CHAPTER 5

FROM RECURSION TO ITERATORS

We've already seen that iterators are useful when a source of data is prepared to deliver more data than we want, or when it takes a long time to come up with each data item and we don't want to waste time by computing more of them than we need to.

Both conditions occur frequently in conjunction with recursive functions. Recursive functions are often used for searching large, hierarchical spaces for solutions to some specification. If solutions are common, the space will contain more of them than we want to use; if solutions are rare, they will take a long time to find. In either case, we don't want our program to have to populate an array with all the possible solutions before it can continue, and it is natural to use an iterator.

We saw another reason to get rid of recursion in the web robot example in Chapter 4: Recursive functions naturally perform depth-first searches. When this is inappropriate, as for a web robot, recursion offers no escape. With an iterator solution, we can order the queue any way we like or even reorder it dynamically when new information arrives.

But recursive functions are often easy to write, whereas iterators seemed to require ingenuity. In this chapter, we'll look at techniques for transforming general recursive functions into iterators.

5.1 THE PARTITION PROBLEM REVISITED

As our prototypical example of such a problem, we're going to look at the *partition problem*, which we saw in Chapters 1 and 3. This is a simple but common problem that arises in many contexts, most commonly in optimization and operations research problems.

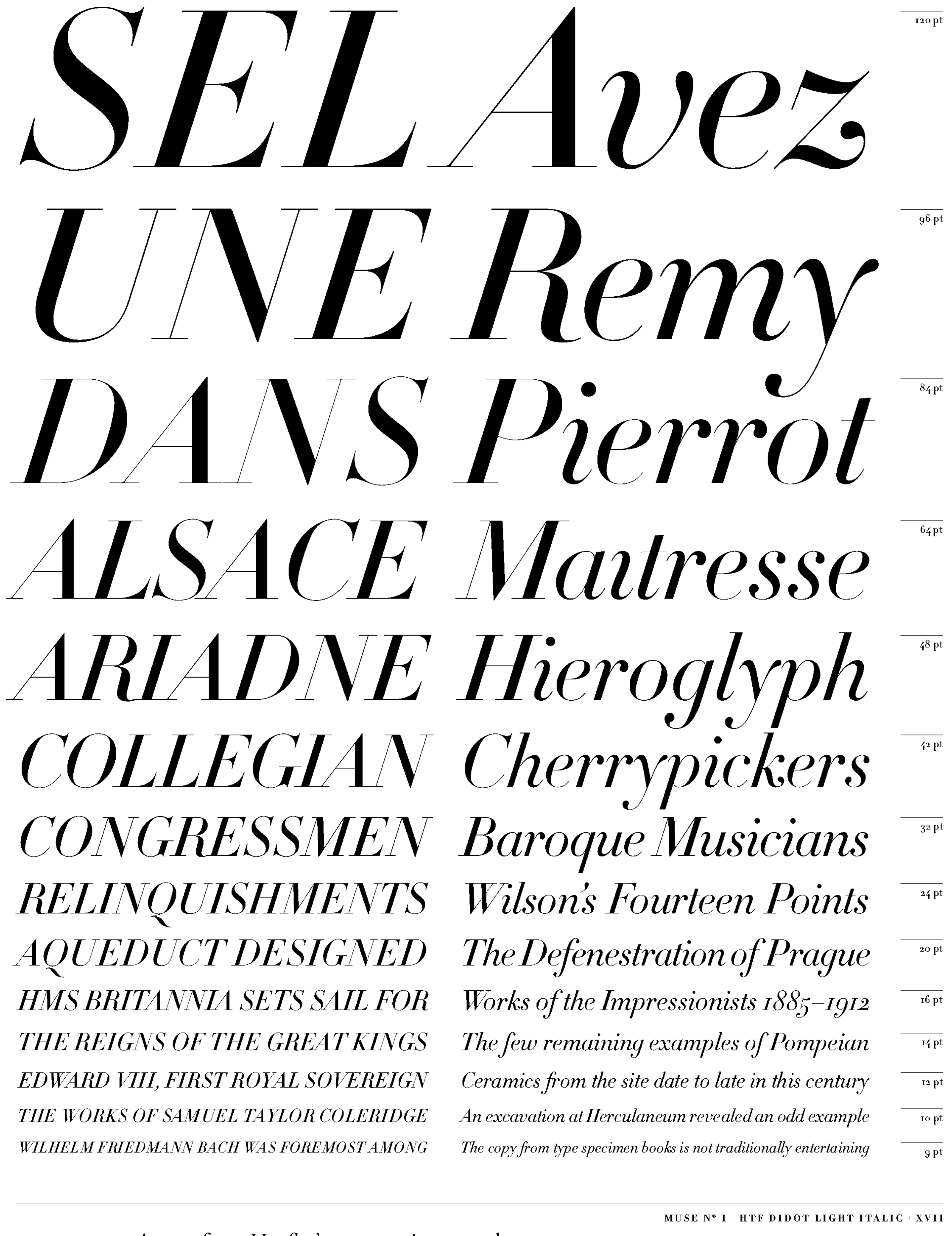
Recall that in the partition problem, we are given a list of treasures, each with a known value, and a target value, which represents the share of the treasures that we are trying to allocate to someone whom we will call the wizard. The question is whether there is any collection of treasures that will add up to the wizard's share exactly, and if so, which treasures?

One runs into this problem and closely related problems everywhere. For example, I once was talking to Jonathan Hoefler, owner of the Hoefler Type Foundry. Hoefler needed to produce type samples for his catalog. For each font, he needed to find an English word or phrase that would fit in a column *exactly* 3.25 inches wide. He had a dictionary, and could compute a table of the length of each word. For large font sizes, this was enough, because a single word such as "Hieroglyph" or "Cherrypickers" at 48- or 42-point size (respectively) would exactly fill the column; solving the problem for large sizes is a simple matter of scanning the table for the single word closest in size to 3.25 inches. But the same column must accommodate fonts of all sizes, from large to small, and there is no word that is 3.25 inches wide when set in 20-point type. Several words have to be put together to add up to the required length. For 20-point type, the example is "The Defenestration of Prague."[1] (See Figure 5.1.)

In regular text, the typesetter will expand the spaces between words slightly to take up extra space when needed, or will press the words more closely together. In ordinary typesetting, this is acceptable. But in a font specimen catalog, the font designer wants everything to look perfect, and the spacing has to be just so. The designer wants to pick text, which, when spaced in the most natural way, happens to fill the column as exactly as possible.

The problem of finding words to fit as perfectly as possible into the space in a font specimen catalog is very similar to the partition problem we saw in Chapter 3. The differences are that some allowance has to be made in the programming to handle appropriate inter-word space that follows every word but the last, that

1 Stay away from the windows if you're ever in Prague; the city is famous for its defenestrations. Probably the most important was on 23 March, 1618, when Bohemian nobles flung two imperial governors out the window into a ditch, touching off the Thirty Years' War. Other notable defenestrations have occurred in 1419 and 1948.


SEL Avez 120 pt

UNE Remy 96 pt

DANS Pierrot 84 pt

ALSACE Maitresse 64 pt

ARIADNE Hieroglyph 48 pt

COLLEGIAN Cherrypickers 42 pt

CONGRESSMEN Baroque Musicians 32 pt

RELINQUISHMENTS Wilson's Fourteen Points 24 pt

AQUEDUCT DESIGNED The Defenestration of Prague 20 pt

HMS BRITANNIA SETS SAIL FOR Works of the Impressionists 1885–1912 16 pt

THE REIGNS OF THE GREAT KINGS The few remaining examples of Pompeian 14 pt

EDWARD VIII, FIRST ROYAL SOVEREIGN Ceramics from the site date to late in this century 12 pt

THE WORKS OF SAMUEL TAYLOR COLERIDGE An excavation at Herculaneum revealed an odd example 10 pt

WILHELM FRIEDMANN BACH WAS FOREMOST AMONG The copy from type specimen books is not traditionally entertaining 9 pt

MUSE N° I HTF DIDOT LIGHT ITALIC · XVII

FIGURE 5.1 A page from Hoefler's type specimen catalog.

words may be re-used, and that it is permissible to miss the target value by a small amount.

Another related problem is how to back up your files from your hard disk onto floppy diskettes, using as few diskettes as possible. It is not permitted to split any file across two or more diskettes. This problem was intensely interesting to me in 1986, because the file backup program for my Macintosh did have just these restrictions, and as a penurious college student, I couldn't afford to buy lots of diskettes.

We've seen the code for a recursive version of this problem already. It looks like this:

```
sub find_share {
  my ($target, $treasures) = @_;
  return [] if $target == 0;
  return    if $target < 0 || @$treasures == 0;
  my ($first, @rest) = @$treasures;
  my $solution = find_share($target-$first, \@rest);
  return [$first, @$solution] if $solution;
  return        find_share($target       , \@rest);
}
```

This function returns an array of treasures that add up to the target sum, if there is such a solution, and `undef` if there is no solution.

5.1.1 Finding All Possible Partitions

We could easily modify it to return *all* possible solutions, instead of only one:

CODE LIBRARY
partition-all

```
sub partition {
  my ($target, $treasures) = @_;
  return [] if $target == 0;
  return () if $target < 0 || @$treasures == 0;

  my ($first, @rest) = @$treasures;
  my @solutions = partition($target-$first, \@rest);
  return ((map {[$first, @$_]} @solutions),
         partition($target, \@rest));
}
```

Why might we want to do such a thing? Suppose we're trying to allocate shares to several people, say a wizard, a barbarian, and a plumber, out of the same pool of treasure. First we allocate the wizard's share. There might be several ways to do this, so we choose one. Next we want to allocate the barbarian's share, but we find that there's no way to do this. It might be that if we had allocated the wizard's share differently, we wouldn't have gotten into trouble over the barbarian's share later. When we find out that we can't allocate the barbarian's share correctly, we want to *backtrack* and try the wizard's share in a different way.

Here's a particularly simple example: Suppose that there are four treasures worth 1, 2, 3, and 4. The wizard is owed treasures worth 5 gold pieces, and the barbarian is owed 3. If we give treasures 2 and 3 to the wizard, we foreclose the only possible solutions for the barbarian. We need to backtrack and try a different distribution of treasures; in this case we should give treasures 1 and 4 to the wizard, and treasure 3 to the barbarian. (The plumber works for union scale and is paid by the hour.)

The preceding partition function delivers all possible shares for the wizard; so if we try `[2,3]` and discover that this causes problems later for the barbarian, we can backtrack and try the other solution, `[1,4]`, instead.

But this function has a serious problem that we might have foreseen: Even simple instances of the partition problem often have many different solutions. For example, the call `partition(105, [1..20])` generates 15,272 solutions. Since we probably won't need to find all these solutions, we would like to convert this function to an iterator.

In Chapter 4, we saw a technique for doing this. It involved replacing the implicit recursion stack with an explicit queue, and appeared to require ingenuity. But it turns out that this technique always works, and doesn't require much ingenuity at all.

This tactic for turning a recursive function into an iterator is to have the iterator retain an agenda[2] or to-do list of partially-complete partition attempts that it has not yet investigated. Each time we invoke the iterator, it will remove an item from the to-do list and investigate it. If the item represents a solution to the problem, the iterator will return it immediately. If the item requires further investigation, the iterator will investigate it a little further, possibly producing some new partially-investigated items, which it will put onto the to-do list be investigated later, and will continue to look through the agenda for solutions. If the agenda is exhausted before a solution is found, the iterator will report failure. Since the agenda is part of the iterator's state, the iterator can return a solution to its caller, and the agenda state will remain intact until the next time the iterator is called.

We saw several examples of this approach, including the web spider, in Chapter 4.

2 *Agenda* is the Latin word for "to-do list."

For this problem, each item in the queue must contain the following information:

- A current target sum
- The *pool* of treasures still available for use
- The *share* containing the treasures already allocated toward the target

In general, with this technique, each agenda item must contain all the information that would have been passed as arguments to the recursive version of the function.

CODE LIBRARY
partition-it

```
sub make_partitioner {
  my ($n, $treasures) = @_;
  my @todo = [$n, $treasures, []];
```

Initially, the queue contains only one item that the iterator must investigate: The target sum is `$n`, the number originally supplied by the user; the pool contains all the treasures; the share is empty. The iterator will move treasures from the pool to the share, deducting their values from the target, until the target is zero.

```
  sub {
    while (@todo) {
      my $cur = pop @todo;
      my ($target, $pool, $share) = @$cur;
```

Here the iterator extracts the tail item from the agenda. This is the "current" item that it must investigate. The iterator extracts the target sum, the available pool of treasures, and the list of treasures already allocated to the share. The presence of this item in the to-do list indicates that if some subset of the treasures in `$pool` can be made to add up to `$target`, then those treasures, plus the ones in `$share`, constitute a solution to the original problem.

The iterator can return under two circumstances. If it finds that the current item represents a solution, it will return the solution immediately. But if the agenda is exhausted before this occurs, then there is nothing left to investigate, there are no more solutions, and the iterator will immediately return failure.

```
      if ($target == 0) { return $share }
```

If the target sum is zero, the current share is already a winner. The iterator returns it immediately. Any items that are still uninvestigated remain on the to-do list, awaiting the next call to the iterator.

```
        next if $target < 0 || @$pool == 0;
```

On the other hand, if the target is negative, the current item is hopeless, and the iterator should immediately discard it and investigate another item; similarly if the pool of treasures in the current item has been exhausted. The `next` restarts the `while` loop from the top, which begins by extracting a new current item from the agenda.

With these simple cases out of the way, the bulk of the code follows:

```
        my ($first, @rest) = @$pool;
        push @todo, [$target-$first, \@rest, [@$share, $first]],
                    [$target       , \@rest,   $share        ];
      }
```

In the typical case, the current item has two sub-items that must be investigated separately: Either the first treasure in the pool is included in the share, and the target is smaller, or it isn't included, and the target is the same. For example, to satisfy `(28, [10,18,27], [1])` we can either investigate `(18, [18,27], [1,10])` or we can investigate `(28, [18,27], [1])`.

The iterator appends the two new items to the end of the queue and returns to the top of the `while` loop to investigate another item.

```
      return undef;
    } # end of anonymous iterator function
  } # end of make_partitioner
```

If the to-do list is exhausted, the `while` loop exits, and the iterator returns `undef` to indicate failure.

5.1.2 Optimizations

There are a few obvious ways to improve the preceding code. Suppose the current item is `[12, [12, ...], [...]]`. The function then constructs two new items, `[0, [...], [..., 12]]` and `[12, [...], [...]]`, and pushes them onto the end of the to-do list. But the first item is obviously a solution (because its target sum is 0), so there's no point in putting it on the end of the queue and working through every other item on the queue looking for a different solution; clearly we should return it right away.

Similarly, if the function constructs an item that is obviously useless, it could throw it away immediately rather than putting it on the queue to be thrown away later:

CODE LIBRARY
partition-it-opt

```
sub make_partitioner {
  my ($n, $treasures) = @_;
  my @todo = [$n, $treasures, []];
  sub {
    while (@todo) {
      my $cur = pop @todo;
      my ($target, $pool, $share) = @$cur;
      if ($target == 0) { return $share }
      next if $target < 0 || @$pool == 0;

      my ($first, @rest) = @$pool;

      push @todo, [$target, \@rest, $share ] if @rest;
      if ($target == $first) {
         return [@$share, $first];
      } elsif ($target > $first && @rest) {
         push @todo, [$target-$first, \@rest, [@$share, $first]],
      }
    }
    return undef;
  } # end of anonymous iterator function
} # end of make_partitioner
```

The first new line here appends to the queue what was previously the second new item. But here it's conditionalized: The item is placed on the queue only if its treasure pool will still contain an unused item. If its pool is empty, then it can't possibly result in a solution, so the function discards it immediately.

The following `if-elsif` block handles what was previously the first new item. The function is about to put the first treasure into the share and to subtract its size from the target sum. But unlike the previous version of the code, here it puts the new item on the queue only if the size of the first treasure is smaller than the target sum. If the first treasure is equal to the target sum, then the item it is about to put on the queue is actually a solution to the problem, so the iterator returns it immediately instead of queuing it. Conversely, if the first treasure is larger than the target sum, then the item the iterator was about to queue would have had a negative target sum, and would have been discarded the next time it was encountered; instead, the iterator never puts it in the queue at all. The `&& @rest` condition makes sure the iterator doesn't queue an item with a positive target sum and an empty pool, which is guaranteed to fail.

It's tempting to remove the:

```
if ($target == 0) { return $share }
next if $target < 0 || @$pool == 0;
```

lines now. They're much less useful, since the cases they check for are all detected at the bottom of the loop, and items that have `$target <= 0` or `@$pool == 0` aren't put into the queue to begin with. The only cases they do catch are when such items are placed directly into the queue by the caller of `make_partitioner`.

There are at least three ways we can deal with this. We can leave the checks in place. We can remove the checks and document the resulting deficiency in the function: If the initial value of `$n` is 0, the iterator fails to report the empty solution. (Even with the extra checks, the function has a few boundary condition errors of this type. For example, it reports only three of the eight possible solutions to `make_partitioner(0, [0,0,0])`.) Or we can remove the checks and add preprocessing code that works around the bug. For example:

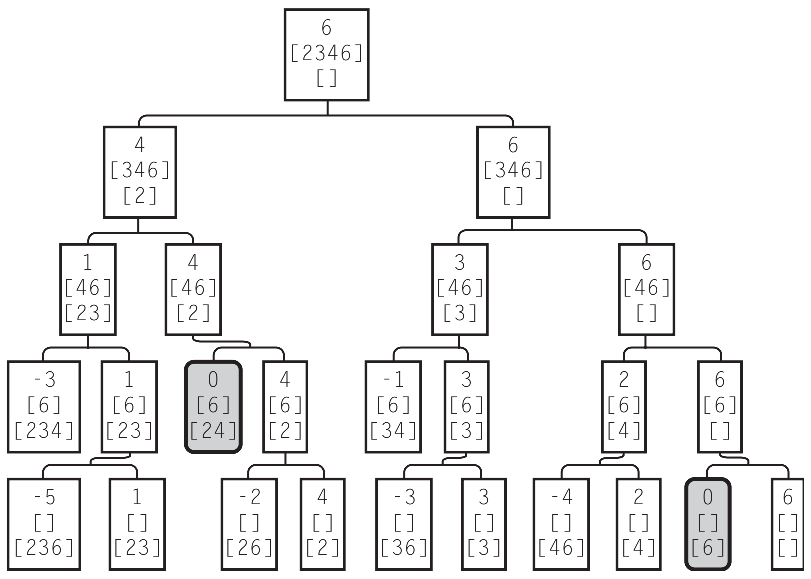
```
sub make_partitioner {
  my ($n, $treasures) = @_;
  my @todo = $n ? [$n, $treasures, []] : [$n, [], []];
  sub {
    ...
  }
}
```

If `make_partitioner` sees that we're about to exercise the bug, which occurs only for $n = 0$ and a nonempty treasure pool, it silently adjusts the pool behind the scenes to a case that *will* produce the correct answer.

These three tactics are presented in increasing order of "cleverness." Such cleverness should be used only when necessary, since it requires a corresponding application of cleverness on the part of the maintenance programmer eight weeks later, and such cleverness may not be available.

5.1.3 Variations

The space searched by this function is organized like a tree:



Each node of this tree represents one of the items that the partitioner investigates, showing the target sum, the pool, and the share so far. For example, the root node represents an item with a target sum of 6, a pool containing 2, 3, 4, and 6, and an empty share. The root node is the item that the user of `make_partitioner` first inserted into the to-do list. Each node has two child nodes, which are the two derived items: one moves the first treasure from pool to share and subtracts it from the target sum, and the other removes the first treasure from the pool and discards it without changing the share or the target sum. The leaf nodes are those from which no further searching is done, because the pool is empty (bottom row) or the target sum is too small.

The partitioner always searches a node before searching its children, so it searches the tree in a generally top-to-bottom order. In fact, the version we saw first searches the nodes in depth-first order, visiting the root node, then the nodes down the leftmost branch, then the three nodes just to the right of the leftmost branch, and so on.

The second version of the partitioner saves time by refusing to investigate items that it sees will be leaves, effectively searching the smaller tree of Figure 5.2 instead.

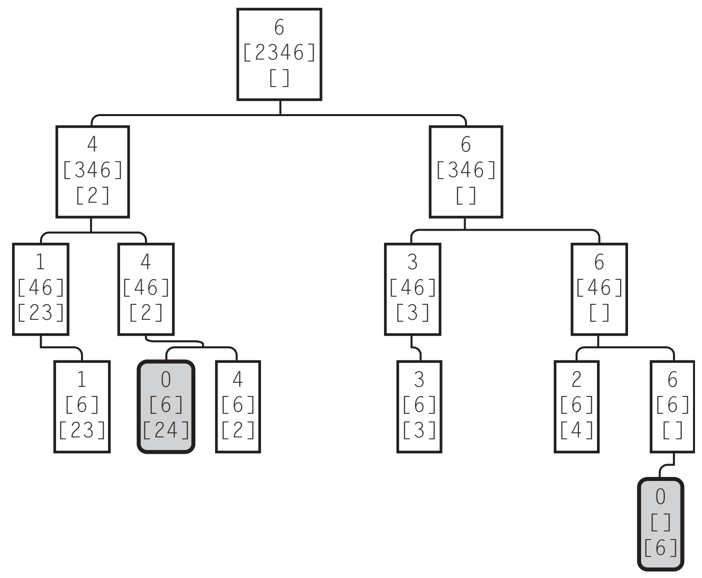


FIGURE 5.2 The search space of `partition(6, [2,3,4,6])`, pruned.

Whether to choose breadth- or depth-first search depends on the nature of the problem. Each has major contraindications. *Depth-first search (DFS)* tends to yield shorter to-do lists. In any depth-first search of a tree, if each node in the tree has no more than n children, and the depth of the tree is d nodes, then the to-do list will contain at most $(n-1)(d-1)+1$ items at any time. For the partition problem, n is 2, and d is no more than the number of items in the original pool. So in depth-first search, the to-do list will never exceed the size of the original pool.

In contrast, *breadth-first search (BFS)* can sometimes lead to enormous to-do lists. The tree is searched top-down, and if all the solutions are in the leaves, each interior tree node must be put on the to-do list and taken off again before the search reaches the leaves where the solutions are. In the unpruned partition search example, shown on page 212, breadth-first search starts with the root node on the agenda, then removes it and replaces it with the two second-level nodes, then removes these and replaces them with the four third-level nodes, then replaces these with the eight fourth-level nodes. These are eventually replaced with the ten fifth-level nodes; if the problem had been bigger, there would have been sixteen fifth-level nodes instead of only ten. Breadth-first search may be contraindicated when the tree branches rapidly or when the solutions are all to be found among the leaves. Depth-first search, which dives straight down to where the solutions are, may be a better choice.

For some applications, however, depth-first search is a loser. Web spidering is one of these. I was once teaching a class in which one of the students decided to write a web spider. The central control of his program was a recursive function, something like this:

```
sub handle_page {
  my $url = shift;
  get the document from the network;
  if (the document is HTML) {
    parse it;
    extract the links;
    for (links) {
      handle_page($_);
    }
  }
}
```

Because the function was recursive, it naturally did a depth-first search on the web space. The result was completely useless. The spider started by reading the initial page and making a list of all the links from that first page. Then it followed the first link on the first page and made a list of all the links on the second page. Then it followed the first link on the second to a third page and made a list of all the links on that page, and so on. The spider went dashing off toward the horizon, never to return, except perhaps by accident. Clearly this wasn't particularly useful. This is the major contraindication for depth-first search: a very large, or infinite search space.

To see a particularly simple example of this, consider a search for strings of the letters `A`, `B`, and `C` that read the same forwards as backwards (see Figure 5.3). We might imagine a search of the space of all strings.

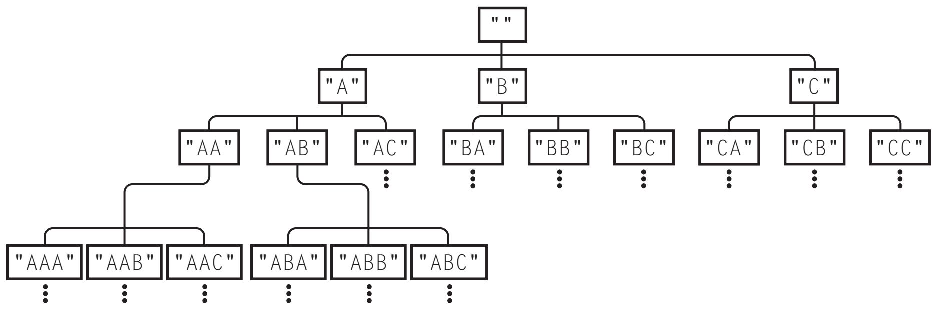


FIGURE 5.3 Searching for palindromes in the space of all strings.

Breadth-first search eventually finds all the desired strings, in order by length: `""`, `"A"`, `"B"`, `"C"`, `"AA"`, `"BB"`, `"CC"`, `"AAA"`, `"ABA"`, `"ACA"`, `"BAB"`, `....`

Depth-first search, however, goes diving down the leftmost branch, finding `""`, `"A"`, `"AA"`, `"AAA"`, `"AAAA"`... and never even looking at any branches that contain `B`s or `C`s.

5.2 HOW TO CONVERT A RECURSIVE FUNCTION TO AN ITERATOR

We've seen several such techniques, including the odometer method and the agenda method. It appears that these took some ingenuity to find. What if they don't happen to work for a particular function, and you don't have enough ingenuity that day to find something that does work?

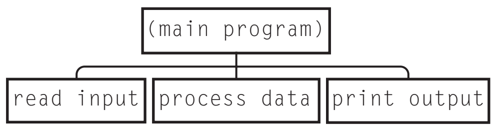
It turns out that that won't happen, because the agenda method *always* works. This is because we can consider every recursive function to be doing a tree search!

Ordinary function call semantics create a notional tree of function calls. Imagine that we have a node for each time a function is called, and node *A* is the parent node of *B* when the function invocation represented by *A* is responsible for invoking the function represented by *B*. The root node is the main program, which is started by some agency outside of the program itself. A simple program like this:

```
#!/usr/bin/perl

$data = read_the_input();
$result = process_the_data($data);
print_the_output($result);
```

evolves the simple tree depicted below. Such a tree is called a *call tree*:



It's important to realize that the call tree has one tree node not for each subroutine, but for each *invocation* of each subroutine (see Figure 5.4).

```
sub read_input {
  for (1..8) {
```

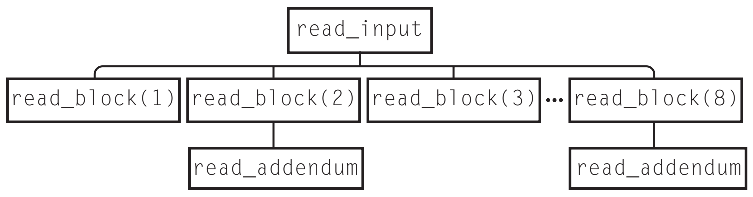


FIGURE 5.4 A more complicated call tree.

```
    read_block($_);
  }
  ...
}

sub read_block {
  my $n = shift;
  if ($n %  2 == 0) { read_addendum() }
  ...
}

sub read_addendum { ... }
```

In the call tree for a recursive function, the node for a subroutine may have children that represent calls to the same subroutine. For a recursive directory tree walker like `walk_tree`, the call tree is exactly the same as the directory tree itself. Figure 5.5 shows a more arbitrary example.

```
sub rec {
  my ($n, $k) = @_;
  print $k x $n, "\n";
  for (1 .. $n-1) {
    rec($n-$_, $_);
  }
}
```

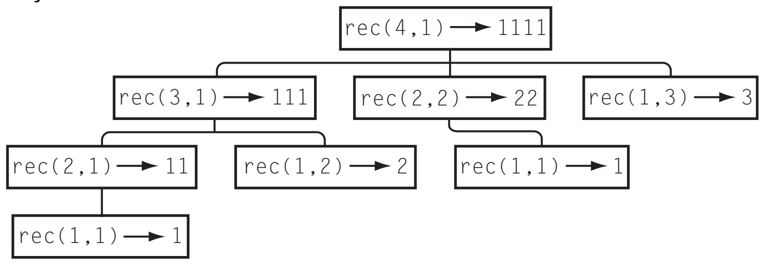


FIGURE 5.5 A call tree for a recursive function.

When a recursive function runs, we can imagine that it is performing a depth-first tree search on its own call tree. It starts at the root, which represents the initial invocation of the function. Each time the function calls itself, it is moving down the tree to a child node; when the call returns, it moves back up the parent. When run, the preceding code example does indeed produce the data from the tree nodes of Figure 5.5 in depth-first order:

```
1111
111
11
1
2
22
1
3
```

As a result, every recursive function is really doing a depth-first tree search. Whenever we want to convert a recursive function to an iterator, we can use the agenda method. Each agenda item will represent one call to the recursive function and will contain all the state information that the recursive function needed to do its work: in general, all its private variables, but often, just the arguments. When the iterator removes an item from the agenda, it starts pretending that it's the recursive function, with the arguments described by the item it removed. If the recursive function would have called itself recursively, the iterator puts an item onto the agenda to represent the new arguments.

Let's look at a new example to see how this works. Some time ago, a friend, Jeff Goff, was working on a game and asked how to write a function that would take a positive integer n and produce a list of all the different ways it could be split into smaller integers. For example, if $n = 6$, the desired list is:

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

Rather confusingly, this is called the *partitions of an integer problem*, and each of the rows in the table is a *partition* of the number 6.

First we have to suppose we have a recursive function that solves this problem. The function will take a number and split a chunk off it. For example, it might split 5 into 4 + 1 or 6 into 3 + 3. It will do this in every possible way. Then it will recurse, and split another chunk off the remainder, and so on:

CODE LIBRARY
partition-repeats

```
sub partition {
   print "@_\n";
   my ($n, @parts) = @_;
   for (1 .. $n-1) {
     partition($n-$_, $_, @parts);
   }
 }
```

This isn't quite what we want, because it generates some of the partitions more than once. For example, if we start with 6, and split off 2 and then 3, we get 1 + 3 + 2; if we split off 3 first and then 2, we get 1 + 2 + 3, which is the same. The preceding function generates 32 partitions of 6, including 3 + 1 + 1 + 1, 1 + 3 + 1 + 1, 1 + 1 + 3 + 1, and 1 + 1 + 1 + 3, but there are only 11 different partitions.

The trick for eliminating extra items in a listing like this is to adopt a *canonical form* for the output. Where there are several items that are essentially the same, a canonical form is just a convention about which item you'll choose to represent all of them.

This idea should be familiar. Suppose we wanted to read a list of words, and report on the ones that appeared more than once. Easy; just use a hash:

```
for (@words) { $seen{$_}++ }
@repeats = grep $seen{$_} > 1, keys %seen;
```

But what if the words are in mixed-case, and the case doesn't matter, so that we want to consider "perl", "Perl", and "PERL" as being the same? There's only one easy way to do it: Use a hash, and store the all-lowercase version of the codes:

```
for (@words) { $seen{lc $_}++ }
@repeats = grep $seen{$_} > 1, keys %seen;
```

The all-lowercase version is the canonical form for the words. Words are divided into groups of equivalent words, sometimes called *equivalence classes*, and a representative is chosen from each group. For the group of equivalent words containing:

perl	Perl	pErl	peRl
perL	PErl	PeRl	PerL
pERl	pErL	peRL	PERl
PErL	PeRL	pERL	PERL

we choose "perl" as the canonical representative. Choosing the all-uppercase member of each group would work as well, of course, as would any other method that chooses exactly one representative from every equivalence class. Another familiar example is numerals: We might consider the numerals "0032.50," "32.5," and "325e-01" to be equivalent; when Perl converts these strings to an internal floating-point format, it is converting them to a canonical representation so that equivalent numerals have the same representation.

Returning to our problem of duplicate partitions, it appears that one solution will be to find a canonical form for partitions, and then discard any partitions that aren't already in canonical form. Sometimes it can be difficult to find an appropriate canonical form. But not in the case of the partition problem. The partitions are lists of numbers, and since every list has one and only one sorted version, we'll just say that the sorted version of the list is its canonical form.

We will produce partitions whose elements are in decreasing order, and no others. (We'll say "decreasing" when what we really mean is "nonincreasing," so we say that 5, 5, 4, 3, 3 is a "decreasing" sequence of numbers. This is more convenient than using the clumsy word "nonincreasing" everywhere.[3])

We could refit our subroutine to suppress the printing for the elements that aren't in decreasing order:

```
sub partition {
  print "@_\n" if decreasing_order(@_);
  my ($n, @parts) = @_;
  for (1 .. $n-1) {
    partition($n-$_, $_, @parts);
  }
}
```

However, it's more efficient to avoid generating noncanonical partitions in the first place. To generate only those partitions whose members are in decreasing

3 If anyone complains about this abuse of terminology, I will just point out that Edsger Dijkstra, a computer scientist famous for precision, did the same thing. See page 3 of *An Introductory Example*, http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1063.PDF.

order, we just have to take care not to split off any parts that are smaller than a part we have already split off:

CODE LIBRARY
partition

```
sub partition {
  print "@_\n";
  my ($largest, @rest) = @_;
  my $min = $rest[0] || 1;
  my $max = int($largest/2);
  for ($min .. $max) {
    partition($largest-$_, $_, @rest);
  }
}
```

Here instead of splitting off parts with any size at all between 1 and `$n-1`, we put conditions on the size of the parts we can split off. We know that the arguments to the function are in decreasing order, so that the first argument is the largest part, the next is the next largest (if it exists), and the rest (if there are any) are no bigger than these two. We don't want to split off a part that is smaller than one we split off before, so it is sufficient to make sure the split-off part is at least as large as `$rest[0]`, if it exists; if not, we haven't split anything off yet, so it's okay to split off any amount down to and including 1.

The split-off value must not be larger than half the largest element, or else the part left over after it is subtracted will be smaller than the part that was split off: we would go from `partition(5,2)` to `partition(2,3,2)`, and then the arguments wouldn't be in decreasing order.

Figure 5.6 shows the call tree for the invocation `partition(7)`.

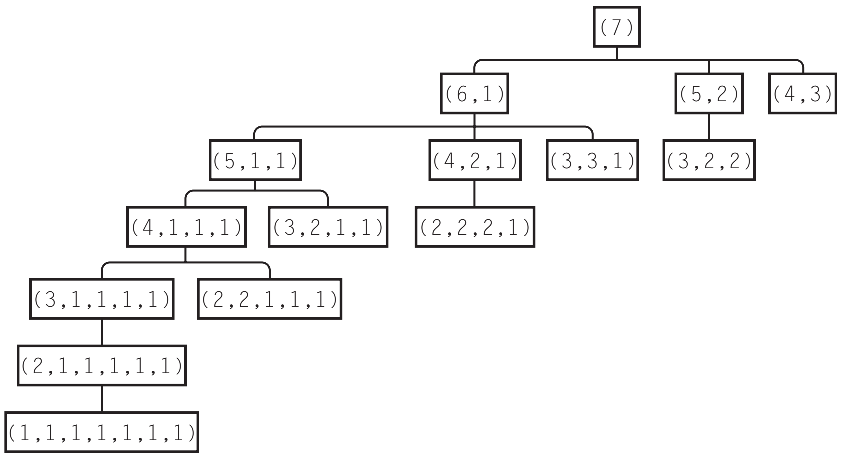


FIGURE 5.6 Partitions of the integer 7, organized as a search space.

The large left branch contains all the partitions that include a part of size 1. The much smaller second branch contains just the partitions whose parts are all at least 2. The third branch contains the single partition, `(4, 3)`, whose parts are all at least 3.

Incidentally, it's quite easy to change the function to solve the slightly different problem of producing the partitions where the parts are all different: Just change `$rest[0]` to `$rest[0]+1` and `$largest` to `($largest-1)`.

The function works just fine, producing each partition exactly once, and every partition in decreasing order, so now we'll try to turn it into an iterator.

To do this, we need to identify the state that the function tracks during each invocation. We'll then package up each state into an agenda item. In general, the state might include all of the function's lexical variables, and it has four: `@rest`, `$largest`, `$min`, and `$max`:

CODE LIBRARY
partition-it-2

```
sub make_partition {
  my $n = shift;
  my @agenda = ([$n,           # $largest
                 [],           # \@rest
                 1,            # $min
                 int($n/2),    # $max
                ]);
  return Iterator {
    while (@agenda) {
      my $item = pop @agenda;
      my ($largest, $rest, $min, $max) = @$item;
      for ($min .. $max) {
        push @agenda, [$largest - $_,          # $largest
                       [$_, @$rest],           # \@rest
                       $_,                     # $min
                       int(($largest - $_)/2), # $max
                      ];
      }
      return [$largest, @$rest];
    }
    return;
  };
}
```

The code here has a strong resemblance to the original recursive function. We can see the `int($largest/2)` and the `for ($min .. $max)` loop lurking inside. But it's rather clumsy. The iterator we've just constructed is more closely analogous to a different version of the recursive function, one that passes all four quantities as arguments:

```
sub partition {
  my ($largest, $rest, $min, $max) = @_;
  for ($min .. $max) {
    partition($largest-$_, [$_, @$rest], $_, int(($largest - $_)/2));
  }
  return [$largest, @$rest];
}
```

This does work, but it's not how we did it originally. Instead, we derived `$min` and `$max` from `$largest` and `$rest`, and these in turn were derived from `@_`, which is the true state of the recursive function. Realizing this leads us to a simpler iterator:

CODE LIBRARY
make-partition-1

```
sub make_partition {
  my $n = shift;
  my @agenda = [$n];
  return Iterator {
    while (@agenda) {
      my $item = pop @agenda;
      my ($largest, @rest) = @$item;
      my $min = $rest[0] || 1;
      my $max  = int($largest/2);
      for ($min .. $max) {
        push @agenda, [$largest-$_, $_, @rest];
      }
      return $item;
    }
    return;
  };
}
```

The code here is quite similar to that of the original function.

Now that we have an iterator, we can play around with it. There's no point to the `while` loop, because it executes at most once, and a `while` loop that executes at most once is just an `if` in disguise:

CODE LIBRARY
make-partition-2

```
sub make_partition {
  my $n = shift;
  my @agenda = [$n];
  return Iterator {
    return unless @agenda;
```

```
      my $item = pop @agenda;
      my ($largest, @rest) = @$item;
      my $min = $rest[0] || 1;
      my $max  = int($largest/2);
      for ($min .. $max) {
        push @agenda, [$largest-$_, $_, @rest];
      }
      return $item;
    };
  }
```

Because we return each partition immediately, after putting its children onto the agenda, old nodes are never preempted by new ones, regardless of whether we use `pop` or `shift`. Consequently this iterator always produces partitions in breadth-first order. The output lists the partitions in increasing order of number of elements:

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

We might prefer it to return the partitions in a different order, say one listing all the partitions with large parts before those with small parts:

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

This is equivalent to sorting the partitions. And we can get this order by sorting the agenda before we process it. To do that, we'll need a comparison function for partitions:

CODE LIBRARY
partitions

```
# Compare two partitions for preferred order
sub partitions {
  for my $i (0 .. $#$a) {
    my $cmp = $b->[$i] <=> $a->[$i];
    return $cmp if $cmp;
  }
}
```

To compare two partitions, we just scan through them both one element at a time until we find a difference; when we do, that's the answer. Since two partitions must have a difference somewhere before the end of either, we don't have to worry what happens if we fall off the end.[4] Now we make a small change to the iterator:

CODE LIBRARY
make-part-sorted

```
sub make_partition {
  my $n = shift;
  my @agenda = [$n];
  return Iterator {
    return unless @agenda;
    my $item = pop @agenda;
    my ($largest, @rest) = @$item;
    my $min = $rest[0] || 1;
    my $max  = int($largest/2);
    for ($min .. $max) {
      push @agenda, [$largest-$_, $_, @rest];
    }
    @agenda = sort partitions @agenda;
    return $item;
  };
}
```

We sort the agenda into the order we want before extracting items from it. Rather than sorting the entire array so that the item we want is at the end, a computationally cheaper approach is to scan the agenda looking for the maximal element and

4 With ordinary lexical sorting, we have to worry about cases where one value is a prefix of another, such as "`fan`" and "`fandango`". In such a case, we *do* fall off the end. But that can't happen with partitions, because two such sequences of positive numbers can't possibly add up to the same thing.

then to `splice` it out once we find it. If we plan to do a lot of heuristically-guided searches, we should invest in building a priority-queue structure for the agenda. A priority queue contains a collection of items, each with an associated priority; it efficiently supports the operations of adding a new item to the collection, and of extracting and removing the item with the largest priority.

5.3 A GENERIC SEARCH ITERATOR

You've probably noticed by now that all these agenda-type iterators look more or less the same. We can abstract out the sameness and make a generic tree-search iterator. To do that, we need to describe the tree. The constructor function will receive two arguments: the root node, and a callback function, which, given a node, generates its children in the tree. It will then carry out a tree search, returning the tree nodes one at a time:

CODE LIBRARY
`make-dfs-simple`

```
use Iterator_Utils 'Iterator';

sub make_dfs_search {
  my ($root, $children) = @_;
  my @agenda = $root;
  return Iterator {
    return unless @agenda;
    my $node = pop @agenda;
    push @agenda, $children->($node);
    return $node;
  };
}
```

With this formulation, `make_partition` becomes:

CODE LIBRARY
`make-part-dfs-1`

```
sub make_partition {
  my $n = shift;
  my $root = [$n];
  my $children = sub {
    my ($largest, @rest) = @{shift()};
    my $min = $rest[0] || 1;
    my $max  = int($largest/2);
    map [$largest-$_, $_, @rest], ($min .. $max);
  };
  make_dfs_search($root, $children);
}
```

Factoring make_partition into two parts in this way allows us to re-use the make_dfs_search part.

We might outfit make_dfs_search with a filter that rejects uninteresting items, since this is sure to be a common usage:

CODE LIBRARY
make-dfs

```
use Iterator_Utils 'Iterator';

sub make_dfs_search {
   my ($root, $children, $is_interesting) = @_;
   my @agenda = $root;
   return Iterator {
     while (@agenda) {
       my $node = pop @agenda;
       push @agenda, $children->($node);
       return $node if !$is_interesting || $is_interesting->($node);
     }
     return;
  };
}

1;
```

We don't need this for make_partition, since every node represents a correct partition. But we might have needed it if we had used a slightly clumsier implementation of the search:

CODE LIBRARY
make-part-dfs-2

```
require 'make-dfs-search';

sub make_partition {
  my $n = shift;
  my $root = [$n, 1, []];
```

Here the nodes will have three parts: $n, the part of the original number that we haven't yet split off to any of the parts of the partition; a minimum part size, initially 1; and a list of the parts we've split off so far, initially empty:

```
  my $children = sub {
    my ($n, $min, $parts) = @{shift()};
    map [$n-$_, $_, [@$parts, $_]], ($min .. $n);
  };
```

For each possible part size $_, from the minimum $min up to the maximum $n, we split off a new part of size $_. To do this, we subtract the size from $n, indicating that we now have to apportion a smaller value among the remaining parts; we adjust the minimum value up to the new part size, so that any future parts are at least that big and therefore the parts will be generated in order of increasing size; and we append the new part to the list of parts.

Note that if `$n < $min`, there's no possible solution. An example of such a node will occur when we try to partition the number 6 and we first split off parts of sizes 2 and then 3. Then we're stuck: Only 1 remains, but `2, 3, 1` is forbidden because the parts aren't in increasing order.

```
    my $is_complete = sub {
      my ($n) = @{shift()};
      $n == 0;
    };
```

The partition is complete once we've reduced `$n` to exactly 0.

By default, `make_dfs_search()` returns interesting nodes from the agenda. Here the nodes have extraneous information in them in addition to the partitions themselves. So we'll wrap `make_dfs_search()` in a call to `imap()` that strips out the extra data, returning only the partition itself:

```
    imap { $_->[2] }
      make_dfs_search($root, $children, $is_complete);
  }
```

We could similarly outfit `make_dfs_search()` with a callback to evaluate nodes and allow the most valuable ones to be processed first. If we did, we would want to rename it, because it would no longer be doing DFS. To do this properly requires a good priority-queue implementation, which is outside the scope of this chapter. Here's an inefficient implementation:

CODE LIBRARY
make-dfs-value

```
  sub make_dfs_value_search {
    my ($root, $children, $is_interesting, $evaluate) = @_;
    $evaluate = memoize($evaluate);
    my @agenda = $root;
    return Iterator {
       while (@agenda) {
         my $best_node_so_far = 0;
         my $best_node_value = $evaluate->($agenda[0]);
         for (0 .. $#agenda) {
           my $val = $evaluate->($agenda[$_]);
           next unless $val > $best_node_value;
           $best_node_value = $val;
           $best_node_so_far = $_;
```

```
            }
            my $node = splice @agenda, $best_node_so_far, 1;
            push @agenda, $children->($node);
            return $node if !$is_interesting || $is_interesting->($node);
          }
          return;
        };
      }
```

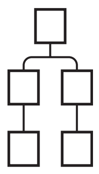
The inefficient part is the scan over the entire agenda and the splice. There are a number of ways to speed this up, but if it matters, the priority queue is probably the best approach.

If we did do this, it would include DFS and BFS as easy special cases, since we could use the following two valuations:

```
      {
         my ($d, $b) = (0, 0);
         sub dfs_value { return $d++ }
         sub bfs_value { return $b-- }
       }
```

bfs_value, like a cantankerous grandfather, always reports the value of an old node as being greater than that of the newer nodes; dfs_value, like the staff at *Wired* magazine, does just the opposite.

One possible trap to be aware of when using make_dfs_search() is that "depth first" doesn't necessarily define the search order uniquely. Consider the tree shown here.



DFS says that once we visit a node, we must visit its children before its siblings. But it doesn't say what order the siblings must be visited in. Both of the orders shown in Figure 5.7 are depth-first for this tree.

Since the nodes generated by the call to $children are pushed onto the end of the agenda and then popped off from the end, the items will be processed in the reverse of the order that $children returned them, with the last item in

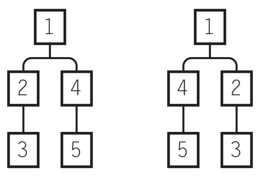


FIGURE 5.7 Two different DFS orders for the same tree.

`$children`'s return list processed immediately. To prevent surprises, we'll make one final change to `make_dfs_search`:

CODE LIBRARY
make-dfs-final

```
sub make_dfs_search {
  my ($root, $children, $is_interesting) = @_;
  my @agenda = $root;
  return Iterator {
    while (@agenda) {
      my $node = pop @agenda;
      push @agenda, reverse $children->($node);
      return $node if !$is_interesting || $is_interesting->($node);
    }
    return;
  };
}
```

Now branches will be traversed in the order they were generated.

5.4 OTHER GENERAL TECHNIQUES FOR ELIMINATING RECURSION

5.4.1 Tail-Call Elimination

In addition to the agenda technique we looked at in detail in the previous section, there are a few other techniques that are generally useful for turning recursive functions into iterative ones. One of the most useful is *tail-call elimination*.

First, let's consider the implementation of function calls generally. Usually there is a stack. When function `B` wants to call `C`, it pushes `C`'s arguments onto this stack and transfers control to `C`. `C` then removes the arguments from the stack, does its computations (possibly including other function calls), pushes its intended return value onto the stack, and transfers control back to `B`. `B` then pops the return value off the stack and continues. If there are three functions, as follows:

```
sub A { A1; $B = B(...); A2; }
sub B { B1; $C = C(...); B2; return $Bval; }
sub C { C1; return $Cval; }
```

then the sequence of events is:

```
A:    A1;
      Push B's arguments
B:    Pop B's arguments
      B1;
      Push C's arguments
C:    Pop C's arguments
      C1;
      Push C's return value
B:    Pop C's return value
      B2;
      Push B's return value
A:    Pop B's return value
      A2;
```

Now let's suppose that function `B` is a little simpler, and doesn't do anything except return after it calls `C`:

```
sub A { A1; $B = B(...); A2; }
sub B { B1; return C(...); }
sub C { C1; return $Cval; }
```

The sequence of events is as before, up to `B2`, which was eliminated; and then goes like this:

```
      ...
C:    Push C's return value
B:    Pop C's return value
      (There is no B2 any more)
      Push B's return value (the same as C's)
A:    Pop B's return value
      A2;
```

All of `B`'s work here is useless. Because `B`'s return value is the same as `C`'s, all `B` is doing is removing `C`'s return value from the stack and then putting it back again immediately. A common optimization in programming-language implementations is to eliminate the return to `B` entirely. The final call to `C` is known as a *tail call*, and the optimization is called *tail-call elimination*. When function B is compiled, the compiler will notice that the call from B to C is a tail call, and will arrange for it to be done in a special way. Normally, B would record its own address so that C would know where to transfer control back to when it was finished. Instead, B erases its own frame from the stack and lets C borrow the return information that B originally got from A. When C returns, it will return directly to A, bypassing B entirely:

```
        ...
C:      Push C's return value
A:      Pop C's return value (thinking it is B's)
        A2;
```

This is the *tail-call optimization*. Perl could in principle perform this optimization, but as of 5.8.6, it doesn't.

Now let's consider the *greatest common divisor function* or *GCD* function. This function takes two numbers, *m* and *n*, and yields the greatest number *g* such that *g* divides evenly into both *m* and *n*. There is always such a number, since 1 divides evenly into both *m* and *n*, although the GCD is often larger than 1. For example, the GCD of 42 and 360 is 6, and the GCD of 48 and 20 is 4. Probably the most well-known application of the GCD is in putting fractions into lowest terms. Given a fraction, say 42/360, one finds the GCD of the numerator and denominator, in this case 6, and then cancels that factor from the top and bottom of the fraction, giving $42/360 = 7 \cdot 6 \,/\, 60 \cdot 6 = 7/60$. Similarly $48/20 = 12 \cdot 4 \,/\, 5 \cdot 4 = 12/5$.

There is a simple algorithm for calculating the GCD of two numbers, called *Euclid's algorithm*, which is in fact the oldest surviving nontrivial numeric algorithm. Here it is translated into Perl:

CODE LIBRARY
gcd

```
sub gcd {
  my ($m, $n) = @_;
  if ($n == 0) {
    return $m;
  }
  return gcd($n, $m % $n);
}
```

The execution of `gcd(48, 20)` goes like this:

```
call gcd(48, 20)            # Call A
  call gcd(20, 8)           # Call B
    call gcd(8, 4)          # Call C
      call gcd(4, 0)        # Call D
```

```
          return 4
        return 4
      return 4
    return 4
```

The stack manipulations are as follows:

original caller:	push 48, 20 onto stack
	transfer control to `gcd`
`A:`	pop 48, 20 from stack
...	
`C:`	push 4, 0 onto stack
	transfer control to `gcd`
`D:`	pop 4, 0 from stack
	push 4 onto stack
	transfer control to `gcd`
`C:`	pop 4 from stack
	push 4 onto stack
	transfer control to `gcd`
`B:`	pop 4 from stack
	push 4 onto stack
	transfer control to `gcd`
`A:`	pop 4 from stack
	push 4 onto stack
	transfer control back to original caller

The tail-call optimization allows call D to return the 4 directly back to the original caller, skipping all the steps at the end.

Since Perl doesn't perform the tail-call optimization automatically, we can help it out. The tail-call optimization would normally replace the current call frame with the one for the function being called. Perl won't do that internally, but since the call frame has nothing in it except a bunch of variable bindings, we can accomplish the same thing by just rebinding the variables to the appropriate variables. "Transfer control to `gcd`," which normally means "create a new call frame and activate it" just becomes "transfer control back the top of the current function" — in other words, a local `goto`. Since `goto` itself is considered naughty, we'll use a loop, which is the same thing:

CODE LIBRARY
gcd2

```
sub gcd {
  my ($m, $n) = @_;
  until ($n == 0) {
    ($m, $n) = ($n, $m % $n);
  }
  return $m;
}
```

The condition for performing the `until` loop is the same as the one guarding the recursive call in the old code. The original function made a recursive call unless `$n` was zero; here it performs the loop body. The body of the loop transforms the arguments `$m` and `$n` in the same way that the recursive code in the original function did, replacing `$m` with `$n` and `$n` with `$m % $n`. Thus the `until` loop sets up the new values of `$m` and `$n` that would have been seen by the recursively-called instance of `gcd`, and then effectively restarts the function. In the case `$n == 0`, there is no recursively-called instance, so the function skips that step and returns immediately.

Here's another example: printing the elements of a sorted binary tree in order. The recursive code looks like this:

```
sub print_tree {
  my $t = shift;
  return unless $t;  # Null tree
  print_tree($t->left);
  print $t->root, "\n";
  print_tree($t->right);
}
```

Replacing the tail call with a loop yields this version:

```
sub print_tree {
  my $t = shift;
  while ($t) {
    print_tree($t->left);
    print $t->root, "\n";
    $t = $t->right;
  }
}
```

Here we've replaced the tail call, `print_tree($t->right)`, with code that modifies `$t` appropriately, replacing it with `$t->right`, and then jumps back up to the top of the function. Since `print_tree($t->left)` isn't a tail call, we can't eliminate it in this way. We'll eliminate it in a different way later on.

A variation of `print_tree()` handles the empty-tree case before the recursive calls, instead of afterwards, potentially optimizing away many such calls:

```
sub print_tree {
  my $t = shift;
  print_tree($t->left) if $t->left;
  print $t->root, "\n";
  print_tree($t->right) if $t->right;
}
```

Eliminating the tail call yields:

```
sub print_tree {
  my $t = shift;
  do {
    print_tree($t->left) if $t->left;
    print $t->root, "\n";
    $t = $t->right;
  } while $t;
}
```

SOMEONE ELSE'S PROBLEM

Here's a particularly interesting example, taken from *Mastering Algorithms with Perl*[5]. Given a set of key–value pairs (represented as a hash, of course), it returns the *power set* of that set. This is the set of all hashes that can be obtained from the original hash by deleting zero or more of the pairs.

For example, the power set of `{apple => 'red', banana => 'yellow', grape => 'purple'}` is:

```
{apple => 'red', banana => 'yellow', grape => 'purple'}
{apple => 'red', banana => 'yellow'}
{apple => 'red',                     grape => 'purple'}
{apple => 'red'}
                {banana => 'yellow', grape => 'purple'}
                {banana => 'yellow'}
                                    {grape => 'purple'}
{}
```

5 This example is taken from Orwant, Hietaniemi, and Macdonald, *Mastering Algorithms with Perl*, pp. 237–238. O'Reilly and Associates, 1999.

The power set is returned as a hash of hashes. The keys of the return value are unimportant, and the values are the elements of the power set. Here's the code that Hietaniemi presents:

CODE LIBRARY
powerset-0

```
sub powerset_recurse ($;@) {
    my ( $set, $powerset, $keys, $values, $n, $i ) = @_;

    if ( @_ == 1 ) { # Initialize.
        my $null   = { };
        $powerset  = { $null, $null };
        $keys      = [ keys   %{ $set } ];
        $values    = [ values %{ $set } ];
        $nmembers  = keys %{ $set };   # This many rounds.
        $i         = 0;                 # The current round.
    }

    # Ready?
    return $powerset if $i == $nmembers;

    # Remap.

    my @powerkeys   = keys   %{ $powerset };
    my @powervalues = values %{ $powerset };
    my $powern      = @powerkeys;
    my $j;

    for ( $j = 0; $j < $powern; $j++ ) {
        my %subset = ( );

        # Copy the old set to the subset.
        @subset{keys   %{ $powerset->{ $powerkeys  [ $j ] } }} =
                values %{ $powerset->{ $powervalues[ $j ] } };

        # Add the new member to the subset.
        $subset{$keys->[ $i ]} = $values->[ $i ];

        # Add the new subset to the powerset.
        $powerset->{ \%subset } = \%subset;
    }

    # Recurse.
    powerset_recurse( $set, $powerset, $keys, $values, $nmembers, $i+1 );
}
```

Clearly, the recursive call here is a tail call. Applying the usual tail-call optimization, we can replace the recursive call with a loop. The special case initialization for the last five parameters no longer needs to be a special case; we just take care of the initialization before we enter the loop. The peculiar `($;@)` prototype goes away entirely, or maybe becomes `($)`:

CODE LIBRARY
powerset-1

```
sub powerset_recurse ($) {
  my ( $set ) = @_;
  my $null = { };
  my $powerset  = { $null, $null };
  my $keys      = [ keys   %{ $set } ];
  my $values    = [ values %{ $set } ];
  my $nmembers  = keys %{ $set };    # This many rounds.
  my $i         = 0;                 # The current round.

  until ($i == $nmembers) {

    # Remap.
    my @powerkeys   = keys   %{ $powerset };
    my @powervalues = values %{ $powerset };
    my $powern      = @powerkeys;
    my $j;

    for ( $j = 0; $j < $powern; $j++ ) {
        my %subset = ( );

        # Copy the old set to the subset.
        @subset{keys   %{ $powerset->{ $powerkeys  [ $j ] } }} =
                values %{ $powerset->{ $powervalues[ $j ] } };

        # Add the new member to the subset.
        $subset{$keys->[ $i ]} = $values->[ $i ];

        # Add the new subset to the powerset.
        $powerset->{ \%subset } = \%subset;
    }

     $i++;
    }

    return $powerset;
}
```

Now we can see that `$i`, the loop counter variable, just runs from 0 up to `$nmembers-1`, so we can rewrite the `while` loop as a `for` loop:

CODE LIBRARY
powerset-2

```
sub powerset_recurse ($) {
    my ( $set ) = @_;
    my $null = { };
    my $powerset  = { $null, $null };
    my $keys      = [ keys   %{ $set } ];
    my $values    = [ values %{ $set } ];
    my $nmembers  = keys %{ $set };    # This many rounds.

    for my $i (0 .. $nmembers-1) {

      #  Remap.
      my @powerkeys   = keys   %{ $powerset };
      my @powervalues = values %{ $powerset };
      my $powern      = @powerkeys;
      my $j;

      for ( $j = 0; $j < $powern; $j++ ) {
          my %subset = ( );

          # Copy the old set to the subset.
          @subset{keys   %{ $powerset->{ $powerkeys  [ $j ] } }} =
                  values %{ $powerset->{ $powervalues[ $j ] } };

          # Add the new member to the subset.
          $subset{$keys->[ $i ]} = $values->[ $i ];

          # Add the new subset to the powerset.
          $powerset->{ \%subset } = \%subset;
      }
    }

    return $powerset;
}
```

Now that we've done this, it appears that the only purpose of `$i` is to index `@$keys and @$values`. Since these are precisely the keys and values of `%$set`, we can eliminate all three variables in favor of a simple `while (each %$set)` loop:

CODE LIBRARY
powerset-3

```
sub powerset_recurse ($) {
    my ( $set ) = @_;
    my $null = { };
    my $powerset  = { $null, $null };

    while (my ($key, $value) = each %$set) {

      # Remap.

      my @powerkeys   = keys   %{ $powerset };
      my @powervalues = values %{ $powerset };
      my $powern      = @powerkeys;
      my $j;

      for ( $j = 0; $j < $powern; $j++ ) {
          my %subset = ( );

          # Copy the old set to the subset.
          @subset{keys   %{ $powerset->{ $powerkeys  [ $j ] } }} =
                  values %{ $powerset->{ $powervalues[ $j ] } };

          # Add the new member to the subset.
          $subset{$key} = $value;

          # Add the new subset to the powerset.
          $powerset->{ \%subset } = \%subset;
      }
    }

    return $powerset;
}
```

If we're feeling sharp, we might notice the same thing about `$j`:

CODE LIBRARY
powerset-4

```
sub powerset_recurse ($) {
    my ( $set ) = @_;
    my $null = { };
    my $powerset  = { $null, $null };

    while (my ($key, $value) = each %$set) {

       my @newitems;

       while (my ($powerkey, $powervalue) = each %$powerset) {
          my %subset = ( );

          # Copy the old set to the subset.
          @subset{keys   %{ $powerset->{$powerkey} } } =
                  values %{ $powerset->{$powervalue} };

          # Add the new member to the subset.
          $subset{$key} = $value;

          # Prepare to add the new subset to the powerset.
          push @newitems, \%subset;
       }

       $powerset->{ $_ } = $_ for @newitems;

    }

    return $powerset;
}
```

Getting rid of the unnecessary recursion made the state changes of the variables clearer and kicked off a series of simplifications that left the function with about one-third less code.

5.4.2 Creating Tail Calls

Often, a function that doesn't have a tail call can be easily converted into one that does. For example, consider the decimal-to-binary conversion function of Chapter 1:

```
sub binary {
  my ($n) = @_;
  return $n if $n == 0 || $n == 1;
  my $k = int($n/2);
  my $b = $n % 2;
  my $E = binary($k);
  return $E . $b;
}
```

Here the recursive call isn't in the tail position. The return value from the recursive call isn't returned directly, but rather is concatenated to `$b`.

The general technique for converting such a function to one that does a tail call is to add an auxiliary parameter that records the return value so far. When the other parameters indicate that the recursion is complete, the function returns the return-value parameter. Instead of making a recursive call, waiting for the return value, modifying it, and returning the result, the modified version takes the return value parameter, modifies it appropriately, and passes it along. When we apply this idea to the `binary()` function, we get this:

CODE LIBRARY
binary-1

```
sub binary {
  my ($n, $RETVAL) = @_;
  $RETVAL = "" unless defined $RETVAL;
  my $k = int($n/2);
  my $b = $n % 2;
  $RETVAL = "$b$RETVAL";
  return $RETVAL if $n == 0 || $n == 1;
  binary($k, $RETVAL);
}
```

`$RETVAL` records the bit sequence computed so far; if unspecified, it defaults to the empty string. On each call, the function appends a new bit to this bit string. If `$n` is 0 or 1, that's the base case, and the function just returns the bit string; otherwise, it makes a recursive call with the new value of `$n` and the new bit string.

Applying the tail-call optimization to this version of `binary()` yields:

CODE LIBRARY
binary-2

```
sub binary {
   my ($n, $RETVAL) = @_;
   $RETVAL = "";
   while (1) {
     my $k = int($n/2);
     my $b = $n % 2;
     $RETVAL = "$b$RETVAL";
     return $RETVAL if $n == 0 || $n == 1;
```

```
    $n = $k;
  }
}
```

and then optimizing away the unnecessary `$k`:

CODE LIBRARY
binary-3

```
sub binary {
  my ($n, $RETVAL) = @_;
  $RETVAL = "";
  while (1) {
    my $b = $n % 2;
    $RETVAL = "$b$RETVAL";
    return $RETVAL if $n == 0 || $n == 1;
    $n = int($n/2);
  }
}
```

Adding an extra parameter to the `factorial()` function of Chapter 1 transforms this:

CODE LIBRARY
factorial-0

```
sub factorial {
  my ($n) = @_;
  return 1 if $n == 0;
  return factorial($n-1) * $n;
}
```

into this:

CODE LIBRARY
factorial-1

```
sub factorial {
  my ($n, $product) = @_;
  $product = 1 unless defined $product;
  return $product if $n == 0;
  return factorial($n-1, $n * $product);
}
```

Then we can eliminate the tail call:

CODE LIBRARY
factorial-2

```
sub factorial {
  my ($n) = @_;
  my $product = 1;
  until ($n == 0) {
    $product *= $n;
    $n--;
  }
  return $product;
}
```

5.4.3 Explicit Stacks

When we last saw the `print_tree()` example, it looked like this:

```
sub print_tree {
  my $t = shift;
  do {
    print_tree($t->left) if $t->left;
    print $t->root, "\n";
    $t = $t->right;
  } while $t;
}
```

The original function had two recursive calls, one of which was a tail call, and was eliminated in this version. The other call remains.

To get rid of a recursive call embedded in the middle of a function may require heavy machinery. The heaviest machinery is to explicitly simulate the same stack operations that Perl normally performs implicitly on function call and return. Making a recursive call records the function's current state on the stack, and returning from a call pops the stack. The function's current state, as we saw earlier, may in general include all of its local variables and parameters.

The state of `print_tree()` comprises nothing more than `$t`, the tree argument itself. So our state-saving operation will be simple. We replace the recursive call `print_tree($t->left)` with a stack push:

```
sub print_tree {
   my $t = shift;
   my @STACK;
   do {
     push(@STACK, $t), $t = $t->left if $t->left;
```

and then, in place of the function return, we add a stack pop and a jump back to the line right after the recursive call:

```
RETURN:
     print $t->root, "\n";
```

```
    $t = $t->right;
  } while $t;
  return unless @STACK;
  $t = pop @STACK;
  goto RETURN;
}
```

(Or, if the stack is empty, then the function returns for real instead of popping.)

One objection to this is likely to be that it uses `goto`, which people think is naughty. We can get rid of the `goto` by transforming the code to this:

```
sub print_tree {
  my $t = shift;
  my @STACK;
 RIGHT: {
    push(@STACK, $t), $t = $t->left while $t->left;
    do {
      print $t->root, "\n";
      $t = $t->right;
      redo RIGHT if $t;
      return unless @STACK;
      $t = pop @STACK;
    } while 1;
  }
}
```

This is really the same thing, except we have cosmetically disguised the `goto` as a `do-while` loop, and turned the old `do-while` loop into a `redo`. Loop control statements such as `next`, `last`, and `redo` are no more than `goto`s in disguise, of course, and in fact so are loops.

ELIMINATING RECURSION FROM fib()

Let's apply the same process to the Fibonacci function:

CODE LIBRARY
fib-0

```
sub fib {
  my $n = shift;
  if ($n < 2) { return $n }
  fib($n-2) + fib($n-1);
}
```

There are no tail calls here. The `fib($n-1)` looks like it might be, but it isn't, because it's not the very last thing the function does before it returns; the addition is. So we can't use tail-call elimination. Instead, we'll roll out the heavy guns and manage the stack explicitly.

The state tracked by `fib()` is more complicated than in the `print_tree()` example. The parameter `$n` is clearly part of the state, but there is some additional state that isn't so obvious. Since there are two recursive calls to `fib`, after we return from a recursive call, we have to remember how to pick up where we left off: Were we about to make the second call, or were we about to perform the addition? Moreover, during the second recursive call, the function's state must include the result from the first recursive call.

In difficult cases, the first step in eliminating recursive calls is to make this state explicit. We rewrite `fib()` as follows:

CODE LIBRARY
fib-1

```
sub fib {
  my $n = shift;
  if ($n < 2) {
    return $n;
  } else {
    my $s1 = fib($n-2);
    my $s2 = fib($n-1);
    return $s1 + $s2;
  }
}
```

The second step is to introduce a loop to separate the initialization of the function from the body:

CODE LIBRARY
fib-2

```
sub fib {
  my $n = shift;
  while (1) {
    if ($n < 2) {
      return $n;
    } else {
      my $s1 = fib($n-2);
      my $s2 = fib($n-1);
      return $s1 + $s2;
    }
  }
}
```

Eventually, we'll have a stack that simulates Perl's call stack; the loop we just introduced is simulating Perl itself.

The third step is to break the body into chunks, each of which contains the code from the end of one recursive call to the beginning of the next. Breaks may occur in the middle of a statement. For example, in `my $s1 = fib($n-1)`, the `$n-1` is computed before the call, but the assignment is done after the call, in a separate chunk. Put each chunk in a separate branch of an `if-else` tree:

CODE LIBRARY
fib-3

```
sub fib {
  my $n = shift;
  my ($s1, $s2, $return);
  while (1) {
    if ($n < 2) {
      return $n;
    } else {
      if ($BRANCH == 0) {
        $return = fib($n-2);
      } elsif ($BRANCH == 1) {
        $s1 = $return;
        $return = fib($n-1);
      } elsif ($BRANCH == 2) {
        $s2 = $return;
        $return = $s1 + $s2;
      }
    }
  }
}
```

Because a statement like `$s1 = fib($n-2)` was split across chunks, I've introduced a temporary value, `$return`, to hold the return value from `fib($n-2)` until it can be assigned to `$s1`. I've also moved the declaration of `$s1` and `$s2` up to the top of the function. Our new `fib()` function is effectively simulating the behavior of the old one, and `$s1` and `$s2` represent information about the function's internal state that are normally traced internally by Perl. They are therefore global to the function itself.

Similarly, `$BRANCH` will record where in the function we left off to make a recursive call. This is another thing Perl normally tracks internally. Initially, it's 0, indicating that we want to start at the top of the body. When we simulate a return from a recursive call, it will be 1 or 2, telling us to pick up later on in the body where we left off:

CODE LIBRARY
fib-4

```
sub fib {
  my $n = shift;
  my ($s1, $s2, $return);
  my $BRANCH = 0;
  while (1) {
    if ($n < 2) {
      return $n;
    } else {
      if ($BRANCH == 0) {
        $return = fib($n-2);
      } elsif ($BRANCH == 1) {
        $s1 = $return;
        $return = fib($n-1);
      } elsif ($BRANCH == 2) {
        $s2 = $return;
        $return = $s1 + $s2;
      }
    }
  }
}
```

Returning directly from the middle of the `while` loop is inappropriate, because the simulated stack might not be empty. So for step 4, we'll convert any remaining `returns` into assignments to `$return`. Later on in the function, we'll return the contents of `$return` if the simulated stack is empty:

CODE LIBRARY
fib-5

```
sub fib {
  my $n = shift;
  my ($s1, $s2, $return);
  my $BRANCH = 0;
  while (1) {
    if ($n < 2) {
      $return = $n;
    } else {
      if ($BRANCH == 0) {
        $return = fib($n-2);
      } elsif ($BRANCH == 1) {
        $s1 = $return;
        $return = fib($n-1);
      } elsif ($BRANCH == 2) {
        $return = $s1 + $s2;
      }
    }
  }
}
```

Step 5 is the important one: Replace all the recursive calls with code that pushes the function state onto the synthetic stack and then transfers control back to the top of the function:

CODE LIBRARY
fib-6

```
sub fib {
  my $n = shift;
  my ($s1, $s2, $return);
  my $BRANCH = 0;
  my @STACK;
  while (1) {
    if ($n < 2) {
      $return = $n;
    } else {
      if ($BRANCH == 0) {
        push @STACK, [ $BRANCH, $s1, $s2, $n ];
        $n -= 2;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 1) {
        $s1 = $return;
        push @STACK, [ $BRANCH, $s1, $s2, $n ];
        $n -= 1;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 2) {
        $s2 = $return;
        $return = $s1 + $s2;
      }
    }
  }
}
```

Since this is important, let's look at one of the calls in detail. When `fib()` calls `fib($n-2)`, it saves all its state and then transfers control back to the top of `fib()`, which starts up just as before, but with argument `$n-2` instead of `$n`. The code we put in is doing exactly that. It saves the current state on the stack:

```
push @STACK, [ $BRANCH, $s1, $s2, $n ];
```

Then it adjusts the value of the argument from `$n` to `$n-2`:

```
$n -= 2;
```

Then it adjusts the value of `$BRANCH` to say that control should continue from the top of the function, not the middle:

```
$BRANCH = 0;
```

This was unnecessary in this case, since `$BRANCH` was already 0, but I left it in for symmetry with the second branch, where it is needed.

Finally, we transfer control back up to the top:

```
next;
```

We're almost done. We've simulated the recursive calls, and the last thing we need to do is simulate the returns. The function's desired return value is in `$return`. To simulate a function return, check to see if the synthetic stack is empty. If so, then the function is really returning to its caller, and should just return `$return`. Otherwise, we pop the saved state off the stack and resume execution where we left off:

CODE LIBRARY
fib-7

```
sub fib {
  my $n = shift;
  my ($s1, $s2, $return);
  my $BRANCH = 0;
  my @STACK;
  while (1) {
    if ($n < 2) {
      $return = $n;
    } else {
      if ($BRANCH == 0) {
        push @STACK, [ $BRANCH, $s1, $s2, $n ];
        $n -= 2;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 1) {
        $s1 = $return;
        push @STACK, [ $BRANCH, $s1, $s2, $n ];
        $n -= 1;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 2) {
        $s2 = $return;
        $return = $s1 + $s2;
      }
    }

    return $return unless @STACK;
    ($BRANCH, $s1, $s2, $n) = @{pop @STACK};
    $BRANCH++;
  }
}
```

We increment `$BRANCH` so that execution will resume with the chunk *following* the one we were in when we made the call.

And amazingly, we're now done. This function does indeed compute Fibonacci numbers.

Because I was showing a general transformation of a recursive into a nonrecursive function, the result has some unnecessary code. For example, I included an unnecessary `$BRANCH = 0` line for symmetry. In branch 1, we assign `$s1` from `$return` and then immediately push its value onto the stack; we may as well push `$return` directly onto the stack without the intervening assignment. In branch 0, we push `$s1` into the stack, but its value is always undefined at this point, so we may as well just push `0` directly:

CODE LIBRARY
fib-8

```
sub fib {
  my $n = shift;
  my ($s1, $s2, $return);
  my $BRANCH = 0;
  my @STACK;
  while (1) {
    if ($n < 2) {
      $return = $n;
    } else {
      if ($BRANCH == 0) {
        push @STACK, [ $BRANCH, 0, $s2, $n ];
        $n -= 2;
        next;
      } elsif ($BRANCH == 1) {
        push @STACK, [ $BRANCH, $return, $s2, $n ];
        $n -= 1;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 2) {
        $s2 = $return;
```

```
          $return = $s1 + $s2;
        }
      }

      return $return unless @STACK;
      ($BRANCH, $s1, $s2, $n) = @{pop @STACK};
      $BRANCH++;
    }
  }
```

Performing the same sort of eliminations for `$s2` as we did for `$s1`, we discover that `$s2` is *entirely unnecessary*. The only place it's used is in branch 2, and it's used immediately after it's assigned:

CODE LIBRARY
fib-9

```
sub fib {
  my $n = shift;
  my ($s1, $return);
  my $BRANCH = 0;
  my @STACK;
  while (1) {
    if ($n < 2) {
      $return = $n;
    } else {
      if ($BRANCH == 0) {
        push @STACK, [ $BRANCH, 0, $n ];
        $n -= 2;
        next;
      } elsif ($BRANCH == 1) {
        push @STACK, [ $BRANCH, $return, $n ];
        $n -= 1;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 2) {
        $return += $s1;
      }
    }

    return $return unless @STACK;
    ($BRANCH, $s1, $n) = @{pop @STACK};
    $BRANCH++;
  }
}
```

We might also optimize branch 0 a little. In branch 0, we push the stack, decrement `$n` by 2, and pass control back to the top of the function. Typically, we then come back immediately and do it again, forming a loop. We can tighten up the loop:

CODE LIBRARY
fib-10

```
sub fib {
  my $n = shift;
  my ($s1, $return);
  my $BRANCH = 0;
  my @STACK;
  while (1) {
    if ($n < 2) {
      $return = $n;
    } else {
      if ($BRANCH == 0) {
        push (@STACK, [ $BRANCH, 0, $n ]), $n -= 2 while $n >= 2;
        $return = $n;
      } elsif ($BRANCH == 1) {
        push @STACK, [ $BRANCH, $return, $n ];
        $n -= 1;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 2) {
        $return += $s1;
      }
    }

    return $return unless @STACK;
    ($BRANCH, $s1, $n) = @{pop @STACK};
    $BRANCH++;
  }
}
```

Since that tight loop is more efficient than the large main loop, we'd like to do it as often as possible. As it is, though, we do it only about *n*/2 times. Since it doesn't matter whether `fib()` makes the `fib($n-2)` or the `fib($n-1)` call first, we can exchange the first and second chunks, giving us:

CODE LIBRARY
fib-11

```
sub fib {
  my $n = shift;
  my ($s1, $return);
  my $BRANCH = 0;
  my @STACK;
  while (1) {
    if ($n < 2) {
      $return = $n;
    } else {
      if ($BRANCH == 0) {
        push (@STACK, [ $BRANCH, 0, $n ]), $n -= 1 while $n >= 2;
        $return = $n;
      } elsif ($BRANCH == 1) {
        push @STACK, [ $BRANCH, $return, $n ];
        $n -= 2;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 2) {
        $return += $s1;
      }
    }

    return $return unless @STACK;
    ($BRANCH, $s1, $n) = @{pop @STACK};
    $BRANCH++;
  }
}
```

This is a little faster than the previous version.

We can also clean up one more line of code by eliminating `$BRANCH++` at the bottom. Instead of pushing the old value of `$BRANCH` onto the stack and then incrementing it after we pop it again, we'll just push the value of `$BRANCH` that we want to have when we return:

CODE LIBRARY
fib-12

```
sub fib {
  my $n = shift;
  my ($s1, $return);
  my $BRANCH = 0;
  my @STACK;
  while (1) {
    if ($n < 2) {
      $return = $n;
    } else {
      if ($BRANCH == 0) {
        push (@STACK, [ 1, 0, $n ]), $n -= 1 while $n >= 2;
        $return = $n;
      } elsif ($BRANCH == 1) {
        push @STACK, [ 2, $return, $n ];
        $n -= 2;
        $BRANCH = 0;
        next;
      } elsif ($BRANCH == 2) {
        $return += $s1;
      }
    }

    return $return unless @STACK;
    ($BRANCH, $s1, $n) = @{pop @STACK};
  }
}
```

There are several things we can learn from all of this. Most important, it affords us a detailed look into what is really required to implement recursive calls. Many of the small tweaks and optimizations we applied at the end of the conversion process are directly analogous to optimizations that compilers and interpreters can perform internally.

Recursion elimination may also be useful in reducing the memory footprint of a function. With Perl's built-in recursion, you don't get a choice about what state is saved on the stack: Absolutely everything is saved. Once we have the stack represented explicitly in the program, it may become clear that not everything needs to be saved on every call, and we may be able to reduce stack usage, as we did by eliminating `$s2`.

Finally, in some cases it will turn out that the iterative version of the code is faster or simpler than the recursive version. In these cases, such as the power set function example, the simplifications suggested by recursion elimination may lead to a cascade of further simplifications.