31 Recursive Design

1 Designing recursively

There are three essential steps in designing a recursive method. The first step is the following:

1. *Write down the header of the method, including a precise description of its purpose and design range.*

This is very important – you will need to invoke the method as you develop the body, and you won't be able to do that unless you have a method name and know the types of the parameters. Suppose, for example, that we want to write a recursive function to compute *powers* such as 3^2 (=9) and 5^3 (=125). We will confine ourselves to natural exponents (the raised number such as the 3 in 5^3 is called the *exponent*). The definition of m^n is $m \times m \times \ldots \times m$ (n times). It is standard to define m^0 to be 1, although this may not seem intuitive. The header of the method is

```
static int power(int m, int n) // m to the power n, assuming n>=0
```

The second step is to consider the base case. Ask yourself:

2. *For what simple values of the arguments can I do what is required without recourse to a recursive call?*

Typical responses will be for 0 or 1 for integer arguments (more usually 0), and empty strings or strings of length 1 for string arguments (usually empty strings). In the case of `power(m,n)`, the base case is clearly going to be `n`=0, for which we will return 1.

The third step is the hardest (but is much harder if you skip the first two steps):

3. *For cases other than the base case, how can I recursively solve a smaller*

instance of the problem and use that to construct a complete solution.

Returning to the example of `power()`, we have already taken care of the case `n=0`, and so we know that `n>0` when the recursive call happens. A simple property of exponents that you will know from school (or will easily convince yourself of) is

$$m^n = m \times m^{n-1} \text{ for } n>0$$

Expressed in terms of `power()` this is:

`power(m,n) = m*power(m,n-1)` for `n`>0

The call `power(m,n-1)` is within the design range because we have `n`>0 at the point of call, and so `n-1`≥0. Furthermore, `n-1` is closer to 0 than `n`, and so we know the recursion will terminate. That's all there is to it! If the recursive call is applied to a smaller instance of the problem, and you have supplied a base case, you have nothing more to worry about. This is the key to mastering recursion – *don't pursue the recursive call, but trust that it will work as long as you have applied it to a smaller instance of the problem*. Finally, we assemble the constituent parts:

```
static int power(int m, int n)  { // m to the power n, assuming n>=0
    if (n==0) return 1;
    else return m*power(m,n-1);
}
```

Example 1: counting letters

We write a function to count the number of letters in a given string `s`. For example, the number of letters in "May 5th 2002" is 5, and the number of letters in "2002" is 0.

```
static int numLetters(String s)
// Number of letters in s, s not null
```

For a base case, it is clear that we can solve the problem directly if `s` is empty, i.e. if `s.length()` is 0 – the result is obviously 0 because the empty string has no letters. For the recursive case, we can assume `s` is non-empty because empty strings are handled by the base case. Hence `s` has a character in position 0 (`s.charAt(0)`). Let us call the string obtained from `s` by dropping its first character *the tail* of `s`. Now observe that the number of letters in `s` is equal to the number of letters in the tail of `s`, plus 1 if the first character is a letter. How may we find the number of letters in the tail of `s`? By invoking `numLetters(s.substring(1))` – just read the header and descriptive comment of `numLetters()`! Despite the fact that `numLetters()` is the method we are in the process of writing, the invocation is effective because the new argument is smaller than the incoming one – the tail of `s` is clearly smaller that `s` itself, i.e. it is closer to the base case of the empty string.

```
static int numLetters(String s) {
```

```
// Number of letters in s (s not null)
    if (s.length()==0) return 0;
    else {
        if (Character.isLetter(s.charAt(0)))
            return 1 + numLetters(s.substring(1));
        else return numLetters(s.substring(1));
    }
}
```

As an example of using `numLetters()`, the following program displays the number of letters in a line of text entered at the keyboard.

```
class NumLetters {

    static int numLetters(String s) {
        .....
    }

    public static void main(String args[]) {
        System.out.print("Enter a line: ");
        String s = Console.readString();
        System.out.println("Number of letters = " + numLetters(s));
    }
}
```

Example 2: counting 9's

We write a function to determine how many 9's occur in the decimal representation of a (natural) number. For example, the number of 9's in 39791 is 2.

```
static int count9(int n) {
// Number of 9's in decimal representation of n, n>=0
```

For example, `count9(39791)` should return 2, and `count9(354)` should return 0. For a base case, we observe that we can solve the problem directly if `n` is a single digit, i.e. if `n`<10. If single-digit `n` is 9 the result is 1, and otherwise the result is 0. For the recursive case, we know that `n` is at least 10 (because smaller values are dealt with as base cases) and so it has at least two decimal digits. We can therefore break `n` into two smaller parts, count the number of 9's in each part, and return the sum of the two counts. How should we break `n` into parts – any way we like. The simplest is to take the rightmost digit (`n%10`) as one part, and all but the rightmost digit as the other part (`n/10`). How do we count the number of 9's in each part? By invoking `count9()` – this is effective because both parts are smaller than the original number `n`.

```
static int count9(int n) {
// Number of 9's in decimal representation of n, n>=0
```

```
        if (n<9) return 0;
        else if (n==9) return 1;
        else return count9(n%10) + count9(n/10);
}
```

Observe that the termination criteria are met:

(i) The method is designed to take natural arguments. The arguments of the recursive calls, `n%10` and `n/10`, are clearly at least 0.
(ii) If `n` is small, which in this case means `n`<10, the result is computed directly.
(iii) The arguments of the recursive call, `n%10` and `n/10`, are smaller than `n` because `n`≥10 holds at the point of recursion.

Example 3: minimum in array

We want a recursive function to compute the minimum value in a (non-empty) array of integers:

```
static int min(int[] w)
// Minimum in w, assuming w.length>0
```

For example, if array `w` is introduced thus:

```
int[] w = {4, 2, -3, 2, 5, 9, 3, -1, 8, 6};
```

then `min(w)` should return -3.

If `w` has just one element then `w[0]` is clearly the minimum. For the recursive case, we know `w` has length 2 or more, so imagine it as being composed of two parts. This is a good way to think about the problem because (as a moment's reflection will convince you) the minimum in `w` is the minimum of the minima in each of the two parts. For example, the minimum in the list 4, 2, -3, 2, 5, 9, 3, -1, 8, 6 can be got by finding the minimum in the first half 4, 2, -3, 2, 5 (that's -3) and the minimum in the second half 9, 3, -1, 8, 6 (that's -1), and then taking the minimum of the respective minima -3 and -1, yielding -3. Here we have broken `w` into two equal parts, but actually the relative sizes of the parts doesn't matter. In coding it is easiest to let the first part consist of `w[0]` only, and the second part consist of `w[1..]`. The minimum in the first part is trivially `w[0]` – there is just one element and so it must be the minimum.

Let *m* denote the minimum in the second part. How can we compute *m*? We would like to invoke `min()` recursively, but that seems very difficult. We propose instead to change the specification of the method by the addition of an extra parameter. The revised method will find the minimum in a given non-empty *tail* of an integer array, where by "tail" we mean any right-hand segment of the array. The method we have in mind is:

```
static int minTail(int[] w, int i)
// Minimum in w[i..], 0<=i<w.length
```

For example, if array `w` is introduced thus:

```
int[] w = {4, 2, -3, 2, 5, 9, 3, -1, 8, 6};
```

then `minTail(w,4)` returns -1 (because -1 is the smallest value in `w[4..]`, i.e. 5, 9, 3, -1, 8, 6), and `minTail(w,0)` returns -3, i.e. the minimum in `w`. In designing recursive methods, it is not uncommon to introduce an extra parameter solely to facilitate the recursion. Indeed, it is quite common in problems concerning arrays.

For the revised problem, we look for a base case. Clearly if the tail `w[i..]` has just one element (i.e. `i=w.length-1`) then its minimum is `w[i]`. For the recursive case, we know `w[i..]` has length 2 or more. Viewing it as being composed of two parts, the minimum in `w[i..]` is again the minimum of the minima in each of the two parts. We let the first part consist of `w[i]` only and the second part consist of `w[i+1..]`. The minimum in the first part is evidently `w[i]`, and the minimum in the second part is just `minTail(w,i+1)` – this is effective because `w[i+1..]` is a smaller tail of `w` than `w[i..]`, i.e. it is closer to the base case of `i=w.length-1`. The solution to the problem is:

```
static int min(int[] w)  { // Minimum in w, assuming w.length>0
    return minTail(w,0);
}

static int minTail(int[] w, int i) { // Minimum in w[i..], 0<=i<w.length
    if (i==w.length-1) return w[i];
    else {
        int m = minTail(w,i+1);
        if (w[i]<m) return w[i];
        else return m;
    }
}
```

Previously we would have solved this problem using iteration. Actually recursion in this case is not any easier to code than iteration, and the recursion is computationally a little more expensive. Nevertheless, it is a simple illustrative example of generalising a problem to facilitate recursion.

2 Recursive procedures

Procedures may be recursive, and their design is no different from that of functions. We illustrate with some examples.

Example 1: verbalising a number

We write a program which reads a natural number and prints it in words digit by digit. The following illustrates a typical input and output:

```
Enter a natural number: 3546
three five four six
```

The core of the program will be a recursive method with the following heading:

```
static void putVerbal(int n)  // print decimal digits of n in words, n>=0
```

For the base case, we can take values of `n` less than 10 because it is easy to print the word equivalent of a single digit. For the recursive case, we know that `n` is at least 10 and so has at least two decimal digits. We break `n` into two smaller parts and verbalise the two parts in turn – this is effective because both parts are smaller than the original number `n`. For the left part we take all but the rightmost (i.e. least significant) digit of `n`, and for the right part we take the rightmost digit. The program is:

```
class Verbalising {

    static void putVerbal(int n) { // print decimal digits of n in words, n>=0
        if (n<10) output(n);
        else {
            putVerbal(n/10);  putVerbal(n%10);
        }
    }

    static void output(int n) { // print n as a word, 0<=n<10
        String[] digit = {  "zero", "one", "two", "three", "four",
                            "five", "six", "seven", "eight", "nine"};
        System.out.print(digit[n] + " ");
    }

    public static void main(String[] args) {
        System.out.println("Enter a natural number: ");
        int n = Console.readInt();
        putVerbal(n);
    }
}
```

`putVerbal()` remains valid if the second recursive call – `putVerbal(n%10)` – is replaced with `output(n)` and it is computationally a little cheaper because it eliminates one method invocation.

The output could as easily be spoken as printed. It only requires that `digit[n]` be the name of a clip that speaks `n`, and that we play it rather than print it. The following version of `output` does that using `.wav` clips under Windows, and utilising some of the Java library's audio classes. The program's directory should contain sound clips `0.wav`, `2.wav`, `3.wav`, ... , `9.wav` (`java.sound.sampled.*` and `java.io.*` must be imported).

```
    static void output(int n) { // speak n, 0<=n<10
        try {
            String[] digit = {  "0.wav", "1.wav", "2.wav", "3.wav", "4.wav",
                                "5.wav", "6.wav", "7.wav", "8.wav", "9.wav"};
            Clip clip = AudioSystem.getClip();
            AudioInputStream ais =
                    AudioSystem.getAudioInputStream(new File(digit[n]));
            clip.open(ais);  clip.loop(0);
            Thread.sleep(clip.getMicrosecondLength()/1000); // wait till clip ends
        }
        catch (Exception e){System.out.println("Oh!Oh!");}
    }
```

Example 2: Scanning an array

We write a recursive procedure which prints the positive elements in a tail of an integer array:

```
static void writePos(int[] w, int i)
// Display the postives in w[i..], 0<=i<=w.length
```

For example, if array `w` is introduced thus:

```
int[] w = {4, -2, -3, 2, -5, 9, 3, -7, 8, -6};
```

then `writePos(w,4)` displays `9 3 8`. `writePos()` is similar in most respects to the recursive function which computes the minimum in the tail of an array. However, on this occasion the base case is the empty tail for which no action is taken – the empty tail has no elements and therefore it has no positive elements.

```
static void writePos(int[] w, int i) {
// Display the postives in w[i..], 0<=i<=w.length
    if (i<w.length) {
        if (w[i]>0) System.out.print(w[i] + " ");
        writePos(w,i+1);
    }
}
```

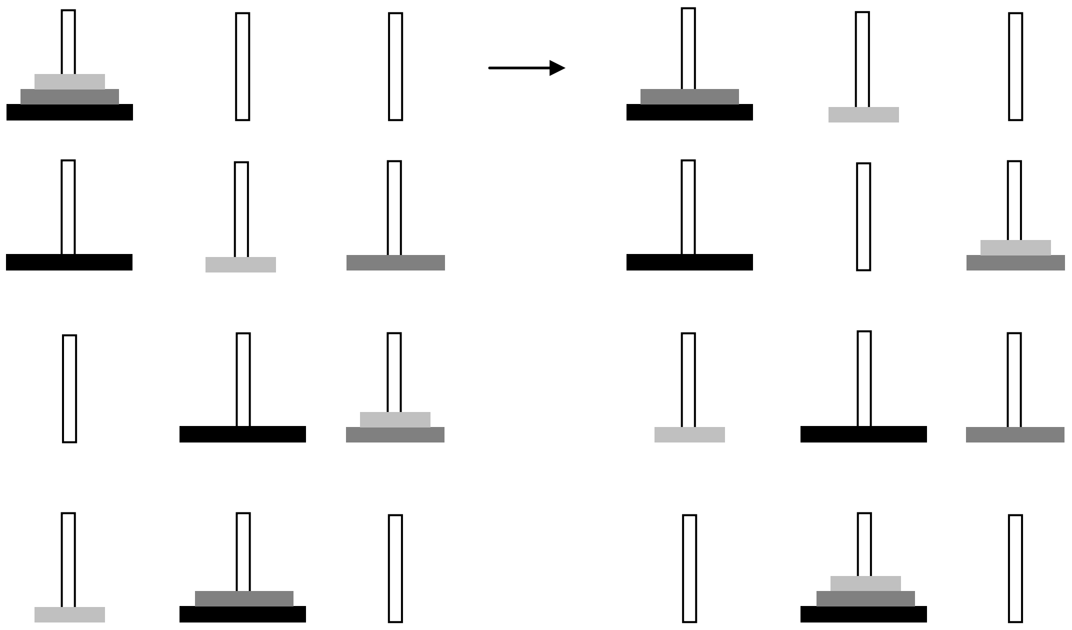
The following trivial program illustrates the use of `writePos()`:

```
class WritePos {

    static void writePos(int[] w, int i) {
        .....
    }

    public static void main(String args[]) {
        int[] w = {4, -2, -3, 2, -5, 9, 3, -7, 8, -6};
        writePos(w,0);
    }
}
```

The program displays `4 2 9 3 8`.

Example 3: towers of Hanoi

Towers of Hanoi is a game for one. We are given three poles and a number of discs, all of different diameters, with holes in their centre. All the discs sit on the left pole in order of



increasing size (downwards). The game consists in attempting to move all the discs to the middle pole, one disc at a time. Discs may only be in transit or resting on one of the three poles. The constraint that makes the game non-trivial is that at no stage may a disc rest on a smaller disc. A solution to the problem for three discs is illustrated. The outline program below generates a solution to the Towers of Hanoi problem for any number of discs, and shows a sample output for three discs (the missing details will be supplied shortly):

```
class Hanoi {
    static void moveHanoi(int n, String pA, String pB, String pC)  {
    // Move n discs Hanoi-style from pole pA to pole pB via pole pC
        .....
    }
    public static void main(String[] args) {
        moveHanoi(3, "Pole 1", "Pole 2", "Pole 3");
    }
}
```

```
Move top disc from Pole 1 to Pole 2
Move top disc from Pole 1 to Pole 3
Move top disc from Pole 2 to Pole 3
Move top disc from Pole 1 to Pole 2
Move top disc from Pole 3 to Pole 1
Move top disc from Pole 3 to Pole 2
Move top disc from Pole 1 to Pole 2
```

We will implement `moveHanoi()` recursively. First, we write down the specification. We expect that although initially `pA` and `pB` are empty, this state of affairs will not continue as discs get moved about. However, each pole will contain a Hanoi tower at all times. By *Hanoi tower* we mean a (possibly empty) stack of discs such that no disc rests on a smaller one. In order to move `n` discs from `pA` to `pB` via `pC`, it would be extremely helpful, if not essential, for the discs on `pA` to be smaller than those on `pB` and `pC`. This is true initially in a trivial way because `pB` and `pC` start off empty; as we move discs about we should see to it that it remains so. We write the header and specification:

```
static void moveHanoi(int n, String pA, String pB, String pC)
// Move the top n discs on pole pA to pole pB, one disc at a time. Poles pA,
// pB, and pC contain Hanoi towers initially,  and must do so at all times.
// Pole pA has (at least) n discs, n>= 0. The top n discs on pole pA
//  are smaller than the discs on poles pB and pC.
```

Although we are asked initially to move *all* discs from `pA`, it is possible that in some phases of the game we may want to move only some of them. We have anticipated that in wording the specification (we say `pA` has *at least* `n` discs, and that we are to move the *top n* discs).

The base case could not be simpler: if `n` is 0 there is no work whatsoever to be done.

For the recursive case, we can start by moving the top `n-1` discs from `pA` to `pC` using `pB` as a temporary. This can be achieved by a recursive call because `n-1` is less than `n` (in the sense of being closer to the base case). This will leave `pB` in its original state and hence still containing a Hanoi tower. Pole `pC` contains a Hanoi tower because it has only gained the top `n-1` discs of `pA` which we know are smaller than those originally on `pC`.

The preceding move exposes what was the `n`'th disc on `pA` and is now its top disc. We move it to its final resting place on `pB`. Pole `pB` continues to have a Hanoi tower because the top `n` discs originally on `pA`, in particular the `n`'th one, are smaller than the discs on `pB`.

It remains to move the `n-1` discs now on `pC` to `pB`. We need a temporary pole to help in this, and the only candidate is `pA`. We can use `pA` provided that the discs we might place on it are smaller than those already on `pA`. But that will indeed be the case because the added discs will be (at most) the top `n-1` discs currently on `pC`, and these were formerly the top `n-1` discs on `pA`. Moving the top `n-1` discs from `pC` to `pB` can be done recursively because `n-1` is less than `n`. The complete solution is now remarkably simple to express as a Java method:

```
static void moveHanoi(int n, String pA, String pB, String pC) {
    if (n>0) {
        moveHanoi(n-1, pA, pC, pB);
        System.out.println("Move top disc from " + pA + " to " + pB);
        moveHanoi(n-1, pC, pB, pA);
    }
}
```

3 When to use recursion

Recursion is an alternative to looping which happens to be very convenient for some problems (but not for all). If there is no compelling reason to use recursion it is better to use loops. Loops are usually easier to write, and are computationally somewhat cheaper. However, there are many problems in computing for which a recursive solution is natural and simpler than an iterative one. An example is the method above for writing the decimal digits of a natural number in words (as an exercise, you might try writing a non-recursive version), and the Towers of Hanoi problem. Another example, and a very important one, is a fast sorting algorithm called *Quicksort*.