Grado en Ciencia e Ingeniería de Datos, 2018-2019

Unit 3.
Analysis of Algorithms

Algorithms and Data Structures (ADS)

Author: Isabel Segura-Bedmar

Index

- **Analysis of Algorithms**
- Empirical Analysis of Algorithms
- Theoretical Analysis of Algorithms

Analysis of Algorithms

- An **algorithm** is a set of steps (instructions) for solving a problem.
- Must be correct!!!.
- Must be efficient!!!

Analysis of Algorithms

- A problem can have several different solutions (**algorithms**)

- Goal: **choose the most efficient algorithm**

Analysis of Algorithms

- Study the performance of algorithms:
  - time complexity.
  - space complexity.
- Compare algorithms
- Focus on time: How to estimate the time required for an algorithm?

Index

- Analysis of Algorithms
- **Empirical Analysis of Algorithms**
- Theoretical Analysis of Algorithms

Empirical Analysis of Algorithms

1. Write the program
2. Include instructions to measure the running time
3. Run the program with inputs of different sizes
4. Plot the results

Empirical Analysis of Algorithms

Given a number *n*, develop a method to sum from 1 to *n*.

1. Write the program:

```
def sumOfN2(n):

   theSum = 0
   for i in range(1,n+1):
      theSum = theSum + i

   return theSum,end-start
```

Empirical Analysis of Algorithms

2. Include instructions to measure the running time

```
import time

def sumOfN2(n):
   start = time.time()

   theSum = 0
   for i in range(1,n+1):
      theSum = theSum + i

   end = time.time()

   return theSum,end-start
```

Empirical Analysis of Algorithms

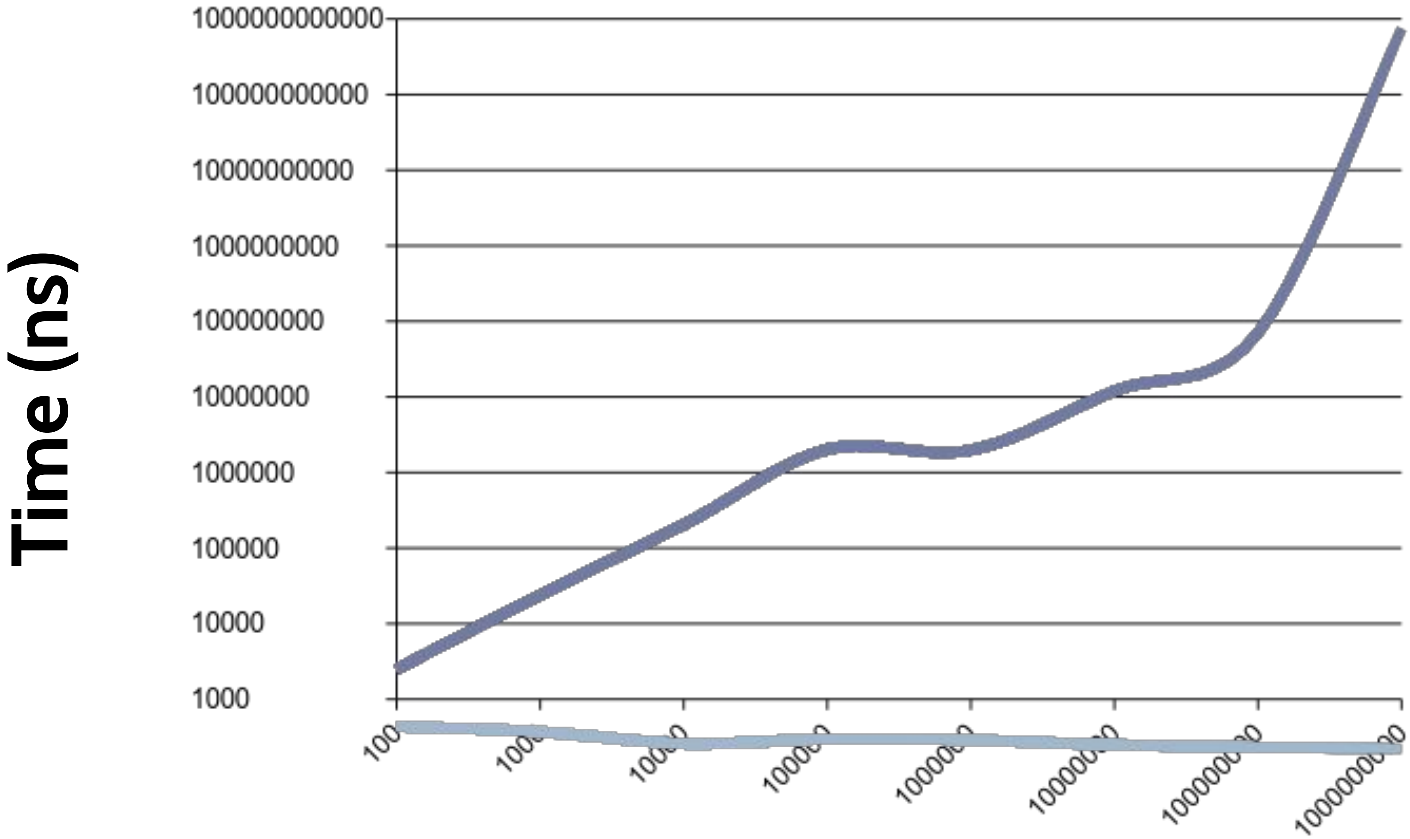
3. Run the program with inputs of different sizes

```
for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(10000))
```

```
Sum is 50005000 required  0.0004773 seconds
Sum is 50005000 required  0.0004287 seconds
Sum is 50005000 required  0.0004508 seconds
Sum is 50005000 required  0.0004337 seconds
Sum is 50005000 required  0.0004413 seconds
```

Empirical Analysis of Algorithms

4. Plot the results



Empirical Analysis of Algorithms

- When you need to show very large ranges (like in the previous example), use a Log-log plot
- **Log–log plot** uses logarithmic scales on both the horizontal and vertical axes.
- How can you make a log-log graph in excel?
  - In your XY (scatter) graph, double-click the scale of each axis.
  - In the Format Axis box, select the Scale tab, and then check Logarithmic scale

Empirical Analysis of Algorithms

Are there other algorithms that solve this problem?

Empirical Analysis of Algorithms

The Gauss's solution for adding numbers from 1 to n

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Nota: You can find an easy explication at :
http://mathandmultimedia.com/2010/09/15/sum-first-n-positive-integers/

Empirical Analysis of Algorithms

- Now, you can implement the Gauss’s solution
- Run the program for different values of n and measure the running time…
- Then, plot the result and compare it with the previous solution.

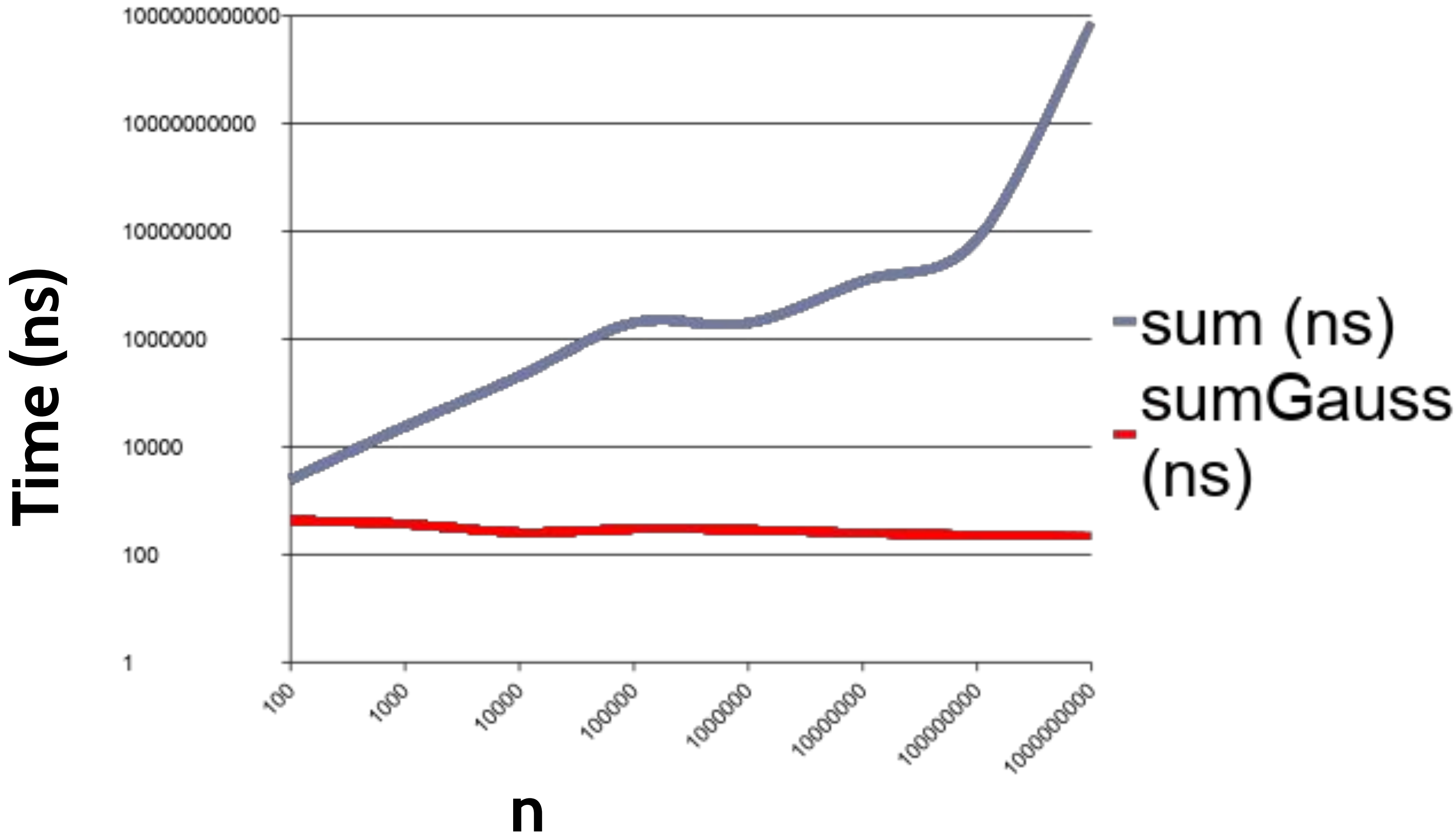
Empirical Analysis of Algorithms

```
def sumOfN2(n):
    return (n*(n+1))/2

print(sumOfN2(10))
```

n	time (ns)
100	436
1.000	371
10.000	259
100.000	298
1.000.000	290
10.000.000	250
100.000.000	233
1.000.000.000	222

Empirical Analysis of Algorithms



Empirical Analysis of Algorithms

- However, some disadvantages:
  - You need to implement the algorithms.
  - Same environment to compare two algorithms.
  - Results may not be indicative for other inputs

Index

- Analysis of Algorithms
- Empirical Analysis of Algorithms
- **Theoretical Analysis of Algorithms**
  - Running Time function (T(n)).
  - Big-O function

Theoretical Analysis of Algorithms

- Pseudocode
- Independent of the hardware/software environment
- Takes into account all possible inputs
- Define **T(n)**, running time function, which represents the running time of an algorithm, as a function of the input size

Theoretical Analysis of Algorithms

- Running time function T(n)
  - T(n)= **number of operations executed by an algorithm to process an input of size n.**

Theoretical Analysis of Algorithms

- Primitive operations take constant amount of time: *c*. (We can assume c=1 ns)
- Examples:
  - Assigning a value to a variable: *x*=2
  - Indexing into an array: *vector[3]*
  - Returning from a method: *return x*
  - Evaluating an arithmetic expression: *x+3*
  - Evaluating a logical expression:
    - *node!=None and i<index*

Theoretical Analysis of Algorithms

- If your function has **consecutive statements**:

B1
B2
..
Bn

Just add the running times of those consecutive statements. **T(n)=T(B1)+T(B2)+...+T(Bn)**

Theoretical Analysis of Algorithms

Example:

Algorithm *swap*(a,b)	**# operations**
`temp=a`	1
`a=b`	1
`b=temp`	1

T(n) = 3*c = 3, **(we assume c=1)**
This algorithm requires 3 nano seconds,
for an input of size n

Theoretical Analysis of Algorithms

- The running time of a **loop** is the running time of the statements inside of that loop times the number of iterations.

	# operations
`for i=1 to n`	n
`total=total+n`	(1+1)*n

Tfor(n) = (3n)*c = 3n, (we assume c=1)

The loop requires 3n nano seconds,
for an input of size n

Theoretical Analysis of Algorithms

- Time complexity of **nested loops** is equal to the number of times the innermost statement is executed.

 # operations

```
for i=1 to n                          n
    for j=1 to n                      n
    print(i*j)                        (1+1)n
```

$\left.\begin{matrix} n \\ (1+1)n \end{matrix}\right\} *n$

T(n) = n + n *(3n)=3n^2+n

Theoretical Analysis of Algorithms

- **If/Else**: As only one of the statements (S1,S2,...Sn) will be executed, we must consider the worst case (the most costly in time)

```
If condition1:
  S1
elif condition2:
  S2
  ...
else:
  Sn
```

Tif-else(n)= max(T(S1),T(S2),...,T(Sn))

Theoretical Analysis of Algorithms

- **If/Else:**

 # operations

```
if opc=0:
  x=0                     #S1          T(S1)=1
else:
  x=0
 for i=1 to n             #S2          T(S2)=1+n+2n=3n+1
   x=x+i
```

$$T_{\text{if-else}}(n) = \max(T(S1),T(S2)) = 3n+1$$

Theoretical Analysis of Algorithms

- Running time functions allow us to compare algorithms, without implementing them
- Let us compare the running time functions for *sumN* and *sumNGauss*.

Theoretical Analysis of Algorithms

Algorithm	# operations
Algorithm *sumN*(n)	**# operations**
total=0	1
for i=1 **to** n	n
total=total+n	2n
return total	1

Theoretical Analysis of Algorithms

```
Algorithm sumN(n)                # operations

 total=0

 for i=1 to n
total=total+n

 return total
```

T(n) = (3n + 2)*c = 3n+2, **(we assume c=1)**
This algorithm requires 3n+2 nano seconds,
for an input of size n

Theoretical Analysis of Algorithms

Algorithm *sumNGauss*(n)	**# operations**
return n*(n+1)/2	1+1+1+1

T(n) = 4*c = 4, **(we assume c=1)**

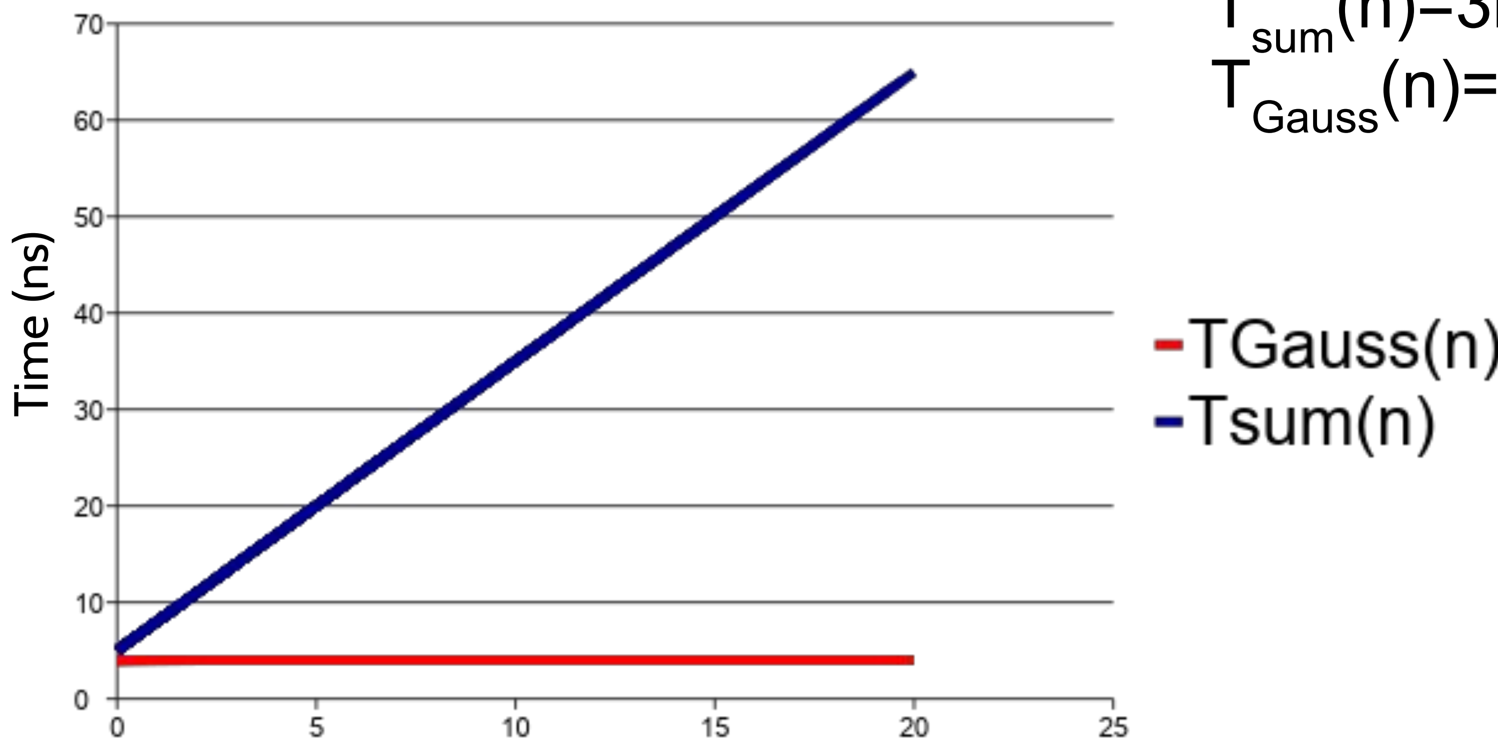
This algorithm requires 4 nano seconds,

for an input of size n

Theoretical Analysis of Algorithms

Time requirements as a function of the problem size n

$T_{sum}(n)=3n+2$
$T_{Gauss}(n)=4$



Theoretical Analysis of Algorithms

```
Algorithm contains(data,x)
    for c in data:
        if c==x:
            return False

    return False
```

What does T(n) depend on? Only depends on n?

Theoretical Analysis of Algorithms

```
Algorithm contains(data,x)
    for c in data:
        if c==x:
            return False

    return False
```

- Size of data,
- But also the value of x

Theoretical Analysis of Algorithms

- **Best-case:** the case that causes the **minimum number of operations** to be executed.
- **Worst-case:** the case that causes **the maximum number of operations** to be executed.
- **Average-case**: a case that requires the average number of operations to be executed. To known this average number, we must take all possible inputs and calculate their running times. Them, we sum them and divided by the total number of inputs.

Theoretical Analysis of Algorithms

- The **average case analysis is not easy to do** in most of the practical cases and it is rarely done.
- Most of the times, we do **worst case analysis** to analyze algorithms. We guarantee an upper bound on the running time of an algorithm.

Theoretical Analysis of Algorithms

```
Algorithm contains(data,x)
    for c in data:
        if c==x:
            return False

    return False
```

- Best case?
- Worst case?

Theoretical Analysis of Algorithms

```
Algorithm contains(data,x)
    for c in data:
        if c==x:
            return False

    return False
```

- Best case?: x is the first element
- Worst case?: x is the last or does not exist

Theoretical Analysis of Algorithms

- **For some algorithms**, all the cases are computationally same, i.e., **there are no worst and best cases** (T(n) will be 3n+2)

```
Algorithm sumList(data)
    total=0
    for c in data:
        total = total + c
    return total
```

Theoretical Analysis of Algorithms

- Running time depends on:
  - The **computer** on which the program is run
  - The **compiler** used to generate the program
- Find an approximation function for $T(n)$, an upper bound

Theoretical Analysis of Algorithms

- Running time depends on:
  - The **computer** on which the program is run
  - The **compiler** used to generate the program

- We must propose an approximation function for T(n), an upper bound.

Theoretical Analysis of Algorithms

- Suppose that you have two algorithms with the following running time functions:
  - $T_1(n)=3n^3$
  - $T_2(n)=3n^3+2n^2+3n+5$

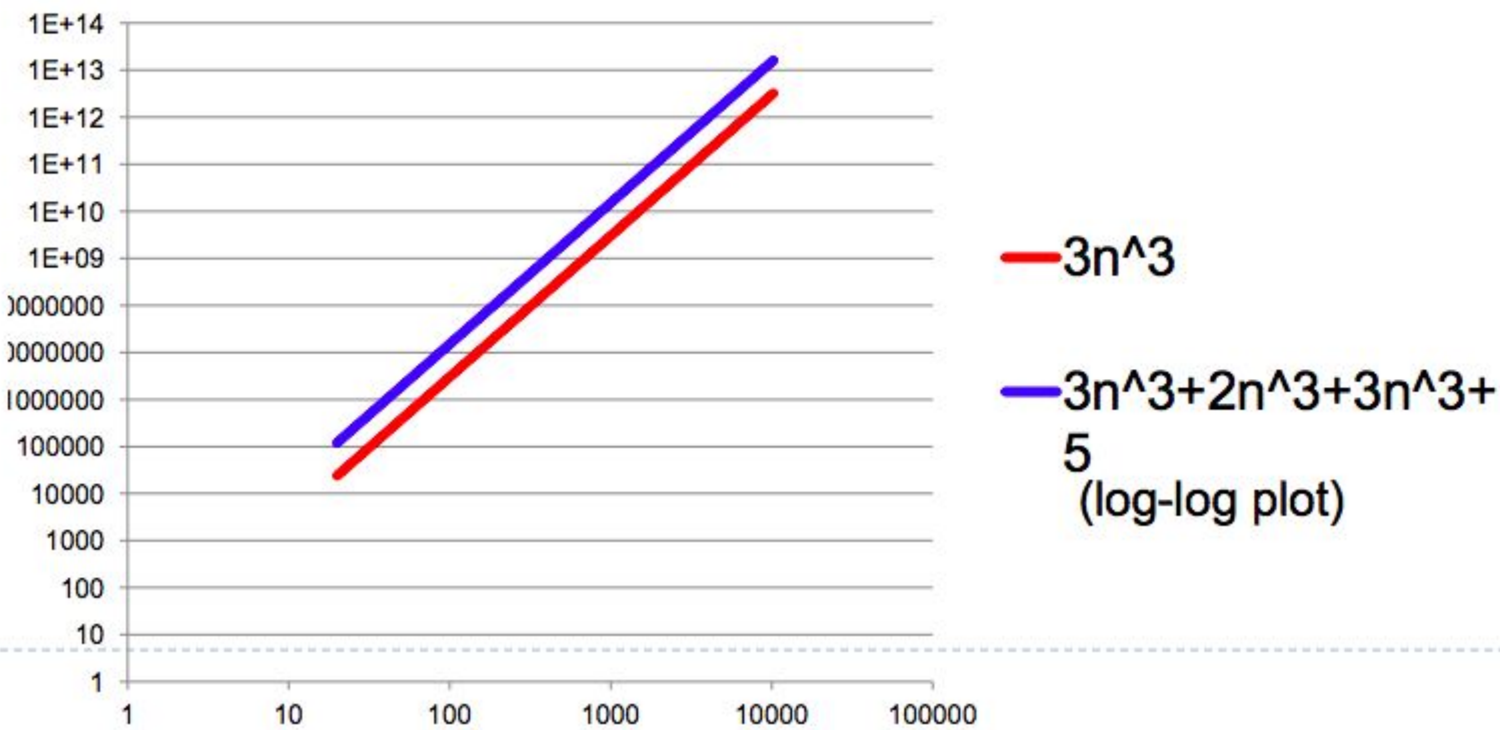
- What is the most efficient?

Theoretical Analysis of Algorithms

$T_1(n)=3n^3$

$T_2(n)=3n^3+2n^2+3n+5$

What is the most efficient?



Theoretical Analysis of Algorithms

Find an approximation function for T(n), an upper bound:

1. Find the fastest growing term.
2. Take out the coefficient.

Theoretical Analysis of Algorithms

Find an approximation function for T(n), an upper bound:

1. **Find the fastest growing term.**
2. Take out the coefficient.

$T_1(n)=3n^3$ -> $3n^3$

$T_2(n)=3n^3+2n^2+3n+5$ -> $3n^3$

Theoretical Analysis of Algorithms

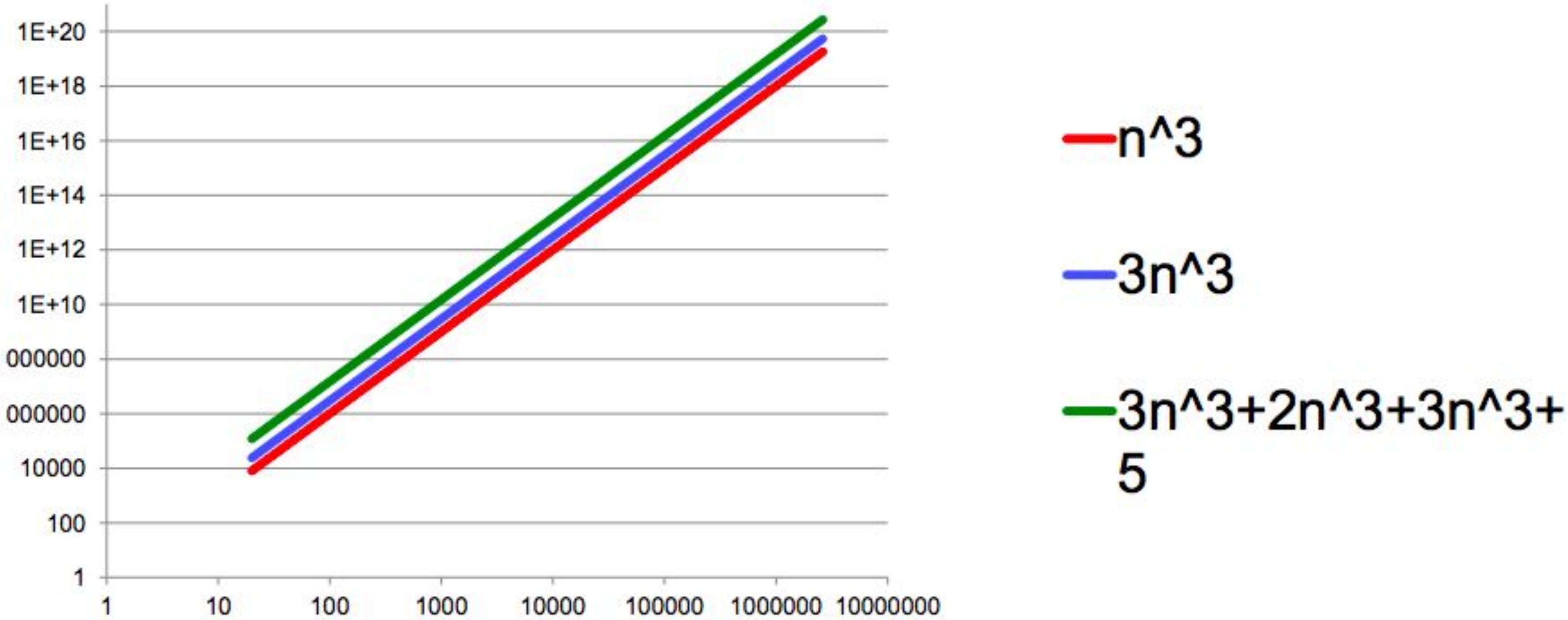
Find an approximation function for T(n), an upper bound:

1. **Find the fastest growing term.**
2. Take out the coefficient.

$T_1(n)=3n^3$ -> $3n^3$ -> n^3

$T_2(n)=3n^3+2n^2+3n+5$ -> $3n^3$ -> n^3

Theoretical Analysis of Algorithms



Theoretical Analysis of Algorithms

Big O functions

notation	**name**
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	linearithmic
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential
$O(n!)$	factorial

Theoretical Analysis of Algorithms

- Good news!!!: a small set of functions:

$$1 < \log n < n < n\log n < n^2 < n^3 < \ldots < 2^n < n!$$

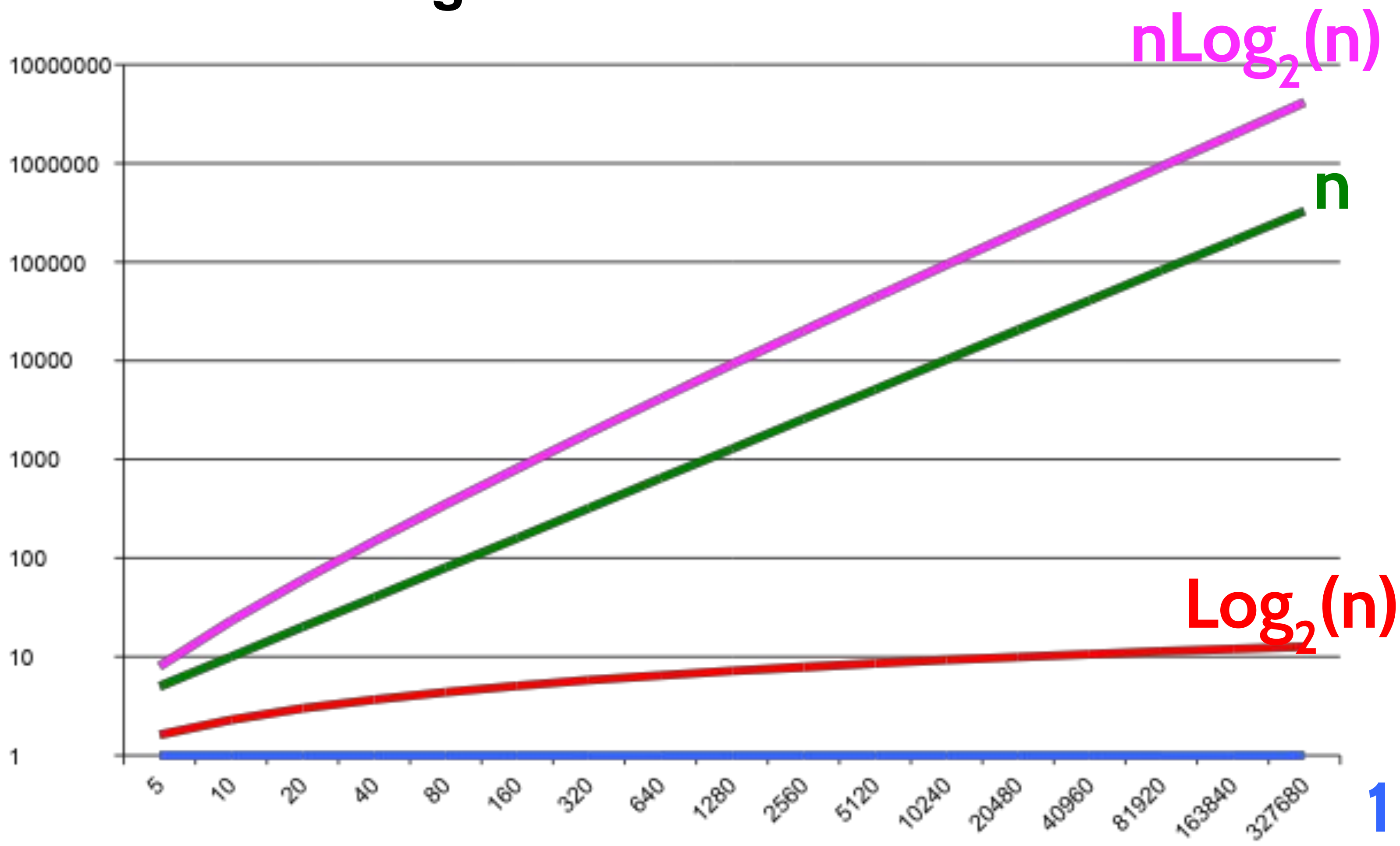
Theoretical Analysis of Algorithms

Efficient orders-of-growth:

Order	Name	Description	Example
1	Constant	Independent of the input size	Remove the first element from a queue
$Log_2(n)$	Logarithmic	Divide in half	Binary search
n	Linear	Loop	Sum of array elements
$nLog_2 (n)$	Linearithmic	Divide and conquer	Mergesort, quicksort

Theoretical Analysis of Algorithms

Efficient orders-of-growth:



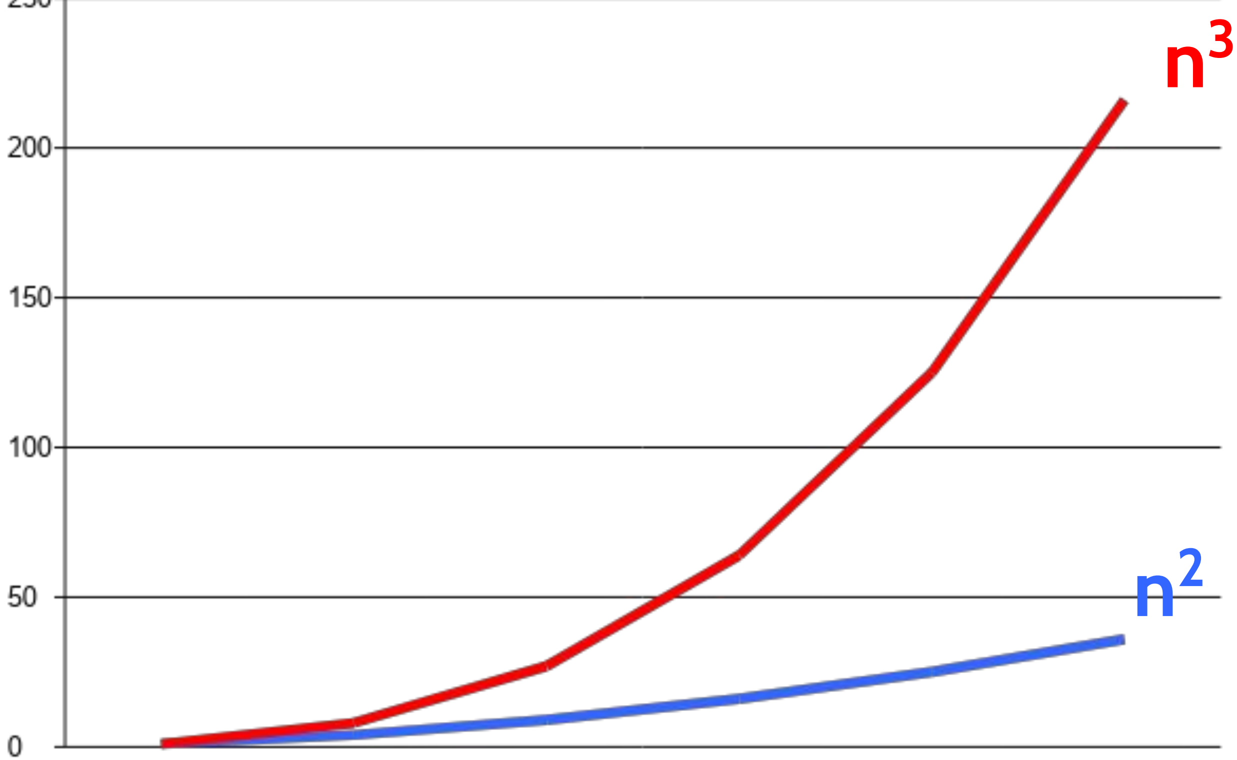
Theoretical Analysis of Algorithms

Tractable orders-of-growth:

Order	Name	Description	Example
n^2	Quadratic	Double loop	Add two matrices; bubble sort
n^3	Cubic	Triple loop	Multiply two matrices

Theoretical Analysis of Algorithms

Tractable orders-of-growth:



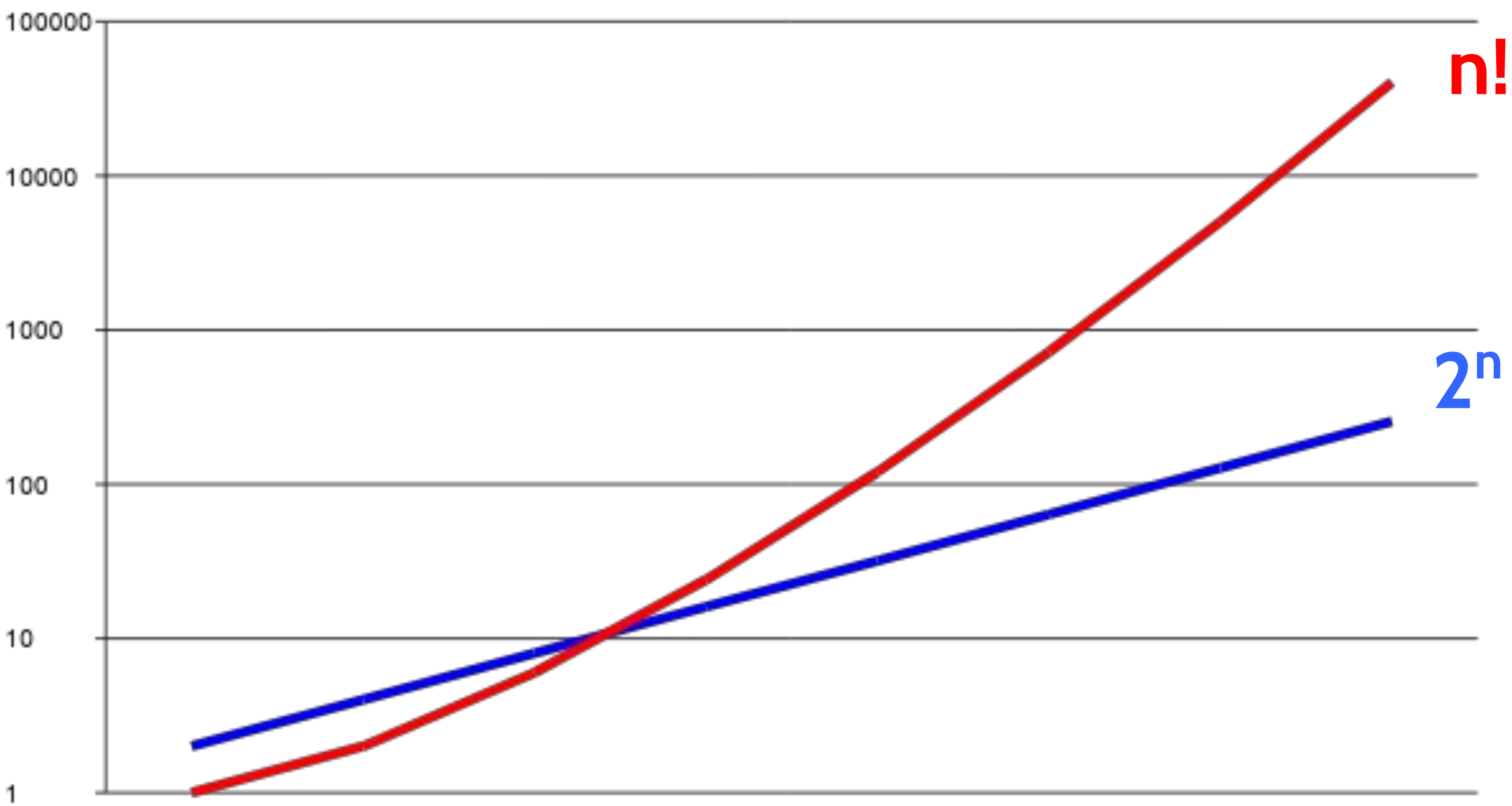
Theoretical Analysis of Algorithms

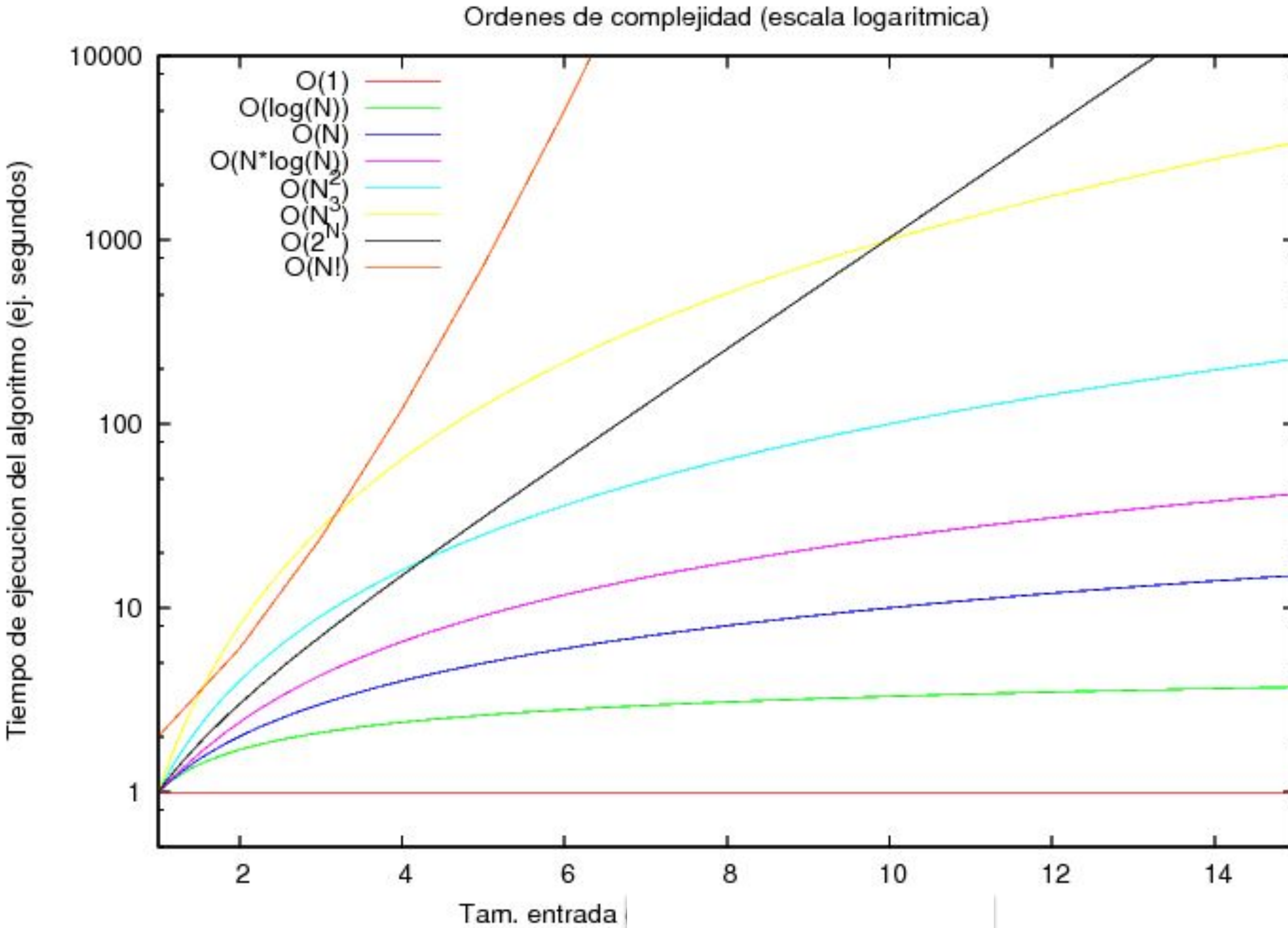
Intractable orders-of-growth:

Order	Name	Description	Example
k^n	Exponential	Exhaustive search	Guess a password,
n!	Factorial	Brute-force search	Enumerate all partitions of a set

Theoretical Analysis of Algorithms

Intractable orders-of-growth:





Ordenes de complejidad (escala logaritmica)
10000
1000
100
10
1
O(1)
O(log(N))
O(N)
O(N*log(N))
O(N^2)
O(N^3)
O(2^N)
O(N!)
Tiempo de ejecucion del algoritmo (ej. segundos)
2
4
6
8
10
12
14
Tam. entrada

Theoretical Analysis of Algorithms

Some examples:

T(n)	Big-O
$n + 2$	O(?)
$\frac{1}{2}(n+1)(n-1)$	O(?)
$3n+\log(n)$	O(?)
$n(n-1)$	O(?)
$7n^4+5n^2+1$	O(?)

Theoretical Analysis of Algorithms

T(n)	Big-O
$n + 2$	$O(n)$
$\frac{1}{2}(n+1)(n-1)$	$O(n^2)$
$3n+\log(n)$	$O(n)$
$n(n-1)$	$O(n^2)$
$7n^4+5n^2+1$	$O(n^4)$

Theoretical Analysis of Algorithms

More examples:

T(n)	BigO
4	O(?)
$3n+4$	O(?)
$5n^2 + 27n + 1005$	O(?)
$10n^3 + 2n^2 + 7n + 1$	O(?)
$n! + n^5$	O(?)

Theoretical Analysis of Algorithms

T(n)	Big-O
4	$O(1)$
$3n+4$	$O(n)$
$5n^2+ 27n + 1005$	$O(n^2)$
$10n^3+ 2n^2 + 7n + 1$	$O(n^3)$
$n!+ n^5$	$O(n!)$

Theoretical Analysis of Algorithms

Example: Calculate its T(n) and BigO functions. Discuss the worst and best cases

```
Algorithm findMax(data)
 max=-999999
 for c in data:
   if c>max then
     max=c
 return max
```

Theoretical Analysis of Algorithms

Example: Calculate its T(n) and BigO functions.
Discuss the worst and best cases

```
Algorithm findMax(data)
 max=-999999                        #1
 for c in data:                     #n
     if c>max then                  #1*n
        max=c                       #1*n
 return max                         #1
```

Answer:

T(n)=3n+2, O(n)=1

There are no worst and best cases, all the elements of data must be visited