Solutions to problems from exercise class 4

1. The basic operation is the assignment to `diff[i][j]`. It is performed $n \times n = n^2$ times. This is quadratic in the matrix order n. It is, however, linear in the total number of elements in the input matrices, which is $2n^2$, and hence linear in the input size.

2. The seven functions in the order of increasing order of growth:

   $2\log_2((n+50)^5), \quad (\ln n)^3, \quad \sqrt{n}, \quad 0.05n^{10}+3n^3+1 \quad 3^{2n}, \quad 3^{3n}, \quad (n^2+3)!$

   To see why:

   - Consider what happens to the functions $2\log_2(n+50)^5$, $(\ln n)^3$ and $\sqrt{n}$, when you replace n by n^2. The first function grows roughly with a factor 2, the second function with a factor 8, the third function with a factor $\sqrt{n}$ (from $\sqrt{n}$ to n).
   - Observe that $\sqrt{n} = n^{\frac{1}{2}}$, which grows much slower than n^{10}.
   - Observe that $3^{2n} = 9^n$ and that $3^{3n} = 27^n$.
   - Observe that $(n^2+3)!$ is the product of n^2+3 factors, most of which are larger than n.

3. (a) This algorithm computes the difference between the last number in array A that is greater than 100, and the last number in array A that is smaller than 100. If there is no number in A greater (smaller) than 100, we use the default value 0 instead. In fact, the variable `sumgreater` may be read as `somegreater`, (and similar for `sumless`), as it a quite arbitary choice from all numbers in A that are greater than 100.

   (b) The basic operation is one of the two comparisons between $A[i]$ and `val`. Both comparisons serve equally well as basic operation.

   (c) The basic operation is executed n times.

   (d) The efficiency class is $\Theta(n)$, i.e., linear in the input size.

   (e) An improvement could be to perform the second comparison only if the condition `(A[i]>val)` does not hold:

```
if (A[i]>val)
  sumgreater = A[i];
else
  if (A[i]<val)
    sumless = A[i];
```

   Indeed, if `(A[i]>val)` holds, then `(A[i]<val)` certainly does not hold, so it is no use testing this. With this improvement, the efficiency class remains $\Theta(n)$, since the first comparison is still performed for every element $A[i]$. In fact, if all elements $A[i]$ happen to be smaller than 100, then also the second comparison is still performed for every element $A[i]$.

   A more significant improvement would be to search the array A in reverse order, starting from $A[n-1]$, and to stop as soon as you find an element greater than `val`. This element is assigned to `sumgreater`. Repeat this search,

this time stopping as soon as you find an element smaller than `val`, which then is assigned to `sumless`. Of course, you should also stop a search after having compared $A[0]$.

In the worst case, this improved algorithm is still in $\Theta(n)$. In the best case, both searches stop after 1 or 2 comparisons, and thus in $\Theta(1)$ time. Assuming that the elements $A[i]$ have a constant (positive) probability of being larger than 100 and a constant (positive) probability of being smaller than 100, the average number of iterations of both searches is hardly dependent on n. Hence, the average case complexity is also in $\Theta(1)$.

4. (a) This algorithm sorts the numbers in array A in increasing order. This can be understood from the following invariant for the outer for-loop:

   $0 \leq i \leq n-1$, and the numbers $A[0], A[1], \ldots, A[i]$ are already sorted in increasing order.

   In fact, this algorithm is an inefficient version of insertionsort.

   (b) A proper basic operation would be the comparison of two array elements $A[i] \leq A[i+1]$ in the condition of the inner while-loop. This is a relatively 'expensive' operation that is executed every time (or almost every time) the condition of the inner while-loop is tested.

   (c) In the best case, the numbers in array A are already sorted in increasing order. In that case, the basic operation is executed $n-1$ times: once for each of $i = 0, 1, \ldots, n-2$.

   In the worst case, the numbers in array A are sorted in decreasing order. In that case, the basic operation is executed a total of

   $$1+2+1+3+1+1+4+1+1+1+5+1+1+1+1+6+\cdots+n-1 + \underbrace{1+\cdots+1}_{n-2 \text{ times}} + n-1 =$$
   $$1+0+2+1+3+2+4+3+5+4+6+\cdots+n-1+n-2+n-1 =$$
   $$1+2+3+4+5+6+\cdots+n-1+n-1+0+1+2+3+4+\cdots+n-2 =$$
   $$\frac{1}{2}n(n-1)+n-1+\frac{1}{2}(n-2)(n-1) = n(n-1)$$

   times. Here, every number in the first line corresponds to an iteration of the outer while-loop.

   (d) It follows from part (c) that the worst case time complexity is in $O(n^2)$.

5. (a) A recurrence relation for $A(n)$ is as follows:

$$A(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + A(n-1) + A(n-2) & \text{if } n \geq 2 \end{cases}$$

   As a result,

n	0	1	2	3	4	5	6	7
$A(n)$	0	0	1	2	4	7	12	20
$F(n)$	0	1	1	2	3	5	8	13

   From this, we conclude that $A(n) = F(n+1) - 1$ for $n \geq 0$. It is not too hard to prove this formally, when you observe that the recurrence $A(n) = 1+A(n-1)+A(n-2)$ is equivalent to $A(n)+1 = (A(n-1)+1)+(A(n-2)+1)$, which is very similar to the recurrence of the Fibonacci numbers.

(b) A recurrence relation for $C(n)$ is as follows:

$$C(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ C(n-1) + C(n-2) & \text{if } n \geq 2 \end{cases}$$

A recurrence relation for $Z(n)$ is as follows:

$$Z(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \\ Z(n-1) + Z(n-2) & \text{if } n \geq 2 \end{cases}$$

As a result,

n	0	1	2	3	4	5	6	7
$C(n)$	0	1	1	2	3	5	8	13
$Z(n)$	1	0	1	1	2	3	5	8
$F(n)$	0	1	1	2	3	5	8	13

From this, we conclude that $C(n) = F(n)$ for $n \geq 0$, and that $Z(0) = 1$ and $Z(n) = F(n-1)$ for $n \geq 1$. Indeed, the recurrence relation for $C(n)$ is equal to that for $F(n)$.

6. (a) Let $M(n)$ be the number of multiplications made by the algorithm for an $n \times n$ matrix. When we count $s_j \cdot a_{0,j} \cdot \det A_j$ as a single multiplication (not two, as s_j is just plus or minus 1), a recurrence relation for $M(n)$ is as follows:

$$M(n) = \begin{cases} 0 & \text{if } n = 1 \\ n + n \cdot M(n-1) & \text{if } n \geq 2 \end{cases}$$

because for $n \geq 2$, we have n multiplications in the original call of the recursive algorithm, while we must compute the determinant of n $(n-1) \times (n-1)$ matrices.

(b) When we substitute $n = 2$ in the recurrence relation, we find that $M(n) = 2$, which is equal to 2!. Now, it is not hard to tell from the recurrence relation (and to prove by induction), that $M(n)$ grows at least as hard as $n!$. As a result, $M(n) \geq n!$ for $n \geq 2$.