We begin with an example aimed at clarifying what we mean by "algorithm" and at demonstrating how the careful selection of an algorithm may significantly affect the performance of a program.

Consider the following problem:

If *m* and *n* are integers such that *m* is less than or equal to *n*, then we use the phrase "the interval from *m* to *n*" to refer to the collection of all the integers from *m* through *n*, inclusive. For example, "the interval from 4 to 9" refers to this collection of integers: 4, 5, 6, 7, 8, and 9. Similarly, the interval from 112 to 199 consists of the integers 112, 113, 114, and so on all the way up through 197, 198, and 199. We would like to write a method that takes integers *m* and *n* as arguments (where *m* is less than or equal to *n*) and calculates the sum of all the integers in the interval from *m* to *n*.

Here is one possibility. The strategy (or algorithm) it uses is simply to take the integers in the interval one at a time and add each one onto a running total. Once the final integer has been added, the running total is the looked-for sum.

  private static int sumInterval( int m, int n )   
  {   
    int t = 0;   
    for ( int i = m ; i <= n ; i++ )   
      t += i;   
    return t;   
  }   
  
  public static void main( String[] args )   
  {   
    System.out.println( sumInterval( , ) );   
  }

[Show program details »](https://www.eimacs.com/eimacs/mainpage?cid=162149&epid=E2306674349)

19855

However, there is a well-known formula for calculating the sum of the integers in an interval. It is this:

One-half of the result of multiplying the number of integers in the interval by the sum of the first and last integers.

In the case of the interval from *m* to *n*, the interval contains *n* – *m* + 1 integers, and the first and last integers are *m* and *n*, respectively. So, according to the formula,

sum = [(n – m + 1) \* (m + n)] / 2

In our particular case, it follows that the sum of the integers from 10 through 199 is:

[(199 – 10 + 1) \* (10 + 199)] / 2  
     = [190 \* 209] / 2 = 39710 / 2 = 19855.

This algorithm may be coded as follows:

  private static int sumInterval( int m, int n )   
  {   
    return ( (n - m + 1) \* (m + n) ) / 2;   
  }   
  
  public static void main( String[] args )   
  {   
    System.out.println( sumInterval( , ) );   
  }

[Show program details »](https://www.eimacs.com/eimacs/mainpage?cid=162149&epid=E2306674349)

19855

Both algorithms achieve the desired result. How do they compare?

* The first algorithm is intuitively simple, and requires less knowledge of the mathematics involved. It is so simple in fact that it should be possible just to look at the method's definition and *see* that it correctly encodes the algorithm; we should be able to *see* that each integer in turn, from the first to the last, is added to the growing total and the final result reported. Because of this simplicity, it is very likely that a programmer, when faced with the task of coding the first algorithm, would be able to do so without making any mistakes.
* The second algorithm gives rise to a method that usually has much less work to do and is therefore probably much more efficient than the first. Provided the integers in question are not so large that an overflow occurs, the second method most likely executes very quickly because, regardless of the input integers, only five straightforward integer arithmetic operations are required. This means in particular that the execution time should be more or less constant, no matter how far apart *m* and *n* are. This contrasts with the first method, in which the further apart *m* and *n* are, the more additions have to be performed and therefore the longer the computation takes.