Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Theoretische Informatik
Dr. B. Gärtner, Prof. Dr. J. Hromkovič

16. Dezember 2008

Informatik für Mathematiker und Physiker Lösung 13 HS 08

URL: http://www.ti.inf.ethz.ch/ew/courses/Info1_08/

Aufgabe 1

The following program computes all the solutions of a quadratic equation and returns them.

Programm: solve_quadratic_equation.C

```
// Prog: solve_quadratic_equation.C
// computes both (possibly complex) solutions to a quadratic equation
#include<iostream>
#include<complex>

// POST: return value is the number of distinct complex solutions
//       of the quadratic equation ax^2 + bx + c = 0. If there
//       are infinitely many solutions (a=b=c=0), the return
//       value is -1. Otherwise, the return value is a number n
//       from {0,1,2}, and the solutions are written to s1,..,sn
int solve_quadratic_equation (std::complex<double> a,
      std::complex<double> b,
      std::complex<double> c,
      std::complex<double>& s1,
      std::complex<double>& s2)
{
  if (a == 0.0)
    // linear case: bx + c = 0
    if (b == 0.0)
      // trivial case: c = 0
      if (c == 0.0)
return -1; // => infinitely many solutions
      else
return 0;  // => no solution
    else {
      // bx + c = 0, b != 0   => one solution
      s1 = -c/b;
      return 1;
    }
  else {
    // ax^2 + bx + c = 0, a != 0 => two solutions
    std::complex<double> d = std::sqrt(b*b-4.0*a*c);
    s1 = (-b + d) / (2.0*a);
    s2 = (-b - d) / (2.0*a);
    return 2;
```

```
  }
}


int main()
{
  // input
  std::cout << "Solve quadratic equation ax^2 + bx + c = 0 for\n";
  std::cout << "a =? ";
  double a;
  std::cin >> a;
  std::cout << "b =? ";
  double b;
  std::cin >> b;
  std::cout << "c =? ";
  double c;
  std::cin >> c;

  // computation
  std::complex<double> s1;
  std::complex<double> s2;

  int n = solve_quadratic_equation (a, b, c, s1, s2);

  // output
  std::cout << "Number of solutions: " << n << "\n";
  std::cout << "Solutions:\n";
  if (n > 0) std::cout << s1 << "\n";
  if (n > 1) std::cout << s2 << "\n";

  return 0;

}
```

Aufgabe 2

Here, we cannot directly use the Fibonacci trick, since it is not sufficient to remeber 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.

Programm: rec2it2.C

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

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

unsigned int f_it (unsigned int n)
{
if (n == 0) return 1;
unsigned int* 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];
unsigned int result = f_values[n];
delete[] f_values;
return result;
}

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;
}
```

Aufgabe 3

Here is the program for a) and b).

Programm: mccarthy.C

```
// Prog: mccarthy.C
// 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(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

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

Aufgabe 4

Here is an implementation of the function that initializes a vector of three rationals.

Programm: create_rational_vector_3.C

```
// Prog: create_rational_vector_3.C
// writes a function that can be used to initialize a
// rational_vector_3 from 6 int's
#include <cassert>

struct rational {
```

```
int n;
int d; // INV: d != 0
};

struct rational_vector_3 {
rational x;
rational y;
rational z;
};

// PRE: d != 0
// POST: the rational number n/d is returned
rational create_rational (int n, int d)
{
assert (d != 0);
rational result;
result.n = n;
result.d = d;
return result;
}

// PRE: d1 != 0, d2 != 0, d3 != 0
// POST: the rational_vector_3 (n1/d1, n2/d2, n3/d3) is returned
rational_vector_3 create_rational_vector_3
(int n1, int d1, int n2, int d2, int n3, int d3)
{
rational_vector_3 result;
result.x = create_rational (n1, d1);
result.y = create_rational (n2, d2);
result.z = create_rational (n3, d3);
return result;
}

int main()
{
rational_vector_3 v = create_rational_vector_3 (1,2,3,4,5,6);
return 0;
}
```