Algorithm A.3 Recursive algorithm for computing Fibonacci numbers

```
    Procedure Fibonacci (
      n
    )
1     if (n = 0 or n = 1) then
2       return (1)
3     return (Fibonacci(n − 1) + Fibonacci(n − 2))
```

Algorithm A.4 Dynamic programming algorithm for computing Fibonacci numbers

```
    Procedure Fibonacci (
      n
    )
1     F_0 ← 1
2     F_1 ← 1
3     for i = 2, . . . , n
4       F_i ← F_{i−1} + F_{i−2}
5     return (F_n)
```

and F_1, compute F_2 from F_0 and F_1, compute F_3 from F_1 and F_2, and so forth. Clearly, this process computes F_n in time $O(n)$. We can view this alternative algorithm as precomputing and then caching (or storing) the results of the intermediate computations performed on the way to each F_i, so that each only has to be performed once.

More generally, if we can define the set of intermediate computations required and how they depend on each other, we can often use this caching idea to avoid redundant computation and provide significant savings. This idea underlies most of the exact inference algorithms for graphical models.

A.3.4 Complexity Theory

In appendix A.3.3, we saw how the same problem might be solvable by two algorithms that have radically different complexities. Examples like this raise an important issue regarding the algorithm design process: If we come up with an algorithm for a problem, how do we know whether its computational complexity is the best we can achieve? In general, unfortunately, we cannot tell. There are very few classes of problems for which we can give nontrivial lower bounds on the amount of computation required for solving them.

complexity theory

However, there are certain types of problems for which we can provide, not a guarantee, but at least a certain expectation regarding the best achievable performance. *Complexity theory* has defined classes of problems that are, in a sense, equivalent to each other in terms of their computational cost. In other words, we can show that an algorithm for solving one problem can be converted into an algorithm that solves another problem. Thus, if we have an efficient algorithm for solving the first problem, it can also be used to solve the second efficiently.

The most prominent such class of problems is that of $\mathcal{NP}$-complete problems; this class