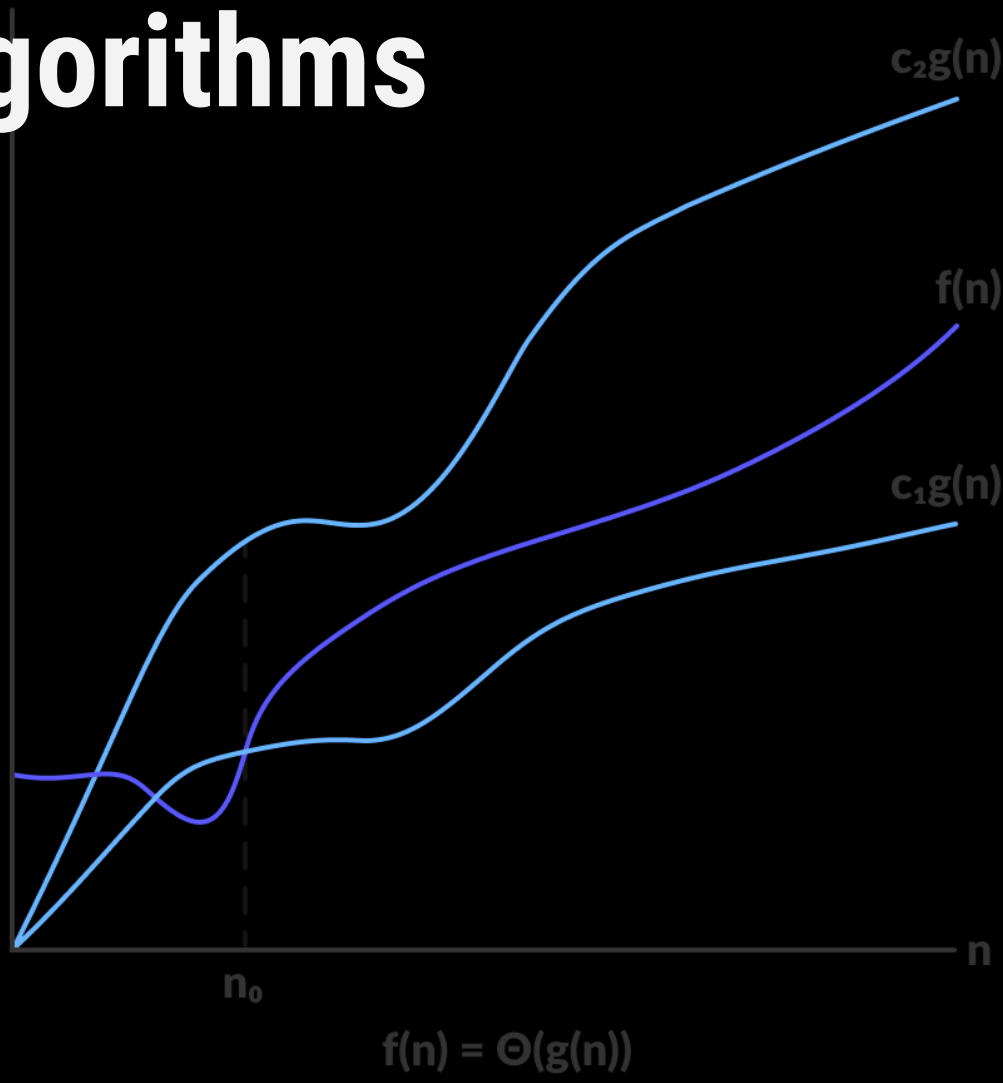
Analysis of Algorithms



Disclaimer / Descargo de Responsabilidad

Esta presentación corresponde a una guía usada por el profesor durante las clases. La misma ha sido modificada para ser utilizado en el modelo de cursos asistidos por tecnología. No es una versión final, por lo que la misma podría requerir todavía hacer algunos ajustes. Para aspectos de evaluación esta presentación es solo una guía, por lo que el estudiante debe profundizar con el material de lectura asignado y lo discutido en clases para aspectos de evaluación.

This presentation corresponds to a guide material used by the professor during classes. It has been modified to be used in the model of technology-assisted courses. It is not a final version, so it may still require some adjustments. For evaluation aspects, this presentation is only a guide, so the student should delve with the assigned reading material and what has been discussed in class.

What is an Algorithm?

- ➔ An algorithm is “a **finite** set of **precise** instructions for performing a computation or for solving a problem
- ➔ An algorithm is “a well-ordered collection of **unambiguous** and **effectively** computable operations that when executed produces a result and halts in a **finite** amount of time”
- ➔ A tool for solving a well-specified computational problem, for example, sorting a list.
- ➔ A recipe
- ➔ Described in three parts: input, process and output

What is an Algorithm?

- An algorithm is “a **finite** set of **precis**
performing a computation or for solvi
- An algorithm is “a well-ordered collec
effectively computable operations th
produces a result and halts in a **finite**
- A tool for solving a well-specified con
example, sorting a list.
- A recipe
- Described in three parts: input, proces

A computer program is the concrete representation of an algorithm in a programming language

How to represent an algorithm?

PSEUDOCODE

```
Input n
Set p to 1
While n is not equal to 0
    Reset p to p * n
    Reset n to n - 1
 End of While
Return p
```

NATURAL LANGUAGE

“the factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n”

FORMAL DEFINITION

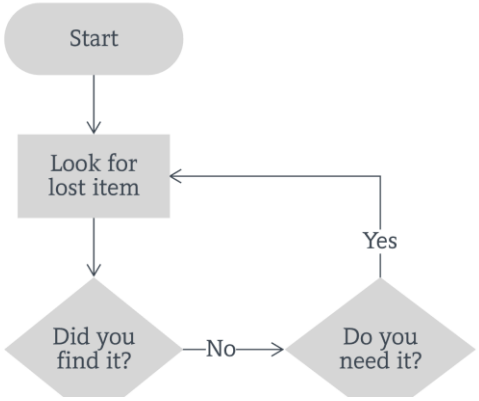
$$factorial : \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall i : \mathbb{N} \bullet$$

$$factorial(0) = 1$$

$$factorial(i+1) = (i+1) * factorial(i)$$

FLOW CHARTS



Characteristics of an Algorithm

- ➔ **Precise** and with a clear sequence of steps. Running the algorithm several times with the same input, should generate the same output.
- ➔ **Finiteness**: Terminate after a finite number of steps
- ➔ **Input/Output**: must receive zero or more inputs and produce at least one output
- ➔ Must be **unambiguous**: every step must be clear and have only one meaning
- ➔ **Effective**: each operation must be basic and do what is supposed to do in a finite time

Effectiveness vs Efficiency

- ➔ Effectiveness is about doing the right task, completing activities and achieving goals.
- ➔ Efficiency is about doing things in an **optimal way**, for example doing it the fastest or in the least expensive way.
- ➔ Algorithm efficiency is the property of an algorithm that indicates its **consumption of computational resources** such as CPU Time and Memory/Disk usage.

Algorithm Efficiency

➔ There are many measures for the efficiency of an algorithm. The most commons are time and space. Other include:

- Transmission speed
- Temporary disk usage
- Long-term disk usage
- Power consumption
- Response time

➔ For the rest of this presentation, we will focus on **time efficiency**, which is the most commonly used metric

Factors affecting run time of a program

- ➔ Computer System (**CPU**, **Memory**, **Disk**)
- ➔ Compiler
- ➔ Programing Language
- ➔ Operating System
- ➔ Developer
- ➔ Size of input
- ➔ Current applications running on the computer.

How to measure Algorithm Efficiency?

EMPIRICAL ANALYSIS

Is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it. Also known as *benchmark*

Very easy to understand and calculate

Results can vary depending on the hardware

Requires execution of the code

THEORETICAL ANALYSIS OF ALGORITHMS

Is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem

Platform independent. Provides a better way to predict performance.

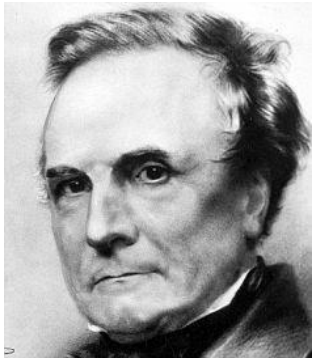
Hard to calculate and understand.

Calculated by just looking at the code.

What is theoretical algorithm analysis?

"As soon as an Analytic Engine exists, it will necessarily guide the future course of the science.Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?"

CHARLES BABBAGE



"It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process"

ALAN TURING

What is theoretical algorithm analysis?

- ➔ It is the process of finding the computational complexity of algorithms – the amount of time, storage, or other resources needed to execute them.

- ➔ Involves **determining a function** that relates the length of an algorithm's input to the number of steps it takes (its time complexity) or the number of storage locations it uses (its space complexity).

- ➔ An algorithm is said to be efficient when this function's values are small, or grow slowly compared to a growth in the size of the input.

What is theoretical algorithm anal

- ➔ It is the process of finding the compu algorithms – the amount of time, stor needed to execute them.
- ➔ Involves determining a function that r algorithm's input to the number of ste complexity) or the number of storage space complexity).
- ➔ An algorithm is said to be efficient wh are small, or grow slowly compared to the input.

Computational complexity theory focuses (part of computer science) on classifying computational problems according to their inherent difficulty, and relating these classes to each other.

Why analyze algorithms?

➔ Discover the characteristics of an algorithm in order to **evaluate its suitability** for various applications.

➔ **Compare** an algorithm with other algorithms for the same application

➔ Classify problems and algorithms by difficulty

➔ Predict performance, compare algorithms, **tune parameters**

➔ Better understand and improve implementations and algorithms

Approaches for theoretical algorithm analysis

1

2

COMPLEXITY AS A FUNCTION OF THE INPUT

Calculate a function of the input size by counting elementary instructions, focusing on the worst-case or the average-case

ASYMPTOTIC BEHAVIOR

Without calculating a detailed function of the input, focus on the behavior of the complexity for large n, that is on its *asymptotic behavior* when *n* tends to the infinity

Elementary Instructions

➔ Instructions that **always** takes the same time and this time is independent of the input size.

➔ We don't care about how many cycles the instruction needs to be executed

➔ **T** is a variable which depends on the architecture and hardware, **T** denotes the time required to execute the instruction.

Basic Arithmetic Operators (+, -, /, %, ^)

Bitwise operators (<< , >>)

Logical operators (==, !=, and, or, ~ ...)

Jumps (returns values, method calls, ...)

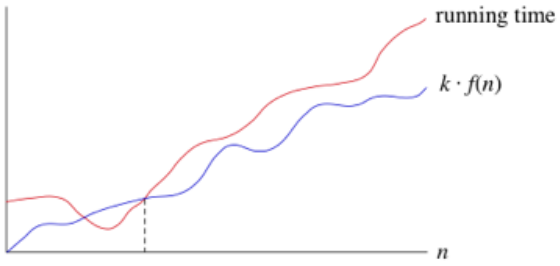
Assignments, access to indexed structures.

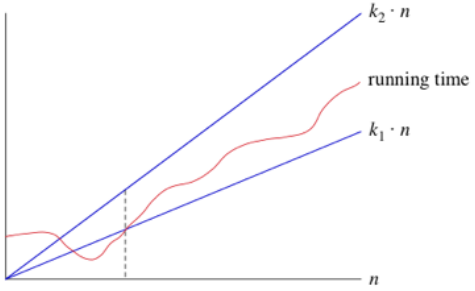
Instruction/Step Count

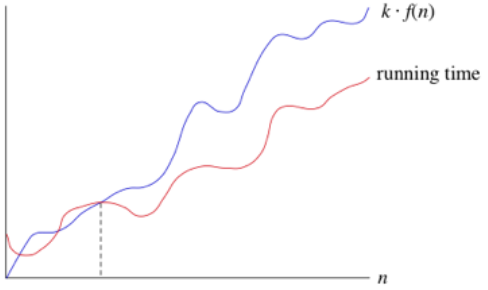
➔ Count the number of times each elementary instruction is executed. May be tedious to calculate the entire function.

➔ The objective is to calculate three functions that describe:

Best-case	Average-case	Worst-case
Time it takes to run with the optimal input	Time it takes to run for any possible input. Very hard to calculate	Time it takes to run with the worst possible input







Instruction/Step Count

➔ Let's calculate the worst-case:

```
// Compute the maximum element in the array a.
Algorithm max(a):
    max ← a[0]                                          ..... 3T
    for i = 1 to len(a)-1 ..... 2T ... (N-1)*T ... N*T .... T + 2NT
        if a[i] > max                                   ..... 2T(N-1)
            max ← a[i]                                  ..... 2T(N-1)
    return max                                          ..... T
```

Worst-Case:	T + 6NT

Worst-case time complexity gives an upper bound on time requirements and is often easy to compute. The drawback is that it’s often overly pessimistic.

Instruction/Step Count

➔ Let's calculate the best-case:

```
// Tell whether the array a contains x.
Algorithm contains(a, x):
    for i = 0 to len(a)-1 ..... 2T
        if x == a[i] ..... 2T
            return true ..... T
    return false
```

Best-Case:	5T
The best case is finding the element at the first position of the array	

Instruction/Step Count

➔ To calculate the **average case**, one needs to consider any possible input and the probability to appear.

➔ This is not usually calculated. Is not very straightforward and sometimes very hard.

➔ For example, to estimate average case for linear search in an unsorted list, let's assume that the probability to find an element *k* in an array of *n* size is distributed uniformly, meaning, each the probability to find it in any position of the array is the same:

$$p(x) = \frac{1}{n}$$

Instruction/Step Count

➔ So, the number of comparisons we have to do to find a value is:

$$\sum_{x=1}^{n} xp(x) = \sum_{x=1}^{n} \frac{x}{n} = \frac{1}{n} + \frac{2}{n} + \cdots + \frac{n}{n} = \frac{n+1}{2}$$

➔ As you can see, is not very easy to calculate. There are situations where the probability of the input is quite complex to calculate.

Instruction/Step Count

➔ Here are some tips to count instructions for common code structures (looking for worst-case):

#1 For a sequence of statements, sum the instruction count of each statement:

```
int i = 0;
boolean b = i == 1;
```

2T

3T

5T

Instruction/Step Count

➔ Here are some tips to count instructions for common code structures (looking for worst-case):

#2 For a conditional statement (like an if):

```
if (condition) {
        first-branch
} else {
        else-branch
}
```

Condition + Max(first-branch, else-branch)

Instruction/Step Count

➔ Here are some tips to count instructions for common code structures (looking for worst-case):

#3 For a for loop:

```
for (init; condition; increment) {
        for-body
}
```

init	**+**	**n+1 * condition**
	+	**n * body**
	+	**n+1 * increment**

Instruction/Step Count

➔ Here are some tips to count instructions for common code structures (looking for worst-case):

#3 For a for loop:

```
for (int i = 0; i < n; n++) {
        for-body
}
```

`int i = 1`	**Done one time**
`i < n`	**Done n + 1 times**
`i++`	**Done n times**
`for-body`	**Done n times**

Instruction/Step Count

➔ Some loops multiply or divide by two the control variable:

#4

```
i = 1;
while (i < 1000) {
        //code
        i = i * 2;
}
```

```
i = 1000;
while (i >= 1) {
        //code
        i = i / 2;
}
```

How many times is the loop body executed?

Complexity as a function of input

Instruction/Step Count

MULTIPLICATION LOOP	
Iteration	**i value**
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128
9	256
10	512

DIVIDE LOOP	
Iteration	**i value**
1	1000
2	500
3	250
4	125
5	62
6	31
7	15
8	7
9	3
10	1

Complexity as a function of input

Instruction/Step Count

MULTIPLICATION LOOP	DIVIDE LOOP
$2^{iterations} < 1000$	$\frac{1000}{2^{iterations}} \geqslant 1$

$$f(n) = log_2 n$$

Instruction/Step Count

MULTIPLICATION LOOP

$$2^{iterations} < 1000$$

DIVIDE LOOP

$$\frac{1000}{2^{iterations}} \geqslant 1$$

$$f(n) = log_2 n$$

In the computational context: $f(n) = \log n$ is taken as $f(n) = log_2 n$

Instruction/Step Count

➔ To count the instructions of nested loops, we multiply the outer loop by the inner loop

➔ There are several types of nested for loops, the most common include:

LINEAR LOGARITHM	QUADRATIC	CUBIC
$f(n) = n \log_2 n$	$f(n) = n^2$	$f(n) = n^3$

Asymptotic analysis

➔ Is counting steps really worth it? Let's say the function for a piece of code is:

$$f(x) = 2n^3 + 3n^2 + \log_{10} n + 333$$

➔ Which term of the function determines its growth?

Input size (n)	f(n)	$2n^3$	$3n^2$	$\log_{10}(n)$	333
1	338	**2**	3	0	333
10	2634	**$2x10^3$**	$3x10^2$	1	333
100	2030335	**$2x10^6$**	$3x10^4$	2	333
1000	2003000336	**$2x10^9$**	$3x10^6$	3	333
10000	2000300000337	**$2x10^{12}$**	$3x10^8$	4	333

Asymptotic Behavior

➔ Instead of looking for an accurate function, just focus on the growth rate of the algorithm

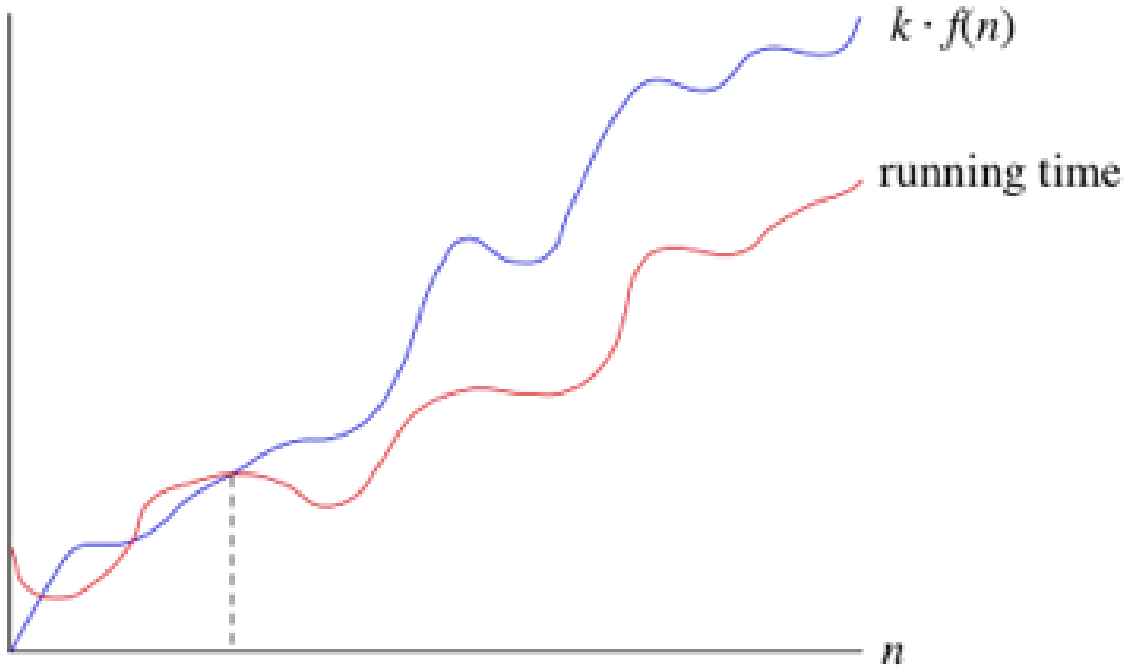
➔ Based on the table we saw before, we can just drop the terms that does not contribute significantly to the growth of the function:

$$f(x) = 2n^3 + 3n^2 + \log_{10} n + 333 \quad \bullet\bullet\bullet \quad f(x) = n^3$$

➔ Under this approach, there are three notations

Asymptotic Behavior: Big O

➔ Indicates the upper bound of the execution time of an algorithm



Upper bound. Run time won't get worse.

Asymptotic Behavior: Big O

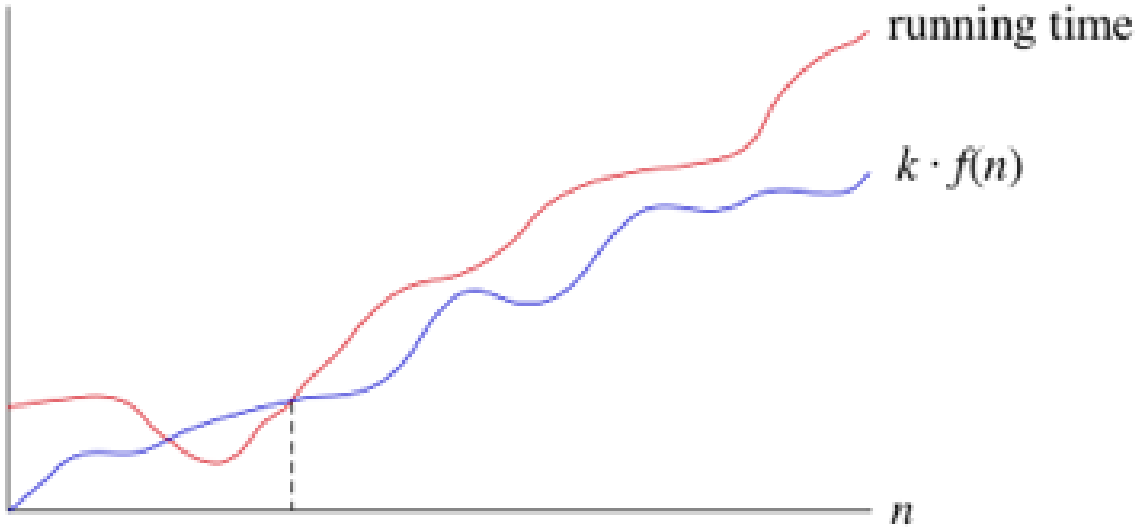
ALGORITHM FUNCTION	ALGORITHM FUNCTION IN BIG O
$f(n) = n^2 - 2n + 3$	$O(n^2)$

BIG O DEFINITION

Be *T(n)* a function of the input *n*, *T(n)* is *O(g(n))* if there is a $\mathbf{n_0}$ and a constant **c** that for all n >= $\mathbf{n_0}$ *T(n)* < *cg(n)*

Asymptotic Behavior: Big Ω

➔ Indicates the lower bound of the execution time of an algorithm



Lower bound. Run time will get worse.

Asymptotic Behavior: Big Ω

ALGORITHM FUNCTION	ALGORITHM FUNCTION IN BIG Ω
$f(n) = n^2 - 2n + 3$	$\Omega(n^2)$

BIG Ω DEFINITION

Be *T(n)* a function of the input *n*, *T(n)* is $\Omega(g(n))$ if there is a $\mathbf{n_0}$ and a constant **c** that for all n >= $\mathbf{n_0}$ (where n >= 1) *T(n) >= cg(n)*

Common Orders of Growth

1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Linear Logarithmic
n^2	Polynomial
n^3	
2^n	Exponential
$n!$	Factorial

Analysis of Algorithms