Exercise 2.1  
Recurrence relations.

We conjecture that the closed form of the recurrence relation is the following series:

\[ T(n) = a^{\log_b(n)}e + cn \cdot \sum_{i=0}^{\log_b(n)-1} \left(\frac{a}{b}\right)^i + d \cdot \sum_{i=0}^{\log_b(n)-1} a^i. \]

We distinguish three cases:

\( a \neq b, a \neq 1 \): In this case, our conjecture becomes:

\[ T(n) = a^{\log_b(n)}e + cn \cdot \left(\frac{a}{b}\right)^{\log_b(n)-1} \frac{a^{\log_b(n)} - 1}{a - 1} + d \cdot \frac{a^{\log_b(n)} - 1}{a - 1}. \]

**Base step:** The conjecture is true for \( n = 1 \), because \( T(1) = a^0e + c \cdot 0 + d \cdot 0 = e \).

**Inductive Hypothesis:** We assume our conjecture to be true for \( T(n/b) \), so (it holds \( \log_b\left(\frac{n}{b}\right) = \log_b(n) - 1 \))

\[ T\left(\frac{n}{b}\right) = a^{\log_b(n/b)}e + \frac{n}{b} \left(\frac{a}{b}\right)^{\log_b(n/b)-1} \frac{a^{\log_b(n/b)} - 1}{a - 1} + d \cdot \frac{a^{\log_b(n/b)} - 1}{a - 1}. \]

**Inductive step:** We show that the conjecture holds for \( T(n) \) by using the inductive hypothesis:

\[
\begin{align*}
T(n) &= aT\left(\frac{n}{b}\right) + cn + d \\
\quad &= a \left( a^{\log_b(n/b)}e + \frac{n}{b} \left(\frac{a}{b}\right)^{\log_b(n/b)-1} \frac{a^{\log_b(n/b)} - 1}{a - 1} + d \cdot \frac{a^{\log_b(n/b)} - 1}{a - 1} \right) + cn + d \\
&= a^{\log_b(n)}e + \frac{n}{b} \left(\frac{a}{b}\right)^{\log_b(n)} \frac{a^{\log_b(n)} - 1}{a - 1} + d \frac{a^{\log_b(n)} - 1}{a - 1} + a \cdot \frac{1}{b - 1} - a + d \frac{1}{a - 1} \\
&= a^{\log_b(n)}e + \frac{n}{b} \left(\frac{a}{b}\right)^{\log_b(n)} \frac{a^{\log_b(n)} - 1}{a - 1} + d \frac{a^{\log_b(n)} - 1}{a - 1}.
\end{align*}
\]

\( a \neq b, a = 1 \): In this case, our conjecture becomes:

\[ T(n) = e + cn \cdot \left(\frac{1}{b}\right)^{\log_b(n)} \frac{1}{b - 1} + d \log_b(n) = d \log_b(n) + c \cdot b \frac{1 - n - b}{1 - b} + e. \]

**Base step:** The conjecture is true for \( n = 1 \), because \( T(1) = d \cdot \log_b(1) + c \cdot b \frac{1 - n - b}{1 - b} + e = 0 + 0 + e \).

**Inductive Hypothesis:** We assume our conjecture to be true for \( T(n/b) \), so

\[ T\left(\frac{n}{b}\right) = d \log_b\left(\frac{n}{b}\right) + c b \frac{1 - n}{1 - b} + e. \]
**Inductive step:** We show that the conjecture holds for \( T(n) \) by using the inductive hypothesis:

\[
T(n) = T(n/b) + cn + d
\]

Ind. Hyp.
\[
d \log_b(n) + c \frac{b-n}{1-b} + cn + e
\]
\[
d \log_b(n) + c \frac{b-n+n-bn}{1-b} + e
\]
\[
d \log_b(n) + c \frac{1-n}{1-b} + e.
\]

**Base step:** The conjecture is true for \( n = 1 \), because
\[
T(1) = 1 \cdot e + c \cdot 0 + d \frac{1}{a-1} = e.
\]

**Inductive Hypothesis:** We assume our conjecture to be true for \( T(n/b) \), so
\[
T(n/b) = a \cdot \frac{n}{b} e + c \frac{n}{b} \log_b(n/b) + d \frac{n-1}{a-1}. \]

**Inductive step:** We show that the conjecture holds for \( T(n) \) by using the inductive hypothesis:

\[
T(n) = a T(n/b) + cn + d
\]

Ind. Ann.
\[
a \left( \frac{n}{b} e + c \frac{n}{b} \log_b(n/b) + d \frac{n-1}{a-1} \right) + cn + d
\]
\[
a \frac{n}{b} e + c \frac{n}{b} \log_b(n) - a \frac{n}{b} c n + c n + d \frac{a n - a + a - 1}{a - 1}
\]
\[
a = b, (a \neq 1): \text{ In this case, our conjecture becomes:}
\]
\[
T(n) = ne + cn \log_b(n) + d \cdot \frac{n-1}{a-1}.
\]

**Exercise 2.2** Estimate asymptotic running time (part of an exam in January 2012).

No explanation was required for this exercise, nevertheless, we give some reasons for the running times.

a) The first (outermost) loop runs exactly \( n \) times. For each iteration of the first loop, the second one runs exactly \([\log_2 n]\) times, and for each iteration of the second loop, the third one (innermost) runs exactly \([n/10]\) times. Overall, the asymptotic running time is \( \Theta(n^2 \log n) \).

b) The first (outermost) loop runs exactly \([n/2]\) times. For each iteration of the first loop, the second one runs exactly \([\log_3 n^2]\) = \([2 \log_3 n]\) times, and the third one runs \([\sqrt{n}] - 1\) times. Overall, the asymptotic running time is \( \Theta(n \log n + n \sqrt{n}) = \Theta(n \sqrt{n}) \).

c) The outermost loop runs exactly \([n/2]\) times. In the \( j \)-th iteration of the first loop, the innermost loop runs exactly \([n^3/j]\) times. The overall running time is (since the Harmonic series grows logarithmically)

\[
\sum_{i=1}^{[n/2]} \frac{n^3}{2i-1} \approx n^3 \sum_{i=1}^{[n/2]} \frac{1}{2i} = \frac{n^3}{2} \sum_{i=1}^{[n/2]} \frac{1}{i} = \Theta(n^3 \log n).
\]
Exercise 2.3  

Comparison of sorting algorithms.

<table>
<thead>
<tr>
<th></th>
<th>insertionSort</th>
<th>selectionSort</th>
<th>bubbleSort</th>
<th>quickSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>max</td>
<td>$\Theta(n^2)$</td>
<td>any</td>
<td>any</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Input sequence</td>
<td>1, 2, \ldots, n</td>
<td>n, n - 1, \ldots, 1</td>
<td>any</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>Permutations</td>
<td>0</td>
<td>$\Theta(n^2)$</td>
<td>0</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>min</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>max</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n)$</td>
<td>1, 2, \ldots, n</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

(\ast): The appropriate sequence is not easy to write. The sequence must be designed such that every chosen pivot halves the area that will be sorted. For \( n = 7 \), the sequence will be: 4, 5, 7, 6, 2, 1, 3.

Exercise 2.4  

Algorithm design: divide-and-conquer.

a) With the „divide-and-conquer“ paradigm we get to the following solution: the array is divided in two parts of equal size (for simplicity we will assume that \( n \) is even). The key observation is: if a majority element exists in \( A[1..n] \) with more than \( n/2 \) occurrences, then we can find it in one of the two halves more than \( n/4 \) times. (If it occurs \( \leq n/4 \) times in one of the two halves, then it occurs in \( A[1..n] \) only at most \( n/2 \) times). It follows that, if an element is a majority in \( A[1..n] \), then it is a majority element also in at least one of the two halves, either in \( A[1..n/2] \) or in \( A[n/2+1..n] \). The same consideration holds also if \( n \) is odd. We can therefore determine recursively whether a majority element exists in one of the two halves. This gives us one, two or no „candidates“ (possible elements) for the majority element of \( A[1..n] \). For each of these candidates we can easily check whether it is the majority element of \( A[1..n] \) by going through the array and counting how many times each candidate appears.

Analysis: Let \( T(n) \) be the running time of the algorithm over \( n \) elements. Then, we get: \( T(n) = 2T(n/2) + c \cdot n + d \), and \( T(1) = e \), where \( c, d, e \) are constants. We can solve this recurrence relation, or we can use the solution of exercise 2.1. This gives us \( T(n) = ne + cn \log_2(n) + d(n - 1) = \Theta(n \log n) \).

Code:

```c
// returns the majority element or "-1" if none exists (we assume A[i]>0)
int majority(const vector<int>& A, int l, int r)
{
    if(l == r)
        return A[l];
    int m = l + (l + r) / 2;
    int candidate1 = majority(l, m-1);
    int candidate2 = majority(m, r);

    int count1 = 0, count2 = 0;
    for(int i = l; i <= r; ++i)
    {
        if(A[i] == candidate1)
            ++count1;
        if(A[i] == candidate2)
            ++count2;
    }
    int n = r - l + 1;
    if(count1 > n/2)
        return candidate1;
    else if(count2 > n/2)
        return candidate2;
    else
        return -1;
}
```
b) It is possible to find the majority element in linear time (i.e., \( O(n) \)). A possible approach works as follows: we go through the array from the beginning to the end one time, and we keep a current candidate for the majority element, and a counter. Initially, we set the counter to 0 and the candidate to some arbitrary element (it does not matter as long as the counter is 0). Now, we look at each element in the array. There are three possible cases:

(a) The current element is not equal to the candidate, and the counter is 0:
   In this case, we set the candidate to the current element, and the counter to 1.

(b) The current element is not equal to the candidate, and the counter is > 0:
   In this case, we reduce the counter by 1.

(c) The current element is equal to the candidate:
   In this case, we increment the counter by 1.

We will show that if there is a majority element, then it must be the last candidate, the one we get after going through the entire array. We need to check again this candidate, and count the exact number of its occurrences in the array (since the counter at the end does not represent the number of its occurrences). It is not trivial to prove this algorithm’s correctness.

**Correctness proof:** We will prove the following statement by induction on the length of the array: *if the array contains a majority element \( m \), then the candidate at the end is \( m \), and the counter is > 0.* Beware: the opposite is not correct!

**Base step:** The statement is trivial for arrays of length 1.

**Inductive hypothesis:** We assume the statement true for arrays with less than \( n \) elements.

**Inductive step:** Using the inductive hypothesis, we show that the statement is true also for arrays of length \( n \). We consider two cases. In the first case, the counter is never decreased to 0. Then the number of occurrences of the first element must be greater than the overall number of the other elements. Since this element was the candidate from the beginning of the computation, the statement is correct in this case. In the second case, the counter is decreased to 0 after \( k \) steps. By inductive hypothesis, we know that the first \( k \) elements do not contain a majority element. If there is a majority element \( m \), then it is a majority element also among the following \( n - k \) elements. Since the counter is reset to the initial state after \( k \) steps, we can consider the remaining elements separately. By inductive hypothesis applied to the remaining \( n - k \) elements, if there is a majority element \( m \), then the counter is > 0 and the candidate is \( m \).

**Code:**

```c
// returns the majority element or "-1" if none exists (we assume A[i]>0)
int majority(const vector<int> & A, int 1, int r) {
    int candidate, count = 0;
    for(int i = 1; i <= r; ++i) {
        if(candidate != A[i] && count == 0) {
            candidate = A[i];
            count = 1;
        } else if(candidate != A[i] && count != 0) {
            count--;
        } else if(candidate == A[i]) {
            ++count;
        }
    }

    count = 0;
    for(int i = 1; i <= r; ++i) {
        if(A[i] == candidate) {
            ++count;
        }
    }

    if(count > r - 1 + 1)
```

---

4
return candidate;
else
    return -1;
}