Computational Complexity: Section 1

By **misof** — *topcoder member*
Discuss this article in the forums

In this article I'll try to introduce you to the area of computation complexity. The article will be a bit long before we get to the actual formal definitions because I feel that the rationale behind these definitions needs to be explained as well – and that understanding the rationale is even more important than the definitions alone.

Why is it important?

Example 1. Suppose you were assigned to write a program to process some records your company receives from time to time. You implemented two different algorithms and tested them on several sets of test data. The processing times you obtained are in Table 1.

Table 1. Runtimes of two fictional algorithms.

In praxis, we probably could tell which of the two implementations is better for us (as we usually can estimate the amount of data we will have to process). For the company this solution may be fine. But from the programmer's point of view, it would be much better if he could estimate the values in Table 1 **before** writing the actual code – then he could only implement the better algorithm.

The same situation occurs during programming challenges: The size of the input data is given in the problem statement. Suppose I found an algorithm. Questions I have to answer before I start to type should be: Is my algorithm worth implementing? Will it solve the largest test cases in time? If I know more algorithms solving the problem, which of them shall I implement?

This leads us to the question: How to compare algorithms? Before we answer this question in general, let's return to our simple example. If we extrapolate the data in Table 1, we may assume that if the number of processed records is larger than 1000, algorithm 2 will be substantially faster. In other words, if we consider all possible inputs, algorithm 2 will be better for almost all of them.

It turns out that this is almost always the case – given two algorithms, either one of them is almost always better, or they are approximately the same. Thus, this will be our definition of a better algorithm. Later, as we define everything formally, this will be the general idea behind the definitions.

A neat trick

If you thing about Example 1 for a while, it shouldn't be too difficult to see that there is an algorithm with runtimes similar to those in Table 2:

Table 2. Runtimes of a new fictional algorithm.

The idea behind this algorithm: Check the number of records. If it is small enough, run algorithm 1, otherwise run algorithm 2.

Similar ideas are often used in praxis. As an example consider most of the sort() functions provided by various libraries. Often this function is an implementation of QuickSort with various improvements, such as:

- if the number of elements is too small, run InsertSort instead (as InsertSort is faster for small inputs)
- if the pivot choices lead to poor results, fall back to MergeSort

[What is efficiency?]

Example 2. Suppose you have a concrete implementation of some algorithm. (The example code presented below is actually an implementation of MinSort – a slow but simple sorting algorithm.)

```
for (int i=0; i<N; i++)
  for (int j=i+1; j<N; j++)
    if (A[i] > A[j])
      swap( A[i], A[j] );
```

If we are given an input to this algorithm (in our case, the array A and its size N), we can exactly compute the number of steps our algorithm does on this input. We could even count the processor instructions if we wanted to. However, there are too many possible inputs for this approach to be practical.

And we still need to answer one important question: What is it exactly we are interested in? Most usually it is the behavior of our program in the **worst possible case** – we need to look at the input data and to determine an upper bound on how long will it take if we run the program.

But then, what is the worst possible case? Surely we can always make the program run longer simply by giving it a larger input. Some of the more important questions are: What is the worst input with 700 elements? **How fast** does the maximum runtime grow when we increase the input size?

[Formal notes on the input size]

What exactly is this "input size" we started to talk about? In the formal definitions this is the size of the input written in some fixed finite alphabet (with at least 2 "letters"). For our needs, we may consider this alphabet to be the numbers 0..255. Then the "input size" turns out to be exactly the size of the input file in bytes.

Usually a part of the input is a number (or several numbers) such that the size of the input is proportional to the number.

E.g. in Example 2 we are given an int N and an array containing N ints. The size of the input file will be roughly 5N (depending on the OS and architecture, but always linear in N).

In such cases, we may choose that this number will represent the size of the input. Thus when talking about problems on arrays/strings, the input size is the length of the array/string, when talking about graph problems, the input size depends both on the number of vertices (N) and the number of edges (M), etc.

We will adopt this approach and use N as the input size in the following parts of the article.

There is one tricky special case you sometimes need to be aware of. To write a (possibly large) number we need only logarithmic space. (E.g. to write 123456, we need only roughly $\log_{10}(123456)$ digits.) This is why the naive primality test does not run in polynomial time – its runtime is polynomial in the **size** of the number, but not in its **number of digits**! If you didn't understand the part about polynomial time, don't worry, we'll get there later.

How to measure efficiency?

We already mentioned that given an input we are able to count the number of steps an algorithm makes simply by simulating it. Suppose we do this for all inputs of size at most N and find the worst of these inputs (i.e. the one that causes the algorithm to do the most steps). Let $f(N)$ be this number of steps. We will call this function the time complexity, or shortly the runtime of our algorithm.

In other words, if we have any input of size N, solving it will require at most $f(N)$ steps.

Let's return to the algorithm from Example 2. What is the worst case of size N? In other words, what array with N elements will cause the algorithm to make the most steps? If we take a look at the algorithm, we can easily see that:

- the first step is executed exactly N times
- the second and third step are executed exactly $N(N-1)/2$ times
- the fourth step is executed at most $N(N-1)/2$ times

Clearly, if the elements in A are in descending order at the beginning, the fourth step will always be executed. Thus in this case the algorithm makes $3N(N-1)/2 + N = 1.5N^2 - 0.5N$ steps. Therefore our algorithm has $f(N) = 1.5N^2 - 0.5N$.

As you can see, determining the exact function *f* for more complicated programs is painful. Moreover, it isn't even necessary. In our case, clearly the -0.5*N* term can be neglected. It will usually be much smaller than the $1.5N^2$ term and it won't affect the runtime significantly. The result "$f(N)$ is roughly equal to $1.5N^2$" gives us all the information we need. As we will show now, if we want to compare this algorithm with some other algorithm solving the same problem, even the constant 1.5 is not that important.

Consider two algorithms, one with the runtime N^2, the other with the runtime $0.001N^3$. One can easily see that for *N* greater than 1 000 the first algorithm is faster – and soon this difference becomes apparent. While the first algorithm is able to solve inputs with $N = 20\,000$ in a matter of seconds, the second one will already need several minutes on current machines.

Clearly this will occur always when one of the runtime functions grows **asymptotically faster** than the other (i.e. when *N* grows beyond all bounds the limit of their quotient is zero or infinity). Regardless of the constant factors, an algorithm with runtime proportional to N^2 will always be better than an algorithm with runtime proportional to N^3 **on almost all inputs**. And this observation is exactly what we base our formal definition on.

Finally, formal definitions

Let *f*, *g* be positive non-decreasing functions defined on positive integers. (Note that all runtime functions satisfy these conditions.) We say that $f(N)$ is $O(g(N))$ *(read: f is big-oh of g)* if for some *c* and N_0 the following condition holds:

In human words, $f(N)$ is $O(g(N))$, if for some *c* almost the entire graph of the function *f* is below the graph of the function *c.g*. Note that this means that *f* grows at most as fast as *c.g* does.

Instead of "$f(N)$ is $O(g(N))$" we usually write $f(N) = O(g(N))$. Note that this "equation" is **not symmetric** – the notion " $O(g(N)) = f(N)$" has no sense and " $g(N) = O(f(N))$" doesn't have to be true (as we will see later). (If you are not comfortable with this notation, imagine $O(g(N))$ to be a set of functions and imagine that there is a $\in$ instead of =.)

What we defined above is known as the big-oh notation and is conveniently used to

specify upper bounds on function growth.

E.g. consider the function $f(N) = 3N(N-1)/2 + N = 1.5N^2 - 0.5N$ from Example 2. We may say that $f(N) = O(N^2)$ (one possibility for the constants is $c = 2$ and $N_0 = 0$). This means that f doesn't grow (asymptotically) faster than N^2.

Note that even the exact runtime function f doesn't give an exact answer to the question "How long will the program run on my machine?" But the important observation in the example case is that the runtime function is quadratic. If we double the input size, the runtime will increase approximately to four times the current runtime, no matter how fast our computer is.

The $f(N) = O(N^2)$ upper bound gives us almost the same – it guarantees that the growth of the runtime function is at most quadratic.

Thus, we will use the O-notation to describe the time (and sometimes also memory) complexity of algorithms. For the algorithm from Example 2 we would say "The time complexity of this algorithm is $O(N^2)$" or shortly "This algorithm is $O(N^2)$".

In a similar way we defined O we may define Ω and Θ.

We say that $f(N)$ is $\Omega(g(N))$ if $g(N) = O(f(N))$, in other words if f grows at least as fast as g.

We say that $f(N) = \Theta(g(N))$ if $f(N) = O(g(N))$ and $g(N) = O(f(N))$, in other words if both functions have approximately the same rate of growth.

As it should be obvious, Ω is used to specify lower bounds and Θ is used to give a tight asymptotic bound on a function. There are other similar bounds, but these are the ones you'll encounter most of the time.

Some examples of using the notation

- $1.5N^2 - 0.5N = O(N^2)$.
- $47N \log N = O(N^2)$.
- $N \log N + 1\,000\,047N = \Theta(N \log N)$.
- All polynomials of order k are $O(N^k)$.
- The time complexity of the algorithm in Example 2 is $\Theta(N^2)$.
- If an algorithm is $O(N^2)$, it is also $O(N^5)$.
- Each comparision-based sorting algorithm is $\Omega(N \log N)$.
- MergeSort run on an array with N elements does roughly $N \log N$ comparisions. Thus the time complexity of MergeSort is $\Theta(N \log N)$. If we trust the previous statement, this means that MergeSort is an asymptotically optimal general sorting algorithm.

- The algorithm in Example 2 uses $\Theta(N)$ bytes of memory.
- The function giving my number of teeth in time is $O(1)$.
- A naive backtracking algorithm trying to solve chess is $O(1)$ as the tre of positions it will examine is finite. (But of course in this case the constant hidden behind the $O(1)$ is unbelievably large.)
- The statement "Time complexity of this algorithm is at least $O(N^2)$" is meaningless. (It says: "Time complexity of this algorithm is at least at most roughly quadratic." The speaker probably wanted to say: "Time complexity of this algorithm is $\Omega(N^2)$.")

When speaking about the time/memory complexity of an algorithm, instead of using the formal $\Theta(f(n))$-notation we may simply state the class of functions f belongs to. E.g. if $f(N) = \Theta(N)$, we call the algorithm *linear*. More examples:

- $f(N) = \Theta(\log N)$: logarithmic
- $f(N) = \Theta(N^2)$: quadratic
- $f(N) = \Theta(N^3)$: cubic
- $f(N) = O(N^k)$ for some k: polynomial
- $f(N) = \Omega(2^N)$: exponential

For graph problems, the complexity $\Theta(N + M)$ is known as "linear in the graph size".

Determining execution time from an asymptotic bound

For most algorithms you may encounter in praxis, the constant hidden behind the O (or Θ) is usually relatively small. If an algorithm is $\Theta(N^2)$, you may expect that the exact time complexity is something like $10N^2$, not 10^7N^2.

The same observation in other words: if the constant is large, it is usually somehow related to some constant in the problem statement. In this case it is good practice to give this constant a name and to include it in the asymptotic notation.

An example: The problem is to count occurences of each letter in a string of N letters. A naive algorithm passes through the whole string once for each possible letter. The size of alphabet is fixed (e.g. at most 255 in C), thus the algorithm is linear in N. Still, it is better to write that its time complexity is $\Theta(|S|.N)$, where S is the alphabet used. (Note that there is a better algorithm solving this problem in $\Theta(|S| + N)$.)

In a topcoder contest, an algorithm doing 1 000 000 000 multiplications runs barely in time. This fact together with the above observation and some experience with topcoder problems can help us fill the following table:

Table 3. Approximate maximum problem size solvable in 8 seconds.

A note on algorithm analysis

Usually if we present an algorithm, the best way to present its time complexity is to give a Θ-bound. However, it is common practice to only give an O-bound - the other bound is usually trivial, O is much easier to type and better known. Still, don't forget that O represents only an upper bound. Usually we try to find an O-bound that's as good as possible.

Example 3. Given is a sorted array A. Determine whether it contains two elements with the difference D. Consider the following code solving this problem:

```
int j=0;
for (int i=0; i<N; i++) {
  while ( (j<N-1) && (A[i]-A[j] > D) )
    j++;
  if (A[i]-A[j] == D) return 1;
}
```

It is easy to give an $O(N^2)$ bound for the time complexity of this algorithm – the inner while-cycle is called N times, each time we increase j at most N times. But a more careful analysis shows that in fact we can give an $O(N)$ bound on the time complexity of this algorithm – it is sufficient to realize that during the **whole execution** of the algorithm the command "`j++;`" is executed no more than N times.

If we said "this algorithm is $O(N^2)$", we would have been right. But by saying "this algorithm is $O(N)$" we give more information about the algorithm.

Conclusion

We have shown how to write bounds on the time complexity of algorithms. We have also demonstrated why this way of characterizing algorithms is natural and (usually more-or-less) sufficient.

The next logical step is to show how to estimate the time complexity of a given algorithm. As we have already seen in Example 3, sometimes this can be messy. It gets really messy when recursion is involved. We will address these issues in the second part of this article.

...continue to Section 2