2^k - 2^{k-2})\frac{1}{2}$
$\ldots$	$\ldots$
2^1	$(2^k - 2^1)\frac{1}{2}$
2^0	$(2^k - 2^0)\frac{1}{2}$

   In total, the number of inner/middle steps is

$$\begin{aligned} 1 + k\cdot 2^{k-1} - (1 + 2 + \ldots + 2^{k-1})\frac{1}{2} &= 1 + k\cdot 2^{k-1} - (2^k - 1)\frac{1}{2} \\ &= 1.5 + (k-1)\cdot 2^{k-1} \equiv (\log_2 n - 1)\frac{n}{2} \\ &= O(n\log n) \end{aligned}$$

   Thus, the total complexity is $O(n(\log n)^2)$.

https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/220exercises1.pdf

Answers to Try it:

3. The outer `for`-loop goes round n times. For each `i`, the next loop goes round $m = \log_2 n$ times, because of doubling the variable `j`. For each `j`, the innermost loop by `k` goes round `j` times, so that the two inner loops together go round $1 + 2 + 4 + \ldots + 2^{m-1} = 2^m - 1 \approx n$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O(n^2)$.

4. The first and second successive innermost loops have $O(n)$ and $O(\log n)$ complexity, respectively. Thus, the overall complexity of the innermost part is $O(n)$. The outermost and middle loops have complexity $O(\log n)$ and $O(n)$, so a straightforward (and valid) solution is that the overall complexity is $O(n^2 \log n)$.