1

# Dynamic Programming

Consider again the recursive function for computing the $n$th Fibonocci number.

```
int Fibr(int n) {
  if (n <= 1) return 1;            // Base case
  return Fibr(n-1) + Fibr(n-2);    // Recursive call
}
```

The cost of this algorithm (in terms of function calls) is the size of the $n$th Fibonocci number itself, which our analysis showed to be exponential (approximately $n^1.62$) Why is this so expensive? It is expensive primarily because two recursive calls are made by the function, and they are largely redundant. That is, each of the two calls is recomputing most of the series, as is each sub-call, and so on. Thus, the smaller values of the function are being recomputed a huge number of times. If we could eliminate this redundancy, the cost would be greatly reduced.

One way to accomplish this goal is to keep a table of values, and first check the table to see if the computation can be avoided. Here is a straightforward example of doing so.

```
int Fibrt(int n, int* Values) {
  // Assume Values has at least n slots, and all
  // slots are initialized to 0
  if (n <= 1) return 1;             // Base case
  if (Values[n] != 0) return Values[n];
  Values[n] = Fibrt(n-1, Values) + Fibrt(n-2, Values);
  return Values[n];
}
```

This version of the algorithm will not compute a value more than once, so its cost should be linear. Of course, we didn't actually need to use a table. Instead, we could build the value by working from 0 and 1 up to $n$ rather than backwards from $n$ down to 0 and 1. Going up from the bottom we only need to store the previous two values of the function, as is done by our iterative version.

```
long Fibi(int n) {
  long past, prev, curr;
  past = prev = curr = 1;     // curr holds Fib(i)
  for (int i=2; i<=n; i++) {  // Compute next value
    past = prev; prev = curr; // past holds Fib(i-2)
    curr = past + prev;       // prev holds Fib(i-1)
  }
  return curr;
}
```

However, this issue of recomputing subproblems comes up frequently. In many cases, arbitrary subproblems (or at least a wide variety of subproblems) might need to be recomputed, so that storing subresults in a fixed number of variables will not work. Thus, there are many times where storing a table of subresults can be useful.

This approach to designing an algorithm that works by storing a table of results for subproblems is called dynamic programming. The name is somewhat arcane, since it doesn't bear much obvious similarity to the process that is taking place of storing subproblems in a table. However, it comes originally from the field of dynamic control systems, which got its start before what we think of as computer programming. The act of storing precomputed values in a table for later reuse is referred to as "programming" in that field.

Dynamic programming is a powerful alternative to the standard principle of divide and conquer. In divide and conquer, a problem is split into subproblems, the subproblems are solved (independently), and the recombined into a solution for the problem being solved. Dynamic programming is appropriate whenever the subproblems to be solved are overlapping in some way. Whenever this happens, dynamic programming can be used if we can find a suitable way of doing the necessary bookkeeping. Dynamic programming algorithms are usually not implemented by simply using a table to store subproblems for recursive calls (i.e., going backwards as is done by `Fibrt`). Instead, such algorithms more typically implemented by building the table of subproblems from the bottom up. Thus, `Fibi` is actually closer in spirit to dynamic programming than is `Fibrt` even though it doesn't need the actual table.