Dirk Praetorius
Michele Ruggeri

Sommersemester 2015
21.04.2015

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 5

Aufgabe 5.1. The *bubblesort* algorithm is an inefficient, but short sorting algorithm which works as follows: You run through the entries of a given vector $x \in \mathbb{R}^n$ several times. For every run, each entry x_j of x is compared to its successor x_{j+1}. If $x_j > x_{j+1}$, then the two entries x_j and x_{j+1} are swapped. After the first complete run through the vector, one knows that (at least) the last element is sorted correctly, i.e. the last element x_n is the maximum of the vector. Thus, in the next run one only has to go up-to the last-but-one entry of the vector. How many loops do you need for this algorithm? Write a function `bubblesort` which sorts a given vector $x \in \mathbb{R}^n$ with this algorithm. Additionally, write a main program that reads in $x \in \mathbb{R}^n$ and sorts it. The length n should be constant. However, your function `bubblesort` should be programmed for arbitrary lengths n. Save your source code as `bubblesort.c` into the directory `serie05`.

Aufgabe 5.2. Write a function `merge` that joins two arrays $a \in \mathbb{R}^m$ and $b \in \mathbb{R}^n$, which are sorted in ascending order, into the array $c \in \mathbb{R}^{m+n}$ such that the array c is sorted in ascending order as well, e.g., $a = (1, 3, 3, 4, 7)$ and $b = (1, 2, 3, 8)$ should be joined into $c = (1, 1, 2, 3, 3, 3, 4, 7, 8)$. Use the fact that the arrays a,b are sorted! The input of the function should be a base-pointer to the array c and the length m, n. It should hold $c_j = a_j$ for $j = 0, \ldots, m-1$ and $c_j = b_{j-m}$ for $j = m, \ldots, m+n-1$, i.e. the array c reads $c = (a, b)$. The input array should be overwritten by the function. You can use a temporary array of length $m + n$ in your function. Furthermore, write a main program that reads in $m, n \in \mathbb{N}$ as well as $a \in \mathbb{R}^m$ and $b \in \mathbb{R}^n$, and prints out the result $c \in \mathbb{R}^{m+n}$.

Aufgabe 5.3. Write a recursive function `mergesort` that sorts an array a in ascending order and returns the correctly sorted array. Use the following strategy:

- If the length of a is ≤ 2, then sort the array a explicitely.
- If the length of a is > 2, then split a into two arrays b, c of half length. Call the function `mergesort` recursively for b and c, and rejoin the arrays with the function `merge` from Exercise 5.2.

Think of this strategy with help of the example $a = (1, 3, 5, 2, 7, 1, 1, 3)$. Test your program appropriately. *Note:* If the length of a is $2n + 1$ with $n \geq 1$, then a is split into b with length $n + 1$ and c with length n. You might want to use *pointer arithmetics*, i.e. if `a` is an array and `p` is a pointer which contains the address of `a[k]` (i.e. `p = &a[k]`), then `p+n` is the address of `a[k+n]` (i.e. `*(p+n)` coincides with `a[k+n]`). Recall that `a` is the base pointer which contains the address of `a[0]`.

Aufgabe 5.4. Let the two series

$$a_N := \sum_{n=0}^{N} \frac{1}{(n+1)^2} \quad \text{und} \quad b_M := \sum_{m=0}^{M} \sum_{k=0}^{m} \frac{1}{(k+1)^2(m-k+1)^2}$$

be given. Write a program that measures the time used for the computation of a_N resp. b_M for different values of N resp. M. Print out the results tabularly. Do the results meet your expectations? Save your source code as `timing.c` into the directory `serie05`. *Hint:* Think of the computational complexity (Aufwand) for the computation of a_N resp. b_M.

Aufgabe 5.5. The sine function can be represented as a series via

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

The corresponding n-th partial sum is given by

$$S_n(x) = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

Write a function `sin_`, which, given $x \in \mathbb{R}$ and $\varepsilon > 0$, returns the first value of $S_n(x)$ such that

$$|S_n(x) - S_{n-1}(x)|/|S_n(x)| \leq \varepsilon \quad \text{or} \quad |S_n(x)| \leq \varepsilon.$$

Then, write a main program, which reads $x \in \mathbb{R}$ and $\varepsilon > 0$ from the keyboard, calls the function and displays the computed value $S_n(x)$, as well as the value $\sin(x)$, the absolute error $|S_n(x) - \sin(x)|$ and the relative error $|S_n(x) - \sin(x)|/|\sin(x)|$ (provided $\sin(x) \neq 0$). Save your source code as `sin.c` into the directory `serie05`.

Aufgabe 5.6. An alternative root-finding algorithm is the *Newton method*. Let $f : [a, b] \to \mathbb{R}$. Given an initial guess x_0, define the sequence $(x_n)_{n\in\mathbb{N}}$ via

$$x_{k+1} = x_k - f(x_k)/f'(x_k).$$

Implement the algorithm in a function `newton`. Given x_0 and a tolerance $\tau > 0$, the function performs the Newton iteration until

$$|f'(x_n)| \leq \tau$$

or

$$|f(x_n)| \leq \tau \quad \text{and} \quad |x_n - x_{n-1}| \leq \begin{cases} \tau & \text{for } |x_n| \leq \tau, \\ \tau|x_n| & \text{else.} \end{cases}$$

In the first case, print a warning to inform that the result is presumably wrong. The function uses suitable implementations of the object function `double f(double x)` and its derivative `double fprime(double x)`. Then, write a main program which reads x_0 from the keyboard and returns x_n. Save your source code as `newton.c` into the directory `serie05`.

Aufgabe 5.7. As for the contents of variables of elementary type (`double,int,...`), you can print out the content of a pointer with help of `printf`. The place-holder `%p` is used for addresses (which are the contents of pointers!). The output is system-dependent, but mostly in hexadecimal numbers. Write a function `void charPointerDist(char* startaddress, char* endaddress)` that prints out the following three values tabularly:

- Starting address
- End address
- Distance (difference) between both addresses (take care of the place-holder in `printf`!)

Since arrays are stored connectedly, the distance between two successive elements corresponds to the memory used for the specific datatype. Check your function with a `char`-array `c[2]` and the follwoing calls:

```
charPointerAbstand(&c[0],&c[1]);
charPointerAbstand(c,c+1);
```

Then, write a function `void doublePointerDist(double* startaddress, double* endaress)` and test it with a `double`-array. Compare the differences between the results of the two functions.
Optionally: Find out how much memory is used for the types `short`, `int`, and `long` on the `lva.student` server.

Aufgabe 5.8. The function `squareVector` should square all entries of a given vector $x \in \mathbb{R}^n$, i.e., the input $(-1, 2, 0)$ should be turned into $(1, 4, 0)$. The input vector should be passed as a pointer.

```
#include <stdio.h>

int squareVec(double vec, int n) {
```

```
  int j=0;
  for(j=1, j<dim; --j) {
    *vec[j] = &vec[j] * &vec[j];
  }
  return vec;
}

main() {
  double vec[3] = {-1.0,2.0,0.0};
  int j=0;

  squareVec(vec,3);
  for(j=0; j<3; ++j) {
    printf("vec[%d] = %f ",j,vec[j]);
  }
  printf("\n");
}
```

Change *only* the function `squareVec`, such that the `main` programm prints out the correct result. How many errors do you find? What is the computational complexity (Aufwand) of `squareVec`?