can lead to an efficient algorithm. Start with a recursive algorithm or definition. Only once we have a correct recursive algorithm do we worry about speeding it up by using a results matrix.

Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent *left to right* order among components. Left-to-right objects includes: character strings, rooted trees, polygons, and integer sequences. Dynamic programming is best learned by carefully studying examples until things start to click. We present three war stories where dynamic programming played the decisive role to demonstrate its utility in practice.

8.1 Caching vs. Computation

Dynamic programming is essentially a tradeoff of space for time. Repeatedly recomputing a given quantity is harmless unless the time spent doing so becomes a drag on performance. Then we are better off storing the results of the initial computation and looking them up instead of recomputing them again.

The tradeoff between space and time exploited in dynamic programming is best illustrated when evaluating recurrence relations such as the Fibonacci numbers. We look at three different programs for computing them below.

8.1.1 Fibonacci Numbers by Recursion

The Fibonacci numbers were originally defined by the Italian mathematician Fibonacci in the thirteenth century to model the growth of rabbit populations. Rabbits breed, well, like rabbits. Fibonacci surmised that the number of pairs of rabbits born in a given year is equal to the number of pairs of rabbits born in each of the two previous years, starting from one pair of rabbits in the first year. To count the number of rabbits born in the nth year, he defined the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

with basis cases $F_0 = 0$ and $F_1 = 1$. Thus, $F_2 = 1$, $F_3 = 2$, and the series continues $\{3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots\}$. As it turns out, Fibonacci's formula didn't do a very good job of counting rabbits, but it does have a host of interesting properties.

Since they are defined by a recursive formula, it is easy to write a recursive program to compute the nth Fibonacci number. A recursive function algorithm written in C looks like this:

```
long fib_r(int n)
{
        if (n == 0) return(0);
        if (n == 1) return(1);

        return(fib_r(n-1) + fib_r(n-2));
}
```

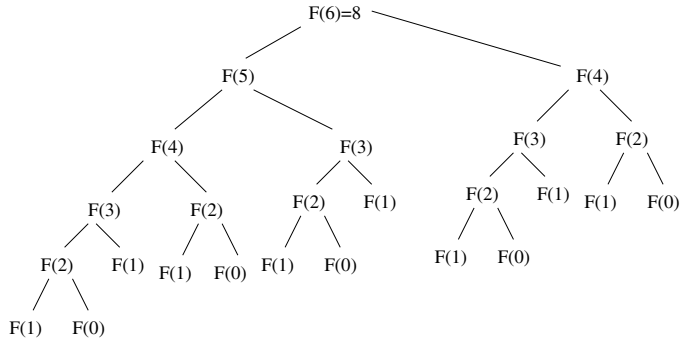

Figure 8.1: The computation tree for computing Fibonacci numbers recursively

The course of execution for this recursive algorithm is illustrated by its *recursion tree*, as illustrated in Figure 8.1. This tree is evaluated in a depth-first fashion, as are all recursive algorithms. I encourage you to trace this example by hand to refresh your knowledge of recursion.

Note that $F(4)$ is computed on both sides of the recursion tree, and $F(2)$ is computed no less than five times in this small example. The weight of all this redundancy becomes clear when you run the program. It took more than 7 minutes for my program to compute the first 45 Fibonacci numbers. You could probably do it faster by hand using the right algorithm.

How much time does this algorithm take to compute $F(n)$? Since $F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$, this means that $F_n > 1.6^n$. Since our recursion tree has only 0 and 1 as leaves, summing up to such a large number means we must have at least 1.6^n leaves or procedure calls! This humble little program takes exponential time to run!

8.1.2 Fibonacci Numbers by Caching

In fact, we can do much better. We can explicitly store (or *cache*) the results of each Fibonacci computation $F(k)$ in a table data structure indexed by the parameter k. The key to avoiding recomputation is to explicitly check for the value before trying to compute it:

```
#define MAXN    45       /* largest interesting n */
#define UNKNOWN   -1     /* contents denote an empty cell */
long f[MAXN+1];          /* array for caching computed fib values */
```

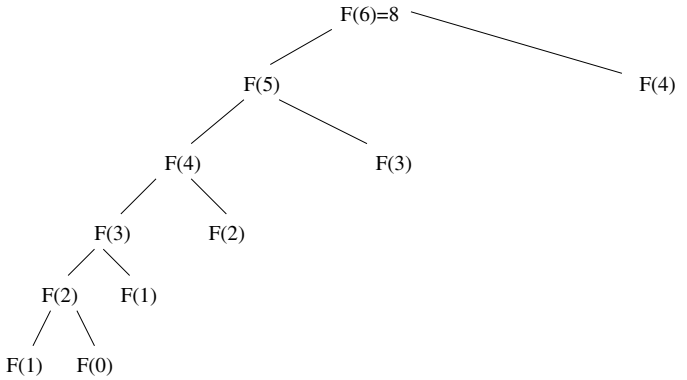

Figure 8.2: The Fibonacci computation tree when caching values

```
long fib_c(int n)
{
        if (f[n] == UNKNOWN)
                f[n] = fib_c(n-1) + fib_c(n-2);

        return(f[n]);
}


long fib_c_driver(int n)
{
        int i;                  /* counter */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = UNKNOWN;

        return(fib_c(n));
}
```

To compute $F(n)$, we call `fib_c_driver(n)`. This initializes our cache to the two values we initially know ($F(0)$ and $F(1)$) as well as the `UNKNOWN` flag for all the rest we don't. It then calls a look-before-crossing-the-street version of the recursive algorithm.

This cached version runs instantly up to the largest value that can fit in a long integer. The new recursion tree (Figure 8.2) explains why. There is no meaningful branching, because only the left-side calls do computation. The right-side calls find what they are looking for in the cache and immediately return.

What is the running time of this algorithm? The recursion tree provides more of a clue than the code. In fact, it computes $F(n)$ in linear time (in other words, $O(n)$ time) because the recursive function `fib_c(k)` is called exactly twice for each value $0 \leq k \leq n$.

This general method of explicitly caching results from recursive calls to avoid recomputation provides a simple way to get *most* of the benefits of full dynamic programming, so it is worth a more careful look. In principle, such caching can be employed on any recursive algorithm. However, storing partial results would have done absolutely no good for such recursive algorithms as *quicksort*, *backtracking*, and *depth-first search* because all the recursive calls made in these algorithms have distinct *parameter values*. It doesn't pay to store something you will never refer to again.

Caching makes sense only when the space of distinct parameter values is modest enough that we can afford the cost of storage. Since the argument to the recursive function `fib_c(k)` is an integer between 0 and n, there are only $O(n)$ values to cache. A linear amount of space for an exponential amount of time is an excellent tradeoff. But as we shall see, we can do even better by eliminating the recursion completely.

> *Take-Home Lesson:* Explicit caching of the results of recursive calls provides *most* of the benefits of dynamic programming, including usually the same running time as the more elegant full solution. If you prefer doing extra programming to more subtle thinking, you can stop here.

8.1.3 Fibonacci Numbers by Dynamic Programming

We can calculate F_n in linear time more easily by explicitly specifying the order of evaluation of the recurrence relation:

```
long fib_dp(int n)
{
        int i;              /* counter */
        long f[MAXN+1];     /* array to cache computed fib values */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = f[i-1]+f[i-2];

        return(f[n]);
}
```

We have removed all recursive calls! We evaluate the Fibonacci numbers from smallest to biggest and store all the results, so we know that we have F_{i-1} and F_{i-2} ready whenever we need to compute F_i. The linearity of this algorithm should

be apparent. Each of the n values is computed as the simple sum of two integers in total $O(n)$ time and space.

More careful study shows that we do not need to store all the intermediate values for the entire period of execution. Because the recurrence depends on two arguments, we only need to retain the last two values we have seen:

```
long fib_ultimate(int n)
{
        int i;                  /* counter */
        long back2=0, back1=1;  /* last two values of f[n] */
        long next;              /* placeholder for sum */

        if (n == 0) return (0);

        for (i=2; i<n; i++) {
                next = back1+back2;
                back2 = back1;
                back1 = next;
        }
        return(back1+back2);
}
```

This analysis reduces the storage demands to constant space with no asymptotic degradation in running time.

8.1.4 Binomial Coefficients

We now show how to compute the *binomial coefficients* as another illustration of how to eliminate recursion by specifying the order of evaluation. The binomial coefficients are the most important class of counting numbers, where $\binom{n}{k}$ counts the number of ways to choose k things out of n possibilities.

How do you compute the binomial coefficients? First, $\binom{n}{k} = n!/((n-k)!k!)$, so in principle you can compute them straight from factorials. However, this method has a serious drawback. Intermediate calculations can easily cause arithmetic overflow, even when the final coefficient fits comfortably within an integer.

A more stable way to compute binomial coefficients is using the recurrence relation implicit in the construction of Pascal's triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```