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 of 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:

\[
\begin{align*}
  n! & := \begin{cases} 
  1, & \text{if } n \leq 1 \\
  n \cdot (n-1)!, & \text{if } n > 1
\end{cases}
\end{align*}
\]

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.

```c++
// 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 \( \text{fac}(n-1) \) is a recursive call of \( \text{fac} \).

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

```c++
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 \( \text{fac} \) above, this is the case: each time \( \text{fac} \) 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 \( \text{fac}(3) \), say. The formal argument \( n \) is initialized with \( 3 \), and since this is greater than \( 1 \), the statement \( \text{return } n \cdot \text{fac}(n-1) \) is executed next. This first evaluates the expression \( n \cdot \text{fac}(n-1) \) and is particular the right operand \( \text{fac}(n-1) \). Since \( n=3 \) had 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 \( f(3) \), 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.\(^1\)

Like a stack of paper 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 accepts 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 called function 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 "pops" it.

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

\(^1\) 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.
### 3.2. Recursion

<table>
<thead>
<tr>
<th>call stack (bottom —&gt; top)</th>
<th>evaluation sequence</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>fac(3)</td>
<td>push 3</td>
</tr>
<tr>
<td>...</td>
<td>n * fac(n-1)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>n * fac(n-1)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>n * (n * fac(n-1))</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>n * fac(2)</td>
<td>push 2</td>
</tr>
<tr>
<td>...</td>
<td>n * fac(2)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>n * (n * fac(n-1))</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>n * fac(1)</td>
<td>push 1</td>
</tr>
<tr>
<td>...</td>
<td>n * (n + 1)</td>
<td>pop</td>
</tr>
<tr>
<td>...</td>
<td>n * (n + 1)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>n * (2 + 1)</td>
<td>pop</td>
</tr>
<tr>
<td>...</td>
<td>n * (2 + 1)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>3 * 2</td>
<td>pop</td>
</tr>
<tr>
<td>...</td>
<td>3 * 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>3 * 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>6</td>
<td>pop</td>
</tr>
<tr>
<td>...</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

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 preconditions, 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
\[
\gcd(a, b) = \gcd(b, a \mod b).
\]

**Proof.** Let \( k \) be a divisor of \( b \), from
\[
a = (a \div b)b + a \mod b
\]

it follows that
\[
a = (a \div b)b + a \mod b.
\]

Since \( a \div b \) and \( b/k \) are integers, we get
\[
\frac{a \mod b}{k} \in \mathbb{N} \iff \frac{a}{k} \in \mathbb{N}.
\]

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

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

```cpp
// 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 \( a \mod b < b \)).

**Lemma 2** For \( b > 0 \), then \( \gcd(a, b) = \gcd(b, a \mod b) \).

**Proof.** Let \( k \) be a divisor of \( b \), from
\[
a = (a \div b)b + a \mod b
\]

it follows that
\[
a = (a \div b)b + a \mod b.
\]

Since \( a \div b \) and \( b/k \) are integers, we get
\[
\frac{a \mod b}{k} \in \mathbb{N} \iff \frac{a}{k} \in \mathbb{N}.
\]

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

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

follows,
\[
F_0 := 0, \\
F_1 := 1, \\
F_n := F_{n-1} + F_{n-2}, \quad n > 1.
\]

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 \( \text{fib}(50) \) computes \( F_{50} \) already twice (once directly in \( \text{fib}(50) \), and once indirectly from \( \text{fib}(49) \)), \( F_{49} \) is computed three times, \( F_{48} \) five times, and \( F_{47} \) 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 \( \text{gcd} \) 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 \( \text{gcd} \), this update corresponds to the transformation \( (a, b) \rightarrow (b, a \mod b) \).

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

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 \( \text{fib} \) for computing Fibonacci numbers is not tail-end recursive, but it is still easy to write it iteratively. Remember that \( F_0 \) is the sum of \( F_{-1} \) and \( F_{-2} \). 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 \( b \).

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

Again, this non-recursive version \( \text{fib2} \) is substantially longer and more difficult to understand than \( \text{fib} \), but this time there is a benefit: \( \text{fib2} \) 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 \( \text{fib}(50) \) to terminate, \( \text{fib2}(50) \) gives us the answer in no time. Unfortunately, this answer may be incorrect, since \( F_{50} \) could exceed the value range of the type \( \text{unsigned int} \).

In this case we would prefer \( \text{fib} \) over \( \text{fib2} \), simply since \( \text{fib} \) is too inefficient for practical use. The more complicated function definition of \( \text{fib2} \) is a moderate price to pay for the speed-up 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,

\[
// POST: return value is the n-th Fibonacci number F_n
{\text {unsigned int f (unsigned int n)}}
{
    if (n == 0) return 1;
}
is not allowed, since \( f \) recursively calls itself with an argument depending on \( f \). 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 20th 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 an argument that depends on \( A \).

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

```c
int A(int m, 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, \( 2^{\text{65536}} \), 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 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 author 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 \([\text{first}, \text{last})\], minimum sort can be realized as follows.

```c
// 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
        ...
    }
}
```

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( \ldots )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 2 )</td>
<td>( 3 )</td>
<td>( \ldots )</td>
<td>( n + 1 )</td>
<td></td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 2 )</td>
<td>( 3 )</td>
<td>( 4 )</td>
<td>( \ldots )</td>
<td>( n + 2 )</td>
<td></td>
</tr>
<tr>
<td>( 2 )</td>
<td>( 3 )</td>
<td>( 7 )</td>
<td>( 9 )</td>
<td>( \ldots )</td>
<td>( 2n + 3 )</td>
<td></td>
</tr>
<tr>
<td>( 3 )</td>
<td>( 5 )</td>
<td>( 13 )</td>
<td>( 20 )</td>
<td>( 64 )</td>
<td>( \ldots )</td>
<td>( 2^{n+3} - 3 )</td>
</tr>
</tbody>
</table>

Table 6: Some values of Ackermann's function.
The standard library function `std::inter_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 comparison 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:

**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 a = new int[n];`) or with approximately the same frequency. This concern the assignment `p_min = q` which may happen up to `n(n−1)/2` times, and the expression `++q != last` clearly 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_1 n(n−1)/2` is negligible compared to the quadratic term `c_2 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 total time `T_{total}` will be proportional to the time `T_{comp}` that is being spent with comparisons between sequence elements.

\[^1\text{Due to the effects of caching and other side effects on the von Neumann architecture, this is not necessarily true on your platform.}\]
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-sorts appears to be pretty inefficient, 1,000,000 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 1,000,000 numbers, say. For this, we derive from Observation 1 the required number of Gcomp comparisons (50,000) and multiply it with 3.4. The resulting 170,000 seconds are almost two days.

Essentially the same figure result from running other well-known simple sorting algorithms like bubble-sort or insertion-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 et conquer.

Here is how this works for sorting. Let us imagine that the number 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 of 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 is 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) { if (n <= 1) return; // nothing to do
    int middle = first + n/2;
    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]
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 skeptical 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 \( n \)merge\_sort at the various levels of recursion, so let us first count the number of comparisons between sequence elements that occur when two arguments are sorted. For merge-sort, this number is equal to \( \frac{n}{2} \) for each call to \( n \)merge\_sort, resulting in a total of \( \frac{2^{(n/2)}}{2} \) comparisons. In a single sorted deck of \( n \) cards, we can make \( \frac{n}{2} \) comparisons by calling \( n \)merge\_sort with \( \frac{n}{2} \) cards.

To see this, let us consider a sequence of \( n \) elements that actually requires the maximum number of comparisons. This number of comparisons is the sum of the respective numbers in the left and the right half, plus the number of comparisons during the merge step. The latter two numbers are by construction of merge\_sort and definition of \( n \)merge\_sort at most \( T(\frac{n}{2}) \) and \( T(\frac{n}{2}) \), 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 \( T(1) = 0 = (1-1)\log_2 2 \), 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 \( n/2, n/2 \geq 1 \)). This goes as follows.

\[
T(n) \leq T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lfloor \frac{n}{2} \right\rfloor) + n - 1 \quad \text{ Equation 3.1} \]

\[
\leq (\left\lfloor \frac{n}{2} \right\rfloor - 1)\log_2 \left\lfloor \frac{n}{2} \right\rfloor + (\left\lfloor \frac{n}{2} \right\rfloor - 1)\log_2 \left\lfloor \frac{n}{2} \right\rfloor + n - 1 \quad \text{ (inductive hypothesis)}
\]

\[
\leq \left(\frac{n}{2} - 1\right)\log_2 n + \left(\frac{n}{2} - 1\right)(\log_2 n - 1) + n - 1 \quad \text{ (Exercise 88)}
\]

\[
= \left(\frac{n-2}{2}\right)\log_2 n + n - 1 \quad \text{ (n = \left\lceil \frac{n}{2} \right\rceil)}
\]

\[
\leq \left(\frac{n-1}{2}\right)\log_2 n + n - 1
\]

As for minimum 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 \( M \)comp, the number of \( M \)merge\_sort comparisons \( 10^6 \) comparisons, according to Theorem 2. In other words, \( M \)comp = \( 10^6 (n-1)\log_2 n \). Time is the absolute runtime of the program, this time in seconds and not minutes. But as in Table 7, see/Geomp tells us how many seconds the program needs to perform one \( M \)merge\_sort.

We first observe that this latter number decreases with \( n \), where the rate of decrease becomes smaller and smaller. Our platform, we can go up to roughly \( n = 51 \), 200, 000 and find that see/Geomp continue 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, add 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 $(n-1)\log n$ 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 $(n-1)\log n$ 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 $(n-1)(\log n)$ 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 no longer 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 application lets us draw beautiful pictures.

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

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

\[
\begin{align*}
\alpha & \mapsto P(\alpha) \\
F & \mapsto F + F + \\
+ & \mapsto + \\
- & \mapsto -
\end{align*}
\]

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

The triple $(\Sigma, P, s)$ 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 $w_{i+1}$ from the previous word $w_i$, we simply substitute all symbols in $w_i$ by their productions.

Recursively drawing Lindenmayer systems. For $\alpha \in \Sigma$, let $w^n_\alpha$ denote the word resulting from $\alpha$ by the $n$-fold substitution of all symbols according to their productions. In our running example, we have for example $w_2 = w_1^F = F + F + F + F + F + F + F$ and $w_3 = w_2^F$ for all $i$.

The point is now that we can express $w^n_\alpha$ in terms of the $w^n_i$'s of other symbols, and this is where recursion comes into play. Suppose that $P(\alpha) = \alpha_1 \ldots \alpha_n$. Then we can obtain $w^n_\alpha$ as follows. We first substitute $\alpha$ by $\alpha_1 \ldots \alpha_n$ (1-fold substitution), and in the resulting word $\alpha_1 \ldots \alpha_n$ we apply $(i-1)$-fold substitution to all the symbols. This shows that

\[
w^n_\alpha = w^n_{\alpha_1} \ldots w^n_{\alpha_n}.
\]
This formula also implies that the drawing of \( w_0 \) is obtained by simply concatenating the drawings for \( w_{i-1} \), \( \ldots, w_0 \). To get the actual word \( w_0 \), we simply concatenate the drawings of all \( w_i \), for \( i \) running through the symbols of the initial word \( w_0 \).

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

\[
w_0 = w_0^\prime = w_{i-1} + w_{i-1}^\prime.
\] (3.2)

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

In the documentation of the program, we have omitted the "trivial" productions \( + \rightarrow +, - \rightarrow - \) 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 // productions \( F \rightarrow F + F \) and initial word \( F \).
4 #include <iostream>
5 #include <ifm/turtle>
6
7 // POST: the word \( w_i \cdot i \cdot F \) is drawn
8 void f (unsigned int i) {
9     if (i == 0) {
10         ifm::forward(); // F
11     } else {
12         f(i-1); // \( w_{i-1} \cdot i \cdot F \)
13         ifm::left(90); // +
14         f(i-1); // \( w_{i-1} \cdot i \cdot F \)
15         ifm::left(90); // +
16     }
17 }
18
19 int main () {
20     std::cout << "Number of iterations =? ";
21     unsigned int n;
22     std::cin >> n;
23     // draw \( w_n = w_n(F) \)
24     f(n);
25     return 0;
26 }

Program 27: 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

\[ w_{\infty} = w_\infty + w_{\infty} \cdot i \cdot F. \]

This is a self-similarity: the drawing of \( w_{\infty} \) 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 \( \alpha \) that may be different from 90 degrees. This is shown in Program 28 that draws a snowflake for input \( n = 5 \).
3.2. RECURSION

Program 28: prog/snowflake.C

To get more flexibility, we can also extend the alphabet \( \sum \) 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 \( \sum = \{ F, +, -, X, Y \} \), initial word \( X \) and productions

\[
X \rightarrow X + Y F +
\]

\[
Y \rightarrow -F X - Y
\]

yields the dragon curve (with angle of 90 degrees).

The corresponding code is shown in Program 29.

Program 29: prog/dragon.C
3.2. RECURSION

```c
30 } return 0;
31 }
32 int main () {
33     std::cout << "Number of iterations =? ";
34     unsigned int n;
35     std::cin >> n;
36     // draw w_n = w_n^X x(n);
37     return 0;
38 } return f(f(n-1));
39 }
```

Program 29: prog/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 (1925-1988) who proposed them in 1968 to model the growth of plants. Lindenmayer systems (with generalisations 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)
```c
a) bool f (int n) {
    if (n == 0) return false;
    return f(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 N. (G2)
```c
a) unsigned int f (unsigned int n) {
    if (n == 0) return 1;
    return f(f(n-1));
}
```
3.2. RECURSION

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));
}

unsigned int f (unsigned int n, unsigned int m) {
  if (m == 0) return n + 1;
  if (n == 0) return A(m - 1, 1);
  return f(n - 1) + 2 * f(n - 3);
}

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

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

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.

Exercise 85 In how many ways can you own CHP? 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 CHP are 1000, 500, 100, 50, 20, 10, 5, 2, 1, 0.50, 0.20, 0.10, 0.05 (coins). The amount of CHP 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).

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

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 \( d = 2 \) and \( k = 1 \), the output may look like this:

12 13 14 15 16 17 18 19 20 30 31 41 51 61 71 81 91

There were 16 possible codes.

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?

\[
\begin{align*}
\text{unsigned int f (unsigned int n)} & \quad \text{ (G4)} \\
& \quad \text{ (G5)} \\
& \quad \text{ (G7)} \\
\left\{ \begin{array}{c}
\text{if } (n <= 2) \text{ return 1} \\
\text{return } f(n-1) + 2 * f(n-3); \\
\end{array} \right.
\end{align*}
\]

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

\[ \log_{2}(\frac{n}{2}) \leq \log_{2}(\frac{n}{2}) = \log_{2}n - 1. \]

Exercise 90. Write programs that produce turtle graphics drawings for the following Lindenmayer systems \((\Sigma, P, s)\), \( \text{a) } \Sigma = \{F, +, -\}, s = F + F + F + F + F-F \text{ and } P \text{ given by} \]

\[ F \mapsto FF + F + F + F + F - F. \]

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

\[ X \mapsto Y + X + Y \]
\[ Y \mapsto X - Y. \]

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 20.

![Figure 20: The Tower of Hanoi](image)

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).

![Diagram](connectfour.png)

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

![Diagram](connectfour2.png)

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 good, 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 `White` plays optimally, or b) `Black` can always win if `Black` plays optimally, or c) the game is a draw, if both players play optimally.