Fundamental Algorithms

Solution example

General Instructions

**Material:** You may only use one hand-written sheet of paper (size A4, on both pages). Any other material including electronic devices of any kind is forbidden.

Use only the exam paper that was handed out to solve the exercises – for notes and sketches, you can obtain additional exam sheets.

Do not use pencil, or red or green ink.

**General hint:** Often, exercises b), c), etc. can be solved without the results from the previous exercise a): if you are stuck with exercise a), then don’t immediately skip exercises b), c), etc.

**Maximum score:** The total number of points for all exercises is 47, however grades will be computed relative to a maximum score of 42 points.

**Working time:** 90 minutes + 5 minutes reading time.

Please switch off your cell phones!

Points and Grades:

<table>
<thead>
<tr>
<th>0.0–8.5</th>
<th>9.0–12.0</th>
<th>12.5–15.5</th>
<th>16.0–17.5</th>
<th>18.0–19.5</th>
<th>20.0–21.5</th>
<th>22.0–23.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>4.7</td>
<td>4.3</td>
<td>4.0</td>
<td>3.7</td>
<td>3.3</td>
<td>3.0</td>
</tr>
<tr>
<td>24.0–26.0</td>
<td>26.5–28.5</td>
<td>29.0–31.0</td>
<td>31.5–33.5</td>
<td>34.0–36.0</td>
<td>36.5–</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>2.3</td>
<td>2.0</td>
<td>1.7</td>
<td>1.3</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
1 Finding the Smallest Difference (4+5+4 = 13 points)

For a given array A of integers (in arbitrary order), we want to find the two array elements A[i] and A[j] that have the smallest difference \(\text{abs}(A[i] - A[j])\), where \(\text{abs}\) computes the absolute value. The following divide-and-conquer algorithm is our first attempt to solve this problem:

```plaintext
QuickDiff(A: Array[p . . r]) : (Integer, Integer) {
    // for small arrays, check all combinations:
    if (p−r <3) then return SimpleDiff(A[p . . r]);

    // for large arrays: partition into two halves:
    q := Partition(A);
    // compute the element pairs with smallest difference
    // in both of the partitions
    (i1, j1) := QuickDiff(A[p . . q]);
    (i2, j2) := QuickDiff(A[q+1 . . r]);

    // return the indices of the pair with the smaller difference:
        then return (i1, j1)
    else return (i2, j2);
}
```

SimpleDiff is a simple algorithm to solve our problem for subarrays with at most 3 elements. Partition is the algorithm you know from Quicksort (using the first array element as “pivot”).

a) Describe shortly and in plain words, what the algorithm Partition (as defined in the lectures) does on the array A, and what the main idea is for its implementation. How many comparisons are required on an input array of size \(n\)?

b) \(T_{\text{QD}}(n)\) shall be defined as the number of all accesses to array elements of A performed by a call to QuickDiff for an array with \(n\) elements. Give a recurrence equation to compute \(T_{\text{QD}}(n)\) and give asymptotic upper bounds for \(T_{\text{QD}}(n)\) for the best, worst and average case (no computation required; refer to similar recurrences solved in the lectures).

For clarification: comparing two elements (as required in Partition) or subtracting two elements requires two accesses to array elements. You can assume that SimpleDiff only requires a constant number of element accesses.

c) The algorithm QuickDiff, as given here, is actually not correct. State what the algorithm does not consider, and describe how the algorithm has to be extended – make sure that the algorithm does not increase the asymptotical complexity!

Solution:

a) Partition picks a pivot element (A[p]) and rearranges the array A such that all elements smaller than the pivot will end up left of the pivot, and all elements larger then the pivot will end up right of the pivot. For that purpose, we look for the leftmost element larger than the pivot and the rightmost element small than the pivot – and exchange the two. This process is repeated until all elements are in the right partition (implemented via respective
“pointer” indices i and j). As every element is compared once with the pivot, we have at most $n$ comparisons.

b) Assume that Partition creates partitions of sizes $n_1$ and $n_2$ (with $n = n_1 + n_2$). Then, we obtain the following recurrence:

$$T_{QD}(n) = \begin{cases} 
T_{QD}(n_1) + T_{QD}(n_2) + cn + 4 & \text{for } n \geq 3 \\
O(1) & \text{for } n < 3 
\end{cases}$$

The term $cn$ results from the number of comparisons in Partition, which are $\Theta(n)$ (note that each element needs to be compared to the pivot). In the best case, the array will be split into two partitions of equal size (i.e., $n_1 = n_2 = n/2$) in each step of recursion, leading to an asymptotic bound of $O(n \log n)$ (cmp. MergeSort). In the worst case, we split off only one element ($n_1 = 1$ and $n_2 = n - 1$ or vice versa), which leads to an upper bound of $O(n^2)$ (QuickSort worst case). In the average case, we expect costs of $O(n \log n)$, as for QuickSort.

c) The algorithm does not work correctly if the smallest distance occurs between two elements that are placed in different partitions. However, as the pivot element separates the two partitions, the pivot element has to be one of the two involved elements in such a case. Hence, you need to add another loop that finds the minimum distance of an element to the pivot element. This can be implemented by $O(n)$ element accesses (simple minimum search). Optional improvement: As the pivot element is placed in the left partition, this check only needs to be done with the elements of the right partition.
Consider the following AVL tree:

\[
\begin{align*}
&n & h(n) = m + 1 \\
&l & h(l) = m & b(l) = 0 \\
&A & h(A) = m - 1 \\
&r & h(r) = m - 1 & b(r) = 0 \\
&B & h(B) = m - 2 \\
&C & h(C) = m - 2 \\
&D & h(D) = m - 1
\end{align*}
\]

a) Compute the heights \( h \) and height balances \( b \) required to show that the tree satisfies the AVL property (assuming that the subtrees \( A, B, C \) and \( D \) all satisfy the AVL property).

b) We now insert a single additional element into the subtree. Assume that the AVL property is now violated only(!) in the top node \( n \). Describe all possibilities where the element could have been added to cause this violation.

c) For all situations, name the respective rotation (if any) that is required to restore the AVL property. Perform this rotation for one situation and show that the height of the tree is the same as before the insertion.

Solution:

a) As can be seen from the picture below, the height balances of the nodes \( n, l \) and \( r \) are all \( \in \{-1, 0, 1\} \). As \( A, B, C \) and \( D \) all satisfy the AVL property, the entire tree is an AVL tree.

\[
\begin{align*}
&h(n) = m + 1 \\
&b(n) = -1 \\
&h(l) = m & b(l) = 0 \\
&A & h(A) = m - 1 \\
&r & h(r) = m - 1 & b(r) = 0 \\
&B & h(B) = m - 2 \\
&C & h(C) = m - 2 \\
&D & h(D) = m - 1
\end{align*}
\]

b) Depending on into which subtree the element is inserted, we have the following situations:
**insert into** $A$: this can increase the height of $A$ to $h(A) = m$ and consequently $h(l) = m + 1$, which changes the height balance in the top node to $b(n) = -2$ (i.e., AVL property is violated). In this case, we need to perform a right rotation:

\[
\begin{align*}
&h(l) = m + 1 \\
&b(l) = 0 \\
&h(A) = m \\
&h(r) = m - 1 \\
&h(B) = m - 2 \\
&h(C) = m - 2 \\
&h(D) = m - 1 \\
&h(n) = m \\
&b(n) = 0
\end{align*}
\]

**insert into** $B$ or $C$: this can increase the height of $B$ or $C$, such that $h(r) = m$ and $h(l) = m + 1$, which again changes the height balance in the top node to $b(n) = -2$ (i.e., AVL property is violated). In this case, we need to perform a left-right rotation (shows only $B$).

\[
\begin{align*}
&h(r) = m + 1 \\
&b(r) = 0 \\
&h(l) = m \\
&b(l) = 0 \\
&h(A) = m - 1 \\
&h(B) = m - 1 \\
&h(C) = m - 2 \\
&h(D) = m - 1 \\
&h(n) = m \\
&b(n) = +1
\end{align*}
\]

**insert into** $D$: leads to $h(D) = m$, such that $b(n) = 0$; the tree still satisfies the AVL property and its height doesn’t change.

c) See pictures for solution of b). Only one rotation needs to be shown, the other named.

### 3 Correctness of a Parallel Prefix Algorithm ($\approx 3+8 = 11$ pts)

Given is the following parallel algorithm (EREW variant, as discussed in the lectures) for the prefix problem:

```plaintext
PrefixPRAM( A: Array[1..n] ) {
    // n assumed to be 2^k
    // Model: EREW PRAM (n−1 processors)
```


for $L$ from 0 to $k{-}1$ do
    for $j$ from $2^L+1$ to $n$ do in parallel {
    }
}

a) State a loop invariant for the $L$-loop (i.e., sequential outer loop) that helps to prove its correctness.

b) Prove the correctness of PrefixPRAM using this loop invariant. You may assume that the $j$-loop (i.e., parallel inner loop) is correct. However, draw a diagram to explain precisely what the $j$-loop computes to make the entire algorithm correct.

Solution:

![Diagram showing the computation of PrefixPRAM](image)

a) Before entering the body of the $L$-loop, the following properties are satisfied:
   - all array elements $A[1..2^L]$ contain the correct result of the prefix problem: $A_1 \cdot A_2 \cdots A_{2^L}$
   - all array elements $A[k]$ with $k = 2^L + 1 \ldots n$ contain the product of the $2^L$ matrices $A_{k-2^L+1} \cdots A_k$.

b) To prove correctness of an invariant, we have to prove correctness for initialisation, maintenance, and termination. We will start with the proof for maintenance and then try to prove initialisation:

Initialisation: In the first execution of the $L$-loop, $l=0$, thus the claim is that
   - $A[1..2^0] = A[1]$ contains the correct result of the prefix problem and
   - all elements $A[k]$ for $k = 2, \ldots, n$ contain the product of the $2^0$ matrices $A_k, \ldots, A_k$.
Both statements are trivially satisfied by the initial setup.

Termination: After the last execution of the $L$-loop, i.e., when $l=k$, the claim is that
   - $A[1..2^k] = A[1..n]$ contains the correct result of the prefix problem and
   - all elements $A[k]$ for $k = 2^l + 1, \ldots, n$ contain certain products
The second statement does not apply, because $2^l + 1 = n + 1 > n$ and thus no elements are specified. The first statement states that the correct result of the prefix problem is stored in every array element, thus proving correctness of the algorithm.

**Maintenance:** We assume that the loop invariant holds for the loop execution for a specific value of $l$. For the proof, we split up the elements into three sections:

1) the array elements $A[1..2^l]$ already contain the correct result of the prefix problem (due to loop invariant) – these are not changed by the $j$-loop and thus stay correct.

Each array element $A_j$ in $A[2^l+1 .. n]$ contains the product of the $2^l$ matrices $A_{j-2^l+1:j}$ – in the $j$-loop, $A_j$ will be multiplied with the element $A_{j-2}$.

- If $j \leq 2^{l+1}$, then element $A_{j-2}$ is in the subarray $A[1..2^l]$, which, by our assumption, contains the prefix result $A_{1:j-2}$. The product of the matrix multiplication is then $A_{1:j-2}A_{j-2+1:j} = A_{1:j}$, which is the correct result.

- If $j > 2^{l+1}$, then element $A_{j-2}$, by our assumption, contains the product $A_{j-2-2^l+1:j-2} = A_{j-2^l+1:j-2}$. The product of the matrix multiplication is then $A_{j-2^l+1:j-2}A_{j-2^l+1:j} = A_{j-2^l+1:j'}$ which matches the correct condition for entering the next $l$-loop.

**Comment:** very detailed proof given here → include ≈ 2 points “overhead” in points for exercise b).
4 Parallel Matrix-Vector Multiplication (≈ 3+5 = 8 pts)

The following algorithm is suggested to compute the Matrix-Vector product \( y = Ax \) for a band matrix \( A \) (i.e., elements \( A_{ij} = 0, \text{if} \ |i - j| > k \)). The author of the algorithm claims that the algorithm runs correctly on a CRCW-PRAM with \( n(2k+1) \) processors.

```
BandMV_CRCW(A: Array[1..n, 1..n], x: Array[1..n], y: Array[1..n]) {
    for i from 1 to n do in parallel {
        y[i] := 0;
    }
    for i from 1 to n do in parallel {
        for j from i-k to i+k do in parallel {
            if j>=1 and j<=n then y[i] := y[i] + A[i,j]*x[j];
        }
    }
}
```

a) Describe where concurrent read or write accesses occur during parallel execution of this algorithm (on 3n processors). What assumptions have to be made on the concurrent accesses such that the algorithm works correctly?

b) Describe how to turn BandMV_CRCW into an EREW PRAM algorithm. State the number of processors that your EREW algorithm uses. Describe in detail why your algorithm fits into the EREW classification. You may use algorithms known from the lecture to make your algorithm EREW. In that case, describe why this algorithm is EREW.

Solution:

a) Concurrent write accesses happen to \( y[i] \): all \( 2k + 1 \) processors that “share” the same \( i \) will access the same \( y[i] \).

Concurrent read accesses happen to \( x[j] \): for a given index \( j_0 \), up to \( 2k + 1 \) processors will have a \( j \in \{i-k, \ldots , i+k\} \) with \( j = j_0 \), leading to a concurrent access to the respective element.

In contrast no concurrent read access happens to \( A \).

As in the lectures, we only consider accesses to vector/matrix elements, but not to pointers (or similar) required to internally realize the data structure.

b) Option 1: Broadcast elements of vector \( x \) to all \( j \)-processors; requires binary broadcast to \( 2k + 1 \) elements and \( n \times (2k + 1) \)-array \( X \) using \( \log(2k + 1) \) processors. Multiplications need to be stored in a \( n \times (2k + 1) \)-array \( Y \); after parallel loop, binary fan-in is required on this array \( Y \) (again using \( \log(2k + 1) \) processors). In the level-wise fan-in/fan-out only two elements are accessed, which may be sequentialized.

Option 2: Make the \( j \)-loop a sequential loop. Access to \( y \)-elements will trivially turn into EREW (one element per process \( i \)). Access to \( x \) is exclusive-read due to lock-step execution of loop: all if-then-statements will be executed for uniform values of \( j \). Hence, any accesses to \( x \) will have a fixed difference (“offset”) between \( i \) and \( j \), such that all processors access a different element (or no element due to failed if-test).
5 Connected Graphs ($\approx 4+2 = 6$ points)

The following (incomplete) algorithm IsConnected is an attempt to construct an algorithm to test whether a (non-directed) graph is connected, i.e., whether there exists a path between any two nodes of the graph. The graph is represented by its adjacency matrix.

IsConnected ($A$: Array $[1..n,1..n]$) {
  ! Input: adjacency matrix $A$
  ! $A[i,j] = 1$, if $i$ connected to $j$, otherwise 0

  // part 1 of the algorithm:
  for $k$ from 1 to $n$ do
    for $i$ from 1 to $n$ do
      for $j$ from 1 to $n$ do
        if $(A[i,k]=1$ and $A[k,j]=1)$ then $A[i,j] = 1$
    end do
  end do

  // part 2 of the algorithm:
  /* ... -> is missing ... */
}

a) Part 1 of the algorithm adapts the idea of an algorithm that was discussed in the lecture. State the name of this algorithm, explain its main idea and describe how it is exploited here to decide whether the graph is connected.

b) Provide an implementation for the missing “part 2” of the algorithm. It should return true, if the graph is connected, and false otherwise.

Solution:

a) The algorithm adapts Floyd’s algorithm for the all-pairs-shortest-path problem. Its main idea is to successively generate the shortest path (in a weighted directed graph) between all nodes, by checking whether shorter paths can be constructed that only run through the nodes $1,\ldots,k$. In this example, we successively connect nodes via a new edge (i.e., set the adjacency matrix entry to 1), if a path between them exists via the nodes $1,\ldots,k$. Hence, in the innermost loop, we exchange the Floyd Algorithm’s check for a shorter path by adding a connection if both $A[i,k]$ and $A[k,j]$ are already connected (i.e., are 1).

b) In the final check, we need to check whether any node is now connected to any other node, i.e., whether all $A[i,i] = 1$. Hence, if we find an $A[i,i] = 1$ we need to return false. See the following listing for the complete algorithm:

IsConnected ($A$: Array $[1..n,1..n]$) {
  ! Input: adjacency matrix $A$
  ! $A[i,j] = 1$, if $i$ connected to $j$, otherwise 0

  // ... part 1 of the algorithm skipped ...

  // part 2 of the algorithm:
  for $i$ from 1 to $n$ do
for j from 1 to n do
    if A[i,j] = 0 then return false;
    end do
end do
return true;