Technische Universität München
Institut für Informatik
Scientific Computing in Computer Science

WS 2013/2014
Final Exam
23.12.2013

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.

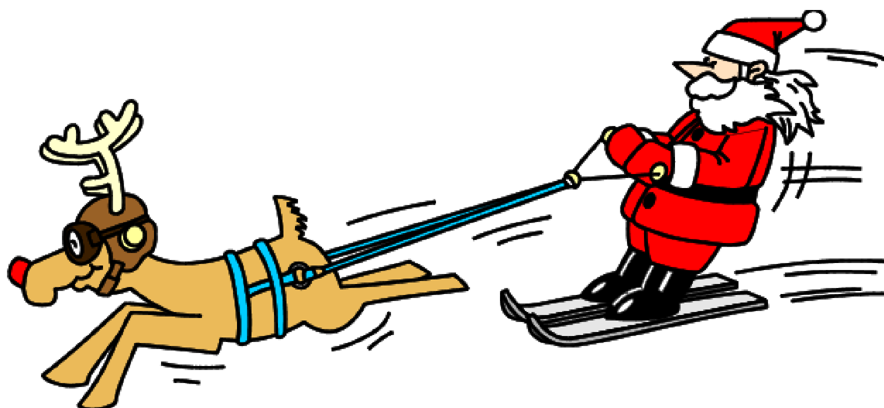
Use only the exam paper that was handed out to solve the exercises – for notes and sketches, you can obtain additional exam sheets.

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.

Working time: 40 minutes + 5 minutes reading time.

Please switch off your cell phones!



Merry Christmas and all the best for the new year!

1 Correctness of InsertionSort

Consider the sorting algorithm InsertionSort, as introduced in the lecture:

```
InsertionSort(A:Array[1..n]) {

   for j from 2 to n {
   // insert A[j] into sequence A[1..j-1]

      key := A[j];

      i := j-1;  //initialize i for while loop
      while i>=1 and A[i]>key {
         A[i+1] := A[i];
         i := i-1;
      }
      A[i+1] := key;
   }
}
```

To prove correctness of the algorithm, someone provides you with the following loop invariant for the while-loop:

At the beginning of the while loop body:

- A[1..i+1] contains the original elements of A[1..i+1] in sorted order
- A[i+2..j] contains the original elements of A[i+1..j-1] in sorted order

a) Prove (or disprove) initialisation and maintenance for this invariant.

b) What does the loop invariant state at termination? Does it help to prove the correctness of the entire InsertionSort? (If not, how do we need to extend the loop invariant?)

Note: the suggested invariant actually contains an error, which makes the exercise more difficult than originally planned – try to work out a correct solution, as well (you will probably need a bit of extra time to figure it out . . .

Solution:

To prove correctness of an invariant, we have to prove correctness for initialisation, maintenance, and termination.

a) We will start with the proof for maintenance and then try to prove initialisation:

Maintenance: We assume that the loop invariant holds for the loop execution for a specific value of i. After copying the value from A[i] into A[i+1], via the second part of the invariant (A[i+2..j] containing the original A[i+1..j-1]), we may infer that then A[i+1..j] contains the original elements A[i..j-1] in sorted order. After decreasing i by one, we have the loop invariant for the next iteration. Note that the subarray A[i+2..j] may also be empty in this maintenance proof!

Initialisation: In the first execution of the while body, i=j-1, thus the claim is that A[1..j] contains all original elements of A[1..j] in sorted order and that A[j+1..j] contains all elements of A[j..j-1]. The latter part is satisfied, because these subarrays are empty. In our maintenance proof,

we allowed the second array to be empty, so this does not lead into problems.
However: the claim that A[1..j] contains all original elements of A[1..j] in sorted order is questionable! According to our comment, we want to insert A[j] into the sorted sequence A[1..j-1]; thus, A[1..j] is not necessarily sorted. At start of the very first while-loop execution, with the value j=2, our claim would be that the array A[1..2] is already sorted. This is not necessarily true.

As our proof for initialisation failed, the invariant is wrong!

As a correct invariant, we may suggest:

At the beginning of the while loop body:

- A[1..i] contains the original elements of A[1..i] in sorted order
- A[i+2..j] is either empty ($i+2 > j$) or contains the original elements of A[i+1..j-1] in sorted order

Again, we try to prove initialisation and maintenance:

Initialisation: In the first execution of the while body, i=j-1, thus the claim is that A[1..j-1] contains all original elements of A[1..j-1] in sorted order and that A[j+1..j] is either empty or contains all elements of A[j..j-1]. The latter part is satisfied, because the subarray is empty. The claim that A[1..j-1] contains all original elements of A[1..j-1] in sorted order is now consistent with our plan to insert A[j] into the sorted sequence A[1..j-1]; for a strict proof of this fact, we'd need to formulate a respective invariant for the outer for-loop on j (may be skipped).

Maintenance: The maintenance proof is the same as for the "wrong" invariant.

b) We will first check termination for the corrected invariant:
At termination, let i := t for some value t. Hence, our invariant for the following execution of the loop body is satisfied (maintenance of the previous execution), but the while-condition was false.
Then, from the invariant, we know that A[1..t] contains all original elements of A[1..t] in sorted order, and A[t+2..j] contains the original elements of A[t+1..j-1] in sorted order. Hence, we may overwrite the array element t+1 by the key, as in the statement A[i+1] := key.
The while loop can terminate in two situations:

(1) A[t] <= key: now A[t+2..j] contains the original elements of A[t+1..j-1]; all these elements are bigger than key (otherwise we'd have terminated earlier); also A[1..t] contains all elements originally in A[1..t]; as the elements are sorted and A[t] <= key, all these elements are <= key; hence, we insert the key at t+1, which is the correct position.

(2) i=t=0: in that case, A[2..j] contains all elements originally in A[1..j-1], and we copy the key into A[1], which is correct, as the key has been smaller than all A[i] before

Hence, at termination, our (corrected) invariant makes sure that we insert the element stored in key at the correct position, thus increasing the sorted subarray A[1..j-1] towards a sorted subarray A[1..j]. We would need this property for a correctness proof of the outer j-loop.

Let's also consider our wrong invariant:
In this case, we claim from the invariant that A[1..t+1] contains all original elements of A[1..t+1] in sorted order, and A[t+2..j] contains the original elements of A[t+1..j-1] in sorted order. In particular, A[t+1] holds the same element as A[t+2]. Hence, we may again conclude that the element A[t+1] may be overwritten by key, and also the check for a finally sorted array A[1..j]

will work. However, note that we have based this proof on a wrong prerequisite (A[1..t+1] being sorted)! Hence, all conclusions we draw may be entirely wrong.

Final comment: please excuse the wrong invariant in the exercise! I originally kept this error, however, because it is an excellent example on what can happen, if you disregard the initialisation: maintenance and termination seem to be correct, but the proof is wrong unless it's based on a correct initialisation, as well.

2 Minimum Search on the PRAM

The following parallel algorithm is suggested for searching the minimum in an array:

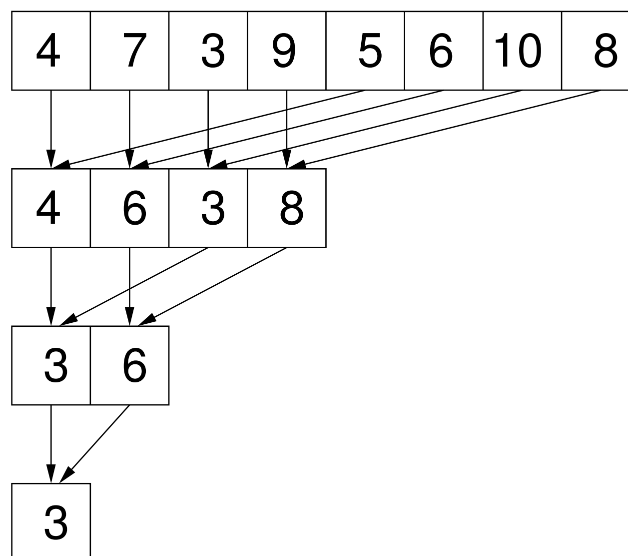
```
MinPRAM( A: Array [1..n]) : Integer {
    // n assumed to be 2^k

    s := n;
    for i from 1 to k do  {
        s := s/2;
        for j from 1 to s do in parallel
            if A[j] > A[j+s]  then A[j] := A[j+s];
    }
    return A[1];
}
```

Explain whether read and write accesses in this PRAM algorithm are exclusive or concurrent. Draw a diagram for the case $n = 8$ to illustrate the access pattern.

Solution:

We first draw the diagram to illustrate the access pattern for $n = 8$:



From the diagram, we can also see that the access to the array elements is strictly exclusive:

- Two processors $j_1 \neq j_2$ will access the elements A[j_1], A[$j_1 + s$], A[j_2], and A[$j_2 + s$]; as $j_1 < s$ and $j_2 <$, the four involved indices are pairwise different.
- Processor j will only write to element $A[j]$, which is not accessed (neither read nor write) by and other processor.

3 QuickSort with Expensive Pivoting

Consider the following variant of the QuickSort algorithm: to determine a good pivot element, we recursively sort the first third of the input array. After sorting, we select the centre element of this third as pivot element. See the following implementation of this algorithm (which – up to the pivoting strategy – is analogous to RandQuickSort discussed in the lectures):

```
TSQuickSort(A:Array[p..r]) {
  n := r-p+1;
  if n<3 then SimpleSort(A)
  else {
    // sort the first third of the array A:
    TSQuickSort(A[p .. p+n/3-1]);
    // make element A[p+n/6] the (new) Pivot element:
    exchange A[p+n/6] and A[p];
    q := Partition(A);
    TSQuickSort(A[p..q]);
    TSQuickSort(A[q+1 .. r]);
  };
}
```

a) In this algorithm, Partition is identical to the respective algorithm defined in the lectures. Describe shortly and in plain words, what the Partition 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) Concerning the quality of the selected pivot element: how many elements are guaranteed to be larger (and smaller) than this pivot element. What are the worst-case sizes of the generated partitions?

c) Set up a recurrence equation for the number $T(n)$ of comparisons executed in the worst case by the algorithm TSQuickSort on an input array of n elements (assuming a suitable implementation of SimpleSort).

d) Apply the substitution method on the recurrence equation derived in Exercise 3c) to explain that $T(n) \notin O(n \log n)$.

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) As we sorted the first $n/3$ elements, the element at position p+n/6 will definitely be smaller and larger than $n/6$ other elements. As we cannot make any assumption on the other $2n/3$ elements, the partition sizes will be $n/6$ and $5n/6$ in the worst case.

c) The recurrence is

$$T(n) = \begin{cases} O(1) & \text{if } n < 3 \\ T(n/3) + T(n/6) + T(5n/6) + n & \text{if } n \geq 3. \end{cases}$$

$T(n/3)$ reflects the cost to sort the first third of the array. $T(n/6) + T(5n/6)$ ist the cost for sorting the two partitions recursively. n comparisons are required by Partition.

d) We assume that $T(n) = an \log n + g(n)$, with $g(n)$ a linear function, solves the recurrence:

$$\begin{aligned}
an \log n + g(n) &= a\frac{n}{3} \log \frac{n}{3} + g(\frac{n}{3}) + a\frac{n}{6} \log \frac{n}{6} + g(\frac{n}{6}) + a\frac{5n}{6} \log \frac{5n}{6} + g(\frac{5n}{6}) + n \\
&= a\frac{n}{3}(\log n - \log 3) + a\frac{n}{6}(\log n - \log 6) + a\frac{5n}{6}(\log n + \log \frac{5}{6}) \\
&\quad + g(\frac{n}{3} + \frac{5n}{6} + \frac{n}{6}) + n \\
&= a\frac{4}{3}n \log n + an \left(\frac{1}{3} \log 3 - \frac{1}{6} \log 6 + \frac{5}{6} \log \frac{5}{6} \right) + g(\frac{4n}{3}) + n
\end{aligned}$$

As this equality has to hold for all n, we require $a = \frac{4}{3}a$, which can only hold for $a = 0$. However, the equality the reduces to $g(n) = g(\frac{4n}{3}) + n$, which again does not lead to a solution. Hence, by contradiction, the number of comparisons cannot be $O(n \log n)$. Our proof indicates that the complexity is larger.