2.2. Complexity and Big-O Notation

For certain parts of the C++ standard library — especially for the STL — the performance of algorithms and member functions was considered carefully. Thus, the standard requires a certain *complexity* of them. Computer scientists use a specialized notation to express the relative complexity of an algorithm. Using this measure, one can quickly categorize the relative runtime of an algorithm, as well as perform qualitative comparisons between algorithms. This measure is called *Big-O notation*.

Big-O notation expresses the runtime of an algorithm as a function of a given input of size *n*. For example, if the runtime grows linearly with the number of elements — doubling the input doubles the runtime — the complexity is O(*n*). If the runtime is independent of the input, the complexity is O(1). Table 2.1 lists typical values of complexity and their Big-O notation.

Table 2.1. Typical Values of Complexity

Type	Notation	Meaning
Constant	O(1)	The runtime is independent of the number of elements.
Logarithmic	O($log(n)$)	The runtime grows logarithmically with respect to the number of elements.
Linear	O(n)	The runtime grows linearly (with the same factor) as the number of elements grows.
n-log-n	O($n * log(n)$)	The runtime grows as a product of linear and logarithmic complexity.
Quadratic	O(n^2)	The runtime grows quadratically with respect to the number of elements.

It is important to observe that Big-O notation hides factors with smaller exponents, such as constant factors. In particular, it doesn't matter how long an algorithm takes. Any two linear algorithms are considered equally acceptable by this measure. There may even be some situations in which the constant is so huge in a linear algorithm that even an exponential algorithm with a small constant would be preferable in practice. This is a valid criticism of Big-O notation. Just be aware that it is only a rule of thumb; the algorithm with optimal complexity is not necessarily the best one.

Table 2.2 lists all the categories of complexity with a certain number of elements to give you a feel of how fast the runtime grows with respect to the number of elements. As you can see, with a small number of elements, the running times don't differ much. Here, constant factors that are hidden by Big-O notation may have a big influence. However, the more elements you have, the bigger the differences in the running times, so constant factors become meaningless. Remember to "think big" when you consider complexity.

Table 2.2. Runtime with Respect to the Complexity and the Number of Elements

Complexity		Number of Elements							
Type	**Notation**	**1**	**2**	**5**	**10**	**50**	**100**	**1,000**	**10,000**
Constant	O(1)	1	1	1	1	1	1	1	1
Logarithmic	O($log(n)$)	1	2	3	4	6	7	10	13
Linear	O(n)	1	2	5	10	50	100	1,000	10,000
n-log-n	O($n * log(n)$)	1	4	15	40	300	700	10,000	130,000
Quadratic	O(n^2)	1	4	25	100	2,500	10,000	1,000,000	100,000,000

Some complexity definitions in the C++ reference manual are specified as *amortized*. This means that the operations *in the long term* behave as described. However, a single operation may take longer than specified. For example, if you append elements to a dynamic array, the runtime depends on whether the array has enough memory for one more element. If there is enough memory, the complexity is constant because inserting a new last element always takes the same time. However, if there is not enough memory, the complexity is linear because, depending on the number of elements, you have to allocate new memory and copy all elements. Reallocations are rather rare, so any sufficiently long sequence of that operation behaves as if each operation has constant complexity. Thus, the complexity of the insertion is "amortized" constant time.