28 Recursion

1 Recursive functions

Methods may employ a technique called *recursion*. We will explain it by writing a function to compute factorials. The *factorial* of a natural number *n* is defined as $1\times2\times3\times \ldots \times n$. We write *n*! to denote the factorial of *n*. For example, $3! = 1\times2\times3 = 6$, and $5! = 1\times2\times3\times4\times5 = 120$. As an aside we mention that *n*! is the number of ways of laying out *n* balls in a line where all the balls have different colours. The idea of 0! may not be intuitive, but we allow it nonetheless and define it to be 1. The defining properties of factorial are:

(i) $0! = 1$

(ii) $n! = n\times(n-1)!$ for $n>0$

For example, property (ii) with $n=5$ states that 5! equals 5×4!, which you will easily convince yourself is true. Properties (i) and (ii) are called *defining properties* because it turns out that they are sufficient to calculate *n*! for any natural n (assuming we know how to do subtraction and multiplication). To compute 3! for example:

3!
= "property (ii) with *n*=3"
3×2!
= "property (ii) with *n*=2"
3×(2×1!)
= "property (ii) with *n*=1"
3×(2×(1×0!))
= "property (i)"

$3\times(2\times(1\times1))$
= "multiply out"
6

Property (ii) is called a *recursive property*. By this is meant that the operation being defined on the left-hand side (here, factorial) occurs also on the right-hand side. A recursive property might at first sight seem to be circular, but this need not be so. All is well if the operation on the right hand side is applied to a smaller term than on the left hand side. That is indeed the case in (ii) which has factorial *n*-1 on the right-hand side, and factorial *n* on the left. The importance of the term being smaller is that it is tending towards a term (here, 0!) for which the operation is defined without recursion. This ensures that we can "unwind" the recursive property as often as it takes to arrive at a non-recursive case (also called a *base* case). The computation of 3! above illustrates this: we applied property (ii) in turn for 3!, 2!, and 1!, after which we arrived at 0! for which property (i) applies. After that it only remained to multiply out.

In summary, properties (i) and (ii) are defining properties of factorial because (a) there is a rule covering every natural argument (in fact, one for 0 and one for positives), and (b) in the recursive property (ii), the factorial on the right-hand side is applied to a smaller term than that on the left. All this might be no more than interesting mathematics but for the fact that such defining properties can be translated directly into Java methods. The following is a method to compute factorials:

```
static int fac(int n) { // factorial n,  n>=0
    if (n==0) return 1;
    else return (n*fac(n-1));
}
```

We say that `fac()` is *recursive* because its body contains an invocation of `fac()`. Let us check that `fac()` really does faithfully encode the defining properties of factorial. We see immediately from an inspection of its body that it satisfies the following:

(a) `fac(0) = 1`
(b) `fac(n) = n*fac(n-1)` when `n>0`

– there is no intelligence being applied in deducing (a) and (b), just a mechanical inspection of the text of `fac()`. Properties (a) and (b) are evidently faithful representations in code of (i) and (ii), respectively – just mentally replace `fac(0)` with 0!, `fac(n)` with *n*!, and `fac(n-1)` with (*n*-1)!. That's it! Having satisfied ourselves at the outset that properties (i) and (ii) suffice to compute *n*!, and now having checked that `fac()` faithfully encodes (i) and (ii), we can relax. The machine will do the rest (we will see how it does so shortly).

Example 1: the first few factorials

The following program prints a table of the factorials of 0, 1, 2, ... 9.

```
class Factorials {

    static int fac(int n) { // factorial n, n>=0
        if (n==0) return 1;
        else return n*fac(n-1);
    }

    public static void main(String[] args) {
        for (int k=0; k<10; k++) {
            System.out.println(k + "! = " + fac(k));
        }
    }
}
```

Example 2: celebrity numbers

A natural number is called a *celebrity* number if it is equal to the sum of the factorials of its decimal digits. 1 and 2 are trivial celebrity numbers (1!=1 and 2!=2) but a more interesting one is 145 (1!+4!+5! = 1+24+120 = 145). The following program finds all celebrity numbers up to one million (in fact celebrity numbers are rare – the program discovers just one more).

```
class Celebrity {

    static int fac(int n) { // factorial n, n>=0
        if (n==0) return 1;
        else return n*fac(n-1);
    }

    static boolean isCelebrity(int n) { // is n a celebrity number, n>0
        int total = 0; // running total of factorials of digits of n
        int highN = n; // the high end of n – its digits remain to be fac'd & added
        while (highN>0) {
            // extract rightmost digit from highN ...
            int digit = highN%10; highN = highN/10;
            // ... and add its factorial to running total
            total = total+fac(digit);
        }
        return (total==n);
    }

    public static void main(String[] args) {
        for (int k=1; k<1000000; k++) {
            if (isCelebrity(k))
                System.out.println(k + " is a celebrity number");
        }
    }
}
```

Recursion vs iteration

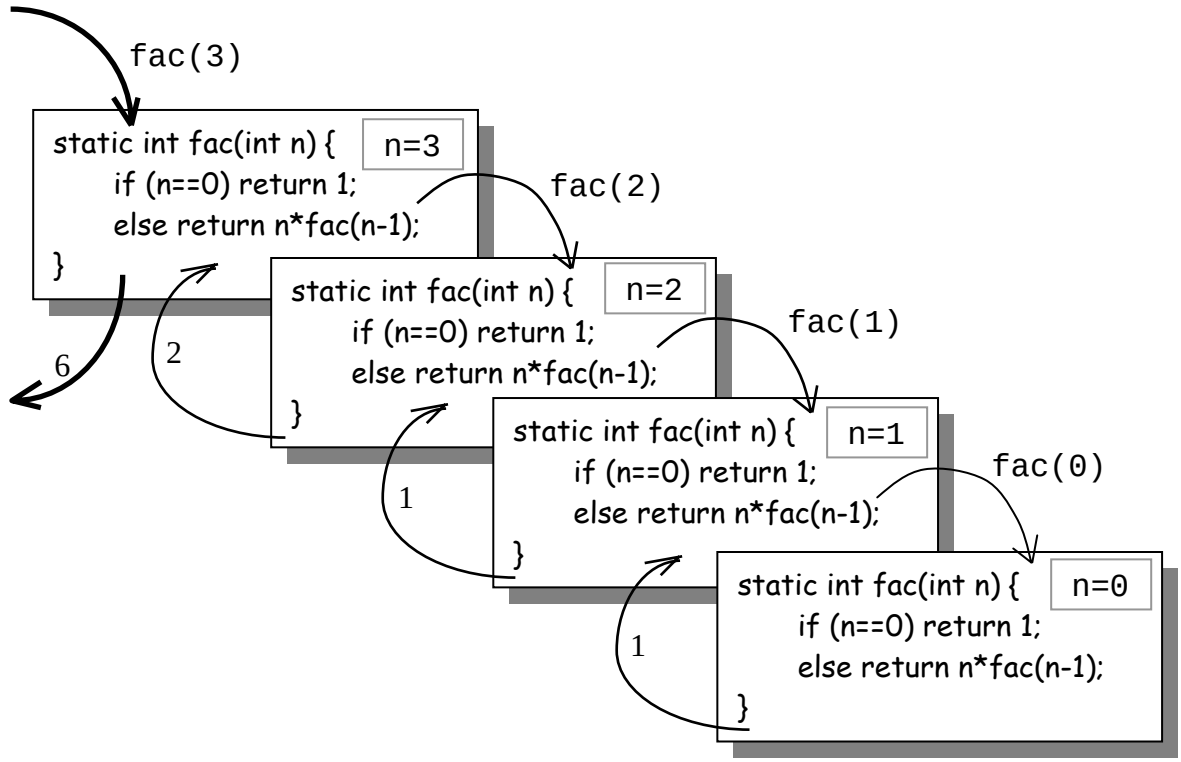
Recursion does not give us any extra power in the sense that it does not enable us to write programs that we could not have written without it, but it can be an elegant and simple way to write some methods. It is an alternative to loops. For example, `fac()` can be written iteratively (i.e. using loops rather than recursion) as follows:

```
static int fac(int n) {  // factorial n,  n>=0
    int z = 1;
    for (int i=1; i<=n; i++)
        z = z*i;
    return z;
}
```

Recursion applies equally to static and dynamic methods.

2 Recursion operationally OPTIONAL

How does the machine implement recursion? Examining the body of a recursive method would seem to suggest that the method calls itself, but that is not the way to look at it. A recursive method does not literally call itself, but rather the system makes a clone of the method and calls the clone. This is the key to understanding recursion operationally: each time the execution of a method encounters a recursive call on the method, it makes a clone and calls the clone. The



clone is identical in all respects to the parent. If the clone in turn encounters a recursive call (which is very likely) it makes a new clone and calls it. Eventually (if the method is properly written) a clone will compute its result without meeting a recursive invocation, and so without cloning, and the answer will be returned to the parent. The parent can then complete its work and pass the result back to *its* parent, and so on. Once a clone completes its work and returns to its parent, the clone dies and all memory it occupied is released. For example, suppose the statement `System.out.print(fac(3))` is executed. This will cause the system to invoke `fac(3)` whose execution can be pictured in the top box above.

- The first box contains the `fac()` that is invoked as a result of executing `System.out.print(fac(3))`. Parameter `n` in `fac()` is given the initial value 3. Therefore the else-branch of the if-statement is taken resulting in an invocation of `fac(n-1)`, i.e. `fac(2)`.
- This causes a clone to be created – the one in the second box – in which parameter `n` has value 2. This looks exactly like its parent and even has the same name. However, it is a different method. In particular, its parameter `n` is a different variable from parameter `n` in the parent. In the parent, `n` has the value 3 at all times, whereas in the clone `n` has the value 2 at all times. The parent continues to exist while the clone does its job, but in suspended animation. The clone takes the else-branch resulting in an invocation of `fac(1)`.
- This causes another clone to be created – the one in the third box. In this clone parameter `n` has the value 1. The original and two clones now exist simultaneously, but the parent and the first clone are suspended. This newest clone takes the else-branch resulting in an invocation of `fac(0)`.
- This causes the clone in the fourth box to be created. Now the parent and three clones exist. In the latest clone, parameter `n` has the value 0.
- The newest clone immediately returns the value 1 to its invoker (clone 2) and dies, releasing the space it occupied.
- Clone 2 (in which `n` has value 1) now computes `n*1` and returns the result 1 to its caller (clone 1) and dies. The space it occupied is released.
- Clone 1 (in which `n` has value 2) now computes `n*1` and returns the result 2 to its caller (the original) and dies. The space it occupied is released.
- The original (in which `n` has value 3) computes `n*2` and returns the result 6 to the `System.out.println()` statement that invoked it, and so 6 is printed.

3 Ensuring termination

It is easy to write a recursive function that never terminates, for example:

```
static int silly(int n) {
  return silly(n);
}
```

Actually, if `silly(3)`, say, is invoked, it will eventually terminate abnormally with a message from the system to the effect that it has run out of memory — because each recursive call consumes memory in the machine until there is no free memory left. Of course, we would never deliberately write a non-terminating recursive function, but we may do so inadvertently and it is wise to double-check. A recursive method is guaranteed to terminate if it satisfies the following criteria:

(i) The initial call and every recursive call is applied to arguments that are within the design range of the method.

(ii) The result is computed for the smallest arguments (the *base* cases) without recourse to recursion.

(iii) Every argument of a recursive call is by some measure smaller than the incoming argument. By "smaller" is meant that the argument is at least one step nearer a base case.

As an example, we show that `fac()` meets the three criteria and so we can be satisfied that it terminates. As regards design range, `fac()` is designed on the assumption that the argument `n` satisfies $n \geq 0$.

(i) The argument in the recursive call is `n-1` and so we have to be sure that at the point of call $n-1 \geq 0$. This is indeed so because by assumption $n \geq 0$, and the else-branch of the if-statement is taken ensuring $n > 0$ at the point where the call `fac(n-1)` occurs.

As regards base cases:

(ii) `fac(0)` is computed without recourse to recursion.

For the recursive case, we take as our measure of the argument the value of `n` itself:

(iii) We observe `n-1` is smaller (i.e. closer to the base case 0) than the incoming `n` (which we know is positive at the point of the recursive call).

It follows that `fac(n)` terminates provided `n` satisfies $n \geq 0$. For negative `n`, `fac()` does not terminate, but it was not designed to handle that case.

Banana skin: recursion in a loop

Some beginners apparently get so concerned about ensuring termination that they seek extra insurance by placing recursive calls inside a loop! The following is typical:

```
static int fac(int n) { // factorial of n, n>=0
    if (n==0) return 1;
    else {
        while (n>0) { // Wrong!
            return (n* fac(n-1));
        }
    }
}
```

This is wrong! Loops rarely occur in recursive methods (at least in the simpler ones programmers are likely to meet in everyday programming), and it is exceedingly rare for a recursive call to occur inside the body of a loop (as above). If you find yourself writing a loop in the body of a recursive method, you may well be going down the wrong track, so think again and make sure that that is really what you want to do. Chances are it isn't.

4 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

$m^n = m \times m^{n-1}$ 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
```

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
    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.

5 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
        String[] digit = {  "zero", "one", "two", "three", "four",
                            "five", "six", "seven", "eight", "nine"};
        if (n<10) System.out.print(digit[n] + " ");
        else {
            putVerbal(n/10);
            putVerbal(n%10);
        }
    }

    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 `System.out.print(digit[n%10]+" ")` and it is computationally a little cheaper because it eliminates a method invocation.

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` because these are the positives in -5, 9, 3, -7, 8, -6. `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);
```

```
    }
}
```

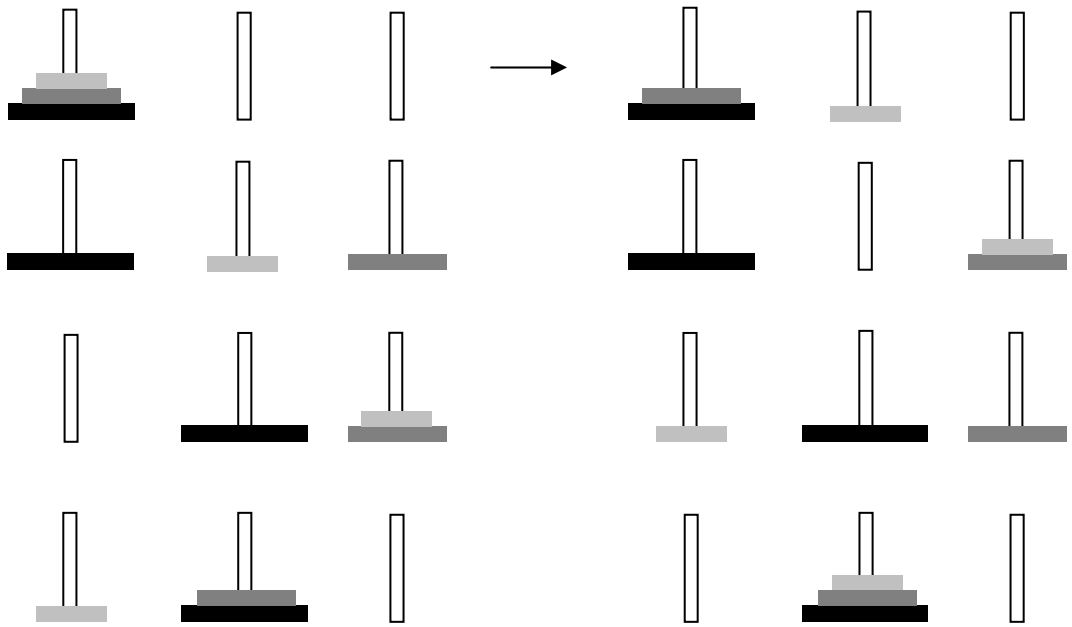
Notice that the base case requires no code (the if-statement has no else-branch). The following trivial program illustrates the use of `writePos()`:

```
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);
    }
}
```

6 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*.