Chapter 3

Time complexity

Use of time complexity makes it easy to estimate the running time of a program. Performing an accurate calculation of a program's operation time is a very labour-intensive process (it depends on the compiler and the type of computer or speed of the processor). Therefore, we will not make an accurate measurement; just a measurement of a certain order of magnitude.

Complexity can be viewed as the maximum number of primitive operations that a program may execute. Regular operations are single additions, multiplications, assignments etc. We may leave some operations uncounted and concentrate on those that are performed the largest number of times. Such operations are referred to as *dominant*.

The number of dominant operations depends on the specific input data. We usually want to know how the performance time depends on a particular aspect of the data. This is most frequently the data size, but it can also be the size of a square matrix or the value of some input variable.

3.1: Which is the dominant operation?

```
1   def dominant(n):
2       result = 0
3       for i in xrange(n):
4           result += 1
5       return result
```

The operation in line 4 is dominant and will be executed n times. The complexity is described in Big-O notation: in this case $O(n)$ — *linear* complexity.

The complexity specifies the order of magnitude within which the program will perform its operations. More precisely, in the case of $O(n)$, the program may perform $c \cdot n$ operations, where c is a constant; however, it may not perform n^2 operations, since this involves a different order of magnitude of data. In other words, when calculating the complexity we omit constants: i.e. regardless of whether the loop is executed $20 \cdot n$ times or $\frac{n}{5}$ times, we still have a complexity of $O(n)$, even though the running time of the program may vary. When analyzing the complexity we must look for specific, worst-case examples of data that the program will take a long time to process.

3.1. Comparison of different time complexities

Let's compare some basic time complexities.

3.2: Constant time — $O(1)$.

```python
1  def constant(n):
2      result = n * n
3      return result
```

There is always a fixed number of operations.

3.3: Logarithmic time — $O(\log n)$.

```python
1  def logarithmic(n):
2      result = 0
3      while n > 1:
4          n //= 2
5          result += 1
6      return result
```

The value of n is halved on each iteration of the loop. If $n = 2^x$ then $\log n = x$. How long would the program below take to execute, depending on the input data?

3.4: Linear time — $O(n)$.

```python
1  def linear(n, A):
2      for i in xrange(n):
3          if A[i] == 0:
4              return 0
5      return 1
```

Let's note that if the first value of array A is 0 then the program will end immediately. But remember, when analyzing time complexity we should check for worst cases. The program will take the longest time to execute if array A does not contain any 0.

3.5: Quadratic time — $O(n^2)$.

```python
1  def quadratic(n):
2      result = 0
3      for i in xrange(n):
4          for j in xrange(i, n):
5              result += 1
6      return result
```

The result of the function equals $\frac{1}{2} \cdot (n \cdot (n + 1)) = \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n$ (the explanation is in the exercises). When calculating the complexity we are interested in a term that grows fastest, so we not only omit constants, but also other terms ($\frac{1}{2} \cdot n$ in this case). Thus we get quadratic time complexity. Sometimes the complexity depends on more variables (see example below).

3.6: Linear time — $O(n + m)$.

```python
1  def linear2(n, m):
2      result = 0
3      for i in xrange(n):
4          result += i
5      for j in xrange(m):
6          result += j
7      return result
```

Exponential and factorial time

It is worth knowing that there are other types of time complexity such as factorial time $O(n!)$ and exponential time $O(2^n)$. Algorithms with such complexities can solve problems only for very small values of n, because they would take too long to execute for large values of n.

3.2. Time limit

Nowadays, an average computer can perform 10^8 operations in less than a second. Sometimes we have the information we need about the expected time complexity (for example, Codility specifies the expected time complexity), but sometimes we do not.

The time limit set for online tests is usually from 1 to 10 seconds. We can therefore estimate the expected complexity. During contests, we are often given a limit on the size of data, and therefore we can guess the time complexity within which the task should be solved. This is usually a great convenience because we can look for a solution that works in a specific complexity instead of worrying about a faster solution. For example, if:

- $n \leqslant 1\,000\,000$, the expected time complexity is $O(n)$ or $O(n \log n)$,

- $n \leqslant 10\,000$, the expected time complexity is $O(n^2)$,

- $n \leqslant 500$, the expected time complexity is $O(n^3)$.

Of course, these limits are not precise. They are just approximations, and will vary depending on the specific task.

3.3. Space complexity

Memory limits provide information about the expected space complexity. You can estimate the number of variables that you can declare in your programs.

In short, if you have constant numbers of variables, you also have constant space complexity: in Big-O notation this is $O(1)$. If you need to declare an array with n elements, you have linear space complexity — $O(n)$.

More specifically, space complexity is the amount of memory needed to perform the computation. It includes all the variables, both global and local, dynamic pointer data-structures and, in the case of recursion, the contents of the stack. Depending on the convention, input data may also be included. Space complexity is more tricky to calculate than time complexity because not all of these variables and data-structures may be needed at the same time. Global variables exist and occupy memory all the time; local variables (and additional information kept on the stack) will exist only during invocation of the function.

3.4. Exercise

Problem: You are given an integer n. Count the total of $1 + 2 + \ldots + n$.

Solution: The task can be solved in several ways. Some person, who knows nothing about time complexity, may implement an algorithm in which the result is incremented by 1:

3.7: Slow solution — time complexity $O(n^2)$.

```
1  def slow_solution(n):
2      result = 0
3      for i in xrange(n):
4          for j in xrange(i + 1):
5              result += 1
6      return result
```

Another person may increment the result respectively by $1, 2, \ldots, n$. This algorithm is much faster:

3.8: Fast solution — time complexity $O(n)$.

```
1  def fast_solution(n):
2      result = 0
3      for i in xrange(n):
4          result += (i + 1)
5      return result
```

But the third person's solution is even quicker. Let us write the sequence $1, 2, \ldots, n$ and repeat the same sequence underneath it, but in reverse order. Then just add the numbers from the same columns:

1	2	3	...	$n-1$	n
n	$n-1$	$n-2$	...	2	1
$n+1$	$n+1$	$n+1$	...	$n+1$	$n+1$

The result in each column is $n + 1$, so we can easily count the final result:

3.9: Model solution — time complexity $O(1)$.

```
1  def model_solution(n):
2      result = n * (n + 1) // 2
3      return result
```

Every lesson will provide you with programming tasks at http://codility.com/programmers.