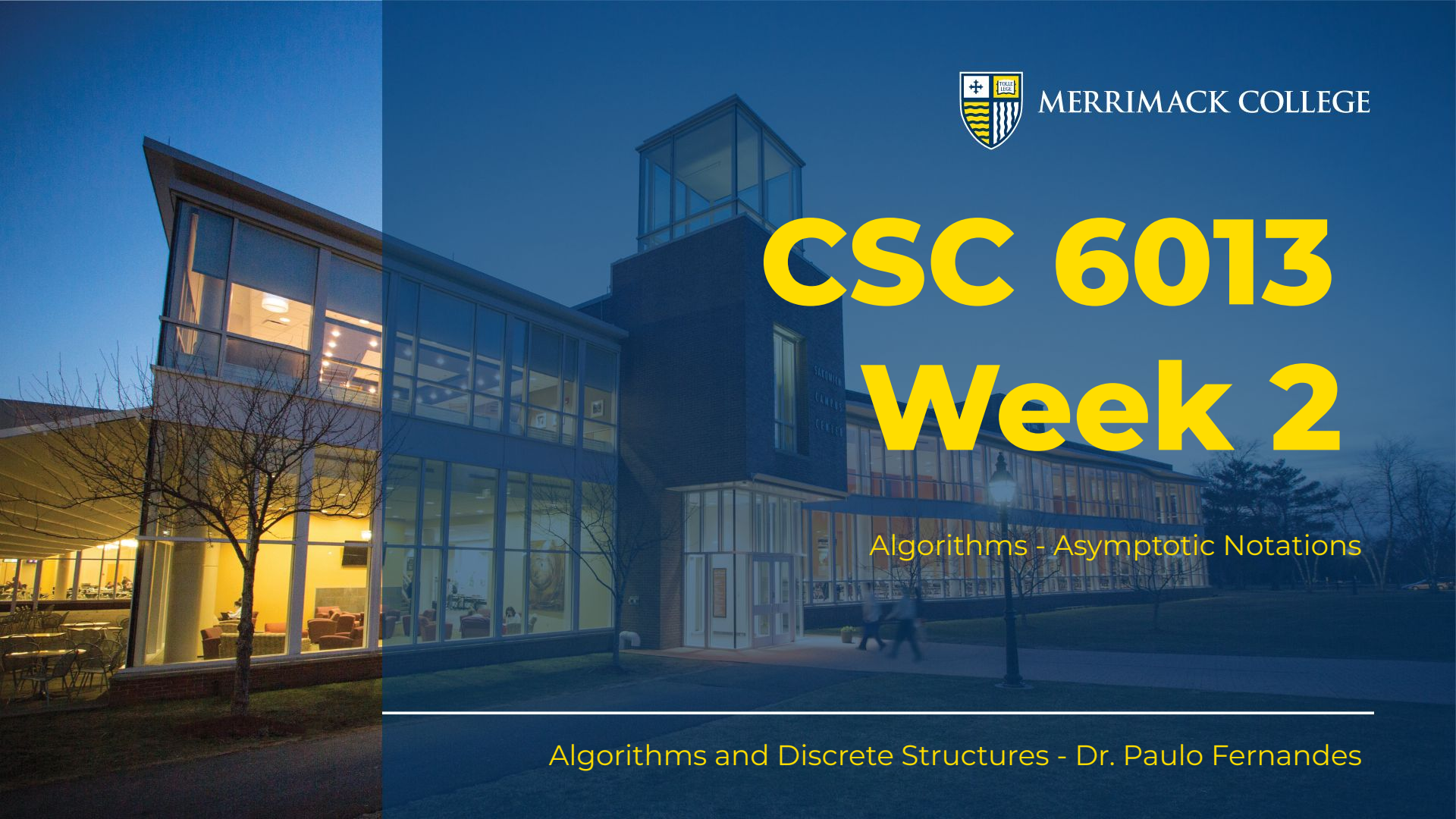


MERRIMACK COLLEGE

CSC 6013 Week 2

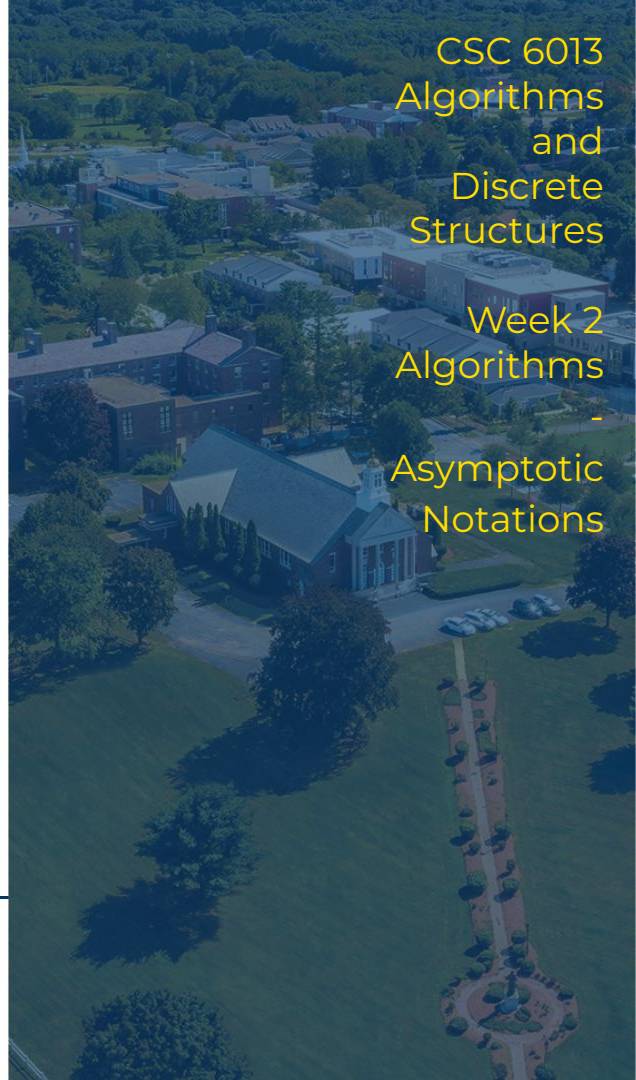
Algorithms - Asymptotic Notations

Algorithms and Discrete Structures - Dr. Paulo Fernandes

Presentation Agenda

Week 2

- Algorithms
  a. What is an algorithm
  b. Algorithm examples
  c. Summations and Logarithms
- Asymptotic Analysis
  a. Counting tasks
  b. Examples
  c. The notations: Big Oh, Omega, and Theta
- This Week's tasks



Algorithms

Effectiveness versus Efficiency

A program has to perform correctly what it is supposed to do. That is the effectiveness.

However, if doing what is supposed to do also requires the usage of limited resources, as time, memory, or even power consumption, this is a matter of efficiency, instead of effectiveness.

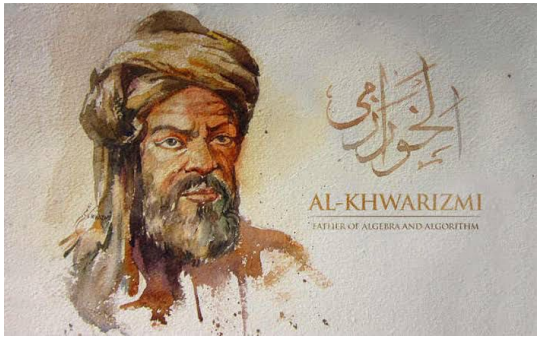
We will study the efficiency of algorithms, thus try to compute how much it takes to execute an algorithm.

- **What is an algorithm**
- Algorithm examples
- Summations and Logarithms

Algorithms - History

What is an algorithm?

- Informally, a strategy for solving a problem.
- Historically, a Latinized version of the name of Persian mathematician ***Abū Ja'far Muhammad ibn Mūsā al-Khwārizmī*** (800-867) who wrote the oldest known math book about using symbols to solve equations titled **al-Kitīb al-mukhtasar fī hisāb al-jabr wa'l muqābala** (The compendious book on calculation by completion and balancing).
- His work also named the latinized word Algebra from the arabic word ***al-jabr*** which means completion and is used as "add the same quantity to both sides of the equation".

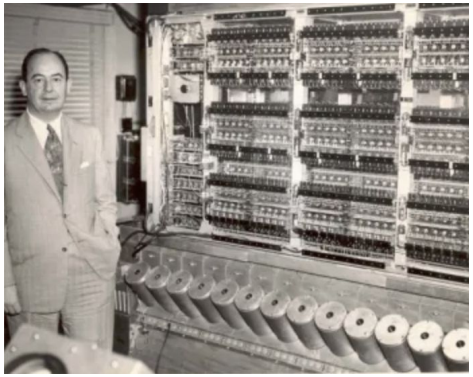


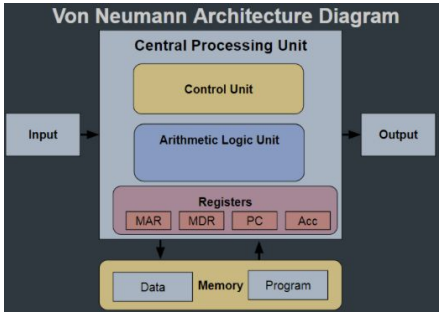
This definition of algorithm is correct, but too generic, thus, we need a more specific one.

Wikipedia: Al-Khwarizmi.

Algorithms - History

What is an algorithm?





- A more useful definition of algorithm in computer science is:
  - A well-defined computational procedure that takes some value, or set of values, as input and produces a value, or set of values, as output;
- More concisely:
  - A sequence of computational steps (instructions) that convert an input to an output;
- Much of it comes from the idea to store a program in the memory of a computer, a concept introduced by John Von Neumann.

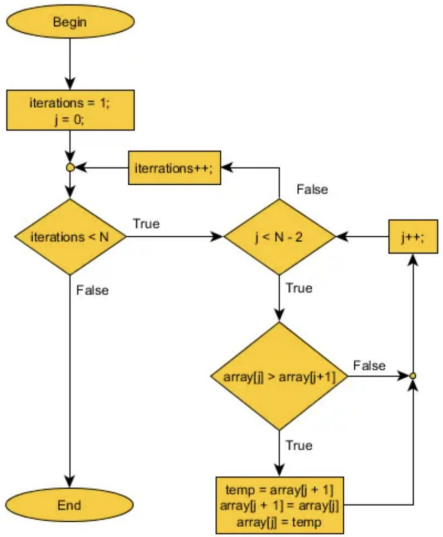
Wikipedia: Stored-program computer.

Algorithms - History

What is an algorithm?

```python
def bubbleSort(array):
    len_array = len(array)
    for j in range(0, len_array-1):
      flag = False
      for i in range(0, len_array-1):
        if array[i]>array[i+1]:
            array[i], array[i+1] = array[i+1], array[i]
            flag = True
      len_array = len_array - 1
      if not(flag):
        break
```

- Despite the association with a specific program, written in a specific language, the idea of algorithm is abstract, it is a way to handle data structures to achieve the desired output;
- As such, an algorithm can be defined independently of the programming language, as it concerns the operations that need to be made, not necessarily the commands of a specific language;
- Here we will use Python code, but what really matters is the algorithm beneath it.



Wikipedia: Algorithm.

Algorithms

Effectiveness versus Efficiency

A program has to perform correctly what it is supposed to do. That is the effectiveness.

However, if doing what is supposed to do also requires the usage of limited resources, as time, memory, or even power consumption, this is a matter of efficiency, instead of effectiveness.

We will study the efficiency of algorithms, thus try to compute how much it takes to execute an algorithm.

- What is an algorithm
- **Algorithm examples**
- Summations and Logarithms

Algorithms - Examples

Example 1 - Counting positive elements in an array

Given an array of ***n*** elements as input, compute the number of strictly positive numbers on it.

In this algorithm, we pass by all elements of the array and we check if they are positive, then, we count plus one.

The driver of this function generates a random array of 1000 values from -50 to 50.

```python
# Input: An array A with n Real numbers
# Output: Return the number of positive values in the array

def countPositiveElements(A):
    count = 0
    for x in A:
        if x>0:
            count += 1
    return count

from random import random
A = []
for _ in range(1000):
    A.append((random()-0.5)*100)
print("Positive elements:", countPositiveElements(A))
```

1 000 000 000

MERRIMACK COLLEGE

Run this code by yourself, then increase the size to 10^9.

Algorithms - Examples

Example 2 - Elements Uniqueness

Given an array of ***n*** elements as input, it returns ***True*** if all elements are unique (there are no two equal elements).

In this algorithm, we pass by all elements of the array and we check if there is one equal element after it.

The driver of this function generates a random array of 1000 values from 0 to 999,999.

```python
# Input: An array A if integers.
# Output: True is elements are unique; otherwise, False.

def uniqueElements(A):
    for i in range(0, len(A)-1):
        for j in range(i+1, len(A)):
            if A[i] == A[j]:
                return False
    return True

from random import randrange
A = []
for _ in range(1000):
    A.append(randrange(1000000))
if uniqueElements(A):
    print("All elements are unique")
else:
    print("There are repeated elements")
```

1 000 000 000

MERRIMACK COLLEGE

Run this code by yourself, then increase the size to 10^9.

Algorithms - Examples

Example 3 - Number of digits in a binary number

Given a decimal number ***n*** it returns how many binary digits it takes to represent it in binary notation.

In this algorithm, we keep dividing n by 2 (Integer division) until nothing remains, this is similar to compute the binary logarithm.

```
1   # Input: n, a non-negative integer.
2   # Output: Return the number of bits in the binary expansion of n.
3   def binaryDigitCount(n):
4       count = 1
5       while n>1:
6           count = count+1
7           n = n//2
8       return count
9
10  for n in [3, 23, 32, 345, 1023, 1024]:
11      print("The number of bits for {} is: {} ({})".format(n, \
12              binaryDigitCount(n), bin(n)[2:]))
```

1 073 741 824

```
The number of bits for 3 is: 2 (11)
The number of bits for 23 is: 5 (10111)
The number of bits for 32 is: 6 (100000)
```

```
The number of bits for 345 is: 9 (101011001)
The number of bits for 1023 is: 10 (1111111111)
The number of bits for 1024 is: 11 (10000000000)
```

The driver of this function also display the binary representation of some numbers.

Run this code by yourself, then include 1,073,741,824.

Algorithms

Effectiveness versus Efficiency

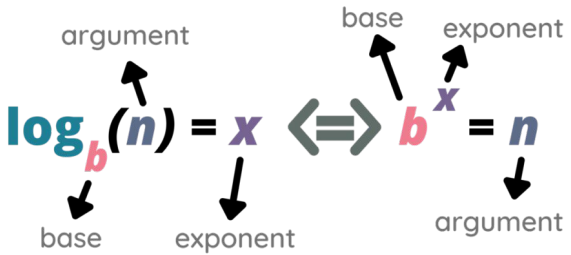
A program has to perform correctly what it is supposed to do. That is the effectiveness.

However, if doing what is supposed to do also requires the usage of limited resources, as time, memory, or even power consumption, this is a matter of efficiency, instead of effectiveness.

We will study the efficiency of algorithms, thus try to compute how much it takes to execute an algorithm.

- What is an algorithm
- Algorithm examples
- **Summations and Logarithms**

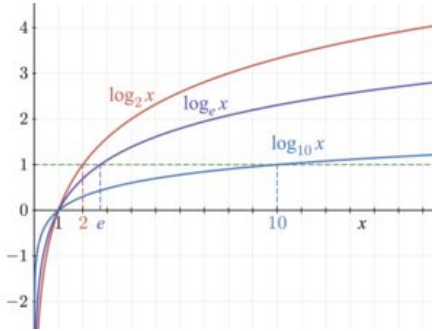
Algorithms - Logarithms



If a number ***b*** power a number ***x*** is equal to ***n***, then the logarithm of ***n*** base ***b*** is ***x***.

- $\log_{10}(1000) = 3$ (decimal or common log)
- $\log_2(1024) = 10$ (binary log)
- $\log_5(15625) = 6$ (logarithm base 5)

$$\log_b(b^x) = x$$



One can use this logarithm property to find logarithms of any base:

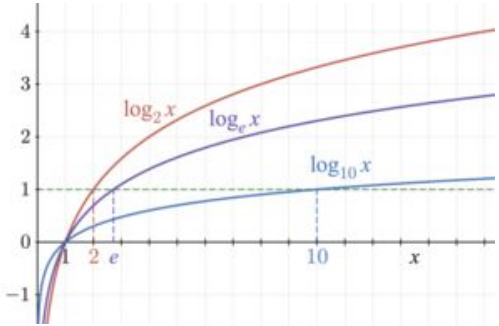
$$\log_b x = \frac{\log_k x}{\log_k b}$$

Numerically the Euler number ***e*** is comfortable to handle mathematical operations, thus the logarithm of base ***e*** is called the natural logarithm. However, in Computer Science the binary logarithm is far more useful.

Wikipedia: Logarithm.

Algorithms - Logarithms

In Python you can use ***math*** module which has the functions ***log(n)***, ***log2(n)***, and ***log10(n)*** to compute the natural, binary and decimal (common) logarithms.



The ***math*** module has other cool stuff about mathematics, like some irrational numbers constants as pi (**π**) and Euler number (***e***).

```
import math

print(math.pi)
print(math.e)

print()
print("common log of 1000:", math.log10(1000))
print("binary log of 1024:", math.log2(1024))
print("natural log of 100:", math.log(100))

a = math.log(15625) / math.log(5)
print("log base 5 of 15625:", a)
```

$$\log_b(b^x) = x$$

$$\log_b x = \frac{\log_k x}{\log_k b}$$

MERRIMACK COLLEGE

Run this code by yourself,
then compute some points of the curve above.

Algorithms - Summations

$$\sum_{\text{variable} \,=\, \text{initial}}^{\text{final}} \text{term}$$

Before continuing, let's set a important mathematical notation. The **summation** is defined by the Greek letter capital **sigma** with:

- one subscript which states the variable and its initial value;
- one superscript that states the final value; and
- the term expression.

A summation represents the sum of all term expressions with the variable going from the initial to the final value.

$$\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = 15$$

Wikipedia: Summation.

Algorithms - Summations

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

There are several ways to use a summation, and some useful properties.

$$\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{i=1}^{5} 2i = 2 + 4 + 6 + 8 + 10 = 30$$

In many ways, a summation is pretty much like a ***for*** command.

$$\sum_{i=0}^{5} 2^i = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 2^6 - 1 = 63$$

$$\sum_{i=0}^{99} 2i + 17$$

```
acc = 0
for i in range(100):
    acc += 2 * i + 17
print("The sum is:", acc)
```

$$\sum_{j=1}^{n} 5j^2 = 5\sum_{j=1}^{n} j^2 \qquad \sum_{i=1}^{n} 3 = 3 + 3 + 3 + \cdots + 3 = 3n$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Video: Summations and Sigma Notation.

Asymptotic Analysis

The effort for an algorithm applied to a problem of a given size.

An algorithm efficiency is measured in how much resources it needs when applied to a problem of a given size.

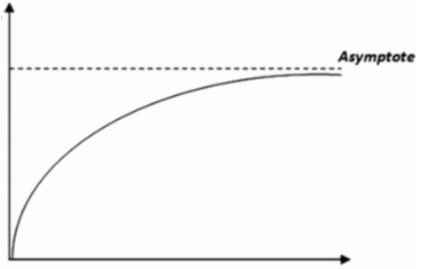
The resources can be the memory that will be necessary, or how much energy will be consumed, or even how much bandwidth it will require.

However, the more usual resource is time. We want to answer:

"how long will it take?"

- **Counting tasks**
- Examples
- The notations Big Oh, Big Omega, and Big Theta

Asymptotic Analysis - The name



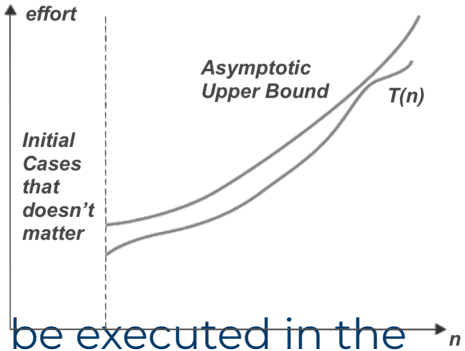
What Asymptotic means?

- According to the Cambridge dictionary:
  - An asymptotic line is a line that gets closer and closer to a curve as the distance gets closer to infinity.
- Practically speaking, we try to estimate how much it cost to run an algorithm;
  - We will focus now on the amount of time, thus the **amount of effort to execute a series of operations**, but similar analysis can be done for other measures.
- Our analysis will be asymptotic because we will try to approach the value of the effort curve as the problem grows, a.k.a., **limiting behavior**.

Wikipedia: Asymptotic analysis.

Asymptotic Analysis - The effort

The idea: count what matters when the problem is large



- Given an algorithm that receives an input of size n, we are looking for a function of ***n***, ***T(n)***, that describes the amount of work done by the algorithm.
- Usually, we count the number of fundamental steps to be executed in the worst case scenario (maximum over the valid inputs of size ***n***).
  - Thus, we are looking for an asymptotic upper bound, rather than an exact count of operations for cases when ***n*** is large enough.
- We want to estimate the effort needed as the algorithm is applied to an input as its size grows.

Text: Asymptotic Analysis.

Asymptotic Analysis

The effort for an algorithm applied to a problem of a given size.

An algorithm efficiency is measured in how much resources it needs when applied to a problem of a given size.

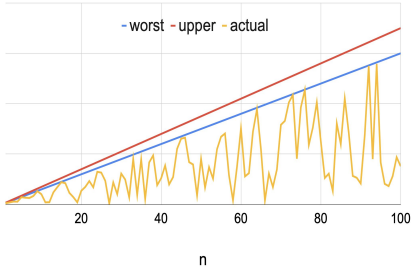
The resources can be the memory that will be necessary, or how much energy will be consumed, or even how much bandwidth it will require.

However, the more usual resource is time. We want to answer:

"how long will it take?"

- Counting tasks
- **Examples**
- The notations Big Oh, Big Omega, and Big Theta

Asymptotic Analysis



Example 1 - Counting positive elements in an array

- The problem size (***n***) is the size of the array ***A***;
- The worst case scenario is counting elements and they are all positive;
- Each line from 5 to 9 will have a cost (let's call them c_1 to c_5) and it will be execute a certain number of times;
- The total effort would be:
  - $T(n) = c_1 + n\,c_2 + n\,c_3 + n\,c_4 + c_5$
  - $T(n) = c_1 + c_5 + n\,(c_2 + c_3 + c_4)$
  - $T(n) = c_6 + n\,c_7$
  - $T(n) <= n\,c_6 + n\,c_7$
  - $T(n) <= n\,(c_6 + c_7)$
  - $T(n) <= n\,c_8$

```
4   def countPositiveElements(A):
5       count = 0
6       for x in A:
7           if x>0:
8               count += 1
9       return count
```

Line	Cost	Count
5	c_1	1
6	c_2	n
7	c_3	n
8	c_4	n (worst)
9	c_5	n

The cost ***T(n)*** is upper bounded by a function ***f(n)***.

As the problem size grows from ***n*** to ***2n***, so does the effort - ***T(2n) = 2 T(n)***.

Asymptotic Analysis

```
4   def uniqueElements(A):
5       for i in range(0, len(A)-1):
6           for j in range(i+1, len(A)):
7               if A[i] == A[j]:
8                   return False
9       return True
```

Example 2 - Elements Uniqueness

- The problem size (***n***) is the size of the array ***A***;
- The worst case scenario is without two equal values (both loops go through and line 8 is never executed);
- Each line from 5 to 9 will have a cost (c_1 to c_5) and it will be execute a certain number of times;
- The total effort would be:
  - $T(n) = (n-1)\, c_1 + ((n-1)n)/2\, c_2 + ((n-1)n)/2\, c_3 + 0\, c_4 + c_5$
  - $T(n) <= n\, c_1 + n^2 (c_2 + c_3) + c_5$
  - $T(n) <= n\, c_1 + n^2\, c_6 + c_5$
  - $T(n) <= n^2\, c_1 + n^2\, c_6 + n^2\, c_5$
  - $T(n) <= n^2 (c_1 + c_6 + c_5)$
  - $T(n) <= n^2\, c_7$

Line	Cost	Count
5	c_1	n-1
6	c_2	n-1 + n-2 + … + 1
7	c_3	n-1 + n-2 + … + 1
8	c_4	0
9	c_5	1

$$\frac{(n-1)n}{2}$$

The cost ***T(n)*** is upper bounded by a function $f(n^2)$.

As the problem size grows from ***n*** to ***2n***, so does the effort - $T(2n) = 2^2\, T(n)$.

Asymptotic Analysis

```
3   def binaryDigitCount(n):
4       count = 1
5       while n>1:
6           count = count+1
7           n = n//2
8       return count
```

Example 3 - Number of digits in a binary number

- The problem size (***n***) is the entered number ***n***;
- The worst case scenario is irrelevant as we always go until number ***n*** end up becoming 0 (no ***if*** command);
- Each line from 4 to 8 will have a cost (c_1 to c_5) and will be execute a certain number of times;
- The total effort would be:
  - $T(n) = c_1 + \log_2(n)\, c_2 + \log_2(n)\, c_3 + \log_2(n)\, c_4 + c_5$
  - $T(n) = c_1 + c_5 + \log_2(n)\,(c_2 + c_3 + c_4)$
  - $T(n) = c_6 + \log_2(n)\, c_7$
  - $T(n) <= \log_2(n)\, c_6 + \log_2(n)\, c_7$
  - $T(n) <= \log_2(n)\,(c_6 + c_7)$
  - $T(n) <= \log_2(n)\, c_8$

Line	Cost	Count
4	c_1	1
5	c_2	$\log_2(n)$
6	c_3	$\log_2(n)$
7	c_4	$\log_2(n)$
8	c_5	1

The cost ***T(n)*** is upper bounded by a function ***f(log$_2$ n)***.

As the problem size grows from ***n*** to ***2n***, so does the effort - ***T(2n) = log(2) + T(n)***.

Asymptotic Analysis

The effort for an algorithm applied to a problem of a given size.

An algorithm efficiency is measured in how much resources it needs when applied to a problem of a given size.

The resources can be the memory that will be necessary, or how much energy will be consumed, or even how much bandwidth it will require.

However, the more usual resource is time. We want to answer:

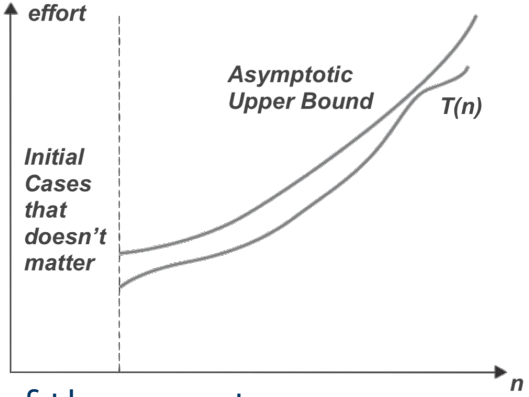
"how long will it take?"

- Counting tasks
- Examples
- **The notations Big Oh, Big Omega, and Big Theta**

Asymptotic Analysis - Notations

Complexity of Time

The more commonly found form of asymptotic analysis is the complexity of time, which is basically the upper bound to the worst case scenario as we consider before. This is denoted usually by the Big Oh notation:



- The **Big-Oh** is a function that is an upper bound of the worst-case of the algorithm:
  - It safely states the longest the algorithm will take;

Practically speaking, an algorithm is in ***O(log n)*** means that this algorithm will require at most ***T(n)*** operations and ***T(n) <= c log n*** for all values of ***n*** above a certain (small) value.

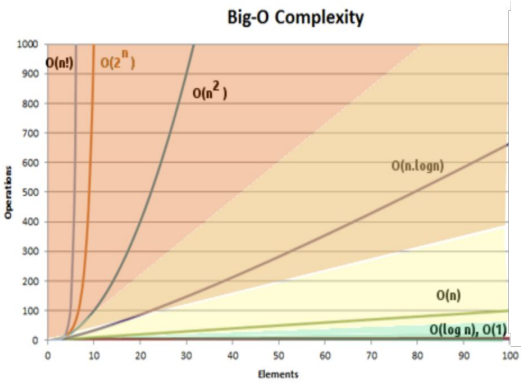
Wikipedia: Big O notation.

Asymptotic Analysis - Big Oh

Practically speaking, an algorithm is in ***O(log n)*** means that this algorithm will require at most ***T(n)*** operations and ***T(n) <= c log n*** for all values of ***n*** above a certain (small) value.

For the examples seen:

- Example 1 - Counting positive numbers in an array
  - Has a time complexity ***O(n)***;
- Example 2 - Elements Uniqueness
  - Has a time complexity ***O(n^2)***;
- Example 3 - Number of digits in a binary number
  - Has a time complexity ***O(log n)***;

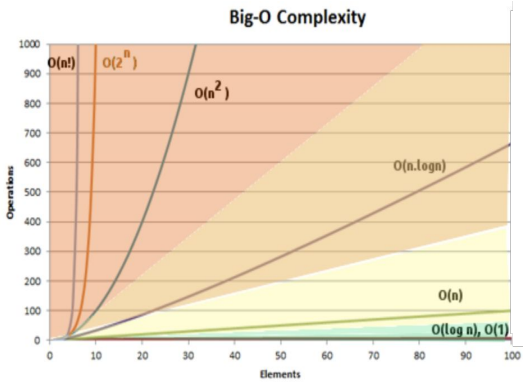


It means that Example 3 will scale better than Example 1, and Example 2 will scale worst than Example 1.

Video: Big-O Notation.

Asymptotic Analysis - Big Oh

The classes of complexity



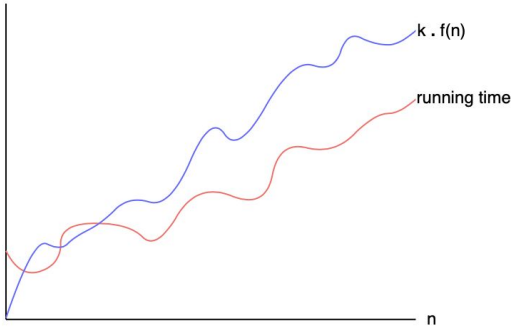
Big Oh is an upper bound... pessimistic (safe).

Big-Oh	complexity	examples
O(c)	constant	First element of an array
O(log n)	logarithmic	Binary search
O(n)	linear	Find the largest in an array
O(n log n)	log-linear	Merge sort
$O(n^c)$	polynomial	Selection sort (n^2 - quadratic)
$O(c^n)$	exponential	Find all subsets of a set (2^n)
O(n!)	factorial	Find all permutations of a set

Video: What is Big O Notation?.

Asymptotic Analysis - Big Oh

The slowest it can be - upper bound

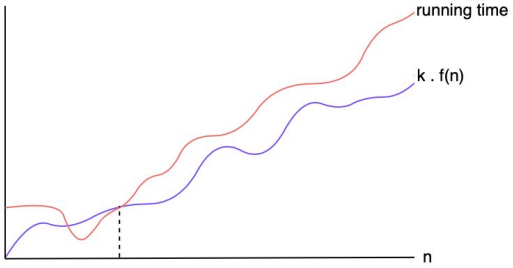


- The **Big-Oh** is a function that is an upper bound of the worst-case of the algorithm;
  - It safely states the longest the algorithm will take.
- Formally, if a function ***T(n)*** denotes the number of operations taken by a given algorithm, this function is said to be in ***O(g(n))*** if ***T(n)*** is bounded above by some constant multiple of ***g(n)*** for all large value of ***n***. In such way, we don't really care about the actual values of the constants c_i.
  - For example, an algorithm is in ***O(log n)*** means that this algorithm will require at most ***T(n)*** operations and ***T(n) <= c log n*** for all values of ***n*** above a certain (small) value.

Most of asymptotic analysis made is Big-Oh for time complexity.

Asymptotic Analysis - Big Omega

The fastest it can be - lower bound

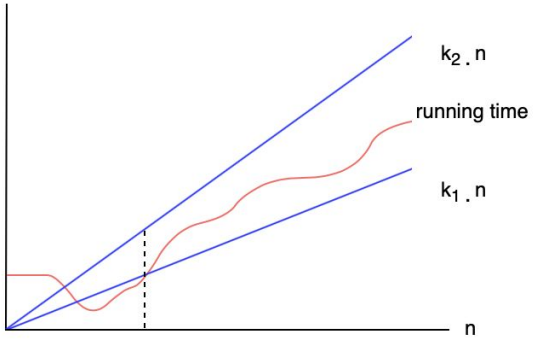


- The **Big-Omega** is a function that is a lower bound of the worst-case of the algorithm;
  - It safely states the shortest the algorithm will take.
- Formally, if a function ***T(n)*** denotes the number of operations taken by a given algorithm, this function is said to be in ***Ω(g(n))*** if ***T(n)*** is bounded below by some constant multiple of ***g(n)*** for all large ***n***.
  - For example, an algorithm is in ***Ω(log n)*** means that this algorithm will require at least ***T(n)*** operations and ***T(n) >= c log n*** for all values of ***n*** above a certain (small) value.

When we are not using
Big-Oh, Big Omega is employed.

Asymptotic Analysis - Big Theta

What to expect - upper and lower bound enveloppe



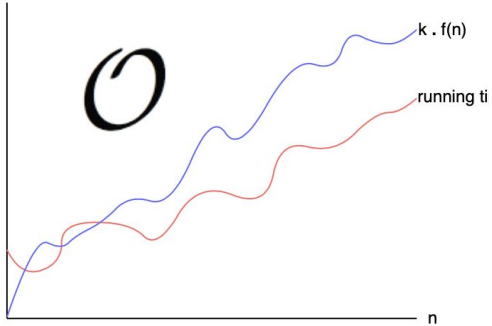
- The **Big-Theta** is a function that express both a lower and upper bound of the algorithm;
  - Big-Theta summarizes Big-Oh and Big-Omega.
- Formally, if a function ***T(n)*** denotes the number of operations taken by a given algorithm, this function is said to be in ***Θ(g(n))*** if ***T(n)*** is bounded above by some constant multiple of ***g(n)*** and bounded below by some constant multiple of ***g(n)*** for all large ***n***.
  - For example, an algorithm is in ***Θ(log n)*** means that this algorithm will require at least ***T(n)*** operations and ***T(n) >= c_1 log n*** and at most ***T(n)*** operations and ***T(n) <= c_2 log n*** for all values of ***n*** above a certain (small) value.

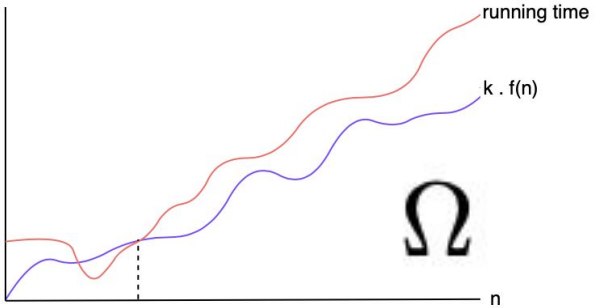
Sometimes there is no Big Theta.

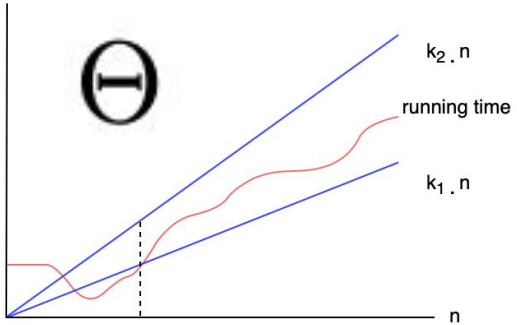
Asymptotic Analysis - The notations

Big Oh
Big Omega
Big Theta

- **Big Oh**
  - The slowest it can be - upper bound
- **Big Omega**
  - The fastest as it can be - lower bound
- **Big Theta**
  - What to expect - upper and lower bound enveloppe







MERRIMACK COLLEGE

Sometimes these bounds may be unknown, sometimes, too loose, sometimes, just right.

This Week's tasks

- In-class Exercise E#2
- Coding Project P#2
- Quiz Q#2

Tasks

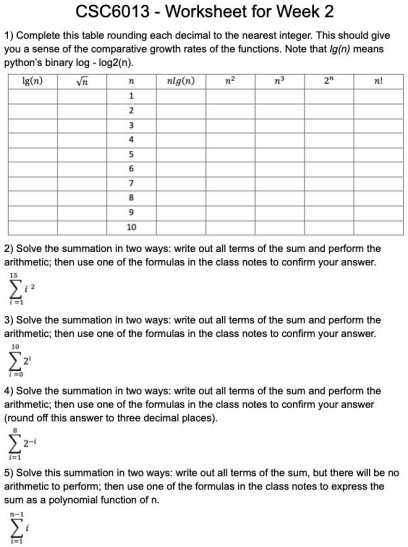
- Fill the worksheet as required.
- Create 3 Python programs:
  - count multiples;
  - find the smallest gap;
  - Multiply two matrices.
- Quiz #2 about this week topics.

In-class Exercise - E#2

Download the pdf (link here also) and perform the following tasks:

- (1) Fill the table computing numerically the values;
- (2 to 5) Solve the summations;
- (6 and 7) Compute the Big Oh value for the two presented algorithms.

You have to submit a ***.pdf*** file with your answers.



CSC6013 - Worksheet for Week 2

1) Complete this table rounding each decimal to the nearest integer. This should give you a sense of the comparative growth rates of the functions. Note that *lg(n)* means python's binary log - log2(n).

$\lg(n)$	$\sqrt{n}$	n	$n\lg(n)$	n^2	n^3	2^n	$n!$
		1					
		2					
		3					
		4					
		5					
		6					
		7					
		8					
		9					
		10					

2) Solve the summation in two ways: write out all terms of the sum and perform the arithmetic; then use one of the formulas in the class notes to confirm your answer.

$$\sum_{i=1}^{15} i^2$$

3) Solve the summation in two ways: write out all terms of the sum and perform the arithmetic; then use one of the formulas in the class notes to confirm your answer.

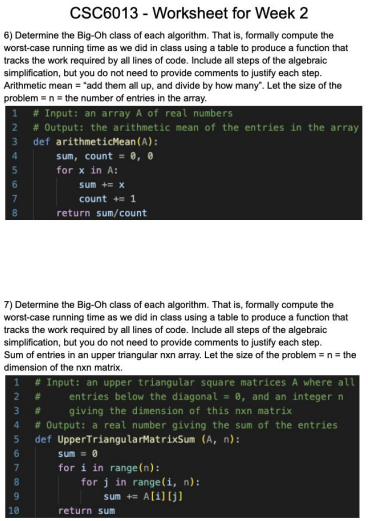
$$\sum_{i=0}^{10} 2^i$$

4) Solve the summation in two ways: write out all terms of the sum and perform the arithmetic; then use one of the formulas in the class notes to confirm your answer (round off this answer to three decimal places).

$$\sum_{i=1}^{8} 2^{-i}$$

5) Solve this summation in two ways: write out all terms of the sum, but there will be no arithmetic to perform; then use one of the formulas in the class notes to express the sum as a polynomial function of n.

$$\sum_{i=1}^{n-1} i$$



CSC6013 - Worksheet for Week 2

6) Determine the Big-Oh class of each algorithm. That is, formally compute the worst-case running time as we did in class using a table to produce a function that tracks the work required by all lines of code. Include all steps of the algebraic simplification, but you do not need to provide comments to justify each step.
Arithmetic mean = "add them all up, and divide by how many". Let the size of the problem = n = the number of entries in the array.

```
# Input: an array A of real numbers
# Output: the arithmetic mean of the entries in the array
def arithmeticMean(A):
    sum, count = 0, 0
    for x in A:
        sum += x
        count += 1
    return sum/count
```

7) Determine the Big-Oh class of each algorithm. That is, formally compute the worst-case running time as we did in class using a table to produce a function that tracks the work required by all lines of code. Include all steps of the algebraic simplification, but you do not need to provide comments to justify each step.
Sum of entries in an upper triangular nxn array. Let the size of the problem = n = the dimension of the nxn matrix.

```
# Input: an upper triangular square matrices A where all
#      entries below the diagonal = 0, and an integer n
#      giving the dimension of this nxn matrix
# Output: a real number giving the sum of the entries
def UpperTriangularMatrixSum (A, n):
    sum = 0
    for i in range(n):
        for j in range(i, n):
            sum += A[i][j]
    return sum
```

Feel free to type it or hand write it, but you have to submit a single ***.pdf*** with your answers.

MERRIMACK COLLEGE

This task counts towards the In-class Exercises grade and the deadline is This Friday.

Second Coding Project - P#2

1. Create a program that finds the number of entries in an array of Integers that are divisible by a given Integer. Your function should have two input parameters – an array of Integers and a positive Integer – and it should return an Integer indicating the count using a return statement.
2. Create a program that, given an array of Reals, without sorting the array, finds the smallest gap between all pairs of elements (for an array A, the absolute value of the difference between each pair of elements). Your function should have one input parameter – an array of Reals – and should return a number indicating the smallest gap using a return statement.
3. Create a program that, given an Integer n>1 and two nxn matrices A and B of Reals, find the product of the matrices. Your function should have three input parameters – an Integer n and two nxn matrices of numbers – and should return the nxn product matrix using a return statement.

More details in Canvas P#2 prompt.

Second Coding Project - P#2

For Program 1) you should execute the following test cases:

A. [20, 21, 25, 28, 33, 34, 35, 36, 41, 42] number of entries that are divisible by 7
B. [18, 54, 76, 81, 36, 48, 99] number of entries that are divisible by 9

For Program 2) you should execute the following test cases:

A. [50, 120, 250, 100, 20, 300, 200]
B. [12.4, 45.9, 8.1, 79.8, -13.64, 5.09]

For Program 3) you should execute the following test cases:

A. $n=2,\ A=\begin{pmatrix}2 & 7\\ 3 & 5\end{pmatrix},\ B=\begin{pmatrix}8 & -4\\ 6 & 6\end{pmatrix}$

B. $n=3,\ A=\begin{pmatrix}1 & 0 & 2\\ 3 & -2 & 5\\ 6 & 2 & -3\end{pmatrix},\ B=\begin{pmatrix}.3 & .25 & .1\\ .4 & .8 & 0\\ -.5 & .75 & .6\end{pmatrix}$

To all programs you have to submit the code (.py file) and a .pdf with the test cases output.

Second Coding Project - P#2

If you are not familiar with matrix multiplication, you might the following internet resources helpful.

The definition of matrix multiplication from Wikipedia:
https://en.wikipedia.org/wiki/Matrix_multiplication
A simpler definition of matrix multiplication:
https://www.mathsisfun.com/algebra/matrix-multiplying.html
Two videos of matrix multiplication:
Video #1: https://youtu.be/sYlOjyPyX3g
Video #2: https://youtu.be/n8ICyS8CKIQ

Your task:

- Go to Canvas, and submit your .py files and .pdf files within the deadline.

This assignment counts towards the Projects grade and the deadline is Next Monday.

Second Quiz - Q#2

- The second quiz in this course covers the topics of Week 2;
- The quiz will be available this Friday, and it is composed by 10 questions;
- The quiz should be taken on Canvas (Module 2), and it is not a timed quiz:
  - You can take as long as you want to answer it (a quiz taken in less than one hour is usually a too short time);
- The quiz is open book, open notes, and you can even use any language Interpreter to answer it;
- Yet, the quiz is evaluated and you are allowed to submit it only once.

Your task:

- Go to Canvas, answer the quiz and submit it within the deadline.

This quiz counts towards the Quizzes grade and the deadline is Next Tuesday.

Welcome to CSC 6013

- Do In-class Exercise E#2 until Friday;
- Do Quiz Q#2 (available Friday) until next Monday;
- Do Coding Project P#2 until next Monday.

Next Week - Brute Force Algorithms

MERRIMACK COLLEGE