

Scientific programming in mathematics

Exercise sheet 5

Sorting algorithms (arrays, pointers, dynamical vectors, and complexity)

Exercise 5.1. Explain the differences between variables and pointers. What are advantages and disadvantages of each of them? Write a function `swap` that swaps the contents of two variables `x`, `y`. What is the problem with the following code?

```
void swap(double x, double y)
{
    double tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Save your source code as `swap.c` into the directory `series05`.

Exercise 5.2. Write a function `void selectionSort(double* x, int n)`, which sorts a given vector $x \in \mathbb{R}^n$ in ascending order using the *selection sort* algorithm described in slide 80 of the lecture notes. Work with dynamically allocated memory. Moreover, write a main program that provides the input vector, calls the function, and prints both the input vector and the sorted vector to the screen. Save your source code as `selectionsort.c` into the directory `series05`. Test your implementation with suitable examples. What is the computational complexity of your implementation of `selectionSort`? Justify your answer!

Exercise 5.3. Write a function `void insertionSort(double* x, int n)`, which sorts a given vector $x \in \mathbb{R}^n$ in ascending order using the *insertion sort* algorithm. The algorithm is very similar to the selection sort algorithm considered in Exercise 5.2. The vector is divided into a partial sorted subvector and an unsorted subvector. Initially, the partial sorted subvector contains only the first entry x_1 of the vector (note that a vector containing only one element is trivially ordered), while the unsorted subvector contains the remaining $n-1$ entries. During the second iteration, the second entry x_2 of the vector (which is the first entry of the unsorted subvector) is compared with the only element of the sorted subvector. If $x_2 < x_1$, they are swapped, so that the partial sorted subvector contains two elements. During the k -th iteration, the k -th entry x_k of the vector (which is the first entry of the unsorted subvector) is inserted into the sorted subvector in the correct place by performing an appropriate number of swaps. During the last iteration, the last entry of the vector (which is the only entry of the unsorted subvector at this point) is inserted into the correct position and the vector is sorted. Work with dynamically allocated memory. Moreover, write a main program that provides the input vector, calls the function, and prints both the input vector and the sorted vector to the screen. Save your source code as `insertionsort.c` into the directory `series05`. Test your implementation with suitable examples. What is the computational complexity of your implementation of `insertionSort`? Justify your answer!

Exercise 5.4. Write a function `void bubbleSort(double* x, int n)`, which sorts a given vector $x \in \mathbb{R}^n$ in ascending order using the *bubble sort* algorithm. You run through the entries of the vector several times. For each 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, one already 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 (and so on). How many loops do you need for this algorithm? Work with dynamically allocated memory. Moreover, write a main program that provides the input vector, calls the function, and prints both the input vector and the sorted vector to the

screen. Save your source code as `bubblesort.c` into the directory `series05`. Test your implementation with suitable examples. What is the computational complexity of your implementation of `bubbleSort`? Justify your answer!

Exercise 5.5. Write a *recursive* function `void mergeSort(double* x, int n)`, which sorts a given vector $x \in \mathbb{R}^n$ in ascending order using the *merge sort* algorithm. Use the following strategy:

- If $n \leq 2$, then the vector $x \in \mathbb{R}^n$ is explicitly sorted.
- If $n > 2$, then the vector x is split into two subvectors y and z of half length. Then the function `mergeSort` is recursively called for y and z . Finally, y and z are merged into a sorted vector. For the merging process, you can exploit the fact that y and z are sorted.

Work with dynamically allocated memory. Moreover, write a main program that provides the input vector, calls the function, and prints both the input vector and the sorted vector to the screen. Save your source code as `mergesort.c` into the directory `series05`. Test your implementation with suitable examples. What is the computational complexity of your implementation of `mergeSort`? Justify your answer!

Exercise 5.6. Write a *recursive* function `void quickSort(double* x, int n)`, which sorts a given vector $x \in \mathbb{R}^n$ in ascending order using the *quick sort* algorithm. Pick an arbitrary entry from the vector x , called the pivot. Reorder the vector so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this procedure, the pivot is in its final position. Recursively apply the above steps to the subvector of elements with smaller values and separately to the subvector of elements with greater values. Work with dynamically allocated memory. Moreover, write a main program that provides the input vector, calls the function, and prints both the input vector and the sorted vector to the screen. Save your source code as `quicksort.c` into the directory `series05`. Test your implementation with suitable examples. What is the computational complexity of your implementation of `quickSort`? Justify your answer!

Hint. Choose x_1 as pivot. Starting with $j = 2$, search for an element $x_j \geq x_1$, i.e., x_j belongs to the subvector $x^{(\geq)}$ of all elements with values greater than or equal to the pivot. Then, starting with $k = n$, search for an element $x_k < x_1$, i.e., x_k belongs to the subvector $x^{(<)}$ of all elements with values smaller than the pivot. In that case, swap x_j and x_k . If j and k coincide, then x has the form $(x_1, x^{(<)}, x^{(\geq)})$. Then, the form $(x^{(<)}, x_1, x^{(\geq)})$ can be obtained immediately and the pivot is in its final position. It remains to sort $x^{(<)}$ and $x^{(\geq)}$ recursively.

Exercise 5.7. Write a function `void cut(double* x, int* n, double cmin, double cmax)`, which, given a vector $x \in \mathbb{R}^n$ and $c_{\min}, c_{\max} \in \mathbb{R}$, removes from the vector all entries x_j such that $x_j < c_{\min}$ or $x_j > c_{\max}$. For instance, for $c_{\min} = 0$ and $c_{\max} = 10$, the vector $x = (-4, 3, -5, 1, 7, 3, 11, -1) \in \mathbb{R}^8$ should be replaced by the vector $x = (3, 1, 7, 3) \in \mathbb{R}^4$. Work with dynamically allocated memory (the input vector must be overwritten with the shortened one, in particular the length must be adjusted accordingly). Write a main program, which provides the length n , the vector $x \in \mathbb{R}^n$, and the thresholds c_{\min}, c_{\max} , calls the function, and prints both vectors (the original one and the shortened one) to the screen. Test your implementation with suitable examples. Save your source code as `cut.c` into the directory `series05`.

Exercise 5.8. Write a function `void unique(double* x, int* n)`, which sorts a given vector $x \in \mathbb{R}^n$ in ascending order and eliminates all entries that appear more than once. For instance, the vector $x = (4, 3, 5, 1, 4, 3, 4) \in \mathbb{R}^7$ should be replaced by the vector $x = (1, 3, 4, 5) \in \mathbb{R}^4$. Work with dynamically allocated memory (the input vector must be overwritten with the shortened one, in particular the length must be adjusted accordingly). To sort the vector, use a sorting algorithm of your choice, e.g., one implemented in one of the previous exercises. Write a main program that provides the vector x and its length n , call the function, and prints to the screen both the vectors (the original one and its sorted/shortened version). Test your implementation with suitable examples. Save your source code as `unique.c` into the directory `series05`.