Exercise 3.1  Sorting methods.

a) A sequence \( f_1, f_2, \ldots, f_n \) is called a Min-Heap if \( f_i \leq f_{2i} \) for every \( i \) with \( 2i \leq n \), and \( f_i \leq f_{2i+1} \) for every \( i \) with \( 2i+1 \leq n \). For the sorted sequence it holds that \( f_i \leq f_j \) for every \( j \geq i \), therefore the sorted sequence is a Min-Heap.

b) A sequence \( f_1, f_2, \ldots, f_n \) is called a Max-Heap if \( f_i \geq f_{2i} \) for every \( i \) with \( 2i \leq n \), and \( f_i \geq f_{2i+1} \) for every \( i \) with \( 2i+1 \leq n \). Then, the smallest element will be in a position \( i \) such that \( 2i > n \). Therefore, we will find it in the second half of the array.

c) In the first step there are two ,,bad“ pivots (the smallest and the biggest element in the array), and the probability to take one of these is \( 2/n \). At the next step, this probability is \( 2/(n-1) \), etc. Since each choice of the pivot is independent from the previous choices, we can multiply their probabilities, and the overall result is

\[
\prod_{i=2}^{n} \frac{2}{i} = \frac{2^{n-1}}{n!}.
\]

For \( n = 10 \) the probability is ca. 0.014\%, and for \( n = 20 \) it is 0.000000000022\%.

d) Insertionsort and Bubblesort are already stable in the naive implementation. Mergesort can easily be made stable, if we remember to take the leftmost element when a tie is encountered while merging. There is no easy way to make the Selectionsort, Quicksort and Heapsort stable.

e) Selectionsort, Insertionsort, Bubblesort and Heapsort work directly on the array to be sorted, and are therefore in-situ. Quicksort requires between \( \Omega(\log n) \) and \( O(n) \) additional space for storing the recursive function calls. This additional space is not used for elements of the sequence, therefore the Quick sort is also in-situ. In the Mergesort, parts of the array must be copied in the for merging. There are (complicated) methods to perform the merge in-situ, but no such methods can be implemented as simple modifications of the standard algorithm.

f) Even though Quicksort has a worst-case running time of \( \Theta(n^2) \), with random selection of the pivots the probability to get the quadratic running time is extremely small (as evidence, but not as proof, see part c)). The expected running time of Quicksort is \( O(n \log n) \). As we have seen in part e), Quicksort is in-situ, while Merge sort is not. This is, in practice, a very big advantage of Quicksort. Furthermore, a much smaller constant is ,,hidden“ in the expected running time of \( O(n \log n) \) of Quicksort than in the one of Mergesort (which also comes, in part, from the need to use more additional memory for merging).

g) If no two numbers in the sequence are equal, we need \( \Omega(\log n) \) digits (depending on the number system, e.g. bits) for the representation of numbers. This means, among other things, that Radixsort requires \( \Omega(\log n) \) iterations and therefore its overall running time is \( \Omega(n \log n) \). In general it hence makes no sense to refer to Radixsort as a ,,linear-time sorting algorithm“.

Only when we restrict the number of digits to a constant we get a running time of \( \Theta(n) \) for Radixsort, because the constant ,,disappears“ in the asymptotic notation. However, this means that there have to be numbers in the input sequence that appear very often.
Exercise 3.2  Various (Part of an exam in August 2011).

a) \( x = 7480, \ y = 9351 \) oder \( x = 5193, \ y = 8074 \).

b) Folge: 5, 4, 3, 2, 1

c)  
<table>
<thead>
<tr>
<th>25</th>
<th>60</th>
<th>32</th>
<th>61</th>
<th>62</th>
<th>52</th>
<th>57</th>
<th>80</th>
<th>86</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Exercise 3.3  Algorithm design: sums of numbers.

a) First, we sort \( A \) in ascending order. This takes \( \mathcal{O}(n \log n) \) steps (using, for example, Mergesort or Heapsort). Then we check, for any choice of \( a \) in the array (we have \( n \) candidates), whether the value \( z - a \) occurs in the array (in this case, we would have found two numbers such that \( a + b = z \)). Since \( A \) is sorted, we can determine whether \( z - a \) occurs in \( A \) in \( \mathcal{O}(\log n) \) steps, using binary search. If the value occurs, we have found a solution, otherwise we try the next candidate, etc.

Overall, this solution requires \( \mathcal{O}(n \log n) \) time, but in part b) we present a more efficient alternative how to proceed once the array has been sorted.

Note: The passionate algorithmist will ask whether a better running time can be achieved. However, this is not possible: the problem posed here is equivalent to the so-called 2-SUM-Problem\(^1\): „Given a list of integers (not necessarily positive), are there two elements \( a, b \) such that \( a + b = 0 \)?“ For these problems, it can be shown that any correct algorithm has a running time of at least \( \Omega(n \log n) \) (details can be found in „Refined Upper and Lower Bounds for 2-SUM“, A.C. Chan, W.I. Gasarch, C.P. Kruskal, 1997).

Example Code:

```cpp
bool containsSum(const vector<int>& A, int z)
{
    vector<int> B = A;
    sort(B.begin(), B.end());
    for(int i = 0; i < (int)B.size(); ++i)
    {
        int a = A[i];
        int b = z - a;

        int l = 0, r = (int)B.size() - 1;
        while(l < r)
        {
            int m = (l + r) / 2;
            if(B[m] < b)
                l = m + 1;
            else
                r = m;
        }
        if(B[l] == b)
            return true;
    }
    return false;
}
```

\(^1\)To be more accurate: each instance of 2-SUM can be transformed in time \( \mathcal{O}(n) \) into an instance of this problem.
b) We can of course proceed exactly as in the previous step, except that we do not have to sort anymore. However, the running time will still be $O(n \log n)$ because we have to invoke a binary search up to $n$ times to search for the value $z - a$.

A running time of $O(n)$ can be achieved, using the following consideration: let $l, r$ be the indices of the left and right ends of the array. If $A[l] + A[r] = z$, then we return $A[l]$ and $A[r]$ and terminate the procedure. If $A[l] + A[r] > z$, then it holds that $A[k] + A[r] > z$ for every $k$ with $l \leq k < r$ (since $A$ is sorted, we have $A[k] \geq A[l]$). Therefore, there are no elements in the array between positions $l$ and $r$ that sum to $z$ with $A[r]$.

Therefore, we can consider only elements in the array with indices $\leq r - 1$ and repeat the procedure.

If, however, we have $A[l] + A[r] < z$, then it holds that $A[l] + A[k] < z$ for every $k$ with $l < k \leq r$. Therefore, there are no elements in the array between positions $l$ and $r$ that sum to $z$ with $A[l]$ (since the sequence is sorted). We can then consider only elements in the array with index $\geq l + 1$. We set $l := l + 1$ and repeat the procedure.

We stop the procedure if we find a solution, or if $l = r$. In this way, we will always find a pair $a, b$ with $a + b = z$ if such a pair exists. The running time of $O(n)$ can be proved by considering the development of $r - l$: in every step that does not terminate the procedure, either $r$ is decreased by one or $l$ is increased by one (but never both). That is, $r - l$ decreases by one in every step. In the first step we start with $r - l = n - 1$, and therefore the procedure terminates after at most $n - 1$ steps.

Note: This procedure could of course also be applied in part a), and it would be more efficient than the binary search we used there. Since the sorting alone requires $\Omega(n \log n)$ steps, the total asymptotic running time would however not improve.

Example Code:

```cpp
bool containsSumSorted(const vector<int>& A, int z)
{
    int l = 0, r = (int)A.size() - 1;
    while (l < r)
    {
            --r;
            ++l;
        else
            return true;
    }
    return false;
}
```

c) Here we can sort and then use the same idea as in b) to achieve a running time of $O(n^2)$: for each of the $n$ elements in $A$, we set it as a candidate for $z$, and look for a possible pair with the algorithm in b).

Note: The algorithmist will ask again whether a better running time can be achieved. The answer to this question is unknown, which makes this an open problem in computer science.

Example Code:

```cpp
bool contains3Sum(const vector<int>& A)
{
    vector<int> B = A;
    sort(B.begin(), B.end());

    for (int i = 0; i < (int)B.size(); ++i)
    {
        if (containsSumSorted(B, B[i]))
            return true;
    }
    return false;
}
```