Solution to Exercise 95.

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 96.

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 97. Here is the program for a) and b).

```
 1  // Prog: mccarthy.C
 2  // defines and calls McCarthy's 91 Function
 3  #include <iostream>
 4
 5  // POST: return value is M(n), where M is McCarthy's 91 Function
 6  unsigned int mccarthy(unsigned int n) {
 7    if (n > 100)
 8      return n - 10;
 9    else
10      return mccarthy(mccarthy(n + 11));
11  }
12
13  int main()
14  {
15    // input
16    std::cout << "Compute McCarthy's 91 Function M(n) for n =? ";
17    unsigned int n;
18    std::cin >> n;
19
20    // computation and output
21    std::cout << "M(" << n << ") = " << mccarthy(n) << "\n";
22
23    return 0;
24  }
```

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 98.   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 99.

```
1    // Prog: partition.C
2    // compute in how many ways a fixed amount of money can be
3    // partioned using the available denominations (banknotes
4    // and coins). This program is for the currency CHF, where
5    // the denominations are (in centimes)
6    //    100000, 20000, 10000, 5000, 2000, 1000 (banknotes)
7    //    500, 200, 100, 50, 20, 10, 5          (coins)
8    //
9    // Example: CHF 0,20 can be partitioned in four ways
10   //    (20), (10, 10), (10, 5, 5), and (5, 5, 5, 5)
11   #include<iostream>
12
13   // PRE: [first, last) is a valid nonempty range that describes
14   //      a sequence of denominations d_1 > d_2 > ... > d_n > 0
15   // POST: return value is the number of ways to partition amount
16   //       using denominations from d_1, ..., d_n
17   unsigned int partitions (unsigned int amount,
18                            unsigned int* first,
19                            unsigned int* last)
20   {
21     if (amount == 0) return 1;
22     unsigned int ways = 0;
23     // ways = ways_1 + ... + ways_n, where ways_i is the number
24     // of ways to partition amount using d_i as the largest
25     // denomination
26     for (unsigned int* d = first; d != last; ++d)
27       // ways_i = number of partitions of the form (d_i, X), with
28       // (X) being a partition of amount-d_i using d_i,...,d_n
29       if (amount >= *d) ways += partitions (amount-*d, d, last);
30     return ways;
31   }
32
33   int main()
34   {
35     // the 13 denominations of CHF
36     unsigned int chf[] =
37       {100000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5};
38
39     // input
40     std::cout << "In how many ways can I own x CHF-centimes for x =? ";
41     unsigned int x;
42     std::cin >> x;
43
44     // comutation and output
45     std::cout << partitions (x, chf, chf+13) << "\n";
46
47     return 0;
48   }
```

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.

```
1   // Prog: partition.C
2   // compute in how many ways a fixed amount of money can be
3   // partioned using the available denominations (banknotes
4   // and coins). This program is for the currency CHF, where
5   // the denominations are (in centimes)
6   //     100000, 20000, 10000, 5000, 2000, 1000 (banknotes)
7   //     500, 200, 100, 50, 20, 10, 5              (coins)
8   //
9   // Example: CHF 0,20 can be partitioned in four ways
10  //      (20), (10, 10), (10, 5, 5), and (5, 5, 5, 5)
11  #include<iostream>
12  #include<algorithm>
13
14  // PRE: [first, last) is a valid nonempty range that describes
15  //      a sequence of denominations d_1 > d_2 > ... > d_n > 0
16  //      memory is a pointer to a twodimensional array with
17  //      number of rows >= amount, and number of columns >=
18  //      last-first, with the following property:
19  //      for 0 < a < amount, and for d in [first, last),
20  //      memory[a-1][last-d-1] either has value -1,
21  //      or it is equal to dyn_prog_partitions (a, d, last, memory)
22  // POST: return value is the number of ways to partition amount
23  //       using denominations from d_1, ..., d_n
24  unsigned int dyn_prog_partitions (unsigned int amount,
25                                    unsigned int* first,
26                                    unsigned int* last,
27                                    int** memory)
28  {
29    if (amount == 0) return 1;
30    unsigned int ways = 0;
31    // ways = ways_1 + ... + ways_n, where ways_i is the number
32    // of ways to partition amount using d_i as the largest
33    // denomination
34    for (unsigned int* d = first; d != last; ++d)
35      // ways_i = number of partitions of the form (d_i, X), with
36      // (X) being a partition of amount-d_i using d_i,...,d_n
37      if (amount >= *d) {
38        // is ways_i already stored in memory?
39        int stored_value = -1;
40        if (amount > *d)
41          stored_value = memory[amount - *d - 1][last-d-1];
42        if (stored_value != -1)
43          ways += stored_value;
44        else
45          ways += dyn_prog_partitions (amount - *d, d, last, memory);
46      }
47    // store the new value
48    memory[amount-1][last-first-1] = ways;
49    return ways;
50  }
51
52  // PRE: [first, last) is a valid nonempty range that describes
53  //      a sequence of denominations d_1 > d_2 > ... > d_n > 0
54  // POST: return value is the number of ways to partition amount
55  //       using denominations from d_1, ..., d_n
56  unsigned int partitions (unsigned int amount,
57                           unsigned int* first,
58                           unsigned int* last)
59  {
60    // allocate memory for dynamic programming approach
61    int** memory = new int*[amount];
62    for (int** m = memory; m < memory + amount; ++m) {
63      *m = new int[last-first];
64      std::fill (*m, *m + (last-first), -1);
```

```
65     }
66
67     // call the version with memory
68     unsigned int result =
69       dyn_prog_partitions (amount, first, last, memory);
70
71     // delete memory
72     for (int** m = memory; m < memory + amount; ++m)
73       delete[] *m;
74     delete memory;
75
76     return result;
77 }
78
79 int main()
80 {
81     // the 13 denominations of CHF
82     unsigned int chf[] =
83       {100000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5};
84
85     // input
86     std::cout << "In how many ways can I own x CHF-centimes for x =? ";
87     unsigned int x;
88     std::cin >> x;
89
90     // comutation and output
91     std::cout << partitions (x, chf, chf+13) << "\n";
92
93     return 0;
94 }
```

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.

Solution to Exercise 100.

```
 1 #include<iostream>
 2
 3 // PRE:  d >= k
 4 // POST: all codes with digits between 1 and 9 are output that
 5 //       result from the partial code p by an extension with d
 6 //       digits, k of which are 1; the return value is the
 7 //       number of such codes
 8 unsigned int crack (unsigned int p, unsigned int d, unsigned int k)
 9 {
10     if (d == 0) {
11       // k == 0 as well by PRE, and we have a full code
12       std::cout << p << " ";
13       return 1;
14     }
15
16     // there are two possibilities to continue:
17     //    next digit is 1, or not
18     unsigned int n = 0; // total number of codes
19
20     // poss 1: only if we still have 1's left
21     if (k > 0)
22       n += crack (10 * p + 1, d - 1, k - 1); // next digit is 1
23
24     // poss 2: only if not all remaining digits have to be 1's
25     if (d > k)
```

```
26       for (unsigned int i=2; i<10; ++i)
27          n += crack (10 * p + i, d - 1, k); // next digit is i != 1
28
29    return n;
30 }
31
32 // PRE:   d >= k
33
34 // POST: all d-digit codes with digits between 1 and 9 are
35 //        output that have exactly k digits equal to one;
36 //        the return value is the number of such codes
37 unsigned int crack (unsigned int d, unsigned int k)
38 {
39    return crack (0, d, k);
40 }
41
42 int main() {
43    int n = crack (2, 1);
44    std::cout << "\nThere were " << n << " possible codes.\n";
45    return 0;
46 }
```

Solution to Exercise 101.   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.

```
1  #include<iostream>
2
3  unsigned int f (unsigned int n)
4  {
5    if (n <= 2) return 1;
6    return f(n-1) + 2 * f(n-3);
7  }
8
9  unsigned int f_it (unsigned int n)
10 {
11     if (n <= 2) return 1;
12     unsigned int a = 1;  // f(0)
13     unsigned int b = 1;  // f(1)
14     unsigned int c = 1;  // f(2)
15     for (unsigned int i = 3; i < n; ++i) {
16       unsigned int a_prev = a;  // f(i-3)
17       a = b;                    // f(i-2)
18       b = c;                    // f(i-1)
19       c = b + 2 * a_prev;       // f(i)
20     }
21     return c + 2 * a;           // f(n-1) + 2 * f(n-3)
22 }
23
24 int main()
25 {
26   std::cout << "Comparing f and f_it...\n";
27   for (int n = 0; n < 100; ++n)
28     std::cout << f(n) << " = " << f_it(n) << "\n";
29
30   return 0;
31 }
```

Solution to Exercise 102. As for *merge-sort*, it can be shown that $T(0) = 0, T(1) = 1$ and

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

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

Solution to Exercise 103. 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

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

Taking logarithms, we get

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

and this means that

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

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

Solution to Exercise 104.

a)

```
1   // Prog: lsys62a.C
2   // Draw turtle graphics for the Lindenmayer system with
3   // production F -> FF+F+F+F+F-F, initial word F+F+F+F
4   // and rotation angle 60 degrees
5
6   #include <iostream>
7   #include <IFM/turtle>
8
9   void f (unsigned int i) {
10    // POST: the word w_i^F is drawn
11    if (i == 0)
12      ifm::forward();   // F
13    else {
14      f(i-1);              // w_{i-1}^F
15      f(i-1);              // w_{i-1}^F
16      ifm::left(90);    // +
17      f(i-1);              // w_{i-1}^F
18      ifm::left(90);    // +
19      f(i-1);              // w_{i-1}^F
20      ifm::left(90);    // +
21      f(i-1);              // w_{i-1}^F
22      ifm::left(90);    // +
23      f(i-1);              // w_{i-1}^F
24      ifm::left(90);    // +
25      f(i-1);              // w_{i-1}^F
26      ifm::right(90);   // -
27      f(i-1);              // w_{i-1}^F
28    }
29  }
30
31  int main () {
32    std::cout << "Number of iterations =? ";
33    unsigned int n;
34    std::cin >> n;
35
36    // draw w_n = w_n(F+F+F+F)
37    f(n); ifm::left(90); f(n); ifm::left(90);
38    f(n); ifm::left(90); f(n);
39    return 0;
40  }
```

```
1   // Prog: lsys62b.C
2   // Draw turtle graphics for the Lindenmayer system with
3   // productions X -> Y+X+Y, Y -> X-Y-X, initial word Y
4   // and rotation angle 60 degrees
5
6   #include <iostream>
7   #include <IFM/turtle>
8
9   void y (unsigned int i);
10  // necessary: x and y call each other
11
12  void x (unsigned int i) {
13    // POST: w_i^X is drawn
14    if (i == 0)
15      ifm::forward();
16    else {
17      y(i-1);              // w_{i-1}^Y
18      ifm::left(60);    // +
19      x(i-1);              // w_{i-1}^X
20      ifm::left(60);    // +
```

```
21      y(i-1);                // w_{i-1}^Y
22    }
23  }
24
25  void y (unsigned int i) {
26    // POST: w_i^Y is drawn
27    if (i == 0)
28      ifm::forward();
29    else {
30      x(i-1);                // w_{i-1}^X
31      ifm::right(60);    // -
32      y(i-1);                // w_{i-1}^Y
33      ifm::right(60);    // -
34      x(i-1);                // w_{i-1}^X
35    }
36  }
37
38  int main () {
39    std::cout << "Number of iterations =? ";
40    unsigned int n;
41    std::cin >> n;
42
43    // draw w_n = w_n^Y
44    y(n);
45
46    return 0;
47  }
```

b)
```
 1  // Prog: lsys62c.C
 2  // Draw turtle graphics for the Lindenmayer system with
 3  // productions X ->  X+Y++Y-X--XX-Y+, Y -> -X+YY++Y+X--X-Y,
 4  // initial word Y  and rotation angle 60 degrees
 5
 6  #include <iostream>
 7  #include <IFM/turtle>
 8
 9  void y (unsigned int i);
10  // necessary: x and y call each other
11
12  void x (unsigned int i) {
13    // POST: w_i^X is drawn
14    if (i == 0)
15      ifm::forward();
16    else {
17      x(i-1);                // w_{i-1}^X
18      ifm::left(60);     // +
19      y(i-1);                // w_{i-1}^Y
20      ifm::left(60);     // +
21      ifm::left(60);     // +
22      y(i-1);                // w_{i-1}^Y
23      ifm::right(60);    // -
24      x(i-1);                // w_{i-1}^X
25      ifm::right(60);    // -
26      ifm::right(60);    // -
27      x(i-1);                // w_{i-1}^X
28      x(i-1);                // w_{i-1}^X
29      ifm::right(60);    // -
30      y(i-1);                // w_{i-1}^Y
31      ifm::left(60);     // +
32    }
33  }
34
```

```
35  void y (unsigned int i) {
36    // POST: w_i^Y is drawn
37    if (i == 0)
38      ifm::forward();
39    else {
40      ifm::right(60);    // -
41      x(i-1);            // w_{i-1}^X
42      ifm::left(60);     // +
43      y(i-1);            // w_{i-1}^Y
44      y(i-1);            // w_{i-1}^Y
45      ifm::left(60);     // +
46      ifm::left(60);     // +
47      y(i-1);            // w_{i-1}^Y
48      ifm::left(60);     // +
49      x(i-1);            // w_{i-1}^X
50      ifm::right(60);    // -
51      ifm::right(60);    // -
52      x(i-1);            // w_{i-1}^X
53      ifm::right(60);    // -
54      y(i-1);            // w_{i-1}^Y
55    }
56  }
57
58  int main () {
59    std::cout << "Number of iterations =? ";
60    unsigned int n;
61    std::cin >> n;
62
63    // draw w_n = w_n^Y
64    y(n);
65
66    return 0;
67  }
```

Solution to Exercise 105.   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 $6-s-t$.

```
1   // Prog: hanoi.C
2   // solves the Tower of Hanoi puzzle
3
4   #include<iostream>
5
6   // PRE:  s and t are different and both in {1,2,3}
7   // POST: the sequence of moves necessary to transfer a stack of n
8   //       disks from peg s to peg t is written to standard output
9   void hanoi (unsigned int n, int s, int t)
10  {
11    if (n > 0) {
12      // move topmost n-1 disks from s to helper peg 6-s-t
13      hanoi (n-1, s, 6-s-t);
14      // move bottommost disk from s to t
15      std::cout << "(" << s << "," << t << ")";
16      // move the n-1 disks from the helper peg to the t
17      hanoi (n-1, 6-s-t, t);
18    }
19  }
20
```

```
21  int main()
22  {
23    // input
24    std::cout << "Move a stack of n disks for n =? ";
25    unsigned int n;
26    std::cin >> n;
27
28    // output
29    hanoi (n, 1, 3);
30    std::cout << "\n";
31
32    return 0;
33  }
```

Solution to Exercise 106. 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 98 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`).

```
1    // Prog: panini.C
2    // computes the expected number of 5-sticker packages that need
3    // to be purchased in order to have all n stickers in the collection
4
5    #include <iostream>
6
7    // POST: computes binomial coefficient "n choose k"
8    double binomial(unsigned int n, unsigned int k)
9    {
10     if (n < k) return 0.0;
11     if (k == 0) return 1.0;
12     return n * binomial(n-1, k-1) / k;
13   }
14
15   int main()
16   {
17     // input
18     std::cout << "Collection size =? ";
19     unsigned int n;
20     std::cin >> n;
21
22     // allocate the array...
23     double* E = new double[n+1]; // E_0,...,E_n
24
25     // ...and fill it
26     E[0] = 0;
27     for (unsigned int i = 1; i <= n; ++i) {
28       double d = binomial(n, 5) - binomial(n-i, 5);
29       E[i] =  binomial(n, 5) / d;
30       for (unsigned int k = 1; k < 6 && k <= i; ++k)
31         E[i] += binomial(i, k) * binomial(n-i, 5-k) / d * E[i-k];
32     }
33
34     // output
35     std::cout << "Average number of packages is " << E[n] << ".\n";
36
```

```
37      // clean up
38      delete[] E;
39
40      return 0;
41  }
```

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 107. 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.

```
1   // Prog: bush.C
2   // Draw turtle graphics for the Lindenmayer system with
3   // production F -> FF+[+F-F-F]-[-F+F+F], initial word F
4   // and rotation angle 22 degrees
5
6   #include <iostream>
7   #include <IFM/turtle>
8
9   // POST: the word w_i^F is drawn
10  void f (unsigned int i) {
11    if (i == 0)
12      ifm::forward();   // F
13    else {
14      f(i-1);           // F
15      f(i-1);           // F
16      ifm::left(22);    // +
17      ifm::save();      // [
18      ifm::left(22);    // +
19      f(i-1);           // F
20      ifm::right(22);   // -
21      f(i-1);           // F
22      ifm::right(22);   // -
23      f(i-1);           // F
24      ifm::restore();   // ]
25      ifm::right(22);   // -
26      ifm::save();      // [
27      ifm::right(22);   // -
28      f(i-1);           // F
29      ifm::left(22);    // +
30      f(i-1);           // F
31      ifm::left(22);    // +
32      f(i-1);           // F
33      ifm::restore();   // ]
34    }
35  }
36
37  int main () {
38    std::cout << "Number of iterations =? ";
39    unsigned int n;
40    std::cin >> n;
41
42    // draw w_n = w_n(F), vertically
43    ifm::left(90);
44    f(n);
45
```

```
46    return 0;
47  }
```