INFDEV036A – Algorithms Lesson unit 1

G. Costantini, F. Di Giacomo

costg@hr.nl, giacf@hr.nl – Office H4.206

Course description in a nutshell

- Why this course?
  - **Algorithms + Data structures = Program**
- Prerequisite
  - Object oriented programming
- Language for assignments (and practical exam)
  - C#
  - In the lessons mainly *pseudocode*

What is pseudo-code?

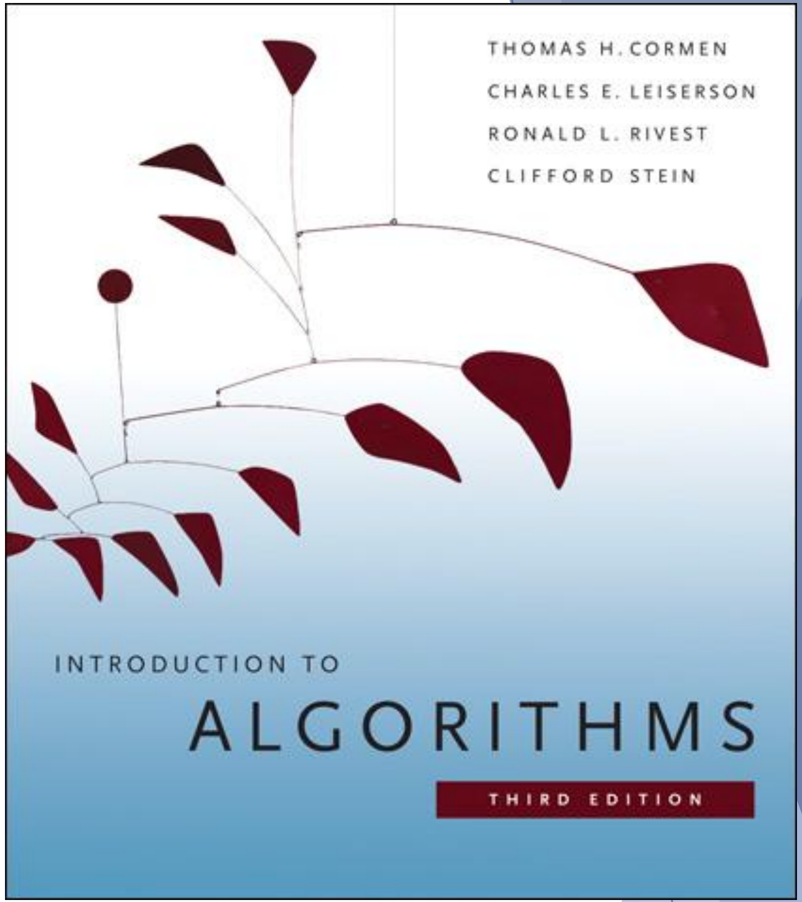
- Informal description of a computer program
  - does not actually obey the syntax rules of any particular language
  - omits non-essential details
  - can include natural language

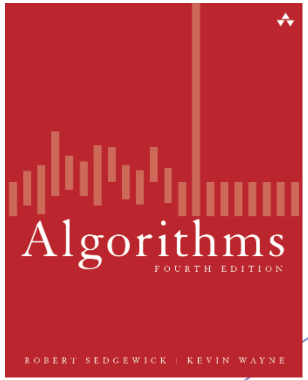
Pseudocode to Calculate the Sum & Average fo 10 Numbers

```
begin
        initialize counter to 0
        initialize accumulator to 0
        loop
                read input from keyboard
                accumulate input
                increment counter
        while counter < 10
        calculate average
        print sum
        print average
end
```

Literature

- All lesson materials (slides, mainly): on N@tschool
- MC questions: on GrandeOmega
- *Introduction to Algorithms*, T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, The MIT Press, ISBN: 978-0-262-53305-8, 3de editie, 2009
  - Complete and general
  - BIBLE OF ALGORITHMS AND EVERYTHING REMOTELY RELATED
- Another book (optional):
  - **Algorithms**, R. Sedgewick, K. Wayne, Addison Wesley, ISBN-13: 978-0321573513, 4th edition, 2011
  - Code and all examples in Java
  - http://algs4.cs.princeton.edu/





Assessment

- Made in two parts
  - **Written exam**
    - Multiple choice questions about reasoning on code and algorithms
    - **Must** be sufficient (≥ 5.5) **to be admitted** to the practical assessment
    - Every week, a set of questions on the topics covered is published on GrandeOmega
      - Exam questions will be similar to those
  - **Practical assessment**
    - Determines the final grade
    - Some exercises where you have to fill in code of some given partial algorithms related to the course
    - To help you practice…
      - Every week, implementation homework
      - Practical assignment (building algorithms in a realistic setting)

How do I pass the course (with a good grade)?

- Pay attention to the lessons

- Do all given homework (multiple times)
  - Study the slides
  - MC questions
  - Implementation exercises

Questions answered by the course

- Why is my code slow?
  - **Empirical and complexity analysis**
- How do I order my data?
  - **Sorting algorithms**
- How do I structure my data?
  - **Linear, tabular, recursive data structures**
- How do I represent relationship networks?
  - **Graphs**

Today

- Why is my code slow?
  - **Empirical and complexity analysis**
- How do I order my data?
  - **Sorting algorithms**
- How do I structure my data?
  - **Linear, tabular, recursive data structures**
- How do I represent relationship networks?
  - **Graphs**

More detailed agenda

- Intro
  - Recap on arrays
  - Our first (simple) algorithms, operating on arrays
- How to measure performance
  - Empirical analysis
  - Complexity analysis

Arrays: a quick summary

Definition, Basic manipulation & properties, Search algorithms

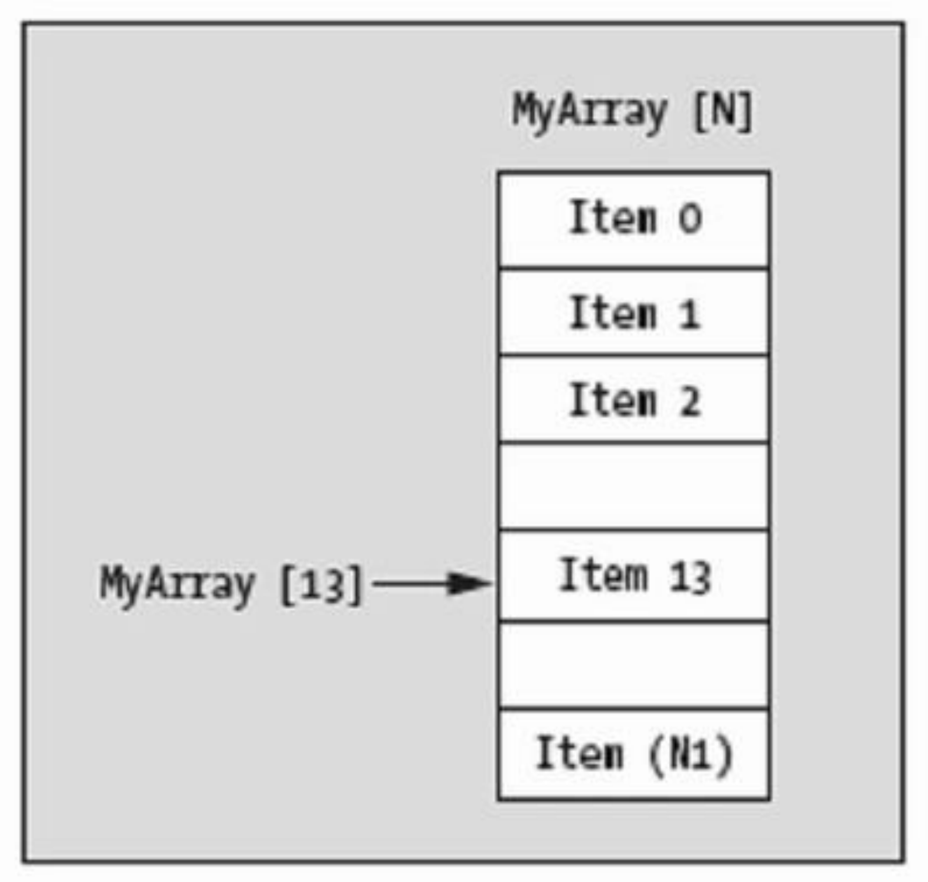
Array

- Definition?
  - Ordered list of values
  - Object that consists of a sequence of elements numbered 0, 1, 2, …
- Each value has a numeric index
  - Index number
  - Array of size N → indices from 0 to $N-1$

0	1	2	3	4	5	6	7	8	9
79	87	94	82	67	98	87	81	74	91

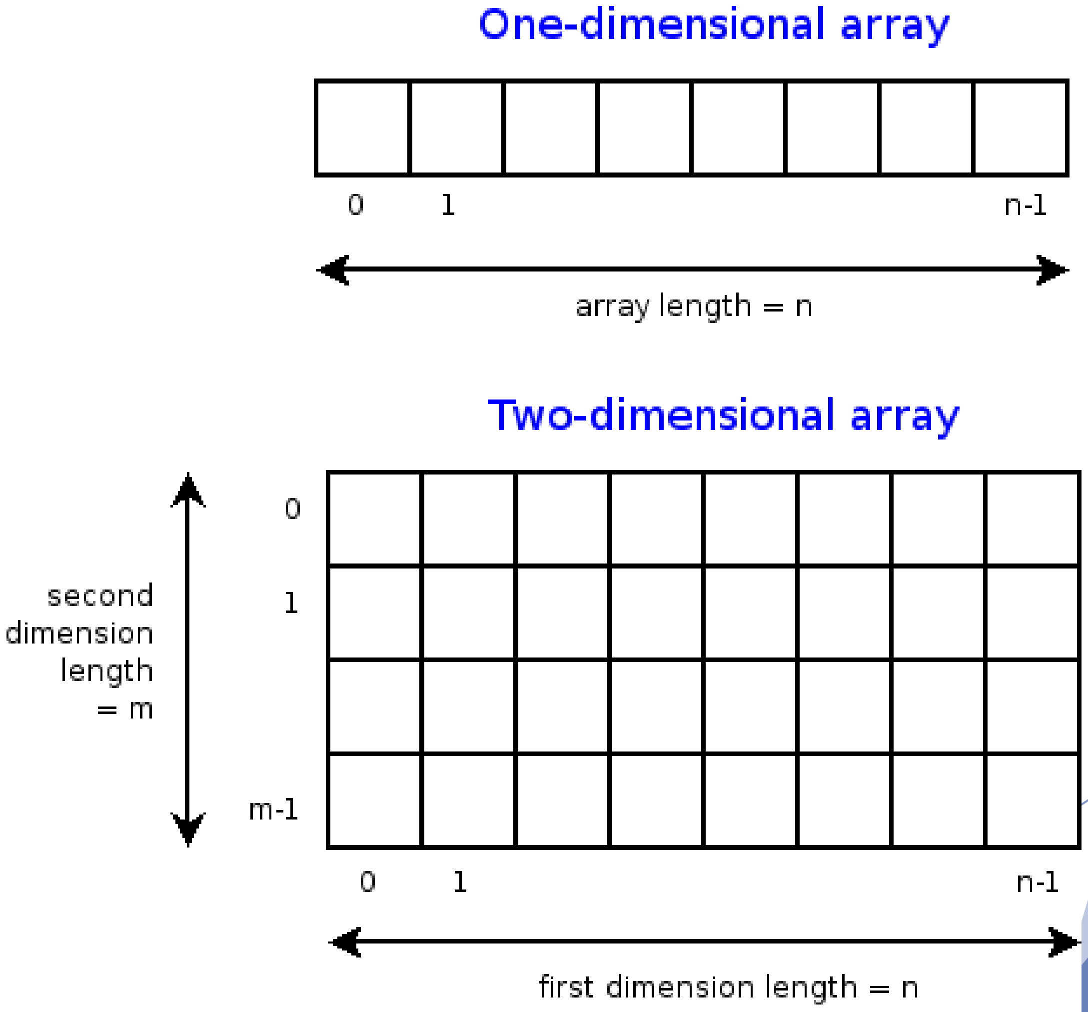
Array – Indexing notation

- Access to elements through their index
  - Usually done with the *subscript operator* **[]**
  - Very efficient because of cache alignment and tightness of representation (no additional data besides content)
    - NOT TRUE IN JAVA because of ref’s everywhere



Multidimensional arrays

- **Dimension:** do you know what it is?
  - number of indices needed to specify an element
- Many languages (i.e., Java) support only one-dimensional arrays
- Two-dimensional arrays
  - Access through two indices
  - $A[i, j]$
  - $int[\,,] \; A = new \; int[n, m];$



Array – Terminology, properties

- Components / Elements?
  - Values which compose the sequence
- Length (fixed)?
  - Number of components
- Bounds checking?
  - Usually, accessing the array outside its bounds $(0, N-1)$ raises an exception
- Origin?
  - First index
  - Some languages provide one-based array types (i.e., the first index is 1 and not 0!)

Array – Sequential search

- Also called *linear search*

- Simplest algorithm possible…
- … but also least efficient!
  - Trade-off: simplicity or performance?

- Examine each element **sequentially**, from the first one to the end of the array
  - Similar to looking for a passenger in a moving train

Array – Sequential search

- Pseudo-code
  - Look for the value v in the array a
  - Return -1 if v is not found

```
FOR i = 0 TO N-1
  IF a[i] = v
    RETURN i
RETURN -1
```

Array - Sequential search

```
FOR i = 0 TO N-1
  IF a[i] = v
    RETURN i
RETURN -1
```

- **Correctness**
  - Why does it work FOR SURE?
  - Principle of *Mathematical Induction*
    - To prove that the loop invariant is true at *every* iteration
    - True at iteration 0; If true at iteration i → true also at iteration $i+1$
    - Here the invariant is "v is not contained in $a[0 \dots i-1]$"
  - Not a big focus on correctness in this course

Array - Sequential search

```
FOR i = 0 TO N-1
  IF a[i] = v
    RETURN i
RETURN -1
```

- **Correctness**
  - Why does it work FOR SURE?
  - Principle of *Mathematical Induction*
    - To prove that the loop invariant is true at *every* iteration
    - True at iteration 0; If true at iteration i → true also at iteration $i + 1$
    - Here the invariant is "v is not contained in $a[0 \ldots i - 1]$"
  - Not a big focus on correctness in this course
- **Performance** (only intuition now… details later)
  - Array of 10 elements → max. 10 iterations
  - Array of 20 elements → max. 20 iterations
  - Array of 100 elements → max. 100 iterations
  - … on average, running time proportional to the number of elements in the array

Array – Binary search

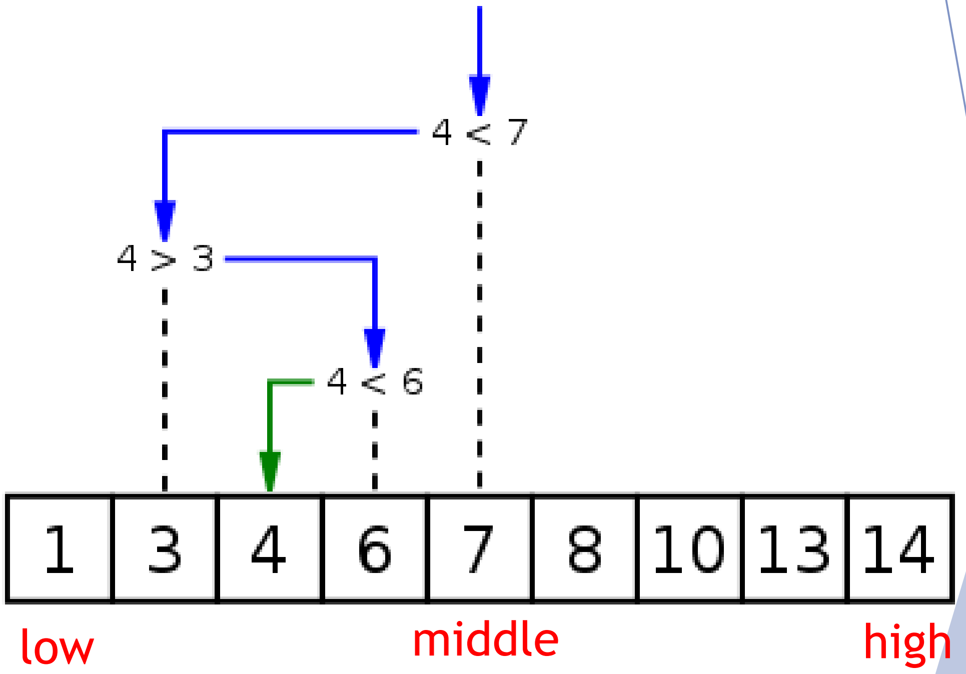
- Standard search algorithm for a **SORTED** sequence
  - More efficient than sequential search
  - Requires the order of elements

- Basic idea: divide the sequence in two and focus on the half which could contain the element
  - Application example: looking up a word in a dictionary

Array – Binary search

- Pseudo-code [iterative version]
  - Look for the value v in the array a
  - Return -1 if v is not found

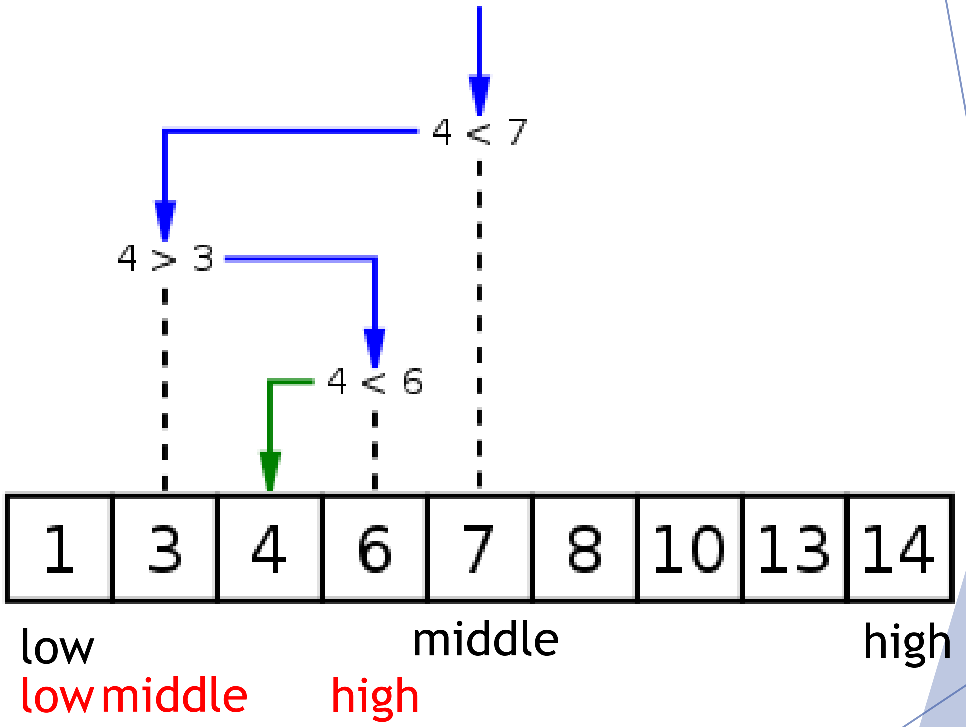
```
low = 0; high = N-1
WHILE low <= high
  middle = (low + high) / 2
  IF v < a[middle]
    high = middle – 1
  ELSE IF v > a[middle]
    low = middle + 1
  ELSE
    RETURN middle
RETURN -1
```



Array – Binary search

- Pseudo-code [iterative version]
  - Look for the value v in the array a
  - Return -1 if v is not found

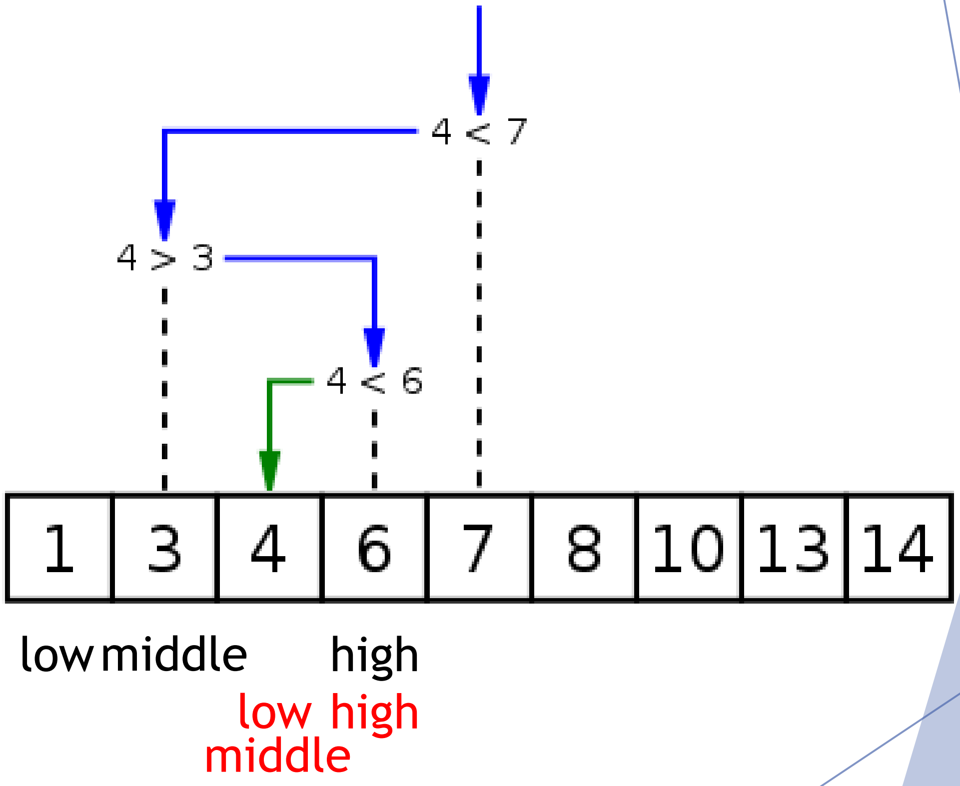
```
low = 0; high = N-1
WHILE low <= high
  middle = (low + high) / 2
  IF v < a[middle]
    high = middle – 1
  ELSE IF v > a[middle]
    low = middle + 1
  ELSE
    RETURN middle
RETURN -1
```



Array – Binary search

- Pseudo-code [iterative version]
  - Look for the value v in the array a
  - Return -1 if v is not found

```
low = 0; high = N-1
WHILE low <= high
  middle = (low + high) / 2
  IF v < a[middle]
    high = middle – 1
  ELSE IF v > a[middle]
    low = middle + 1
  ELSE
    RETURN middle
RETURN -1
```



Array – Binary search

- Pseudo-code [recursive version]
  - Look for the value v in the array a
  - Return -1 if v is not found
  - First call?

  `BinSearch(a, 0, N-1, v)`

```
BinSearch(a, low, high, v)
  IF low > high
    RETURN -1
  middle = (low + high) / 2
  IF a[middle] > v
    BinSearch(a, low, middle – 1, v)
  ELSE IF a[middle] < v
    BinSearch(a, middle + 1, high, v)
  ELSE
    RETURN middle
```

Array – Binary search

- Performance
  - More complex to determine than in linear search
  - Given the number of elements `N` in the array, how many iterations will be done *at most* by the loop?

Performance of algorithms

Empirical analysis; Complexity analysis

Studying algorithms

- Intuition
  - **How** does it work?
- Invariant (*correctness*)
  - **Why** does it work? What are the fundamental properties that guarantee the correct answer?
- ***Complexity***
  - **How fast** is it, and how does it scale to very large inputs?
    - Through observation … *Empirical analysis*
    - Through reasoning … *Complexity analysis*

Empirical analysis

Empirical analysis

- How to make quantitative measurements of the running time of our programs?
  - Using the Stopwatch!

```
public class Stopwatch
           Stopwatch()        create a stopwatch
double     elapsedTime()      return elapsed time since creation
```

	Name	Description
	Elapsed	Gets the total elapsed time measured by the current instance.
	ElapsedMilliseconds	Gets the total elapsed time measured by the current instance, in milliseconds.

- If we execute a program more than once and/or on different machines, will it always have the same running time?
  - **No!!!** It depends on…
    - The PC on which it is executed
    - The "problem size"

Empirical analysis

- More interesting question:

  “How much does the running time of a program increase when the problem size increases?”

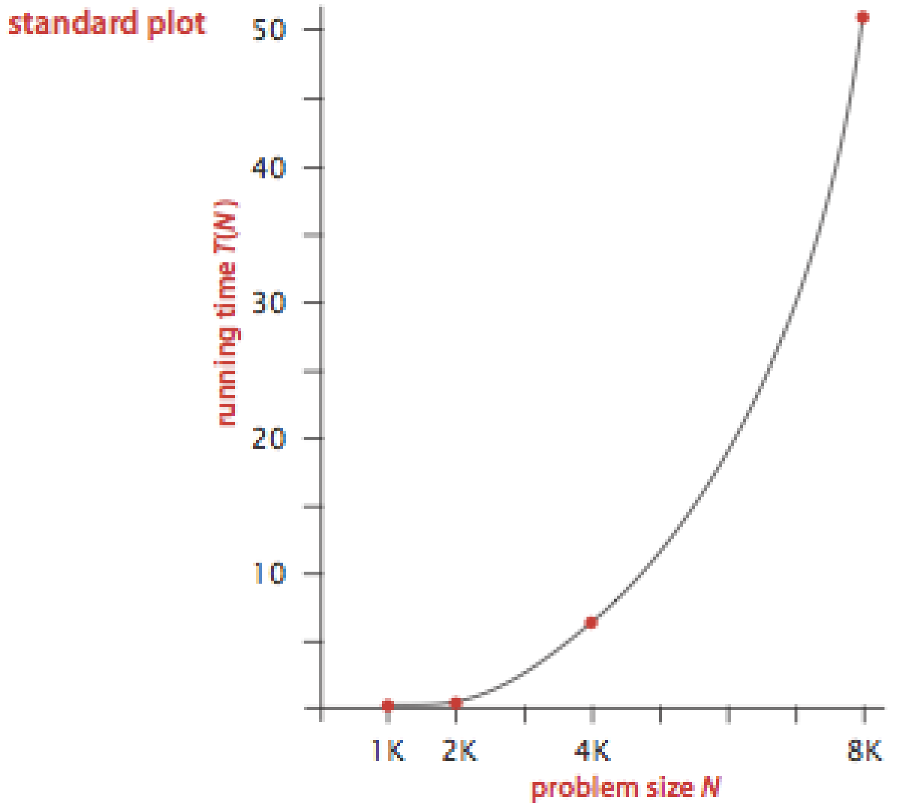
- We look for a dependency/relationship between
  - Problem size
  - Running time

Empirical analysis

- Example
  - a program (*ThreeSum*) which counts the triples in an array of N integers that sum to 0
- Question
  - What is the relationship between the problem size N and the running time of ThreeSum?
- Emipirical observations
  - N = 1000 → 0.1 seconds
  - N = 2000 → 0.8 seconds
  - N = 4000 → 6.4 seconds
  - N = 8000 → 51.1 seconds
  - …

Empirical analysis

- What can we do with the running times collected?
  - Plot them and try to infer the equation of the function
    - In this case, cubic relationship: $T(N) = aN^3$
  - We can use such function to make predictions (and then to validate them)



Empirical analysis

- To get information on the performance of an algorithm, do we **need** to use the Stopwatch?
  - No!

- It is possible to describe the running time of a program independently of concrete execution, by determining the frequency of execution of statements
  - Complexity analysis

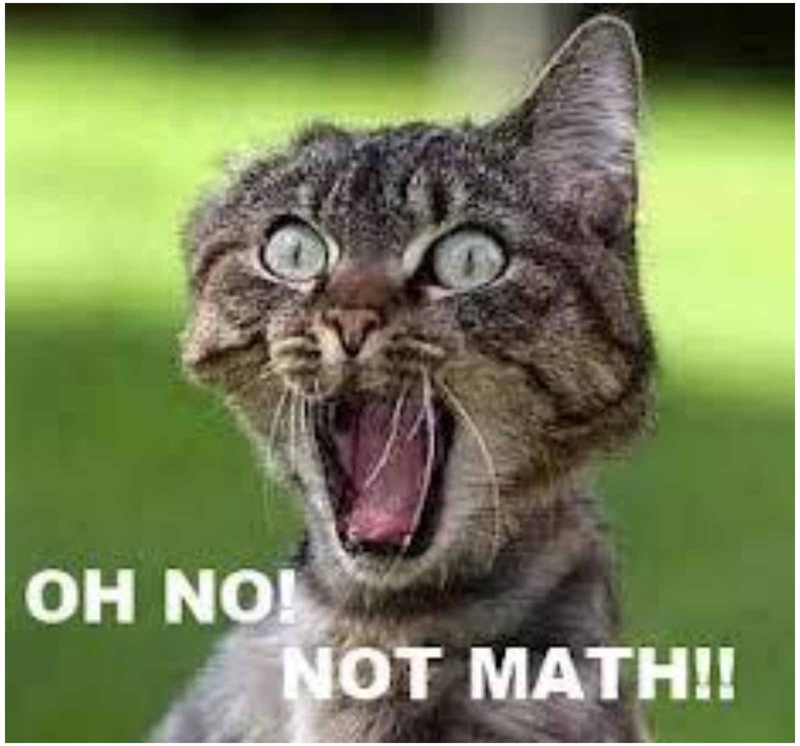
Complexity analysis

Definition, Intuition, Examples

Big O notation

- A relative representation of the complexity of an algorithm
- Scaling nature of an algorithm
  - how the resource use (mostly time) of an algorithm scales in response to the input size
  - worse case analysis: **upper-bound** of the resource use as `N` gets larger and larger (the algorithm will never take more space/time above that limit)
- Why do we need it?
  - To compare the <u>worse case performance</u> of our algorithms in a standardized way

Big O notation



Big O notation

- Mathematical definition

$$f(x) = O\big(g(x)\big) \text{ as } x \to +\infty$$

if and only if

$$\exists c\,, x_0 \text{ such that } |f(x)| \le c \times |g(x)|\ \forall x \ge x_0$$

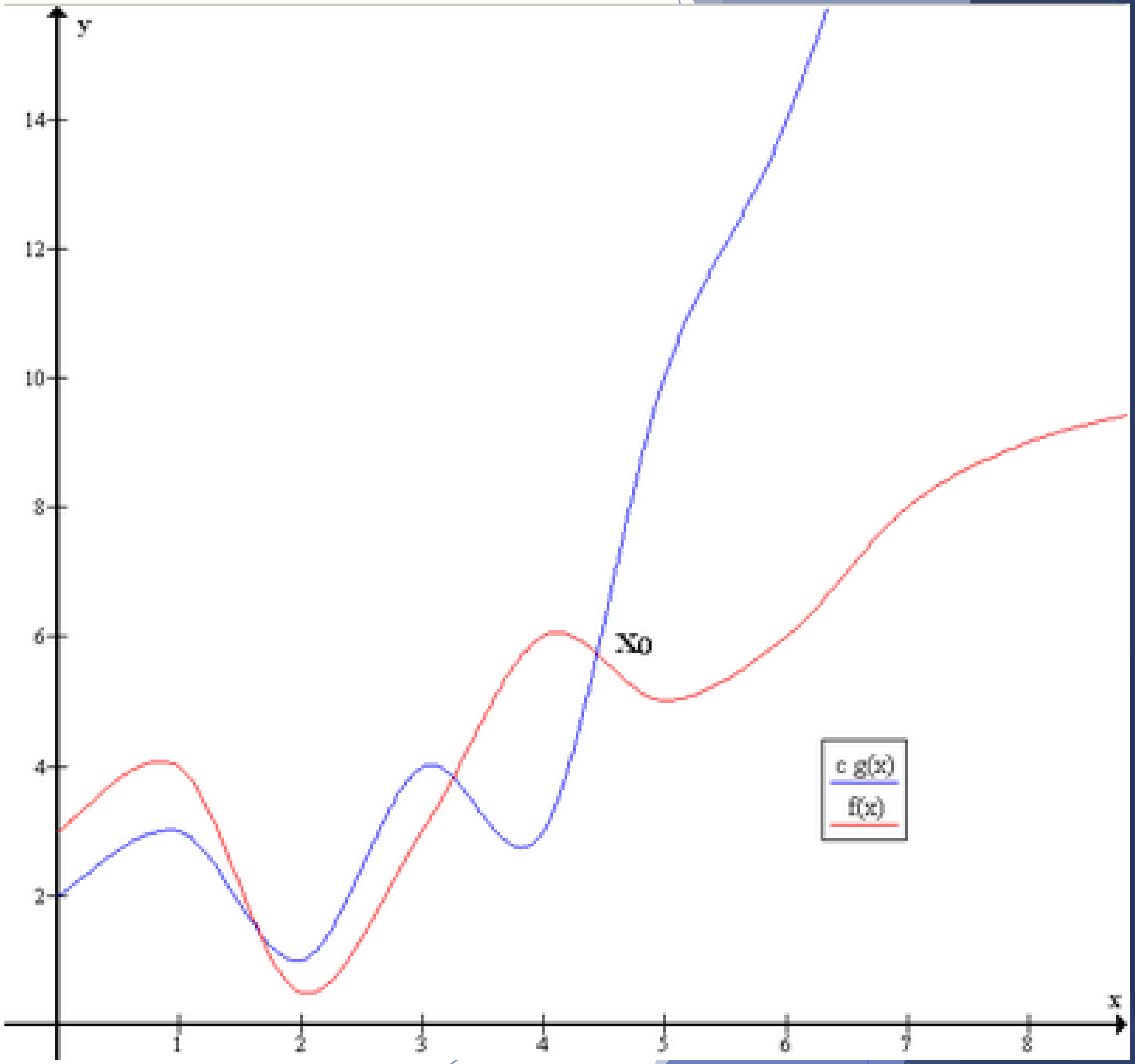
- In English, we say that "the function $f(x)$ has **O**rder $g(x)$", or "is Oh of $g(x)$"
- $f(x)$ represents the algorithm; x is the input size (N)
  - each algorithm is related to its own $g(x)$: each algorithm has a specific order/class

Big O notation

$$f(x) = O(g(x)) \text{ as } x \to +\infty$$

if and only if

$$\exists c\,, x_0 \text{ such that } |f(x)| \leq c \times |g(x)|\ \forall x \geq x_0$$



Big O notation

Example of orders (classes)

- Constant-time $O(1)$
- Logarithmic-time $O(\log N)$
- Linear-time $O(N)$
- Quasilinear-time $O(N \log N)$ (also called linearithmic)
- Quadratic-time $O(N^2)$
- Polynomial-time $O(N^k)$
- Exponential-time $O(k^N)$
- Factorial-time $O(N!)$

Operations with Big O notation

- $O(c) = O(1)\ \forall c$ constant
- $c \times O(f(n)) = O(c \times f(n)) = O(f(n))\ \forall c$ constant
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  - What happens with $O(n) + O(n)$?
- $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$
  - What happens with $O(n) \times O(n)$?
- $O(n^k + n^{k-1} + \cdots + n + c) = O(n^k)$
  - we take the highest exponent

Big O notation examples

- $\boldsymbol{O(1)}$

```
x[1] + y[4]
```

Big O notation examples

- $\boldsymbol{O}(\mathbf{1})$

```
FOR i = 1 TO 10
  x += a[i]
```

Big O notation examples

- $\boldsymbol{O}(\boldsymbol{N})$

Summing all the elements of an array

```
x = 0
FOR i = 0 TO N-1
  x += a[i]
```

Big O notation examples

- $O(N)$

Sequential search in an array… remember?

```
FOR i = 0 TO N-1
  IF a[i] = v
    RETURN i
RETURN -1
```

Big O notation examples

- $\boldsymbol{O}(\boldsymbol{N})$

Computing the factorial of a number N

$$N! = N \times (N-1) \times (N-2) \times \cdots \times 1$$

```
Fact(N)
  IF N = 0
    1
  ELSE
    N × Fact(N-1)
```

Big O notation examples

- $\boldsymbol{O(log\,N)}$

Binary search in array… remember?

- How many times can we divide N by 2?
  - $\log_2 N$
- Running time proportional to the logarithm of the number of elements in the array

```
BinSearch(a, low, high, v)
  IF low > high
    RETURN -1
  middle = (low + high) / 2
  IF a[middle] > v
    BinSearch(a, low, middle – 1, v)
  ELSE IF a[middle] < v
    BinSearch(a, middle + 1, high, v)
  ELSE
    RETURN middle
```

Big O notation examples

- $O(N^2)$

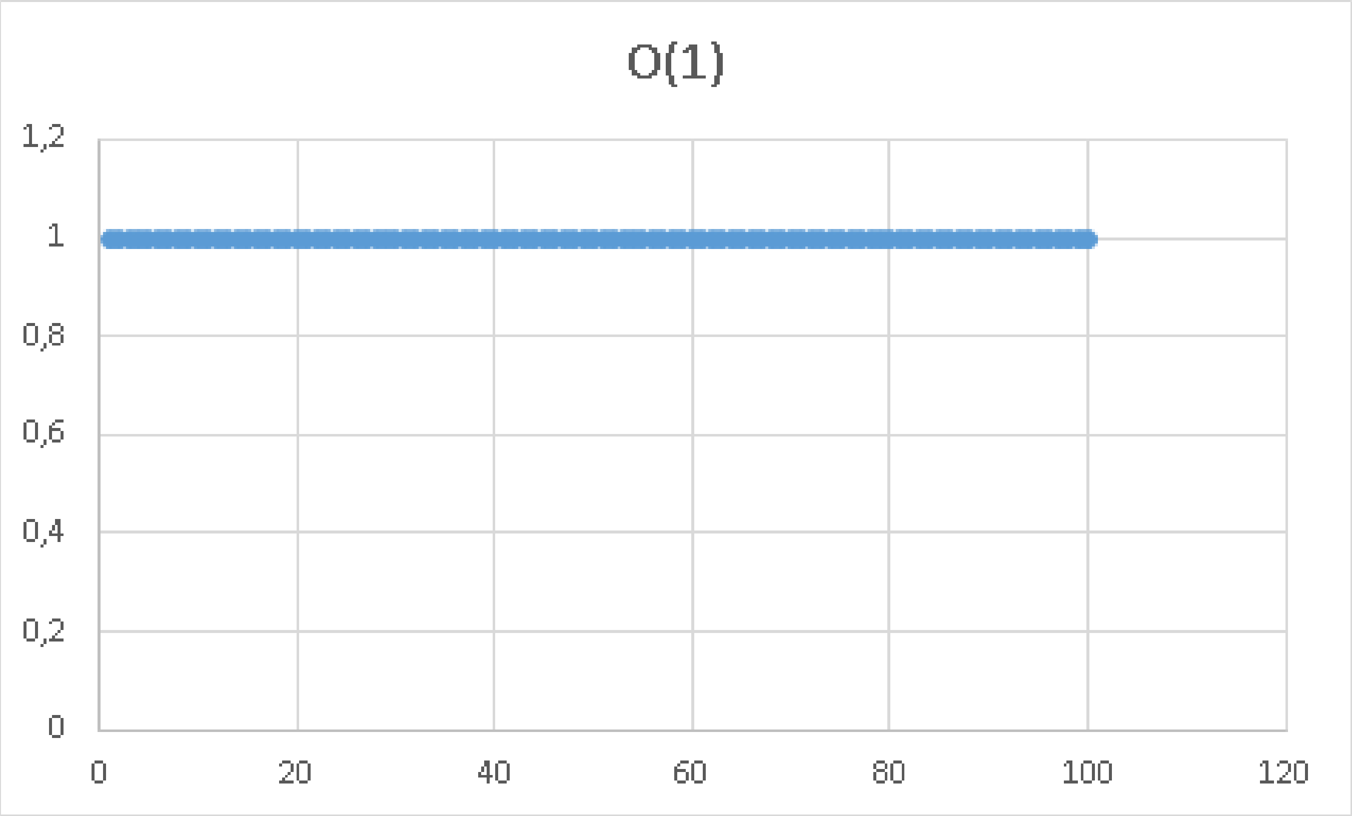
```
FOR i = 1 TO N
  FOR j = 1 TO N
    v += i + j * N
```

Big O notation examples

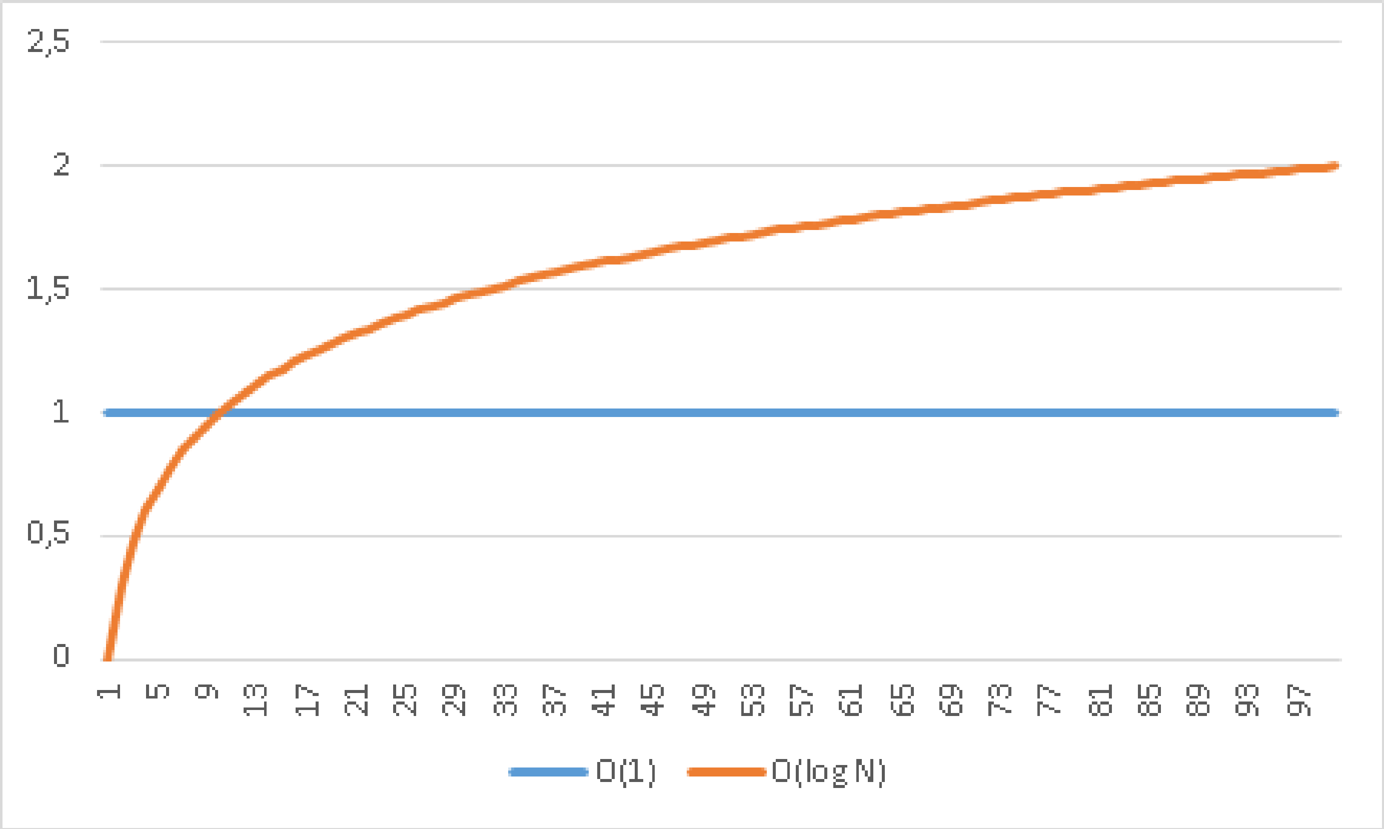
- $O(N^3)$

```
cnt = 0
FOR i = 1 TO N
  FOR j = i+1 TO N
    FOR k = j+1 TO N
      IF a[i] + a[j] + a[k] == 0
        cnt++
```

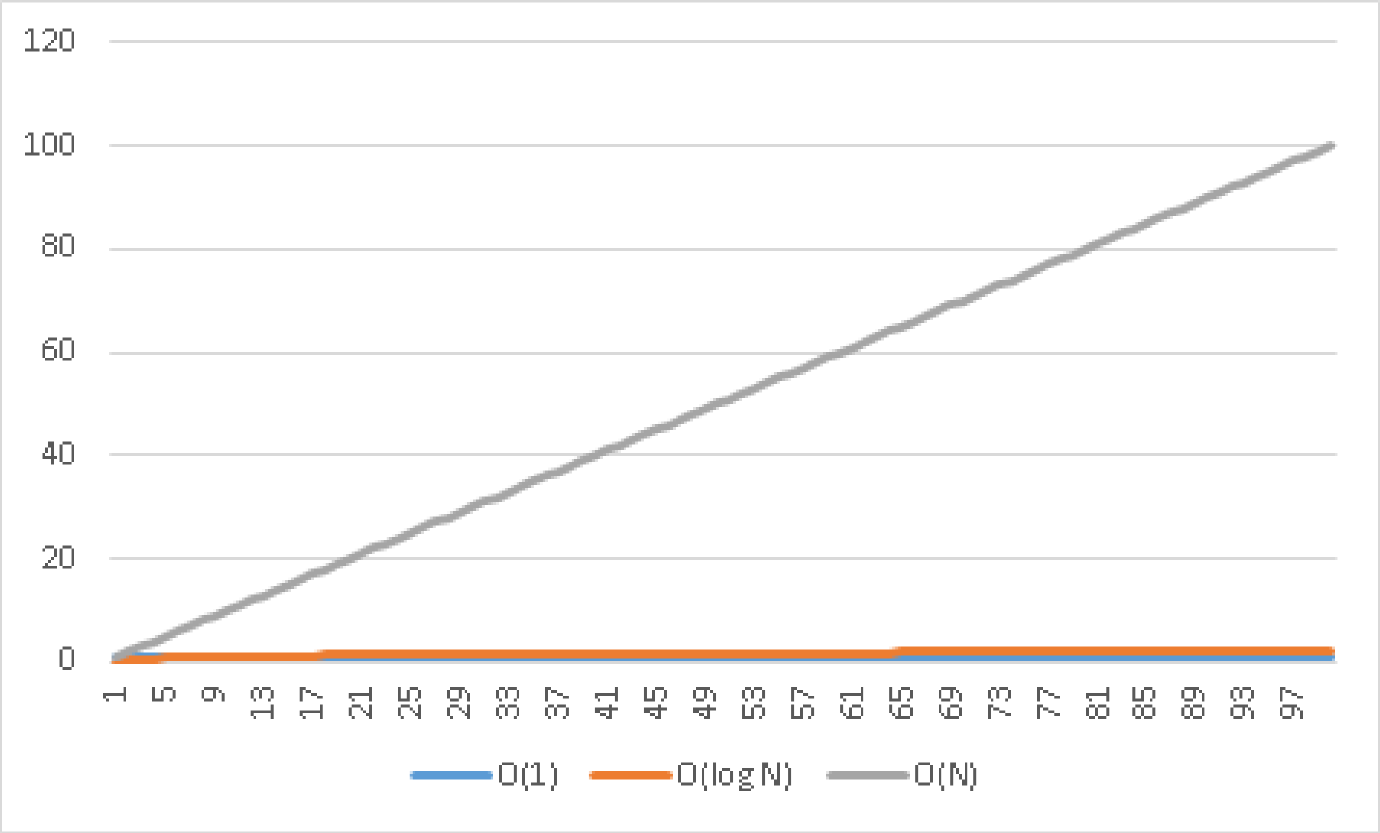
Big O notation comparison



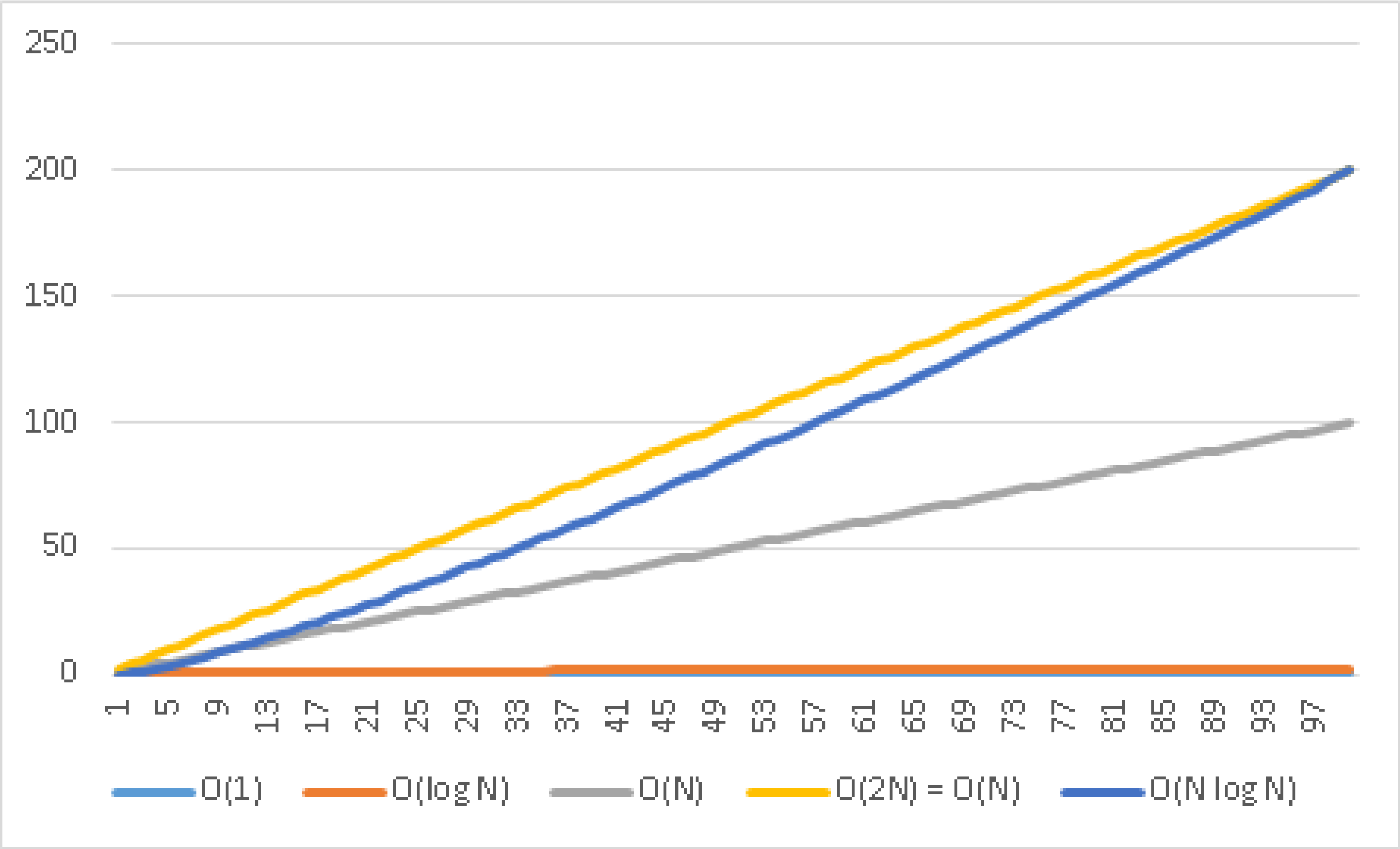
Big O notation comparison



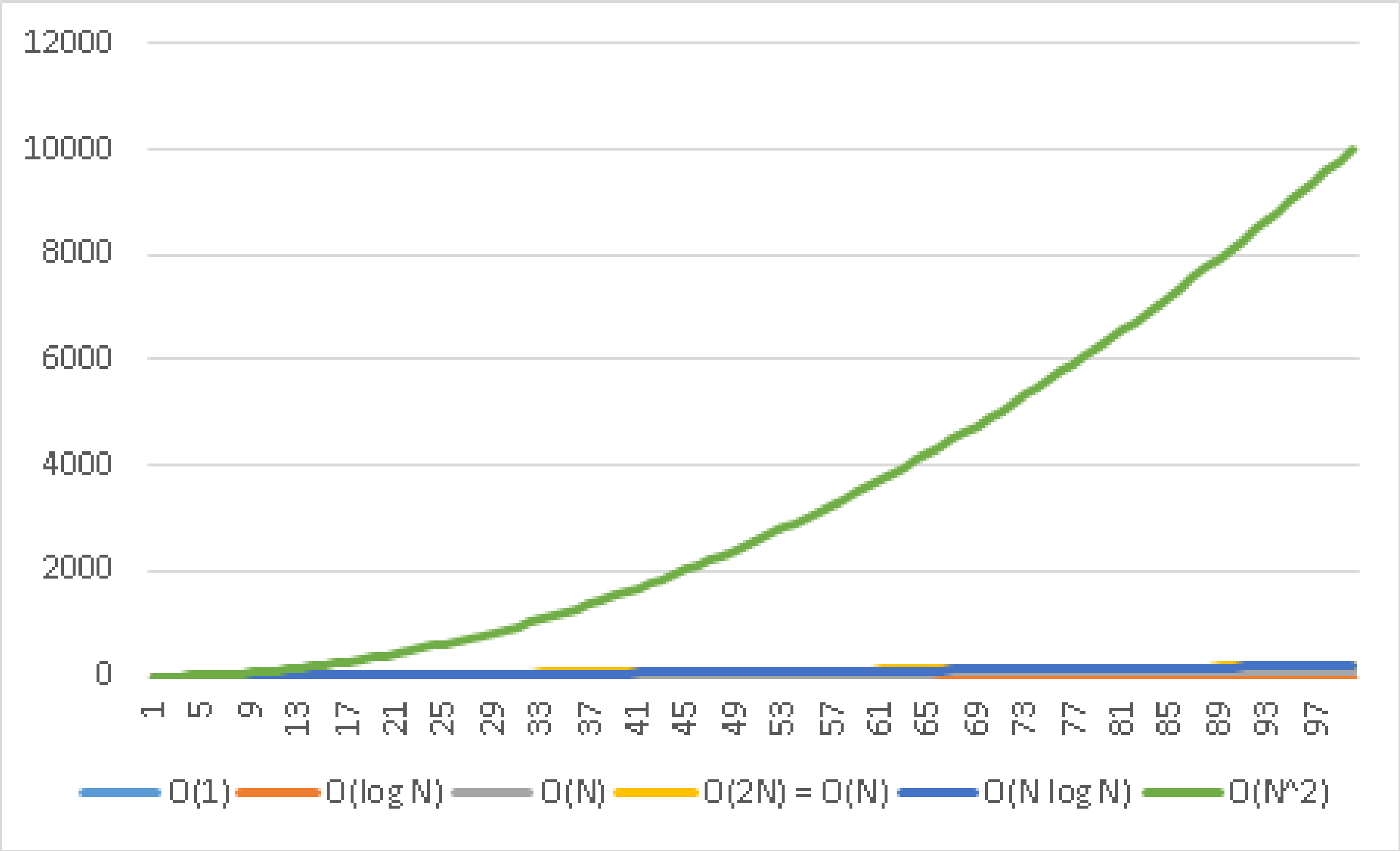
Big O notation comparison



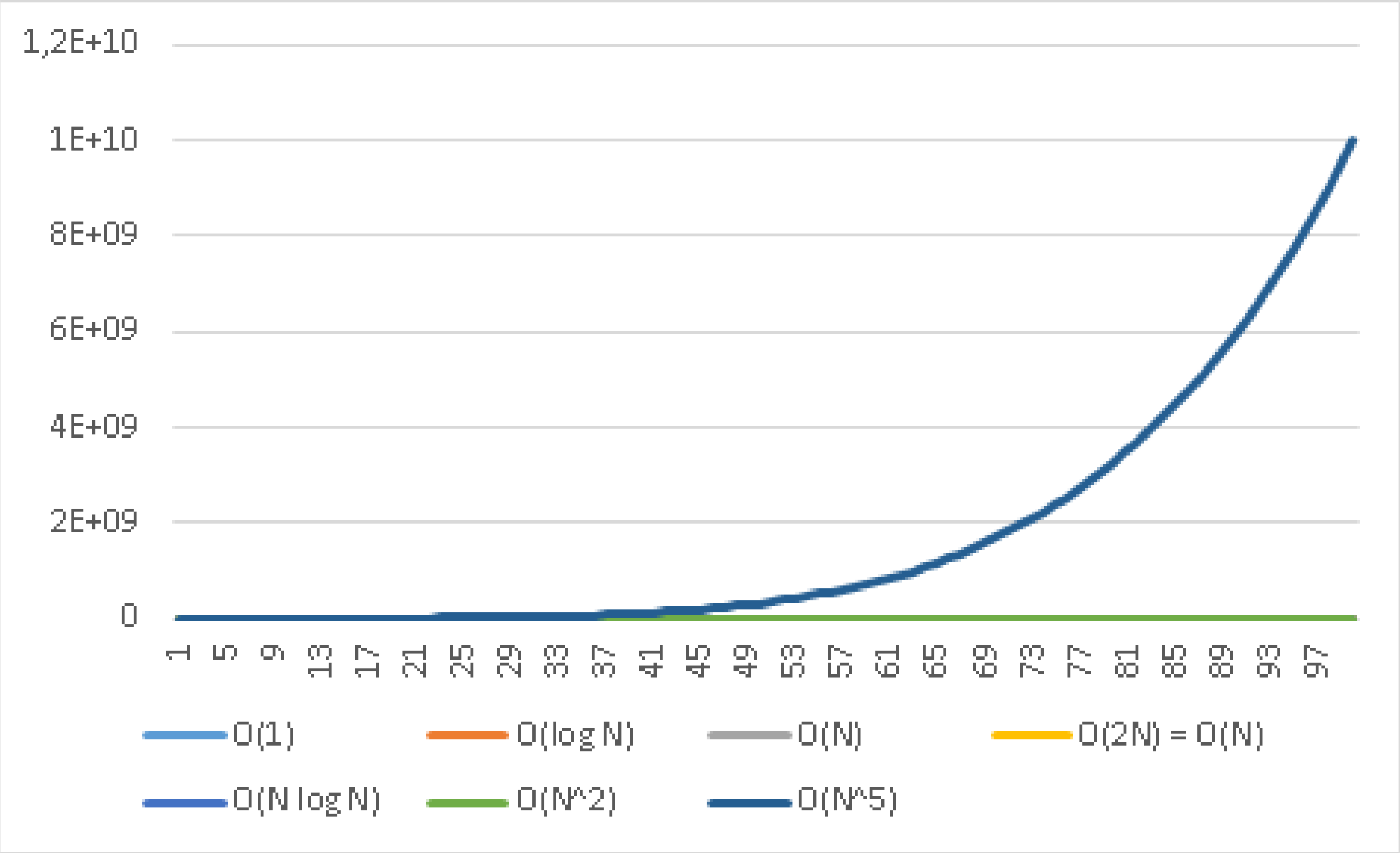
Big O notation comparison



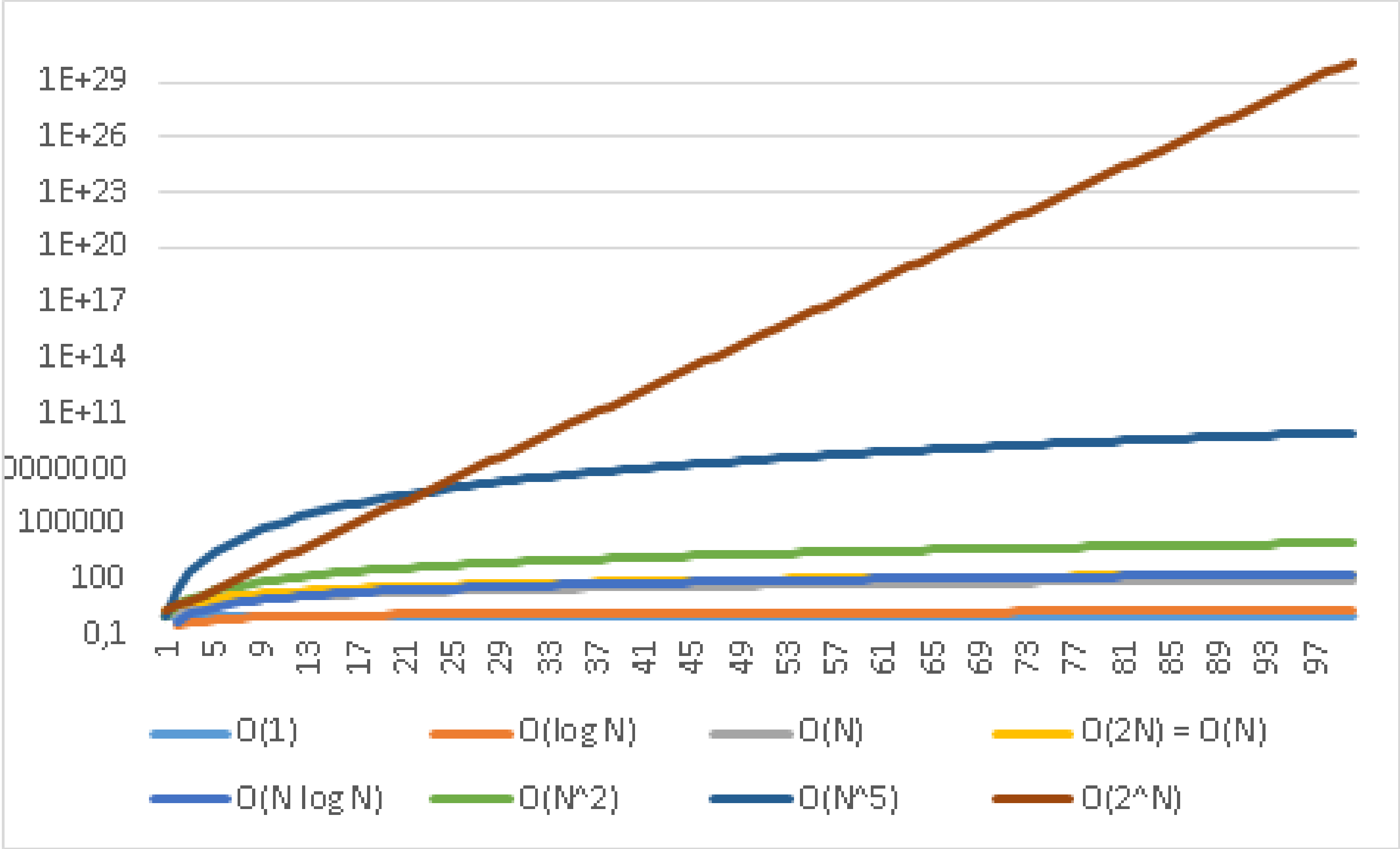
Big O notation comparison



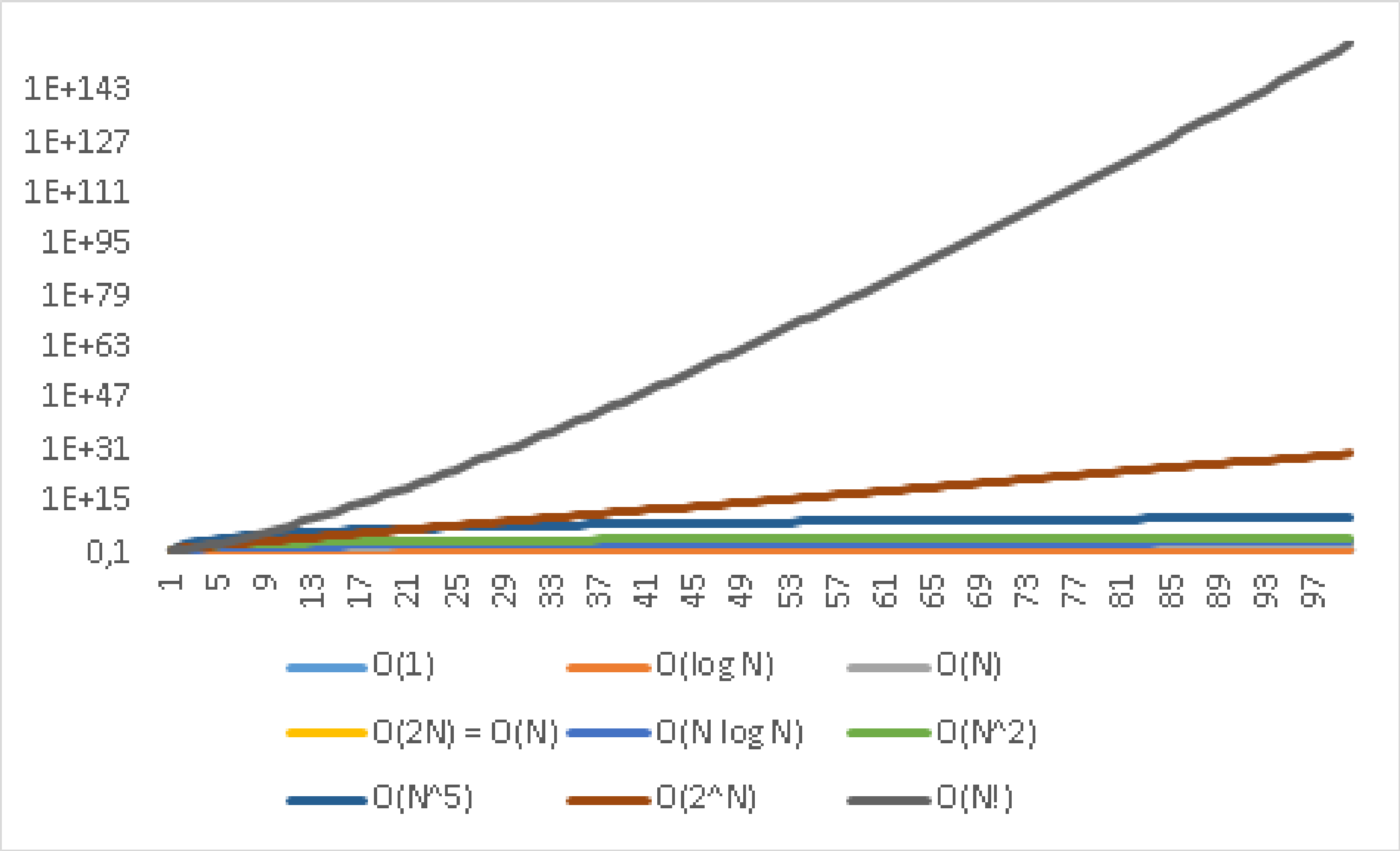
Big O notation comparison



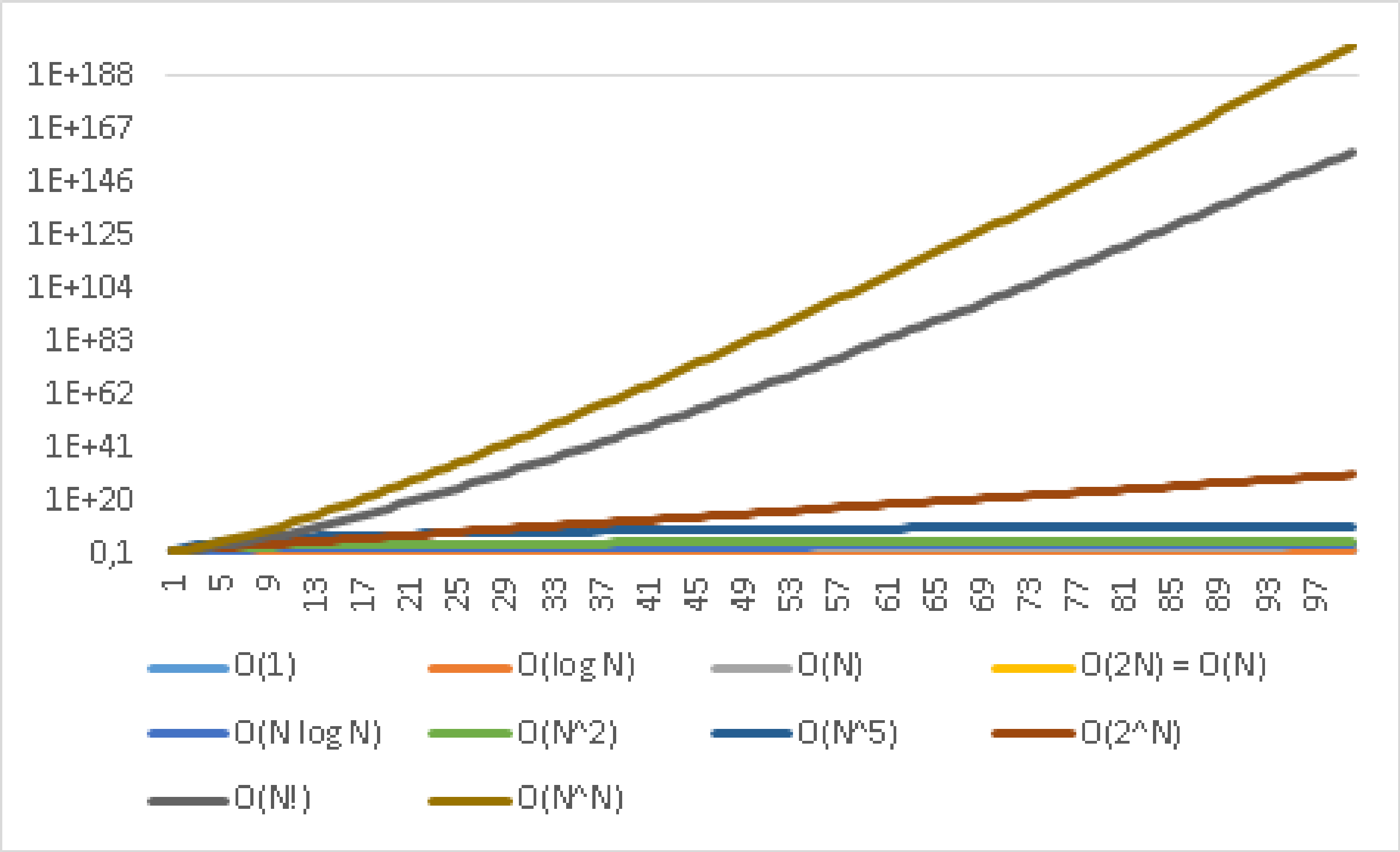
Big O notation comparison



Big O notation comparison



Big O notation comparison



Homework

- Multiple choice questions on GrandeOmega
- Practice using C#
  - Implement linear search and binary search
- Read modulewijzer
- Study the slides

- … See you next week ☺