3.2 Recursion

This section introduces recursive functions, functions that directly or indirectly call themselves. You will see that recursive functions are very natural in many situations, and that they lead to compact and readable code close to mathematical function definitions. We will also explain how recursive function calls are processed, and how recursion can (in principle) be replaced with iteration. In the end, you will see two applications (sorting, and drawing fractals) that demonstrate the power or recursion.

3.2.1 A warm-up

Many mathematical functions are naturally defined *recursively*, meaning that the function to be defined appears in its own definition. For example, for any $n \in \mathbb{N}$, the number $n!$ can recursively be defined as follows.

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

In C++ we can also have recursive functions: a function may call itself. This is nothing exotic, since after all, a function call is just an expression that can in principle appear anywhere in the function's scope, and that scope includes the function body. Here is a recursive function for computing $n!$; in fact, this definition exactly matches the mathematical definition from above.

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Here, the expression `fac(n-1)` is a *recursive call* of `fac`.

Infinite recursion. With recursive functions, we have the same issue as with loops (Section 2.4.2): it is easy to write down function calls whose evaluation does not terminate. Here is the shortest way of creating an infinite recursion: define the function

```
void f()
{
  f();
}
```

with no arguments and evaluate the expression `f()`. The reason for non-termination is clear: the evaluation of `f()` consists of an evaluation of `f()` which consists of an evaluation of `f()` which... you get the picture.

Like for loops, the function definition has to make sure that progress towards termination is made in every function call. For the function `fac` above, this is the case: each time `fac` is called recursively, the value of the call argument becomes smaller, and when the value reaches 1, no more recursive calls are performed: we say that the recursion "bottoms out".

3.2.2 The call stack

Let's try to understand what exactly happens during the evaluation of `fac(3)`, say. The formal argument `n` is initialized with 3, and since this is greater than 1, the statement `return n * fac(n-1);` is executed next. This first evaluates the expression `n * fac(n-1)` and in particular the right operand `fac(n-1)`. Since `n-1` has value 2, the formal argument `n` is therefore initialized with 2.

But wait: what is "the" formal argument? Automatic storage duration implies that each function call has its "own" fresh instance of the formal argument, and the lifetime of this instance is the respective function call. In evaluating `f(n-1)`, we therefore get a new instance of the formal argument `n`, on top of the previous instance from the call `f(3)` (that has not yet terminated). But which instance of `n` do we use in the evaluation of `f(n-1)`? Quite naturally, it will be the new one, the one that "belongs" to the call `f(n-1)`. This rule is in line with the general scope rules from Section 2.4.3: the relevant declaration is always the most recent one that is still visible.

The technical realization of this is very simple. Everytime a function is called, the call argument is evaluated, and the resulting value is put on the *call stack* which is simply a region in the computer's memory.[5]

Like a stack of papers on your desk, the call stack has the property that the object that came last is "on top". Upon termination of a function call, the top object is taken off the stack again. Whenever a function call accesses or changes its formal argument, it does so by accessing or changing the corresponding object on top of the stack.

This has all the properties we want: every function call works with its own instance of the formal argument; when it calls another function (or the function itself recursively), this instance becomes temporarily hidden, until the nested call has terminated. At that point, the instance reappears on top of the stack and allows the original function call to work with it again.

Table 5 shows how this looks like for `f(3)`, assuming that the right operand of the multiplication operator is always evaluated first. Putting an object on the stack "pushes" it, and taking the top object of "pops" it.

Because of the call stack, infinite recursions do not only consume time but also

[5] if the function has several arguments, several values are put on the call stack; to keep the description simple, we concentrate on the case of one argument.

call stack (bottom $\longleftrightarrow$ top)	evaluation sequence	action
···	`fac(3)`	push 3
··· n: 3	`n * fac(n-1)`	
··· n: 3	`n * fac(2)`	push 2
··· n: 3 n: 2	`n * (n * fac(n-1))`	
··· n: 3 n: 2	`n * (n * fac(1))`	push 1
··· n: 3 n: 2 n: 1	`n * (n * 1)`	pop
··· n: 3 n: 2	`n * (2 * 1)`	
··· n: 3 n: 2	`n * 2`	pop
··· n: 3	`3 * 2`	
··· n: 3	`6`	pop
···		

Table 5: *The call stack, and how it evolves during an evaluation of* `fac(3)`*; the respective value of* `n` *to use is always the one on top*

memory. Unlike infinite loops, they usually lead to a program abortion as soon as the memory reserved for the call stack is full.

3.2.3 Basic practice

Let us consider two more simple recursive functions that are somewhat more interesting than `fac`. They show that recursive functions are particularly amenable to correctness proofs of their postconditions, and this makes them attractive. On the other hand, we also see that it is easy to write innocent-looking recursive functions that are very inefficient to evaluate.

Greatest common divisor. Consider the problem of finding the greatest common divisor $\gcd(a, b)$ of two natural numbers a, b. This is defined as the largest natural number that divides both a and b without remainder. In particular, $\gcd(n, 0) = \gcd(0, n) = n$ for $n > 0$; let us also define $\gcd(0, 0) := 0$.

The *Euclidean algorithm* finds $\gcd(a, b)$, based on the following

Lemma 1 *If* $b > 0$*, then*

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

Proof. Let k be a divisor of b. From

$$a = (a \operatorname{div} b)b + a \bmod b$$

it follows that

$$\frac{a}{k} = (a \operatorname{div} b)\frac{b}{k} + \frac{a \bmod b}{k}.$$

Since $a \operatorname{div} b$ and b/k are integers, we get

$$\frac{a \bmod b}{k} \in \mathbb{N} \quad \Leftrightarrow \quad \frac{a}{k} \in \mathbb{N}.$$

In words, if k is a divisor of b, then k divides a if and only if k divides $a \bmod b$. This means, the divisors of a *and* b are exactly the divisors of b *and* $a \bmod b$. This proves that $\gcd(a, b)$ and $\gcd(b, a \bmod b)$ are equal. □

Here is the corresponding C++ function for computing the greatest common divisor of two `unsigned int` values, according to the Euclidean algorithm.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

The Euclidean algorithm is very fast. We can easily call it for any `unsigned int` values on our platform, without noticing any delay in the evaluation.

Correctness and termination. For recursive functions, it is often very easy to prove that the postcondition is correct, by using the underlying mathematical definition directly (like $n!$ for `fac`), or by using some facts that follow from the mathematical definition (like Lemma 1 for `gcd`).

The correctness proof must involve a termination proof, so let's start with this: any call to `gcd` terminates, since the value b of the second argument is bounded from below by 0 and gets smaller in every recursive call (we have $a \bmod b < b$).

Given this, the correctness of the postcondition follows from Lemma 1 by induction on b. For $b = 0$, this is clear. For $b > 0$, we inductively assume that the postcondition is correct for all calls to `gcd` where the second argument has value $b' < b$. Since $b' = a \bmod b$ satisfies $b' < b$, we may assume that the call `gcd(b, a % b)` correctly returns $\gcd(b, a \bmod b)$. But by the lemma, $\gcd(b, a \bmod b) = \gcd(a, b)$, so the statement

```
return gcd(b, a % b);
```

correctly returns $\gcd(a, b)$.

Fibonacci numbers. The sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$ of *Fibonacci numbers* is one of the most famous sequences in mathematics. Formally, the sequence is defined as

follows.

$$\begin{aligned} F_0 &:= 0, \\ F_1 &:= 1, \\ F_n &:= F_{n-1} + F_{n-2}, \quad n > 1. \end{aligned}$$

This means, every element of the sequence is the sum of the two previous ones. From this definition, we can immediately write down a recursive C++ function for computing Fibonacci numbers, getting termination and correctness for free.

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib (unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

If you write a program to compute the Fibonacci number F_n using this function, you will notice that somewhere between $n = 30$ and $n = 50$, the program becomes very slow. You even notice how much slower it becomes when you increase n by just 1.

The reason is that the mathematical definition of F_n does not lead to an efficient algorithm, since all values $F_i, i < n-1$, are repeatedly computed, some of them extremely often. You can for example check that the call to `fib(50)` computes F_{48} already twice (once directly in `fib(n-2)`, and once indirectly from `fib(n-1)`. F_{47} is computed three times, F_{46} five times, and F_{45} eight times (do you see a pattern?).

3.2.4 Recursion versus iteration

From a strictly functional point of view, recursion is superfluous, since it can be simulated through iteration (and a call stack explicitly maintained by the program; we could simulate the call stack with an array). We don't have the means to prove this here, but we want to show it for the recursive functions that we have seen in the previous section.

The function `gcd` is very easy to write iteratively, since it is *tail-end recursive*. This means that there is only one recursive call, and that one appears at the very end of the function body. Tail-end recursion can be replaced by a simple loop that iteratively updates the formal arguments until the termination condition is satisfied. In the case of `gcd`, this update corresponds to the transformation $(a, b) \to (b, a \bmod b)$.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
```

```
  }
  return a;
}
```

You see that we get longer and less readable code, and that we need an extra variable to remember the previous value of a before the update step; in the spirit of Section 2.4.8, we should therefore use the original recursive formulation.

Our function fib for computing Fibonacci numbers is not tail-end recursive, but it is still easy to write it iteratively. Remember that F_n is the sum of F_{n-1} and F_{n-2}. We can therefore write a loop whose iteration i computes F_i from the previously computed values F_{i-2} and F_{i-1} that we maintain in the variables a and b.

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;  // F_{i-2}
    a = b;                    // F_{i-1}
    b += a_prev;              // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

Again, this non-recursive version fib2 is substantially longer and more difficult to understand than fib, but this time there is a benefit: fib2 is much faster, since it computes every number $F_i, i \leq n$ *exactly once*. While we would grow old in waiting for the call fib(50) to terminate, fib2(50) gives us the answer in no time. (Unfortunately, this answer may be incorrect, since F_{50} could exceed the value range of the type unsigned int.)

In this case we would prefer fib2 over fib, simply since fib is too inefficient for practical use. The more complicated function definition of fib2 is a moderate price to pay for the speedup that we get.

3.2.5 Primitive recursion

Roughly speaking, a mathematical function is *primitive recursive* if it can be written as a C++ function f in such a way that f neither directly nor indirectly calls itself with call arguments depending on f. For example,

```
unsigned int f (unsigned int n)
{
  if (n == 0) return 1;
```

```
  return f(f(n-1) - 1);
}
```

is not allowed, since `f` recursively calls itself with a argument depending of `f`. This does *not* mean that the underlying mathematical function is not primitive recursive, it just means that we have taken the wrong C++ function. Indeed, the above `f` implements the mathematical function satisfying $f(n) = 1$ for all n, and this function is obviously primitive recursive.

In the early 20-th century, it was believed that the functions whose values can in principle be computed by a machine are exactly the primitive recursive ones. Indeed, the function values one computes in practice (including $\gcd(a, b)$ and F_n) come from primitive recursive functions.

It later turned out that there are computable functions that are not primitive recursive. A simple and well-known example is the binary *Ackermann function* $A(m, n)$, defined by

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

The fact that this function is not primitive recursive requires a proof (that we don't give here). As already noted above, it is necessary but not sufficient that this definition recursively uses A with a argument that depends on A.

It may not be immediately clear that the corresponding C++ function

```
// POST: return value is the Ackermann function value A(m,n)
unsigned int A (unsigned int m, unsigned int n) {
  if (m == 0) return n+1;
  if (n == 0) return A(m-1,1);
  return A(m-1, A(m, n-1));
}
```

always terminates, but Exercise 83 asks you to show this. Table 6 lists some Ackermann function values. For $m \leq 3$, $A(m, n)$ looks quite moderate, but starting from $m = 4$, the values get extremely large. You can still compute $A(4, 1)$, although this takes surprisingly long already. You *might* be able to compute $A(4, 2)$; after all, $2^{65536}-3$ has "only" around $20,000$ decimal digits. But the call to `A(4,3)` will not terminate within any observable period.

It can in fact be shown that $A(n, n)$ grows faster than any primitive recursive function in n (and this is a proof that A cannot be primitive recursive). Recursion is a powerful but also dangerous tool, since it is easy to encode (too) complicated computations with very few lines of code.

3.2.6 Sorting

Sorting a sequence of values (numbers, texts, etc.) into ascending order is a very basic and important operation. For example, a specific value can be found much faster in a

m \ n	0	1	2	3	⋯	n
0	1	2	3	4	⋯	$n+1$
1	2	3	4	5	⋯	$n+2$
2	3	5	7	9	⋯	$2n+3$
3	5	13	29	61	⋯	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	⋯	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3}-3$

Table 6: *Some values of Ackermann's function*

sorted than in an unsorted sequence (see Exercise 88). You know this from daily life, and that's why you sort your CDs, and why the entries in a telephone directory are sorted by name.

We have asked you in Exercise 60 to write a program that sorts a given sequence of integers; Exercise 77 was about making this into a function that sorts all numbers described by a given pointer range. In both exercises, you were not supposed to do any efficiency considerations.

Here we want to catch up on this and investigate the *complexity* of the sorting problem. Roughly speaking, the complexity of a problem is defined as the complexity (runtime) of the fastest algorithm that solves the problem. In computing Fibonacci numbers in Section 3.2.3 and Section 3.2.4, we have already seen that the runtimes of different algorithms for the same problem may vary a lot. The same is true for sorting algorithms, as we will discover shortly.

Let us start by analyzing one of the "obvious" sorting algorithms that you may have come up with in Exercise 60. The simplest one that the authors can think of is *minimum-sort*. Given the sequence of values (let's assume they are integers), *minimum-sort* first finds the smallest element of the sequence; then it interchanges this element with the first element. The sequence now starts with the smallest element, as desired, but the remainder of the sequence still needs to be sorted. But this is done in the same way: the smallest element among the remaining ones is found and interchanged with the *second* element of the sequence, and so on.

Assuming that the sequence is described by a pointer range `[first, last)`, *minimum-sort* can be realized as follows.

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in ascending order
void minimum_sort (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer to current element
```

```
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

The standard library function `std::iter_swap` interchanges the values of the objects pointed to by its two arguments. There is also a function `std::min_element` that we could use to get rid of the inner loop; however, since we want to analyze the function `minimum_sort` in detail, we refrain from calling any nontrivial standard library function here.

What can we say about the runtime of `minimum_sort` for a given range? That it depends on the platform, this is for sure. On a modern PC, the algorithm will run much faster than on a vintage computer from the twentieth century. There is no such thing as "the" runtime. But if we look at what the algorithm does, we can find a measure of runtime that is platform-independent.

A dominating operation in the sense that it occurs very frequently during a call to `minimum_sort` is the comparison `*q < *p_min`. We can even exactly count the number of such comparisons, depending on the number of elements n that are to be sorted. In the first execution of the `while` statement, the first element is compared with all $n-1$ succeeding elements. In the second execution, the second element in compared with all the $n-2$ succeeding elements, and so on. In the second-to-last execution of the `while` statement, finally, we have one comparison, and that's it. We therefore have the following

Observation 1 *The function* `minimum_sort` *sorts a sequence of* n *elements with*

$$1 + 2 + \ldots n - 1 = \frac{n(n-1)}{2}$$

comparisons between sequence elements.

Why do we specifically count these comparisons? Because any other operation is either performed much less frequently (for example, the declaration statement `int* q = p` is executed only n times), or with approximately the same frequency. This concerns the assignment `p_min = q` which may happen up to $n(n-1)/2$ times, and the expression `++q != last`; this one is evaluated even more frequently, namely $n(n-1)/2 + n$ times. The total number of operations is therefore at most $c_1 n(n-1)/2 + c_2 n$ for some constants c_1, c_2. For large n, the linear term $c_2 n$ is negligible compared to the quadratic term $c_1 n(n-1)/2$; we can therefore conclude that the total number of operations needed to sort n numbers is proportional to the number of comparisons between sequence elements.

This implies the following: if you measure the runtime of the whole sorting algorithm, the resulting time T_{total} will be proportional to the time T_{comp} that is being spent with comparisons between sequence elements.[6] Since T_{comp} is in turn proportional to the

[6] Due to the effects of caching and other add-ons to the von-Neumann architecture, this is not necessarily true on your platform.

number of comparisons itself, this number is a good indicator for the efficiency of the algorithm.

If you think about sorting more complicated values (like names in a telephone directory), a comparison between two elements might even become the single most time-consuming operation. In such a scenario, T_{comp} may eat up almost everything of T_{total}, making the comparison count an even more appropriate measure of efficiency.

To check that all this is not only grey theory, let us make some experiments and measure the time that it takes to execute a program with the following `main` function, for various values of n. As our "test case", we use the jumbled sequence $0, n-1, 1, n-2, ...$, and after having called the function `minimum_sort` from above, we check whether we now indeed have the ascending sequence $0, 1, \ldots, n-1$. Yes, this program does other things apart from the actual sorting, but all additional operations are "cheap" in the sense that their number is proportional to n at most; according to our above line of arguments, they should therefore not matter.

```
int main()
{
  int n = 100000; // number of values to be sorted
  int* a = new int[n];

  std::cout << "Sorting " << n << " integers...\n";

  // create sequence: 0, n-1, 1, n-2,...
  for (int i=0; i<n; ++i)
    if (i % 2 == 0) a[i] = i/2; else a[i] = n-1-i/2;

  // sort into ascending order
  minimum_sort (a, a+n);

  // is it really sorted ?
  for (int i=0; i<n-1;++i)
    if (a[i] != i) std::cout << "Sorting error!\n";

  delete[] a;

  return 0;
}
```

Table 7 summarizes the results. For every value of n, **Gcomp** is the number of Gigacomparisons (10^9 comparisons), according to Observation 1. In other words, **Gcomp**$= 10^{-9}n(n-1)/2$. **Time** is the absolute runtime of the program in minutes and seconds, on a modern PC. **sec/Gcomp** is **Time** (in seconds) divided by **Gcomp** and tells us how many seconds the program needs to perform one Gigacomparison.

The table shows that the number of seconds per Gigacomparison is around 3.4 for all considered values of n. As predicted above, the runtime in practice is therefore indeed

n	100,000	200,000	400,000	800,000	1,600,000
Gcomp	5	20	80	320	1280
Time (min)	0:17	1:07	4:24	18:06	73:32
sec/Gcomp	3.4	3.35	3.3	3.4	3.45

Table 7: *Number of comparisons and runtime of* minimum-sort

proportional to the number of comparisons between sequence elements. This number quadruples from one column to the next, and so does the runtime.

We also see that sorting numbers using *minimum-sort* appears to be pretty inefficient. 1,600,000 is not large by today's standards, but to sort that many numbers takes more than one hour! Given that **sec/Gcomp** appears to be constant, we can even estimate the time that it would take to sort 10,000,000 numbers, say. For this, we derive from Observation 1 the required number of Gigacomparisons (50,000) and multiply it with 3.4. The resulting 170,000 seconds are almost two days.

Essentially the same figures result from running other well-known simple sorting algorithms like *bubble-sort* or *insert-sort*. Can we do better? Yes, we can, and recursion helps us to do it!

Merge-sort. The paradigm behind the *merge-sort* algorithm is this: if a problem is (too) large to be solved directly, subdivide it into smaller subproblems that are easier to solve, and then put the overall solution together from the solutions of the subproblems. This paradigm is known as *divide and conquer*.

Here is how this works for sorting. Let us imagine that the numbers to be sorted come as a deck of cards, with the numbers written on them. The first step is to partition the deck into two smaller decks of half the size each. These two decks are then sorted independently from each other, with the same method; but the main ingredient of this method comes only now: we have to merge the two sorted decks into one sorted deck. But this is not hard: we put the two decks in front of us (both now have their smallest card on top); as long as there are still cards in one or both of the decks, the smaller of the two top cards (or the single remaining top card) is taken off and put upside down on a new deck that in the end represents the result of the overall sorting process. Figure 18 visualizes the merge step.

Here is how *merge-sort* can be realized in C++, assuming that we have a function `merge` that performs the above operation of merging two sorted sequences into one sorted sequence.

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in ascending order
void merge_sort (int* first, int* last)
{
  int n = last - first;
  if (n <= 1) return;           // nothing to do
  int* middle = first + n/2;
```

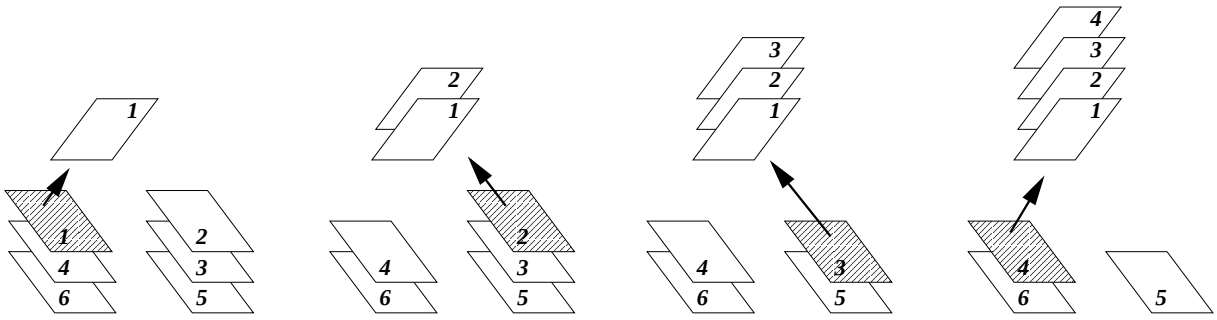


Figure 18: *Merging two sorted decks of cards into one sorted deck*

```
  merge_sort (first, middle);  // sort first half
  merge_sort (middle, last);   // sort second half
  merge (first, middle, last); // merge both halfs
}
```

If there is more than one element to sort, the function splits the range `[first, last)` into two ranges `[first, middle)` and `[middle, last)` of lengths $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Just as a reminder, for any real number x, $\lceil x \rceil$ is the smallest integer greater or equal to x ("x rounded up"), and $\lfloor x \rfloor$ is the largest integer smaller or equal to x ("x rounded down"). If n is even, both values $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ are equal to $n/2$, and otherwise, the first value is smaller by one.

As its next step, the algorithm recursively sorts the elements described by both ranges. In the end, it calls the function `merge` on the two ranges. In commenting the latter function, we stick to the deck analogy that we have used above. If you have understood the deck merging process, you will perceive the definition of `merge` as being straightforward.

```
// PRE: [first, middle), [middle, last) are valid ranges; in
//      both of them, the elements are in ascending order
void merge (int* first, int* middle, int* last)
{
  int n = last - first;   // total number of cards
  int* deck = new int[n]; // new deck to be built

  int* left = first;   // top card of left deck
  int* right = middle; // top card of right deck
  for (int* d = deck; d != deck + n; ++d)
    // put next card onto new deck
    if      (left == middle) *d = *right++; // left deck is empty
    else if (right == last)  *d = *left++;  // right deck is empty
    else if (*left < *right) *d = *left++;  // smaller top card left
    else                     *d = *right++; // smaller top card right

  // copy new deck back into [first, last)
```

```
  int *d = deck;
  while (first != middle) *first++ = *d++;
  while (middle != last) *middle++ = *d++;

  delete[] deck;
}
```

Analyzing merge-sort. As for *minimum-sort*, we will count the number of comparisons between sequence elements that occur when a sequence of n numbers is being sorted. Again, we can argue that the total number of operations is proportional to this number of comparisons. For *merge-sort*, this fact is not so immediate, though, and we don't expect you to understand it now. But for the benefit of (not only) the sceptic reader, we will check this fact experimentally below, as we did for *minimum-sort*.

All the comparisons take place during the calls to the function `merge` at the various levels of recursion, so let us first count the number of comparisons between sequence elements that one call to `merge` performs in order to create a sorted deck of n cards from two sorted decks.

It is apparent from the function body (and also from our informal description of the merging process above) that *at most one* comparison is needed for every card that is put on the new deck. Indeed, we may have to compare the two top cards of the left and the right deck in order to find out which card to take off next. But if one of the two decks becomes empty (this situation definitely occurs before the last card is put on the new deck), we don't do any further comparisons. This means that *at most* $n-1$ comparisons between sequence elements are performed in merging two sorted decks into one sorted deck with n cards.

Knowing this, we can now prove our main result.

Theorem 2 *The function* `merge_sort` *sorts a sequence of* $n \geq 1$ *elements with at most*

$$(n-1)\lceil \log_2 n \rceil$$

comparisons between sequence elements.

Proof. We define $T(n)$ to be the *maximum* possible number of comparisons between sequence elements that can occur during a call to `merge_sort` with an argument range of length n. For example, $T(0) = T(1) = 0$, since for ranges of lengths 0 and 1, no comparisons are made. We also get $T(2) = 1$, since for a range of length 2, *merge-sort* performs one comparison (in merging two sorted decks of one card each into one sorted deck of two cards). In a similar way, we can convince ourselves that $T(3) = 2$. There *are* sequences of length 3 for which one comparison suffices (the first card may be taken off the left deck which consists only of one card), but the maximum number that defines $T(3)$ is 2.

For general $n \geq 2$, we have the following recurrence relation:

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1. \tag{3.1}$$

n	100,000	200,000	400,000	800,000	1,600,000
Mcomp	1.7	3.6	7.6	16	33.6
Time (sec)	0.75	1.29	1.96	3.20	5.36
sec/Gcomp	441	358	257	200	160

Table 8: *Number of comparisons and runtime of* merge-sort

To see this, let us consider a sequence of n elements that actually requires the maximum number of $T(n)$ comparisons. This number of comparisons is the sum of the respective numbers in sorting the left and the right half, plus the number of comparisons during the merge step. The former two numbers are (by construction of `merge_sort` and definition of T) at most $T(\lfloor n/2 \rfloor)$ and $T(\lceil n/2 \rceil)$, while the latter number is at most $n-1$ by our previous considerations regarding `merge`. It follows that $T(n)$, the actual number of comparisons, is bounded by the sum of all three numbers.

Now we can prove the actual statement of the theorem. Since the *merge-sort* algorithm is recursive, it is natural that the proof is inductive. For $n = 1$, we have $T(1) = 0 = (1-1)\lceil \log_2 2 \rceil$, so the statement holds for $n = 1$.

For $n \geq 2$, let us assume that the statement of the theorem holds for *all* values in $\{1, \ldots, n-1\}$ (this is the inductive hypothesis). From this hypothesis, we need to derive the validity of the statement for the number n itself (note that $\lfloor n/2 \rfloor, \lceil n/2 \rceil \geq 1$). This goes as follows.

$$
\begin{aligned}
T(n) &\leq T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + n - 1 \quad \text{(Equation 3.1))} \\
&\leq (\lfloor \tfrac{n}{2} \rfloor - 1)\lceil \log_2 \lfloor \tfrac{n}{2} \rfloor \rceil + (\lceil \tfrac{n}{2} \rceil - 1)\lceil \log_2 \lceil \tfrac{n}{2} \rceil \rceil + n - 1 \quad \text{(inductive hypothesis)} \\
&\leq (\lfloor \tfrac{n}{2} \rfloor - 1)(\lceil \log_2 n \rceil - 1) + (\lceil \tfrac{n}{2} \rceil - 1)(\lceil \log_2 n \rceil - 1) + n - 1 \quad \text{(Exercise 89)} \\
&= (n-2)(\lceil \log_2 n \rceil - 1) + n - 1 \quad (\, n = \lfloor \tfrac{n}{2} \rfloor + \lceil \tfrac{n}{2} \rceil) \\
&\leq (n-1)(\lceil \log_2 n \rceil - 1) + n - 1 \\
&= (n-1)\lceil \log_2 n \rceil .
\end{aligned}
$$

□

As for *min-sort*, let us conclude with some experiments to check whether the number of comparisons between sequence elements is indeed a good indicator for the runtime in practice. The results in Table 8 look very different from the ones in Table 7.

Since `merge_sort` incurs much less comparisons than `minimum_sort`, our unit here is just **Mcomp**, the number of Megacomparisons (10^6 comparisons), according to Theorem 2. In other words, **Mcomp**$= 10^{-6}(n-1)\lceil \log_2 n \rceil$. **Time** is the absolute runtime of the program, this time in seconds and not minutes. But as in Table 7, **sec/Gcomp** tells us how many seconds the program needs to perform one Gigacomparison.

We first observe that this latter number *decreases* with n, where the rate of decrease becomes smaller and smaller. On our platform, we can go up to roughly $n = 51,200,000$ and find that **sec/Gcomp** continues like this in Table 8: 154, 146, 135, 128, 124.

This seems to indicate that the runtime is proportional to the number of comparisons only for very large n. If you think about it, this is not surprising. Cheap operations that are performed n times, say, eat up a much higher fraction of the total runtime when n is small. This is because n is relatively large compared to the upper bound of $(n-1)\lceil \log_2 n \rceil$ on the number of comparisons between sequence elements. But since we ignore the cheap operations in our comparison count, this count is too optimistic for small n. Only as n becomes very large, the ratio between n and $(n-1)\lceil \log_2 n \rceil$ becomes negligible and we start to see the predicted proportionality.

For *minimum-sort*, this phenomenon does not show since n is negligible compared to $n(n-1)/2$ already for small n.

The most positive news of Table 8 is that `merge_sort` is actually a practical sorting algorithm. While it takes `minimum-sort` more than two hours to process $1,600,000$ numbers, `merge_sort` does the same in around 5 seconds. This is mainly due to the fact that $(n-1)\lceil \log_2 n \rceil$ is a much smaller number than $n(n-1)/2$, the number of comparisons needed by `minimum_sort` (that's why we switched from Gcomp to Mcomp).

On the other hand, the time needed by `merge_sort` per Gcomp is dramatically higher than in `minimum-sort`; for $n = 1,600,000$, we observe a factor of around 50. It may be surprising that the factor is this large, but the fact that it *is* larger can be explained. `merge_sort` is a more complicated algorithm than `minimum-sort`, with its recursive structure, the extra memory needed for the new deck, etc. The price to pay is that less comparisons can be done per second, since a lot of time is needed for other operations. But this is a moderate price, since we can more than pay for it by the gain in total runtime.

3.2.7 Lindenmayer systems

In this final section we want to present another application in which recursion is predominant and difficult to avoid (an iterative version would indeed require an explicit stack). As a bonus, this applications lets us draw beautiful pictures.

Let us first fix an *alphabet* Σ which is simply a finite set of symbols, for example $\Sigma = \{F, +, -\}$. Let Σ^* denote the set of all *words* that we can form from symbols in Σ. For example, $F + F + \in \Sigma^*$.

Next, we fix a function $P : \Sigma \to \Sigma^*$. P maps every symbol to a word, and these are the *productions*. We might for example have the productions

σ	$\mapsto$	$P(\sigma)$
F	$\mapsto$	$F+F+$
$+$	$\mapsto$	$+$
$-$	$\mapsto$	$-$

Finally, we fix an *initial word* $s \in \Sigma^*$, for example $s = F$.

The triple $\mathcal{L} = (\Sigma, P, s)$ is called a *Lindenmayer system*. Such a system generates an infinite sequence of words $s = w_0, w_1, \ldots$ as follows. To get the next word w_i from the previous word w_{i-1}, we simply substitute all symbols in w_{i-1} by their productions.

Figure 19: *The turtle before and after processing the command sequence* $F+F+$

In our example, this yields

$$\begin{aligned} w_0 &= F, \\ w_1 &= F+F+ \\ w_2 &= F+F++F+F++ \\ w_3 &= F+F++F+F+++F+F++F+F+++ \\ &\vdots \end{aligned}$$

The next step is to "draw" these words, and this gives the pictures we were talking about.

Turtle graphics. Imagine a turtle sitting at some point p on a large piece of paper, with its head pointing in some direction, see Figure 19 (left). The turtle can understand the commands F, $+$, and $-$. F means "move one step forward", $+$ means "turn counterclockwise by an angle of 90 degrees", and $-$ means "turn clockwise by an angle of 90 degrees". The turtle can process any sequence of such commands, by executing them one after another. We are interested in the resulting path taken by the turtle on the piece of paper. The path generated by the command sequence $F+F+$, for example, is shown in Figure 19 (right), along with the position and orientation of the turtle after processing the command sequence.

The turtle can therefore graphically interpret any word generated by a Lindenmayer system over the alphabet $\{F, +, -\}$.

Recursively drawing Lindenmayer systems. For $\sigma \in \Sigma$, let w_i^σ denote the word resulting from σ by the i-fold substitution of all symbols according to their productions. In our running example, we have for example $w_2 = w_2^F = F+F++F+F++$ and $w_i^+ = +$ for all i.

The point is now that can we express w_i^σ in terms of the w_{i-1}'s of other symbols, and this is where recursion comes into play. Suppose that $P(\sigma) = \sigma_1 \cdots \sigma_k$. Then we can obtain w_i^σ as follows. We first substitute σ by $\sigma_1 \cdots \sigma_k$ (1-fold substitution), and in the resulting word $\sigma_1 \cdots \sigma_k$ we apply $(i-1)$-fold substitution to all the symbols. This shows that $=$

$$w_i^\sigma = w_{i-1}^{\sigma_1} \cdots w_{i-1}^{\sigma_k}.$$

This formula also implies that the drawing of w_i^σ is obtained by simply concatenating the drawings for $w_{i-1}^{\sigma_1}, \ldots, w_{i-1}^{\sigma_k}$. To get the actual word w_i, we simply concatenate the drawings of all w_i^σ, for σ running through the symbols of the initial word s.

Program 27 shows how this works for our running example with productions $F \mapsto F+F+, + \mapsto +, - \mapsto -$ and initial word F. Since $P^i(+) = +, P^i(-) = -$ for all i, we do not need to substitute $+$ and $-$ and get

$$w_i = w_i^F = w_{i-1}^F + w_{i-1}^F + . \tag{3.2}$$

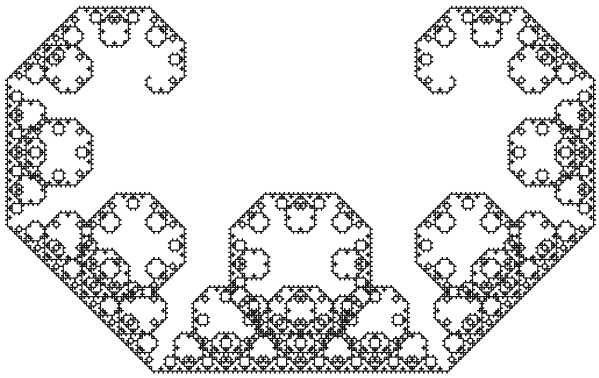
The program assumes the existence of a library `turtle` with predefined turtle command functions `forward`, `left` (counterclockwise rotation with some angle) and `right` (clockwise rotations with some angle) in namespace `ifm`.

In the documentation of the program, we have omitted the "trivial" productions $+ \mapsto +, - \mapsto -$, and in specifying a Lindenmayer system, we can do so as well: we will usually only list productions for symbols that are not mapped to themselves.

```
1  // Prog: lindenmayer.C
2  // Draw turtle graphics for the Lindenmayer system with
3  // production F -> F+F+ and initial word F.
4
5  #include <iostream>
6  #include <IFM/turtle>
7
8  // POST: the word w_i^F is drawn
9  void f (unsigned int i) {
10   if (i == 0)
11     ifm::forward();  // F
12   else {
13     f(i-1);          // w_{i-1}^F
14     ifm::left(90);   // +
15     f(i-1);          // w_{i-1}^F
16     ifm::left(90);   // +
17   }
18 }
19
20 int main () {
21   std::cout << "Number of iterations =? ";
22   unsigned int n;
23   std::cin >> n;
24
25   // draw w_n = w_n(F)
26   f(n);
27
28   return 0;
29 }
```

Program 27: *progs/lindenmayer.C*

For input $n = 14$, the program will produce the following drawing.

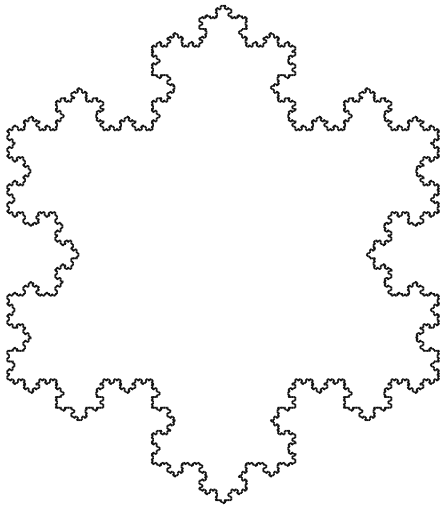


As n gets larger, the picture does not seem to change much; it rotates, and some more details develop, but apart from that the impression is the same. Assume you could draw the picture for $n = \infty$. Then equation (3.2) would give

$$w_\infty = w_\infty + w_\infty + .$$

This is a *self-similarity*: the drawing of w_∞ consists of two rotated drawings of itself. We have a *fractal*!

Additional features. We can extend the definition of a Lindenmayer system to include a rotation angle α that may be different from 90 degrees. This is shown in Program 28 that draws a snowflake for input $n = 5$.



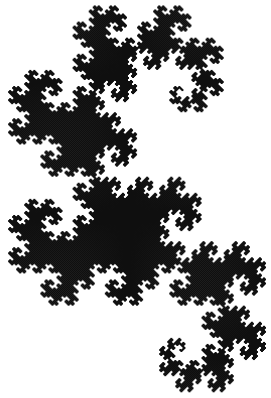
```
1  // Prog: snowflake.C
2  // Draw turtle graphics for the Lindenmayer system with
3  // production F -> F-F++F-F, initial word F++F++F and
4  // rotation angle 60 degrees.
5  #include <iostream>
6  #include <IFM/turtle>
7
8  // POST: the word w_i^F is drawn
9  void f (unsigned int i) {
10   if (i == 0)
11     ifm::forward();  // F
12   else {
13     f(i-1);          // w_{i-1}^F
14     ifm::right(60);  // -
15     f(i-1);          // w_{i-1}^F
16     ifm::left(120);  // ++
17     f(i-1);          // w_{i-1}^F
18     ifm::right(60);  // -
19     f(i-1);          // w_{i-1}^F
20   }
21 }
22
23 int main () {
24   std::cout << "Number of iterations =? ";
25   unsigned int n;
26   std::cin >> n;
27
28   // draw w_n = w_n^F++w_n^F++w_n^F
29   f(n);              // w_n^F
30   ifm::left(120);    // ++
31   f(n);              // w_n^F
32   ifm::left(120);    // ++
33   f(n);              // w_n^F
34
35   return 0;
36 }
```

Program 28: *progs/snowflake.C*

To get more flexibility, we can also extend the alphabet Σ of symbols. For example, we may add symbols without any graphical interpretation; these are still useful, though, since they may be used in productions. For example, the Lindenmayer system with $\Sigma = \{F, +, -, X, Y\}$, initial word X and productions

$$\begin{aligned} X &\mapsto X + YF + \\ Y &\mapsto -FX - Y \end{aligned}$$

yields the *dragon curve* (w_{14}, angle of 90 degrees).



The corresponding code is shown in Program 29.

```
1  // Prog: dragon.C
2  // Draw turtle graphics for the Lindenmayer system with
3  // productions X -> X+YF+, Y -> -FX-Y, initial word X
4  // and rotation angle 90 degrees
5  #include <iostream>
6  #include <IFM/turtle>
7
8  void y (unsigned int i);  // necessary: x and y call each other
9
10 // POST: w_i^X is drawn
11 void x (unsigned int i) {
12   if (i > 0) {
13     x(i-1);            // w_{i-1}^X
14     ifm::left(90);     // +
15     y(i-1);            // w_{i-1}^Y
16     ifm::forward();    // F
17     ifm::left(90);     // +
18   }
19 }
20
21 // POST: w_i^Y is drawn
22 void y (unsigned int i) {
23   if (i > 0) {
24     ifm::right(90);    // -
25     ifm::forward();    // F
26     x(i-1);            // w_{i-1}^X
27     ifm::right(90);    // -
28     y(i-1);            // w_{i-1}^Y
29   }
```

```
30 }
31
32 int main () {
33   std::cout << "Number of iterations =? ";
34   unsigned int n;
35   std::cin >> n;
36
37   // draw w_n = w_n^X
38   x(n);
39
40   return 0;
41 }
```

Program 29: *progs/dragon.C*

Finally, one can add symbols with graphical interpretation. Commonly used symbols are f (jump one step forward, this doesn't leave a trace), [(remember current position) and] (jump back to last remembered position). It is also typical to add new symbols with the same interpretation as F, say.

3.2.8 Details

Lindenmayer systems. Lindenmayer systems are named after the Danish biologist Aristide Lindenmayer (1925–1985) who proposed them in 1968 to model the growth of plants. Lindenmayer systems (with generalizations to 3-dimensional space) have found many applications in computer graphics.

3.2.9 Goals

Dispositional. At this point, you should ...

1) understand the concept of recursion, and why it makes sense to define a function through itself;

2) understand the semantics of recursive function calls and be aware that they do not always terminate;

3) appreciate the power of recursion in sorting and drawing Lindenmayer systems.

Operational. In particular, you should be able to ...

(G1) find pre- and postconditions for given recursive functions;

(G2) prove or disprove termination and correctness of recursive function calls;

(G3) translate recursive mathematical function definitions into C++ function definitions;

(G4) rewrite a given recursive function in iterative form;

(G5) recognize inefficient recursive functions and improve their performance;

(G6) count the number of operations of a given type in a recursive function call, using induction as the main tool;

(G7) write recursive functions for given tasks.

3.2.10 Exercises

Exercise 82 *Find pre- and postconditions for the following recursive functions.* (G1)

a)
```
bool f (int n)
{
  if (n == 0) return false;
  return !f(n-1);
}
```

b)
```
void g (unsigned int n)
{
  if (n == 0) {
    std::cout << "*";
    return;
  }
  g(n-1);
  g(n-1);
}
```

c)
```
unsigned int h (unsigned int n, unsigned int b) {
  if (n == 1) return 0;
  return 1 + h (n / b, b);
}
```

Exercise 83 *Prove or disprove for any of the following recursive functions that it terminates for all possible arguments. In this theory exercise, overflow should not be taken into account, i.e. you should pretend that the value range of* `unsigned int` *is equal to* $\mathbb{N}$. (G2)

a)
```
unsigned int f (unsigned int n)
{
  if (n == 0) return 1;
  return f(f(n-1));
}
```

b)
```
// POST: return value is the Ackermann function value A(m,n)
unsigned int A (unsigned int m, unsigned int n) {
  if (m == 0) return n+1;
  if (n == 0) return A(m-1,1);
  return A(m-1, A(m, n-1));
}
```

c)
```
unsigned int f (unsigned int n, unsigned int m)
{
  if (n == 0) return 0;
  return 1 + f ((n + m) / 2, 2 * m);
}
```

Exercise 84

a) Write and test a C++ function that computes binomial coefficients $\binom{n}{k}, n, k \in \mathbb{N}$. *These may be defined in various equivalent ways. For example,*

$$\binom{n}{k} := \frac{n!}{k!(n-k)!},$$

or

$$\binom{n}{k} := \begin{cases} 0, & \text{if } n < k \\ 1, & \text{if } n = k \text{ or } k = 0 \\ \binom{n-1}{k} + \binom{n-1}{k-1}, & \text{if } n > k, k > 0 \end{cases},$$

or

$$\binom{n}{k} := \begin{cases} 0, & \text{if } n < k \\ 1, & \text{if } n \geq k, k = 0 \\ \frac{n}{k}\binom{n-1}{k-1} & \text{if } n \geq k, k > 0 \end{cases}$$

b) Which of the three variants is best suited for the implementation, and why? Argue theoretically, but underpin your arguments by comparing at least two different implementations of the function.

(G3)(G5) (G7)

Exercise 85 *In how many ways can you own CHF 1? Despite its somewhat philosophical appearance, the question is a mathematical one. Given some amount of money, in how many ways can you partition it using the available denominations (bank notes and coins)? The denominations in CHF are 1000, 200, 100, 50, 20, 10 (banknotes), 5, 2, 1, 0.50, 0.20, 0.10, 0.05 (coins). The amount of CHF 0.20,*

for example, can be owned in four ways (to get integers, let's switch to centimes): $(20), (10,10), (10,5,5), (5,5,5,5)$.

Solve the problem for any given input amount, by writing a program `partition` *that defines the following function (all values to be understood as centimes).*

```
// PRE: [first, last) is a valid nonempty range that describes
//      a sequence of denominations d_1 > d_2 > ... > d_n > 0
// POST: return value is the number of ways to partition amount
//       using denominations from d_1, ..., d_n
unsigned int partitions (unsigned int amount,
                         unsigned int* first,
                         unsigned int* last);
```

Use your program to determine in how many ways you can own CHF 1, and CHF 10. Can your program compute the number of ways for CHF 50? (G7)

Exercise 86 *Suppose you want to crack somebody's secret code, consisting of* d *digits between* 1 *and* 9. *You have somehow found out that exactly* k *of these digits are* 1*'s.*

a) Write a program that generates all possible codes. The program should contain a function that solves the problem for given arguments d *and* k.

b) Adapt the program so that it also outputs the number of possible codes.

For example, if $\mathrm{d} = 2$ *and* $\mathrm{k} = 1$, *the output may look like this:*

```
12 13 14 15 16 17 18 19 21 31 41 51 61 71 81 91
There were 16 possible codes.
```

(G7)

Exercise 87 *Rewrite the following recursive function in iterative form and test with a program whether your iterative version is correct. What can you say about the runtimes of both variants for values of* n *up to* 100, *say?* (G4)(G5)

```
unsigned int f (unsigned int n)
{
  if (n <= 2) return 1;
  return f(n-1) + 2 * f(n-3);
}
```

Exercise 88 *The following function finds an element with a given value* x *in a sorted sequence (if there is such an element).*

```
// PRE: [first, last) is a valid range, and the elements *p,
//      p in [first, last) are in ascending order
// POST: return value is a pointer p in [first, last) such
//       that *p = x, or the pointer last, if no such pointer
```

```
//        exists
int* binary_search (int* first, int* last, int x)
{
  int n = last - first;
  if (n == 0) return last;           // empty range
  if (n == 1)
    if (*first == x)
      return first;
    else
      return last;
  // n >= 2
  int* middle = first + n/2;
  if (*middle > x) {
    // x can't be in [middle, last)
    int* p = binary_search (first, middle, x);
    if (p == middle)
      return last; // x not found
    else
      return p;
  } else
    // *middle <= x; we may skip [first, middle)
    return binary_search (middle, last, x);
}
```

What is the maximum number $T(n)$ *of comparisons between sequence elements and* x *that this function performs if the number of sequence elements is* n*? Try to find an upper bound on* $T(n)$ *that is as good as possible. (You may use the statement of Exercise 89.)* (G6)

Exercise 89 *For any natural number* $n \geq 2$*, prove the following two (in)equalities.* (G6)

$$\lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil \leq \lceil \log_2 \lceil \frac{n}{2} \rceil \rceil = \lceil \log_2 n \rceil - 1.$$

Exercise 90 *Write programs that produce turtle graphics drawings for the following Lindenmayer systems* (Σ, P, s)*.* (G7)

a) $\Sigma = \{F, +, -\}$, $s = F + F + F + F$ and P given by

$$F \mapsto FF + F + F + F + F + F - F.$$

b) $\Sigma = \{X, Y, +, -\}$, $s = Y$, and P given by

$$\begin{aligned} X &\mapsto Y + X + Y \\ Y &\mapsto X - Y - X. \end{aligned}$$

For the drawing, use rotation angle $\alpha = 60$ degrees and interpret *both* X and Y as "move one step forward".

c) Like b), but with the productions

$$\begin{aligned} X &\mapsto X+Y++Y-X--XX-Y+ \\ Y &\mapsto -X+YY++Y+X--X-Y. \end{aligned}$$

Exercise 91 *The* Towers of Hanoi *puzzle (that can actually be bought from shops) is the following. There are three wooden pegs labeled* $1,2,3$*, where the first peg holds a stack of* n *disks, stacked in decreasing order of size, see Figure Figure 20.*

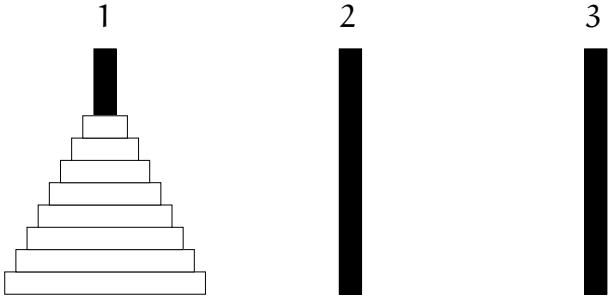


Figure 20: *The Tower of Hanoi*

The goal is to transfer the stack of disks to peg 3, by moving one disk at a time from one peg to another. The rule is that at no time, a larger disk may be on top of a smaller one. For example, we could start by moving the topmost disk to peg 2 *(move* $(1,2)$*), then move the next disk from peg* 1 *to peg* 3 *(move* $(1,3)$*), then move the smaller disk from peg* 2 *onto the larger disk on peg* 3 *(move* $(2,3)$*), etc.*

Write a program `hanoi.C` *that outputs a sequence of moves that does the required transfer, for given input* n*. For example, if* $n = 2$*, the above initial sequence* $(1,2)(1,3)(2,3)$ *is already complete and solves the puzzle. Check the correctness of your program by hand at least for* $n = 3$*, by manually reproducing the sequence of moves on a piece of paper (or an actual Tower of Hanoi, if you have one).* (G7)

3.2.11 Challenges

Exercise 92 *The game* Connect Four *is played by two players on a rectangular upright board of* n *rows and* m *columns that is initially empty. The players (let's call them White and Black) move alternatingly, where the player whose turn it is drops a piece of her color onto a column that is not full yet.*

The first player to get four pieces of her color consecutively aligned in a column, row, or diagonal wins the game. If the board becomes full before this happens, the game is a draw.

Figure 21 shows a possible sequence of initial moves, and Figure 22 depicts a winning situation for White (you can tell right away that Black was stupid).

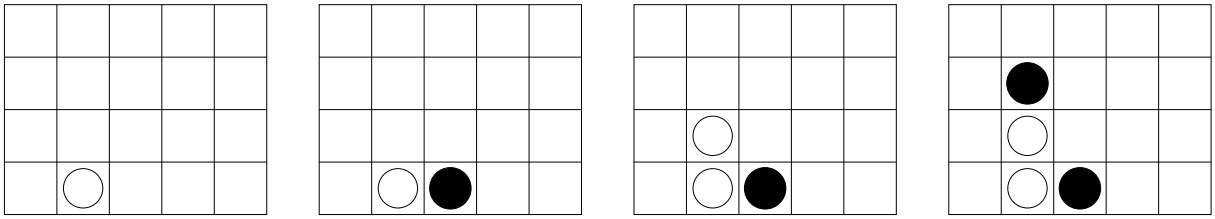


Figure 21: *First moves in a Connect Four game with 4 rows and 5 columns*

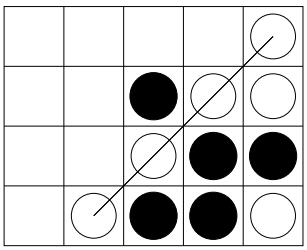


Figure 22: *White wins in a Connect Four game with 4 rows and 5 columns*

Write a program `connectfour.C` *that can play Connect Four against a human player, either as White or Black. But we don't want a program that can play decently, we want a program that plays optimally! This means that at any time during the game, the machine player selects a best possible move. This is a move that leads to a winning situation for the machine player as fast as possible if the machine player can win at all. If the machine player is bound to lose, the best move is the one that keeps the game going for as long as possible. If the game can't be won by either player, the best move is any move that keeps the game in a state of draw. In all cases, the machine player assumes that the human player also plays optimally (or, that the machine player plays against another instance of itself).*

The program should at least be able to handle boards with 4 *rows and up to* 6 *columns. If the program is really mean, it may tell the human player after how many more pieces it will have won. If this number goes down substantially after a move of the human player, it is clear that the human didn't play optimally.*

Use the program to determine for all games with 4 *rows and up to* 6 *columns whether a) White can always win if she plays optimally, or b) Black can always win if she plays optimally, or c) the game is a draw, if both players play optimally.*