**Algorithm Complexity**

We cannot talk about **efficiency of algorithms and data structures** without explaining the term "algorithm complexity", which we have already mentioned several times in one form or another. We will avoid the mathematical definitions and we are going to give a simple explanation of what the term means.

**Algorithm complexity**is a **measure** which evaluates the order of the **count of operations**, performed by a given or algorithm as a function of the size of the input data. To put this simpler, complexity is a rough **approximation of the number of steps**necessary to execute an algorithm. When we evaluate complexity we speak of order of operation count, not of their exact count. For example if we have an order of N2 operations to process N elements, then N2/2 and 3\*N2 are of one and the same quadratic order.

Algorithm complexity is commonly represented with the **O(f) notation**, also known as **asymptotic notation** or “**Big O notation**”, where **f** is the function of the size of the input data. The asymptotic computational complexity **O(f)** measures the order of the consumed resources (CPU time, memory, etc.) by certain algorithm expressed as function of the input data size.

Complexity can be **constant**, **logarithmic**, **linear**, **n\*log(n)**, **quadratic**, **cubic**, **exponential**, etc. This is respectively the order of constant, logarithmic, linear and so on, number of steps, are executed to solve a given problem. For simplicity, sometime instead of “**algorithms complexity**” or just “**complexity**” we use the term “**running time**”.

**Typical Algorithm Complexities**

This table will explain what every type of complexity (running time) means:

|  |  |  |
| --- | --- | --- |
| **Complexity** | **Running Time** | **Description** |
| constant | O(1) | It takes a **constant number of steps** for performing a given operation (for example 1, 5, 10 or other number) and this count does not depend on the size of the input data. |
| logarithmic | O(log(N)) | It takes the order of **log(N) steps**, where the base of the logarithm is most often 2, for performing a given operation on N elements. For example, if N = 1,000,000, an algorithm with a complexity O(log(N)) would do about 20 steps (with a constant precision). Since the base of the logarithm is not of a vital importance for the order of the operation count, it is usually omitted. |
| linear | O(N) | It takes nearly the **same amount of steps as the number of elements** for performing an operation on N elements. For example, if we have 1,000 elements, it takes about 1,000 steps. Linear complexity means that the number of elements and the number of steps are linearly dependent, for example the number of steps for N elements can be N/2 or 3\*N. |
|  | O(n\*log(n)) | It takes **N\*log(N) steps** for performing a given operation on N elements. For example, if you have 1,000 elements, it will take about 10,000 steps. |
| quadratic | O(n2) | It takes the order of **N2 number** of steps, where the N is the size of the input data, for performing a given operation. For example if N = 100, it takes about 10,000 steps. Actually we have a quadratic complexity when the number of steps is in quadratic relation with the size of the input data. For example for N elements the steps can be of the order of 3\*N2/2. |
| cubic | O(n3) | It takes the order of **N3 steps**, where N is the size of the input data, for performing an operation on N elements. For example, if we have 100 elements, it takes about 1,000,000 steps. |
| exponential | O(2n), O(N!), O(nk), … | It takes a number of steps, which is with an **exponential** dependability with the size of the input data, to perform an operation on N elements. For example, if N = 10, the exponential function 2N has a value of 1024, if N = 20, it has a value of 1 048 576, and if N = 100, it has a value of a number with about 30 digits. The exponential function N! grows even faster: for N = 5 it has a value of 120, for N = 10 it has a value of 3,628,800 and for N = 20 – 2,432,90,008,176,640,000. |

When evaluating complexity, **constants are not taken into account**, because they do not significantly affect the count of operations. Therefore an algorithm which does N steps and algorithms which do N/2 or 3\*N respectively are considered linear and approximately equally efficient, because they perform a number of operations which is of the same order.

**Complexity and Execution Time**

The **execution speed** of a program depends on the complexity of the algorithm, which is executed. If this complexity is low, the program will execute fast even for a big number of elements. If the complexity is high, the program will execute slowly or will not even work (it will hang) for a big number of elements.

If we take an average computer from 2008, we can assume that it can perform about **50,000,000 elementary operations per second**. This number is a rough approximation, of course. The different processors work with a different speed and the different elementary operations are performed with a different speed, and also the computer technology constantly evolves. Still, if we accept we use an average home computer from 2008, we can make the following conclusions about the **speed of execution** of a given program depending on the algorithm complexity and size of the input data.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm** | **10** | **20** | **50** | **100** | **1,000** | **10,000** | **100,000** |
| O(1) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. |
| O(log(n)) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. |
| O(n) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. |
| O(n\*log(n)) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. |
| O(n2) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | 2 sec. | 3-4 min. |
| O(n3) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | 20 sec. | 5.55 hours | 231.5 days |
| O(2n) | < 1 sec. | < 1 sec. | 260 days | hangs | hangs | hangs | hangs |
| O(n!) | < 1 sec. | hangs | hangs | hangs | hangs | hangs | hangs |
| O(nn) | 3-4 min. | hangs | hangs | hangs | hangs | hangs | hangs |