Some time-complexity classes

We list the most commonly used time-complexity classes and a few algorithms that lie in each. For a more complete set of complexity classes, visit this webpage: https://en.wikipedia.org/wiki/Time_complexity. After the list of time-complexity classes, we give a table that vividly illustrates how the time an algorithm takes determines the sizes of input that can be feasibly handled by the algorithm.

O(1). Constant time. Algorithms whose running time does not depend on the size of the input; there is an upper bound on the number of basic steps executed.

Examples: Finding the maximum of two integers. Also, the algorithm shown in the box to the right, even though the number of basic steps executed is rather large. Also, the expected time for searching in a hash table is O(1) if the load factor is ½.

```
int s= 0;
for (int i= 1; i < 20; i++)
   for (int j= 1; i < 20; j++)
      s= s + i*j;
```

O(log n). Logarithmic time. Algorithms whose running time is logarithmic in the size `n` of its input. Note that three different logarithms are often used in math:

$\log_c x = (\log_c b)\,(\log_b x)$

If $n = 10^k$, $\log_{10} n = k$ (the base-10 log)
If $n = 2^k$, $\log_2 n = k$ (the base-2 log)
If $n = e^k$, $\log_e n = k$ (the natural log, base-`e` log, where `e` = 2.71828…)

The box to the right above shows that for any bases `c` and `b`, $\log_c x$ and $\log_b x$ differ by a constant factor, $\log_c b$. Therefore, since O(log `n`) and O(5 log `n`) (replace 5 by any positive constant) describe the same set, it doesn't matter which logarithm is meant by log `n`. However, for computer scientists, log `n` usually means $\log_2 n$.

Examples: Binary search in an array of size `n`; the fast version of exponentiation b^n for int $n \geq 0$.

O(n). Linear time. Algorithms whose running time is linear in n.

Examples: Summing the values in an array of size `n`. Linear search in an array of size `n`. Best-case execution of insertion sort (when the array is already sorted). The worst-case time for searching a hash table containing a set of size `n`.

O(n log n).
Examples: Expected- and worst-case time of mergesort (to sort an array of size `n`). Expected-case time of quicksort.

O(n^2). Quadratic time.
Examples: Expected- and worst-case time of insertion sort and selection sort. Worst-case time of quicksort.

O(n^3). Cubic time.
Examples: Multiplication of two `n` x `n` matrices, using the standard method of multiplication, is in O(n^3). But in 1969, Volker Strassen showed how to do the multiplication in time O($n^{2.807}$). Others reduced it further. The current best appears to be by Virginia Williams, who in 2014 gave an O($n^{2.373}$) algorithm. Some people conjecture that matrix multiplication is in O(n^2), and people continue to pursue this bound. In later years, will you work on this problem?

O(2^n). Exponential time.
Example: Calculating Fibonacci number `n` using the recursive definition (fib(0) = 0; fib(1) = 1; for `n` > 1, fib(`n`) = fib(`n`−1) + fib(`n`−2)) is in O(2^n). It's an infeasible way to compute Fibonacci number `n` —see the next page.

Some time-complexity classes

Comparing complexity classes

The table below gives information concerning the sizes of input that can be handled in 1 second, 1 minute, and 1 hour depending on how many basic steps an algorithm takes, assuming that a computer can execute 1,000 basic steps per second. Assume that the algorithm in set O(n) executes exactly n basic steps, that the algorithm in set O(n log n) executes exactly n log n basic steps, etc.

Suppose the algorithm is processing an array of size n. If the O(n) algorithm is used, you can run it on an array of size 1000 in 1 second. But if the exponential algorithm, the $O(2^n)$ algorithm, is used, you can run it only on arrays of size 9 or less in 1 second. Even if you could run the $O(2^n)$ algorithm for 1 hour, you could only process arrays of size 21 or less! This shows you how infeasible an $O(2^n)$ algorithm is.

O(…)	**1 second**	**1 minute**	**1 hour**
O(n)	1000	60,000	3,600,000
O(n log n)	140	4893	200,000
$O(n^2)$	31	244	1897
$O(n^3)$	10	39	153
$O(2^n)$	9	15	21