B.8 Recursion

Solution to Exercise 110.

a)
```
// PRE: n >= 0
// POST: return value is false if n is even and true if n is odd
```

b)
```
// PRE: n >= 0
// POST: 2^n stars have been written to standard output
```

c) Let's analyze this function: if $n = b^e, e \geq 0$, then it is not hard to see that the function outputs $e = \log_b n$. An obvious precondition is $b \neq 0$. If n is not a power of b, things may go wrong, since the first argument may become 0 at some point, and then we enter an infinite recursion. This happens for example if $1 < n < b$. Also, if $b = 1$, we have an infinite recursion if $n > 1$. Let us therefore assume that $b \geq 2$.

Since n can always be written in the form

$$n = \sum_{i=0}^{k} \beta_i b^i, \quad 0 \leq \beta_i < b \ \forall i,$$

we can easily compute what happens, assuming that $k \geq 1$ and $\beta_k > 0$ (this just means $n \geq b$). The first integer division by b "cuts off" β_0 and yields

$$n' = \sum_{i=1}^{k} \beta_i b^{i-1}.$$

The next division cuts off β_1 and yields

$$n'' = \sum_{i=2}^{k} \beta_i b^{i-2}.$$

The pattern is clear: we eventually get the number β_k. If $\beta_k = 1$, the function call terminates, otherwise, the next division yields 0 as the first call parameter, and we have an infinite recursion. This yields the following pre- and postconditions.

```
// PRE: b >= 2, there exists e with b^e <= n < 2*b^e
// POST: return value is e = log_b(n), rounded down
```

Solution to Exercise 111.

a) This does not always terminate. Consider the call `f(1)`. This recursively calls `f(f(0))` which is `f(1)` again, and so on.

b) Let us move the call parameter `A(m, n-1)` out of the recursive call to make things more clear. This does not change anything since that parameter has to be evaluated before the recursive function call anyway.

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

Now we see that the pair (m, n) gets *lexicographically* smaller in every recursive call. Under lexicographic order, $(m', n') < (m, n)$ if $m' < m$, or if $m' = m$ and $n' < n$. Therefore, starting from (m, n), the first parameter must go down by one after a finite number of recursive calls, meaning that it must also reach 0 after a finite number of recursive calls. At that point, the recursion bottoms out.

c) This one is somewhat tricky, since when you run it, it always seems to terminate. But this is only due to overflow in the arithmetic. Mathematically, this function does not terminate: if $m \geq n$, then no parameter decreases in the recursive call, and we again have

$$n' := (n + m) \operatorname{div} 2 \leq m < 2m =: m'.$$

Solution to Exercise 112. Here is the program for a) and b).

```
// Prog: mccarthy.cpp
// defines and calls McCarthy's 91 Function
#include <iostream>

// POST: return value is M(n), where M is McCarthy's 91 Function
unsigned int mccarthy(const unsigned int n) {
  if (n > 100)
    return n - 10;
  else
    return mccarthy(mccarthy(n + 11));
}

int main()
{
  // input
  std::cout << "Compute McCarthy's 91 Function M(n) for n =? ";
  unsigned int n;
  std::cin >> n;

  // computation and output
  std::cout << "M(" << n << ") = " << mccarthy(n) << "\n";

  return 0;
}
```

For c), you play with the program a little and start to guess that

$$M(n) = \begin{cases} n - 10, & \text{if } n > 100 \\ 91, & \text{if} n \leq 100 \end{cases},$$

and this obviously explains the name *McCarthy's 91 Function.* Here is an inductive proof of this fact. Actually, we only need to handle the finitely many cases $n = 0, \ldots, 100$ since for $n > 100$, the result $n - 10$ follows from the definition. So we do backwards induction. Assume that we need to establish the validity of the formula for a given $n \leq 100$. We assume that the formula is already correct for all larger values of n. The definition gives us

$$M(n) = M(M(n + 11)).$$

If $n + 11 > 100$, we thus get $M(n) = M(n + 11 - 10) = M(n + 1)$. If $n = 100$, this is $M(101) = 91$, and if $n < 100$, then $n+1 \leq 100$, so by induction we also get $M(n+1) = 91$. If $n + 11 \leq 100$, we inductively get $M(n + 11) = 91$, hence

$$\begin{aligned} M(n) &= M(91) := M(M(102)) \\ &= M(92) := M(M(103)) \\ &= M(93) := \cdots \\ &= M(99) := M(M(110)) \\ &= M(100) := M(M(111)) \\ &= M(101) = 91. \end{aligned}$$

Solution to Exercise 113. Let us start with part b). The first variant has the problem that if we *first* compute $n!$ and *only then* divide by $k!$ and $(n - k)!$, we can't compute many binomial coefficients, since $n!$ does not fit into an `unsigned int` variable already for small n (if we have 32-bit arithmetic, then 12! is the highest we can do).

The second variant is bad since it is very slow. We have the same phenomenon as with the Fibonacci numbers: the computation time is at least proportional to the computed number itself, and binomial coefficients are quite large.

The third solution seems to be best in practice, but even here, we have to be somewhat careful in the implementation. We should *first* multiply $\binom{n-1}{k-1}$ with n, and *only then* divide by k. First dividing by k is tempting in order to keep the numbers small but that doesn't work, since $\binom{n-1}{k-1}$ might not be divisible by k. This third method produces intermediate numbers that *are* larger than absolutely necessary, but only by at most n.

```
unsigned int binomial(unsigned int n, unsigned int k)
{
  if (n < k) return 0;
  if (k == 0) return 1;
  return n * binomial(n-1, k-1) / k;
}
```

Solution to Exercise 114.

```
// Prog: partition.cpp
// compute in how many ways a fixed amount of money can be
// partioned using the available denominations (banknotes
// and coins). This program is for the currency CHF, where
// the denominations are (in centimes)
//    100000, 20000, 10000, 5000, 2000, 1000 (banknotes)
//    500, 200, 100, 50, 20, 10, 5           (coins)
//
// Example: CHF 0,20 can be partitioned in four ways
//     (20), (10, 10), (10, 5, 5), and (5, 5, 5, 5)
#include<iostream>

// 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 (const unsigned int amount,
                         const unsigned int* first,
                         const unsigned int* last)
{
  if (amount == 0) return 1;
  unsigned int ways = 0;
  // ways = ways_1 + ... + ways_n, where ways_i is the number
  // of ways to partition amount using d_i as the largest
  // denomination
  for (const unsigned int* d = first; d != last; ++d)
    // ways_i = number of partitions of the form (d_i, X), with
    // (X) being a partition of amount-d_i using d_i,...,d_n
    if (amount >= *d) ways += partitions (amount-*d, d, last);
  return ways;
}

int main()
{
  // the 13 denominations of CHF
  unsigned int chf[] =
    {100000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5};

  // input
  std::cout << "In how many ways can I own x CHF-centimes for x =? ";
  unsigned int x;
  std::cin >> x;

  // comutation and output
  std::cout << partitions (x, chf, chf+13) << "\n";

  return 0;
}
```

The number of ways in which you can own CHF 1 is 50, and CHF 10 can be owned in 104561 ways. The above program becomes very slow for larger values, since during the recursive calls, many values are computed over and over again. For CHF 50, we already have to wait "forever". We can speed things up by using dynamic programming. We don't even have to change the structure of our function, but we provide it with an additional twodimensional array to store the values that have already been computed. Whenever we need a value, we first check whether it has already been computed, and only if this is not the case, we recursively call the function.

```cpp
// Prog: partition2.cpp
// compute in how many ways a fixed amount of money can be
// partioned using the available denominations (banknotes
// and coins). This program is for the currency CHF, where
// the denominations are (in centimes)
//    100000, 20000, 10000, 5000, 2000, 1000 (banknotes)
//    500, 200, 100, 50, 20, 10, 5           (coins)
//
// Example: CHF 0,20 can be partitioned in four ways
//     (20), (10, 10), (10, 5, 5), and (5, 5, 5, 5)
#include<iostream>
#include<algorithm>

// PRE: [first, last) is a valid nonempty range that describes
//      a sequence of denominations d_1 > d_2 > ... > d_n > 0
//      memory is a pointer to a twodimensional array with
//      number of rows >= amount, and number of columns >=
//      last-first, with the following property:
//      for 0 < a < amount, and for d in [first, last),
//      memory[a-1][last-d-1] either has value -1,
//      or it is equal to dyn_prog_partitions (a, d, last, memory)
// POST: return value is the number of ways to partition amount
//       using denominations from d_1, ..., d_n
unsigned int dyn_prog_partitions (const unsigned int amount,
                                  const unsigned int* first,
                                  const unsigned int* last,
                                  int** memory)
{
  if (amount == 0) return 1;
  unsigned int ways = 0;
  // ways = ways_1 + ... + ways_n, where ways_i is the number
  // of ways to partition amount using d_i as the largest
  // denomination
  for (const unsigned int* d = first; d != last; ++d)
    // ways_i = number of partitions of the form (d_i, X), with
    // (X) being a partition of amount-d_i using d_i,...,d_n
    if (amount >= *d) {
      // is ways_i already stored in memory?
      int stored_value = -1;
      if (amount > *d)
        stored_value = memory[amount - *d - 1][last-d-1];
      if (stored_value != -1)
        ways += stored_value;
      else
        ways += dyn_prog_partitions (amount - *d, d, last, memory);
    }
  // store the new value
  memory[amount-1][last-first-1] = ways;
  return ways;
}

// 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,
                         const unsigned int* first,
                         const unsigned int* last)
{
  // allocate memory for dynamic programming approach
  int** const memory = new int*[amount];
  for (int** m = memory; m < memory + amount; ++m) {
    *m = new int[last-first];
    std::fill (*m, *m + (last-first), -1);
  }

  // call the version with memory
  const unsigned int result =
    dyn_prog_partitions (amount, first, last, memory);

  // delete memory
  for (int** m = memory; m < memory + amount; ++m)
    delete[] *m;
  delete[] memory;

  return result;
}

int main()
{
  // the 13 denominations of CHF
  unsigned int chf[] =
    {100000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5};

  // input
  std::cout << "In how many ways can I own x CHF-centimes for x =? ";
  unsigned int x;
  std::cin >> x;

  // comutation and output
  std::cout << partitions (x, chf, chf+13) << "\n";

  return 0;
}
```

With this, we can very quickly compute the number of ways for CHF 50; it is 513269191. Beyond that, the type unsigned int is at some point no longer sufficient to represent the number of ways, since these are simply too many. But we may switch to ifm::integer and then even get the answer for CHF 100 quickly (it is 47580509178).

```
// Prog: partition3.cpp
// compute in how many ways a fixed amount of money can be
// partioned using the available denominations (banknotes
// and coins). This program is for the currency CHF, where
// the denominations are (in centimes)
//    100000, 20000, 10000, 5000, 2000, 1000 (banknotes)
//    500, 200, 100, 50, 20, 10, 5           (coins)
//
// Example: CHF 0,20 can be partitioned in four ways
//     (20), (10, 10), (10, 5, 5), and (5, 5, 5, 5)
#include<iostream>
#include<algorithm>
#include<IFM/integer.h>

// PRE: [first, last) is a valid nonempty range that describes
//      a sequence of denominations d_1 > d_2 > ... > d_n > 0
//      memory is a pointer to a twodimensional array with
//      number of rows >= amount, and number of columns >=
//      last-first, with the following property:
//      for 0 < a < amount, and for d in [first, last),
//      memory[a-1][last-d-1] either has value -1,
//      or it is equal to dyn_prog_partitions (a, d, last, memory)
// POST: return value is the number of ways to partition amount
//       using denominations from d_1, ..., d_n
ifm::integer dyn_prog_partitions (const unsigned int amount,
                                  const unsigned int* first,
                                  const unsigned int* last,
                                  ifm::integer** memory)
{
  if (amount == 0) return 1;
  ifm::integer ways = 0;
  // ways = ways_1 + ... + ways_n, where ways_i is the number
  // of ways to partition amount using d_i as the largest
  // denomination
  for (const unsigned int* d = first; d != last; ++d)
    // ways_i = number of partitions of the form (d_i, X), with
    // (X) being a partition of amount-d_i using d_i,...,d_n
    if (amount >= *d) {
      // is ways_i already stored in memory?
      ifm::integer stored_value = -1;
      if (amount > *d)
        stored_value = memory[amount - *d - 1][last-d-1];
      if (stored_value != -1)
        ways += stored_value;
      else
        ways += dyn_prog_partitions (amount - *d, d, last, memory);
    }
  // store the new value
  memory[amount-1][last-first-1] = ways;
  return ways;
}

// 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
ifm::integer partitions (unsigned int amount,
                         const unsigned int* first,
                         const unsigned int* last)
{
  // allocate memory for dynamic programming approach
  ifm::integer** const memory = new ifm::integer*[amount];
  for (ifm::integer** m = memory; m < memory + amount; ++m) {
    *m = new ifm::integer[last-first];
    std::fill (*m, *m + (last-first), -1);
  }

  // call the version with memory
  const ifm::integer result =
    dyn_prog_partitions (amount, first, last, memory);

  // delete memory
  for (ifm::integer** m = memory; m < memory + amount; ++m)
    delete[] *m;
  delete[] memory;

  return result;
}

int main()
{
  // the 13 denominations of CHF
  unsigned int chf[] =
    {100000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5};

  // input
  std::cout << "In how many ways can I own x CHF-centimes for x =? ";
  unsigned int x;
  std::cin >> x;

  // comutation and output
  std::cout << partitions (x, chf, chf+13) << "\n";

  return 0;
}
```

Solution to Exercise 115.

```
#include<iostream>

// PRE:  d >= k
// POST: all codes with digits between 1 and 9 are output that
//       result from the partial code p by an extension with d
//       digits, k of which are 1; the return value is the
//       number of such codes
unsigned int crack (const unsigned int p, const unsigned int d,
                    const unsigned int k)
{
  if (d == 0) {
    // k == 0 as well by PRE, and we have a full code
    std::cout << p << " ";
    return 1;
  }

  // there are two possibilities to continue:
  //   next digit is 1, or not
  unsigned int n = 0; // total number of codes

  // poss 1: only if we still have 1's left
  if (k > 0)
    n += crack (10 * p + 1, d - 1, k - 1); // next digit is 1

  // poss 2: only if not all remaining digits have to be 1's
  if (d > k)
    for (unsigned int i=2; i<10; ++i)
      n += crack (10 * p + i, d - 1, k); // next digit is i != 1

  return n;
}

// PRE:  d >= k

// POST: all d-digit codes with digits between 1 and 9 are
//       output that have exactly k digits equal to one;
//       the return value is the number of such codes
unsigned int crack (const unsigned int d, const unsigned int k)
{
  return crack (0, d, k);
}

int main() {
  const int n = crack (2, 1);
  std::cout << "\nThere were " << n << " possible codes.\n";
  return 0;
}
```

Solution to Exercise 116. The function is rewritten in a way similar to what we did with the Fibonacci numbers. The recursive variant becomes very slow at some point, while the iterative version is always fast.

```
// Prog: rec2it2.cpp
// rewrites a recursive function in iterative form a la Fibonacci
#include<iostream>

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

unsigned int f_it (const unsigned int n)
{
   if (n <= 2) return 1;
   unsigned int a = 1;  // f(0)
   unsigned int b = 1;  // f(1)
   unsigned int c = 1;  // f(2)
   for (unsigned int i = 3; i < n; ++i) {
     const unsigned int a_prev = a;  // f(i-3)
     a = b;                          // f(i-2)
     b = c;                          // f(i-1)
     c = b + 2 * a_prev;             // f(i)
   }
   return c + 2 * a;                 // f(n-1) + 2 * f(n-3)
}

int main()
{
  std::cout << "Comparing f and f_it...\n";
  for (int n = 0; n < 100; ++n)
    std::cout << f(n) << " = " << f_it(n) << "\n";

  return 0;
}
```

Solution to Exercise 117. Here, we cannot directly use the Fibonacci trick, since it is not sufficient to remember only a constant number of previous function values. But we can simply remember *all* values by using an array. Here, both versions are reasonably fast, since the second recursive call is to a problem of size $n/2$ only.

```
// Prog: rec2it2.cpp
// rewrites a recursive function in iterative form by using an array
#include<iostream>

unsigned int f (const unsigned int n)
{
  if (n == 0) return 1;
  return f(n-1) + 2 * f(n/2);
}

unsigned int f_it (const unsigned int n)
{
  if (n == 0) return 1;
  unsigned int* const f_values = new unsigned int[n+1]; // f(0),...,f(n)
  f_values[0] = 1;
  for (unsigned int i=1; i<=n; ++i)
    f_values[i] = f_values[i-1] + 2 * f_values[i/2];
  const unsigned int result = f_values[n];
  delete[] f_values;
  return result;
}
```

n	1,600,000	3,200,000	6,400,000	128,000,000
merge_sort	0.83	1.75	3.65	7.4
std::sort	0.32	0.76	1.7	3.65
speedup	2.6	2.3	2.14	2.0

Table 10: *Runtime in seconds of* merge-sort *vs./* std::sort

```

int main()
{
  std::cout << "Comparing f and f_it...\n";
  for (int n = 0; n < 100; ++n)
    std::cout << f(n) << " = " << f_it(n) << "\n";

  return 0;
}
```

Solution to Exercise 118. Here is the sorting program.

```
// Prog: std_sort.cpp
// tests std::sort on random input

#include<iostream>
#include<algorithm>

int main()
{
  // input of number of values to be sorted
  unsigned int n;
  std::cin >> n;

  int* const a = new int[n];

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

  // create sequence:
  for (int i=0; i<n; ++i) a[i] = i;
  std::random_shuffle (a, a+n);

  // sort into ascending order
  std::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;
}
```

On the platform of the authors, the following happens Table 10). Here, *Speedup* is the factor by which `std::sort` is faster than `merge_sort` in absolute runtime. We see that the speedup is larger for smaller inputs but then approaches a factor of 2.

Solution to Exercise 119. As for *merge-sort*, it can be shown that $T(0) = 0, T(1) = 1$ and

$$T(n) \leq 1 + \max\left(T(\lfloor \frac{n}{2} \rfloor), T(\lceil \frac{n}{2} \rceil)\right), \quad n \geq 2.$$

Given this, a bound of $T(n) \leq 1 + \lceil \log_2 n \rceil$ can be shown by induction.

Solution to Exercise 120. The inequality simply follows from the fact that $\log_2 x$ is a monotone increasing function. To see the equation, let us first consider the case where n is a power of two, $n = 2^k$, say, with $k \geq 1$. Then all involved numbers are integers, the rounding operation does nothing, and the statement follows from

$$\log_2 \frac{n}{2} = \log_2 n - 1.$$

Otherwise (if n is not a power of two), let us choose the unique number k such that

$$2^k < n < 2^{k+1}.$$

By taking logarithms, it follows that

$$k < \log_2 n < k + 1,$$

so

$$\lceil \log_2 n \rceil = k + 1. \tag{B.1}$$

We also have

$$2^{k-1} < \frac{n}{2} < 2^k,$$

and this yields

$$2^{k-1} < \lceil \frac{n}{2} \rceil \leq 2^k.$$

Taking logarithms, we get

$$k - 1 < \log_2 \lceil \frac{n}{2} \rceil \leq k,$$

and this means that

$$\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil = k. \tag{B.2}$$

The desired equality now follows from (B.1) and (B.2).

Solution to Exercise 121.

a)

```
// Prog: lindenmayer_a.cpp
// Draw turtle graphics for the Lindenmayer system with
// production F -> FF+F+F+F+F+F-F, initial word F+F+F+F
// and rotation angle 60 degrees

#include <iostream>
#include <IFM/turtle>

void f (const unsigned int i) {
  // POST: the word w_i^F is drawn
  if (i == 0)
    ifm::forward();  // F
  else {
    f(i-1);          // w_{i-1}^F
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
    f(i-1);          // w_{i-1}^F
    ifm::right(90);  // -
    f(i-1);          // w_{i-1}^F
  }
}

int main () {
  std::cout << "Number of iterations =? ";
  unsigned int n;
  std::cin >> n;

  // draw w_n = w_n(F+F+F+F)
  f(n); ifm::left(90); f(n); ifm::left(90);
  f(n); ifm::left(90); f(n);
  return 0;
}
```

```
// Prog: lindenmayer_b.cpp
// Draw turtle graphics for the Lindenmayer system with
// productions X -> Y+X+Y, Y -> X-Y-X, initial word Y
// and rotation angle 60 degrees

#include <iostream>
#include <IFM/turtle>

void y (const unsigned int i);
// necessary: x and y call each other

void x (const unsigned int i) {
  // POST: w_i^X is drawn
  if (i == 0)
    ifm::forward();
  else {
    y(i-1);           // w_{i-1}^Y
    ifm::left(60);    // +
    x(i-1);           // w_{i-1}^X
    ifm::left(60);    // +
    y(i-1);            // w_{i-1}^Y
  }
}

void y (const unsigned int i) {
  // POST: w_i^Y is drawn
  if (i == 0)
    ifm::forward();
  else {
    x(i-1);            // w_{i-1}^X
    ifm::right(60);    // -
    y(i-1);            // w_{i-1}^Y
    ifm::right(60);    // -
    x(i-1);            // w_{i-1}^X
  }
}

int main () {
  std::cout << "Number of iterations =? ";
  unsigned int n;
  std::cin >> n;

  // draw w_n = w_n^Y
  y(n);

  return 0;
}
```

b)

```
// Prog: lindenmayer_c.cpp
// Draw turtle graphics for the Lindenmayer system with
// productions X ->  X+Y++Y-X--XX-Y+, Y -> -X+YY++Y+X--X-Y,
// initial word Y  and rotation angle 60 degrees

#include <iostream>
#include <IFM/turtle>

void y (const unsigned int i);
// necessary: x and y call each other

void x (const unsigned int i) {
  // POST: w_i^X is drawn
  if (i == 0)
    ifm::forward();
  else {
    x(i-1);            // w_{i-1}^X
    ifm::left(60);     // +
    y(i-1);            // w_{i-1}^Y
    ifm::left(60);     // +
    ifm::left(60);     // +
    y(i-1);            // w_{i-1}^Y
    ifm::right(60);    // -
    x(i-1);            // w_{i-1}^X
    ifm::right(60);    // -
    ifm::right(60);    // -
    x(i-1);            // w_{i-1}^X
    x(i-1);            // w_{i-1}^X
    ifm::right(60);    // -
    y(i-1);            // w_{i-1}^Y
    ifm::left(60);     // +
  }
}

```

```
void y (const unsigned int i) {
  // POST: w_i^Y is drawn
  if (i == 0)
    ifm::forward();
  else {
    ifm::right(60);    // -
    x(i-1);            // w_{i-1}^X
    ifm::left(60);     // +
    y(i-1);            // w_{i-1}^Y
    y(i-1);            // w_{i-1}^Y
    ifm::left(60);     // +
    ifm::left(60);     // +
    y(i-1);            // w_{i-1}^Y
    ifm::left(60);     // +
    x(i-1);            // w_{i-1}^X
    ifm::right(60);    // -
    ifm::right(60);    // -
    x(i-1);            // w_{i-1}^X
    ifm::right(60);    // -
    y(i-1);            // w_{i-1}^Y
  }
}

int main () {
  std::cout << "Number of iterations =? ";
  unsigned int n;
  std::cin >> n;

  // draw w_n = w_n^Y
  y(n);

  return 0;
}
```

Solution to Exercise 122. To move n disks from peg s to peg t, we can first move the topmost $n-1$ disks to the helper peg, then move the bottommost disk directly to peg t, and then move the $n-1$ disks again, this time from the helper peg to peg t. In the following program, we do this through a recursive function. As a small hack, we determine the number of the third helper peg as $6-s-t$.

```
// Prog: hanoi.cpp
// solves the Tower of Hanoi puzzle

#include<iostream>

// PRE:  s and t are different and both in {1,2,3}
// POST: the sequence of moves necessary to transfer a stack of n
//       disks from peg s to peg t is written to standard output
void hanoi (const unsigned int n, const int s, const int t)
{
  if (n > 0) {
    // move topmost n-1 disks from s to helper peg 6-s-t
    hanoi (n-1, s, 6-s-t);
    // move bottommost disk from s to t
    std::cout << "(" << s << "," << t << ")";
    // move the n-1 disks from the helper peg to the t
    hanoi (n-1, 6-s-t, t);
  }
}

```

```
int main()
{
  // input
  std::cout << "Move a stack of n disks for n =? ";
  unsigned int n;
  std::cin >> n;

  // output
  hanoi (n, 1, 3);
  std::cout << "\n";

  return 0;
}
```

Solution to Exercise 123. The Lucas-Lehmer test is based on the recursive sequence $s_1, s_2, \ldots$ defined as follows.

$$\begin{aligned} s_1 &= 4, \\ s_i &= s_{i-1}^2 - 2, \quad i > 1. \end{aligned}$$

The actual test is provided by the following

> **Theorem.** Let $n \geq 3$ be prime. The Mersenne number $2^n - 1$ is prime if and only if s_{n-1} is divisible by $2^n - 1$.

Thus, we simply need to compute s_{n-1} and check this. The good thing is (see also Exercise 55) that in order to compute the remainder of s_{n-1} modulo $2^n - 1$, it suffices to do *all* calculations modulo $2^n - 1$. This is quite a saving, since the number s_{n-1} is huge compared to $2^n - 1$ itself, for larger n. Here is the program.

```
// Program: lucas_lehmer.cpp
// finds Mersenne primes using the Lucas-Lehmer test

#include <iostream>
#include <cmath>
#include <cassert>
#include <IFM/integer.h>

// PRE:  n > 2
// POST: return value is true if and only if n is prime
bool is_prime (const unsigned int n)
{
  assert (n > 2);

  // Computation: test possible divisors d up to sqrt(n)
  const unsigned int bound = (unsigned int)(std::sqrt(n));
  unsigned int d;
  for (d = 2; d <= bound && n % d != 0; ++d);

  // Output
  return d > bound;
}

// PRE:  p > 2 is prime, M_p = 2^p-1
// POST: returns true if and only if M_p = 2^p-1 is prime
//       according to the Lucas-Lehmer test
bool lucas_lehmer (const unsigned int p, const ifm::integer M_p)
```

```
{
  assert (p > 2 && is_prime (p));

  // compute (p-1)-st term of sequence s, modulo M_p
  ifm::integer s = 4; // s_1
  for (unsigned int i = 2; i < p; ++i)
    s = (s * s - 2) % M_p;
  return (s == 0);
}

int main()
{
  // try all prime exponents
  unsigned int p = 3;
  ifm::integer m_p = 8; // 2^p = M_p + 1
  std::cout << "List of Mersenne primes 2^p-1 (p > 2):\n";
  for(;;) {
    if (is_prime (p) && lucas_lehmer (p, m_p-1))
      std::cout << "2^" << p << "-1" << std::endl;
    ++p; m_p *= 2;
  }

  return 0;
}
```

Running it for something like eight days produced the following output (it gets really slow only beyond exponent 1,000).

```
List of Mersenne primes 2^p-1 (p > 2):
2^3-1
2^5-1
2^7-1
2^13-1
2^17-1
2^19-1
2^31-1
2^61-1
2^89-1
2^107-1
2^127-1
2^521-1
2^607-1
2^1279-1
2^2203-1
2^2281-1
2^3217-1
2^4253-1
2^4423-1
2^9689-1
2^9941-1
2^11213-1
```

Solution to Exercise 124. Here comes our handwaving: what is the average number E_i

of packages that you need to buy, given that you are still missing i stickers? Obviously, $E_0 = 0$, and E_n is the value that we are interested in.

If you are missing $i > 0$ stickers, you definitely have to buy another package, and this might decrease the number of missing stickers. In fact, the new number is between i (if you already have all five stickers that were in the package) and $i - 5$ (if all stickers were new). What is the probability that you go down to $i - k$ missing stickers, $k = 0, ..., 5$? For this event to happen, exactly k out of the 5 stickers have to come from the i that you are missing, and $5 - k$ have to come from the $n - i$ that you already have. The total number of 5-tuples of stickers is

$$\binom{n}{5} = \frac{n!}{5!(n-5)!},$$

and

$$\binom{i}{k}\binom{n-i}{5-k}$$

of them lead to the aforementioned event. (Recall that the binomial coefficient $\binom{m}{\ell}$ counts the number of ways in which ℓ objects can be selected from a total of m objects.)

This gives us the recursive formula

$$E_i = 1 + \sum_{k=0}^{5} \frac{\binom{i}{k}\binom{n-i}{5-k}}{\binom{n}{5}} E_{i-k}, \quad i > 0.$$

Note that for $i < 5$, this formula features negative subscripts $i - k$, but all these have multiplicative factor 0.

This formula is an incarnation of the partition theorem for conditional expectation, but the handwaving is this: given that you are missing $i > 0$ stickers, you need another package, and then the average number of packages for $i - k$ missing stickers, where the probability that a given k occurs is the above fraction.

The recursive formula is still not useful since E_i depends on E_i (for $k = 0$), but we can get rid of this by multiplying with $\binom{n}{5}$:

$$\left(\binom{n}{5} - \binom{n-i}{5}\right) E_i = \binom{n}{5} + \sum_{k=1}^{5} \binom{i}{k}\binom{n-i}{5-k} E_{i-k}.$$

Thus we get

$$E_i = \frac{\binom{n}{5}}{\binom{n}{5} - \binom{n-i}{5}} + \sum_{k=1}^{5} \frac{\binom{i}{k}\binom{n-i}{5-k}}{\binom{n}{5} - \binom{n-i}{5}} E_{i-k}, \quad i > 0.$$

The program now simply computes (and stores) the E_i's according to this formula, in the order E_0 (which we know), $E_1, E_2, \ldots, E_n$. In this way, we have always already computed the values that we need.

We use the third function from Exercise 113 to compute binomial coefficients, but since these may become pretty large, we work over the type double right away. $\binom{555}{5}$, the largest binomial coefficient that may come up, is 430960344486, a number that would require more than 32 bits but less than 53, so it will even exactly fit into a double).

```
// Prog: panini.cpp
// computes the expected number of 5-sticker packages that need
// to be purchased in order to have all n stickers in the collection

#include<iostream>

// POST: computes binomial coefficient "n choose k"
double binomial(const unsigned int n, const unsigned int k)
{
  if (n < k) return 0.0;
  if (k == 0) return 1.0;
  return n * binomial(n-1, k-1) / k;
}

int main()
{
  // input
  std::cout << "Collection size =? ";
  unsigned int n;
  std::cin >> n;

  // allocate the array...
  double* const E = new double[n+1]; // E_0,...,E_n

  // ...and fill it
  E[0] = 0;
  for (unsigned int i = 1; i <= n; ++i) {
    const double d = binomial(n, 5) - binomial(n-i, 5);
    E[i] =  binomial(n, 5) / d;
    for (unsigned int k = 1; k < 6 && k <= i; ++k)
      E[i] += binomial(i, k) * binomial(n-i, 5-k) / d * E[i-k];
  }

  // output
  std::cout << "Average number of packages is " << E[n] << ".\n";

  // clean up
  delete[] E;

  return 0;
}
```

According to this program, the expected number of packages for $n = 555$ is 763.213, so up to rounding to integer, the newspaper was right.

Solution to Exercise 125. Naturally, it's hard to present a solution here. Instead, let us just list the program corresponding to the production

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

given as an example in the challenge.

```
// Prog: bush.cpp
// Draw turtle graphics for the Lindenmayer system with
// production F -> FF+[+F-F-F]-[-F+F+F], initial word F
// and rotation angle 22 degrees

#include <iostream>
#include <IFM/turtle>

// POST: the word w_i^F is drawn
void f (const unsigned int i) {
  if (i == 0)
    ifm::forward();  // F
  else {
    f(i-1);          // F
    f(i-1);          // F
    ifm::left(22);   // +
    ifm::save();     // [
    ifm::left(22);   // +
    f(i-1);          // F
    ifm::right(22);  // -
    f(i-1);          // F
    ifm::right(22);  // -
    f(i-1);          // F
    ifm::restore();  // ]
    ifm::right(22);  // -
    ifm::save();     // [
    ifm::right(22);  // -
    f(i-1);          // F
    ifm::left(22);   // +
    f(i-1);          // F
    ifm::left(22);   // +
    f(i-1);          // F
    ifm::restore();  // ]
  }
}

int main () {
  std::cout << "Number of iterations =? ";
  unsigned int n;
  std::cin >> n;

  // draw w_n = w_n(F), vertically
  ifm::left(90);
  f(n);

  return 0;
}
```