Complexity of Algorithms

Victor Adamchik

1.1 Introduction

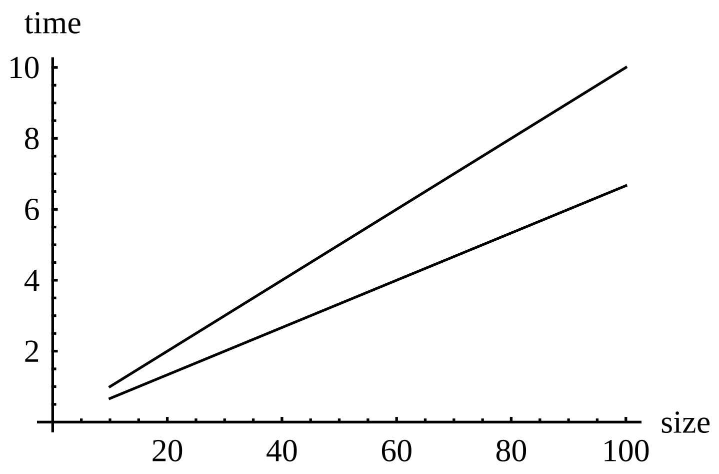
Algorithmic complexity is concerned about how fast or slow particular algorithm performs. We define complexity as a numerical function $T(n)$ - time versus the input size n. We want to define time taken by an algorithm without depending on the implementation details. But you agree that $T(n)$ does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc. The way around is to estimate efficiency of each algorithm asymptotically. We will measure time $T(n)$ as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

We consider a problem of addition of two n-bit binary numbers. Let $T(n)$ represent an amount of time grade school addition uses to add two n-bit numbers. We want to define "time" $T(n)$ taken by the method of addition without having to worry about implementation details. The process of addition consists of the following two steps

- adding 3 bits (one bit is a carry bit)
- writing down 2 bits (again, one bit is a carry bit)

```
  1 0 1 0
  1 1 1 1
 ---------
1 1 0 0 1
```

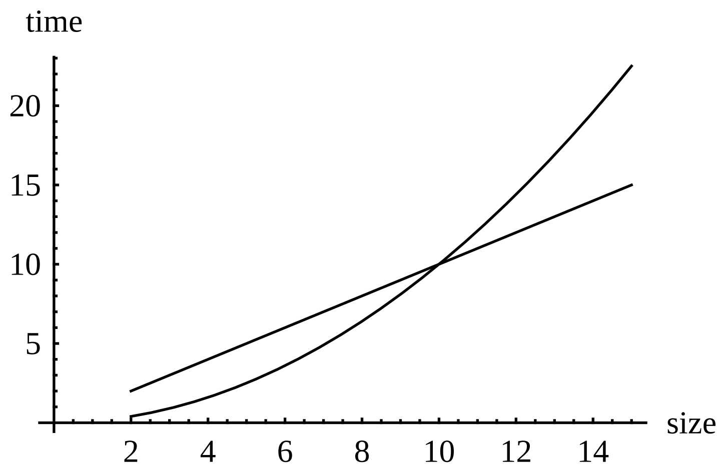
On any computer M_1, adding 3 bits and writing down 2 bits can be done in constant time c_1. By constant time we mean that the time is independent of the input size. The total time of grade school addition of two n-bit binary numbers is $T_{M_1}(n) = n * c_1$ on a computer M_1. On computer M_2, adding 3 bits and writing down 2 bits can be done in different but still constant time c_2. The total time is $T_{M_2}(n) = n * c_2$. Different machines result in different slopes, but time grows linearly as input size increases. We say that grade school addition is a linear time algorithm. Let us draw $T_{M_1}(n)$ and $T_{M_2}(n)$



The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science. As a second example, let us consider multiplication of two *n*-bit numbers.

```
          1 0 1 0
          1 1 1 1
        -------------
           1 0 1 0
         1 0 1 0
       1 0 1 0
     1 0 1 0
    ---------------------
  1 0 0 1 0 1 1 0
```

Abstracting away implementation details, the total time consists of *n* additions, each of which takes a linear time. Therefore, the multiplication time is bounded by $c\,n^2$. We say that grade school multiplication is a quadratic time algorithm. No matter how dramatic the difference in the constants, the quadratic curve will eventually dominate the linear curve:



1.2 Asymptotic Notations

The goal of computational complexity is to classify algorithms according to their performances. With each algorithm we associate a sequence of steps comprising this algorithm. We measure the run time of an algorithm by counting the number of steps, and therefore define an algorithmic complexity as a numerical function $T(n)$ where n is the input size. The term analysis of algorithms is used to describe approaches to the study of the performance of computer programs. In this course we will perform the following types of analysis:

the worst case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n.

the best case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

the average case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size n.

the amortized time complexity of the algorithm is the function defined by a sequence of operations applied to the input of size n and averaged over time.

We will represent the time function $T(n)$ using the *asymptotic notations.*

Definition. (Big-O) *We say that* $f(n) = O(g(n))$ *when there exist constants* $c > 0$ *and* $n_0 > 0$ *such that*

$$|f(n)| \leq c * |g(n)|, \ \forall\, n \geq n_0$$

Intuitively, this means that function f does not grow faster than g, or that function g is an *upper bound* for f. In the analysis of algorithms we usually dropped the absolute value around the complexity function. Here is the rewritten definition

$$(\exists\, c > 0,\ \exists\, n_0 > 0,\ \forall\, n \geq n_0),\ f(n) \leq c\, g(n)$$

In the analysis of algorithm O is often used to describe *the worst-case behavior*.

Example. Prove that

$$n^2 + 2\,n + 1 = O\left(n^2\right)$$

We must find such c and n_0 that

$$n^2 + 2\,n + 1 \leq\ c * n^2,\ \forall\, n \geq n_0$$

Let $n_0 = 1$, then $1 \leq\ n \leq\ n^2$ for $\forall n \geq 1$. Then

$$n^2 + 2\,n + 1 \leq\ n^2 + 2\,n^2 + n^2 = 4\,n^2$$

If follows that $c = 4$.

Example. Prove that

$$7\,n^2 = O(n^3)$$

Observe, that $7\,n^2 < n^3$ if $7 < n$. Therefore, we can choose $n_0 = 8$ and $c = 1$. It follows

$$7\,n^2 < 1 * n^3, \ \forall\, n \geq 8$$

Basic properties of the big-*O*

$$f(n) = O(f(n))$$

$$c * O(f(n)) = O(f(n))$$

$$O(f(n)) * O(g(n)) = O(f(n) * g(n))$$

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$

The big-O belongs to an entire family of notation. Consider two sorting algorithms: insertion sort and mergesort. The insertion sort has a running-time $O(n^2)$, and the merge-sort does it in $O(n \log_2 n)$. However, from the asymptotic point of view (prove it!)

$$O(n \log_2 n) = O(n^2)$$

This is confusing at first. Clearly, we need the notation for the <u>lower</u> *bound*. A capital omega Ω notation is used in this case.

Definition. (Big-Ω) *We say that $f(n) = \Omega(g(n))$ when there exist constant $c > 0$ that for any $n_0 > 0$ there exists an input $n \geq n_0$ such that*

$$|f(n)| \geq c * |g(n)|$$

In the analysis of algorithm Ω is often used to describe *the best-case behavior*.

Example. Prove that

$$n^2 = \Omega(n \log_2 n)$$

We must find such c and n_0 that

$$n^2 \geq c\, n \log_2 n, \ \forall\, n \geq n_0$$

Let $n_0 = 1$ then for $\forall n \geq 1$ we know from calculus course that

$$n \geq \log_2 n$$

Multiplying it by n

$$n^2 \geq 1 * n \log_2 n$$

we find that $c = 1$.

Lemma (the duality rule) $g(n) = \Omega(f(n)) \iff f(n) = O(g(n))$.

Proof.

$$g(n) = \Omega(f(n)) \Longrightarrow g(n) \geq b\, f(n) \Longrightarrow f(n) \leq \frac{1}{b}\, g(n) \Longrightarrow f(n) = O(g(n))$$

To measure the complexity $T(n)$ of a particular algorithm, means to find the upper and lower bounds. A new notation is used in this case.

Definition. (Big-Θ) *We say that* $f(n) = \Theta(g(n))$ *if and only* $f(n) = O(g(n))$ *and* $g(n) = O(f(n))$

In mathematical notation

$$(\exists\, c_1,\ c_2 > 0,\ \exists\, n_0 > 0,\ \forall\, n \geq n_0)\ \ c_1\, g(n) \leq f(n) \leq c_2\, g(n)$$

Example. Prove that

$$n = \Omega(2\, n)$$

We must find c_1, c_2 and n_0 such that

$$c_1 (2\, n) \leq n \leq c_2 * (2\, n)$$

Choose $n_0 = 1$ and $c_1 = 1/4$, $c_2 = 1$.

Example. Show that

$$f(n) = n \log n + O(n) \Longrightarrow f(n) = \Theta(n \log n)$$

$f(n) =$

$$n \log n + O(n) \Longrightarrow f(n) - n \log n < c_1\, n \Longrightarrow f(n) < c_1\, n + n \log n < n \log n$$

Since $n < n \log n$, we have

$$f(n) < c_1\, n + n \log n < c_1\, n \log n + n \log n = (c_1 + 1)\, n \log n$$

On the other hand,

$$f(n) = n \log n + O(n) \Longrightarrow f(n) > c_2\, n \log n$$

Combining the last two inequalities, we obtain

$$c_2\, n \log n < f(n) < (c_1 + 1)\, n \log n$$

This proves, that

$$f(n) = \Theta(n \log n)$$

Exercise. Is the following implication correct?

$$g(n) = \Omega(f(n)) \Longrightarrow 2^{g(n)} = \Omega\left(2^{f(n)}\right)$$

In many practical application it makes sense to define another measure for running time, when the input is randomly drawn.

Definition. (average case) *The average running time on inputs of size n is*

$$T(n) = \sum_{|k|=n} p_k \, T(k)$$

where p_k is the probability of instance k.

Average case running time is often much harder to calculate then other cases.

The amortized complexity will be discussed later in the course.

Exercise. Clearly mark each of the following assertions as true or false.

$T/F \quad 10 + n + n^2 = O(n^2)$

$T/F \quad 10 + n + n^2 = \Theta(n^2)$

$T/F \quad 10 + n + n^2 = \Omega(n)$

$T/F \quad 10 + n + n^2 = \Theta(n)$

$T/F \quad 10 + n + \log n = \Omega(n)$

$T/F \quad 10 + n + \log n = O(n^2)$

$T/F \quad 7\log^2 n + 2\, n \log n = O(n)$

$T/F \quad 7\log^2 n + 2\, n \log n = \Omega(\log n)$

$T/F \quad \left(\frac{1}{3}\right)^n + 100 = O(1)$

$T/F \quad 3^n + 100 = O(1)$

$T/F \quad 2^n + 100\, n^2 + n^{100} = O(n^{101})$

1.3 The Input Size

In sorting and searching (array) algorithms, the input size is the number of items. In graph algorithms the input size is presented by $V + E$ (vertices and edges). In number-theoretic algorithms, the input size is the number of bits. In this section we consider a couple of examples of number algorithms. What is the complexity of the following algorithm?

```
bool prime(int n)  {
  int k = 2;
  while( k <= sqrt(n) )   {
      if( n%k == 0 ) return false;
      k++;
  }
  return true;
}
```

The number of passes through the loop is $\sqrt{n}$. Therefore, $T(n) = O(\sqrt{n})$. But what is n? It's a number, not an input size. For a given algorithm, the input size is defined as the number of characters it takes to write (or encode) the input. If we use a bibary base to encode the number n, it takes $O(\log_2 n)$ characters. Therefore,

$$\text{inputSize} = O(\log_2 n)$$

$$n = O\left(2^{\text{inputSize}}\right)$$

and time complexity in the input size is

$$T(n) = O(\sqrt{n}) = O\left(2^{\text{inputSize}/2}\right)$$

which is exponential. In the next example we consider time complexity of the Fibonacci algorithm

```
int fib( int n ) {
    int [] f = alloc_array(int,n+2);
    f[0]=0; f[1]=1;
    for( int k = 2; k<=n; k++)
        f[k] = f[k-1] +f[k-2];
    return f[n];
}
```

The algorithm is linear in n, but exponential in the input size (we chose 2 as encoding base)

$$T(n) = O(n) = O\left(2^{\text{inputSize}}\right)$$