**SECTION 5. COMPLEXITY AND STABILITY ANALYSIS OF ALGORITHM**

The previous sections focused on the description of the most common data structures, on the one hand as abstract entities, and on the other - their concrete implementation in the programming language Golang. This section will deal with the theoretical analysis of algorithms for processing data structures from the perspective of estimating the time spent on program code execution and computer memory usage.

In theory and in practice, data structure processing algorithms are defined as a set of sequentially performed, stable procedures that provide a final computational result based on a fixed and limited set of source data. Most data processing algorithms include basic data processing procedures such as searching, sorting, adding, updating, deleting, etc.

Foremost, it should be noted that a "good algorithm" must have two characteristics: a) correctness and b) finality. *Correctness* means that if an algorithm is designed to solve a particular problem, it must always produce the correct result for all the source data and no input data will get the wrong result. The *finiteness* of the algorithm means that a finite number of computational operations must be performed to obtain the result, i.e. the execution of the program code must stop at some point in time.

Once the algorithm is correct and finite, the most important characteristic of the algorithm is efficiency, which is achieved by meeting the following requirements:

1. The algorithm should effectively use the resources available to the system;
2. The computational time (time taken to generate an output data corresponding to a particular input ones) should be minimal;
3. The memory used by the algorithm should also be as small as possible.

In most computational tasks, it is necessary to strive for a compromise between computation time and occupied memory. In other words, when choosing a method for solving a computational problem, you need to decide on priorities: what is more important - the computation time or the amount of memory occupied. It may seem that this is not so important for modern computers. However, when solving complex problems associated with the processing of huge amounts of data, for example, for artificial intelligence problems or for bioinformatics problems (for example, genome decoding), the search for the optimal combination of computing time and computer memory becomes decisive. In this regard, we will define the term "algorithm complexity" as a measure of the amount of time and/or space required to solve a problem, depending on the size of the data.

Such analysis involves determining a function that links the length of the input data of the algorithm to the time it takes (its *time complexity*) or the number of storage locations it uses (*the complexity of the space*). The algorithm is considered effective when the values of this function are either negligible or slowly increasing relative to the size of the input data.

**5.2. Theoretical analysis of algorithm complexity**

Ultimately, the created algorithms must be implemented on some computing device using the generated program code. Therefore, one should distinguish between the complexity of the algorithm itself and the complexity of its computer implementation. The fastest algorithm implemented on a slow computing device may be less efficient than a less successful algorithm implemented on a computer with more processing power or a programming language capable of parallel computing.

Thus, when assessing the complexity of the algorithms, they are carried out on an abstract machine with random access to memory, which makes it possible not to take into account the low-level parameters of the computing device (processor memory size, multitasking, etc.). The model of such a machine consists of memory and a processor, which work as follows:

* + - memory consists of cells, each of which has an address and can store one data element;
    - each memory call takes one unit of time, regardless of the number of the addressable cell.
    - the amount of memory is sufficient to execute any algorithm;
    - the processor performs any elementary operation in one time step;
    - loops and functions are not considered elementary operations.

A Labor intensity  of the Tn algorithm associated with estimates of its complexity is determined by counting the number of operations performed. For example, consider the algorithm for finding the maximum element of an array.

package main

import "fmt"

func main() {

 array:= []int{11, 9, 17, 45, 411}

N := len(array)

 maxNumber := array[0]

 for item:= 1; item < N; item++ {

    if array[item] > maxNumber {

    maxNumber = array[item]

  }

 }

 fmt.Println("The smallest value of an array element = ", maxNumber)

}

When implementing this algorithm, the following will be performed:

1. (N - 1) the operation of assigning a new value to the loop counter i;

2. (N - 1) the operation of comparing the counter with the value N;

3. (N - 1) the operation of comparing an array element with the value of maxNumber;

4. from 1 to N operations of assigning a value to the variable maxNumber.

It is apparent that such a calculation for determining the complexity of the algorithm and estimating its complexity is individual and depends on the size of the input data. Therefore, it was quite natural to propose a qualitative assessment of complexity: depending on the “quality” of the initial data - the best option, the worst and the average.

5.3. Complexity qualitative assessment Asymptotic notations such as omega-notation, theta-notation, and Big-O notation that reflect the three time bounds of the algorithm (lower, middle, and upper) are used to quantify the complexity of algorithms. Omega notation is the lower bound of the algorithm’s running time, which characterizes the best algorithm complexity in terms of achieving efficiency. For any value of n of the source data, the minimum time required by the algorithm is set to Ω(f(n)).

The Big-O notation represents the upper bound on the execution time of the algorithm, which characterizes the worst complexity of the algorithm in terms of efficiency. In most practical cases, this notation is used to evaluate the effectiveness of algorithms, since it is the worst-case scenario of the computational process that the software developer must evaluate. Finally, theta notation (Θ-notation) is an upper and lower bound on the execution time of algorithms and is used to analyze their average complexity. Consider in more detail the Big O notation that determines the upper bound of any algorithm, that is, the algorithm cannot take longer than the upper bound. To begin with the simplest time-constant algorithm is an algorithm that requires the same amount of time, regardless of its input data, whose complexity is denoted as O(1). For example, given two numbers, you need to calculate the sum. This is followed by the logarithmic time complexity O(log n). When the time taken by the algorithm is proportional to the logarithm of input size n, it is said to have logarithmic time complexity. An example is the simple calculation of the number of print operations of the index cycle:

func main() {

    n := 16

    for i:= 1; i < n; i=i\*2 {

        fmt.Println("i = ", i)

        }

    }

Recall that log216 = log224 = 4 log22 = 4 since log22 = 1. Algorithms with logarithmic complexity are considered highly efficient because the ratio of execution time of one operation to the size of the data array decreases with its increasing size. Algorithms that work in logarithmic time are usually found in binary tree operations or binary search operations. Such algorithms will be discussed in the next section.

Linear time complexity (O) characterizes algorithms whose execution time linearly depends on the size of the input data (n). The simplest example is the problem of computing the sum of n numbers:

func main() {

    n := 16

    sum := 0

    for i:= 0; i < n; i++ {

        sum = sum + i

    }

    fmt.Println("sum =", sum)

}

Typical examples of linear time complexity are algorithms associated with arrays sorting, which will be shown in the next section. It is important to note that linear complexity algorithms do not require additional memory. This is followed by the quadratic complexity of O(n2) algorithms, the number of operations in which is quadratic depending on the size of the input data. As an example we can give the algorithm of multiplication of two numbers in the interval [1:n;1:m]:

func main() {

    n := 3

    m := 5

    pow := 1

    for i:= 1; i < n; i++ {

        for j:= 1; j < m; j++ {

            pow = pow\*i\*j

        }

    }

    fmt.Println("pow =", pow)

}

The comparison of the discussed copmlexities is presented in the form of a table. Table. 5.1. Comparison of general copmlexities

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | Сonstant  О(1) | Logarithmic  O(log n) | Linary  O(n) | Linear logarithmic O(n log n) | Quadratic  O(n2) |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 2 | 2 | 4 |
| 4 | 1 | 2 | 4 | 8 | 16 |
| 8 | 1 | 3 | 8 | 24 | 64 |
| 16 | 1 | 4 | 16 | 64 | 256 |

Or in graphic form (Figure.5.1.).

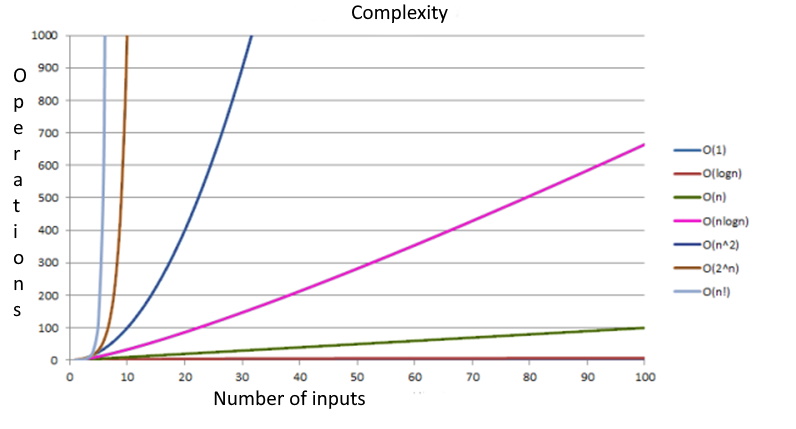


Figure 5.1. Comparison of general copmlexities

As shown in this table and graph, as the complexity of the function increases, the number of computations or the time required to perform the function can increase significantly. Next, the algorithms will be described with estimates of their complexity. Another characteristic of the computational complexity of algorithms is *the spatial complexity*, estimated by the amount of memory required by the algorithm in its life cycle. This space consists of fixed and variable parts [Vasiliev]. A fixed part is the space required to store simple variables and constants, as well as the program size, which is independent of the complexity of the task. A part variable is the space required by variables, the size of which depends entirely on the size of the task. For example, recursion stack space, dynamic memory allocation, etc.

Spatial complexity is evaluated as O(1) if only the input data set is used. If the problem requires an auxiliary memory of the same size as the input memory (the memory to store the original array), then the spatial complexity is evaluated as O(n2). It is especially important to know whether the variety used is stable. Especially in situations where your data already has some order that you would like to maintain when you sort it with a different sort key. For example, a spreadsheet has rows that contain student data, which is sorted by name by default. You'd also want to sort it by class, keeping the name order in order.

In other hand, sorting stability doesn't matter when the sort keys of the objects in the collection are the objects themselves – an array of integers or strings, for example, because we can't tell the difference between duplicate keys.