CS-111 Recursion Notes and Exercises

Contents

1 A First Example **1**

2 Fibonacci Numbers **4**

3 Organizing a Mob **5**

4 The Towers of Hanoi **6**

5 Ackermann's Function **8**

6 Exhaustive Search via Backtracking **9**

October 14, 1999

1 A First Example

In natural languages, a recursive (circular) definition is meaningless. For example, suppose that I define *"vituperate"* as *"to speak with vituperation"*. This is a recusive definition, since it refers to the word being defined.

Some kinds of computational problems, however, have natural and meaningful recursive definitions. A recursive function—one that calls itself—can solve such a problem in a direct and simple way. Iterative (loop-based) solutions to the same problem are often complicated and hard to understand.

The standard first example is the factorial function $n!$ (pronounced **n factorial**), the product of all positive integers up to n. For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

We wish to write a function that takes an integer n and returns $n!$. The question to ask is, how could the problem be solved by breaking it into smaller pieces? The crucial observation is that $n! = n \cdot (n-1)!$. The term $(n-1)!$ is the smaller piece; once a solution is found for it, a multiplication by n gives the final result. How is $(n-1)!$ computed? The same way, recursively, by breaking it into the product $(n-1) \cdot (n-2)!$. We can continue this process with smaller and smaller pieces, until we reach the *base case*—an instance of the problem so simple that it can be computed directly, without recursion. Our base case is $n = 0$, since $0!$ is defined to be 1. So—

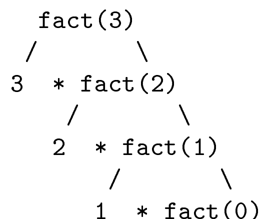
$$n! = \begin{cases} 1 & : \text{ if } n = 0 \\ n \cdot (n-1)! & : \text{ if } n > 0 \end{cases}$$

Translating a recursive definition into a C++ function is easy:

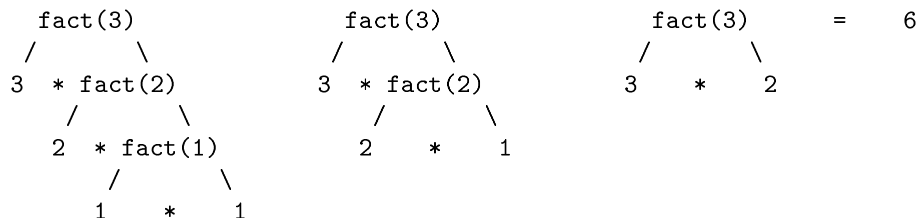
```
int fact(int n)
{
  if (n==0)
    return 1;
  else return n * fact(n-1);
}
```

How would a computer evalutate $fact(n)$? Consider, for example, the case $n = 3$. According to our code, $fact(3)$ returns the integer $3 \cdot fact(2)$, but $fact(2)$ must first be evaluated in order to compute this product. It is evaluated the same way, recursively.

A *recursion tree* helps to visualize the action:



We have now reached the base case. The value 1 is returned and the recursion "unwinds" like this—



The trick to tracking the action in your mind is to imagine each execution of the function being performed on a separate machine with its own local variable(s). Each machine waits for the one below it in the recursion tree to return a result so that the multiplication can be performed. This agrees with our understanding of C++ functions as completely self-contained modules that receive inputs and return outputs without any other connection to the outside world.

Our factorial function could easily have been implemented as a multiplication within a loop, and this would in fact have been much more efficient since function calls are expensive—the operating system must halt the program and perform certain administrative tasks before control is transferred to the function. But we will soon see some problems that do not have obvious iterative solutions. In these cases recursion can still be removed, but only at the expense of clarity and simplicity.

There are many basic computational tasks that have nice recursive solutions. One example is sorting a list. Consider also the compiler's job of translating source code into assembly code. It is relatively easy to design a compiler that performs this task recursively.

Exercise 1 *Evaluate $f(3), f(4), f(5)$ and $f(6)$ for*

$$f(n) = \begin{cases} n & : \text{ if } n \leq 1 \\ n + f(n/2) & : \text{ if } n > 1 \text{ is even} \\ f(\frac{n+1}{2}) + f(\frac{n-1}{2}) & : \text{ if } n > 1 \text{ is odd} \end{cases}$$

Exercise 2 *Write a recursive function that takes an integer and returns the sum of the digits of that integer. Four or five lines of code should suffice. Hint: Consider the sequence of digits $d_k d_{k-1} \cdots d_1$ of input x, where d_i is the ith digit of x from the right. Observe that the sum would be easy to compute if we knew the sum $d_k + \cdots + d_2$, because then all we would have to do is add d_1.*

Exercise 3 *What is returned by $f(5,3)$?*

```
// Assume x greater than or equal to y.
int f(int x, int y)
{
  if (y==0 || y==x)
    return 1;
  return f(x-1,y) + f(x-1,y-1);
}
```

Exercise 4 *What is returned by $g(2,7)$?*

```
// Assume x less than or equal to y.
int g(int x, int y)
{
  if (x==y)
    return x*y;
  return g(x, (x+y)/2) + g((x+y)/2+1,y);
}
```

Exercise 5 *Write a recursive function f with one positive int parameter n. The function prints out $2^n - 1$ integers as follows:*

```
f(1) outputs 1
f(2) outputs 1 2 1
f(3) outputs 1 2 1 3 1 2 1
f(4) outputs 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
f(5) outputs 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

and so on.
```

The function does not return a value. Four or five lines of code will do the job.

Exercise 6 *Write a recursive function that takes a positive integer and returns the number of 1's in the binary representation of that integer.*

Exercise 7 *Explain in one simple sentence what the following function computes.*

```
int f(int n)
{
   if (n==0)
      return 0;
   if (n%2==0)
      return f(n/2);
   else return 1+f(n/2);
}
```

Exercise 8 *Evaluate $f(5)$. What familiar function does f compute?*

```
int f(int b)
{
  if (b==0)
    return 1;
  else return g(b, f(b-1));
}
```

```
int g(int a, int b)
{
  if (b==0)
    return 0;
  else return a + g(a,b-1);
}
```

Exercise 9 *Explain what common task the following function performs.*

```
double m(double x, int y)
{
  if (y==0)
    return 1;
  if (y%2==0)
    return m(x,y/2) * m(x,y/2);
  else return x*m(x,y-1);
}
```

2 Fibonacci Numbers

The Fibonacci numbers are the terms of the infinite sequence

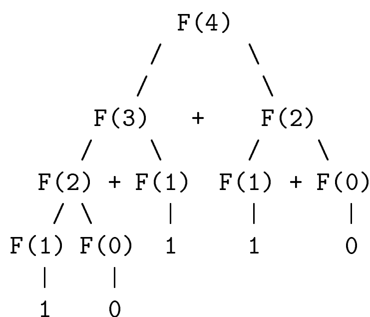
$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

where the two initial terms are 0 and 1, and any other term is the sum of the two preceding it. Let us denote by $F(n)$ the n-th Fibonacci number. We consider 0 to be the 0th Fibonacci number, i.e. $F(0) = 0$ and, for example, $F(7) = 13, F(8) = 21$, etc. Formally—

$$F(n) = \begin{cases} n & : \text{ if } n = 0, 1 \\ F(n-2) + F(n-1) & : \text{ if } n > 1 \end{cases}$$

An interpretation of the sequence relates its terms to the (exponentially growing) population of rabbits after fixed units of time, say months. We begin with a single pair of rabbits and apply the rule that every female rabbit at age two months gives birth to a pair of rabbits. How many pairs of rabbits are there after n months (assuming that none has died)? The rabbits that were living during the previous month are still alive, but also the rabbits that were born two months ago have just reproduced. So the current number of living pairs is the number of pairs living in the previous month plus the number of pairs living two months ago.

Here is a recursion tree for $F(4)$:



You can see that there is a lot of redundancy—$F(2)$, for example, is computed twice. In fact, the amount of duplicated computation grows exponentially. An iterative solution is far more efficient.

Exercise 10 *Write an iterative function that takes an integer* n *and returns the* n*th Fibonacci number. You need a single loop and several integer variables, properly initialized. Do not use an array.*

Exercise 11 *Write an iterative function that computes the first* n *Fibonacci numbers, saving them all in a dynamically allocated array. At the expense of memory, this function is even simpler than the one of the previous exercise.*

Exercise 12 *This exercise assumes that you have studied elementary discrete math, specifically inductive proofs. Prove the following facts about Fibonacci numbers:*

1. $F(1) + F(3) + F(5) + \cdots + F(2n-1) = F(2n)$.
2. $F(2) + F(4) + F(6) + \cdots + F(2n) = F(2n+1) - 1$.
3. $F(n+1)^2 - F(n)F(n+2) = (-1)^n$.
4. $F(1)F(2) + F(2)F(3) + F(3)F(4) + \cdots + F(2n-1)F(2n) = F(2n)^2$.
5. $F(1)F(2) + F(2)F(3) + F(3)F(4) + \cdots + F(2n)F(2n+1) = F(2n+1)^2 - 1$.

Exercise 13 *This exercise assumes that you know about binary trees, a certain recursively defined data structure.*

Define the Fibonacci tree of order n *as follows: if $n = 0$ or $n = 1$, the tree consists of a single node. If $n \geq 2$, the tree consists of a root, with the Fibonacci tree of order $n-1$ as the left subtree and the Fibonacci tree of order $n-2$ as the right subtree.*

What is the number of leaves in the Fibonacci tree of order n? What is its depth?

3 Organizing a Mob

100 students wish to form an unruly mob of size 10 in order to demand easier homeworks from the instructor. How many possible 10-student mobs are there? More generally, how many ways are there to choose k objects from a group of n?

If you have studied discrete math, then you know that the number is given by a simple formula. But an easy recursive solution can be derived using a standard technique. The question is, as always, how can we break the problem into smaller, simpler pieces?

Let us solve the problem for the particular instance given above, and then generalize it. So we wish to compute $d(100, 10)$, the number of ways to choose 10 outraged students from a class of 100. We designate a distinguished member of the class, say Jane. Now, in every mob Jane is either present or not present, and we consider these two cases separately.

1. Jane is present. There are $d(99, 9)$ such mobs, since the remaining 9 members must be chosen from a group of 99 (100 minus Jane) students.
2. Jane is not present. There are $d(99, 10)$ such mobs since all 10 members must be chosen from a group of 99 students.

So we get $d(100, 10) = d(99, 9) + d(99, 10)$. Both terms of the sum can be computed recursively by the same process. In general—

$$d(n, k) = d(n-1, k-1) + d(n-1, k).$$

Note that it is possible for both arguments to be decremented or for only the first argument to be decremented in the recursive calls. This gives rise to two base cases: $d(n, n) = 1$ since there is only one way to choose n objects from a group of size n, and $d(n, 0) = 1$ since there is only one way to choose none of them. We have defined—

$$d(n, k) = \begin{cases} 1 & : \text{ if } k = 0 \\ 1 & : \text{ if } k = n \\ d(n-1, k-1) + d(n-1, k) & : \text{ otherwise} \end{cases}$$

and translating this to C++ is trivial.

The usual notation for $d(n, k)$ is $\binom{n}{k}$, pronounced **n choose k**. The following exercises use this notation.

Exercise 14 *Prove that* $\binom{n}{k} = \binom{n}{n-k}$.

Exercise 15 *This exercise assumes that you have studied inductive proofs. Prove by induction that for any real numbers* x *and* y *and any non-negative integer* n*:*

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}$$

For this reason, $\binom{n}{k}$ *is also called the* binomial coefficient.

Exercise 16 *Can you generalize the result of the previous exercise to get multinomial coefficients, e.g.*

$$(x_1 + x_2 + \cdots + x_k)^n = ?$$

4 The Towers of Hanoi

Three huge pegs stand in the courtyard of a monastery in Hanoi. 64 disks are stacked on the first peg, in order of size from the largest on the bottom to the smallest on the top.

The monks want to move the entire stack from the first peg to the third, using the second as a temporary holding spot. There are two restrictions on how the disks can be moved. First, the disks are heavy, so only one at a time can be moved. Second, the size ordering must be maintained on every peg so that each disk is smaller than the one below it.

It was foretold that once the monks accomplish their task, the world will end. We will see that this is a reasonable prophesy, for even generously supposing that the monks could move a disk each second, the sun would burn out long before the monks could move the entire stack.

Let's label the pegs A, B and C. We want to write a function

```
void towers(int n, char start, char finish, char spare);
```

that outputs the steps required to move n disks from A to B, using C as a spare.

For example, towers(3, 'A', 'B', 'C') would output:

```
Move disk from A to B
Move disk from A to C
Move disk from B to C
Move disk from A to B
Move disk from C to A
Move disk from C to B
Move disk from A to B
```

Let us concentrate on the case $n = 64$. It seems, at first, that we are up against an exceedingly difficult problem, but there is a natural recursive solution.

The crucial observation is that this problem would be easy to solve if only we knew how to move 63 disks from one peg to another. To see why, suppose we have this ability. We can move the tower from A to B in 3 steps:

1. Move 63 disks from peg A to peg C.
2. Move the single disk on peg A to peg B.
3. Move the 63 disks from peg C to peg B.

But how do we move 63 disks from one peg to another? Just repeat the same procedure recursively. The base case is when we are moving a single disk—there is no real work to be done, just pick it up and move it. As always, having defined the problem recursively, the code follows easily:

```
void towers(int n, char start, char finish, char spare)
{
  if (n == 1)
    cout << "Move disk from " << start << " to "  <<  finish << endl;
  else {
    towers(n-1, start, spare, finish);
    cout << "Move disk from " << start << " to "  << finish << endl;
    towers(n-1, spare, finish, start);
  }
}
```

You should step through the code on a small sample input, say $towers(4,'A','B','C')$.

How many times must a disk be moved? The answer is $2^n - 1$, which as an exercise you can easily prove. So to move a tower of 64 disks at a rate of one per second takes $2^{64} - 1$ seconds, or about 500 trillion years. There is no faster way, as you can also easily determine.

The second recursive call can be replaced with a loop, as is always the case with tail recursion (i.e. when no code follows the recursive call), but the first call is not so easily replaced. It *can* be done, but at considerable expense of clarity and simplicity.

Exercise 17 *This exercise assumes that you have had an introduction to data structures, or at least that you know about stacks. Remove the recursion from the* towers *function. Push the appropriate parameters onto a stack to simulate the recursive calls, and pop the parameters from the stack to handle the return from recursion.*

Open Question. Consider the problem of moving a tower from one peg to another using *two* spare pegs. There is no known solution that is provably optimal. The above algorithm will do the job—it just ignores the second spare peg—but perhaps you can improve on this...?

5 Ackermann's Function

Ackermann's function, conceived in the 1920s, has an interesting theoretical property and arises in the analysis of certain algorithms. Even small inputs result in a huge number of recursive calls, and for this reason it is often used to test how efficiently a particular computer implements recursion.

$$Ack(m,n) = \begin{cases} n+1 & : \text{ if } m = 0 \text{ and } n > 0 \\ Ack(m-1,1) & : \text{ if } n = 0 \text{ and } m > 0 \\ Ack(m-1, Ack(m, n-1)) & : \text{ if } m > 0 \text{ and } n > 0 \end{cases}$$

For example—

$$\begin{aligned} Ack(2,1) &= Ack(1, Ack(2,0)) \\ &= Ack(1, Ack(1,1)) \\ &= Ack(1, Ack(0, Ack(1,0))) \\ &= Ack(1, Ack(0, Ack(0,1))) \\ &= Ack(1, Ack(0,2)) \\ &= Ack(1,3) \\ &= Ack(0, Ack(1,2)) \\ &= Ack(0, Ack(0, Ack(1,1))) \\ &= Ack(0, Ack(0, Ack(0, Ack(1,0)))) \\ &= Ack(0, Ack(0, Ack(0, Ack(0,1)))) \\ &= Ack(0, Ack(0, Ack(0,2))) \\ &= Ack(0, Ack(0,3)) \\ &= Ack(0,4) \\ &= 5 \end{aligned}$$

Exercise 18 *What is $Ack(3,2)$? What is the smallest natural number n such that the number of recursive calls in $Ack(3,n)$ is greater than 10,000?*

Exercise 19 *(Difficult) Write an iterative function that computes Ackermann's function.*

6 Exhaustive Search via Backtracking

It is possible to write an optimal chess-playing program. It simply examines all possible opening moves, and for each of these it examines all possible next moves, and so on, until all possibilities have been exhausted.[1] The problem with this strategy is that the sun will burn out long before it could be computed up to even 20 levels.

Actual chess-playing programs choose only a few profitable-looking paths at each level to pursue, up to a certain number of levels. But for less complicated problems, exhaustive search may be feasible. An example is the well-known *8 Queens* problem. You are given an 8x8 chessboard and 8 queens, and you must place these queens on the board so that none of them is attacking any other.[2]

Obviously there must be exactly one queen in each row. A *safe* position is one that is not being attacked by any other queen on the board. Let n represent the number of queens that are currently on the board. Here, in pseudo-code, is the basic idea:

```
AddQueen()
{
  for (every safe position p in row n) {
    Place a queen in position p;
    n = n + 1;
    if (n == 8)
      Print the configuration and exit program;
    else AddQueen();
    Remove the queen from position p;
    n = n - 1;
  }
}
```

The only way to return from the *AddQueen* function is by being unable to find a safe position in the current row, i.e. by reaching a deadend. Therefore, after the return from the recursive call, we need to remove the current queen and continue looking for the next safe position in its row. This is called *backtracking*.

Here is a solution:

```
Q  .  .  .  .  .  .  .

.  .  .  .  Q  .  .  .

.  .  .  .  .  .  .  Q

.  .  .  .  .  Q  .  .

.  .  Q  .  .  .  .  .

.  .  .  .  .  .  Q  .

.  Q  .  .  .  .  .  .

.  .  .  Q  .  .  .  .
```

[1] All chess games end after a finite number of steps. If 50 moves are made without a piece being captured, or if any player repeats a sequence of moves 3 times in a row, then the game ends in a draw.

[2] A queen can move any number of squares left, right, up, down, or diagonally.

Exercise 20 *Implement the backtracking algorithm to solve the* 8 Queens *problem. Maintain the board as a boolean matrix. Display a solution as above. It should be possible to modify a single line of your code in order to solve the general* N Queens *problem for any positive integer N.*

Exercise 21 *This exercise assumes that you have know about stacks. Implement the backtracking algorithm using a stack rather than recursion. For each queen placed, push its coordinates onto the stack. When a deadend is reached, backtrack to the previous position by popping a coordinate off the stack. Count the number of stack calls.*

Exercise 22 *Write a recursive backtracking function to solve the* Knight's Tour *problem. The idea is to place a knight at a designated position on an empty chessboard and to compute a sequence of legal moves such that each square is visited exactly once.*[3] *For example, starting in the lower left corner, here is one solution:*

```
14 11 16  5 22  9 20  7
17  4 13 10 19  6 23 64
12 15 18 29 24 21  8 31
 3 28 25 44 33 30 63 42
26 45 34 59 38 43 32 55
35  2 27 48 51 54 41 62
46 49 58 37 60 39 56 53
 1 36 47 50 57 52 61 40
```

[3] A knight moves one square horizontally then two vertically, or two squares horizontally then one vertically.