CS5800: Algorithms

Week 1 - Introduction to Algorithms & Growth of Functions

Dr. Ryan Rad

Fall 2024

Northeastern University

Welcome to the course!

Khoury School of Computer Science
was the first dedicated CS school in the US (since 1980)

From my last visit - Late Aug 2023

Introduction to Algorithm

- Course Structure
- What is an Algorithm & Computational Problem
- Designing Algorithm
- Algorithm Efficiency
- Asymptotic notation
- Problems with $O(n^2)$
- Efficiency of Selection & Insertion sorts
- Review of Inductive Proof

About me

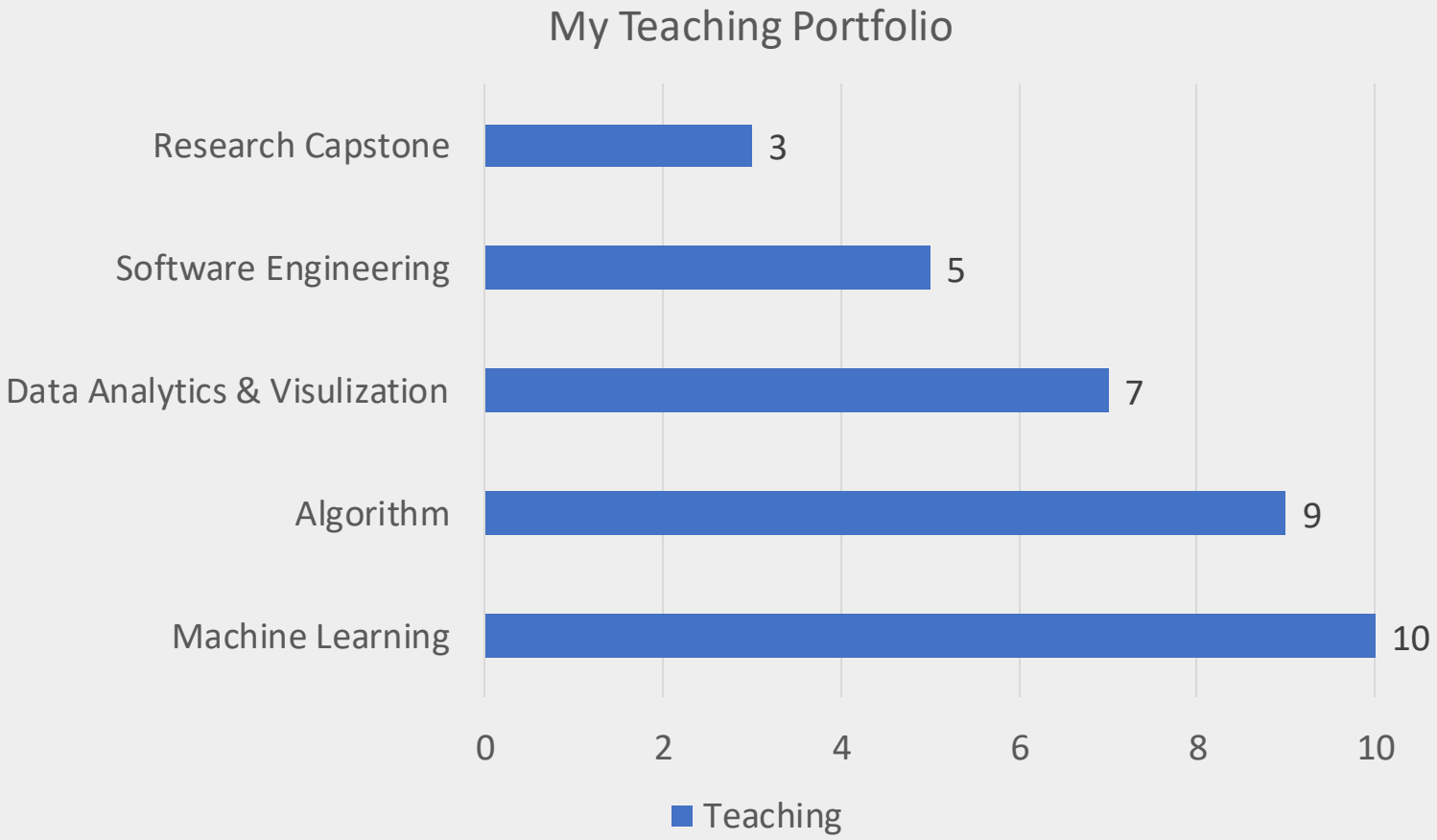
Ryan Rad, PhD

Experience

- **Industry Experience:**
- **Academic Experience:**

- **Research Interests:** Machine Learning, Deep Learning, Big Data Analytics, Health Informatics, Medical Imaging, Computer Vision
- **Entrepreneurship:** Founded 3 companies (2 failed, 1 profitable)
- **Current Role:**
  - Assistant Professor Northeastern University

Connect with me on


My Teaching Portfolio
Research Capstone
3
Software Engineering
5
Data Analytics & Visulization
7
Algorithm
9
Machine Learning
10
0
2
4
6
8
10
Teaching

Your (wonderful) TAs

Zuoyang (Godot) Xu
(He/Him)

Ji (Eric) Chen
(He/Him)

Yichen (Eric) Wang
(He/Him)

Zuoyang (Godot) Xu

Ji (Eric) Chen

Yichen (Eric) Wang

Learning Objectives for this course:

By the end of this course you will:

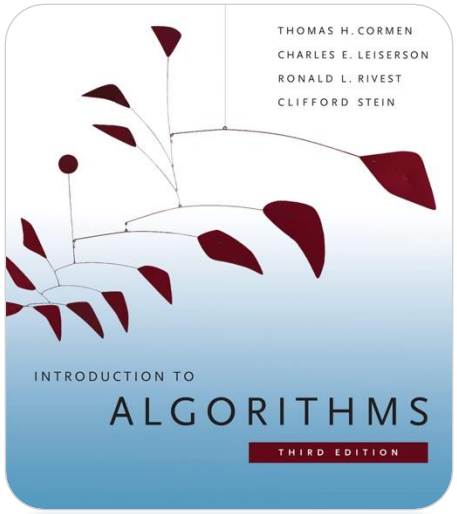
- Exhibit proficiency in the design, implementation, and testing of algorithms
- Demonstrate skills and experience working in small teams and communicating your ideas and solutions.
- Apply algorithmic and theoretical computer-science principles to solve computing problems from a variety of application areas.
- Demonstrate the ability to learn and develop competencies in specialized or emerging computer science fields.

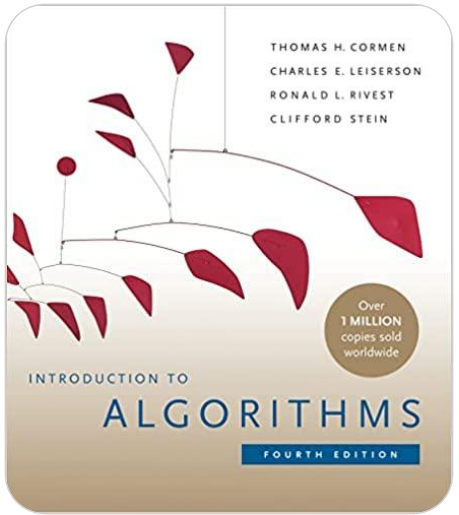
Course Resources:

[More Free access material will be added]

Introduction to Algorithms, Fourth Edition

By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein



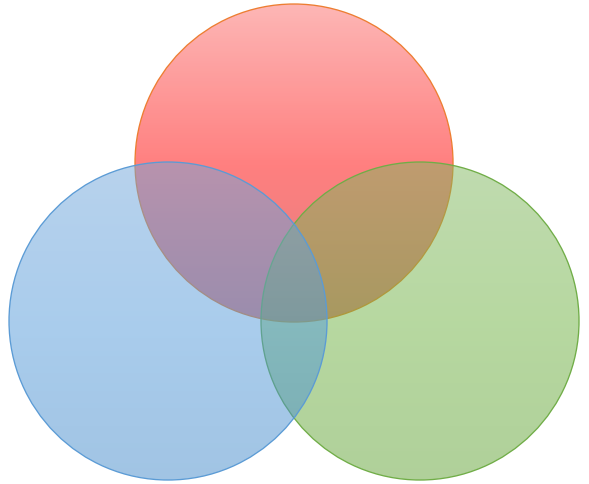


Free Access via Northeastern Library:

https://onesearch.library.northeastern.edu/discovery/fulldisplay?vid=01NEU_INST:NU&search_scope=MyInst_and_CI&tab=Everything&docid=alma9952146249801401&lang=en&context=L&adaptor=Local%20Search%20Engine&query=any,contains,Introduction%20to%20Algorithms,%20Thomas%20H.%20Cormen&offset=0&virtualBrowse=true

Course Mechanics

- Our goal is to provide a productive learning environment
  - Learning as much as possible
  - Having fun
  - What else?



- 100% attendance is expected (missing 1-2 sessions would be fine!)
- Classes will include lectures, in-class activities, and seminars
- Be sure to bring your laptop
- There will be a mandatory reading and quiz each week.
- Weekly class plan & slides will be posted in advance

Communication

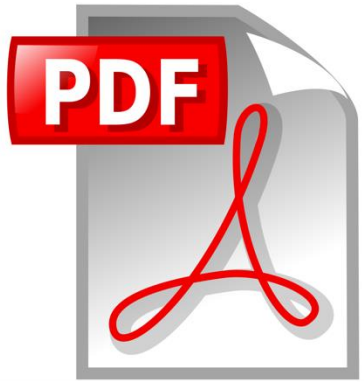
- Canvas
  - For most updated schedule, information, announcements, and materials.
  - For submission, feedback, (re-)grading, etc.

- Piazza (see Canvas for link)
  - For any general course-related discussion and questions about assignments, etc.

- For getting ahold of me / personal or confidential matter.
  - Email or Piazza (Private, direct message) - Canvas Inbox is fine too
  - But NOT MS Teams or any other platform

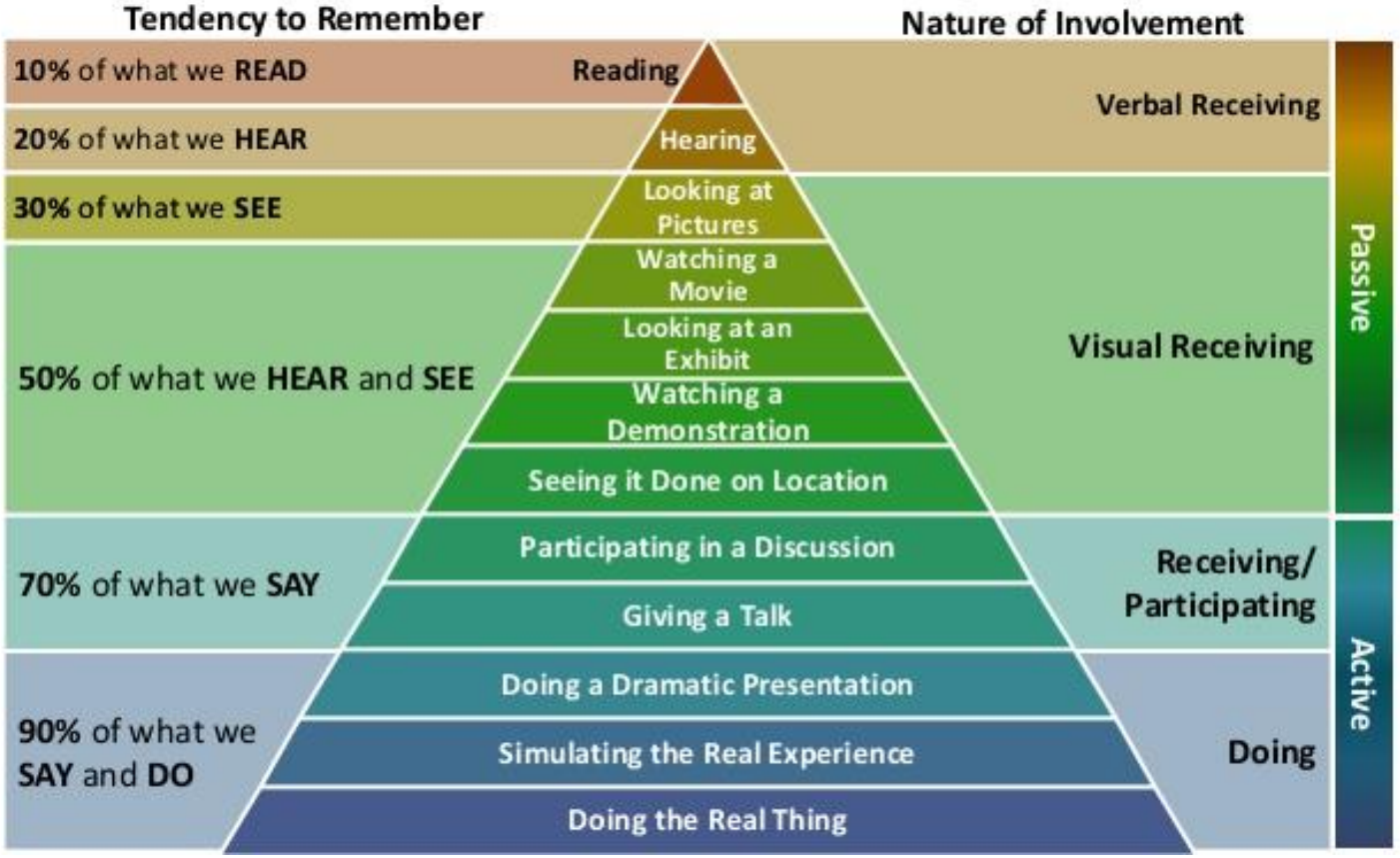
Course Syllabus

Let’s take a careful look at Course syllabus on Canvas:



Dale Edgar's Cone of Learning

"I see and I forget.
I hear and I remember.
I do and I understand."
– *Confucius*



LEARNING TIMELINE

Class Prep

0%-20%

In Class

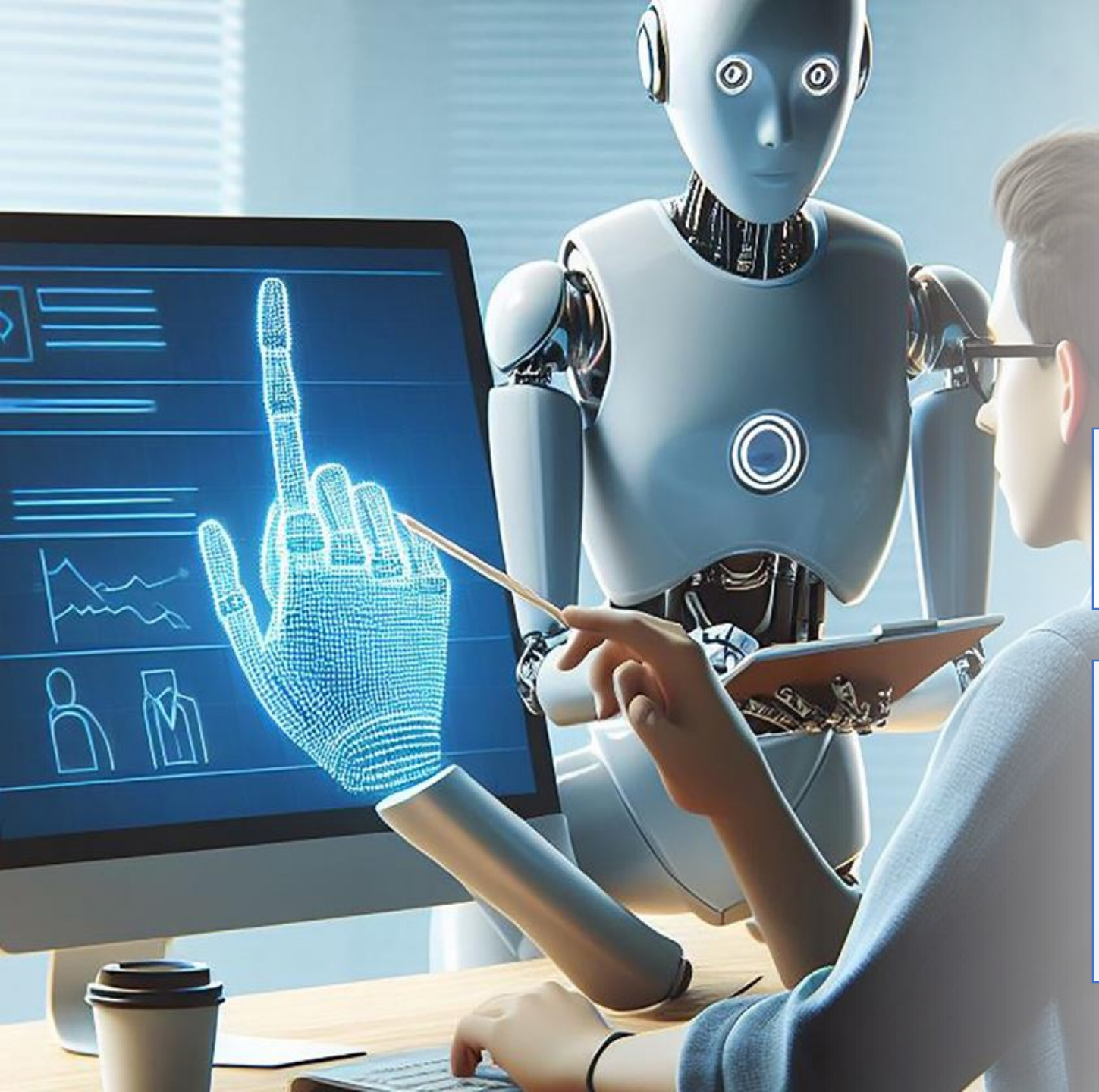
20%-70%

Homework

70%-85%

Midterm/Project

85%-100%



AI Policy

For the project:

- You are highly **encouraged** to use AI. It will be difficult to complete the project without consulting AI tools, and I expect you to do so.

For assignments and online quizzes:

- You are highly **discouraged** from using AI. While I recognize that I have no control over this, the assignments are designed in a way that it's beneficial to do them in an old-school way to practice and understand the concepts better. This comes as sincere advice, as I believe it will aid your learning experience.

Weekly Schedule

Week	Date	Topics	Book	Quiz	Notes
1	Sep 10	• Introduction to the Course • Review Fundamentals	Chapters 2-3		*Bring a smile & tons of energy!* *HW1 released*
2	Sep 17	• Divide-and-Conquer	Chapters 4	Quiz 1	*HW 1 due Jan 20*
3	Sep 24	• Linear Sorting and Selection	Chapter 8, 9	Quiz 2	*HW 2 released*
4	Oct 01	• Dynamic Programming	Chapters 14	Quiz 3	*HW 2 due Oct 4* *HW 3 released*
5	Oct 08	• Greedy Algorithms	Chapter 15	Quiz 4	
6	Oct 15	• Graphs Algorithms	Chapters 20-21	Quiz 5	*HW 3 due Oct 18* *Mid-course feedback due Oct 18* *HW 4 released*
7	Oct 22	• Linear Programming • Online Algorithms	Chapters 27 & 29	Quiz 6 (on paper)	Class Activity #1
8	Oct 29	• Problem Solving • Midterm Practice			*HW 4 due Oct 28*
9	Nov 5	• Midterm Exam	Weeks 1-7		
10	Nov 12	• Machine Learning (Supervised)	Chapter 33	Quiz 7 (on paper)	Project Proposal due Oct 15
11	Nov 19	• Machine Learning (Unsupervised)	Chapter 33	Quiz 8 (on paper)	
12	Nov 26	• NP-Completeness		No Quiz!	Class Activity #2
13	Dec 03	• Project Presentation			Project Milestone #2 due Dec 3
14	Dec 10	• Final Project Deliverables			Project Milestone #3 due Dec 10

What is Algorithm?

What is Algorithm?

- **Algorithms is all about:**
  - Finding good step-by-step procedures to solve computational problems.
  - Wait…! What are “Computational Problems” ?

What is Computational Problems?

- In a computational problem, we are given an input that and we want to return as output a solution satisfying some property.

Examples of Computational Problems:

Decision problems: the answer for every instance is either yes or no.

- Example: given a positive integer n, determine if n is prime.

Search problems: the answers can be arbitrary strings

- Example: factoring is a search problem where the instances are (string representations of) positive integers and the solutions are (string representations of) collections of primes.

Optimization problems: asks for finding a "best possible" solution among the set of all possible solutions to a search problem.

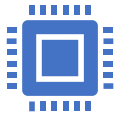
Counting problems: asks for the number of solutions to a given search problem.

- Example: Given a positive integer n, count the number of nontrivial prime factors of n.

Function problems: a single output is expected for every input, but the output is more complex

- Example: "Given a list of cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the origin city."

Types of Algorithms (1/2)



Brute Force Algorithm:

It is the simplest approach for a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

Recursive Algorithm:

A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

Divide and Conquer Algorithm:

This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps: Divide, Solve, and Combine

Searching Algorithm:

Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

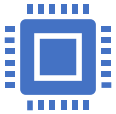
Sorting Algorithm:

Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

Types of Algorithms (2/2)

Hashing Algorithm:

Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.



Backtracking Algorithm:

The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

Greedy Algorithm:

In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part.

Dynamic Programming Algorithm:

This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

Randomized Algorithm:

In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

How to Design an Algorithm?

1. The **problem** that is to be solved by this algorithm i.e. clear problem definition.
2. The **constraints** of the problem must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem is solved.
5. The **solution** to this problem, is within the given constraints.

What is Algorithm?

An algorithm is a finite procedure, governed by precise instructions and discrete steps, whose execution requires no insight or intelligence.

Characteristics of an Algorithm

The goal is to create algorithms that achieve the following objectives:

Accuracy:

- The output is always correct

Efficiency:

- The output returns quickly

Simplicity:

- The program is clean, easy to understand, and easy to debug.

Why Efficient Code?

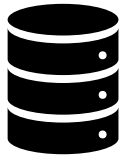
- Computers are faster, have larger memories
  - So why worry about efficient code?
- And … how do we measure efficiency?

What is “BEST”?

The two main criteria to judge which sorting algorithm is better:

- Time taken to sort the given data.

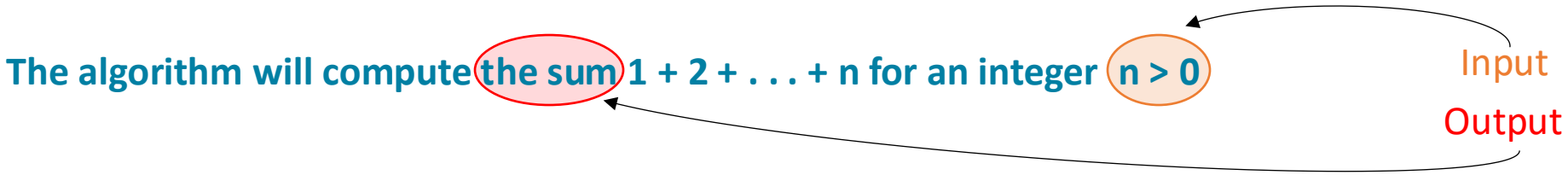
- Memory Space required to do so.



This study is called analysis of algorithms

Importance of Efficiency

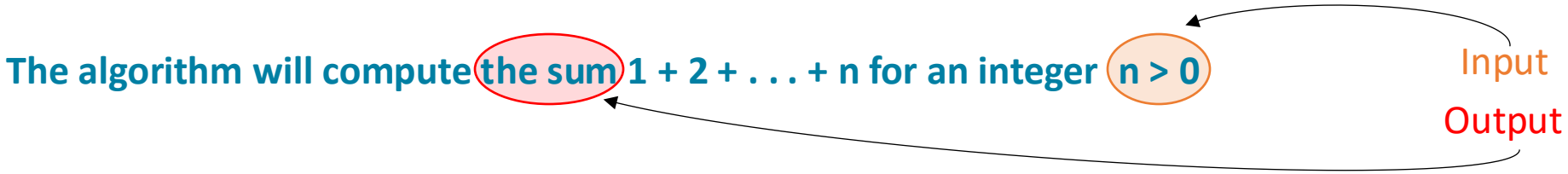
- Consider the problem of summing $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n$



Algorithm A	Algorithm B	Algorithm C

Importance of Efficiency

- Consider the problem of summing $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n$



Algorithm A	Algorithm B	Algorithm C

long sum = 0;
for (long i = 1; i <= n; i++)
  sum = sum + i;
	sum = 0;
	for (long i = 1; i <= n; i++)
	{
	  for (long j = 1; j <= i; j++)
	  sum = sum + 1;
	} // end for
		sum = n * (n + 1) / 2;

Counting Total Operations

Loop Logic:

```
i=1
while (i <= n) {
        ...
i=i+1 }
```

Algorithm A

```
long sum = 0;
for (long i = 1; i <= n; i++)
  sum = sum + i;
```

This logic requires an assignment to *i*, n+1 comparisons between *i* and *n*, *n* additions to *i*, and *n* more assignments to *i*.

- In total, the loop-control logic requires **n+1 assignments**, **n+1 comparisons**, and ***n* additions**.
- Furthermore, Algorithm A requires for its initialization and loop body another **n+1 assignments** and ***n* additions**.
- All together, Algorithm A requires **2n+2 assignments**, **2*n* additions**, and **n+1 comparisons**.

These various operations probably take different amounts of time to execute. For example, if each assignment takes no more than t_1 time units, each addition takes no more than t_+ time units, and each comparison takes no more than t_c time units, Algorithm A would require no more than $(2n+2)\,t_1 + (2n)\,t_2 + (n+1)\,t_3$ time units

If we replace t_1, t_2, and t_3 with the largest of the three values and call it t, Algorithm A requires no more than $(5n+3)\,t$ time units. We conclude that Algorithm A requires time directly proportional to **5n+3**.

5n+3: n=100 needs 503t
5n+3: n=200 needs 1003t

n^2+n+3: n=100 needs (10000 + 100 + 3) t
n^2: n=100 needs (10000 + 0 + 0) t

Counting Basic/Elementary Operations

- An algorithm's basic operation is the **most significant contributor** to its total time requirement.

- **Ignoring** operations that are not basic, such as initializations of variables, the operations that control loops and so on will not affect our final conclusion about algorithm speed. For example, Algorithm A requires *n* additions of *i* to sum in the body of the loop. We can conclude that Algorithm A requires time that increases linearly with *n*, even though we ignored operations that are not basic to the algorithm.

- Whether we look at the number, *n*, of basic operations or the total number of operations, 5n+3, we can draw the same conclusion: Algorithm A requires time directly proportional to *n*. Thus, Algorithm A's growth-rate function is *n*.

Team Canada show off their 4x100-metre bronze medals at the 2016 Summer Olympics Brazil on Saturday, 2016.

In a relay race (e.g., 4x800 meters), the slowest runner on a team can significantly influence the outcome.

Counting Basic/Elementary Operations

- A basic operation of an algorithm
  - Most significant contributor to its total time requirement

	Algorithm A	Algorithm B	Algorithm C

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

Additions			
Multiplications			
Divisions			
Total Basic Operations			

The number of basic operations required by the algorithms

Counting Basic/Elementary Operations

- A basic operation of an algorithm
  - Most significant contributor to its total time requirement

	Algorithm A	Algorithm B	Algorithm C

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

Additions	**n**	$n(n+1)/2$	**1**
Multiplications	**0**	**0**	**1**
Divisions	**0**	**0**	**1**
Total Basic Operations	**n**	$(n^2+n)/2$	**3**

The number of basic operations required by the algorithms

Picturing Efficiency

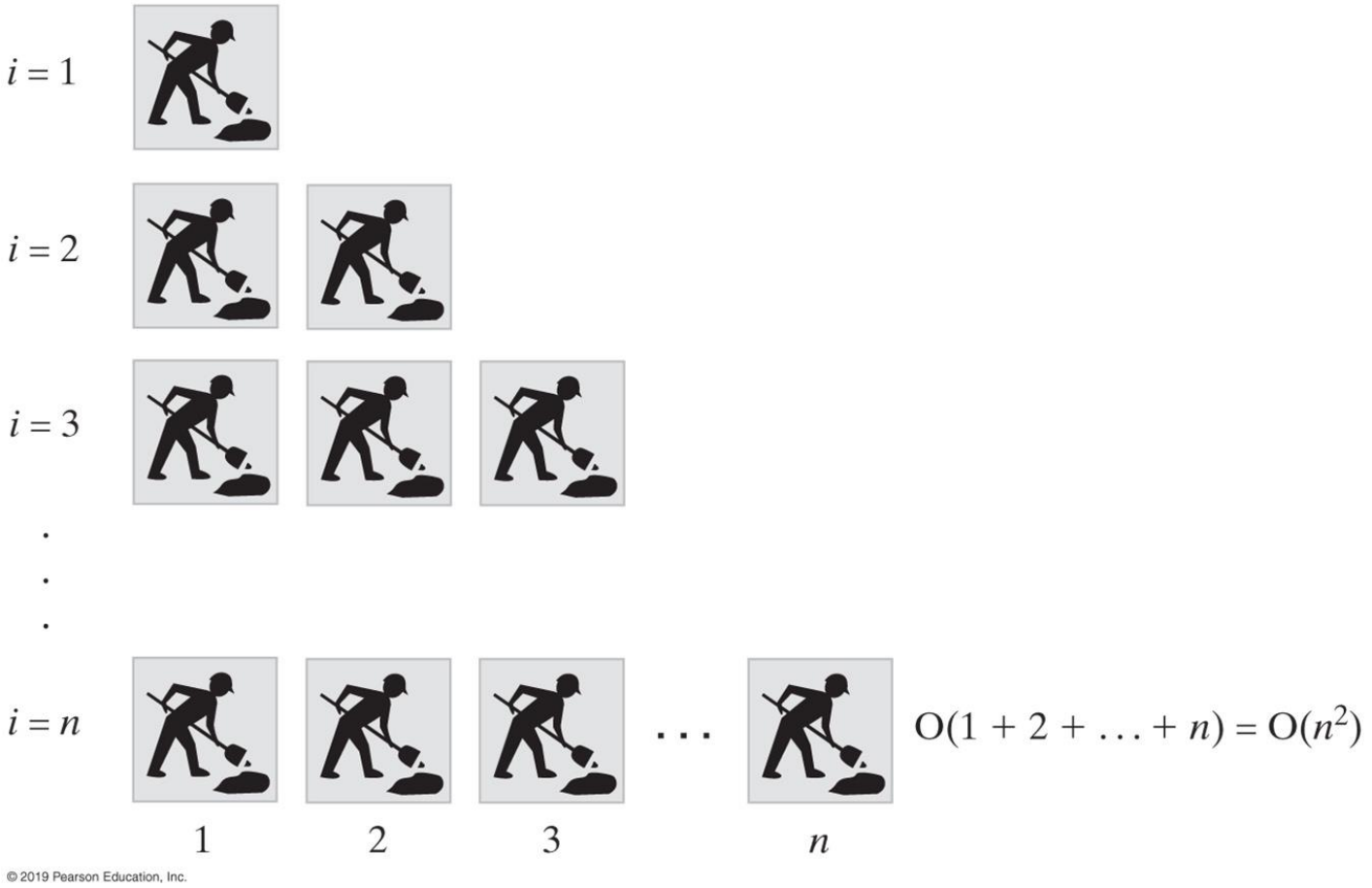
An O(n) algorithm

```
long sum = 0;
for (long i = 1; i <= n; i++)
  sum = sum + i;
```

Picturing Efficiency

An $O(n^2)$ algorithm

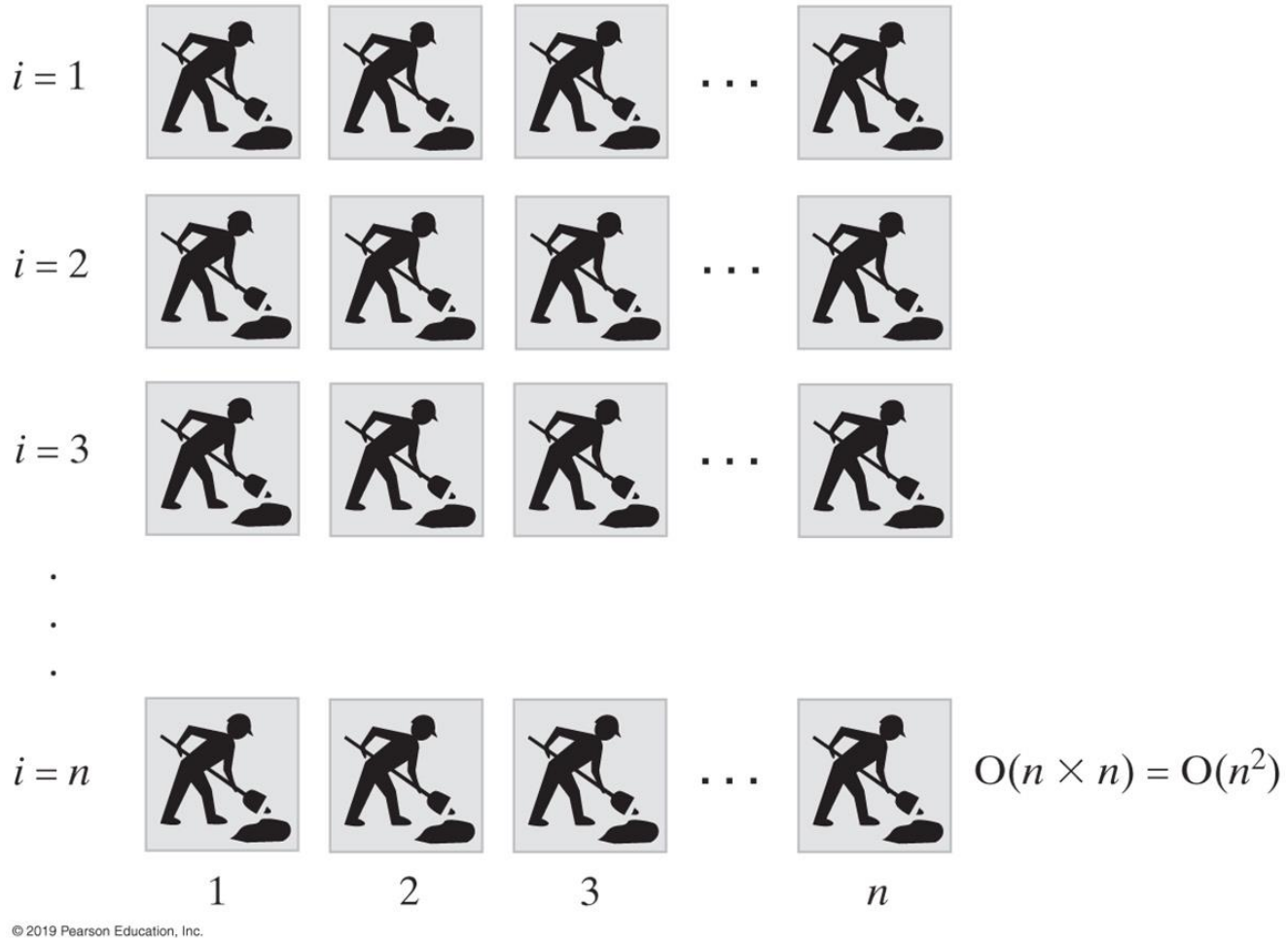
```
sum = 0;
for (long i = 1; i <= n; i++)
{
  for (long j = 1; j <= i; j++)
    sum = sum + 1;
} // end for
```



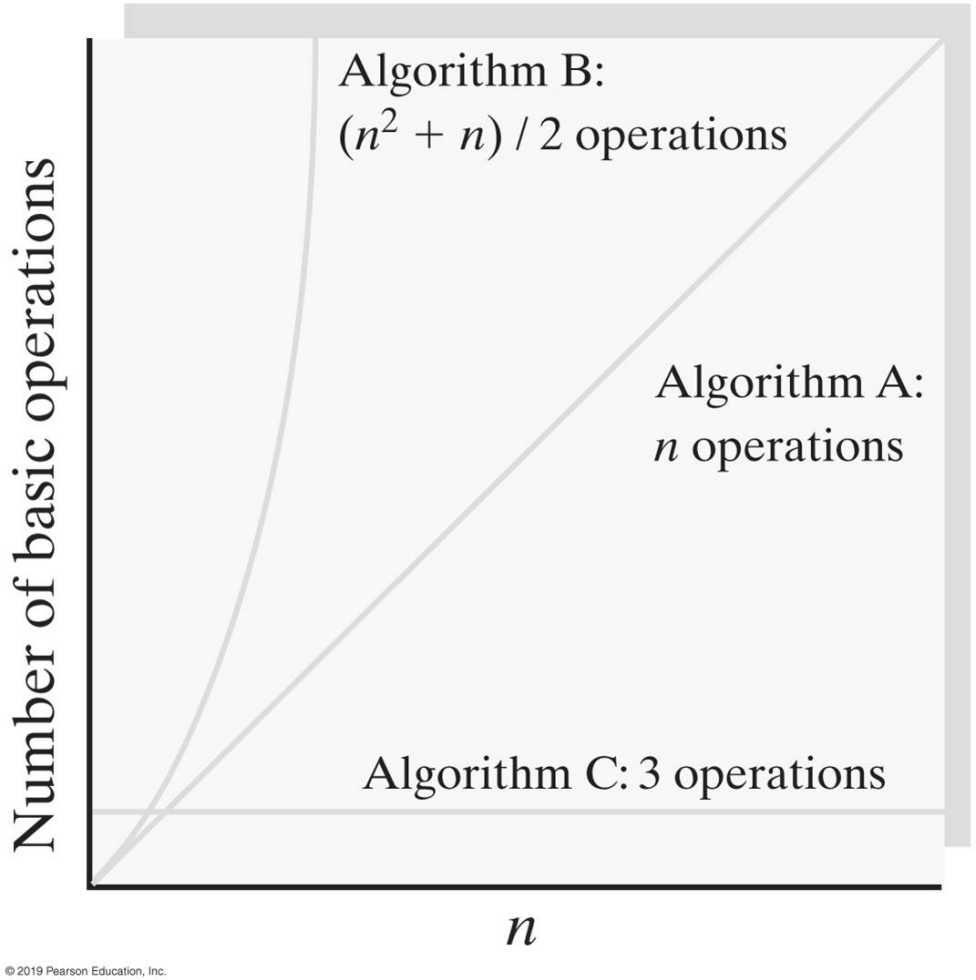
Picturing Efficiency

An $O(n^2)$ algorithm

```
sum = 0;
for (long i = 1; i <= n; i++)
{
  for (long j = 1; j <= n; j++)
    sum = sum + 1;
} // end for
```



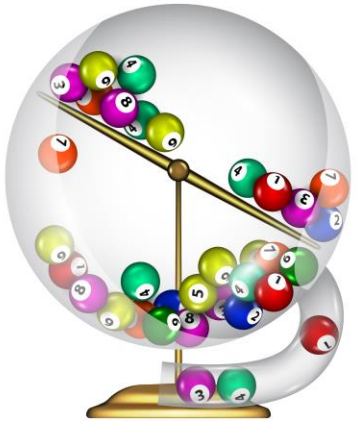
Counting Basic/Elementary Operations



Number of basic operations required by the algorithms as a function of *n*

Our position on Complexity Analysis

What would the reasoning be on buying a lottery ticket
on the basis of best, worst, and average-case complexity ?

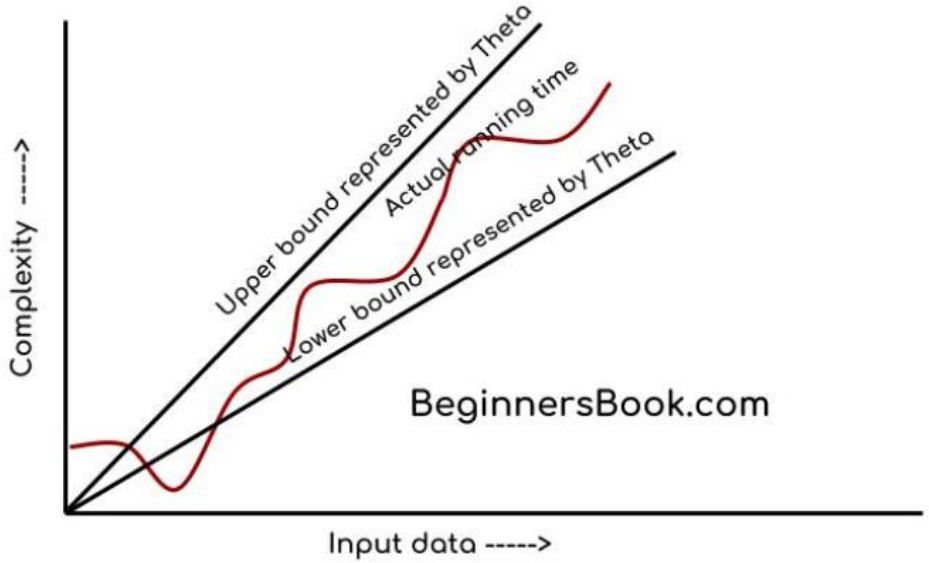


Generally speaking, we will use the worst-case complexity as our preferred measure of algorithm efficiency. Worst-case analysis is generally easy to do, and “usually” reflects the average case.

Therefore, it’s safe to assume we are talking about worstcase analysis unless otherwise specified!

Randomized algorithms are of growing importance, and require an average-case type analysis to show off their merits.

Exact Analysis is Hard!



It easier to talk about upper and lower bounds of the function.

Asymptotic notation (O, Θ,Ω) are as well as we can practically deal with complexity functions.

Names of Bounding Functions

- $\mathbf{g(n) = O(f(n))}$ means $C \times f(n)$ is an upper bound on $g(n)$.
- $\mathbf{g(n) = \Omega(f(n))}$ means $C \times f(n)$ is a lower bound on $g(n)$.
- $\mathbf{g(n) = \Theta(f(n))}$ means $C1 \times f(n)$ is an upper bound on $g(n)$ and $C2 \times f(n)$ is a lower bound on $g(n)$. C, $C1$, and $C2$ are all constants independent of n.

Names of Bounding Functions

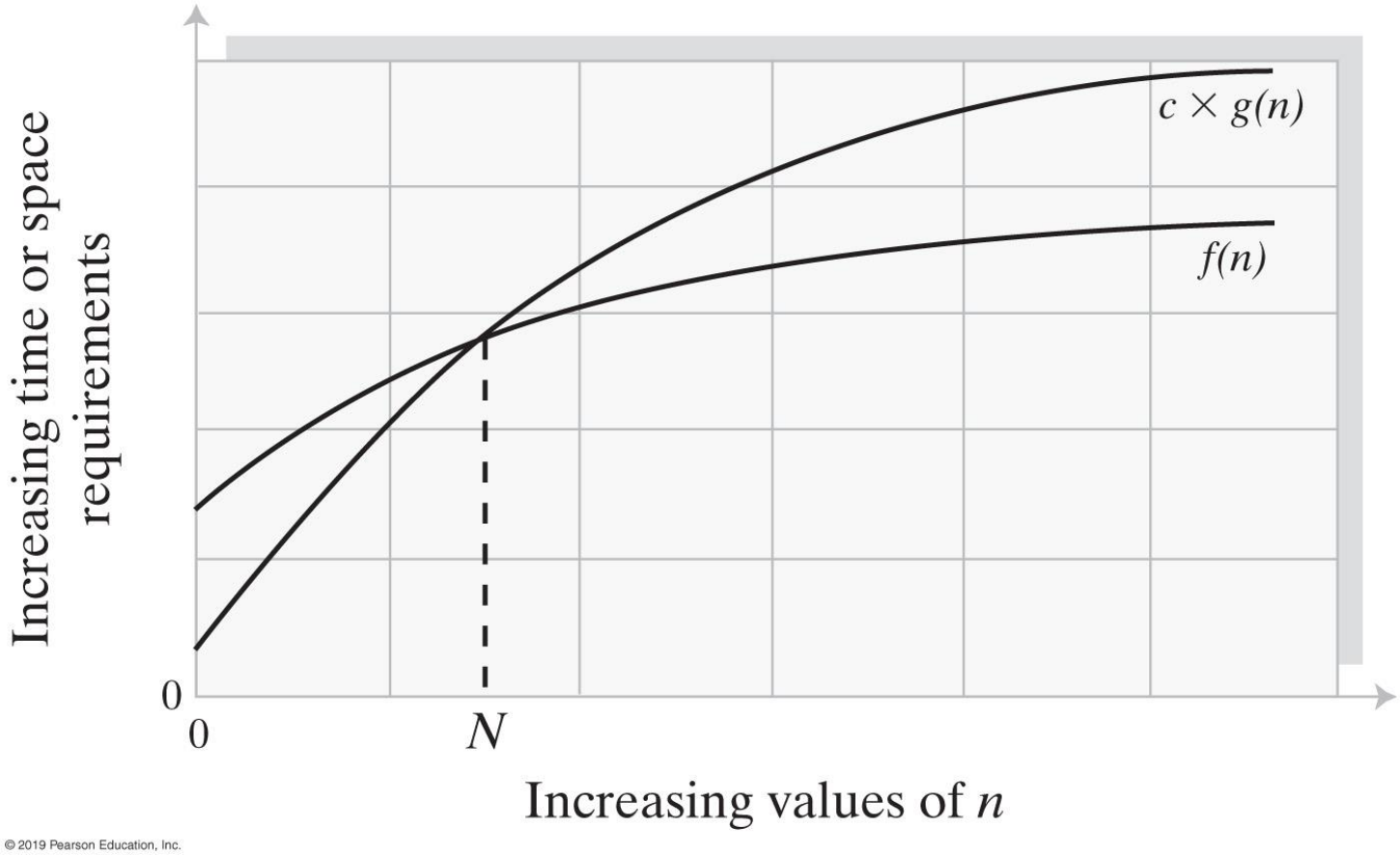
- $\mathbf{f(n) = O(g(n))}$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $c \cdot g(n)$
- $\mathbf{f(n) = \Omega(g(n))}$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $c \cdot g(n)$.
- $\mathbf{f(n) = \Theta(g(n))}$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ inclusive

Big Oh Notation

- A function *f(n)* is of order at most *g(n)*
- That is, *f(n)* is *O(g(n))* — if
  - A positive real number *c* and positive integer *N* exist …
  - Such that $f(n) \leq c \times g(n)$ for all $n \geq N$
  - That is:
    - $c \times g(n)$ is an upper bound on *f(n)* when *n* is sufficiently large

Big Oh Notation

- **An illustration of the values of two growth-rate functions**



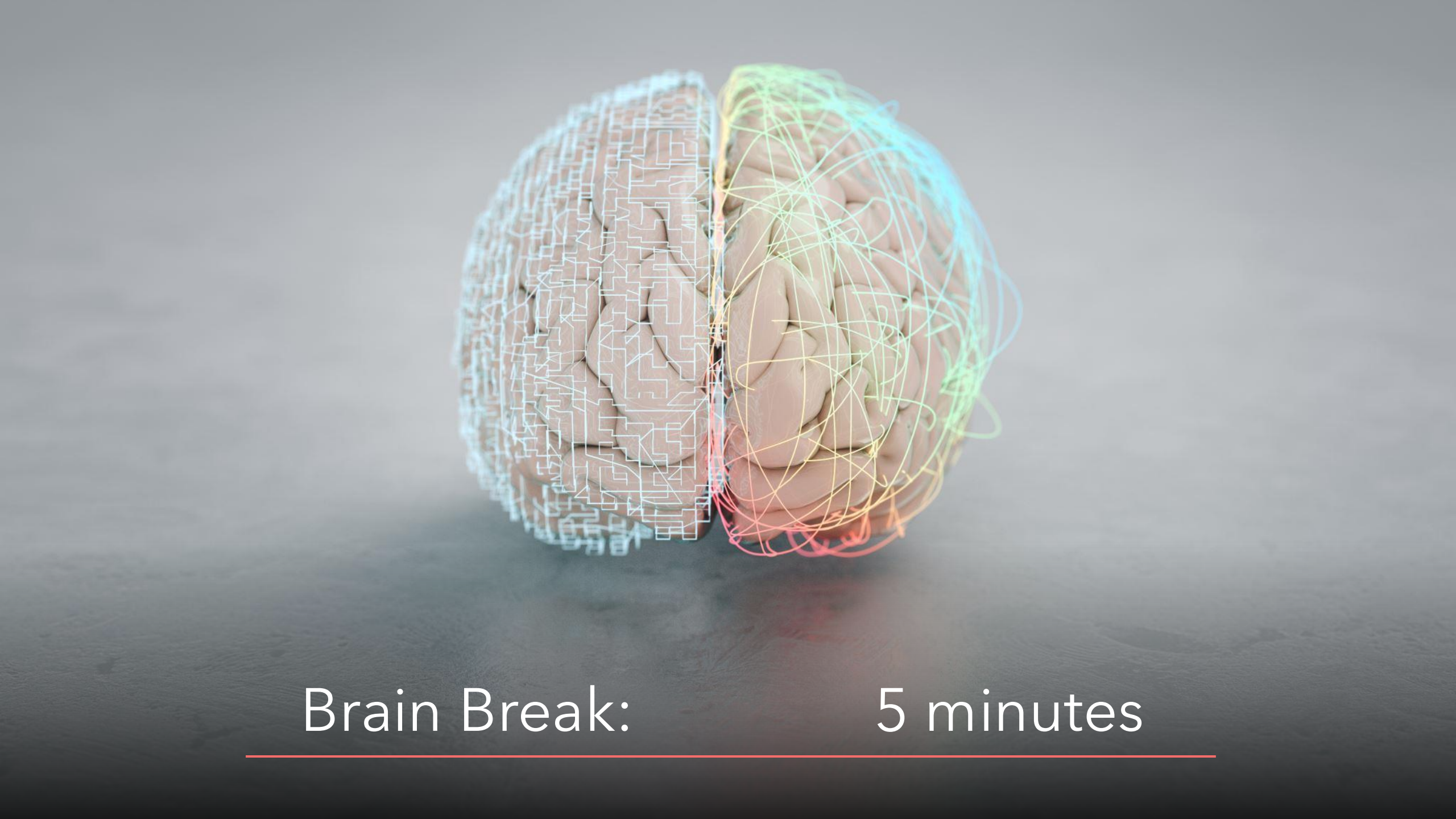
Problems with $O(n^2)$

Real-World Example

- **Assumption**: your machine is capable of performing **1M computations per second**.
- **Task**: Searching for the record of a citizen of a country with **25M population**
- **How long the search would take?**

with $O(n^2)$ a search operation will take 625,000,000 seconds!

~10,416,667 minutes
~173,611 hours
~7,234 days
~241 months
~20 years



Brain Break: 5 minutes

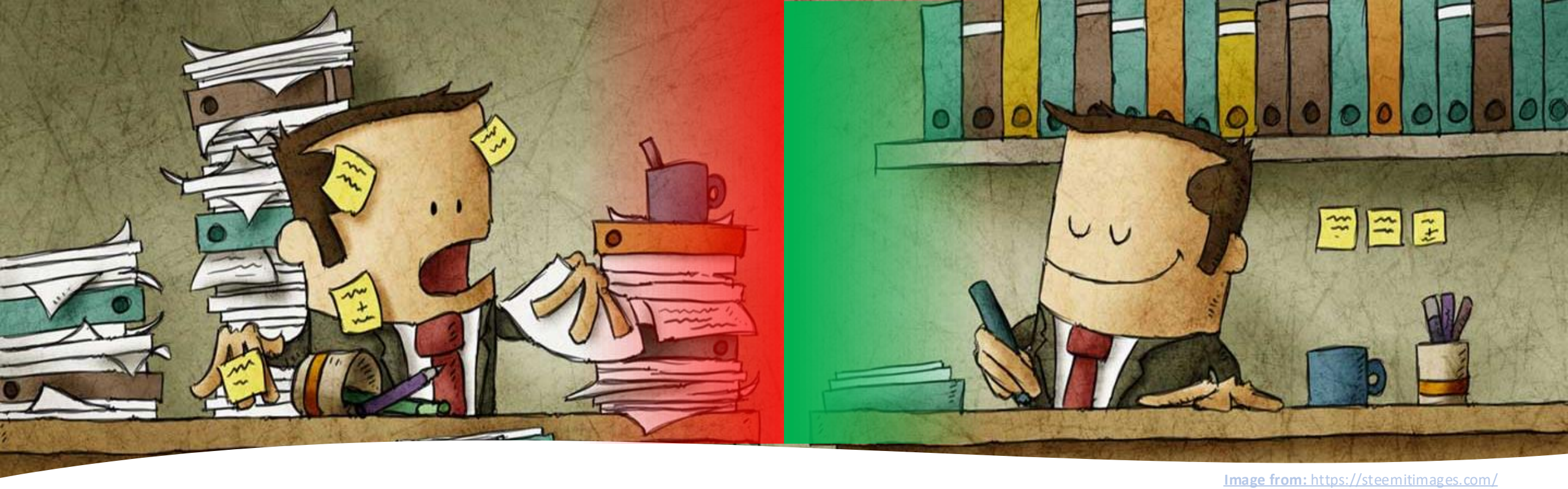


Image from: https://steemitimages.com/

Why sorting is important?

- Sorting is **arranging** the data in ascending or descending order.
- Since sorting can often reduce the complexity of a problem, it is an important algorithm in Computer Science.
- These algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more.
- Fundamental problems that illustrate **Algorithmic Complexity**.

Sorting in Real Life

Sorting: a definition

We seek algorithms to arrange items, a_i such that:

entry 1 ≤ entry 2 ≤ . . . ≤ entry n

Unsorted Array

9	1	3	2	7	4

↓ sorting algorithm

Sorted Array

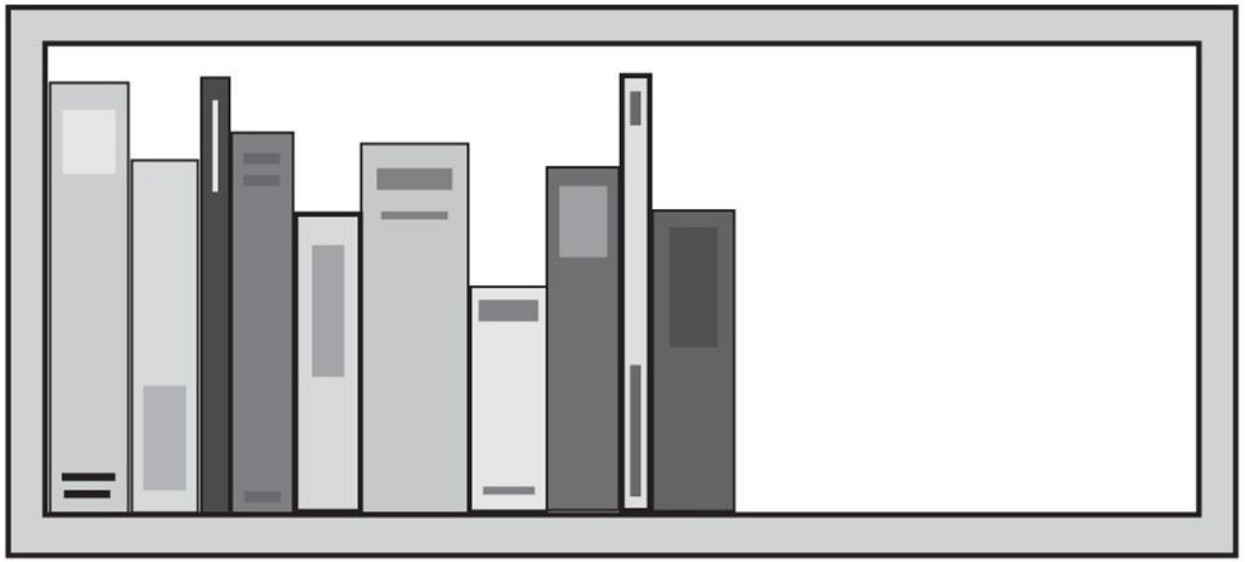
1	2	3	4	7	9

Efficiency of a sorting algorithm is significant

Source: By mentatdgt on Pexels

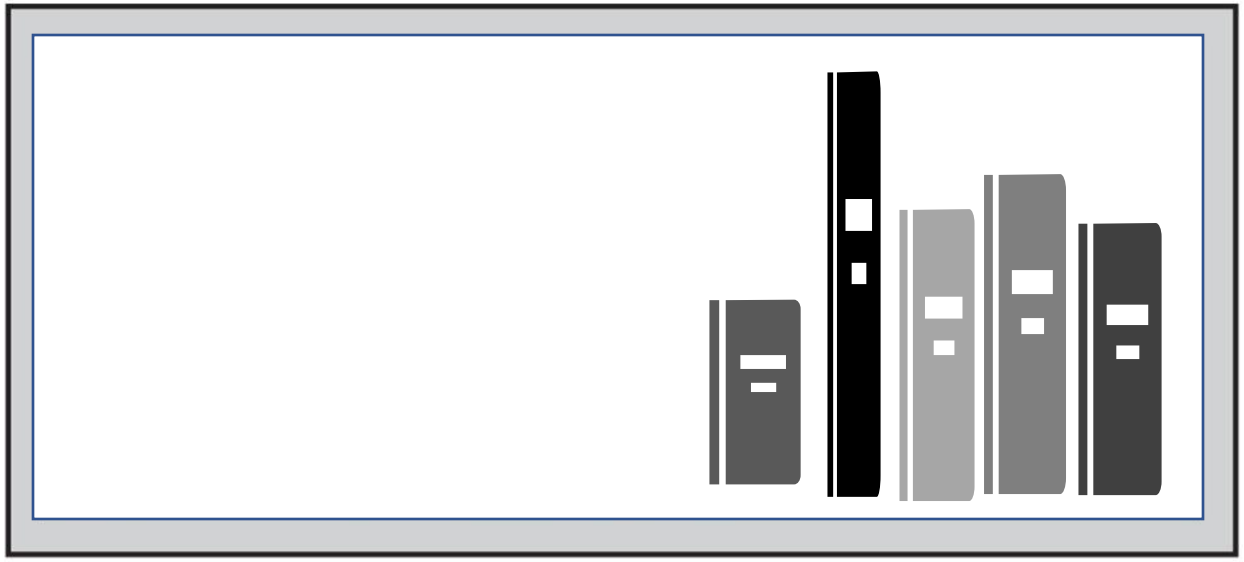
How do you rearrange the books on your bookshelf by height?

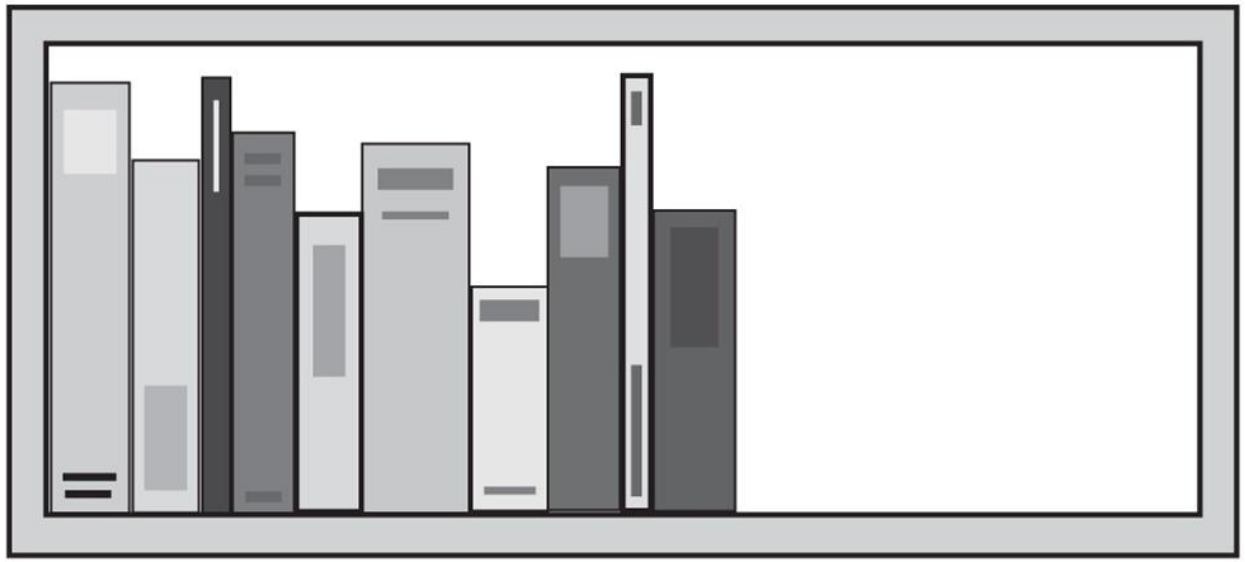
with the shortest book on the left



How do you rearrange the books on your bookshelf by height?

with the shortest book on the left



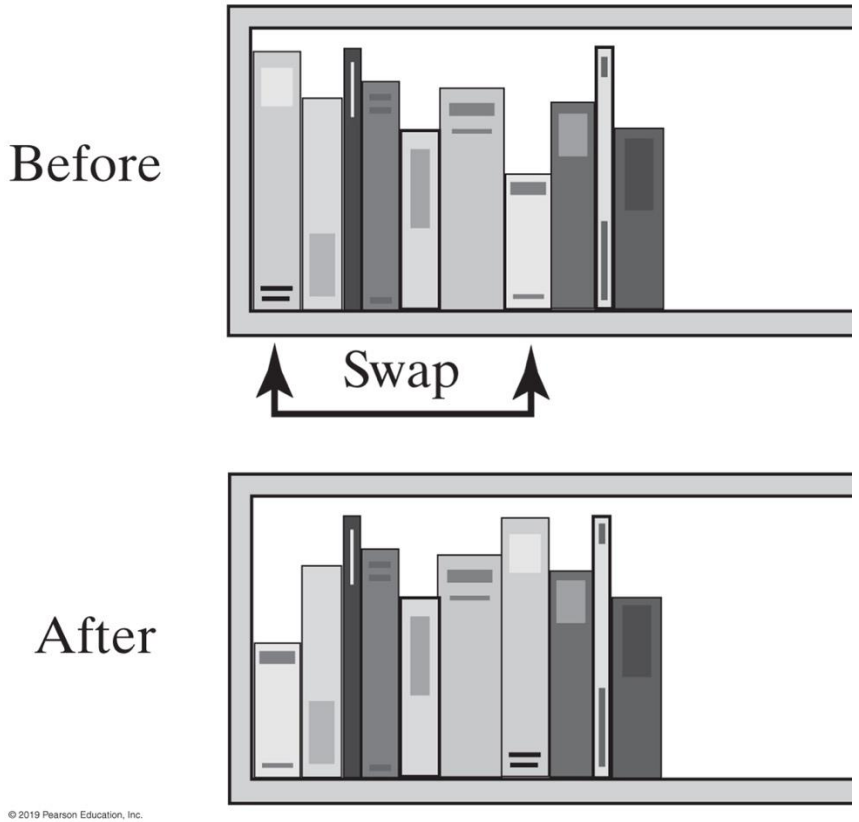


Selection Sort

You might begin by tossing all of the books onto the floor. You then could return them to the shelf one by one, in their proper order. If you first return the shortest book to the shelf, and then the next shortest, and so on, you would perform a kind of selection sort.

But using the floor—or another shelf—to store your books temporarily uses extra space needlessly.

Start by swapping the shortest book with the most left book:

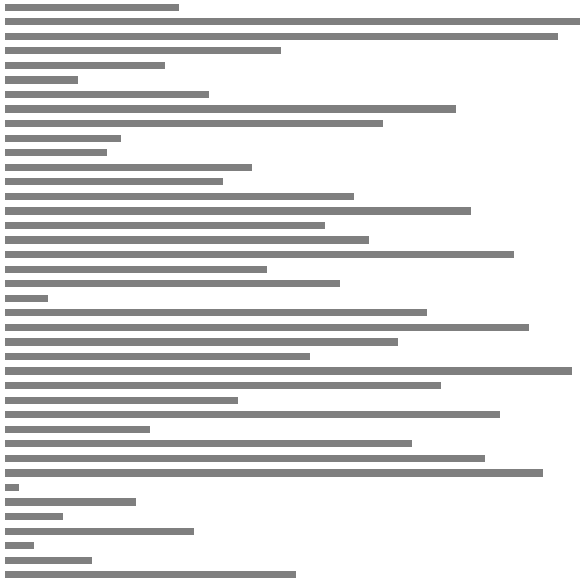


How Selection Sort works?

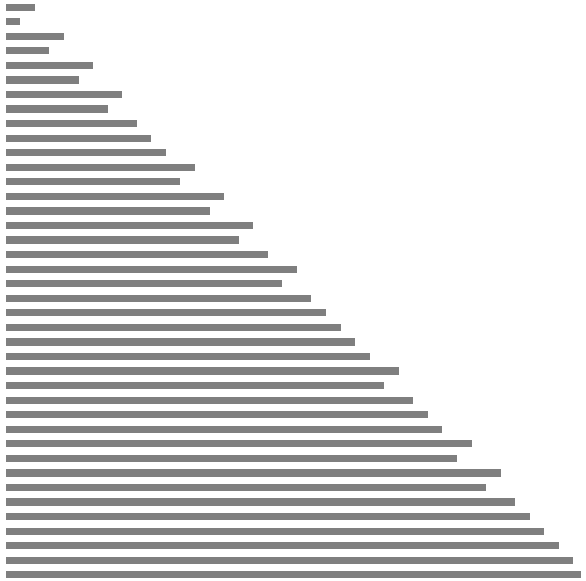
- Selection sort is conceptually one of the most **simplest** sorting algorithms.
- Data divided into sorted (left part) and unsorted (right part). Initially, sorted subarray is empty and unsorted array is the complete given array.
- It is called "**Selection sort**" because it repeatedly **selects** the next-smallest element and swaps it into the right place.
- Starting from the first element, we search for the smallest element, and replace it with the element in the first position.
- Moving on to the second position, we look for the smallest element in the subarray, starting from index 1, till the last index. Then, replace it with the element in the second position. This is repeated, until the array is completely sorted.

How Selection Sort works?

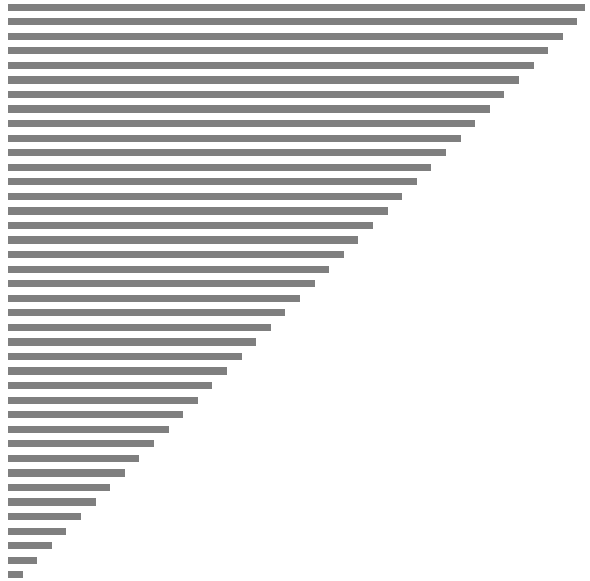
Random



Nearly Sorted



Reversed



Source: Animated Sorting Algorithms: Insertion Sort at the Wayback Machine

Efficiency of Selection Sort

Selection Sort requires two nested for loops:

- One for loop is needed to visit each index, and
- Another for loop within the first one to find the element with minimum value

Hence for a given input size of n:

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)

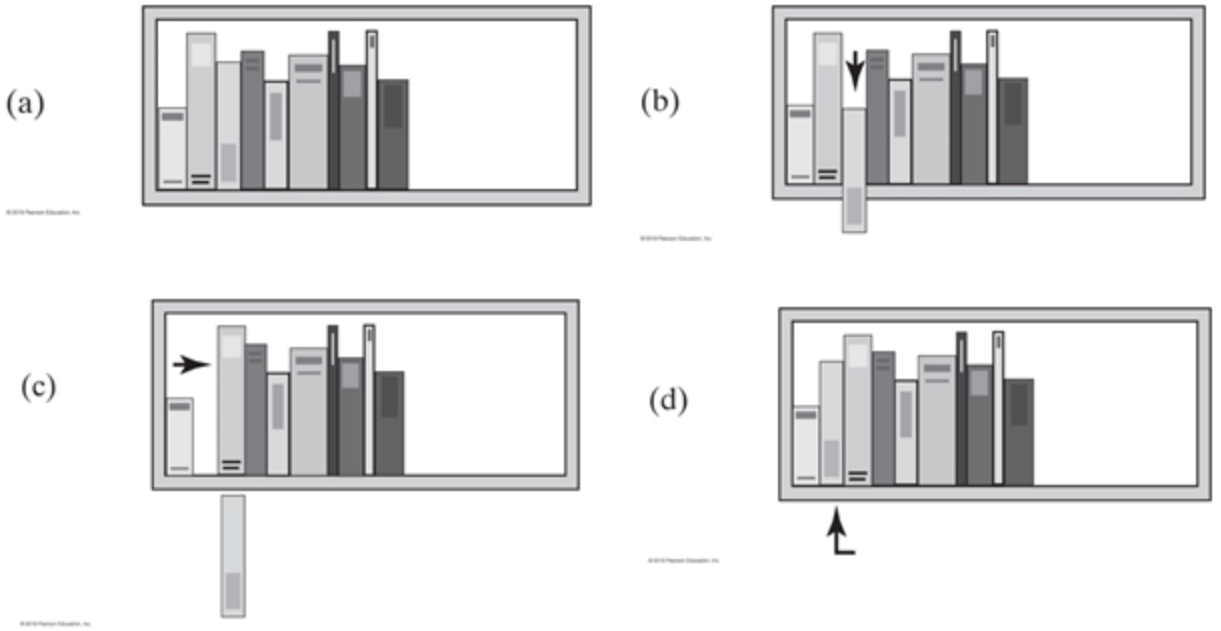
Insertion Sort

Suppose again that you want to rearrange the books on your bookshelf by height, with the shortest book on the left.

If the leftmost book on the shelf were the only book, your shelf would be sorted. But you also have all the other books to sort.

Consider the second book. If it is taller than the first book, you now have two sorted books. If not, you remove the second book, slide the first book to the right, and *insert* the book you just removed into the first position on the shelf. The first two books are now sorted.

This intuitive sorting is the insertion sort.

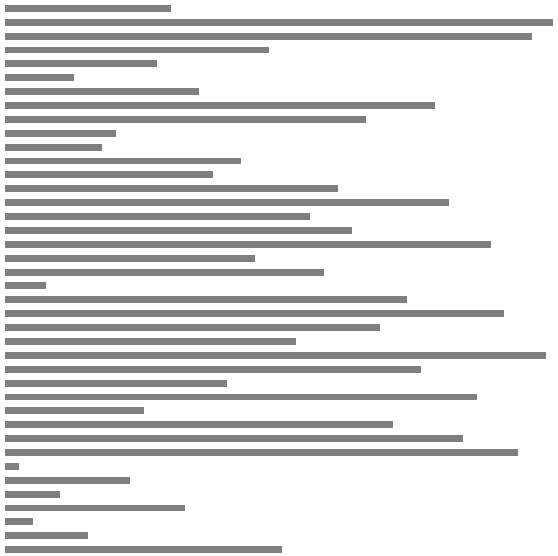


How Insertion Sort works?

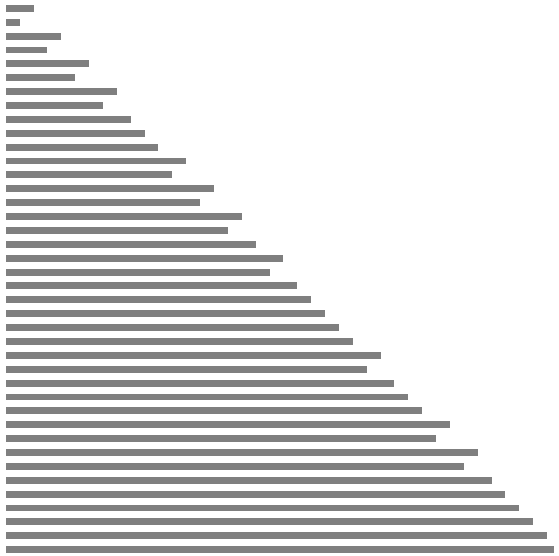
- Data divided into sorted (left part) and unsorted (right part). A set of size 1 is always sorted, let's keep it that way.
- This sort works on the principle of inserting an element at a particular position, hence the name **Insertion Sort**.
- Starting with the second element (the key). The key element needs to be added to our existing sorted set.
- We compare the key element with the element(s) before it, in this case, first element to keep the list sorted.
- Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position. Repeat this, until the array is sorted.

How Insertion Sort works?

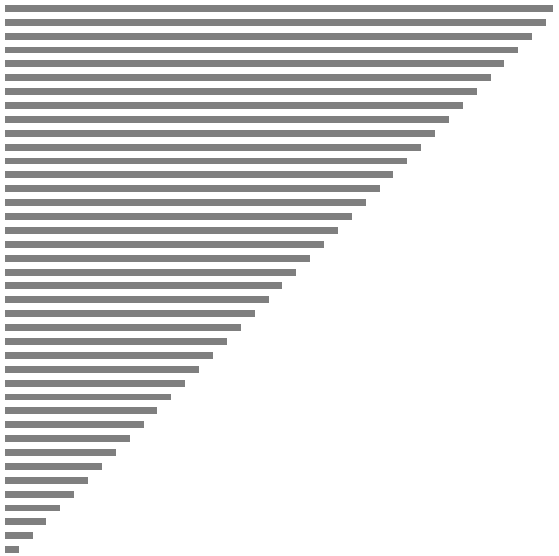
Random



Nearly Sorted



Reversed



Source: Animated Sorting Algorithms: Insertion Sort at the Wayback Machine

Efficiency of Insertion Sort

In Insertion, for an unsorted array, it takes for an element to compare with all the other elements which mean every n element compared with all other n elements.

- Thus, making it for n x n comparisons => O(n^2)

Hence for a given input size of n

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)

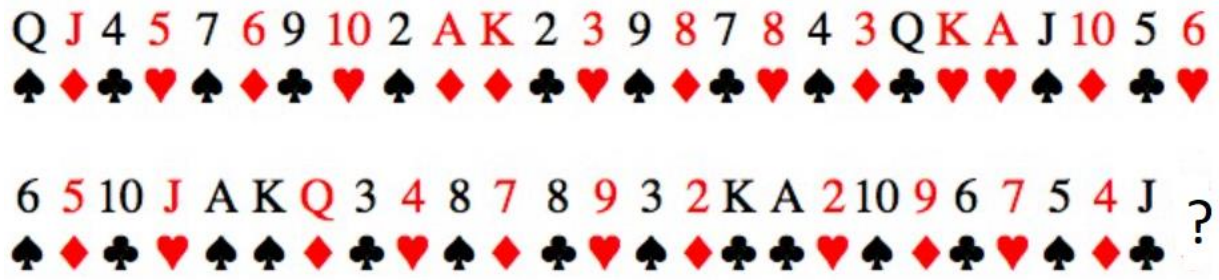
Sample Quiz

Week #1 Quiz – Q1

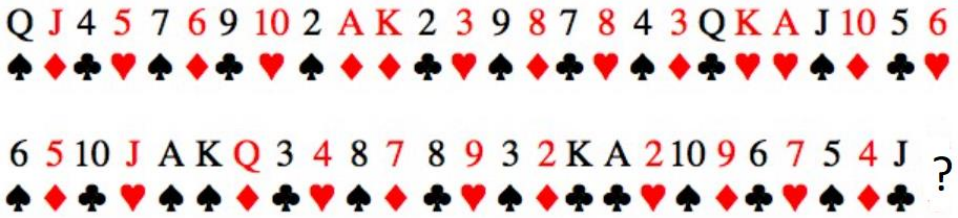
There are many algorithms for determining the missing card.

Here is one (inefficient) algorithm: for each of the 52 possible cards, go through the entire pile and check whether that card appears in the shuffled deck of 51 cards. If it does not, then that card must be the missing card.

Design a better algorithm for determining the missing card.



Week #1 Quiz – Q1



Design a better algorithm for determining the missing card.

Given an array containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

If the array is [5,3,7,1,2,0,4], return 6.

If the array is [5,3,7,1,2,6,4], return 0.

If the array is [5,3,0,1,2,6,4], return 7.

Algorithm #1: for each number x between 0 and n, go through the entire array to see if x appears. If it does not, then return x. In the worst case, this algorithm requires (n+1)n comparisons, so the running time is $O(n^2)$.

Algorithm #2: sort the array using Merge Sort. We know that this can be done in O(n log n) time. For example, [5,3,7,1,2,0,4] becomes [0,1,2,3,4,5,7]. Now we just scan the array once to find the missing number. The running time is O(n log n).

Algorithm #3: add the elements and subtract it from (0+1+2+…+n) = n*(n+1)/2. Adding is very easy, and each element is only looked at once – so the running time is O(n).

Week #1 Quiz – Q2

Let f(n) and g(n) be two functions, defined for each positive integer n.

By definition, f(n) = O(g(n)) if there exist positive constants **c** and $\mathbf{n_0}$ for which for all integers .

Here are six claims:

1. $10n^2 + 20n + 30 = O(n^2)$
2. $5000n^3 + 5000 = O(n^4)$
3. $4^n = O(2^n)$
4. $4^n = O(n!)$
5. $n^{1000} = O(2^n)$
6. $2^n = O(n^{1000})$

Determine how many of these statements are correct.

Week #1 Quiz – Q2

Here are six claims:

1. $10n^2 + 20n + 30 = O(n^2)$
2. $5000n^3 + 5000 = O(n^4)$
3. $4^n = O(2^n)$
4. $4^n = O(n!)$
5. $n^{1000} = O(2^n)$
6. $2^n = O(n^{1000})$

Correct answers: TRUE, TRUE, FALSE, TRUE, TRUE, FALSE.

1. $10n^2 + 20n + 30 = O(n^2)$ is TRUE
2. $5000n^3 + 5000 = O(n^4)$ is TRUE
3. $4^n = O(2^n)$ is FALSE, since $4^n <= c * 2^n$ implies $2^n <= c$. No matter what c is, you will always be able to find a large enough n for which this statement is false.
4. $4^n = O(n!)$ is TRUE, since $4^n <= 1 * n!$ is true for all n>=9. Easy proof by induction. If $4^k < k!$, then $4^{k+1} = 4 * 4^k < 4\ k! < (k+1)\ k! = (k+1)!$
5. $n^{1000} = O(2^n)$ is TRUE, since every polynomial function is bounded by any exponential function – i.e., eventually $n^{1000} < 2^n$ for all large enough n. And this implies $2^n = O(n^{1000})$ is FALSE.

Questions?

Northeastern University