Computer Science

An Interdisciplinary Approach

Robert Sedgewick
Kevin Wayne

Princeton University

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Dynamic programming A general approach to implementing recursive programs, known as *dynamic programming*, provides effective and elegant solutions to a wide class of problems. The basic idea is to recursively divide a complex problem into a number of simpler subproblems; store the answer to each of these subproblems; and, ultimately, use the stored answers to solve the original problem. By solving each subproblem only once (instead of over and over), this technique avoids a potential exponential blow-up in the running time.

For example, if our original problem is to compute the nth Fibonacci number, then it is natural to define $n + 1$ subproblems, where subproblem i is to compute the ith Fibonacci number for each $0 \le i \le n$. We can solve subproblem i easily if we already know the solutions to smaller subproblems—specifically, subproblems $i-1$ and $i-2$. Moreover, the solution to our original problem is simply the solution to one of the subproblems—subproblem n.

Top-down dynamic programming. In *top-down* dynamic programming, we store or *cache* the result of each subproblem that we solve, so that the next time we need to solve the same subproblem, we can use the cached values instead of solving the subproblem from scratch. For our Fibonacci example, we use an array `f[]` to store the Fibonacci numbers that have already been computed. We accomplish this in Java by using a `static` variable, also known as a *class variable* or *global variable*, that is declared outside of any method. This allows us to save information from one function call to the next.

```
public class TopDownFibonacci
{
   private static long[] f = new long[92];

   public static long fibonacci(int n)
   {
      if (n == 0) return 0;
      if (n == 1) return 1;
      if (f[n] > 0) return f[n];
      f[n] = fibonacci(n-1) + fibonacci(n-2);
      return f[n];
   }
}
```

cached values

static variable (declared outside of any method)

return cached value (if previously computed)

compute and cache value

Top-down dynamic programming approach for computing Fibonacci numbers

Top-down dynamic programming is also known as *memoization* because it avoids duplicating work by *remembering* the results of function calls.

Bottom-up dynamic programming. In *bottom-up* dynamic programming, we compute solutions to *all* of the subproblems, starting with the "simplest" subproblems and gradually building up solutions to more and more complicated subproblems. To apply bottom-up dynamic programming, we must *order* the subproblems so that each subsequent subproblem can be solved by combining solutions to subproblems earlier in the order (which have already been solved). For our Fibonacci example, this is easy: solve the subproblems in the order 0, 1, and 2, and so forth. By the time we need to solve subproblem i, we have already solved all smaller subproblems—in particular, subproblems $i-1$ and $i-2$.

```
public static long fibonacci(int n)
{
   long[] f = new int[n+1];
   f[0] = 0;
   f[1] = 1;
   for (int i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

When the ordering of the subproblems is clear, and space is available to store all the solutions, bottom-up dynamic programming is a very effective approach.

NEXT, WE CONSIDER A MORE SOPHISTICATED application of dynamic programming, where the order of solving the subproblems is not so clear (until you see it). Unlike the problem of computing Fibonacci numbers, this problem would be much more difficult to solve without thinking recursively and also applying a bottom-up dynamic programming approach.

Longest common subsequence problem. We consider a fundamental string-processing problem that arises in computational biology and other domains. Given two strings x and y, we wish to determine how *similar* they are. Some examples include comparing two DNA sequences for homology, two English words for spelling, or two Java files for repeated code. One measure of similarity is the length of the *longest common subsequence* (LCS). If we delete some characters from x and some characters from y, and the resulting two strings are equal, we call the resulting string a *common subsequence*. The LCS problem is to find a common subsequence of two strings that is as long as possible. For example, the LCS of GGCACCACG and ACGGCGGATACG is GGCAACG, a string of length 7.

Algorithms to compute the LCS are used in data comparison programs like the `diff` command in Unix, which has been used for decades by programmers wanting to understand differences and similarities in their text files. Similar algorithms play important roles in scientific applications, such as the Smith–Waterman algorithm in computational biology and the Viterbi algorithm in digital communications theory.

Longest common subsequence recurrence. Now we describe a recursive formulation that enables us to find the LCS of two given strings `s` and `t`. Let `m` and `n` be the lengths of `s` and `t`, respectively. We use the notation `s[i..m)` to denote the *suffix* of `s` starting at index `i`, and `t[j..n)` to denote the suffix of `t` starting at index `j`. On the one hand, if `s` and `t` begin with the same character, then the LCS of `x` and `y` contains that first character. Thus, our problem reduces to finding the LCS of the suffixes `s[1..m)` and `t[1..n)`. On the other hand, if `s` and `t` begin with different characters, both characters cannot be part of a common subsequence, so we can safely discard one or the other. In either case, the problem reduces to finding the LCS of two strings—either `s[0..m)` and `t[1..n)` or `s[1..m)` and `t[0..n)`—one of which is strictly shorter. In general, if we let `opt[i][j]` denote the length of the LCS of the suffixes `s[i..m)` and `t[j..m)`, then the following recurrence expresses `opt[i][j]` in terms of the length of the LCS for shorter suffixes.

```
                0                                 if i = m or j = n
opt[i][j] =     opt[i+1, j+1] + 1                 if s[i] = t[j]
                max(opt[i, j+1], opt[i+1, j])     otherwise
```

Dynamic programming solution. `LongestCommonSubsequence` (PROGRAM 2.3.6) begins with a bottom-up dynamic programming approach to solving this recurrence. We maintain a two-dimensional array `opt[i][j]` that stores the length of the LCS of the suffixes `s[i..m)` and `t[j..n)`. Initially, the bottom row (the values for `i = m`) and the right column (the values for `j = n`) are 0. These are the initial values. From the recurrence, the order of the rest of the computation is clear: we start with `opt[m][n]`. Then, as long as we decrease either `i` or `j` or both, we know that we will have computed what we need to compute `opt[i][j]`, since the two options involve an `opt[][]` entry with a larger value of `i` or `j` or both. The method `lcs()` in PROGRAM 2.3.6 COMPUTES the elements in `opt[][]` by filling in values in rows from bottom to top (`i = m-1` to `0`) and from right to left in each row (`j = n-1` to `0`). The alternative choice of filling in values in columns from right to left and

Program 2.3.6 Longest common subsequence

```
public class LongestCommonSubsequence
{
   public static String lcs(String s, String t)
   {  // Compute length of LCS for all subproblems.
      int m = s.length(), n = t.length();
      int[][] opt = new int[m+1][n+1];
      for (int i = m-1; i >= 0; i--)
         for (int j = n-1; j >= 0; j--)
         if (s.charAt(i) == t.charAt(j))
            opt[i][j] = opt[i+1][j+1] + 1;
         else
            opt[i][j] = Math.max(opt[i+1][j], opt[i][j+1]);

      // Recover LCS itself.
      String lcs = "";
      int i = 0, j = 0;
      while(i < m && j < n)
      {
         if (s.charAt(i) == t.charAt(j))
         {
            lcs += s.charAt(i);
            i++;
            j++;
         }
         else if (opt[i+1][j] >= opt[i][j+1]) i++;
         else                                 j++;
      }
      return lcs;
   }

   public static void main(String[] args)
   {  StdOut.println(lcs(args[0], args[1]));  }
}
```

s, t	*two strings*
m, n	*lengths of two strings*
opt[i][j]	*length of LCS of x[i..m) and y[j..n)*
lcs	*longest common subsequence*

The function `lcs()` computes and returns the LCS of two strings `s` and `t` using bottom-up dynamic programming. The method call `s.charAt(i)` returns character `i` of string `s`.

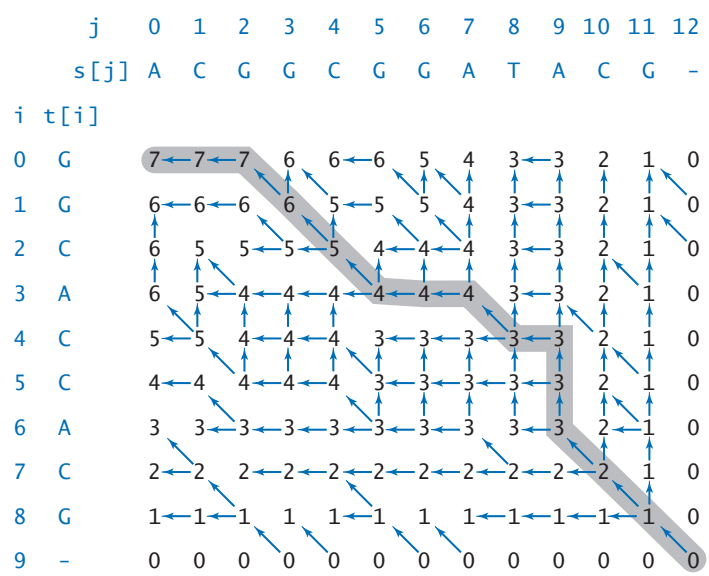
```
% java LongestCommonSubsequence GGCACCACG ACGGCGGATACG
GGCAACG
```

from bottom to top in each row would work as well. The diagram at the bottom of this page has a blue arrow pointing to each entry that indicates which value was used to compute it. (When there is a tie in computing the maximum, both options are shown.)

The final challenge is to recover the longest common subsequence itself, not just its length. The key idea is to retrace the steps of the dynamic programming algorithm *backward*, rediscovering the path of choices (highlighted in gray in the diagram) from `opt[0][0]` to `opt[m][n]`. To determine the choice that led to `opt[i][j]`, we consider the three possibilities:

- The character `s[i]` equals `t[j]`. In this case, we must have `opt[i][j]` = `opt[i+1][j+1]` + 1, and the next character in the LCS is `s[i]` (or `t[j]`), so we include the character `s[i]` (or `t[j]`) in the LCS and continue tracing back from `opt[i+1][j+1]`.
- The LCS does not contain `s[i]`. In this case, `opt[i][j]` = `opt[i+1][j]` and we continue tracing back from `opt[i+1][j]`.
- The LCS does not contain `t[j]`. In this case, `opt[i][j]` = `opt[i][j+1]` and we continue tracing back from `opt[i][j+1]`.

We begin tracing back at `opt[0][0]` and continue until we reach `opt[m][n]`. At each step in the traceback either `i` increases or `j` increases (or both), so the process terminates after at most `m` + `n` iterations of the `while` loop.



Longest common subsequence of GGCACCACG and ACGGCGGATACG

DYNAMIC PROGRAMMING IS A FUNDAMENTAL ALGORITHM design paradigm, intimately linked to recursion. If you take later courses in algorithms or operations research, you are sure to learn more about it. The idea of recursion is fundamental in computation, and the idea of avoiding recomputation of values that have been computed before is certainly a natural one. Not all problems immediately lend themselves to a recursive formulation, and not all recursive formulations admit an order of computation that easily avoids recomputation—arranging for both can seem a bit miraculous when one first encounters it, as you have just seen for the LCS problem.

Perspective Programmers who do not use recursion are missing two opportunities. First recursion leads to compact solutions to complex problems. Second, recursive solutions embody an argument that the program operates as anticipated. In the early days of computing, the overhead associated with recursive programs was prohibitive in some systems, and many people avoided recursion. In modern systems like Java, recursion is often the method of choice.

Recursive functions truly illustrate the power of a carefully articulated abstraction. While the concept of a function having the ability to call itself seems absurd to many people at first, the many examples that we have considered are certainly evidence that mastering recursion is essential to understanding and exploiting computation and in understanding the role of computational models in studying natural phenomena.

Recursion has reinforced for us the idea of proving that a program operates as intended. The natural connection between recursion and mathematical induction is essential. For everyday programming, our interest in correctness is to save time and energy tracking down bugs. In modern applications, security and privacy concerns make correctness an *essential* part of programming. If the programmer cannot be convinced that an application works as intended, how can a user who wants to keep personal data private and secure be so convinced?

Recursion is the last piece in a programming model that served to build much of the computational infrastructure that was developed as computers emerged to take a central role in daily life in the latter part of the 20th century. Programs built from libraries of functions consisting of statements that operate on primitive types of data, conditionals, loops, and function calls (including recursive ones) can solve important problems of all sorts. In the next section, we emphasize this point and review these concepts in the context of a large application. In CHAPTER 3 and in CHAPTER 4, we will examine extensions to these basic ideas that embrace the more expansive style of programming that now dominates the computing landscape.