Algorithm analysis to describe their complexity

Assessing the difficulty of an algorithm

Algorithms

Piotr Fulmański

Lecture goals

Learn and UNDERSTAND

- Theory of computation
- Computational complexity
- Dominant relation and asymptotic analysis
- Different time complexities
- General complexity classes
- Working with the Big Oh

Theory of computation

Theory of computation

The *theory of computation* is the branch of *theoretical computer science* that deals with how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches:

- automata theory and languages,
- computability theory,
- and computational complexity theory.

All of them allow us to search an answer for the following question

What are the fundamental capabilities and limitations of computers?

Theory of computation

Automata theory and languages

Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved using them.

The word automata (the plural of automaton) comes from the Greek word *αὐτόματα* which means *self-acting*.

Theory of computation

Computability theory

The field of the theory of computation that deals with researching which problems are solvable using computers.

One of the fundamental question of computer science is to determine the power of computers by understanding the problems that can be solved using them. Modern computers allow to compute so many things that it is tempting to think that solving each problem by them is only a matter of time. However, it turns out that we can find problems that computers will never be able to solve, regardless of the resources available.

Theory of computation

Computability theory – the halting problem

In computability theory, the *halting problem* is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running (i.e., halt) or continue to run forever.

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof was a mathematical definition of a computer and program, which became known as a Turing machine; the halting problem is undecidable over Turing machines.

Theory of computation

Computational complexity theory

The field of the theory of computation that deals with determining the amount of resources needed to solve computational problems (which is known to be computable).

The resources considered are such as time, memory or the number of processors.

Computational complexity

Computational complexity

Strong need for models

Computational complexity determine the amount of resources needed to solve a given computational problem. **Computational complexity**, in most cases the amount of memory or time, **is expressed as a function of input data size**. It is common for this function **to expressed the worst-case complexity**, that is the maximum of the amount of resources that are needed for all inputs of a given size.

Computational complexity

The space complexity

The space complexity of an algorithm or a computer program is the amount of **memory space, expressed in bytes or in number of basic data types variables like `int` or `float`**, required to solve an instance of the computational problem as a function of the size of the input.

Computational complexity

The time complexity

The time complexity describes the amount of time it takes to run an algorithm.

We do not express time complexity in standard units of time. Providing time complexity in units of time is inconvenient, because the result depends on the speed of the computer on which the measurements were made and it is difficult to refer such results to other computers, equipped with other/different hardware resources, where the time of performing similar operations may vary significantly.

Computational complexity

The time complexity

Therefore, **we express the computational complexity in the number of elementary (dominating) operations**.

The *dominant operation* is an operation whose execution **directly affects the overall execution** time of the entire algorithm. We treat the other operations as irrelevant – that is, their execution time is negligibly small compared to the time of execution of all dominant operations.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to differ by at most a constant factor.

Computational complexity

Strong need for models

Take, for example, the model that the Earth is flat. You might argue that this is a bad model, since it is quite well established that the Earth is round. But, in everyday routines this model is more than enough to be applied in meany practical problems, like laying the foundation of a house, or navigate with the help of map.

RAM – Random Access Machine

Under the RAM model of computation, you are confronted with a hypothetical computer where:

- Each simple operation (+, *, –, =, if, call) takes exactly one time unit. Exact time unit is not important.
- Loops and subroutines are not considered simple operations. The time it takes to run through a loop or execute a subprogram depends upon the number of loop iterations or the specific nature of the subprogram.
- Each memory access takes exactly one time unit.
- You have as much memory as you need.

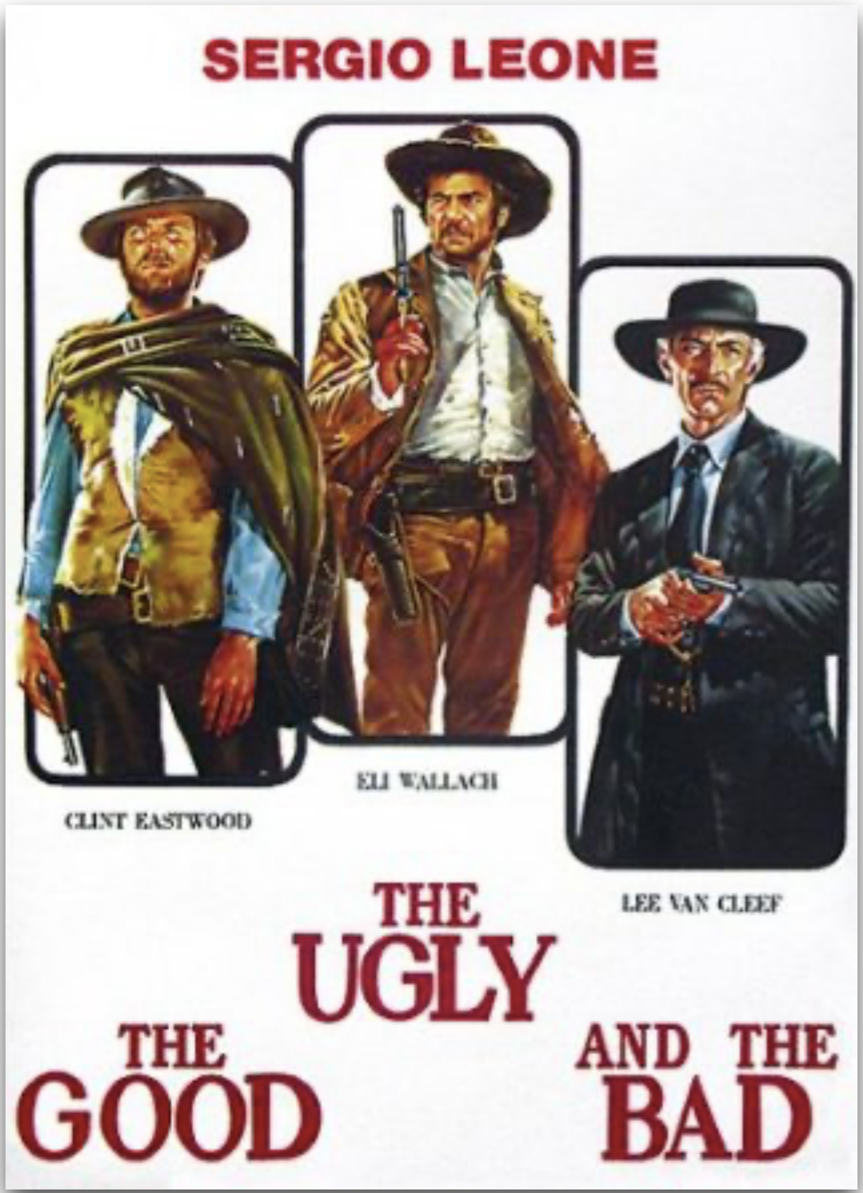
The RAM model is not perfect as flat model of the Earth is not perfect too. However being simple to work with it proves to be an excellent model for understanding how an algorithm performs on a real computer and turned out to be very practical.

The robustness of this model enables us to analyze algorithms in a machine-independent way.

The Good, the Bad and the Ugly

Best case, worst case, and average case complexity

Using the RAM model of computation, you can count how many steps you algorithm takes on any given input instance by executing it. However, to understand how good or bad an algorithm is in general, you must know how it works overall possible instances, which of course is impossible.



The only thing you can do is to assess complexity for the best case you can imagine, worst case and average case.

The Good, the Bad and the Ugly

Best case

The best-case complexity of the algorithm is the function defined by the **minimum** number of steps taken in any instance of size n.

The Good, the Bad and the Ugly

Worst case

The worst-case complexity of the algorithm is the function defined by the **maximum** number of steps taken in any instance of size n.

The Good, the Bad and the Ugly

Average case

The average-case complexity of the algorithm is the function defined by the **average** number of steps taken in any instance of size n.

The Good, the Bad and the Ugly

However, even with limiting yourself to only three cases it is very difficult to be precise because:

- You may encounter many "bumps". For example binary search algorithm typically runs a bit faster for arrays of size exactly $n = 2k - 1$ (where k is an integer), because the array partitions work out nicely.

- Require too much detail. Counting the exact number of RAM instructions executed in the worst case requires the algorithm be specified to the detail of a complete computer program. Furthermore, the very precise answer depends upon uninteresting coding details (e.g. did the code use a case statement or nested ifs?).

It turned out to be much easier to talk in terms of simple ***upper and lower*** bounds of time-complexity functions which simplifies analysis by **ignoring levels of detail that do not impact our comparison of algorithms**.

Dominant relation and asymptotic analysis

Details – are they important?

Do you really need to know that time complexity of your algorithm is equal to

$$T(n) = 987n^2 + 654n + 321\,\lg_2 n + 99887766$$

Details – are they important?

Growth rates and dominance relations

See Excel table number 1.

Details – are they important?

Growth rates and dominance relations

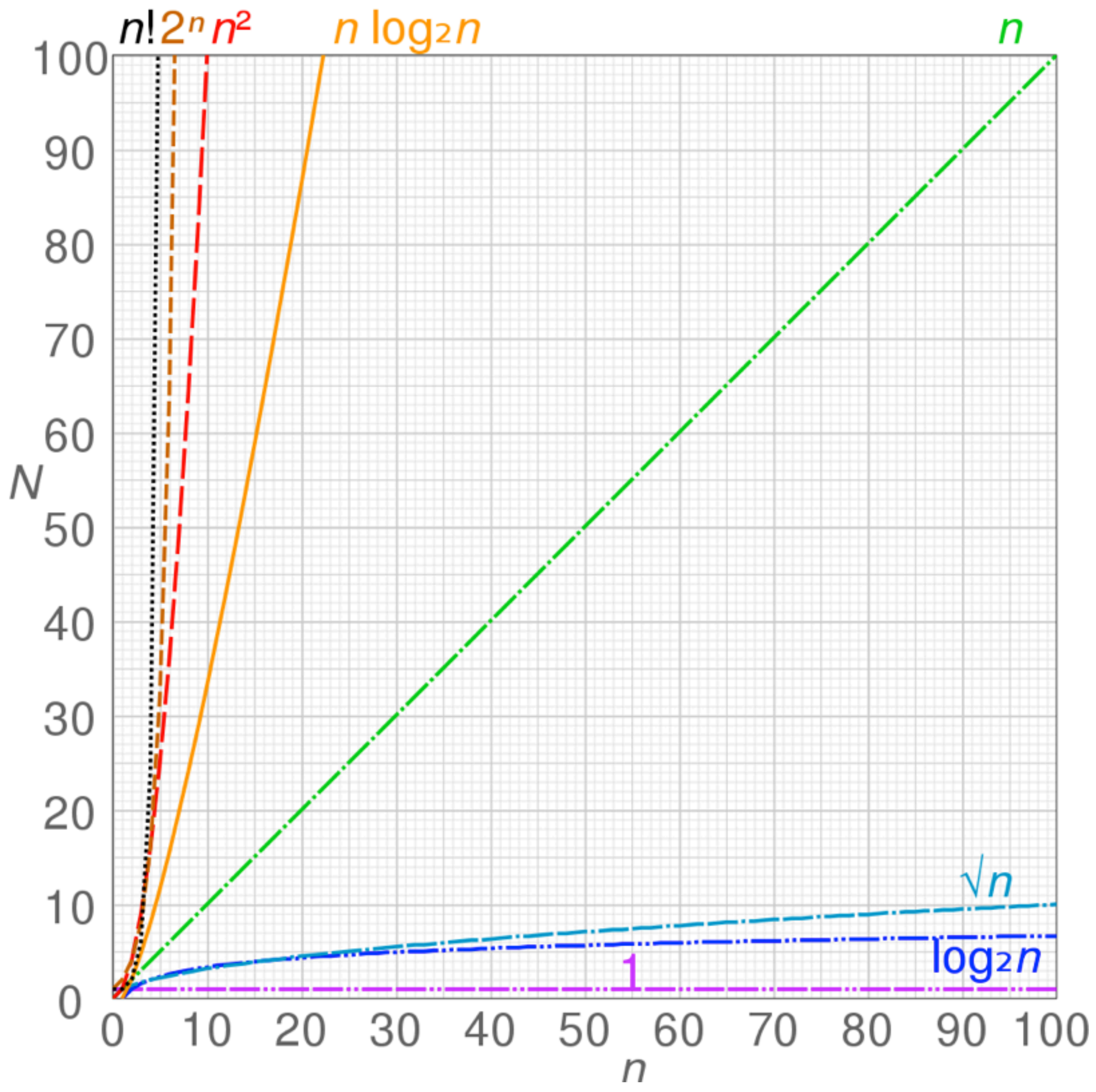
In our case

$$T(n) = 987n^2 + 654n + 321 \lg_2 n + 99887766$$

you obtain the following percentage share in the final result – see Excel table number 2.

Details – are they important?

Growth rates and dominance relations



Details – are they important?

Growth rates and dominance relations

In our case

$$T(n) = 987n^2 + 654n + 321\,\lg_2 n + 99887766$$

you obtain the following percentage share in the final result – see Excel table number 2.

For

- $n = 1$, 99.9% of the value of $T(n)$ is given by 99887766 factor.
- $n = 10^6$, 50.3% of the value of $T(n)$ is given by 99887766 factor and 49.7% is given by $987n^2$ factor.
- $n = 10^9$, 99.9% of the value of $T(n)$ is given by $987n^2$ factor.

Conclusion:

When you talk about $T(n)$ for high n, it doesn't make any sene to consider any other factor than $987n^2$.

Details – are they important?

Growth rates and dominance relations

Conclusion:

When you talk about $T(n)$ for high n, it doesn't make any sene to consider any other factor than $987n^2$.

You say that a faster growing function dominates a slower growing one.

That's why:

- You shouldn't care about small values of n, say anything smaller than some n_0. After all, probably you don’t really care whether one sorting algorithm sorts ten items faster than another because both will do this in in the blink of an eye. Think you should really car is which one will do its job for huge data, for example 1,000,000 or 1,000,000,000 items.

- You should take into account only the fastest growing factor as for huge data it will consume all computation time.

The Big Oh notation enables you to ignore details and focus on the big picture.

Asymptotic analysis

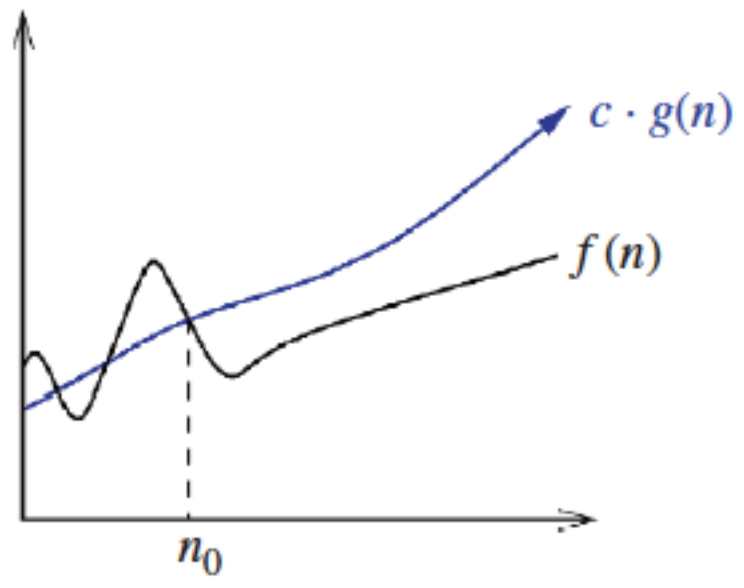
Big Oh

$f(n) = O(g(n))$ if there exists:

- real constant $c > 0$,
- natural constant n_0

such that for every $n > n_0$:

$f(n) \leq c \cdot g(n)$.



Asymptotic analysis

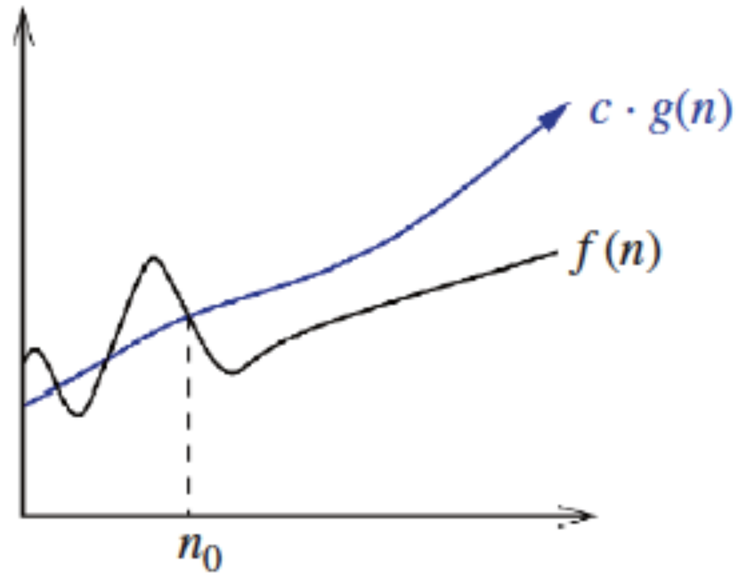
Big Oh

$f(n) = O(g(n))$ if there exists:

- real constant $c > 0$,
- natural constant n_0

such that for every $n > n_0$:

$f(n) \leq c \cdot g(n)$.



For example if $f(n)$ is defined as:

$f(n) = 3n^2 + 2n + 1$

then

$f(n) = O(n^2)$

because for every $n > n_0 = 2$ and $c = 4$.

Notice that this is not the only possible solution.

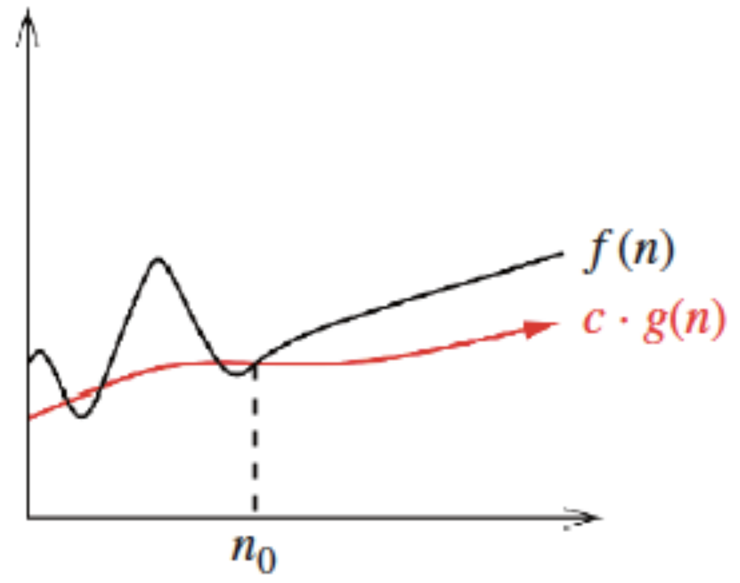
Asymptotic analysis

$f(n) = \Omega(g(n))$ if there exists:

- real constant $c > 0$,
- natural constant n_0

such that for every $n > n_0$:

$f(n) \geq c \cdot g(n)$.



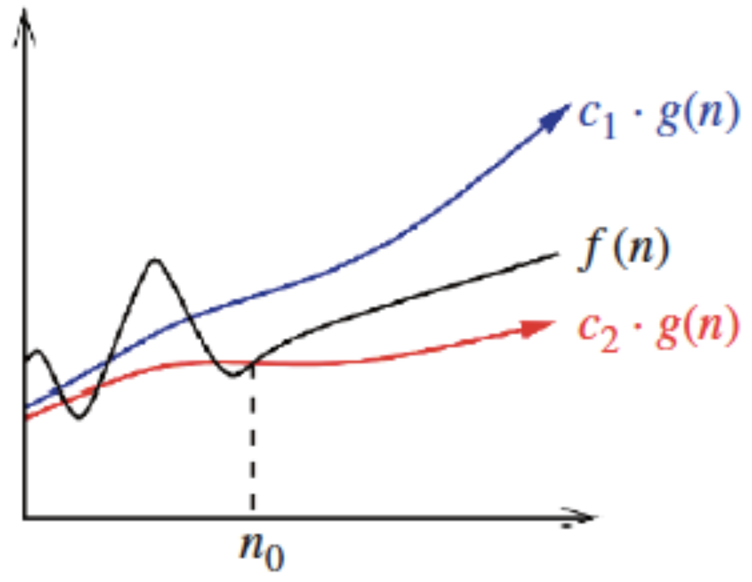
Asymptotic analysis

$f(n) = \Theta(g(n))$ if there exist:

- real constant $c_1 > 0$ and $c_2 > 2$,
- natural constant n_0

such that for every $n > n_0$:

$f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$.



Asymptotic analysis

$(x+y)^2 = O(x^2+y^2)$

Prove:

Notice that:

- If $x \leq y$ then $2xy \leq 2y^2 \leq 2(x^2+y^2)$
- If $x \geq y$ then $2xy \leq 2x^2 \leq 2(x^2+y^2)$

Either way, you now can bound $2xy$ by $2(x^2+y^2)$.

This means that:

$$\begin{aligned}(x+y)^2 &= x^2+2xy+y^2\\ &= x^2+y^2+2xy\\ &\leq x^2+y^2+2(x^2+y^2)\\ &= 3(x^2+y^2)\end{aligned}$$

and in consequence you obtain:

$(x+y)^2 \leq 3(x^2+y^2)$

and in consequence:

$(x+y)^2 = O(x^2+y^2)$

Asymptotic analysis

Adding functions

The sum of two functions is governed by the dominant one:

$$f(n) + g(n) = \Theta(max(f(n), g(n))).$$

It gives that, for example:

- $n^3 + n^2 + n + 1 = \Theta(n^3)$
- If $f(n) = O(n^2)$ and $g(n) = O(n^2)$, then $f(n) + g(n) = O(n^2)$ as well.

Asymptotic analysis

Multiplying functions

Because multiplication is like repeated addition. In consequence, multiplying a function by a constant cannot affect its asymptotic behavior, because you can multiply the bounding constants in the Big Oh analysis to account for it.

$$\Theta(c \cdot f(n)) = \Theta(f(n))$$

$$\Theta(f(n)) \cdot \Theta(g(n)) = O(f(n) \cdot g(n))$$

Asymptotic analysis

Transitivenes

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.

Different time complexities

Different time complexities

O(1)

If the method's time does not vary and **remains constant as the input size increases**, the algorithm is said to have O(1) complexity.

The algorithm is not affected by the size of the input.

It takes a fixed number of steps to complete a particular operation, and this number is **independent** of the quantity of the input data.

Different time complexities

O(1)

```
func foo(array):
  last = array.count - 1
  sum = array[0] + array[last]
  return sum
```

Different time complexities

O(log(n))

A method with a logarithmic complexity divides the issue into smaller/simpler cases for each iteration.

It takes $log(n)$ steps to execute a particular operation on n items, where the logarithm base is usually 2.

Because the logarithm base is not critical to the order of the operation count, it is frequently ignored.

Explanation of the last sentence is as follow:

Compare the following three values:

- $\log_2(1{,}000{,}000) = 19.9316$, $\log_2(1{,}000{,}000{,}000) = 29.9874$ ($+49.5\,\%$)
- $\log_3(1{,}000{,}000) = 12.5754$, $\log_3(1{,}000{,}000{,}000) = 18.8631$ ($+50\,\%$),
- and $log_{100}(1{,}000{,}000) = 3$, $log_{100}(1{,}000{,}000{,}000) = 4.5$ ($+50\,\%$).

A big change in the base of the logarithm produces little difference (compared to n tending to plus infinity) in the value of the logarithm.

Changing the base of the logarithm from a to c involves multiplying by $log_c a$. This conversion factor is absorbed in the Big Oh notation whenever a and c are constants. Thus, you are usually justified in ignoring the base of the logarithm when analyzing algorithms.

Different time complexities

O(log(n))

Search for x element in sorted array 'array' and returns its index if exists or -1 in other case.

```
func binarySearch(array, int x):
  l = 0
  r = array.count-1

  while (l <= r):
    candidate = l + (r - l) / 2

    if (array[m] == x):
      return m
    else if (array[m] < x):
      l = m + 1
    else:
      r = m - 1

  return -1
```

Different time complexities

O(n)

The algorithm is said to have complexity O(n) when to execute an operation on n items it takes about the same number of steps as the number of elements.

The time consumed by the process changes linearly as the input size rises.

Different time complexities

O(n)

```
func sumOfElements(array):
  sum = 0

  for index in 0...(array.count-1):
    sum += array[index]

  return sum
```

Different time complexities

O(n log(n))

An algorithm with an $O(n \log(n))$ complexity divides the problem into little chunks for each call, then takes each of their results and combine them together to get final result.

Different time complexities

O(n log(n))

Merge Sort has complexity of this type. What is interesting, it has this complexity for all cases: best, worst and average because it always divides the array into two halves and takes linear time to merge two halves.

Different time complexities

O(n^2)

Square complexity – his is a specific case of polynomial complexity.

Different time complexities

O(n^2)

```
func foo(array):
  sum = 0

  for i in 0...(array.count-1):
    for j in 0...i:
      sum += i + j

  return sum
```

Different time complexities

O(n^x)

Polynomial complexity.

Different time complexities

O(n^x)

- $O(n^3)$: multiplication of square matrix of size n.

Different time complexities

O(2^n)

Exponential complexity.

Different time complexities

O(2^n)

As the input there is an array of numbers with n elements. All elements are different. Return an array that will contain all possible subsets of the elements of the input array.

Different time complexities

O(n!)

Factorial complexity.

Different time complexities

O(n!)

The travelling salesman problem (TSP) asks the following question:

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

It is an NP-hard.

General complexity classes

General complexity classes

- P (polynomial time) complexity class – problem solving during polynomial time.

  It contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.

- NP (nondeterministic polynomial time) complexity class – the problem is not known at polynomial time, but the solution can be checked in polynomial time.

  Other words, although a solution to a problem can be verified quickly, there is no known way to find a solution quickly.

- NP-hard – even solution to a problem, no matter how we get it, cannot be verified "quickly", in polynomial time.

General complexity classes

Simple problems – P (polynomial time)

- Finding the smallest value in an ordered sequence of n elements $O(1)$.
- Searching for a certain value in an unordered sequence of n elements - sequential search $O(n)$.
- Searching for a certain value in an ordered sequence of n elements - binary search $O(log_2 n)$.
- Ordering n-element sequence – selection sort $O(n^2)$.
  Ordering n-element sequence – quick sort $O(n \log_2 n)$.
- Multiplication of square matrices of size $n \times n$ $O(n^3)$.

General complexity classes

Difficult problems – NP (nondeterministic polynomial time)

Finding a subset with a given sum of values in a set of n elements, e.g. in the set {-2, 6, -3, 72, 10, -11} we are looking for a subset with sum 0.

In the pessimistic case, it is necessary to determine all subsets of the given set.

Computational complexity $O(2^n)$.

Assuming the time of one operation equals 1ms, for a set of 64 elements the problem will be solved after 500 million years!

On the other side, checking if the candidate is a solution to the problem: {-2, 6, -3, 10, -11} is simple.

Computational complexity O(n) - polynomial.

General complexity classes

Really difficult problems – NP-hard

TSP – raveling salesman problem:

Finding the shortest path connecting n points in a plane such that point must be visited exactly once, except for the first, which is also the last (the so-called Hamiltonian cycle).

In the pessimistic case, it is necessary to determine all permutations (half of the permutations when the routes between points are symmetrical).

Computational complexity $O(n!)$.

For n=24 it will take almost 20 trillion years!

Unfortunately verification if candidate: A-B-C-D-A is a solution, also has the same computational complexity $O(n!)$ because needs to check all possible solutions.

Working with the Big Oh

Working with the Big Oh

Selection sort

The exact number of times the if statement is executed is given by:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} (n - i - 1)$$

In consequence:

$$\begin{aligned} T(n) &= (n-1) + (n-2) + \ldots + 2 + 1 \\ &= \frac{(1 + (n-1))(n-1)}{2} \\ &= \frac{(n)(n-1)}{2} \\ &= \frac{1}{2} n(n-1) \\ &= O(n^2) \end{aligned}$$

```
func selectionSort(array):
  if array.count < 2:
    return

  for current in 0...(array.count-2):
    minimal = current

    for other in (current+1)..(array.count-1):
      if array[minimal] > array[other]:
        minimal = other

    if minimal != current:
      array.swap(minimal, current)
```

Working with the Big Oh

Selection sort

In this case best case, worst case, and average case are equal.

Working with the Big Oh

Insertion sort

In contrast to selection sort, here inner loop iterate a different number of times, because you break it every time when the element finds its proper place in sorted order (`array[shifting] >= array[shifting-1]`).

Since worst-case analysis is what you should think in Big Oh notation, then you should ignore any early termination and assume that this loop **always** goes to its maximum, so `current` times.

In fact, you can go further making simplifications and assume it always goes around `array.count` times since `current < array.count`.

Since the outer loop goes around `array.count` times, insertion sort (in worst case) must be a quadratic-time algorithm, that is, $O(n^2)$.

```
func insertionSort(array):
  if array.count < 2:
    return

  for current in 1...(array.count-1):
    for shifting in current...1:
      if array[shifting] < array[shifting-1]:
        array.swap(shifting, shifting - 1)
      else:
        break
```

Working with the Big Oh

Overestimating or underestimating

The presented *round it up* approach to algorithm analysis analysis always does the job, in that the case of Big Oh.

This type of bounding running time will always be correct.

From time to time, it might be too pessimistic, overestimating the time needed to complete your algorithm. If it is a case, you are lucky – you will get result earlier than expected.

This is much better than underestimating running time and wait for result much longer than you expected.