

ICS 240: Introduction to Data Structures

Module 5 Complexity Analysis

Running time Analysis of Algorithms

- Reading:
 - Chapter 1:
 - Section 1.2: pages 16 to 26
 - Appendix G: pages 805 - 806

Main points

- Runtime analysis
- Counting operations
- How does the algorithm scale?
- Big-O notation
- Common big-O functions
- Analysis: Linked lists vs arrays
 - complexity of adding
 - complexity of removing

Introduction

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



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

Example 1

- Assume you are given an array, `originalArr`.
- **Required:** compute another array, `averageArr`, such that, for each value of `i`, `averageArr[i]` is the average of all elements at position 0 to `i` in `originalArr`

`originalArr`

10
20
30
40
50
60

`averageArr`

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

Example 1 (continued)

- Usually, there are more than one method to solve a given problem.

Solution 1

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

Solution 2

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

Which solution is faster? Why?

Example 2:

Real Life example (as discussed in the textbook)

- Assume while you are at the top of Eiffel tower with a friend, and your friend asked you: “How many steps there are to the bottom?”
- You have three different options to answer your friends question:
 - **Option 1:** take the stair way and keep a tally with you, then take the stairs down to tell her the answer.
 - **Option 2:** assume you cannot keep a tally, however, you have to go back and forth to record each step you found. Walk down one step, put your hat, go back and let your friend increment the tally.
 - **Option 3:** ask someone who already knows the answer.

How long my algorithm will 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 in 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 affects time taken by the algorithm (e.g., hardware, compiler, libraries, etc.)
 - the algorithm may take a different time if run on a different computer
 - What about **input data** sets that were not tested?
 - The algorithm may take a different time if you change the input data size.

How long my algorithm will take to solve the problem? **Approach 2**

- Count the operations that your algorithm performs
 - Count the number of operations that are performed for a given input size (e.g., array length)
 - 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**) *but not always*
 - Characterize how the number of operations changes as **n** changes
- This method of evaluation is independent of the actual input or implementation environment.

Our goal:

Characterize how algorithm performance changes as the Input size (n) changes.

How Many Operations are Executed by the Following Method?

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

Line 2 & 3: 2 operations (assignment)

Line 6: 1 operation (return)

Lines 3, 4 and 5: 4 operations (++, =, < & <) * the number of times the loop is iterated.

Line 3: loop iterates **n** times for an array of size **n**

Total: $2 + 1 + 4*(n) = 4n + 3$

What Counts as an Operation?

- **A primitive operation** is a basic unit of execution that does not change as the input changes. These are counted as 1 operation:
 - Addition (+)
 - Assignment (=)
 - Method call
 - Returning from a method
 - Reading an element from a specific index in the array
 - Comparison (==)
 - Variable declaration
- Ignore:
 - Initialization time
 - Implementation of specific operations (e.g., addition or `if` condition)
- Focus only on how the performance of an algorithm scales:
 - If input is **twice** as big, how many **more operations** will we need?

How Many Operations are Executed by the Following Method?

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 Data Structure Leads to a Better Approach to Finding Max?

Approach	Number of Operations
Bag (unordered)	$4n + 3$
Ordered List	1

Main ideas for Performance Analysis

- #1: Count the number of operations
 - Avoid effects of specific hardware, software, or input
- #2: Focus on how the performance scales
 - As the size of the input grows bigger, how does the algorithm perform?

How Long Will My Algorithm Take to Solve the Problem?

- Count the operations that your algorithm performs
 - Can use a high-level description of the algorithm instead of an implementation in a programming language
 - Count the number of operations that are performed for a given input size (e.g., number of items stored in an array)
 - 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.

Example 3: counting operations

- How many operations are executed to search for the letter 'a' in the string 'san diego'.

```
public static boolean hasLetter(String word, char letter){  
  
    for (int i=0 ; i < word.length() ; i++)  
        if (word.charAt(i) == letter )  
            return true;  
    return false;  
  
}
```

Example 3 (continued)

- `hasLetter("san diego ", 'a ')` takes 7 operations.
- Is there another letter, `x`, where `hasLetter("san diego ", x)` takes a different number of operations

- Yes. Any other letter will take a different number of operations.
- For example, `hasLetter("san diego ", 's')` takes only 4 operations to return true when finding the character ' s ' after one iteration of the loop.
- `hasLetter("san diego ", 'Y')` will perform many more operations to search through the entire string "san diego " for the letter 'Y' before returning false because the letter 'Y' is not found in "san diego".

```
public static boolean hasLetter(String word, char letter){  
    for (int i=0 ; i < word.length() ; i++)  
        if (word.charAt(i) == letter )  
            return true;  
    return false;  
}
```

hasLetter("san diego", 'y')
3 operations in each iteration of the for loop
9 iterations because "san diego" length = 9
2 additional operations for (i = 0 and return)
 $3*9 + 2 = 29$ total operations

Example 3 conclusion

- How many operations are taken to search for letter `x` in `"san diego"` depends on the letter we are searching for.
- Two steps to count the number of operations performed by a `for` loop:
 - How many operations is each iteration?
 - test loop counter `<` boundary
 - increment the loop counter
 - Test whether the letter at current iteration equals to input letter
 - How many iterations are there?
 - One iteration of the `for` loop for every single position in the input string.
- Line by line counting of operations is painful.

How does the number of operations changes as **n** changes?

```
if (word.charAt(i) == letter)
    return true
```

2 OPS

Input: word (String)

This condition check takes 1 operation to check and sometimes 1 operation to return

→ takes 1 or 2 operations

Running time is **constant** because it does not depend on the size of the input

```
int count = 0;
for (int i = 0; i < word.length(); i++)
    Count++
```

1

1

1

2 + 3n OPS

Input: word (String)

The number of operations is going to grow as the size of input grows because each new character in the string adds one more iteration

→

Running time is $3n + 2$

Running time is **linear** in terms of the size of input

Big-O Notation

Introducing Big-O Notation

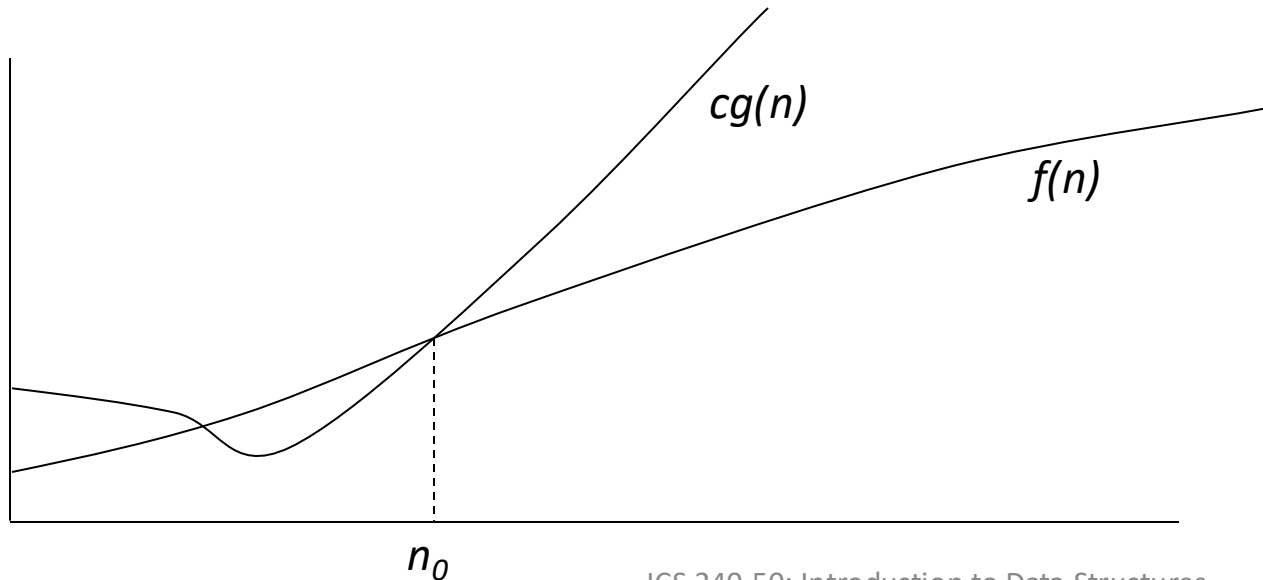
- **Big O** notation is used in Computer Science to describe the performance or complexity of an algorithm.
- Big-O classes of algorithms:
 - **$O(1)$** – constant time algorithm
 - The performance of the algorithm does not depend on input size
 - **$O(n)$** -- linear time algorithm
 - The performance of the algorithm changes linearly as the input size
 - **$O(n^2)$** – quadratic time algorithm.
 - The performance of the algorithm changes exponentially with 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

Formal Big-O notation

$$f(n) = O(g(n))$$

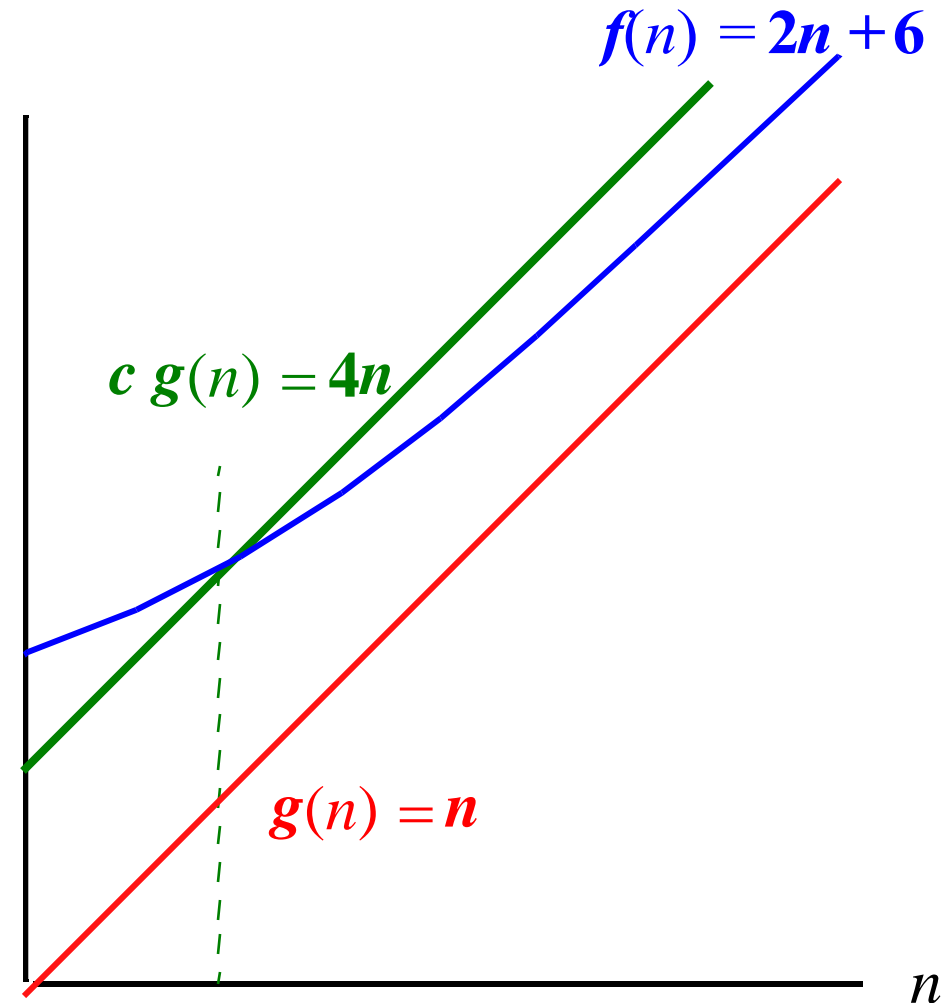
means that there are constants n_0 and c such that for every $n > n_0$,

$$f(n) < c g(n)$$



Graphic Illustration

- Given, $f(n) = 2n + 6$
- We need to find a function $g(n)$ and a constant c such that $f(n) < cg(n)$
- We found that:
 - $g(n) = n$ and $c = 4$
 - $2n + 6 < 4n$ for all $n > 3$
- Then we can say $f(n)$ is $O(n)$
 - In other words, the order of $f(n)$ is n



Formal Big-O Notation (continued)

- Big-O notation captures the the rate of growth of a function
- Intuitively, $f(n)$ is $O(g(n))$ if $f(n)$ grows at the same rate or more slowly than $g(n)$.
 - i.e., growth rate of $f(n)$ is \leq growth rate of $g(n)$
- Example: $5n + 30$ is $O(n)$
 - For $n \geq 30$
 - $5n + 30 \leq 5n + n = 6n$.
 - Let $c = 6$ and $n_0 = 30$. **$\rightarrow 5n + 30 < 6n$ for all $n > 30$**

Big-O Notation Simplification

- Drop constants:
 - $1000000 = O(1)$
 - A million is big O of 1
 - We do not worry about constant because they do not change as the input size change
- Keep only dominant term (fastest growing term)
 - $3n + 3$ is $O(n)$
 - $3n + 3 = O(3n)$ – ignore 3 because $3n$ is the dominant term
 - $O(3n) = O(n)$ – drop constants

Example: what is n?

- Consider the following method that an input string and threshold. The method then checks whether the count of occurrences of letter `a` in the string is above the given threshold.
- Which of the following makes the most sense to consider as the "**size of input**" for the following method:
 - a) the length of the string `s1` , or
 - b) the size of the integer threshold.

```
public static boolean count_a(String s1, int threshold){  
    int total = 0;  
    for (int i = 0; i < s1.length(); i++) {  
        char c = s1.charAt(i);  
        if (c == 'a')  
            total++;  
    }  
    if (total > threshold) {  
        System.out.println("Wow! That's a lot of a's!");  
        return true;  
    }  
    else  
        return false;  
}
```

We loop over the string
so the input size is the
length of `s1`

Examples

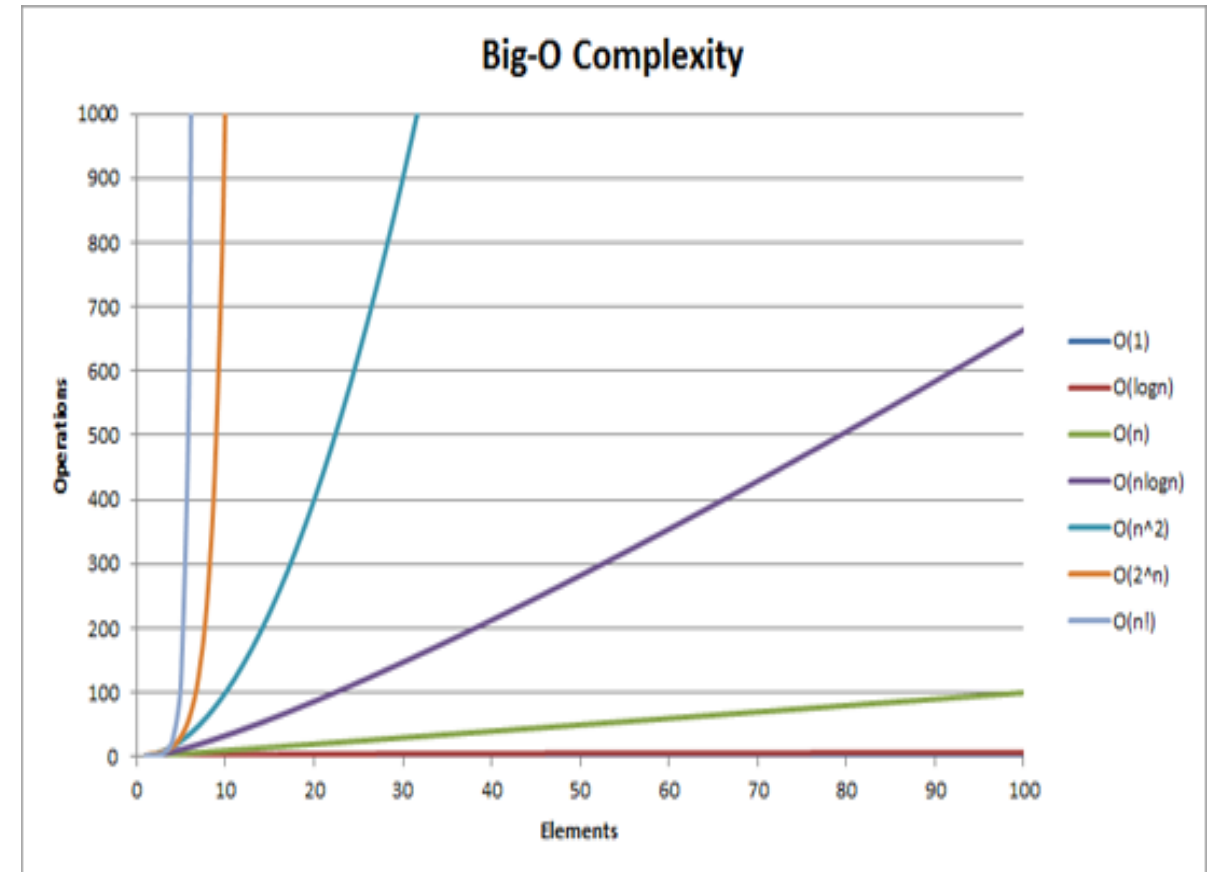
- $f(n) = 4n^2$
 - We cannot say $f(n)$ is $O(n)$?
 - Because there is no constant c such that $4n^2 < cn$ for any $n > n_0$.
- $f(n) = 50n^3 + 20n + 4$
 - We say that $f(n)$ is $O(n^3)$
 - It is also correct to say $f(n)$ is $O(n^3+n)$
 - However, this is not useful, as n^3 exceeds by far n , for large value
 - It is also correct to say $f(n)$ is $O(n^5)$
 - However, $g(n)$ should be as closed as possible to $f(n)$

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

constant time		log time		Polynomial time			Exponential time		Factorial time	
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!	

Tractable

Intractable (approximate)

Guidelines for finding Big-O values

- **for loops:**

- running time = running time of the statements inside the loop * number of iterations

```
for (i = 0; i < n; i++)  
    Primitive operations;
```

$O(n)$

- **Nested loops:**

- running time of the statements inside the loop * the product of the sizes of all the loops

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        primitive operations;
```

$O(n^2)$

Guidelines for finding Big-O values (continued)

- **Conditional statements:**

```
if condition then
    primitive statements with  $O(n)$  performance
else
    primitive statements with  $O(n^2)$  performance
```

$O(n)$ or $O(n^2) = O(n^2)$

In the worst case, this if statement has $O(n^2)$ performance.

- **Consecutive statements:** add running times

```
for (i = 0; i < n; i++)
    primitive statements;
for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
        primitive statements;
```

$O(n) + O(n^2) = O(n^2)$

Example 1 solutions: Revisited

Usually, there are more than one method to solve a given problem.

Solution 1

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

$O(n^2)$

Solution 2

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

$O(n)$

Analyzing nested loops (1)

- What about the following nested loops?

```
for i = 1..n loop
  for j = i..n loop
    primitive operations
```

- Analysis:
 - The primitive operations are performed $n + (n - 1) + \dots + 1$ times.
 - There is a well-known formula in mathematics that says $1 + 2 + \dots + n = \frac{n(n+1)}{2}$ or $\frac{1}{2}n^2 + \frac{1}{2}n$
 - Therefore performance is $O(n^2)$ even though the primitive operations are clearly executed fewer times.

Important big-O functions

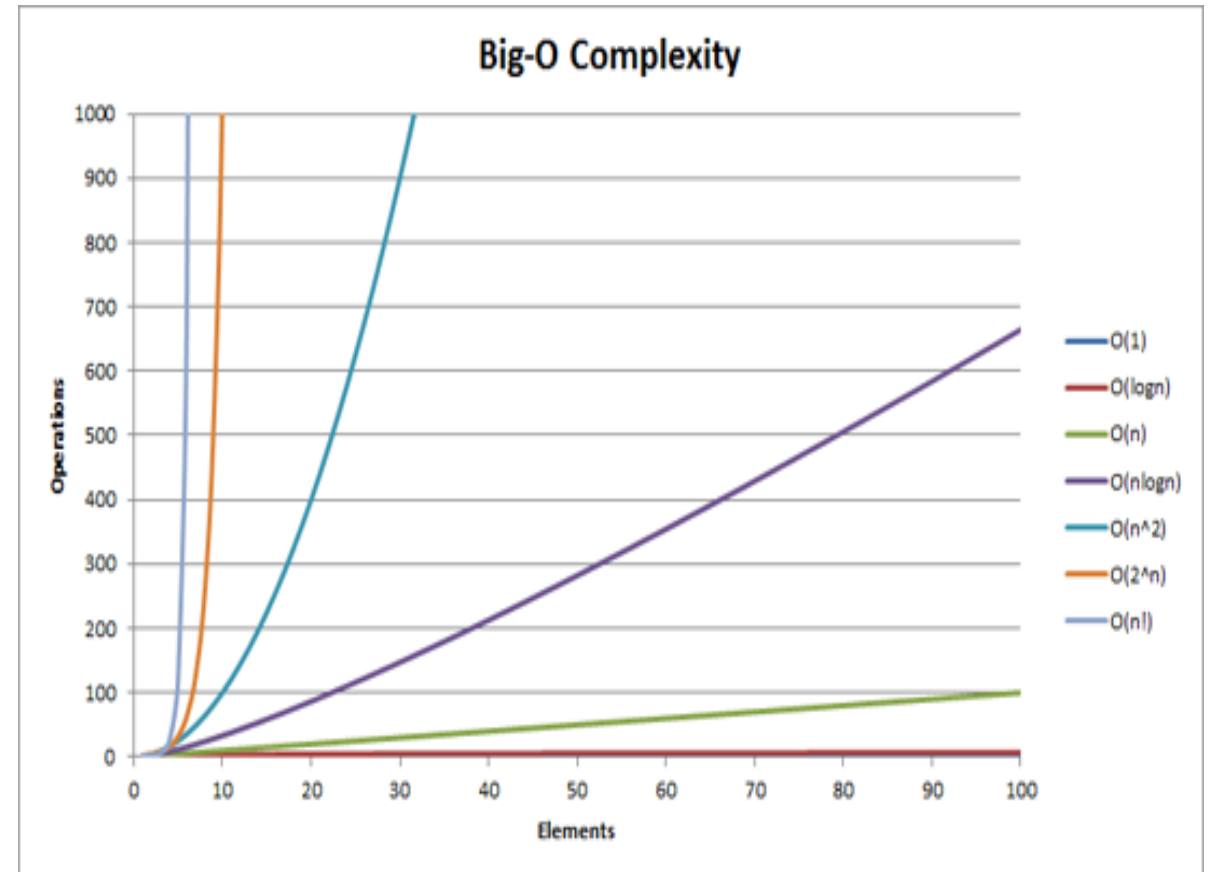
- $O(1)$
- $O(\lg N)$
- $O(N)$
- $O(N \lg N)$
- $O(N^2)$
- $O(2^N)$

Important 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



Important Functions:

The Constant Function – $O(1)$

- The Constant Function
 $f(n) = c$, where c is a constant.
- Examples
 - $f(n) = 1$
 - $f(n) = 1842$
 - $f(n) = 37$
 - $f(n) = 1,000,000$
- All constant functions have the same growth rate; i.e., none.
- In terms of big-O notation, we say simply that all constant functions are $O(1)$, meaning that they are $O(f(n))$, where $f(n) = 1$.

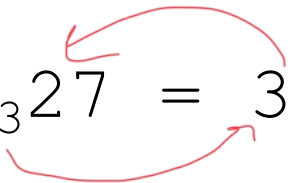
Important Functions:

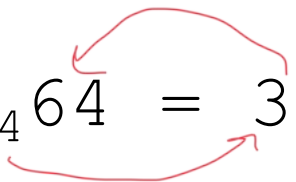
The Logarithm Function – $O(\log n)$

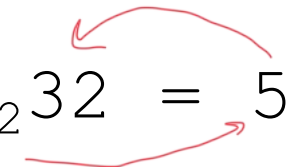
- The Logarithm Function $f(n) = \log_b n$ where b is a positive number.
 - The number b is called the base of the logarithm
- Examples
 - $f(n) = \log_{10} n$ (common logarithm, often written as $\log n$)
 - $f(n) = \log_e n$ (natural logarithm, often written as $\ln n$)
 - $f(n) = \log_2 n$ (most common in computer science)
- Recall that $\log_b n = \frac{\log_2 n}{\log_2 b}$. Since $\log_2 b$ is simply a constant, then all logarithm functions have the same growth rate as $\log_2 n$.
- For this course, $\log n$ (no subscript) will mean $\log_2 n$.

What does $\log_b n$ represent

- $\log_b n$ = How many times you divide n by b to reach 1.

- $\log_3 27 = 3$  (27/3 \rightarrow 9/3 \rightarrow 3/3 = 1) or ($3^3 = 27$)

- $\log_4 64 = 3$  (64/4 \rightarrow 16/4 \rightarrow 4/4 = 1) or ($4^3 = 64$)

- $\log_2 32 = 5$  (32/2 \rightarrow 16/2 \rightarrow 8/2 \rightarrow 4/2 \rightarrow 2/2 = 1) or ($2^5 = 32$)

A $\log n$ Loop

```
for(int i= n; i > 0; i/=2)
    primitive operations;
```

$i = 32, 16, 8, 4, 2, 1, 0$
 $O(\log_2 n)$

- Log = number of times can a number be divided by its base

Important Functions:

The Linear Function – $O(n)$

- The Linear Function
 $f(n) = an + b$, where $a \neq 0$.
- Examples
 - $f(n) = n$
 - $f(n) = 6n + 3$
- All linear functions have the same growth rate as
 $f(n) = n$
- In terms of big-O notation, we say simply that all linear functions are $O(n)$, meaning that they are $O(f(n))$, where $f(n) = n$.

A Linear Loop

```
for(int i= n; i > 0; i--)  
    primitive operations;
```

$O(n)$

Important Functions in Analysis of Algorithms

The N-Log-N Function – $O(n \log n)$

- The $n \log n$ Function
 $f(n) = n \log n$.
- This function arises naturally in the analysis of sorting algorithms (as we will cover later).
- The $n \log n$ function grows a little faster than a linear function ($O(n)$) but a lot more slowly than a quadratic function ($O(n^2)$).

Important Functions:

The Quadratic Function – $O(n^2)$

- The quadratic Function
 $f(n) = an^2 + bn + c$, where $a \neq 0$.
- Examples
 - $f(n) = n^2$
 - $f(n) = 18n^2 - 2n + 3$
- All quadratic functions have the same growth rate as
 $f(n) = n^2$
- In terms of big-O notation, we say simply that all quadratic functions are $O(n^2)$, meaning that they are $O(f(n))$, where $f(n) = n^2$.

A Nested Loop

```
for(int i= 0;i < n; i++)  
    for(int j=1; j < n; j++)  
        primitive operations;
```

Outer loop $O(n)$
Inner loop $O(n)$
Overall: $O(n^2)$

Special cases of nested loops

```
for(int i= 0; i < n; i+=2)
    for(int j=1; j < n; j++)
        primitive operations;
```

Outer loop $O(n/2)$
Inner loop $O(n)$
Overall: $O(n^2)$

```
for(int i= 1; i < n; i*=2)
    for(int j=1; j < n; j++)
        primitive operations;
```

Outer loop $O(\log_2 n)$
Inner loop $O(n)$
Overall: $O(n \log n)$

```
for(int i= 0; i < n; i++)
    for(int j=1; j < 3; j++)
        primitive operations;
```

Outer loop $O(n)$
Inner loop $O(1)$
Overall: $O(n)$

Important Functions:

Cubic and Other Polynomial Functions

- The Cubic Function
 $f(n) = an^3 + bn^2 + cn + d$, where $a \neq 0$.
- All cubic functions have the same growth rate as
 $f(n) = n^3$
- In terms of big-O notation, we say simply that all cubic functions are $O(n^3)$, meaning that they are $O(f(n))$, where $f(n) = n^3$.
- In general, a polynomial function is $O(n^d)$, where d is the degree of the highest power in the polynomial.

Important Functions:

The Exponential Function – $O(2^n)$

- The Exponential Function
 $f(n) = b^n$, where b is a positive constant.
 - The number b is called the base of the exponential function.
- Examples
 - $f(n) = 2^n$ (most common in computer science)
 - $f(n) = 10^n$
- Exponential functions with a positive base grow faster than any polynomial.
- Example: 1.1^n grows faster than $n^{1,000,000}$.

Exercise1

- Consider the function $g(n) = 100 + n^2 + 2^n$ which of the following is true:
 - $g(n) = O(1)$
 - $g(n) = O(n^2)$
 - $g(n) = O(2^n)$

2^n is the fastest growing term then $g(n)$ is $O(2^n)$

Exercise 2

- Consider the function $f(n) = 4\log_2 n + 3n\log_2 n + n$ which of the following is true:

- $f(n) = O(\log_2 n)$
- $f(n) = O(n\log_2 n)$
- $f(n) = O(n^2)$
- $f(n) = O(n)$

$$\log_2 n + \boxed{n \log_2 n} + n$$

After dropping constants, the terms that we need to consider are $\log_2 n$, $n\log_2 n$, and n . Out of these three terms the term $n\log_2 n$ is the fastest growing then $f(n) = O(n\log_2 n)$

Exercise 3

- How would read the following: $3n^2 + 4 = O(n^2)$:
 - Three n squared plus 4 is big-O of n-squared.
 - n-squared is big-O of three n-squared plus 4.

Three n squared plus 4 is big-O of n-squared.

Complexity analysis of Bag Operations: Arrays vs. Linked Lists

Why Do We Need Linked Lists?

- To overcome the disadvantages of using arrays:
 - **Fixed size:** to create an array you have to specify the size, however, a linked list can grow and shrink dynamically
 - **Adding at random positions:** adding an element to the front of an array (or in the middle) is very hard since a lot of elements need to be copied to other locations in order to make space for the new element to be inserted
 - However, for a linked list, an element can be added at any location by performing a few assignment statements to adjust the links.

Guidelines for Choosing Between an Array and a Linked List

Operation	Which data structure to use?
Frequent random access operations	Array
Frequent Insertion and deletion at random locations	Linked list (to avoid moving elements up and down)
Frequent capacity change	Linked list (to avoid the resizing inefficiency)

Complexity of Adding to an **Unordered Array-Based List**

STEPS

```
0      public void add(int element)  {  
1          data[manyItems] = element;  
1          manyItems++;  
          }
```

Total: 2 (constant) = $O(1)$

Complexity of Adding to an **Unordered Linked List**

STEPS

```
0      public void add(int element)    {  
1          this.head = new IntNode(element, head);  
1          this.manyNodes++;  
      }
```

Total: 2 (constant) = $O(1)$

Complexity of Inserting into an **Ordered Array-Based List**

STEPS

```
0      public void insert(int num)      {
1          int i=0;
2*n      while (i < manyItems && (data[i] <= num))      {
1*n          i++;
          }

1          for (int move = manyItems; move > i; move-- ) {
1              data[move] = data[move-1];
          }

1          data[i] = num;
1          manyItems++;
      }
```

Total: $1 + 3*n + 2 + 2 = 3*n + 5 = O(n)$

Complexity of Inserting into an **Ordered Linked List**

STEPS

```
0      public void insert (int newValue)      {
1          if (head == null)
1              head = new IntNode(newValue, head);
1          else if (newValue < head.getData())
1              head = new IntNode(newValue, head);
          else {
??              IntNode previousNode = findPrevious(newValue);
1              previousNode.addNodeAfrer(newValue);
          }
      }
```

Total: 2 OR 2 OR (?? + 1) = ??

findPrevious() Method for Ordered Lists

STEPS

```
0      public IntNode findPrevious(int newValue)  {
1          IntNode cur = head;
1*n      while (cur.getLink() != null &&
1*n          cur.getLink().getData() <
1*n          newValue) {
1*n          cur = cur.getLink();
          }
1      return cur;
      }
```

Total: $3*n + 2 = O(n)$

Complexity of Inserting into an **Ordered Linked List**

STEPS

```
0      public void insert (int newValue)      {
1          if (head == null)
1              head = new IntNode(newValue, head);
1          else if (newValue < head.getData())
1              head = new IntNode(newValue, head);
          else {
3*n+2      IntNode previousNode = findPrevious(newValue);
1          previousNode.addNodeAfrer(newValue);
          }
      }
```

Total: 2 OR 2 OR $(3*n + 2 + 1) = 3*n + 3 = O(n)$ (worst case)

Complexity of Adding or Inserting into a List

add()/insert()	Unordered List	Ordered List
Array-based list	$O(1)$	$O(n)$
Linked list	$O(1)$	$O(n)$

Complexity of Removing from a List

remove()	Unordered List	Ordered List
Array-based list	$O(n)$	$O(n)$
Linked list	$O(n)$	$O(n)$

Removing from an array is $O(n)$ because you need to search for the element to be removed. Once found, you can replace it with the last element in the array.

Time analysis for data structure operations

IntArrayBag

Operation	Time
add (without capacity increase)	$O(1)$
add (with capacity increase)	$O(n)$
countOccurrences	$O(n)$
remove	$O(n)$
getByIndex	$O(1)$
size	$O(1)$

IntLinkedBag

Operation	Time
add	$O(1)$
countOccurrences	$O(n)$
listSearch	$O(n)$
listPosition	$O(n)$