Universidad Carlos III de Madrid

Unit 3
Analysis of Algorithms

Data Structures and Algorithms

Analysis of Algorithms

- An **algorithm** is a set of steps (instructions) for solving a problem.
- A problem can have several different solutions **algorithms**

  Sorting Algorithms:
  - Bubble Sort
  - Quick sort
  - Insertion Sort
  - Selection Sort
- Goal: **choose the most efficient algorithm**

Analysis of Algorithms

- Study the efficiency of algorithms:
  - time complexity.
  - space complexity.

- Focus on time: How to estimate the time required for an algorithm?

Analysis of Algorithms

to estimate the required time:

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

Empirical Analysis of Algorithms

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

Empirical Analysis of Algorithms

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

1. Write the program:

```
def sumOfN (n):

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

    return theSum
```

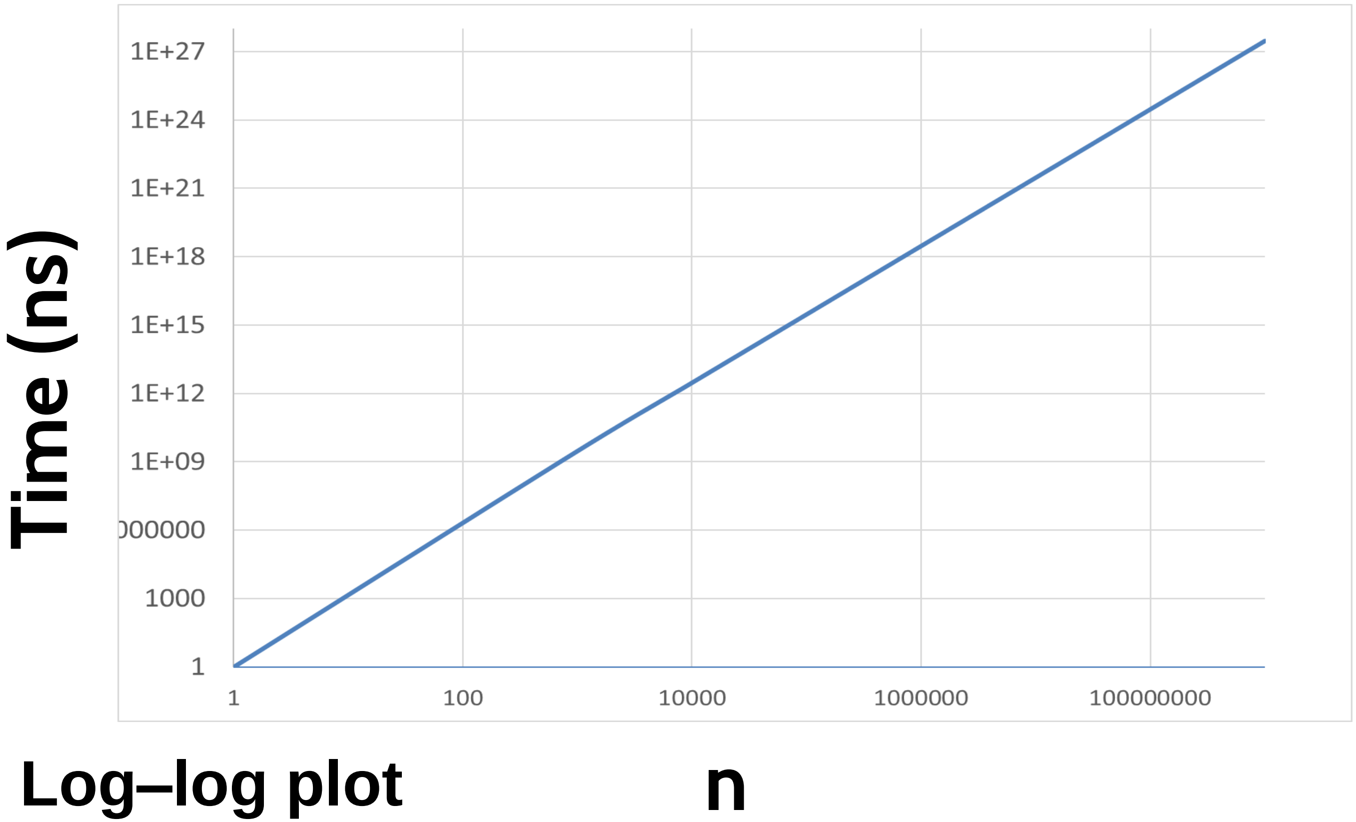
Empirical Analysis of Algorithms

2. Include instructions to measure the execution time

```
import time

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

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

   end = time.time()

   return theSum,end-start
```

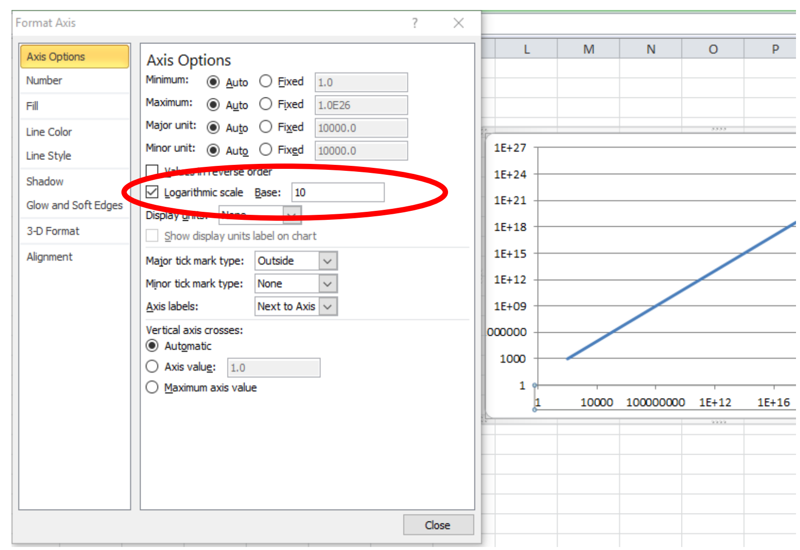
Empirical Analysis of Algorithms

3. Run the program with inputs of different sizes
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, Options, select Logarithmic scale



Empirical Analysis of Algorithms

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

Is there another algorithm to solve the same problem?

1+2+3+4+5 = 15

Empirical Analysis of Algorithms

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

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

$$\sum_{k=1}^{5} k = \frac{5(5+1)}{2} = \frac{30}{2} = 15$$

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

Empirical Analysis of Algorithms

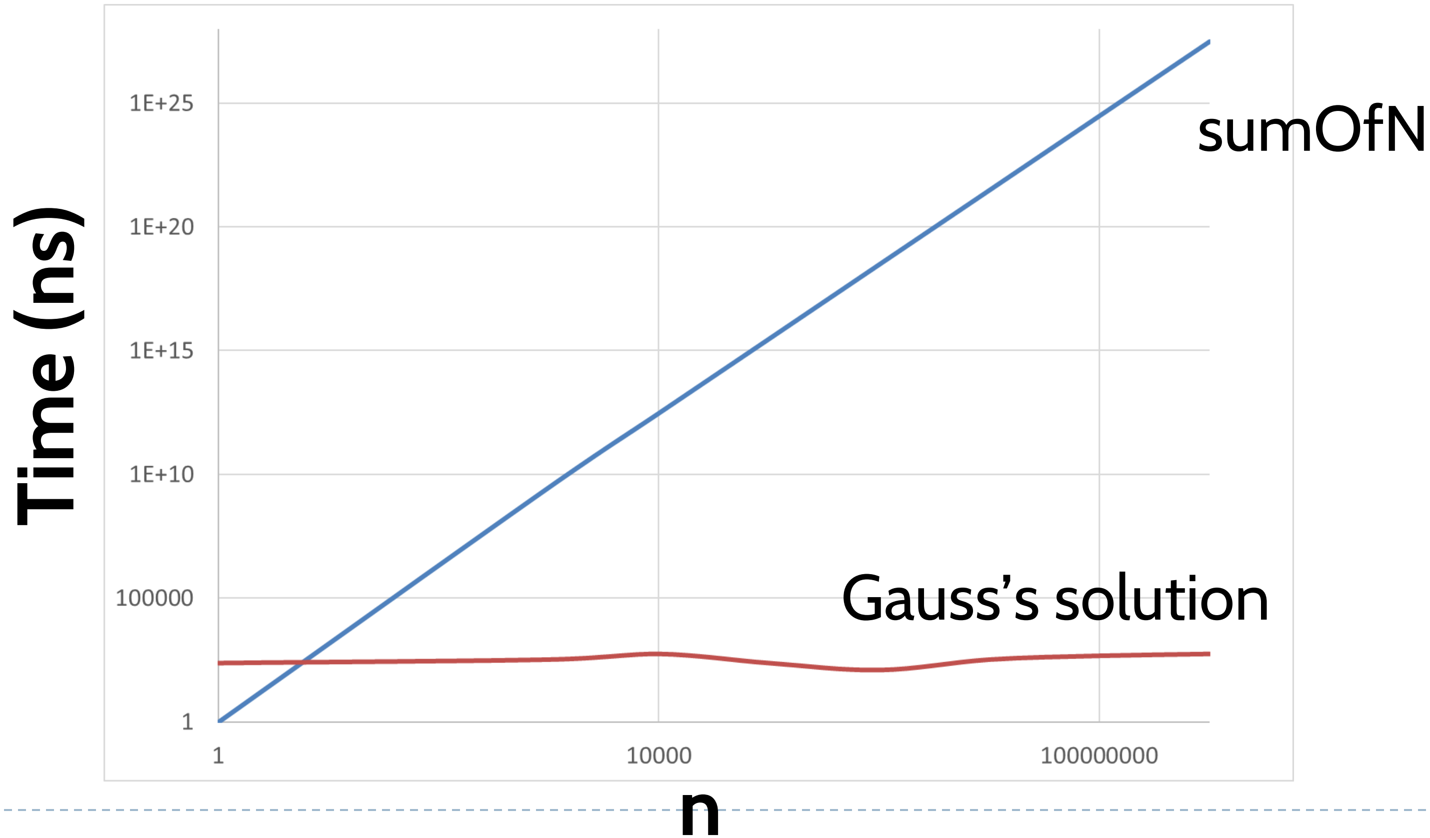
- Now, you can implement the Gauss's solution
- Run the program for different values of n and measure the running time…

```
import time

def sumOfN2(n):
    start = time.time()
    theSum = (n*(n+1))/2
    end = time.time()
    return theSum, end-start
```

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

- Then, plot the result and compare it with the previous solution



Empirical Analysis of Algorithms

Disadvantages:

- You need to implement the algorithms.
- Same environment to compare two algorithms.
- You can not conclude the time for other inputs not included in the experiment.

Analysis of Algorithms

to estimate the required time:

- Empirical Analysis of Algorithms
- **Theoretical Analysis of Algorithms**
  - **Running Time function.**
  - **Big-O notation**

Running time function T(n)

T(n) is the **number of operations executed by an algorithm** to process an input of size n.

Examples: $T_1(n) = 3n+5$, $T_2(n) = n^2-4n+6$

- Pseudocode
- Independent of the hardware/software environment
- Takes into account all possible inputs, whatever is the size of n.

Running time function T(n)

Primitive operations take constant time (1 nanoseconds)

Examples:

- Assigning a value to a variable: *x=2*
- Indexing into an array: *vector[3]*
- Returning from a method: *return x*
- Evaluating an arith. expression: *x+3*
- Evaluating a logical expression: *i<size*

Running time function T(n)

Consecutive statements:

- Just add the running times of those consecutive statements.
  T(n)=T(S1)+T(S2)+…+T(Sn)

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

T(n) = 3,
This algorithm requires 3 Nanoseconds,
for an input of size n

Running time function T(n)

Loop statements

- 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`	1*n
`    total=total+i`	(1+1)*n

T (n) = (n+2n) = 3n
The loop requires 3n Nanoseconds,
for an input of size n

Running time function T(n)

Nested loops

- Time complexity of **nested loops** is equal to the number of iterations of the outer loop times the number of iterations of the inner loop.

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

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

Running time function T(n)

If/else statements

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

 # operations

```
if opc=0:
    x=0                  1   } 1
else:
    x=0                  1   }
    for i=1 to n         n   } 3n+1
        x=x+i            2*n }
```

Running time function T(n)

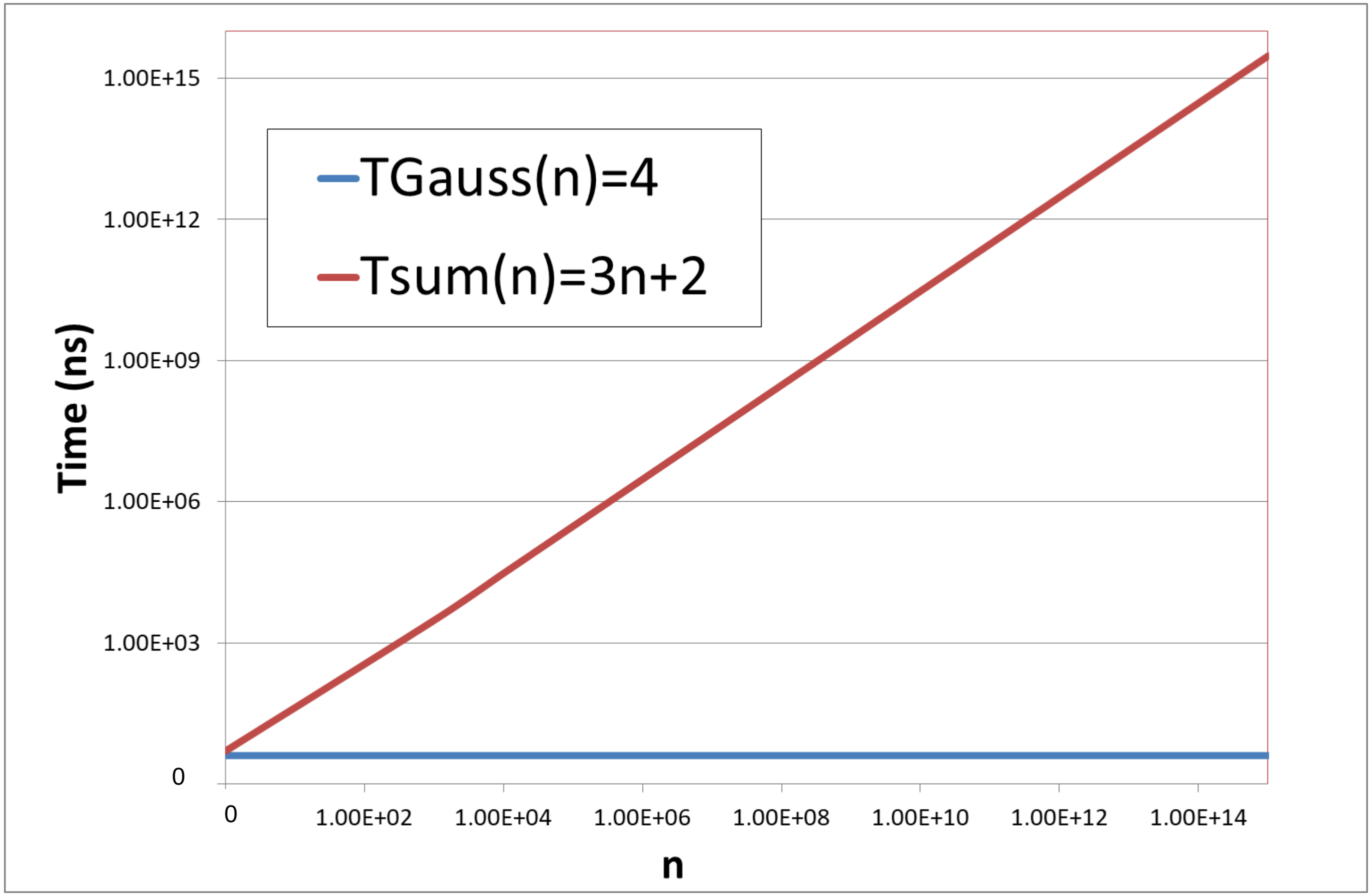
T(n) allows to compare algorithms without implementing them

Algorithm *sumN*(n)	**# operations**
total=0	1
for i=1 **to** n	n
total=total+i	(1+1)*n
return total	1
	3n+2

Algorithm *sumNGauss*(n)	**# operations**
return n*(n+1)/2	1+1+1+1
	4

Running time function T(n)

Time requirements as a function of the problem size n



Running time function T(n)

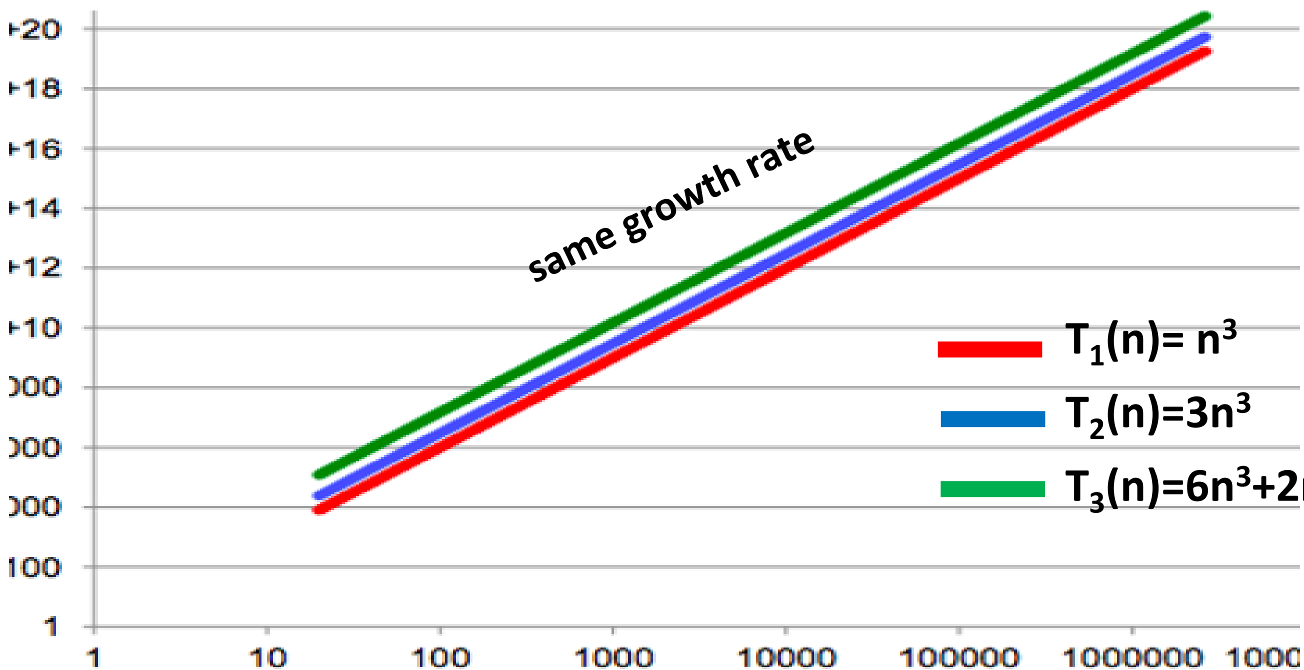
T(n) depends on: size of data, But also on the value of x

Algorithm *contains*(data,x)		
for c **in** data	**n**	
if (c==x)	**1*n**	**2n+1**
return True	**1**	
return False		

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

Big O notation

- Big O is used to describe the runtime tendency of an algorithm (**growth rate**)
- Different functions with the same growth rate may be represented using the same O notation.



$$\left.\begin{array}{l}T_1(n)= n^3\\T_2(n)=3n^3\\T_3(n)=6n^3+2n^2+3n+5\end{array}\right\}\in O(n^3)$$

growth rate is not affected by

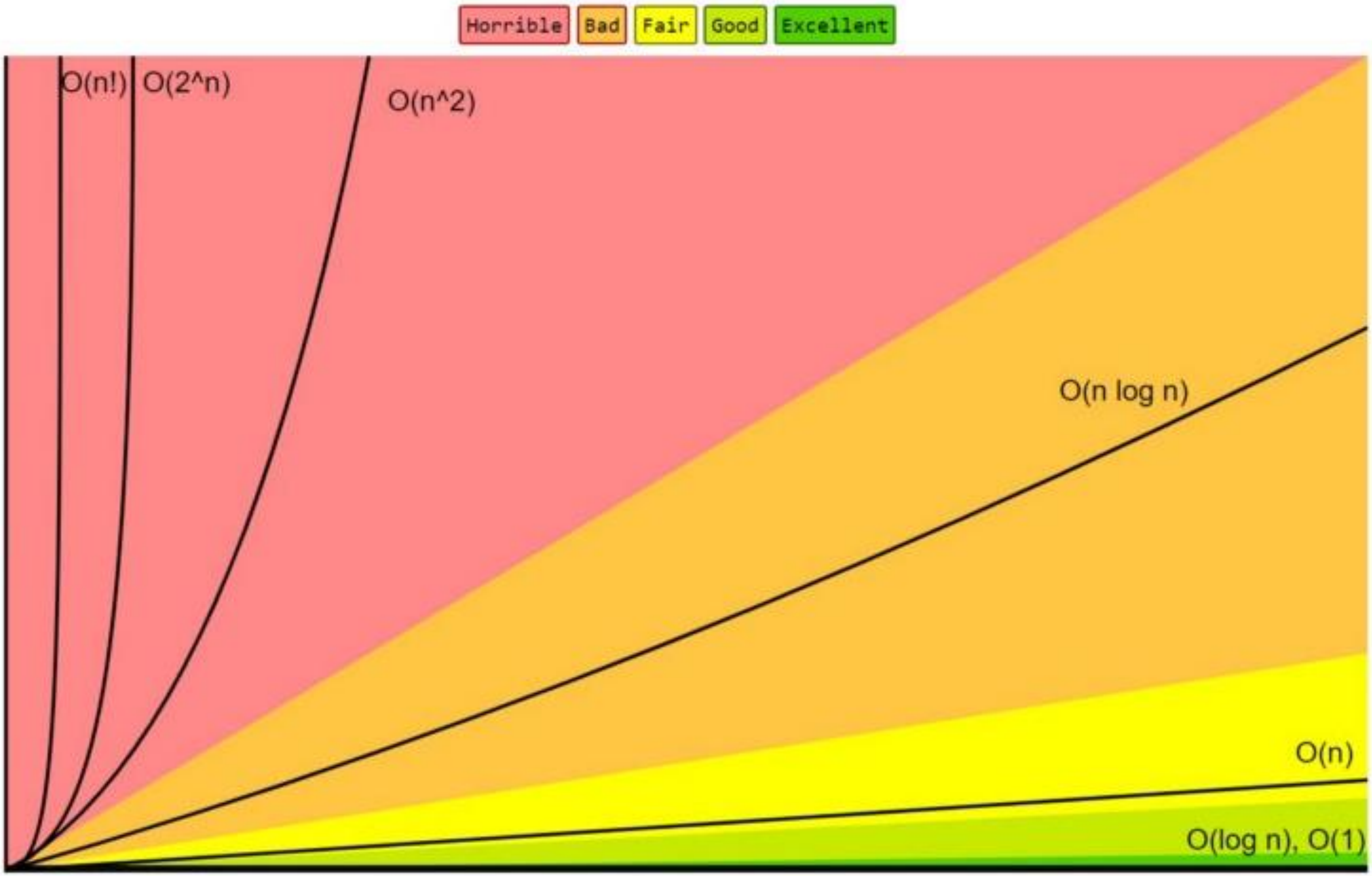
- Lower-order terms.
- Constant factors.

Big O notation

Efficient orders-of-growth (Big O):

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
N^2	Quadratic	Double loop	Add two matrices; bubble sort
N^3	Cubic	Triple loop	Multiply two matrices
k^n	Exponential	Exhaustive search	Guess a password,
n!	Factorial	Brute-force search	Enumerate all partitions of a set

Big O notation



https://www.freecodecamp.org/news/all-you-need-to-know-about-big-o-notation-to-crack-your-next-coding-interview-9d575e7eec4/

Big O notation

Some examples:

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

Big O notation

Some examples:

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

Big O notation

Some 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(?)

Big O notation

Some examples:

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

Big O notation

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:	1*n
max=c	1*n
return max	1