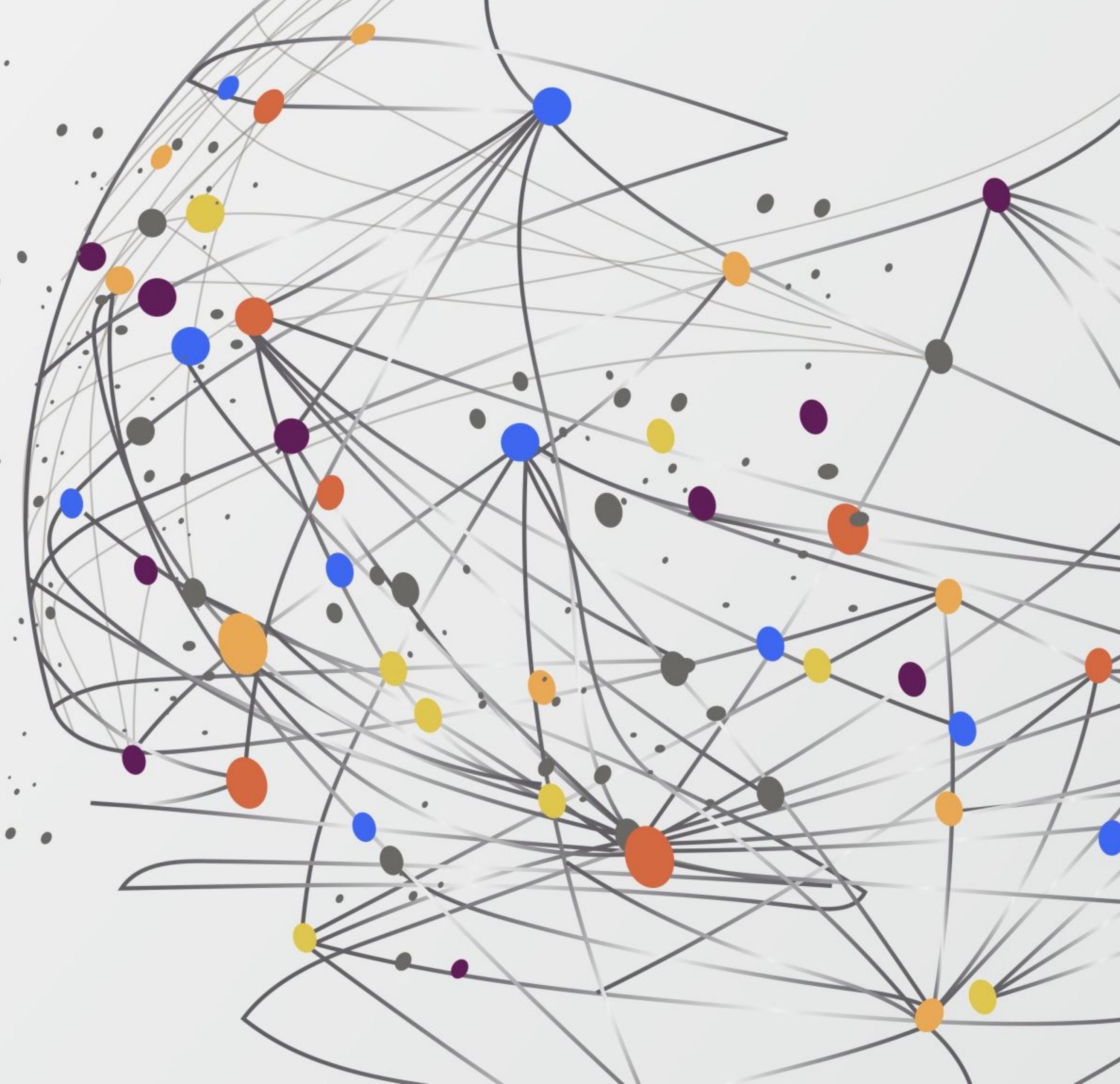
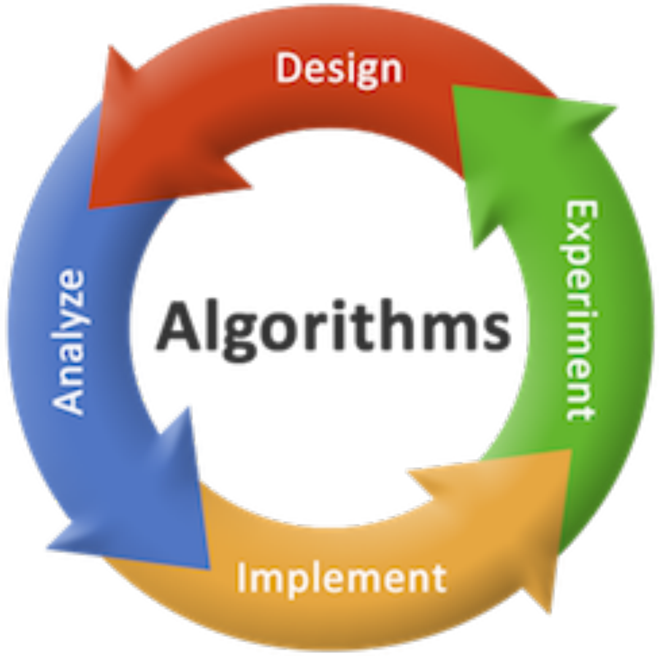
Analysis of Algorithms: Terminology and Concepts

Carlos J. Barrios H. PhD

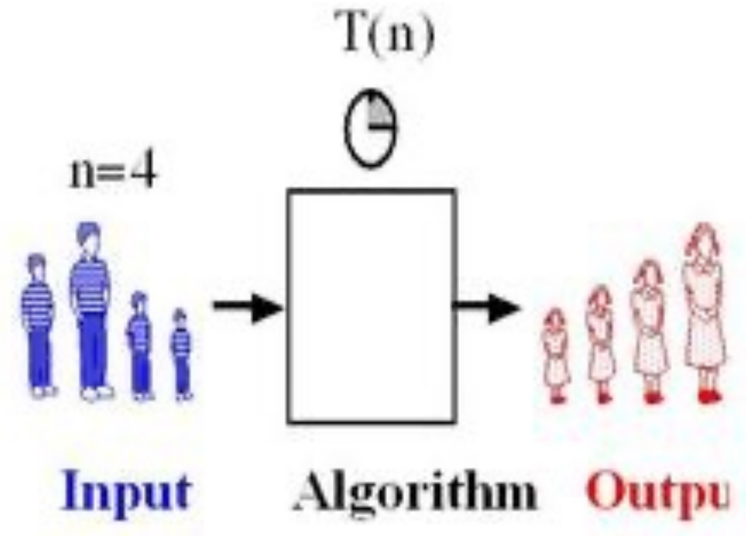
@carlosjaimebh



What and how?

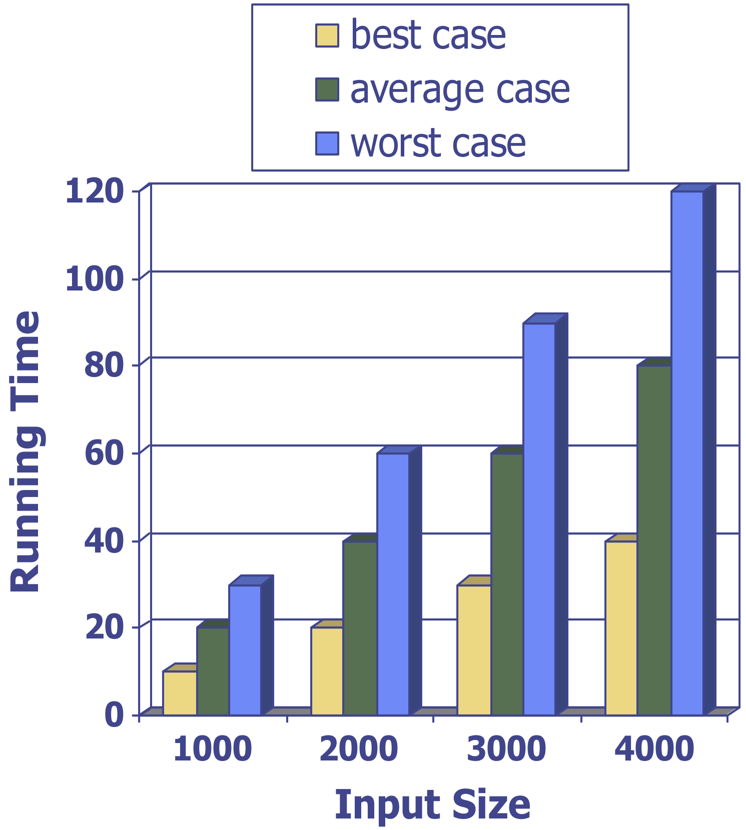


- Running Time
- Pseudo-Code
- Analysis of Algorithms
- Asymptotic Notation
- Asymptotic Analysis
- Mathematical facts



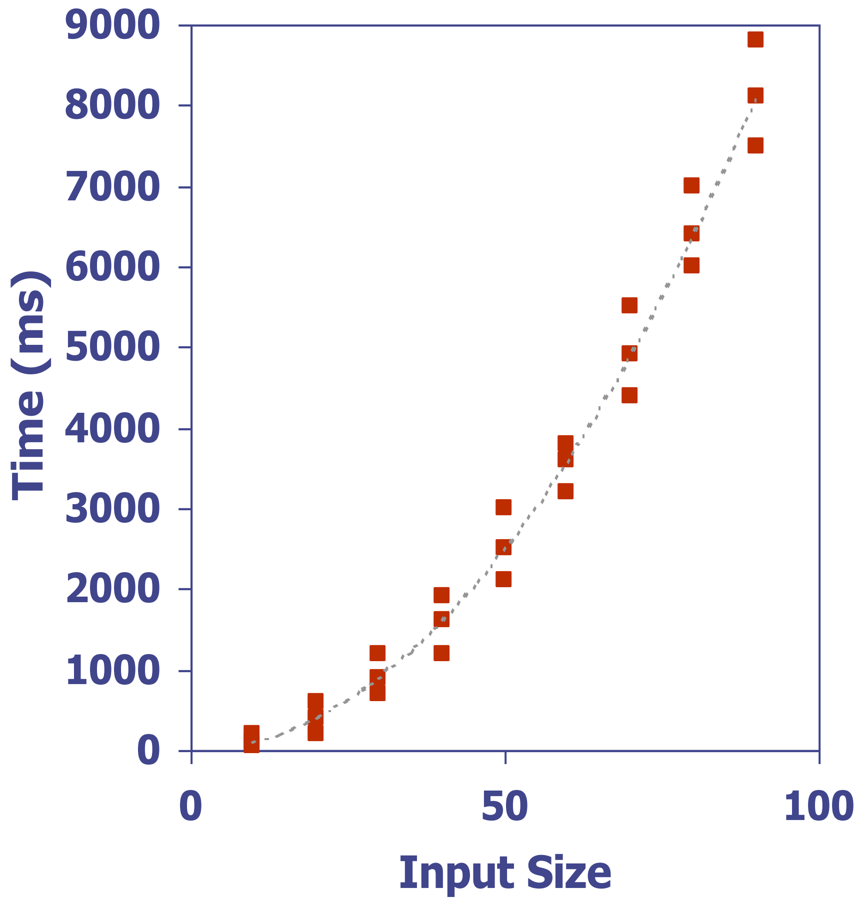
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



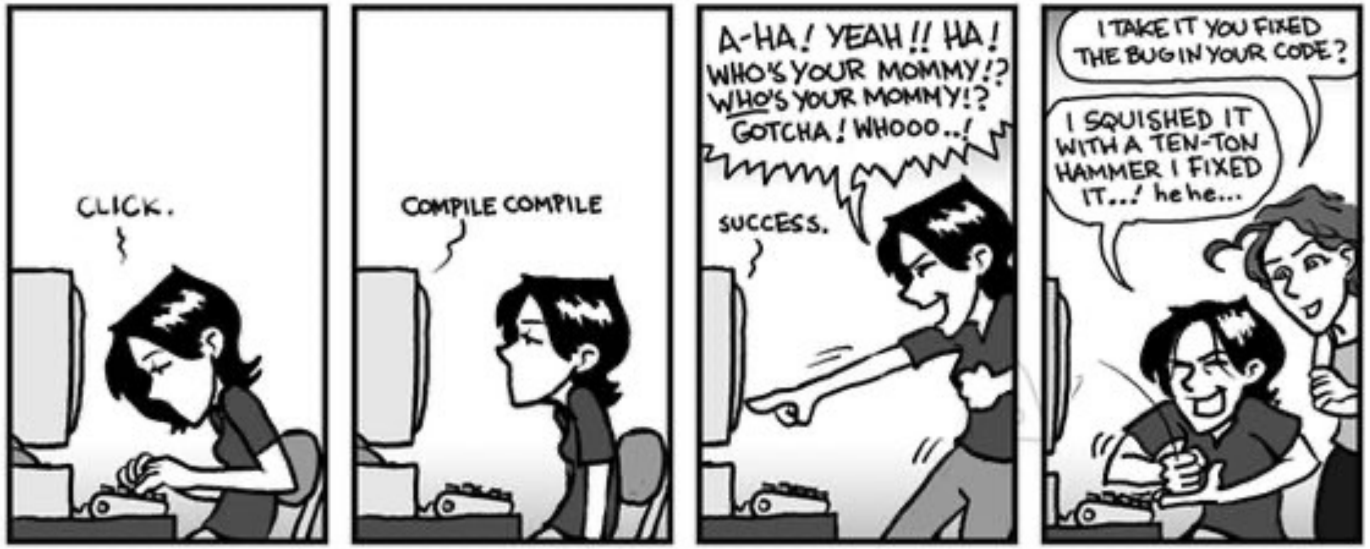
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

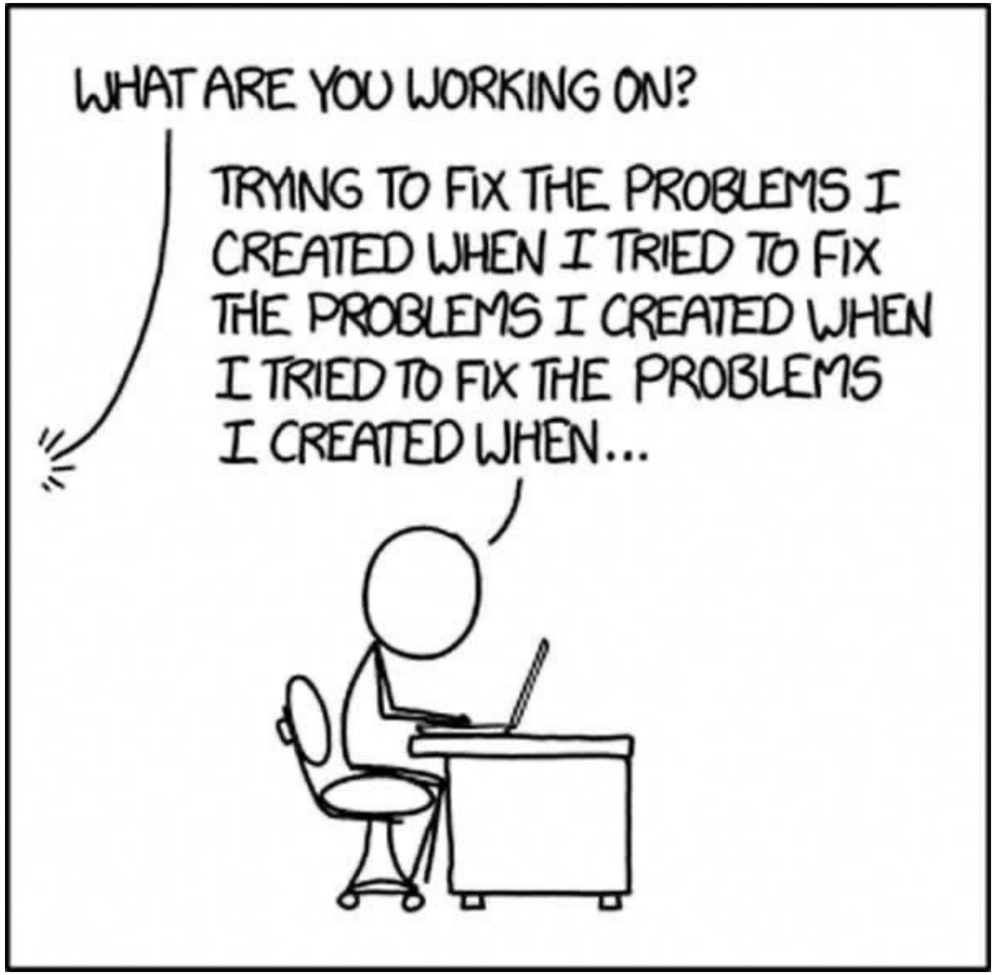
- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used



phd.stanford.edu/

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment



Remember the Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm $arrayMax(A, n)$
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode Details (from the first class)

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm *method* (*arg* [, *arg*…])

  Input …

  Output …

- Method call

  var.method (*arg* [, *arg*…])

- Return value

  return *expression*

- Expressions

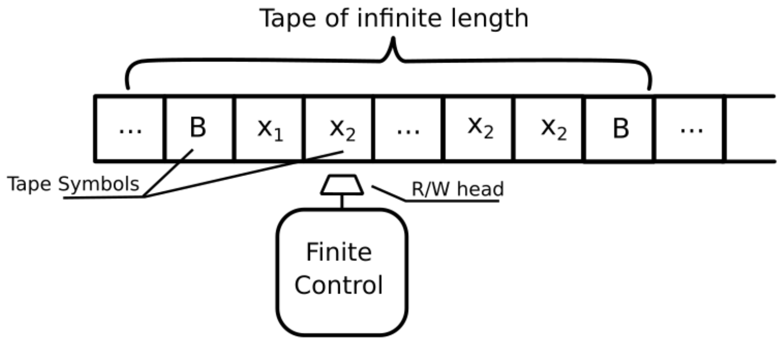
  $\leftarrow$Assignment (like $=$ in Java)

  $=$ Equality testing (like $==$ in Java)

  n^2 Superscripts and other mathematical formatting allowed

An Introduction to Machine Models

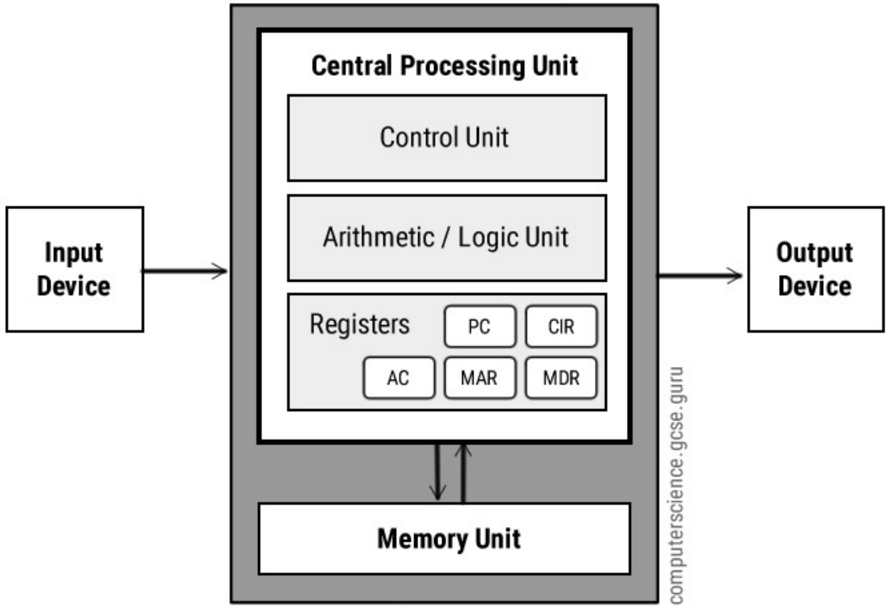
- What is a machine model?
  - A abstraction describes the operation of a machine.
  - Allowing to associate a value (cost) to each machine operation.
- Why do we need models?
  - Make it easy to reason algorithms
  - Hide the machine implementation details so that general results that apply to a broad class of machines to be obtained.
  - Analyze the achievable complexity (time, space, etc) bounds
  - Analyze maximum parallelism (to see later)
  - Models are directly related to algorithms.



A **Turing machine** is a mathematical model of computation that defines an abstract machine[1] that manipulates symbols on a strip of tape according to a table of rules.[2] Despite the model's simplicity, given any computer algorithm, a Turing machine capable of simulating that algorithm's logic can be constructed.

More in: https://plato.stanford.edu/entries/turing-machine/

(A Parenthesis about Computer Architecture)



(Remember this for later)

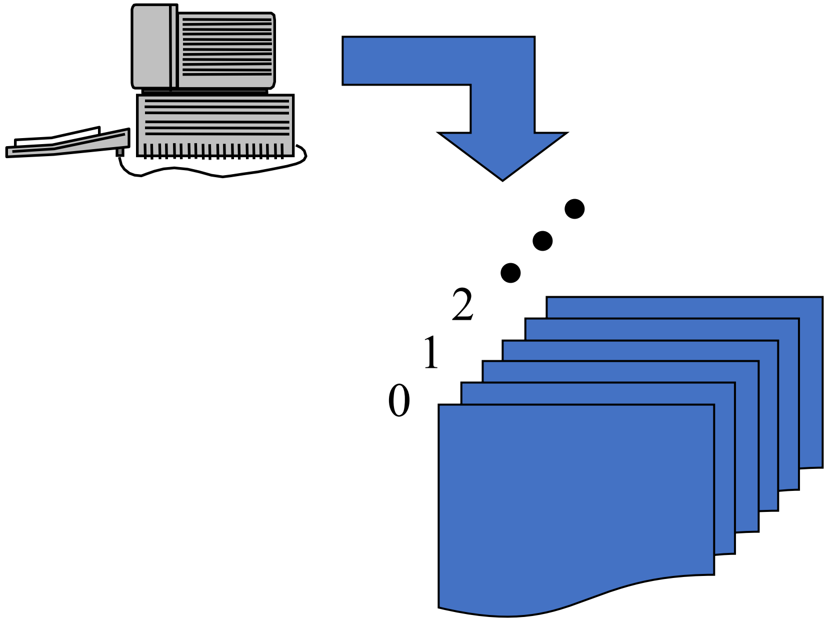
- **Von Neumann Architecture**
- Von Neumann architecture was first published by John von Neumann in 1945.
- His computer architecture design consists of a Control Unit, Arithmetic and Logic Unit (ALU), Memory Unit, Registers and Inputs/Outputs.
- Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.

More in: https://www.computerscience.gcse.guru/theory/von-neumann-architecture

The Random Access Memory (RAM) Model

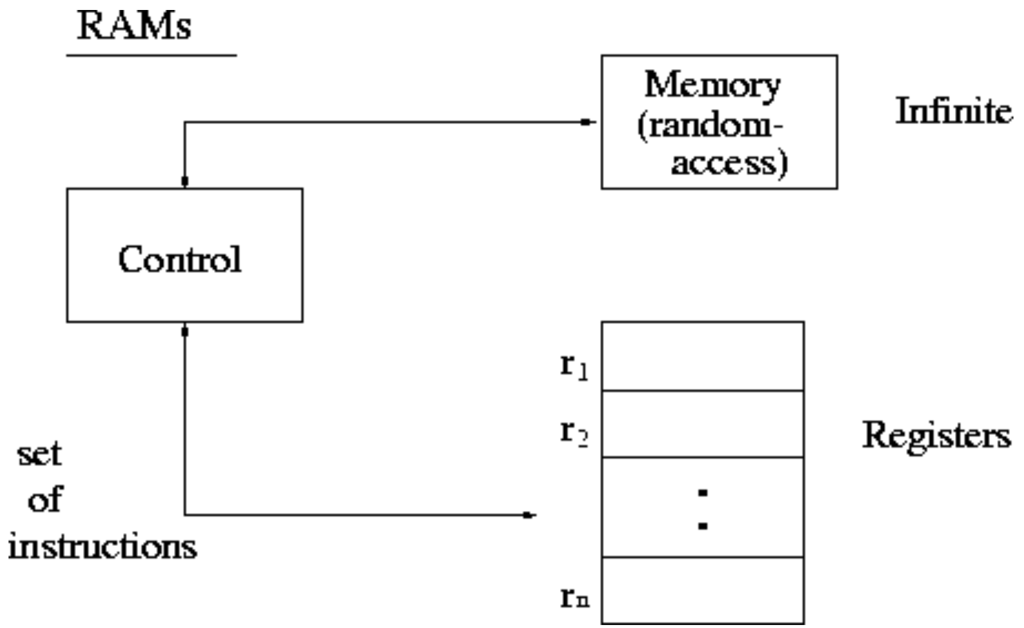
- A **CPU**

- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



◆ Memory cells are numbered and accessing any cell in memory takes unit time.

RAM (Random Access Machine) model in detail



- Memory consists of infinite array (memory cells).
- Each memory cell holds an infinitely large number.
- Instructions execute sequentially one at a time.
- All instructions take unit time
  - Load/store
  - Arithmetic
  - Logic
- Running time of an algorithm is the number of instructions executed.
- Memory requirement is the number of memory cells used in the algorithm.

Important points of RAM (random access machine) model

- The RAM model is the base of algorithm analysis for sequential algorithms although it is not perfect.
  - Memory not infinite
  - Not all memory access take the same time
  - Not all arithmetic operations take the same time
  - Instruction pipelining is not taken into consideration
- The RAM model (with asymptotic analysis) often gives relatively realistic results.

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return $currentMax$	1
Total	$8n - 2$

Estimating Running Time

- Algorithm ***arrayMax*** executes $8\boldsymbol{n} - 2$ primitive operations in the worst case. Define:

  $\boldsymbol{a}$ = Time taken by the fastest primitive operation

  $\boldsymbol{b}$ = Time taken by the slowest primitive operation

- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***arrayMax.*** Then

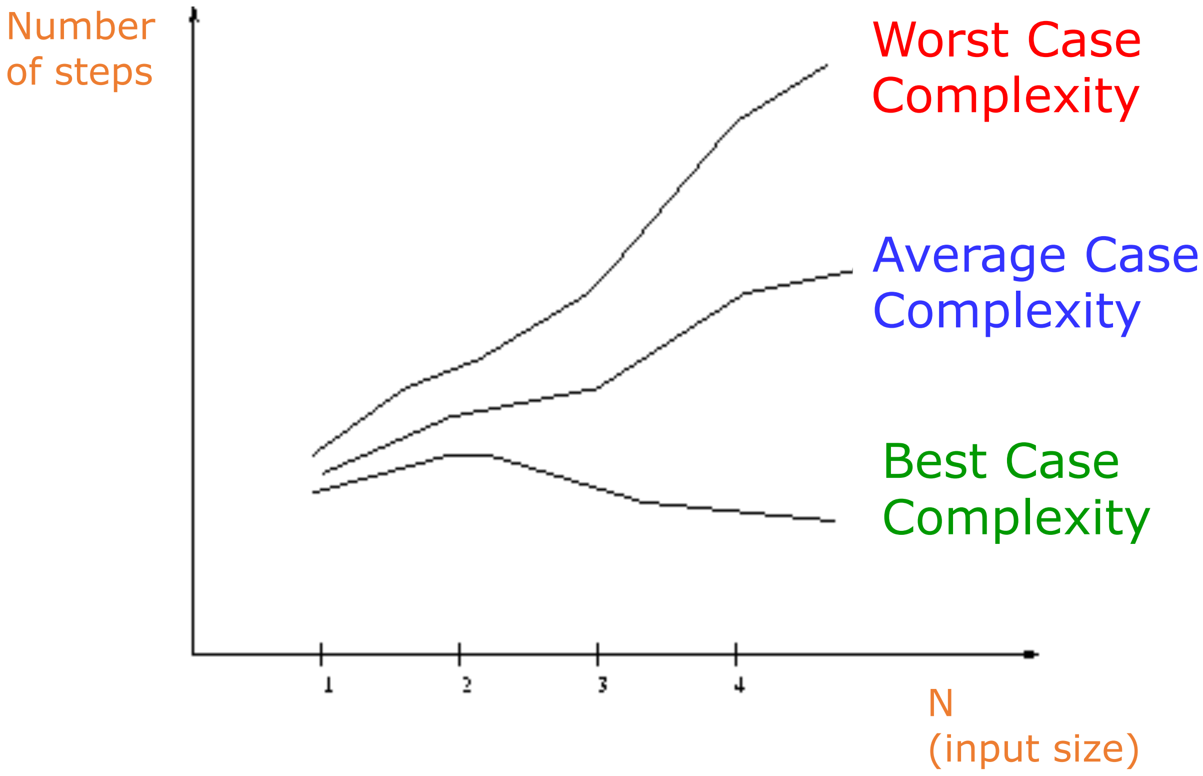
$$\boldsymbol{a}\,(8\boldsymbol{n} - 2) \leq \boldsymbol{T}(\boldsymbol{n}) \leq \boldsymbol{b}(8\boldsymbol{n} - 2)$$

- Hence, the running time $\boldsymbol{T}(\boldsymbol{n})$ is bounded by two linear functions

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $\boldsymbol{T(n)}$ by a constant factor, but
  - Does not alter the growth rate of $\boldsymbol{T(n)}$
- The linear growth rate of the running time $\boldsymbol{T(n)}$ is an intrinsic property of algorithm ***arrayMax***

Remember the Best, Worst, and Average Case Complexity



- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*

Algorithm Complexity

- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average* number of steps taken on any instance of size *n*

Principles

- **Ignore Machine-Dependent Constants:** It will not be concerned how fast an individual processor executes a machine instruction.
- **Look at growth of T(n) as n → ∞ (where T(n) is the running time of an algorithm operating on a data ser of size n):** Even an inefficient algorithm will often finish its work in an acceptable time when operating on a small data set.
- **Growth Rate:** Taking a function as n gets large, it will ignore constant factors when expressing asymptotic analysis.

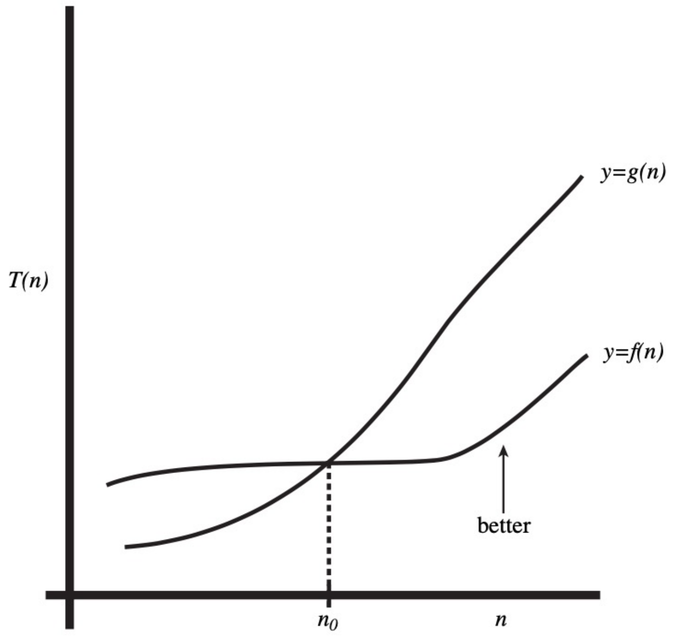


FIGURE 1.1 *An illustration of the growth rate of two functions,* f(n) *and* g(n)*. Notice that for large values of* n*, an algorithm with an asymptotic running time of* f(n) *is typically more desirable than an algorithm with an asymptotic running time of* g(n)*. In this illustration, "large" is defined as* $n \geq n_0$.

From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Asymptotic Notation (1/3)

- The goal is to express the asymptotic behavior of a function.
  - Suppose f and g are positive functions of n. Then

f (n) = Θ(g(n)) (read "f of n is **theta** of g of n") if and only if there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ whenever $n \geq n_0$ (See Figure 1.2)

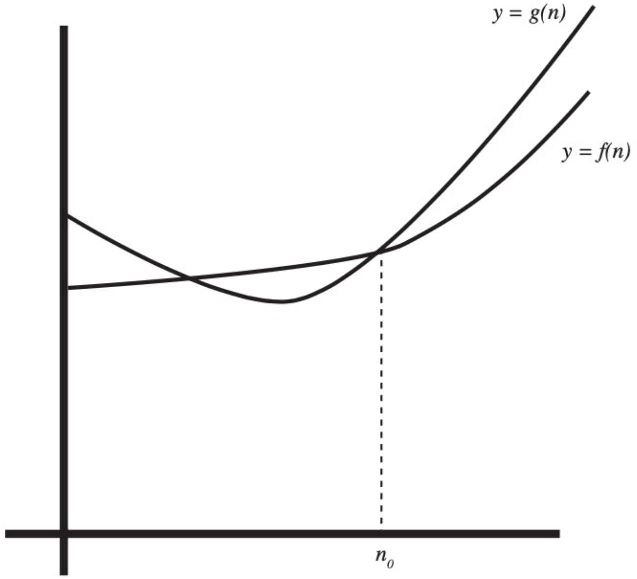


FIGURE 1.2 *An illustration of* Θ *notation.* f(n) = Θ(g(n)) *because functions* f(n) *and* g(n) *grow at the same rate for all* $n \geq n_0$.

From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Asymptotic Notation (2/3)

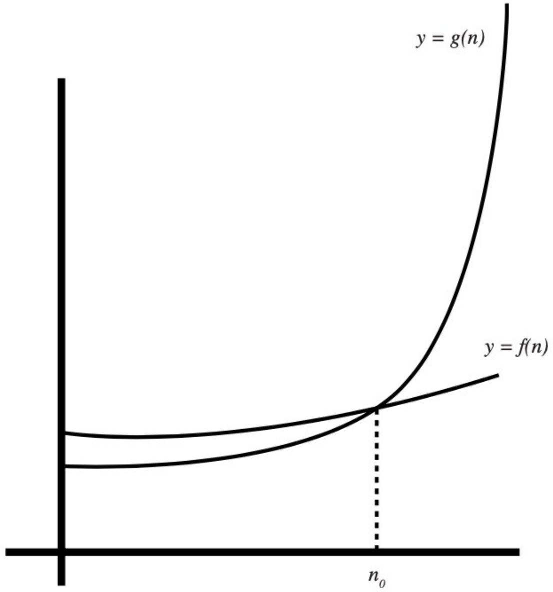


FIGURE 1.3 *An illustration of* O *notation.* f(n) = O(g(n)) *because function* f(n) *is bounded from above by* g(n) *for all* $n \geq n_0$.

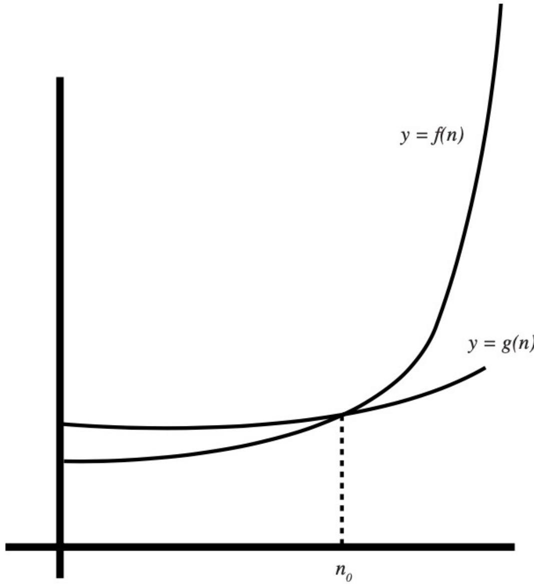


FIGURE 1.4 *An illustration of* Ω *notation.* f(n) = Ω(g(n)) *because function* f(n) *is bounded from below by* g(n) *for all* $n \geq n_0$.

- f(n) = O(g(n)) (read "f of n is **oh** of g of n") if and only if there exist positive constants c and n_0 such that f (n) ≤ cg(n) whenever $n \geq n_0$. (See Figure 1.3).
- f (n) = **Ω**(g(n)) (read "f of n is **omega** of g of n") if and only if there exist positive constants c and n_0 such that $cg(n) \leq f\ (n)$ whenever $n \geq n_0$. (See Figure 1.4).

From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Asymptotic Notation (3/3)

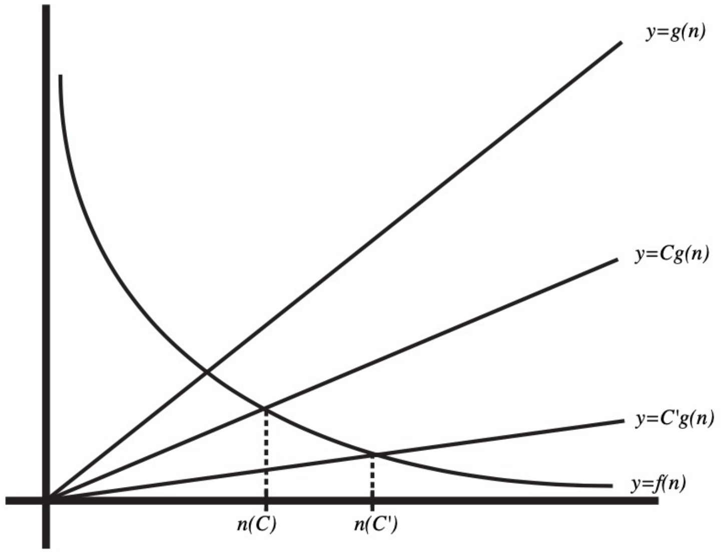


FIGURE 1.5 *An illustration of* o *notation:* f(n) = o(g(n)).

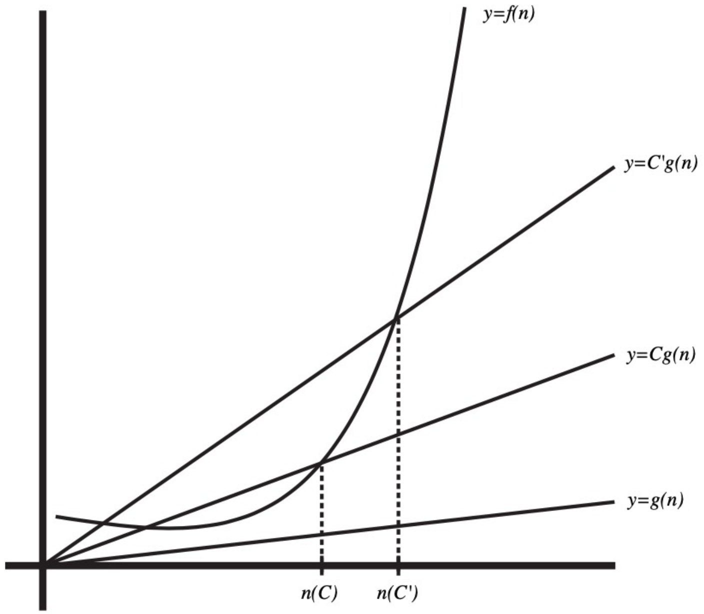


FIGURE 1.6 *An illustration of* ω *notation:* f(n) = ω(g(n)).

- f(n) = o(g(n)) (read "f of n is **little oh** of g of n") if and only if for every positive constant C there is a positive integer n_0 such that f (n) < Cg(n) whenever $n \geq n_0$. (See Figure 1.5).
- f(n) = **ω**(g(n)) (read "f of n is **little omega** of g of n") if and only if for every positive constant C there is a positive integer n_0 such that f (n) > Cg(n) whenever $n \geq n_0$. (See Figure 1.6).

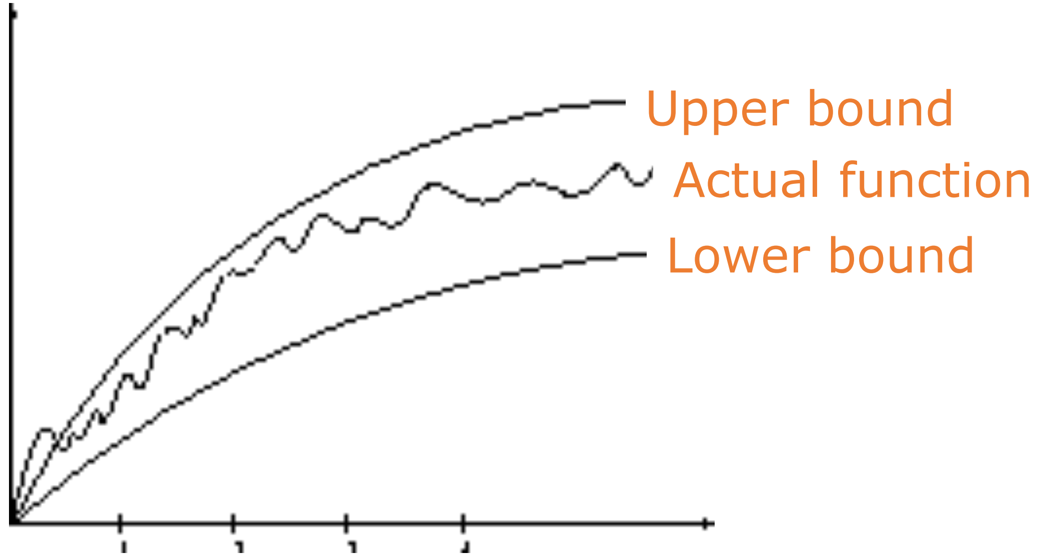
From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Asymptotic Notation for Asymptotic Analysis

- **Θ, *O*, Ω, *o*** *and* ω are set-valued functions, in practice used to compare two function sizes.
- The notations describe different ***rate-of-growth*** relations between the defining function and the defined set of functions.
- Asymptotic growth rate, asymptotic order, or order of functions
  - Comparing and classifying functions that ignores
    - *constant factors* and
    - *small inputs*.
- The Sets big oh O(g), big theta Θ(g), big omega Ω(g).
  - O(g(n)), Big-Oh of g of n, the Asymptotic Upper Bound;
  - Θ(g(n)), Theta of g of n, the Asymptotic Tight Bound; and
  - Ω(g(n)), Omega of g of n, the Asymptotic Lower Bound.

Doing the Analysis

- It's hard to estimate the running time exactly
  - Best case depends on the input
  - Average case is difficult to compute
  - So we usually focus on worst case analysis
    - Easier to compute
    - Usually close to the actual running time
- Strategy: find a function (an equation) that, for large n, is an upper bound to the actual function (actual number of steps, memory usage, etc.)



Asymptotic Analysis and Limits

To determine the relationship between functions f and g, it is often useful to examine

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = L$$

- The possible outcomes of this relationship, and their implications, follow:
  - **L = 0**: This means that g(n) grows at a faster rate than f(n), and hence that f = O(g) (indeed, f = o(g) and f$\neq$ Θ(g)).
  - **L = ∞:** This means that f (n) grows at a faster rate than g(n), and hence that f = Ω(g) (indeed, f = ω(g) and f $\neq$ Θ(g)).
  - **L $\neq$ 0 is finite:** This means that f (n) and g(n) grow at the same rate, to within a constant factor, and hence that f = Θ(g), or equivalently, g = Θ(f). Notice that this also means that f = O(g), g = O(f), f = Ω(g), and g = Ω(f).
  - **There is no limit:** In the case where $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)$ **= L** does not exist, this technique cannot be used to determine the asymptotic relationship between f(n) and g(n).

From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Rules for Analysis of Algorithms (1/3)

- **Fundamental operations execute in $\Theta(1)$ time:** Traditionally, it is assumed that “fundamental” operations require a constant amount of time (that is, a fixed number of computer “clock cycles”) to execute. Actually, it assumes that the running time of a fundamental operation is bounded by a constant, irrespective of the data being processed.

From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Rules for Analysis of Algorithms (2/3)

- **Arithmetic operations (+, , ×, /)** as applied to a constant number (typically two) of fixed-size operands.
- **Comparison operators (<,≤ , > ,≥ , = ,≠)** as applied to two fixed- size operands.
- **Logical operators (AND, OR, NOT, XOR)** as applied to a constant number of fixed-size operands.
- **Bitwise operations**, as applied to a constant number of fixed-size operands.
- **Conditional/branch operations.**

From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Rules for Analysis of Algorithms (3/3)

- **I/O operations** that are used to read or write a constant number of fixed-size data items. Note this does not include input from a keyboard, mouse, or other human-operated device, because the user's response time is unpredictable.
- **The evaluation of certain elementary functions**. Notice that such functions need to be considered carefully.
  - For example, when the function sin Θ is to be evaluated for "moderate-sized" values of Θ, it is reasonable to assume that $\Theta(1)$ time is required for each application of the function. However, for very large values of Θ, a loop dominating the calculation of sin Θ might require a significant number of operations before stabilizing at an accurate approximation. In this case, it might not be reasonable to assume $\Theta(1)$ time for this operation.

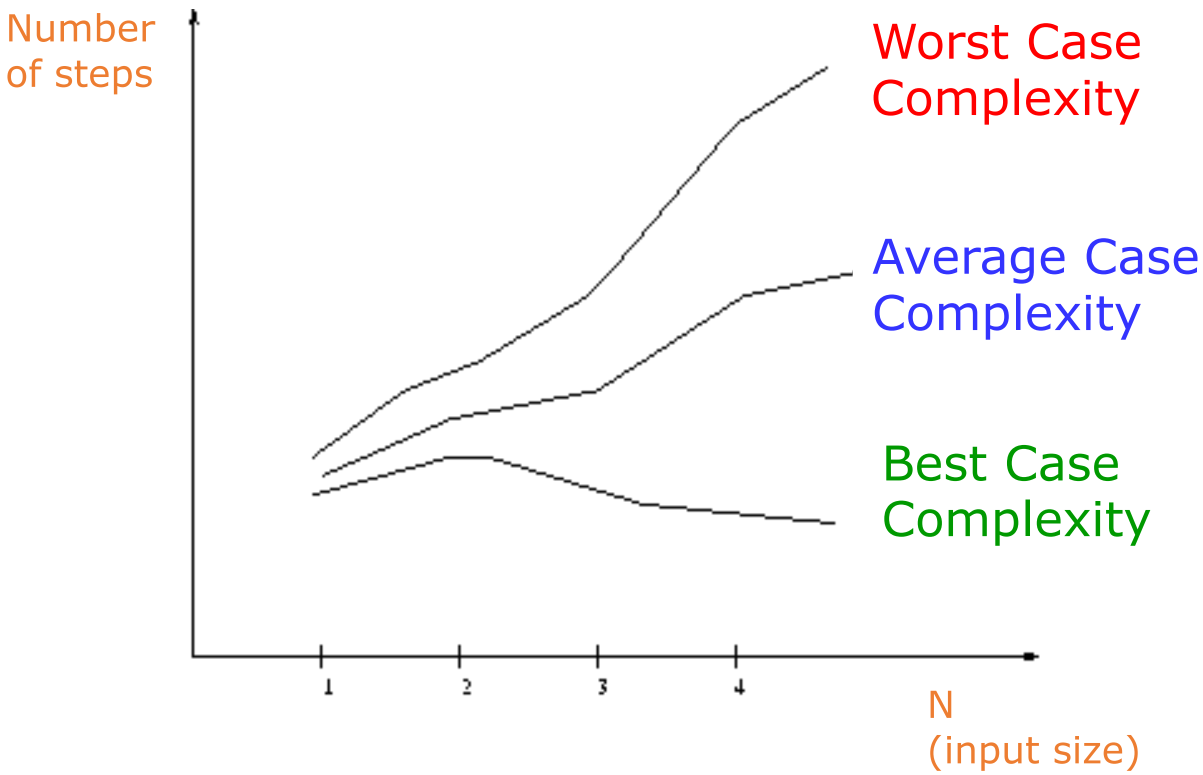
From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Summarizing Terminology

An algorithm with running time	is said to run in
$\Theta(1)$	constant time
$\Theta(\log n)$	logarithmic time
$O(\log^k n)$, k a positive integer	polylogarithmic time
$o(\log n)$	sublogarithmic time
$\Theta(n)$	linear time
$o(n)$	sublinear time
$\Theta(n^2)$	quadratic time
$O(f(n))$, where $f(n)$ is a polynomial	polynomial time

From : https://www.researchgate.net/publication/265962317_Algorithms_Sequential_Parallel_A_Unified_Approach

Workclass Exercises

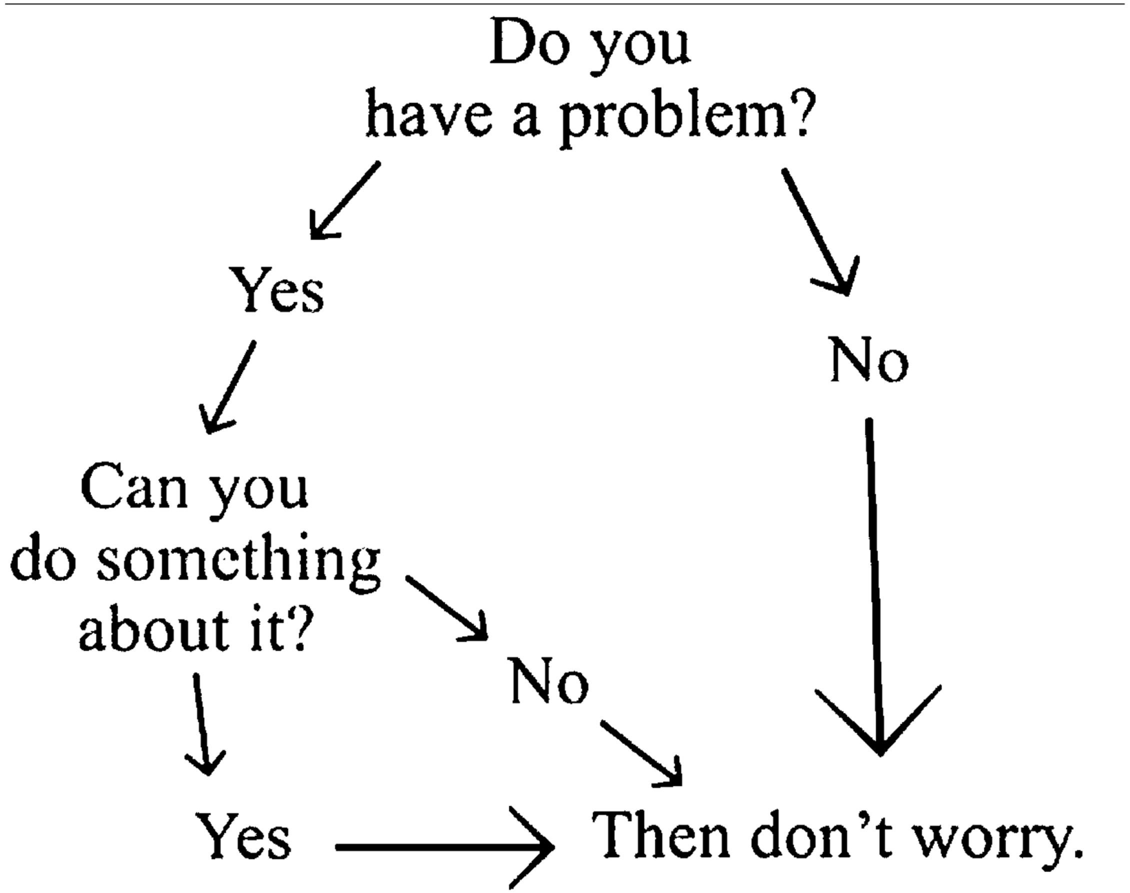


1. Propose and example for each one of the complexity cases, oberving the terminology of Today's class.
2. The sum of an erroneous compute of the first n powers of 2 (starting with zero) is given the by formula:

$$P(n) = 2^n-1$$

- (Not forget to propose pseudocode and flowchart)

Questions?



@carlosjaimebh