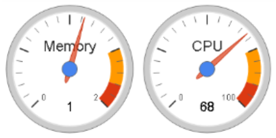
Analysis of Algorithms

"Algorithm analysis is concerned with comparing algorithms based upon the amount of computing resources that each algorithm uses. We want to be able to consider two algorithms and say that one is better than the other because it is more efficient in its use of those resources or perhaps because it simply uses fewer."

1. Empirical Analysis

We have compared different search algorithms by measuring their execution time for specific lists of integer numbers. This approach has major drawbacks:

- Experiments are performed on a **limited set of test data**. Other data may affect algorithm efficiency.
- Meaningful comparison of two algorithms requires that experiments are done in the **same environment** (same hardware and same software).
- The algorithms **must be fully implemented**. Writing multiple programs for the same problem is time consuming and inefficient approach.



2. Analysis with Mathematical Models

The goal should be to use a more general approach for algorithm comparison that:

- Takes into account **all possible sets of input data**.
- Evaluates relative efficiencies of different algorithms **independently from the environment**.
- Is performed by studying the **high-level representation of the algorithms**, without actually implementing them.

3. Primitive Operations

Primitive operations are **low-level** operations that have (relatively) **constant execution time**, e.g.:

- assigning a value to a variable
- performing an arithmetic operation
- comparing two objects/values
- traversing through a list/array/collection
- calling/returning from a simple method

The total execution time of an algorithm can be approximated by simply counting the primitive operations, assuming that different primitive operations take fairly similar time to be executed.

4. Time Function

Each portion of a **unit of code** (e.g. a method) is assessed as follows:

- **Consecutive operations**: the run time of the slowest and most dominant operation (e.g. comparison)
- **Decision structures**: the run time of the slowest branch
- **Loops:** sum of the run time of the operations in the loop, multiplied by the number of iterations **n**
- **Nested loops**: product of the sizes of all loops and the total run time of the innermost operations; the analysis is performed inside out;

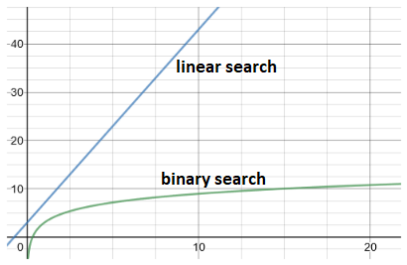
To assess the execution time of a unit of code, the running times of all sequential parts are added to get a polynomial function (execution time as a function of the **problem size**), e.g.:

Linear search: assig. + n*(comparison + comparison + assig. + arithmetic) + comparison + return

➔ **T(n) = 4 * n + 3**

Binary search: assig. + assig. + log(n)*(comparison + assig. + arithmetic + comparison + assig. + arithmetic) + return

➔ **T(n) = 6 * log n + 3**



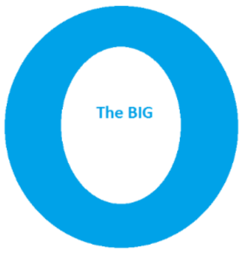
5. Growth Rate

When comparing two algorithms, two aspects are to be considered:

- speed at performing the **same task with the same input data**
- **change of speed when the data changes** (e.g. there is more data)

The speed at which the execution time increases is called **growth rate**.
Running time is an issue when data size is large, so growth rate is a better measure of algorithm efficiency.

6. Time Complexity. Big-O

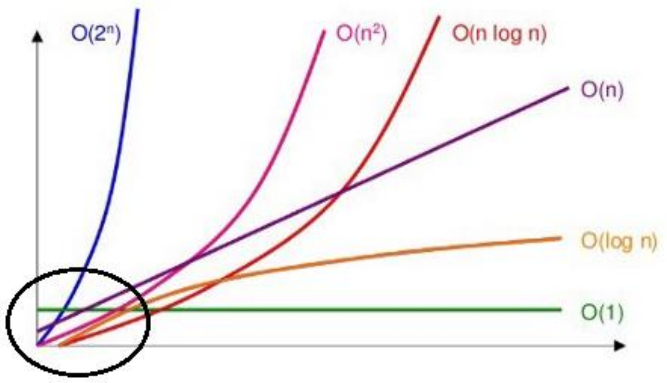


Evaluating the limit of the time function **$T(n) = 6n^2 + n*\log n - 3$** when **$n \to \infty$** is called **asymptotic analysis.** Such analysis is beyond the scope of this course.
We will simply consider the **most dominant term** in the time function.
The time function above is of **time complexity** $O(n^2)$. It describes the part of the code with the fastest increase of run time, as the size of the input data increases.

The representation of the run time of an algorithm as a function of the problem size is referred to as **Time Complexity**, and its upper limit is expressed using **Big-O** notation.

Functions with the same growth rate fall in the same complexity class. Here are the main complexity classes:

- **constant:** The method always performs certain number of iterations, regardless of the input parameters. There are **no loops, dependent on input parameter *n*** (e.g. Gauss's addition, push() and pop() on stack).
- **logarithmic:** The number of iterations is a logarithmic function of an input parameter ***n***. This means use of **loop which is executed log *n* times** (e.g. binary search, divide-and-conquer algorithms).
- **linear:** The number of iterations is a multiple of an input parameter ***n***. Usually most methods include a **loop, which iterates based on *n***, (e.g. traversing an array, linear search).
- **linearithmic:** The number of iterations is in order ***n*log n*** of an input parameter ***n***. This occurs when **methods call other methods** and the resultant time complexity function is a product of **T(n)** functions.
- **quadratic:** The number of iterations is a quadratic function of an input parameter ***n***. This happens when using **2 nested loops, dependent on *n*** (e.g. traversing 2D array).
- **exponential:** The number of iterations grows exponentially with the increase of an input parameter (e.g. Tower of Hanoi). Usually, some **recursive functions** may show such behaviour.



Thus, the efficiency of an algorithm could be evaluated by reading the code and considering in what class of ***O( )*** complexity it falls, **irrespective of programming language, data, or environment**.

For (expected) input data of small size it is best to use the simplest algorithm!

When comparing algorithms, we may consider the **best** (fastest), the **average** and the **worst** (slowest) scenarios. Best-case scenarios (e.g. small problem size) are usually irrelevant. Average-case analysis is challenging because it requires sophisticated estimates about the probability of certain input data (distribution). It is much easier to consider the worst case.
To ensure good performance of an algorithm in all possible scenarios, **worst-case analysis** is used, i.e. the longest amount of time it would possibly take to complete the task.

$$T_{\text{best}}(n) \le T_{\text{avg}}(n) \le T_{\text{worst}}(n)$$