Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
H. Ghodselahi, Y. Maus

October 31, 2016

Algorithm Theory, Winter Term 2016/17
Problem Set 1 - Sample Solution

Exercise 1: Complexity (12 points)

Characterize the relationship between $f(n)$ and $g(n)$ in the following examples by using the O, Θ, or Ω notation. Hence, state for **each** of the following three statements whether $f(n) = \Theta(g(n))$, $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$ holds.
Explain your answers (formally)!

a) $f(n) = n^2 + 1$ $\qquad$ $g(n) = e^{-n^2}$

b) $f(n) = n^\epsilon$ (for a positive constant $\epsilon < \frac{1}{2}$) $\qquad$ $g(n) = \log_2 n$

c) $f(n) = \log_2(n!)$ $\qquad$ $g(n) = n \log_2 n$

d) $f(n) = \lceil \ln n \rceil !$ $\qquad$ $g(n) = n^2$

Solution:

According to definitions of $\mathcal{O}(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$ *(the Landau notation)*:

- $f(n) \in \mathcal{O}(g(n))$ if $\exists$ constant $c > 0$ and $n_0 \in N$, such that for all $n \geq n_0$: $f(n) \leq c \cdot g(n)$,
- $f(n) \in \Omega(g(n))$ if $\exists$ constant $c > 0$ and $n_0 \in N$, such that for all $n \geq n_0$: $f(n) \geq c \cdot g(n)$,
- $f(n) \in \Theta(g(n))$ if $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$

In order to solve this task, some additional facts might be useful:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \exists n_0 \in N$ such that $f(n) < g(n)$ for all $n \geq n_0$
  $\Longrightarrow f(n) \in \mathcal{O}(g(n)), f(n) \notin \Omega(g(n)), f(n) \notin \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow \exists n_0 \in N$ such that $f(n) > g(n)$ for all $n \geq n_0$
  $\Longrightarrow f(n) \in \Omega(g(n)), f(n) \notin \mathcal{O}(g(n)), f(n) \notin \Theta(g(n))$
- $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Longrightarrow f(n) \in \Omega(g(n)), f(n) \in \mathcal{O}(g(n)), f(n) \in \Theta(g(n))$

Now we can continue figuring out the relations between the functions.

a) $f(n) \in \Omega(g(n)), f(n) \notin \mathcal{O}(g(n)), f(n) \notin \Theta(g(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2+1}{\frac{1}{e^{n^2}}} = \lim_{n\to\infty} e^{n^2} \cdot (n^2+1) = \infty$$

b) $f(n) \in \Omega(g(n))$, $f(n) \notin \mathcal{O}(g(n))$, $f(n) \notin \Theta(g(n))$.

Consider some positive constant $c > 1$. Then,

$$\lim_{n\to\infty} \frac{\log_2 n}{n^\epsilon} = \lim_{n\to\infty} \frac{\frac{\log_c n}{\log_c 2}}{(c^\epsilon)^{\log_c n}} = \lim_{n\to\infty} \frac{\log_c n}{\log_c 2 \cdot (c^\epsilon)^{\log_c n}} \overset{(1)}{=} 0.$$

(1) c^ϵ is a constant greater than 1.

From the above inequality, it follows that $f(n) \notin O((g(n))$. Therefore, we have $f(n) \notin \Theta(g(n))$.

c) $f(n) \in \Theta(g(n))$, $f(n) \in \mathcal{O}(g(n))$, $f(n) \in \Omega(g(n))$.

Note: For the sake of simplicity the base of logarithms are supposed to be 2 wherever it is not mentioned.

1) $f(n) \in O(g(n))$

$f(n) = \log n! = \log(n \cdot (n-1) \cdot (n-2) \dots 1) = \log n + \log(n-1) + \dots + \log 1$

$g(n) = n \log n = \underbrace{\log n + \log n + \dots + \log n}_{n \text{ times}}$

$\Rightarrow f(n) \leq g(n)$ for any $n > 1$. Therefore, based on the formal definition of the big $\mathcal{O}$ notation, by taking $c = 1$ and $n_0 = 1$ we have $f(n) \in O(g(n))$.

2) $f(n) \in \Omega(g(n))$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} \log(i)}{n \log n} \geq \lim_{n\to\infty} \frac{\sum_{i=n/2}^{n} \log(i)}{n \log n} \geq \lim_{n\to\infty} \frac{\frac{n}{2} \cdot \log(n/2)}{n \log n} = \frac{1}{2} > 0$$

From (1) and (2), we can conclude that $f(n) \in \Theta(n)$.

d) $f(n) \in \Omega(g(n))$, $f(n) \notin \mathcal{O}(g(n))$, $f(n) \notin \Omega(g(n))$.

Let us consider the following chain of inequalities:

$$k! = \prod_{1}^{k} i \geq \prod_{i=27}^{k} i \geq \prod_{i=27}^{k} 27 = 27^{k-26} = 3^{3^{(k-26)}} = 3^{3k} \cdot 3^{-3 \cdot 26} \geq c \cdot e^{3k}$$

If we consider our function $f(n) = \lceil \ln(n) \rceil !$ in the same manner as $k!$ above, we get:

$$\lceil \ln(n) \rceil ! \geq c \cdot e^{3 \cdot \lceil \ln(n) \rceil} \geq c \cdot e^{3 \cdot \ln(n)} = c \cdot n^3$$

Finally,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq \lim_{n\to\infty} \frac{c \cdot n^3}{n^2} = \infty.$$

Exercise 2: Recurrence Relations (6 points)

a) **(3 points)** Guess the solution of the following recurrence relation by repeated substitution.

$$T(n) \leq 3 \cdot T\left(\frac{n}{3}\right) + c \cdot n \log_3 n, \quad T(1) \leq c$$

where $c > 0$ is a constant.

b) **(3 points)** Use induction to show that your guess is correct.

Remark: You can assume that n equals 3^j for some $j \in \mathbb{N}$.

Solution:

a) First, by using substitution, we achieve a guess for the answer of the inequality.

$$\begin{aligned}
T(n) &\leq 3T(n/3) + cn \log_3 n \\
&\leq 3\big(3T(n/9) + cn/3(\log_3 n - 1)\big) + cn \log_3 n \\
&\leq 9T(n/9) + 2cn \log_3 n - cn \\
&\leq 9\big(3T(n/27) + cn/9(\log_3 n - 2)\big) + 2cn \log_3 n - cn \\
&\leq 27T(n/27) + 3cn \log_3 n - 3cn \\
&\vdots \\
&\leq 3^i T(n/3^i) + icn \log_3 n - \sum_{j=1}^{i-1} j \cdot cn
\end{aligned}$$

By considering $i = \log_3 n$,

$$T(n) \leq cn + \frac{1}{2}cn \log_3^2 n + \frac{1}{2}cn \log_3 n$$

b) Here we use induction to prove our guess achieved by induction.

Induction base: $T(1) \leq c \cdot (1) + \frac{1}{2}c \cdot (1) \log_3^2 1 + \frac{1}{2}c \cdot (1) \log_3 1 = c \leq c$ ✓

Induction hypothesis: $\forall n' < n : T(n') \leq cn' + \frac{1}{2}cn' \log_3^2 n' + \frac{1}{2}cn' \log_3 n'$

Induction step:

$$\begin{aligned}
T(n) &\leq 3T(n/3) + cn \log_3 n \\
&\leq 3\left(c\frac{n}{3} + \frac{1}{2}c\frac{n}{3} \log_3^2 \frac{n}{3} + \frac{1}{2}c\frac{n}{3} \log_3(\frac{n}{3})\right) + cn \log_3 n \\
&\leq cn + \frac{1}{2}cn \log_3^2 n + \frac{1}{2}cn \log_3 n \text{ ✓}
\end{aligned}$$

From the induction we have $T(n) \leq cn + \frac{1}{2}cn \log_3^2 n + \frac{1}{2}cn \log_3 n$, for all $n \geq 1$.

Exercise 3: Find Local Maximum (22 points)

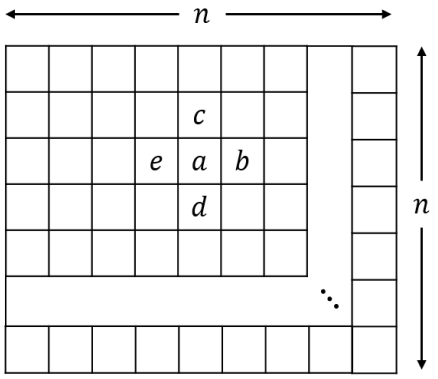

Figure 1: a is a local maximum when a is not smaller than b, c, d, or e.

Consider a one dimensional array $A[1 \dots n]$ such that all values in the array are positive integers. The value $A[i]$ for $i \in \{2, 3, \dots, n-1\}$ is a *local maximum* of the array if

$$A[i-1] \leq A[i] \qquad\qquad A[i+1] \leq A[i] \tag{1}$$

and the value $A[1]$ (resp. $A[n]$) is a local maximum if

$$A[1] \geq A[2] \text{ (resp. } A[n-1] \leq A[n]). \tag{2}$$

In other words, for all $i \in \{1, \dots, n\}$, $A[i]$ is a local maximum if none of its neighboring values are larger.

a) **(4 points)** Prove that any array with positive integers must contain at least one local maximum.

b) **(8 points)** Devise a divide and conquer algorithm to find a local maximum of A in $\mathcal{O}(\log n)$ time.

First you need to formally argue about the correctness of your algorithm (i.e., prove that it computes a local maximum). Second, prove that the runtime of your algorithm is $\mathcal{O}(\log n)$.

c) **(10 points)** Now we want to extend the problem from a one-dimensional array to a two-dimensional array, i.e., an $n \times n$ matrix with positive (integer) entries. Two entries $A[i, j]$ and $A[i', j']$ are neighboring values if $|i - i'| + |j - j'| = 1$.

An entry is a *local maximum* of the matrix if it is not smaller than all its neighboring values (note that an element at the side of the matrix has two or three neighboring values and any other element has four neighboring values). In Figure 1, a is a local maximum if $a \geq b, a \geq c, a \geq d$, and $a \geq e$.

Devise a divide and conquer algorithm that finds a local maximum in a twodimensional $n \times n$ matrix in time $\mathcal{O}(n \log n)$. Again formally prove the correctness of your algorithm and its runtime.

Solution

a) We first give a very simple solution to a). Afterwards we provide a little more general result which we will need for part b).

Simple solution. The array contains a finite number of natural numbers. Thus there is one element which is at least as large as all other elements. This element will for sure be a local maximum regardless of its position in the array.

More general Solution. We first show that any array in which the side elements are smaller than their neighbors, i.e., $A[1] < A[2]$ and $A[n] < A[n-1]$, contains a local maximum. If an array satisfies this condition we say it satisfies *(condition 1)*.

Now, assume that we are given an array satisfying *(condition 1)*. Beginning from the left side of the array one notices that $A[1]$ is not a local maximum, because $A[1] < A[2]$. Now, $A[2]$ is a local maximum if and only if $A[2] \geq A[3]$ holds. If $A[2]$ is not a local maximum $A[2] < A[3]$ holds and $A[3]$ is a local maximum if and only if $A[3] \geq A[4]$ holds. We continue like this until we either we find a local maximum or none of the elements at positions $1, \ldots, n-1$ is a local maximum. But then we obtained that $A[n-1] < A[n]$ holds, which is a contradiction to $A[n] < A[n-1]$.

Now, to prove the claim we can simply enlarge the given array A by two entries. We append a $zero$ before the array and one at the end of the array. Note that this modified array contains a local maximum if and only if the original array contains a local maximum. Furthermore the modified array fulfills *(condition 1)*.

b) As usually for divide and conquer algorithms we use a recursive algorithm. At the beginning, before starting the recursive part, the algorithm checks whether the two side elements of the given array are local maxima. If one of them is a local maximum the algorithm returns it. Otherwise it continues with the recursive part.

For the recursion the algorithm checks whether the element in the middle position is a local maximum. If it is a local maximum the algorithm returns it. Otherwise at least one of its neighboring elements is larger, which we call l. In that case the algorithm continues its search in the half of the array which begins/ends at the middle element and contains l. Note that this array satisfies *(condition 1)*.

Correctness: If the algorithm does not find a local maximum in the current step it continues its search in an array which satisfies *(condition 1)*, i.e., in an array which contains a local maximum after part $a)$. If the size of the array is 3 (and *(condition 1)* is satisfied) the middle element is always a local maximum. Thus the algorithm always has to either find a local maximum before or as soon as the array only has three elements.

Runtime: The recurrence relation of the algorithm is

$$T(n) = T(\frac{n}{2}) + \mathcal{O}(1),\ T(1) = \mathcal{O}(1),$$

which yields $T(n) = \mathcal{O}(\log n)$ via the master theorem.

c) We extend the results from $a)$ and $b)$ to matrices.

Existence of a local maximum: We say that a matrix satisfies *(condition 2)* if the maximum of each side column is not a local maximum.

We first want to show that any such matrix contains a local maximum. If one of the elements in the side columns is a local maximum, we are done, so we can assume that the matrix satisfies *(condition 2)*.

By induction we show that any matrix with positive numbers which satisfies *(condition 2)* contains a local maximum. For fixed n consider an $m \times n$ matrix and perform an induction on m.

Induction base: For $m = 1$ the matrix is simply an array and the result holds by $a)$.

Induction hypotheses: Any $m \times n$ matrix which satisfies *(condition 2)* contains a local maximum.

Induction step $\boldsymbol{n \to n+1}$*:* Let A be a $(m+1) \times n$ matrix satisfying *(condition 2)*.

Case 1: The maximum of row $A[m]$ is a local maximum. We are done.

Case 2: The maximum of row $A[m]$ is not a local maximum. Then we can apply the induction hypothesis with $A[1..m][1..n]$

The claim holds for all m by the principle of induction.

One can perform the same proof by an induction on rows and one can even perform an interleaved induction.

Algorithm: Again we use a recursive algorithm. In each step it splits the currently considered matrix into two parts and only recurses on one of them. In even 'rounds' of the recursion the matrix is split at the middle column and in odd 'rounds' it is split in the middle row.

Before starting the recursion we check whether the maxima of the side columns (rows) are local maxima. If one of them is a local maxima we return it, otherwise we continue with the recursive part.

Applied to a matrix the algorithm searches for the maximum m of the middle column (row) in time $\mathcal{O}(n)$. If that maximum is a local maximum it returns it, otherwise there is an element l neighboring m which is larger. The algorithm recursively continues the search in the half of the matrix which contains l and m. If the matrix is a single element the algorithm will return this element.

Correctness: The algorithm is correct because it either finds a local maximum in the current step or continues its search in a matrix which satisfies *(condition 2)* and thus is guaranteed to contain a local maximum.

Runtime: The runtime is dominated by the recursive part. For an $n \times n$ matrix it is $\mathcal{O}(n \log n)$ as the recurrence relation ($m = n$) of the algorithm is

$$T(n) = T(\frac{n}{2}) + \mathcal{O}(n),\ T(1) \leq \mathcal{O}(\log n).$$

The master theorem immediately yields that the runtime is $\mathcal{O}(n \log n)$.