The [**logarithm**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-logarithm) of base bb for value yy is the power to which bb is raised to get yy. Normally, this is written as logby=xlogb⁡y=x. Thus, if logby=xlogb⁡y=x then bx=ybx=y, and blogby=yblogby=y.

Consider the [**binary search**](https://traky.cs.hut.fi/Books/CSE-A1141/html/AnalProgram.html#analprogram) algorithm for finding a given value within an array sorted by value from lowest to highest. Binary search first looks at the middle element and determines if the value being searched for is in the upper half or the lower half of the array. The algorithm then continues splitting the appropriate subarray in half until the desired value is found. How many times can an array of size (n) be split in half until only one element remains in the final subarray? The answer is ⌈log2n⌉⌈log2⁡n⌉ times.

In OpenDSA, nearly all logarithms used have a base of two. This is because data structures and algorithms most often divide things in half, or store codes with binary bits. Whenever you see the notation lognlog⁡n in OpenDSA, either log2nlog2⁡n is meant or else the term is being used asymptotically and so the actual base does not matter. Logarithms using any base other than two will show the base explicitly.

1. log(nm)=logn+logmlog⁡(nm)=log⁡n+log⁡m.
2. log(n/m)=logn−logmlog⁡(n/m)=log⁡n−log⁡m.
3. log(nr)=rlognlog⁡(nr)=rlog⁡n.
4. logan=logbn/logbaloga⁡n=logb⁡n/logb⁡a.

When discussing logarithms, exponents often lead to confusion. Property (3) tells us that logn2=2lognlog⁡n2=2log⁡n. How do we indicate the square of the logarithm (as opposed to the logarithm of n2n2)? This could be written as (logn)2(log⁡n)2, but it is traditional to use log2nlog2⁡n. On the other hand, we might want to take the logarithm of the logarithm of nn. This is written loglognlog⁡log⁡n.

A special notation is used in the rare case when we need to know how many times we must take the log of a number before we reach a value ≤1≤1. This quantity is written log∗nlog∗⁡n. For example, log∗1024=4log∗⁡1024=4 because log1024=10log⁡1024=10, log10≈3.33log⁡10≈3.33, log3.33≈1.74log⁡3.33≈1.74, and log1.74<1log⁡1.74<1, which is a total of 4 log operations.

Most programs contain loop constructs. When analyzing running time costs for programs with loops, we need to add up the costs for each time the loop is executed. This is an example of a [***summation***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-summation). Summations are simply the sum of costs for some function applied to a range of parameter values. Summations are typically written with the following "Sigma" notation:

∑i=1nf(i).

Given a summation, you often wish to replace it with an algebraic equation with the same value as the summation. This is known as a [**closed-form solution**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-closed-form-solution), and the process of replacing the summation with its closed-form solution is known as solving the summation. For example, the summation ∑ni=11∑i=1n1 is simply the expression "1" summed nn times (remember that ii ranges from 1 to nn). Because the sum of nn 1s is nn, the closed-form solution is nn.

The running time for a recursive algorithm is most easily expressed by a recursive expression because the total time for the recursive algorithm includes the time to run the recursive call(s). A [***recurrence relation***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-recurrence-relation) defines a function by means of an expression that includes one or more (smaller) instances of itself. A classic example is the recursive definition for the factorial function:

n!=(n−1)!⋅n for n>1;1!=0!=1.n!=(n−1)!⋅n for n>1;1!=0!=1.

Another standard example of a recurrence is the Fibonacci sequence:

Fib(n)=Fib(n−1)+Fib(n−2) for n>2;Fib(1)=Fib(2)=1.Fib(n)=Fib(n−1)+Fib(n−2) for n>2;Fib(1)=Fib(2)=1.

From this definition, the first seven numbers of the Fibonacci sequence are

1,1,2,3,5,8, and 13.

Recurrence relations are often used to model the cost of recursive functions. For example, the number of multiplications required by a recursive version of the factorial function for an input of size nn will be zero when n=0n=0 or n=1n=1 (the base cases), and it will be one plus the cost of calling fact on a value of n−1n−1. This can be defined using the following recurrence:

T(n)=T(n−1)+1 for n>1;T(0)=T(1)=0.

## 1.6 Recursion

An [**algorithm**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-algorithm) (or a function in a computer program) is [**recursive**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-recursion) if it calls itself to do part of its work. For this approach to be successful, the "call to itself" must be on a smaller problem then the one originally attempted. In general, a recursive algorithm must have two parts: the [**base case**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-base-case), which handles a simple input that can be solved without resorting to a recursive call, and the recursive part which contains one or more recursive calls to the algorithm where the parameters are in some sense "closer" to the base case than those of the original call. Here is a recursive function to compute the factorial of nn.