Complexity

Common complexities of algorithms

It is the common types that are considered here, since it is hardly possible to consider all options. It all depends on the algorithm you are evaluating. There may always be some additional variable (not a constant) that needs to be taken into account in the Big O function.

**Constant - O(1).**

It means that the computational complexity of the algorithm does not depend on the input data. However, this does not mean that the algorithm is executed in a single operation or requires very little time. This means that the time does not depend on the input data.

**Example No. 1.**

We have an array of 5 numbers and we need to get the first element.

val nums = intArrayOf(1, 2, 3, 4, 5)

val firstNumber = nums[0]

How much will the number of operations increase with an increase in the size of the input parameters?

Nina is a few. Even if the array consists of 100, 1000 or 10,000 elements, we still need one operation.

**Example No. 2.**

The addition of two numbers. The function always performs a constant number of operations.

fun pairSum(a: Int, b: Int) = a + b

**Linear - O(n).**

This means that the complexity of the algorithm increases linearly with increasing input data. In other words, doubling the size of the input data will double the time required to execute the algorithm.

Such algorithms are easily recognized by the presence of a loop for each element of the array.

**Example #1 is a recursive function.**

added sum(n: Int): Int {

if (n == 1) returns 1

returns n + sum(n - 1)

}

**Example # 2 is a linear function.**

fun pairSum(n: Int): Int {

var sum = 0

for (i in 0 to n) {

sum += pairSum (i, i + 1)

}

Return sum

}

pairSum (a: Int, b: Int) = a + b

The quadratic is O(n2), O(n^2).

This means that doubling the size of the input data increases the execution time by 4 times. For example, if the data is increased by 10 times, the number of operations (and execution time) will increase by about 100 times. If the algorithm has quadratic complexity, then this is a reason to reconsider the need to use this algorithm. But sometimes it can't be avoided.

Such algorithms are easily recognized by nested loops.

Example No. 1.

Fun printPair(nums: List<Int) {

for (i in nums) {

for (j in nums) {

println(nums[i], nums[j])

}

}

}

**Cheat sheet**

Small hints that will help you determine the complexity of the algorithm.

Getting a collection item is O(1). Whether it's getting by index in an array, or by key in a dictionary in Big O notation, it will be O(1).

Iterating through a collection is O(n).

Nested loops over the same collection are O(n2).

Divide and Conquer is always O(log n).

Iterations that use Divide and Conquer are O(n log n).