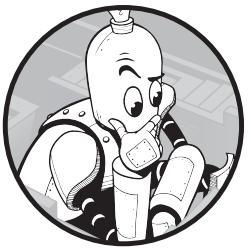
3

MEMOIZATION AND DYNAMIC PROGRAMMING



In this chapter, we'll study four problems that appear to be solvable using recursion. As you'll see, while in theory we can use recursion, in practice it leads to an explosion of work that renders the problems unsolvable. Not to worry: you'll learn two powerful, related techniques, called memoization and dynamic programming, that will lead to shocking performance increases, morphing runtimes from hours or days to seconds. These techniques aren't just for the four problems that I've selected for this chapter. Once you learn these techniques, you'll be able to solve hundreds of other programming problems. If you're going to read one chapter in this book, read this one.

Problem 1: Burger Fervor

This is UVa problem 10465.

The Problem

A man named Homer Simpson likes to eat and drink. He has t minutes that he'll spend eating burgers and drinking beer. There are two kinds of burgers. One of them takes m minutes to eat, and the other takes n minutes to eat.

Homer likes burgers more than beer, so he'd like to spend the entire t minutes eating burgers. However, doing so isn't always possible. For example, if $m = 4$, $n = 9$, and $t = 15$, then no combination of the 4-minute and 9-minute burgers can take him exactly 15 minutes to eat. If that's the case, he'll spend as much time as possible eating burgers and then fill the rest of the time drinking beer. Our task is to determine the number of burgers that Homer can eat.

Input

We read test cases until there is no more input. Each test case is represented by a line of three integers: m, the number of minutes it takes to eat the first kind of burger; n, the number of minutes it takes to eat the second kind of burger; and t, the number of minutes that Homer will spend eating burgers and drinking beer. Each m, n, and t value is less than 10,000.

Output

For each test case:

- If Homer can spend exactly t minutes eating burgers, then output the maximum number of burgers that he can eat.
- Otherwise, output the maximum number of burgers that Homer can eat when maximizing his time eating burgers, a space, and the number of remaining minutes (during which he'll drink beer).

The time limit for solving the test cases is three seconds.

Forming a Plan

Let's start by thinking about a few different test cases. Here's the first one:

```
4 9 22
```

Here, the first kind of burger takes 4 minutes to eat ($m = 4$), the second kind of burger takes 9 minutes to eat ($n = 9$), and Homer has 22 minutes to spend ($t = 22$). This is an example in which Homer can fill the entire time by eating burgers. The maximum number of burgers that Homer can eat here is three, so `3` is the correct output for this test case.

The three burgers that Homer should eat are one four-minute burger and two nine-minute burgers. This takes him $1 \times 4 + 2 \times 9 = 22$ minutes, as

required. Notice, though, that we are *not* being asked to indicate the number of each kind of burger he should eat. All we're asked to do is output the total number of burgers. When I provide the number of each kind of burger below, I do so only to offer evidence that the proposed output is feasible.

Here's another test case:

```
4 9 54
```

The correct output here is `11`, obtained by eating nine four-minute burgers and two nine-minute burgers. Unlike the `4 9 22` test case, here Homer has multiple ways to spend exactly 54 minutes eating burgers. For example, he could eat six nine-minute burgers—that fills up the 54 minutes, too—but, remember, if we can fill the entire `t` minutes, then we want to output the *maximum* number of burgers.

As noted in the problem description, it's not always possible for Homer to completely fill the *t* minutes by eating burgers. Let's study the example that I gave there as our next test case:

```
4 9 15
```

How many burgers should Homer eat here? He can eat a maximum of three burgers by eating three four-minute burgers. By doing so, Homer would spend 12 minutes eating burgers, and he would have to spend the remaining 15 − 12 = 3 minutes drinking beer. So, he eats three burgers and has three minutes' beer drinking time—have we solved this problem?

We have not! Carefully reread the problem description, and zone in on this: "output the maximum number of burgers that Homer can eat when maximizing his time eating burgers." That is, when Homer cannot fill the entire time by eating burgers, we want to maximize the *time* that he spends eating burgers and then the maximum number of burgers he can eat in that time. The correct output for `4 9 15` is therefore `2 2`: the first two means that he eats two burgers (one four-minute burger and one nine-minute burger, for a total of 13 minutes) and the second two means that he has to spend 2 minutes (15 − 13) drinking beer.

In the `4 9 22` and `4 9 54` test cases, we're being asked to solve the problem for 22 and 54 minutes, respectively. We'll find in these cases that there is indeed a way to spend the entire time eating burgers, and we can report that as our solution. However, in the `4 9 15` case, we'll find that there is no way to completely fill the 15 minutes by eating burgers. What do we do there?

One idea is that we can next try to fill exactly 14 minutes with the four-minute and nine-minute burgers. If we succeed, then we have our answer: we report the maximum number of burgers that Homer can eat in exactly 14 minutes, followed by one, the number of minutes Homer spends drinking beer. This would maximize the amount of time that Homer can spend eating burgers. We already know that eating burgers for exactly 15 minutes is impossible, so 14 minutes is the next best option.

Let's see if 14 minutes works. Can we fill exactly 14 minutes with the four-minute and nine-minute burgers? No! Like the 15-minute case, this is impossible.

We can, though, fill exactly 13 minutes by eating two burgers: one four-minute burger and one nine-minute burger. That leaves Homer two minutes for drinking beer. This justifies `2 2` as the correct output.

In summary, our plan is to determine whether Homer can eat burgers for exactly t minutes. If he can, then we're done: we report the maximum number of burgers he can eat. If he can't, then we determine whether Homer can eat burgers for exactly $t - 1$ minutes. If he can, then we're done, and we report the maximum number of burgers he can eat and the number of minutes spent drinking beer. If he can't, then we move on to trying $t - 2$ minutes, then $t - 3$ minutes, and so on, stopping when the time can be completely filled by eating burgers.

Characterizing Optimal Solutions

Consider the `4 9 22` test case. Whatever combination of burgers and beer we propose as the solution better take exactly 22 minutes, and it better actually be doable using the four-minute and nine-minute burgers. Such a solution, which adheres to the rules of a problem, is called a *feasible* solution. A solution attempt that does not follow the rules is called an *infeasible* solution. For example, having Homer spend 4 minutes eating burgers and 18 minutes drinking beer is feasible. Having Homer spend 8 minutes eating burgers and 18 minutes drinking beer is infeasible, because 8 + 18 is not 22. Having Homer spend 5 minutes eating burgers and 17 minutes drinking beer is also infeasible, because there's no way we can use the four-minute and nine-minute burgers to get a total of 5 minutes.

Burger Fervor is an *optimization problem*. An optimization problem involves choosing the *optimal* (best) solution out of all feasible solutions. There may be many feasible solutions of varying quality. Some will be really poor, such as drinking beer for 22 minutes. Others will be optimal. Still others will be close to but not quite optimal—maybe they're off by one or two. Our goal is to cut through the clutter and identify an optimal solution.

Suppose we're solving a case where the first kind of burger takes m minutes to eat, the second kind of burger takes n minutes to eat, and we have to spend exactly t minutes.

If $t = 0$, then the correct output is `0`, because we can fill the entire zero minutes by eating zero burgers. As we continue, we'll therefore focus on what to do when t is greater than zero.

Let's think about what an optimal solution for t minutes must look like. Of course, we can't possibly know anything specific, such as "Homer eats a four-minute burger, then a nine-minute burger, then another nine-minute burger, then . . ." We haven't done anything yet to solve the problem, so obtaining this level of detail is wishful thinking.

There is, however, something we can say that's not wishful thinking. It's at once so inane that you'd be forgiven for wondering why I am stating it at all and so powerful that at its core lies a solution strategy for a bewildering number of optimization problems.

Here it is. Suppose that Homer can fill exactly t minutes by eating burgers. The final burger that he eats, the one that finishes off his t minutes, must be an m-minute burger or an n-minute burger.

How could that final burger be anything else? Homer can only eat m-minute and n-minute burgers, so there are only two choices for the last burger that he eats and so two choices for what the end of the optimal solution must look like.

If we know that the final burger that Homer eats in an optimal solution is an m-minute burger, we know he has $t - m$ minutes left to spend. We must be able to fill those $t - m$ minutes with burgers, without drinking any beer: remember that we are assuming that Homer can spend the entire t minutes by eating burgers. If we could spend those $t - m$ minutes optimally, with Homer eating the maximum number of burgers, then we'd have an optimal solution to the original problem of t minutes. We'd take the number of burgers that he can eat in $t - m$ minutes and add one m-minute burger to fill the remaining m minutes.

Now, what if we knew that the final burger that Homer eats in an optimal solution is an n-minute burger? Then he has $t - n$ minutes left to spend. Again, by virtue of the entire t minutes being spent eating burgers, we know that it must be possible for Homer to eat burgers for the first $t - n$ of those minutes. If we could spend those $t - n$ minutes optimally, then we'd have an optimal solution to the original problem of t minutes. We'd take the number of burgers that he can eat in $t - n$ minutes and add one n-minute burger to fill the remaining n minutes.

Now we seem to be squarely in farce territory. We just assumed that we knew what the final burger was! However, there's no way we could know this. We do know that the final burger is an m-minute burger *or* an n-minute burger. We definitely don't know which it is.

The wonderful truth is that we don't need to know. We can assume that the final burger is an m-minute burger and solve the problem optimally given that choice. We then make the other choice—assume that the final burger is an n-minute burger—and solve the problem optimally given that choice. In the first case, we have a subproblem of $t - m$ minutes to solve optimally; in the second case, we have a subproblem of $t - n$ minutes to solve optimally. Whenever we have characterized a solution to a problem in terms of solutions to subproblems, we would do well to try a recursive approach as we did in Chapter 2.

Solution 1: Recursion

Let's attempt a recursive solution. We'll begin by writing a helper function to solve for exactly t minutes. Once we're done with that, we'll write a function that solves for t minutes, $t - 1$ minutes, $t - 2$ minutes, and so on, until we can completely fill some number of minutes with burgers.

The Helper Function: Solving for the Number of Minutes

Each problem and subproblem instance is characterized by three parameters: m, n, and t. We'll therefore write the body of the following function:

```
int solve_t(int m, int n, int t)
```

If Homer can spend exactly `t` minutes eating burgers, then we'll return the maximum number of burgers he can eat. If he can't spend exactly `t` minutes eating burgers—meaning he must spend at least one minute drinking beer—then we'll return `-1`. A return value of `0` or more means that we've solved the problem using burgers alone; a return value of `-1` means that the problem cannot be solved using burgers alone.

If we call `solve_t(4, 9, 22)`, we expect to get `3` as the return value: three is the maximum number of burgers that Homer can eat in exactly `22` minutes. If we call `solve_t(4, 9, 15)`, we expect to get `-1` as the return value: no combination of four-minute and nine-minute burgers gives us exactly 15 minutes.

We've already settled on what to do when $t = 0$: in this case, we have zero minutes to spend, and we do so by having Homer eat zero burgers:

```
if (t == 0)
  return 0;
```

That's the base case of our recursion. To implement the rest of this function, we need the analysis from the last section. Remember that, to solve the problem for t minutes, we think about the final burger that Homer eats. Maybe it's an m-minute burger. To check that possibility, we solve the subproblem for $t - m$ minutes. Of course, the final burger can only be an m-minute burger if we've got at least m minutes to spend. This logic can be coded as follows:

```
int first;
if (t >= m)
  first = solve_t(m, n, t - m);
else
  first = -1;
```

We use `first` to store the optimal solution to the `t - m` subproblem, with `-1` indicating "no solution." If `t >= m`, then there's a chance that an m-minute burger is the final one, so we make a recursive call to compute the optimal number of burgers that Homer can eat in exactly `t - m` minutes. That recursive call will return a number greater than `-1` if it can be solved exactly or `-1` if it can't. If `t < m`, then there's no recursive call to make: we set `first = -1` to signify that an m-minute burger isn't the final burger and that it can't participate in an optimal solution for `t` minutes.

Now, what about when an *n*-minute burger is the final burger? The code for this case is analogous to the *m*-minute burger case, this time using the variable `second` instead of `first`:

```
int second;
if (t >= n)
  second = solve_t(m, n, t - n);
else
  second = -1;
```

Let's summarize our current progress:

- The variable `first` is the solution to the `t - m` subproblem. If it's `-1`, then we can't fill `t - m` minutes with burgers. If it's anything else, then it gives the optimal number of burgers that Homer can eat in exactly `t - m` minutes.
- The variable `second` is the solution to the `t - n` subproblem. If it's `-1`, then we can't fill `t - n` minutes with burgers. If it's anything else, then it gives the optimal number of burgers that Homer can eat in exactly `t - n` minutes.

There's a chance that both `first` and `second` have values of `-1`. A value of `-1` for `first` means that an *m*-minute burger can't be the final burger. A value of `-1` for `second` means that an *n*-minute burger can't be the final burger. If the final burger can't be an *m*-minute burger and can't be an *n*-minute burger, then we're out of options and have to conclude that there's no way to solve the problem for `t` minutes:

```
if (first == -1 && second == -1)
  return -1;
```

Otherwise, if `first` or `second` or both are greater than `-1`, then we have at least one solution for `t` minutes. In this case, we take the maximum of `first` and `second` to choose the better subproblem solution. If we add one to that maximum, thereby incorporating the final burger, then we obtain the maximum for the original problem of `t` minutes:

```
return max(first, second) + 1;
```

The full function is given in Listing 3-1.

```
int max(int v1, int v2) {
  if (v1 > v2)
    return v1;
  else
    return v2;
}

int solve_t(int m, int n, int t) {
  int first, second;
```

```
  if (t == 0)
    return 0;
  if (t >= m)
❶ first = solve_t(m, n, t - m);
  else
    first = -1;
  if (t >= n)
❷ second = solve_t(m, n, t - n);
  else
    second = -1;
  if (first == -1 && second == -1)
❸ return -1;
  else
❹ return max(first, second) + 1;
}
```

Listing 3-1: Solving for t minutes

Whether or not I've convinced you of this function's correctness, it's worth spending a few minutes getting a feel for what the function does in practice.

Let's begin with `solve_t(4, 9, 22)`. The recursive call for `first` ❶ solves the subproblem for 18 minutes (22−4). That recursive call returns `2`, because two is the maximum number of burgers that Homer can eat in exactly 18 minutes. The recursive call for `second` ❷ solves the subproblem for 13 minutes (22 − 9). That recursive call returns `2` as well, because two is the maximum number of burgers that Homer can eat in 13 minutes. That is, both `first` and `second` are `2` in this case; tacking on the final four-minute or nine-minute burger gives a solution of `3` ❹ for the original problem of 22 minutes.

Let's now try `solve_t(4, 9, 20)`. The recursive call for `first` ❶ solves the subproblem for 16 minutes (20 − 4) and yields `4` as a result, but what about the recursive call for `second` ❷? Well, that one is asked to solve the subproblem for 11 minutes (20 − 9), but there is no way to spend exactly 11 minutes by eating four-minute and nine-minute burgers! Thus this second recursive call returns `-1`. The maximum of `first` and `second` is therefore `4` (the value of `first`), and so we return `5` ❹.

So far we've seen an example where the two recursive calls both give subproblem solutions with the same number of burgers and an example where only one recursive call gives a subproblem solution. Now let's look at a case where each recursive call returns a subproblem solution—but where one is better than the other! Consider `solve_t(4, 9, 36)`. The recursive call for `first` ❶ yields `8`, the maximum number of burgers that Homer can eat in exactly 32 minutes (36 − 4). The recursive call for `second` ❷ yields `3`, the maximum number of burgers that Homer can eat in exactly 27 minutes (36 − 9). The maximum of `8` and `3` is `8`, and so we return `9` as the overall solution ❹.

Finally, try `solve_t(4, 9, 15)`. The recursive call for `first` ❶ is asked to solve for 11 minutes (15 − 4) and, since this is impossible with these kinds of burger, returns `-1`. The result is similar to the recursive call for `second` ❷:

solving for 6 minutes (15 − 9) is impossible, so it also returns -1. There is therefore no way to solve for exactly 15 minutes; hence, we return -1 ❸.

The solve and main Functions

Recall from "Forming a Plan" on page 72 that if we can fill exactly *t* minutes by eating burgers, then we output the maximum number of burgers. Otherwise, Homer has to spend at least one minute drinking beer. To figure out the number of minutes that he must spend drinking beer, we try to solve for $t-1$ minutes, $t-2$ minutes, and so on, until we find a number of minutes that can be filled by eating burgers. Happily, with our `solve_t` function, we can set the `t` parameter to whatever we want. We can start at the given value of `t` and make calls on `t - 1`, `t - 2`, and so on. We effect this plan in Listing 3-2.

```
void solve(int m, int n, int t) {
  int result, i;
❶ result = solve_t(m, n, t);
   if (result >= 0)
    ❷ printf("%d\n", result);
  else {
     i = t - 1;
  ❸ result = solve_t(m, n, i);
     while (result == -1) {
       i--;
    ❹ result = solve_t(m, n, i);
     }
  ❺ printf("%d %d\n", result, t - i);
  }
}
```

Listing 3-2: Solution 1

First, we solve the problem for exactly `t` minutes ❶. If we get a result that's at least zero, then we output the maximum number of burgers ❷ and stop.

If it wasn't possible for Homer to eat burgers for the entire `t` minutes, we set `i` to `t - 1`, since `t - 1` is the next-best number of minutes that we should try. We then solve the problem for this new value of `i` ❸. If we don't get a value of -1, we're successful and the while loop is skipped. If we're not successful, the while loop executes until we successfully solve a subproblem. Inside the while loop, we decrement the value of `i` and solve that smaller subproblem ❹. The while loop will eventually terminate; for example, we can certainly fill zero minutes with burgers. Once we escape the while loop, we've found the largest number of minutes, `i`, that can be filled by burgers. At that point, `result` will hold the maximum number of burgers, and `t - i` is the number of minutes that remain, so we output both values ❺.

That's that. We use recursion in `solve_t` to solve for `t` exactly. We tested `solve_t` on different kinds of test cases, and everything looked good. Not being able to solve for `t` exactly poses no problem: we use a loop inside of `solve` to try the minutes one by one, from largest to smallest. All we need now is a

little `main` function to read the input and call `solve`; Listing 3-3 provides the code.

```
int main(void) {
  int m, n, t;
  while (scanf("%d%d%d", &m, &n, &t) != -1)
    solve(m, n, t);
  return 0;
}
```

Listing 3-3: The main function

Ah, a harmonious moment. We're now ready to submit Solution 1 to the judge. Please do that now. I'll wait . . . and wait . . . and wait.

Solution 2: Memoization

Solution 1 fails, not because it's incorrect, but because it's too slow. If you submit Solution 1 to the judge, you'll receive a "Time-Limit Exceeded" error. This reminds us of the "Time-Limit Exceeded" error we received in Solution 1 of the Unique Snowflakes problem. There, the inefficiency was emblematic of doing unnecessary work. Here, as we'll soon see, the inefficiency does not lie in doing unnecessary work but in doing necessary work over and over and over.

The problem description says that t can be any number of minutes less than 10,000. Surely, then, the following test case should pose no problem at all:

```
4 2 88
```

The m and n values, `4` and `2`, are very small. Relative to 10,000, the t value of `88` is very small as well. You may be surprised and disappointed that our code on this test case may not run within the three-second problem time limit. On my laptop, it takes about 10 seconds. That's 10 seconds on a puny `88` test case. While we're at it, let's try this slightly bigger test case:

```
4 2 90
```

All we did was increase `t` from `88` to `90`, but this small increase has a disproportionate effect on runtime: on my laptop, this test case takes about 18 seconds—almost double what the `88` test case takes! Testing with a `t` value of 92 just about doubles the runtime again, and so on and so on. No matter how fast the computer, you're unlikely to ever make it to a `t` value of even 100. By extrapolating from this trend, it's unfathomable how much time it would take to run our code on a test case where `t` is in the thousands. This kind of algorithm, in which a fixed increment in problem size leads to a doubling of runtime, is called an *exponential-time algorithm*.

We've established that our code is slow—but why? Where's the inefficiency?

Consider a given *m*, *n*, *t* test case. Our `solve_t` function has three parameters, but only the third parameter *t* ever changes. There are therefore only *t* + 1 different ways that `solve_t` can be called. For example, if *t* in a test case is `4`, then the only calls that can be made to `solve_t` are those with *t* values of `4`, `3`, `2`, `1`, and `0`. Once we call `solve_t` with some *t* value, such as `2`, there's no reason to ever make that same call again: we already have our answer, so there's no point kicking off a recursive call to compute that answer again.

Counting the Function Calls

I'm going to take Solution 1 and add some code that counts the number of times that `solve_t` is called; see Listing 3-4 for the new `solve_t` and `solve` functions. I added a global variable `total_calls` that is initialized to `0` on entry to `solve` and is increased by `1` on every call of `solve_t`. That variable is of type `long long`; `long` or `int` simply isn't big enough to capture the explosion of function calls.

```
unsigned long long total_calls;

int solve_t(int m, int n, int t) {
  int first, second;
❶ total_calls++;
  if (t == 0)
    return 0;
  if (t >= m)
    first = solve_t(m, n, t - m);
  else
    first = -1;
  if (t >= n)
    second = solve_t(m, n, t - n);
  else
    second = -1;
  if (first == -1 && second == -1)
    return -1;
  else
    return max(first, second) + 1;
}

void solve(int m, int n, int t) {
  int result, i;
❷ total_calls = 0;
  result = solve_t(m, n, t);
  if (result >= 0)
    printf("%d\n", result);
  else {
    i = t - 1;
    result = solve_t(m, n, i);
    while (result == -1) {
      i--;
```

```
      result = solve_t(m, n, i);
    }
    printf("%d %d\n", result, t - i);
  }
❸ printf("Total calls to solve_t: %llu\n", total_calls);
}
```

Listing 3-4: Solution 1, instrumented

At the start of `solve_t`, we increase `total_calls` by `1` ❶ to count this function call. In `solve`, we initialize `total_calls` to `0` ❷ so that the count of calls is reset before each test case is processed. For each test case, the code prints the number of times that `solve_t` was called ❸.

If we give it a go with this input:

```
4 2 88
4 2 90
```

we get this as output:

```
44
Total calls to solve_t: 2971215072
45
Total calls to solve_t: 4807526975
```

We've made billions of frivolous calls, when only about 88 or 90 of them can be distinct. We conclude that the same subproblems are being solved a staggering number of times.

Remembering Our Answers

Here's some intuition for the staggering number of calls we make. Suppose we call `solve_t(4, 2, 88)`. It makes two recursive calls: one to `solve_t(4, 2, 86)` and the other to `solve_t(4, 2, 84)`. So far, so good. Now consider what will happen for the `solve_t(4, 2, 86)` call. It will make two recursive calls of its own, the first of which is `solve_t(4, 2, 84)`—exactly one of the recursive calls made by `solve_t(4, 2, 88)`! That `solve_t(4, 2, 84)` work will therefore be performed twice. Once would have been enough!

However, the imprudent duplication is only just beginning. Consider the two `solve_t(4, 2, 84)` calls. By reasoning as in the previous paragraph, each of these calls will eventually lead to two calls of `solve_t(4, 2, 80)`, for a total of four. Again, once would have been enough!

Well, it would have been enough if we had somehow remembered the answer from the first time we computed it. If we remember the answer to a call of `solve_t` the first time we compute it, we can just look it up later when we need that answer again.

Remember, don't refigure. That's the maxim of a technique called *memoization*. Memoization comes from the word *memoize*, which means to store as if on a memo. It is a clunky word, sure, but one that's in widespread use.

Using memoization involves two steps:

1. Declare an array large enough to hold the solutions to all possible subproblems. In Burger Fervor, `t` is less than 10,000, so an array of 10,000 elements suffices. This array is typically given the name `memo`. Initialize the elements of `memo` to a value reserved to mean "unknown value."
2. At the start of the recursive function, add code to check whether the subproblem solution has already been solved. This involves checking the corresponding index of `memo`: if the "unknown value" is there, then we have to solve this subproblem now; otherwise, the answer is already stored in `memo`, and we simply return it, without doing any further recursion. Whenever we solve a new subproblem, we store its solution in `memo`.

Let's augment Solution 1 with memoization.

Implementing Memoization

The appropriate place to declare and initialize the `memo` array is in `solve`, since that's the function that first gets triggered for each test case. We'll use a value of `-2` to represent an unknown value: we can't use positive numbers because those would be confused with numbers of burgers, and we can't use `-1` because we're already using `-1` to mean "no solution possible." The updated `solve` function is given in Listing 3-5.

```
#define SIZE 10000

void solve(int m, int n, int t) {
  int result, i;
❶ int memo[SIZE];
  for (i = 0; i <= t; i++)
    memo[i] = -2;
  result = solve_t(m, n, t, memo);
  if (result >= 0)
    printf("%d\n", result);
  else {
    i = t - 1;
    result = solve_t(m, n, i, memo);
    while (result == -1) {
      i--;
      result = solve_t(m, n, i, memo);
    }
    printf("%d %d\n", result, t - i);
  }
}
```

Listing 3-5: Solution 2, with memoization implemented

We declare the `memo` array using the maximum possible size for any test case ❶. Then we loop from `0` to `t` and set each element in the range to `-2`.

There's also a small but important change in our calls to `solve_t`. Now we're passing in `memo`; in this way, `solve_t` can check `memo` to determine whether the current subproblem has already been solved and update `memo` if it has not.

The updated `solve_t` code is given in Listing 3-6.

```
int solve_t(int m, int n, int t, int memo[]) {
  int first, second;
❶ if (memo[t] != -2)
    return memo[t];
  if (t == 0) {
    memo[t] = 0;
    return memo[t];
  }
  if (t >= m)
    first = solve_t(m, n, t - m, memo);
  else
    first = -1;
  if (t >= n)
    second = solve_t(m, n, t - n, memo);
  else
    second = -1;
  if (first == -1 && second == -1) {
    memo[t] = -1;
    return memo[t];
  } else {
    memo[t] = max(first, second) + 1;
    return memo[t];
  }
}
```

Listing 3-6: Solving for `t` minutes, with memoization implemented

The game plan is the same as it was in Solution 1, Listing 3-1: if `t` is `0`, solve the base case; otherwise, solve for `t - m` minutes and `t - n` minutes and use the better one.

To this structure we fasten memoization. The huge reduction in time is realized when we check whether a solution for `t` is already in the `memo` array ❶, returning that stored result if it is. There is no fussing over whether the final burger takes m or n minutes. There is no recursion. All we have is an immediate return from the function.

If we don't find a solution in `memo`, then we have work to do. The work is the same as before—except that, whenever we're about to return the solution, we first store it in the memo. Before each of our `return` statements, we store the value we're about to return in `memo` so that our program maintains a memory of it.

Testing Our Memoization

I demonstrated that Solution 1 was doomed by showing you two things: that small test cases took far too long to run and that the slowness was caused by making an exorbitant number of function calls. How does Solution 2 fare in terms of these metrics?

Try Solution 2 with the input that bested Solution 1:

```
4 2 88
4 2 90
```

On my laptop, the time taken is imperceptibly small.

How many function calls are made? I encourage you to instrument Solution 2 in the way that I did for Solution 1 (Listing 3-4). If you do that and run it with the above input, you should get this output:

```
44
Total calls to solve_t: 88
45
Total calls to solve_t: 90
```

88 calls when `t` is 88. 90 calls when `t` is 90. The difference between Solution 2 and Solution 1 is like night and a few billion days. We've gone from an exponential-time algorithm to a linear-time algorithm. Specifically, we now have an $O(t)$ algorithm, where t is the number of minutes for the test case.

It's judge time. If you submit Solution 2, you'll see that we pass all of the test cases.

This is certainly a milestone, but it is not the last word on Homer and his burgers.

Solution 3: Dynamic Programming

We'll bridge our way from memoization to dynamic programming by making explicit the purpose of recursion in Solution 2. Consider the `solve_t` code in Listing 3-7; it's the same as the code in Listing 3-6 except that I'm now highlighting just the two recursive calls.

```
int solve_t(int m, int n, int t, int memo[]) {
  int first, second;
  if (memo[t] != -2)
    return memo[t];
  if (t == 0) {
    memo[t] = 0;
    return memo[t];
  }
  if (t >= m)
❶ first = solve_t(m, n, t - m, memo);
  else
    first = -1;
  if (t >= n)
```

```
  ❷ second = solve_t(m, n, t - n, memo);
   else
     second = -1;
   if (first == -1 && second == -1) {
     memo[t] = -1;
     return memo[t];
   } else {
     memo[t] = max(first, second) + 1;
     return memo[t];
   }
}
```

Listing 3-7: Solving for t minutes, focusing on recursive calls

At the first recursive call ❶, one of two very different things will happen. The first is that the recursive call finds its subproblem solution in the memo and returns immediately. The second is that the recursive call does not find the subproblem solution in the memo, in which case it carries out its own recursive calls. All of this is true of the second recursive call ❷ as well.

When we make a recursive call, and the recursive call finds its subproblem solution in the memo, we have to wonder why we made the recursive call at all. The only thing that the recursive call will do is check the memo and return; we could have done that ourselves. If the subproblem solution is not in the memo, however, then the recursion is really necessary.

Suppose that we could orchestrate things so that the `memo` array always holds the next subproblem solution that we need to look up. We want to know the optimal solution when `t` is `5`. It's in `memo`. What about when `t` is `18`? That's in `memo`, too. By virtue of always having the subproblem solutions in the memo, we'll never require a recursive call; we can just look up the solution right away.

Here we have the difference between memoization and dynamic programming. A function that uses memoization makes a recursive call to solve a subproblem. Maybe the subproblem was already solved, maybe it wasn't—regardless, it will be solved when the recursive call returns. A function that uses *dynamic programming* organizes the work so that a subproblem is already solved by the time we need it.

We then have no reason to use recursion: we just look up the solution. Memoization uses recursion to ensure that a subproblem is solved; dynamic programming ensures that the problem to be solved has no use for recursion.

Our dynamic-programming solution dispenses with the `solve_t` function and systematically solves for all values of `t` in `solve`. The code is given in Listing 3-8.

```
void solve(int m, int n, int t) {
  int result, i, first, second;
  int dp[SIZE];
❶ dp[0] = 0;
  for (i = 1; i <= t; i++) {
```

```
    ❷ if (i >= m)
        ❸ first = dp[i - m];
      else
        first = -1;
    ❹ if (i >= n)
        second = dp[i - n];
      else
        second = -1;
      if (first == -1 && second == -1)
      ❺ dp[i] = -1;
      else
      ❻ dp[i] = max(first, second) + 1;
    }

  ❼ result = dp[t];
    if (result >= 0)
      printf("%d\n", result);
    else {
      i = t - 1;
      result = dp[i];
      while (result == -1) {
        i--;
      ❽ result = dp[i];
      }
      printf("%d %d\n", result, t - i);
    }
  }
```

Listing 3-8: Solution 3, with dynamic programming

The canonical name for a dynamic-programming array is `dp`. We could have called it `memo`, since it serves the same purpose as a memo table, but we call it `dp` to follow convention. Once we declare the array, we solve the base case, explicitly storing the fact that the optimal solution for zero minutes is to eat zero burgers ❶. Then we have the loop that controls the order in which the subproblems are solved. Here, we solve the subproblems from smallest number of minutes (`1`) to largest number of minutes (`t`). The variable `i` determines which subproblem is being solved. Inside our loop, we have the familiar check of whether it makes sense to test the `m`-minute burger as the final burger ❷. If so, we look up the solution to the `i - m` subproblem in the `dp` array ❸.

Notice how we just look up the value from the array ❸, without using any recursion. We can do that because we know, by virtue of the fact that `i - m` is less than `i`, that we've already solved subproblem `i - m`. This is precisely why we solve subproblems in order, from smallest to largest: larger subproblems will require solutions to smaller subproblems, so we must ensure that those smaller subproblems have already been solved.

The next if statement ❹ is analogous to the previous one ❷ and handles the case when the final burger is an `n`-minute burger. As before, we look up the solution to a subproblem using the `dp` array. We know for sure that the `i - n` subproblem has already been solved, because the `i - n` iteration took place before this `i` iteration.

We now have the solutions to both of the required subproblems. All that's left to do is store the optimal solution for `i` in `dp[i]` ❺ ❻.

Once we've built up the `dp` array, solving subproblems `0` to `t`, we can look up subproblem solutions at will. We thus simply look up the solution to subproblem `t` ❼, printing it if there's a solution and looking up solutions to progressively smaller subproblems if there's not ❽.

Let's present one example `dp` array before moving on. For the following test case:

```
4 9 15
```

the final contents of the `dp` array are

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	0	–1	–1	–1	1	–1	–1	–1	2	1	–1	–1	3	2	–1	–1

We can trace the code in Listing 3-8 to confirm each of these subproblem solutions. For example, `dp[0]`, the maximum number of burgers that Homer can eat in zero minutes, is `0` ❶. `dp[1]` is `-1` because both tests ❷ ❹ fail, meaning we store `-1` ❺.

As a final example, let's reverse-engineer how `dp[12]` got its value of `3`. Since 12 is greater than 4, the first test passes ❷. We then set `first` to `dp[8]` ❸, which has a value of `2`. Similarly, 12 is greater than 9, so the second test passes ❹, and we set `second` to `dp[3]`, which has a value of `-1`. The maximum of `first` and `second` is therefore `2`, so we set `dp[12]` to `3`, one more than that maximum ❻.

Memoization and Dynamic Programming

We solved Burger Fervor in four steps. First, we characterized what an optimal solution must look like; second, we wrote a recursive solution; third, we added memoization; fourth, we eliminated the recursion by explicitly solving subproblems from smallest to largest. These four steps offer a general plan for tackling many other optimization problems.

Step 1: Structure of Optimal Solutions

The first step is to show how to decompose an optimal solution to a problem into optimal solutions for smaller subproblems. In Burger Fervor, we did this by reasoning about the final burger that Homer eats. Is it an m-minute burger? That leaves the subproblem of filling $t - m$ minutes. What if it is an n-minute burger? That leaves the problem of filling $t - n$ minutes. We don't know which it is, of course, but we can simply solve these two subproblems to find out.

Often left implicit in these kinds of discussions is the requirement that an optimal solution to a problem contains within it not just some solution to the subproblems but *optimal* solutions to those subproblems. Let's make this point explicit here.

In Burger Fervor, when supposing that the final burger in an optimal solution is an m-minute burger, we argued that a solution to the $t - m$ subproblem was part of the solution to the overall t problem. Moreover, an optimal solution for t must include the optimal solution for $t - m$: if it didn't, then the solution for t wouldn't be optimal after all, since we could improve it by using the better solution for $t - m$! A similar argument can be used to show that, if the last burger in an optimal solution is an n-minute burger, then the remaining $t - n$ minutes should be filled with an optimal solution for $t - n$.

Let me unpack this a little through an example. Suppose that $m = 4$, $n = 9$, and $t = 54$. The value of an optimal solution is 11. There is an optimal solution S where the final burger is a nine-minute burger. My claim is that S must consist of this nine-minute burger along with an optimal solution for 45 minutes. The optimal solution for 45 minutes is 10 burgers. If S used some suboptimal solution for the first 45 minutes, then S wouldn't be an example of an optimal 11-burger solution. For example, if S used a suboptimal five-burger solution for the first 45 minutes, then it would use a total of only six burgers!

If an optimal solution to a problem is composed of optimal solutions to subproblems, we say that the problem has *optimal substructure*. If a problem has optimal substructure, the techniques from this chapter are likely to apply.

I've read and heard people claim that solving optimization problems using memoization or dynamic programming is formulaic, that once you've seen one such problem, you've seen them all, and can just turn the crank when a new problem arises. I don't think so. That perspective belies the challenges of both characterizing the structure of optimal solutions and identifying that this will be fruitful in the first place. We'll make headway with these challenges in this chapter by solving several additional problems using memoization and dynamic programming. The sheer breadth of problems that can be solved using these approaches suggests to me that practicing with and generalizing from as many problems as possible is the only way forward.

Step 2: Recursive Solution

Step 1 not only suggests to us that memoization and dynamic programming will lead to a solution but also leaves in its wake a recursive approach for solving the problem. To solve the original problem, try each of the possibilities for an optimal solution, solving subproblems optimally using recursion. In Burger Fervor, we argued that an optimal solution for t minutes might consist of an m-minute burger and an optimal solution for $t - m$ minutes or an n-minute burger and an optimal solution to $t - n$ minutes. Solving the $t - m$ and $t - n$ subproblems is therefore required and, as these are smaller subproblems than t, we used recursion to solve them. In general, the number of

recursive calls depends on the number of available candidates competing to be the optimal solution.

Step 3: Memoization

If we succeed with Step 2, then we have a correct solution to the problem. As we saw with Burger Fervor, though, such a solution may require an absolutely unreasonable amount of time to execute. The culprit is that the same subproblems are being solved over and over, as a result of a phenomenon known as *overlapping subproblems*. Really, if we didn't have overlapping subproblems, then we could stop right here: recursion would be fine on its own. Think back to Chapter 2 and the two problems we solved there. We successfully solved those with recursion alone, and that worked because each subproblem was solved only once. In Halloween Haul, for example, we calculated the total amount of candy in a tree. The two subproblems were finding the total amounts of candy in the left and right subtrees. Those problems are independent: there's no way that solving the subproblem for the left subtree could somehow require information about the right subtree, or vice versa.

If there's no subproblem overlap, we can just use recursion. When there is subproblem overlap, it's time for memoization. As we saw in Burger Fervor, memoization means that we store the solution to a subproblem the first time we solve it. Then, whenever that subproblem solution is needed in the future, we simply look it up rather than recalculate it. Yes, the subproblems still overlap, but now they are solved only once, just like in Chapter 2.

Step 4: Dynamic Programming

Very likely, the solution resulting from Step 3 will be fast enough. Such a solution still uses recursion, but without the risk of duplicating work. As I'll explain in the next paragraph, sometimes we want to eliminate the recursion. We can do so as long as we systematically solve smaller subproblems before larger subproblems. This is dynamic programming: the use of a loop in lieu of recursion, explicitly solving all subproblems in order from smallest to largest.

So what's better: memoization or dynamic programming? For many problems, they are roughly equivalent and, in those cases, you should use what you find more comfortable. My personal choice is memoization. We'll see an example (Problem 3) where the `memo` and `dp` tables have multiple dimensions. In such problems, I often have trouble getting all of the base cases and bounds for the `dp` table correct.

Memoization solves subproblems on an as-needed basis. For example, consider the Burger Fervor test case where we have a kind of burger that takes two minutes to eat, a kind of burger that takes four minutes to eat, and 90 minutes of time. A memoized solution will never solve for odd numbers of minutes, such as 89 or 87 or 85, because those subproblems do not result from subtracting multiples of two and four from 90. Dynamic programming, by contrast, solves all subproblems on its way up to 90. The difference here

seems to favor memoized solutions; indeed, if huge swaths of the subproblem space are never used, then memoization may be faster than dynamic programming. This has to be balanced against the overhead inherent in recursive code though, with all of the calling and returning from functions. If you're so inclined, it wouldn't hurt to code up both solutions to a problem and see which is faster!

You'll commonly see people refer to memoized solutions as *top-down* solutions and dynamic-programming solutions as *bottom-up* solutions. It's called "top-down" because, to solve large subproblems, we recurse down to small subproblems. In "bottom-up" solutions, we start from the bottom—the smallest subproblems—and work our way to the top.

Memoization and dynamic programming are captivating to me. They can solve so many types of problems; I don't know another algorithm design technique that even comes close. Many of the tools that we learn in this book, such as hash tables in Chapter 1, offer valuable speedups. The truth is that even without those tools we could solve many problem instances—not in time to have such solutions accepted by the judge but perhaps still in time to be practically useful. However, memoization and dynamic programming are different. They vivify recursive ideas, turning algorithms that are astonishingly slow into those that are astonishingly fast. I hope I can pull you into the fold with the rest of this chapter and that you won't stop when the chapter does.

Problem 2: Moneygrubbers

In Burger Fervor, we were able to solve each problem by considering only two subproblems. Here, in Problem 2, we'll see that each subproblem requires more work.

This is UVa problem 10980.

The Problem

You want to buy apples, so you go to an apple store. The store has a price for buying one apple—for example, \$1.75. The store also has m pricing schemes, where each pricing scheme gives a number n and a price p for buying n apples. For example, one pricing scheme might state that three apples cost a total of \$4.00; another might state that two apples cost a total of \$2.50. You want to buy *at least* k apples and to do so as cheaply as possible.

Input

We read test cases until there's no more input. Each test case consists of the following lines:

- A line containing the price for buying one apple, followed by the number m of pricing schemes for this test case. m is at most 20.
- m lines, each of which gives a number n and total price p for buying n apples. n is between 1 and 100.

- A line containing integers, where each integer *k* is between 0 and 100 and gives the desired number of apples to buy.

Each price in the input is a floating-point number with exactly two decimal digits.

In the problem description, I gave the price of one apple as $1.75. I also gave two pricing schemes: three apples for $4.00 and two apples for $2.50. Suppose we wanted to determine the minimum price for buying at least one apple and at least four apples, respectively. Here's the input for this test case:

```
1.75 2
3 4.00
2 2.50
1 4
```

Output

For each test case, output the following:

- A line containing `Case c:`, where *c* is the number of the test case starting at 1.
- For each integer *k*, a line containing `Buy k for $d`, where *d* is the cheapest way that we can buy at least *k* apples.

Here's the output for the above sample input:

```
Case 1:
Buy 1 for $1.75
Buy 4 for $5.00
```

The time limit for solving the test cases is three seconds.

Characterizing Optimal Solutions

The problem description specifies that we want to buy *at least k* apples as cheaply as possible. This doesn't mean that buying exactly *k* apples is the only option: we can buy more than *k* if it's cheaper that way. We're going to start by trying to solve for exactly *k* apples, much as we solved for exactly *t* minutes in Burger Fervor. Back then, we found a way when necessary to move from exactly *t* minutes to smaller numbers of minutes. The hope is that we can do something similar here, starting with *k* apples and finding the cheapest cost for *k*, *k* + 1, *k* + 2, and so on. If it ain't broke . . .

Before just recalling the title of this chapter and diving headlong into memoization and dynamic programming, let's make sure that we really do need those tools.

What's better: buying three apples for a total of $4.00 (Scheme 1) or two apples for a total of $2.50 (Scheme 2)? We can try to answer this by calculating the cost per apple for each of these pricing schemes. In Scheme 1, we have $4.00/3 = $1.33 per apple, and in Scheme 2 we have $2.50/2 = $1.25

per apple. It looks like Scheme 2 is better than Scheme 1. Let's also suppose that we can buy one apple for $1.75. We therefore have the cost per apple, from cheapest to most expensive, as follows: $1.25, $1.33, $1.75.

Now, suppose that we want to buy *exactly* k apples. How's this for an algorithm: at each step, use the cheapest cost per apple, until we've bought k apples?

If we wanted to buy exactly four apples for the above case, then we'd start with Scheme 2, because it lets us buy apples with the best price per apple. Using Scheme 2 once costs us $2.50 for two apples, and it leaves us with two apples to buy. We can then use Scheme 2 again, buying two more apples (for a total now of four apples) for another $2.50. We'd have spent $5.00 for the four apples and, indeed, we cannot do better.

Note that just because an algorithm is intuitive or works on one test case does not mean that it is correct in general. This algorithm of using the best-available price per apple is flawed, and there are test cases that prove it. Try to find such a test case before continuing!

Here's one: suppose that we want to buy exactly three apples, not four. We'd start with Scheme 2 again, giving us two apples for a total of $2.50. Now we have only one apple to buy—and the only choice is to pay $1.75 for the one apple. The total cost is $4.25—but there is a better way. Namely, we should simply have used Scheme 1 once, costing us $4.00: yes, it has a higher cost per apple than Scheme 2, but it makes up for that by freeing us from paying for one apple that has a still higher cost per apple.

It's tempting to start affixing extra rules to our algorithm to try to fix it; for example, "if there's a pricing scheme for exactly the number of apples that we need, then use it." Suppose, however, we want to buy exactly three apples. We can easily break this augmented algorithm by adding a scheme in which the store sells three apples for $100.00.

When using memoization and dynamic programming, we try all the available options for an optimal solution, and then pick the best one. In Burger Fervor, should Homer end with an m-minute burger or an n-minute burger? We don't know, so we try both. By contrast, a *greedy algorithm* is an algorithm that doesn't try multiple options: it tries just one. Using the best price per apple, as we did above, is an example of a greedy algorithm, because at each step it chooses what to do without considering other options. Sometimes greedy algorithms work. Moreover, as they often run faster and are easier to implement than dynamic-programming algorithms, a working greedy algorithm may be better than a working dynamic-programming algorithm. For this problem, it appears that greedy algorithms—whether the one above or others that might come to mind—are not sufficiently powerful.

In Burger Fervor, we reasoned that, if it's possible to spend t minutes eating burgers, then the final burger in an optimal solution must be an m-minute burger or an n-minute burger. For the present problem, we want to say something analogous: that an optimal solution for buying k apples must end in one of a small number of ways. Here's a claim: if the available pricing schemes are Scheme 1, Scheme 2, . . . , Scheme m, then the final thing we do

must be to use one of these m pricing schemes. There can't be anything else for us to do, right?

Well, this is not quite true. The final thing that we do in an optimal solution might be buying one apple. We always have that as an option. Rather than solve two subproblems as in Burger Fervor, we solve $m + 1$ subproblems: one for each of the m pricing schemes and one for buying one apple.

Suppose that an optimal solution for buying k apples ends with us paying p dollars for n apples. We then need to buy $k - n$ apples and add that cost to p. Importantly, we need to establish that the overall optimal solution for k apples contains within it an optimal solution for $k - n$ apples. This is the optimal substructure requirement of memoization and dynamic programming. As with Burger Fervor, optimal substructure holds. If a solution for k didn't use an optimal solution for $k-n$, then that solution for k cannot be optimal after all: it's not as good as what we'd get if we built it on the optimal solution for $k - n$.

Of course, we don't know what we should do at the end of the solution to make it optimal. Do we use Scheme 1, use Scheme 2, use Scheme 3, or just buy one apple? Who knows? As in any memoization or dynamic-programming algorithm, we simply try them all and choose the best one.

Before we look at a recursive solution, note that, for any number k, we can always find a way to buy exactly k apples. Whether one apple, two apples, five apples, whatever, we can buy that many. The reason is that we always have the option of buying one apple, and we can do that as many times as we like. Compare this to Burger Fervor, where there were values of t such that t minutes could not be filled by the available burgers. As a consequence of this difference, here we won't have to worry about the case where a recursive call on a smaller subproblem fails to find a solution.

Solution 1: Recursion

Like in Burger Fervor, the first thing to do is write a helper function.

The Helper Function: Solving for the Number of Apples

Let's write the function `solve_k`, whose job will be analogous to the `solve_t` functions that we wrote for Burger Fervor. The function header is as follows:

```
double solve_k(int num[], double price[], int num_schemes,
               double unit_price, int num_items)
```

Here's what each parameter is for:

`num` An array of numbers of apples, one element per pricing scheme. For example, if we have two pricing schemes, the first for three apples and the second for two apples, then this array would be `[3, 2]`.

`price` An array of prices, one element per pricing scheme. For example, if we have two pricing schemes, the first with cost `4.00` and the second with cost `2.50`, then this array would be `[4.00, 2.50]`. Notice that

`num` and `price` together give us all of the information about the pricing schemes.

`num_schemes` The number of pricing schemes. It's the `m` value from the test case.

`unit_price` The price for one apple.

`num_items` The number of apples that we want to buy.

The `solve_k` function returns the minimum cost for buying exactly `num_items` apples.

The code for `solve_k` is given in Listing 3-9. In addition to studying this code on its own, I strongly encourage you to compare it to the `solve_t` from Burger Fervor (Listing 3-1). What differences do you notice? Why are these differences present? Memoization and dynamic-programming solutions share a common code structure. If we can nail that structure, then we can focus on what's different in and specific to each problem.

```
❶ double min(double v1, double v2) {
    if (v1 < v2)
      return v1;
    else
      return v2;
  }

  double solve_k(int num[], double price[], int num_schemes,
                 double unit_price, int num_items) {
    double best, result;
    int i;
  ❷ if (num_items == 0)
    ❸ return 0;
    else {
    ❹ result = solve_k(num, price, num_schemes, unit_price,
                           num_items - 1);
    ❺ best = result + unit_price;
      for (i = 0; i < num_schemes; i++)
      ❻ if (num_items - num[i] >= 0) {
        ❼ result = solve_k(num, price, num_schemes, unit_price,
                               num_items - num[i]);
        ❽ best = min(best, result + price[i]);
         }
          return best;
    }
  }
```

Listing 3-9: Solving for `num_items` items

We start with a little `min` function ❶: we'll need that for comparing solutions and picking the smaller one. In Burger Fervor, we used a similar `max` function, because we wanted the maximum number of burgers. Here, we

want the minimum cost. Some optimization problems are *maximization problems* (Burger Fervor) and others are *minimization problems* (Moneygrubbers)—carefully read problem statements to make sure you're optimizing in the right direction!

What do we do if asked to solve for `0` apples ❷? We return `0` ❸, because the minimum cost to buy zero apples is exactly $0.00. Our base cases are these: zero minutes to spend in Burger Fervor and zero apples to buy. As with recursion in general, at least one base case is required for any optimization problem.

If we're not in the base case, then `num_items` will be a positive integer, and we need to find the optimal way to buy exactly that many apples. The variable `best` is used to track the best (minimum-cost) option that has been found so far.

One option is to optimally solve for `num_items - 1` apples ❹ and add the cost of the final apple ❺.

We now hit the big structural difference between this problem and Burger Fervor: a loop inside of the recursive function. In Burger Fervor, we didn't need a loop, because we only had two subproblems to try. We just tried the first one and then tried the second one. Here, though, we have one subproblem per pricing scheme, and we have to go through all of them. We check whether the current pricing scheme can be used at all ❻: if its number of apples is no larger than the number that we need, then we can try it. We make a recursive call to solve the subproblem resulting from removing the number of apples in this pricing scheme ❼. (It's similar to the earlier recursive call where we subtracted one for the single apple ❹.) If that subproblem solution plus the price of the current scheme is our best option so far, then we update `best` accordingly ❽.

The solve Function

We've optimally solved for exactly k apples, but there's this detail from the problem statement that we haven't addressed yet: "You want to buy *at least* k apples and to do so as cheaply as possible." Why does the difference between exactly k apples and at least k apples matter in the first place? Can you find a test case where it's cheaper to buy more than k apples than it is to buy k apples?

Here's one for you. We'll say that one apple costs $1.75. We have two pricing schemes: Scheme 1 is that we can buy four apples for $3.00; Scheme 2 is that we can buy two apples for $2.00. Now, we want to buy at least three apples. This test case in the form of problem input is as follows:

```
1.75 2
4 3.00
2 2.00
3
```

The cheapest way to buy exactly three apples is to spend $3.75: one apple for $1.75 and two apples using Scheme 2 for $2.00. However, we can spend less money by in fact buying four apples, not three. The cheapest way

to buy four apples is to use Scheme 1 once, which costs us only $3.00. That is, the correct output for this test case is:

```
Case 1:
Buy 3 for $3.00
```

(This is a bit confusing, because we're actually buying four apples, not three, but it is correct to output `Buy 3` here. We always output the number of apples that we're asked to buy, whether or not we buy more than that to save money.)

What we need is a `solve` function like the one we had for Burger Fervor in Listing 3-2. There, we tried smaller and smaller values until we found a solution. Here, we'll try larger and larger values, keeping track of the minimum as we go. Here's a first crack at the code:

```
double solve(int num[], double price[], int num_schemes,
             double unit_price, int num_items) {
  double best;
  int i;
❶ best = solve_k(num, price, num_schemes,
                 unit_price, num_items);
❷ for (i = num_items + 1; i < ???; i++)
    best = min(best, solve_k(num, price, num_schemes,
                             unit_price, i));
  return best;
}
```

We initialize `best` to the optimal solution for buying exactly `num_items` apples ❶. Then, we use a for loop to try larger and larger numbers of apples ❷. The for loop stops when . . . uh oh. How do we know when it's safe to stop? Maybe we're being asked to buy 3 apples, but the cheapest thing to do is to buy 4 or 5 or 10 or even 20. We didn't have this problem in Burger Fervor, because there we were making our way downward, toward zero, rather than upward.

The game-saving observation is that the number of apples in a given pricing scheme is at most 100. How does this help?

Suppose we're being asked to buy at least 50 apples. Might it be best to buy exactly 60 apples? Sure! Maybe the final pricing scheme in an optimal solution for 60 apples is for 20 apples. Then we could combine those 20 apples with an optimal solution for 40 apples to get a total of 60 apples.

Suppose again that we're buying 50 apples. Could it make sense for us to buy exactly 180 apples? Well, think about an optimal solution for buying exactly 180 apples. The final pricing scheme that we use gives us at most 100 apples. Before using that final pricing scheme, we'd have bought at least 80 apples and had done so more cheaply than we did for 180 apples. Crucially, 80 is still greater than 50! Therefore, buying 80 apples is cheaper than buying 180 apples. Buying 180 apples cannot be the optimal thing to do if we want at least 50 apples.

In fact, for 50 apples, the maximum number of apples we should even consider buying is 149. If we buy 150 or more apples, than removing the final pricing scheme gives us a cheaper way to buy 50 or more apples.

The input specification for the problem not only limits the number of apples per pricing scheme to 100 but also limits the number of apples to buy to 100. In the case in which we are asked to buy 100 apples, then, the maximum number of apples we should consider buying is 100 + 99 = 199. Incorporating this observation leads to the `solve` function in Listing 3-10.

```
#define SIZE 200

double solve(int num[], double price[], int num_schemes,
             double unit_price, int num_items) {
  double best;
  int i;
  best = solve_k(num, price, num_schemes, unit_price, num_items);
  for (i = num_items + 1; i < SIZE; i++)
    best = min(best, solve_k(num, price, num_schemes,
                             unit_price, i));
  return best;
}
```

Listing 3-10: Solution 1

Now all we need is a `main` function and we can start submitting stuff to the judge.

The main Function

Let's get a `main` function written. See Listing 3-11. It's not completely self-contained—but all we'll need is one helper function, `get_number`, that I'll describe shortly.

```
#define MAX_SCHEMES 20

int main(void) {
  int test_case, num_schemes, num_items, more, i;
  double unit_price, result;
  int num[MAX_SCHEMES];
  double price[MAX_SCHEMES];
  test_case = 0;
❶ while (scanf("%lf%d", &unit_price, &num_schemes) != -1) {
    test_case++;
    for (i = 0; i < num_schemes; i++)
  ❷ scanf("%d%lf", &num[i], &price[i]);
  ❸ scanf(" ");
    printf("Case %d:\n", test_case);
    more = get_number(&num_items);
    while (more) {
```

```
      result = solve(num, price, num_schemes, unit_price,
                     num_items);
      printf("Buy %d for $%.2f\n", num_items, result);
      more = get_number(&num_items);
    }
  ❹ result = solve(num, price, num_schemes, unit_price,
                   num_items);
  ❺ printf("Buy %d for $%.2f\n", num_items, result);
  }
  return 0;
}
```

Listing 3-11: The main function

We begin by trying to read the first line of the next test case from the input ❶. The next `scanf` call ❷ is in a nested loop, and it reads the number of apples and price for each pricing scheme. The third occurrence of `scanf` ❸ reads the newline character at the end of the last line of pricing-scheme information. Reading that newline leaves us at the start of the line containing the numbers of items that we are asked to buy. We can't just airily keep calling `scanf` to read those numbers, though, because we have to be able to stop at a newline. I address this with my `get_number` helper function, described further below. It returns `1` if there are more numbers to read and `0` if this is the last number on the line. This explains the code below the loop ❹ ❺: when the loop terminates because it has read the final number on the line, we still need to solve that final test case.

The code for `get_number` is given in Listing 3-12.

```
int get_number(int *num) {
  int ch;
  int ret = 0;
  ch = getchar();
❶ while (ch != ' ' && ch != '\n') {
    ret = ret * 10 + ch - '0';
    ch = getchar();
  }
❷ *num = ret;
❸ return ch == ' ';
}
```

Listing 3-12: The function to get an integer

This function reads an integer value using an approach reminiscent of Listing 2-17. The loop continues as long as we haven't yet hit a space or newline character ❶. When the loop terminates, we store what was read in the pointer parameter passed to this function call ❷. I use that pointer parameter, rather than return the value, because I use the return value for something else: to indicate whether or not this is the last number on the line ❸. That is, if `get_number` returns `1` (because it found a space after the number

that it read), it means that there are more numbers on this line; if it returns `0`, then this is the final integer on this line.

We've got a complete solution now, but its performance is glacial. Even test cases that look small will take ages, because we're going all the way up to 299 apples no matter what.

Oh well. Let's memoize the heck out of this thing.

Solution 2: Memoization

When memoizing Burger Fervor, we introduced the `memo` array in `solve` (Listing 3-5). That was because each call of `solve` was for an independent test case. However, in Moneygrubbers, we have that line where each integer specifies a number of apples to buy, and we have to solve each one. It would be wasteful to throw away the `memo` array before we've completely finished with a test case!

We're therefore going to declare and initialize `memo` in `main`; see Listing 3-13 for the updated function.

```
int main(void) {
  int test_case, num_schemes, num_items, more, i;
  double unit_price, result;
  int num[MAX_SCHEMES];
  double price[MAX_SCHEMES];
❶ double memo[SIZE];
  test_case = 0;
  while (scanf("%lf%d", &unit_price, &num_schemes) != -1) {
    test_case++;
    for (i = 0; i < num_schemes; i++)
      scanf("%d%lf", &num[i], &price[i]);
    scanf(" ");
    printf("Case %d:\n", test_case);
  ❷ for (i = 0; i < SIZE; i++)
    ❸ memo[i] = -1;
    more = get_number(&num_items);
    while (more) {
      result = solve(num, price, num_schemes, unit_price
                     num_items, memo);
      printf("Buy %d for $%.2f\n", num_items, result);
      more = get_number(&num_items);
    }
    result = solve(num, price, num_schemes, unit_price,
                   num_items, memo);
    printf("Buy %d for $%.2f\n", num_items, result);
  }
  return 0;
}
```

Listing 3-13: The main function, with memoization implemented

We declare the `memo` array ❶, and we set each element of `memo` to `-1` ("unknown" value) ❷ ❸. Notice that the initialization of `memo` occurs just once per test case. The only other change is that we add `memo` as a new parameter to the `solve` calls.

The new code for `solve` is given in Listing 3-14.

```
double solve(int num[], double price[], int num_schemes,
             double unit_price, int num_items, double memo[]) {
  double best;
  int i;
  best = solve_k(num, price, num_schemes, unit_price,
                 num_items, memo);
  for (i = num_items + 1; i < SIZE; i++)
    best = min(best, solve_k(num, price, num_schemes,
                             unit_price, i, memo));
  return best;
}
```

Listing 3-14: Solution 2, with memoization implemented

In addition to adding `memo` as a new parameter at the end of the parameter list, we pass `memo` to the `solve_k` calls. That's it.

Finally, let's take a look at the changes required to memoize `solve_k`. We will store in `memo[num_items]` the minimum cost of buying exactly `num_items` apples. See Listing 3-15.

```
double solve_k(int num[], double price[], int num_schemes,
               double unit_price, int num_items, double memo[]) {
  double best, result;
  int i;
❶ if (memo[num_items] != -1)
    return memo[num_items];
  if (num_items == 0) {
    memo[num_items] = 0;
    return memo[num_items];
  } else {
    result = solve_k(num, price, num_schemes, unit_price,
                     num_items - 1, memo);
    best = result + unit_price;
    for (i = 0; i < num_schemes; i++)
      if (num_items - num[i] >= 0) {
        result = solve_k(num, price, num_schemes, unit_price,
                         num_items - num[i], memo);
        best = min(best, result + price[i]);
      }
        memo[num_items] = best;
```

```
        return memo[num_items];
    }
}
```

Listing 3-15: Solving for `num_items` items, with memoization implemented

Remember that the first thing we do when solving with memoization is check whether the solution is already known ❶. If any value besides `-1` is stored for the `num_items` subproblem, we return it. Otherwise, as with any memoized function, we store a new subproblem solution in `memo` before returning it.

We've now reached a natural stopping point for this problem: this memoized solution can be submitted to the judge and should pass all test cases. If you'd like more practice with dynamic programming, though, here's a perfect opportunity for you to convert this memoized solution into a dynamic-programming solution! Otherwise, we'll put this problem on ice.

Problem 3: Hockey Rivalry

Our first two problems used a one-dimensional `memo` or `dp` array. Let's look at a problem whose solution dictates using a two-dimensional array.

I live in Canada, so I suppose we weren't getting through this book without some hockey. Hockey is a team sport like soccer . . . but with goals.

This is DMOJ problem `cco18p1`.

The Problem

The Geese played *n* games, each of which had one of two outcomes: a win for the Geese (`W`) or a loss for the Geese (`L`). There are no tie games. For each of their games, we know whether they won or lost, and we know the number of goals that they scored. For example, we might know that their first game was a win (`W`) and that they scored four goals in that game. (Their opponent must therefore have scored fewer than four goals.) The Hawks also played *n* games and, the same as the Geese, each game was a win or loss for the Hawks. Again, for each of their games, we know whether they won or lost, and we know the number of goals that they scored.

Some of the games that these teams played may have been against each other, but there are other teams, too, and some of the games may have been against these other teams.

We have no information about who played whom. We might know that the Geese won a certain game and that they scored four goals in that game, but we don't know who their opponent was—their opponent could have been the Hawks but also could have been some other team.

A *rivalry game* is a game where the Geese played the Hawks.

Our task is to determine the maximum number of goals that could have been scored in rivalry games.

Input

The input contains one test case, the information for which is spread over five lines as follows:

- The first line contains n, the number of games that each team played. n is between 1 and 1,000.
- The second line contains a string of length n, where each character is a `W` (win) or `L` (loss). This line tells us the outcome of each game played by the Geese. For example, `WLL` means that the Geese won their first game, lost their second game, and lost their third game.
- The third line contains n integers, giving the number of goals scored in each game by the Geese. For example, `4 1 2` means that the Geese scored four goals in their first game, one goal in their second game, and two goals in their third game.
- The fourth line is like the second and tells us the outcome of each game for the Hawks.
- The fifth line is like the third and tells us the number of goals scored in each game by the Hawks.

Output

The output is a single integer: the maximum number of goals scored in possible rivalry games.

The time limit for solving the test case is one second.

About Rivalries

Before jumping to the structure of optimal solutions, let's be sure that we understand exactly what's being asked by working through some test cases.

We'll start with this one:

```
3
WWW
2 5 1
WWW
7 8 5
```

There can't be *any* rivalry games at all here. A rivalry game, like any game, requires that one team win and the other lose—but the Geese won all their games and the Hawks won all their games, so the Geese and Hawks could not have played each other. Since there are no rivalry games possible, there are no goals scored in rivalry games. The correct output is `0`.

Let's now have the Hawks lose all their games:

```
3
WWW
2 5 1
```

```
LLL
7 8 5
```

Are there any rivalry games now? The answer is still no! The Geese won their first game by scoring two goals. For that game to be a rivalry game, it must be a game where the Hawks lost and where the Hawks scored fewer than two goals. Since the fewest goals scored by the Hawks was five though, none of those games can be a rivalry game with the Geese's first game. Similarly, the Geese won their second game by scoring five goals, but there is no loss for the Hawks where they scored four goals or fewer. That is, there is no rivalry involving the Geese's second game. The same kind of analysis shows that the Geese's third game also cannot be part of a rivalry. Again, `0` is the correct output.

Let's move past these zero cases. Here's one:

```
3
WWW
2 5 1
LLL
7 8 4
```

We've changed the last Hawks game so that they scored four goals instead of five, and this is enough to produce a possible rivalry game! Specifically, the second game played by the Geese, where the Geese won and scored five goals, could be a rivalry game with the third game by the Hawks, where the Hawks lost and scored four goals. That game had nine goals scored in it, so the correct output here is `9`.

Now consider this one:

```
2
WW
6 2
LL
8 1
```

Look at the final game that each team played: the Geese won and scored two goals, and the Hawks lost and scored one goal. That could be a rivalry game, with a total of three goals. The first game played by each team cannot be a rivalry game (the Geese won with six goals and the Hawks could not have lost the same game with eight goals), so we can't add any more goals. Is `3` the correct output?

It is not! We chose poorly, matching those final games. What we should have done is match the first game played by the Geese with the second game played by the Hawks. That could be a rivalry game, and it has seven goals. This time we've got it: the correct output is `7`.

Let's look at one more example. Try to figure out the maximum before reading my answer:

```
4
WLWW
3 4 1 8
WLLL
5 1 2 3
```

The correct output is `20`, witnessed by having two rivalry games: the second Geese game with the first Hawks game (9 goals there) and the fourth Geese game with the fourth Hawks game (11 goals there).

Characterizing Optimal Solutions

Consider an optimal solution to this problem: a solution that maximizes the number of goals scored in rivalry games. What might this optimal solution look like? Assume that the games for each team are numbered from one to n.

Option 1. One option is that the optimal solution uses the final game n played by the Geese and the final game n played by the Hawks as a rivalry game. That game has a certain number of goals scored in it: call that g. We can then strip out both of these games and optimally solve the smaller subproblem on the Geese's first $n - 1$ games and the Hawks' first $n - 1$ games. That subproblem solution, plus g, is the optimal solution overall. Note, though, that this option is only available if the two n games can really be a rivalry game. For example, if both teams have a `W` for that game, then this cannot be a rivalry game, and Option 1 cannot apply.

Remember this test case from the prior section?

```
4
WLWW
3 4 1 8
WLLL
5 1 2 3
```

That's an example of Option 1: we match the two rightmost scores, `8` and `3`, and then optimally solve the subproblem for the remaining games.

Option 2. Another option is that the optimal solution has nothing to do with these final games at all. In that case, we strip out game n played by the Geese and game n played by the Hawks, and we optimally solve the subproblem on the Geese's first $n - 1$ games and the Hawks' first $n - 1$ games.

The first test case from the prior section is an example of Option 2:

```
3
WWW
2 5 1
WWW
7 8 5
```

The 1 and 5 at the right are not part of an optimal solution. The optimal solution for the other games is the optimal solution overall.

So far we've covered the cases where both game n scores are used and where neither game n score is used. Are we done?

To see that we are not done, consider this test case from the prior section:

```
2
WW
6 2
LL
8 1
```

Option 1, matching the 2 and 1, leads to a maximum of three goals in rivalry games. Option 2, throwing away both the 2 and 1, leads to a maximum of zero goals in rivalry games. However, the maximum overall here is seven. Our coverage of types of optimal solutions, using only Option 1 and Option 2, is therefore spotty.

What we need to be able to do here is drop a game from the Geese but not from the Hawks. Specifically, we'd like to drop the Geese's second game and then solve the subproblem consisting of the Geese's first game and *both* of the Hawks' games. For symmetry, we should also be able to drop the second Hawks game and solve the resulting subproblem on the first Hawks game and both Geese games. Let's get these two additional options in there.

Option 3. Our third option is that the optimal solution has nothing to do with the Geese's game n. In that case, we strip out game n played by the Geese, and we optimally solve the subproblem on the Geese's first $n - 1$ games and the Hawks' first n games.

Option 4. Our fourth and final option is that the optimal solution has nothing to do with the Hawks' game n. In that case, we strip out game n played by the Hawks, and we optimally solve the subproblem on the Geese's first n games and the Hawks' first $n - 1$ games.

Options 3 and 4 induce a change in the structure of a solution to this problem—whether that solution uses recursion, memoization, or dynamic programming. In the previous problems of this chapter, our subproblems were characterized by only one parameter: t for Burger Fervor and k for Moneygrubbers. Without Options 3 and 4, we'd have gotten away with a single parameter, n, for the Hockey Rivalry problem, too. That n parameter would have reflected the fact that we were solving a subproblem for the first n games played by the Geese and the first n games played by the Hawks. With Options 3 and 4 in the mix, however, these n values are no longer yoked: one can change when the other does not. For example, if we're solving a subproblem concerning the first five games played by the Geese, this does not mean that we're stuck looking at the first five games played by the Hawks. Symmetrically, a subproblem concerning the first five games played by the Hawks doesn't tell us anything about the number of games played by the Geese.

We therefore need two parameters for our subproblems: *i*, the number of games played by the Geese, and *j*, the number of games played by the Hawks.

For a given optimization problem, the number of subproblem parameters could be one, two, three, or more. When confronting a new problem, I suggest beginning with one subproblem parameter. Then, think about the possible options for an optimal solution. Perhaps each option can be solved by solving one-parameter subproblems, in which case additional parameters are not required. However, sometimes it will be that one or more options require the solution to a subproblem that cannot be pinned down by one parameter. In these cases, a second parameter can often help.

The benefit of adding additional subproblem parameters is the larger subproblem space in which to couch our optimal solutions. The cost is the responsibility of solving more subproblems. Keeping the number of parameters small—one, two, or perhaps three—is key for designing fast solutions to optimization problems.

Solution 1: Recursion

It's now time for our recursive solution. Here's the `solve` function that we'll write this time:

```
int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int i, int j)
```

As always, the parameters are of two types: information from the test case and information about the current subproblem. Here are brief descriptions of the parameters:

`outcome1` The array of `W` and `L` characters for the Geese.

`outcome2` The array of `W` and `L` characters for the Hawks.

`goals1` The array of goals scored for the Geese.

`goals2` The array of goals scored for the Hawks.

`i` The number of Geese games that we're considering in this subproblem.

`j` The number of Hawks games that we're considering in this subproblem.

The last two parameters are the ones specific to the current subproblem, and they are the only parameters that change on recursive calls.

If we started each of the arrays at index `0`, as is standard for C arrays, then we'd have to keep in our minds that information for some game `k` was not at index `k` but at index `k-1`. For example, information about game four would be at index `3`. To avoid this, we'll store information about games starting at index `1`. In that way, information about game four will be at index `4`. This leaves us with one less mistake to make!

The code for the recursive solution is given in Listing 3-16.

```
❶ int max(int v1, int v2) {
    if (v1 > v2)
      return v1;
    else
      return v2;
  }

  int solve(char outcome1[], char outcome2[], int goals1[],
            int goals2[], int i, int j) {
  ❷ int first, second, third, fourth;
  ❸ if (i == 0 || j == 0)
      return 0;
  ❹ if ((outcome1[i] == 'W' && outcome2[j] == 'L' &&
         goals1[i] > goals2[j]) ||
        (outcome1[i] == 'L' && outcome2[j] == 'W' &&
         goals1[i] < goals2[j]))
    ❺ first = solve(outcome1, outcome2, goals1, goals2, i - 1, j - 1) +
              goals1[i] + goals2[j];
    else
      first = 0;
  ❻ second = solve(outcome1, outcome2, goals1, goals2, i - 1, j - 1);
  ❼ third = solve(outcome1, outcome2, goals1, goals2, i - 1, j);
  ❽ fourth = solve(outcome1, outcome2, goals1, goals2, i, j - 1);
  ❾ return max(first, max(second, max(third, fourth)));
  }
```

Listing 3-16: Solution 1

This is a maximization problem: we want to maximize the number of goals scored in rivalry games. We start with a `max` function ❶—we'll use that when we need to determine which of the options is best. We then declare four integer variables, one for each of the four options ❷.

Let's begin with base cases: what do we return if both `i` and `j` are `0`? In this case, the subproblem is for the first zero Geese games and zero Hawks games. Since there are no games, there are certainly no rivalry games; and since there are no rivalry games, there are no goals scored in rivalry games. We should therefore return `0` here.

That isn't the only base case though. For example, consider the subproblem where the Geese play zero games (`i = 0`) and the Hawks play three games (`j = 3`). As with the case in the prior paragraph, there can't be any rivalry games here, because the Geese don't have any games! A similar situation arises when the Hawks play zero games: even if the Geese play some games, none of them can be against the Hawks.

That captures all of the base cases. That is, if `i` has value `0` *or* `j` has value `0`, then we have zero goals scored in rivalry games ❸.

With the base cases out of the way, we must now try the four possible options for an optimal solution and choose the best one.

Option 1. Recall that this option is valid only when the final Geese game and final Hawks game can be a rivalry game. There are two ways for this game to be a rivalry game:

1. The Geese win, the Hawks lose, and the Geese score more goals than the Hawks.
2. The Geese lose, the Hawks win, and the Geese score fewer goals than the Hawks.

We encode these two possibilities ❹. If the game can be a rivalry game, we compute the optimal solution for this case ❺: it consists of the optimal solution for the first `i-1` Geese games and `j-1` Hawks games plus the total goals scored in the rivalry game.

Option 2. For this one, we solve the subproblem for the first `i-1` Geese games and `j-1` Hawks games ❻.

Option 3. Here, we solve the subproblem for the first `i-1` Geese games and `j` Hawks games ❼. Notice that `i` changes but `j` does not. This is exactly why we need two subproblem parameters here, not one.

Option 4. We solve the subproblem for the first `i` Geese games and `j-1` Hawks games ❽. Again, one subproblem parameter changes but the other does not; it's a good thing there's no need for us to keep them at the same value!

There we go: `first`, `second`, `third`, and `fourth`—those are the only four possibilities for our optimal solution. We want the maximum of these, and that is what we compute and return ❾. The innermost `max` call calculates the maximum of `third` and `fourth`. Working outward, the next `max` call calculates the maximum of that winner and `second`. Finally, the outermost call calculates the maximum of that winner and `first`.

We're just about there. All we need now is a `main` function that reads the five lines of input and calls `solve`. The code is given in Listing 3-17. Compared to the `main` function for Moneygrubbers, this is not bad!

```
#define SIZE 1000

int main(void) {
  int i, n, result;
❶ char outcome1[SIZE + 1], outcome2[SIZE + 1];
❷ int goals1[SIZE + 1], goals2[SIZE + 1];
❸ scanf("%d ", &n);
  for (i = 1; i <= n; i++)
    scanf("%c", &outcome1[i]);
  for (i = 1; i <= n; i++)
    scanf("%d ", &goals1[i]);
  for (i = 1; i <= n; i++)
    scanf("%c", &outcome2[i]);
  for (i = 1; i <= n; i++)
    scanf("%d ", &goals2[i]);
  result = solve(outcome1, outcome2, goals1, goals2, n, n);
```

```
  printf("%d\n", result);
  return 0;
}
```

Listing 3-17: The main function

We declare the outcome (`W` and `L`) ❶ and goals-scored arrays ❷. The `+ 1` there is because of our choice to begin indexing at `1`. If we had used just `SIZE`, then valid indices would go from zero to 999, when what we need is to include index 1,000.

We then read the integer on the first line ❸, which gives the number of games played by the Geese and Hawks. There's a space right after the `%d` and before the closing quote. That space causes `scanf` to read whitespace following the integer. Crucially, this reads the newline character at the end of the line, which otherwise would be included when we use `scanf` to read individual characters . . . which we do next!

We read the `W` and `L` information for the Geese and then read the goals-scored information for the Geese. We then do the same for the Hawks. Finally, we call `solve`. We want to solve the problem on all n Geese games and all n Hawks games, which explains why the last two arguments are `n`.

Is there any chance you'll submit this solution to the judge? The "Time-Limit Exceeded" error should come as no surprise.

Solution 2: Memoization

In Burger Fervor and Moneygrubbers, we used a one-dimensional array for the memo. That's because our subproblems had but one parameter: the number of minutes and number of items, respectively. In contrast, subproblems in Hockey Rivalry have two parameters, not one. We'll correspondingly need a memo array with two dimensions, not one. Element `memo[i][j]` is used to hold the solution to the subproblem on the first `i` Geese games and the first `j` Hawks games. Other than switching from one to two dimensions in the memo, the technique remains as before: return the solution if it's already stored, calculate and store it if it's not.

The updated `main` function is given in Listing 3-18.

```
int main(void) {
  int i, j, n, result;
  char outcome1[SIZE + 1], outcome2[SIZE + 1];
  int goals1[SIZE + 1], goals2[SIZE + 1];
  static int memo[SIZE + 1][SIZE + 1];
  scanf("%d ", &n);
  for (i = 1; i <= n; i++)
    scanf("%c", &outcome1[i]);
  for (i = 1; i <= n; i++)
    scanf("%d ", &goals1[i]);
  for (i = 1; i <= n; i++)
    scanf("%c", &outcome2[i]);
  for (i = 1; i <= n; i++)
```

```
    scanf("%d ", &goals2[i]);
  for (i = 0; i <= SIZE; i++)
    for (j = 0; j <= SIZE; j++)
      memo[i][j] = -1;
  result = solve(outcome1, outcome2, goals1, goals2, n, n, memo);
  printf("%d\n", result);
  return 0;
}
```

Listing 3-18: The main function, with memoization implemented

Notice that the `memo` array is huge—over 1 million elements—so we make the array static as in Listing 1-8.

The memoized `solve` function is given in Listing 3-19.

```
int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int i, int j, int memo[SIZE + 1][SIZE + 1]) {
  int first, second, third, fourth;
  if (memo[i][j] != -1)
    return memo[i][j];
  if (i == 0 || j == 0) {
    memo[i][j] = 0;
    return memo[i][j];
  }
  if ((outcome1[i] == 'W' && outcome2[j] == 'L' &&
       goals1[i] > goals2[j]) ||
      (outcome1[i] == 'L' && outcome2[j] == 'W' &&
       goals1[i] < goals2[j]))
    first = solve(outcome1, outcome2, goals1, goals2, i - 1, j - 1, memo) +
            goals1[i] + goals2[j];
  else
    first = 0;
  second = solve(outcome1, outcome2, goals1, goals2, i - 1, j - 1, memo);
  third = solve(outcome1, outcome2, goals1, goals2, i - 1, j, memo);
  fourth = solve(outcome1, outcome2, goals1, goals2, i, j - 1, memo);
  memo[i][j] = max(first, max(second, max(third, fourth)));
  return memo[i][j];
}
```

Listing 3-19: Solution 2, with memoization implemented

This solution passes all test cases and does so quickly. If we simply wanted to solve this problem, we would stop right now, but here we have the opportunity to plumb further and learn more about dynamic programming as we do so.

Solution 3: Dynamic Programming

We just saw that to memoize this problem we needed a two-dimensional `memo` array, not a one-dimensional array. To develop a dynamic-programming

solution, we'll correspondingly need a two-dimensional dp array. In Listing 3-18, we declared the memo array like this:

```
static int memo[SIZE + 1][SIZE + 1];
```

and we'll do likewise for the dp array:

```
static int dp[SIZE + 1][SIZE + 1];
```

As in the memo array, element dp[i][j] will hold the subproblem solution for the first i Geese games and first j Hawks games. Our task, then, is to solve each of these subproblems and return dp[n][n] once we're done.

In memoized solutions to optimization problems, it's not our responsibility to determine an order in which to solve the subproblems. We make our recursive calls, and those calls return to us the solutions for their corresponding subproblems. In dynamic-programming solutions, however, it *is* our responsibility to determine an order in which to solve the subproblems. We can't just solve them in any order we want, because then a subproblem solution might not be available when we need it.

For example, suppose we wanted to fill in dp[3][5]—that's the cell for the first three Geese games and the first five Hawks games. Take another look back at the four options for an optimal solution.

- Option 1 requires us to look up dp[2][4].
- Option 2 also requires us to look up dp[2][4].
- Option 3 requires us to look up dp[2][5].
- Option 4 requires us to look up dp[3][4].

We must arrange it so that these elements of dp are already stored by the time we want to store dp[3][5].

For subproblems with only one parameter, you generally solve those subproblems from smallest index to largest index. For subproblems with more than one parameter, things are not so simple, as there are many more orders in which the array can be filled. Only some of these orders maintain the property that a subproblem solution is available by the time we need it.

For the Hockey Rivalry problem, we can solve dp[i][j] if we have already stored dp[i-1][j-1] (Option 1 and Option 2), dp[i-1][j] (Option 3), and dp[i][j-1] (Option 4). One technique we can use is to solve all of the dp[i-1] subproblems before solving any of the dp[i] subproblems. For example, this would result in dp[2][4] being solved before dp[3][5], which is exactly what we need to satisfy Options 1 and 2. It would also result in dp[2][5] being solved before dp[3][5], which is what we need for Option 3. That is, solving row i-1 before row i satisfies Options 1 to 3.

To satisfy Option 4, we can solve the dp[i] subproblems from smallest j index to largest j index. That, for example, would solve dp[3][4] before dp[3][5].

In summary, we solve all of the subproblems in row 0 from left to right, then all of the subproblems in row 1 from left to right, and so on, until we have solved all subproblems in row n.

The `solve` function for our dynamic-programming solution is given in Listing 3-20.

```
int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int n) {
  int i, j;
  int first, second, third, fourth;
  static int dp[SIZE + 1][SIZE + 1];
  for (i = 0; i <= n; i++)
    dp[0][i] = 0;
  for (i = 0; i <= n; i++)
    dp[i][0] = 0;
❶ for (i = 1; i <= n; i++)
  ❷ for (j = 1; j <= n; j++) {
      if ((outcome1[i] == 'W' && outcome2[j] == 'L' &&
           goals1[i] > goals2[j]) ||
          (outcome1[i] == 'L' && outcome2[j] == 'W' &&
           goals1[i] < goals2[j]))
        first = dp[i-1][j-1] + goals1[i] + goals2[j];
      else
        first = 0;
      second = dp[i-1][j-1];
      third = dp[i-1][j];
      fourth = dp[i][j-1];
      dp[i][j] = max(first, max(second, max(third, fourth)));
    }
❸ return dp[n][n];
}
```

Listing 3-20: Solution 3, with dynamic programming

We begin by initializing the base case subproblems, which are those in which at least one of the indices is `0`. Then, we hit the double for loop ❶ ❷, which controls the order in which the non-base-case subproblems are solved. We first range over the rows ❶ and then the elements in each row ❷, which, as we have argued, is a valid order for solving the subproblems. Once we have filled in the table, we return the solution for the original problem ❸.

We can visualize the array produced by a two-dimensional dynamic-programming algorithm as a table. This is helpful for getting a feel for how the elements of the array are filled in. Let's look at the final array for the following test case:

```
4
WLWW
3 4 1 8
WLLL
5 1 2 3
```

Here's the resulting array:

	0	1	2	3	4
4	0	9	18	19	20
3	0	9	9	9	9
2	0	9	9	9	9
1	0	0	4	5	5
0	0	0	0	0	0

Consider, for example, the computation for the element in row `4`, column `2` or, in terms of the `dp` table, `dp[4][2]`. This is the subproblem for the first four Geese games and first two Hawks games. Looking at the Geese's game four and the Hawks' game two, we see that the Geese won with eight goals and the Hawks lost with one goal, so this game could be a rivalry game. Option 1 is therefore a possible option. Nine goals were scored in this game. To that nine, we add the value at row `3`, column `1`, which is nine again. This gives us a total of 18. That's our maximum so far—now we have to try Options 2 to 4 to see whether they are better. If you do that, you should observe that they all happen to have the value nine. We therefore store 18, the maximum of all available options, in `dp[4][2]`.

The only quantity of real interest here, of course, is that in the topmost, rightmost cell, corresponding to the subproblem on the full `n` games for the Geese and `n` games for the Hawks. That value, 20, is what we return as the optimal solution. The other quantities in the table are only useful insofar as they help us make progress toward calculating that 20.

In terms of the `main` function, we make one small change to the code of Listing 3-17: the only thing to do is remove the final `n` passed to `solve`, resulting in

```
result = solve(outcome1, outcome2, goals1, goals2, n);
```

A Space Optimization

I mentioned in "Step 4: Dynamic Programming" on page 90 that memoization and dynamic programming are roughly equivalent. *Roughly*, because sometimes there are benefits to be had by choosing one or the other. The Hockey Rivalry problem furnishes an example of a typical optimization that we can perform when using dynamic programming but not when using memoization. The optimization is not one of speed but of space.

Here's the key question: when solving a subproblem in row `i` of the `dp` array, which rows do we access? Look back at the four options. The only rows used are `i-1` (the previous row) and `i` (the current row). There's no `i-2` or `i-3` or anything else in there. As such, keeping the entire two-dimensional array in memory is wasteful. Suppose we're solving subproblems in row 500. All we need is access to row 500 and row 499. We may as well not have row 498 or 497 or 496 or any other row in memory, because we'll never look at these again.

Rather than a two-dimensional table, we can pull through with only two one-dimensional arrays: one for the previous row and one for the current row we are solving.

Listing 3-21 implements this optimization.

```
int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int n) {
  int i, j, k;
  int first, second, third, fourth;
  static int previous[SIZE + 1], current[SIZE + 1];
❶ for (i = 0; i <= n; i++)
  ❷ previous[i] = 0;
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
      if ((outcome1[i] == 'W' && outcome2[j] == 'L' &&
           goals1[i] > goals2[j]) ||
          (outcome1[i] == 'L' && outcome2[j] == 'W' &&
           goals1[i] < goals2[j]))
        first = previous[j-1] + goals1[i] + goals2[j];
      else
        first = 0;
      second = previous[j-1];
      third = previous[j];
      fourth = current[j-1];
      current[j] = max(first, max(second, max(third, fourth)));
    }
  ❸ for (k = 0; k <= SIZE; k++)
    ❹ previous[k] = current[k];
  }
  return current[n];
}
```

Listing 3-21: Solution 3, with space optimization implemented

We initialize `previous` to all zeros ❶ ❷, thereby solving all subproblems in row `0`. In the rest of the code, whenever we previously referred to row `i-1`, we now use `previous`. In addition, whenever we previously referred to row `i`, we now use `current`. Once a new row has been fully solved and stored in `current`, we copy `current` into `previous` ❸ ❹ so that `current` can be used to solve the next row.

Problem 4: Ways to Pass

Here's one final (very short!) example. Our first three problems in this chapter asked us to maximize (Burger Fervor and Hockey Rivalry) or minimize (Moneygrubbers) the value of a solution. I'd like to end the chapter with a problem of a slightly different flavor: rather than find an optimal solution, we'll count the number of possible solutions. We'll see that we can once again count on memoization and dynamic programming.

This is UVa problem 10910.

The Problem

Passing a course requires at least p marks. (p isn't necessarily 50 or 60 or whatever is needed in school; it could be any positive integer.) A student took n courses and passed them all.

Adding up all of the student's marks in these n courses gives a total of t marks that the student earned, but we don't know how many marks the student earned in each course. So we ask the following: in how many distinct ways could the student have passed all of the courses?

For example, suppose that the student took two courses and earned a total of nine marks and that each course requires at least three marks to pass. Then, there are four ways in which the student could have passed these courses:

- three marks in course 1 and six marks in course 2
- four marks in course 1 and five marks in course 2
- five marks in course 1 and four marks in course 2
- six marks in course 1 and three marks in course 2

Input

The first line of input is an integer k indicating the number of test cases to follow. Each of the k test cases is on its own line and consists of three integers: n (the number of courses taken, all of which the student passed), t (total marks earned), and p (marks required to pass each course). Each n, t, and p value is between 1 and 70.

Here is the input for the above example:

```
1
2 9 3
```

Output

For each test case, output the number of ways that the marks can be distributed so that the student passes all courses. For the above example, the output would be the integer `4`.

The time limit for solving the test cases is three seconds.

Solution: Memoization

Notice here that there's no optimal way to distribute the marks. A student crushing it in one course and barely passing all others is as good a solution as any. (As a teacher, that was hard to write.)

As there's no optimal solution, it doesn't make sense to think about the structure of an optimal solution. Rather, let's think about what any solution must look like. In the first course, the student must have earned at least p

marks and at most t marks. Each of these choices leads to a new subproblem with one fewer course. Suppose that the student earns m marks in the first course. Then we solve the subproblem on $n - 1$ courses in which the student earned exactly $t - m$ marks.

Rather than using `max` or `min` to choose a best solution, we use addition to total the number of solutions.

With practice, you'll often be able to identify when memoization or dynamic programming is required without first stepping through a nonperformant recursive solution. Memoization adds so little code to a recursive solution that it can make sense to come out of the gate with memoization. I present a complete, memoized solution to Ways to Pass in Listing 3-22.

```
#define SIZE 70

int solve(int n, int t, int p, int memo[SIZE + 1][SIZE + 1]) {
  int total, m;
  if (memo[n][t] != -1)
    return memo[n][t];
❶ if (n == 0 && t == 0)
    return 1;
❷ if (n == 0)
    return 0;
  total = 0;
  for (m = p; m <= t; m++)
    total = total + solve(n - 1, t - m, p, memo);
  memo[n][t] = total;
  return memo[n][t];
}

int main(void) {
  int k, i, x, y, n, t, p;
  int memo[SIZE + 1][SIZE + 1];
  scanf("%d", &k);
  for (i = 0; i < k; i++) {
    scanf("%d%d%d", &n, &t, &p);
    for (x = 0; x <= SIZE; x++)
      for (y = 0; y <= SIZE; y++)
        memo[x][y] = -1;
    printf("%d\n", solve(n, t, p, memo));
  }
  return 0;
}
```

Listing 3-22: Solution with memoization implemented

What's going on with the base cases ❶ ❷? The base case is when the number of courses `n` is `0`, but there are two subcases here. First, suppose that `t` is also `0`. How many ways are there to distribute zero marks to pass zero courses? It's easy to err here and say that the answer is zero—but the

answer is one, because we can succeed here by not allocating any marks at all. That's certainly a way to pass zero courses! Now, what if `n` is `0` but `t` is greater than `0`? Here the answer really is zero: there is no way to distribute a positive number of marks across zero courses.

The rest of the code tries each legal number `m` of marks for the current course, and it solves the subproblem with one fewer course and `m` fewer marks to distribute.

Summary

I've presented what I think of as the core of memoization and dynamic programming: explicating the structure of an optimal solution, developing a recursive algorithm, speeding it up through memoization, and optionally replacing the recursion by filling a table. Once you're comfortable solving problems with one- or two-dimensional tables, I'd suggest working on problems where three or more dimensions are required. The principles are the same as what I've presented here, but you'll have to work harder to discover and relate the subproblems.

Ideas related to dynamic programming often make cameos in other algorithms. In the next chapter, for example, you'll see that we'll once again store results for later lookup. In Chapter 6, you'll see a problem in which dynamic programming plays a supporting role, speeding up computation required by the main algorithm of interest.

Notes

Hockey Rivalry is originally from the 2018 Canadian Computing Olympiad.

Many algorithm textbooks delve deeper into the theory and application of memoization and dynamic programming. My favorite treatment is *Algorithm Design* by Jon Kleinberg and Éva Tardos (2006).