Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
Johannes Lengler, Gleb Novikov, Chris Wendler

07. October 2019

Algorithms & Data Structures — Exercise sheet 3 — HS 19

Exercise Class (Room & TA): ______________________

Submitted by: ______________________

Peer Feedback by: ______________________

Points: ____

The solutions for this sheet are submitted at the beginning of the exercise class on October 14th.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

Exercise 3.1 *Counting Operations in Loops* **(2 Points)**.

For the following code fragments count how many times the function f is called. Report the number of calls as nested sum, and then simplify your expression in $\mathcal{O}$-notation (as tight and simplified as possible) and prove your result. For example, in the code fragment

Algorithm 1

for $k = 1, \dots, 100$ **do**
    $f()$

the function f is called $\sum_{k=1}^{100} 1 = 100$ times, so the amount of calls is in $\mathcal{O}(1)$.

a) Consider the snippet:

Algorithm 2

for $j = 1, \dots, n$ **do**
    for $k = j, \dots, n$ **do**
        $f()$

b) Consider the snippet:

Algorithm 3

for $j = 1, \dots, n$ **do**
    for $k = j, \dots, n$ **do**
        for $l = 1, \dots, 100$ **do**
            $f()$
            $f()$

c) Consider the snippet:

Algorithm 4

```
for k = 1, ..., 100 do
    f()
for j = 1, ..., n do
    f()
    for k = 1, ..., j do
        for l = 1, ..., j do
            for m = 1, ..., j do
                f()
```

d) Consider the snippet:

Algorithm 5

```
for j = 1, ..., n do
    for k = 1, ..., j do
        l ← 1
        while l ≤ j do
            f()
            l ← 2l
```

*e) Consider the snippet:

Algorithm 6

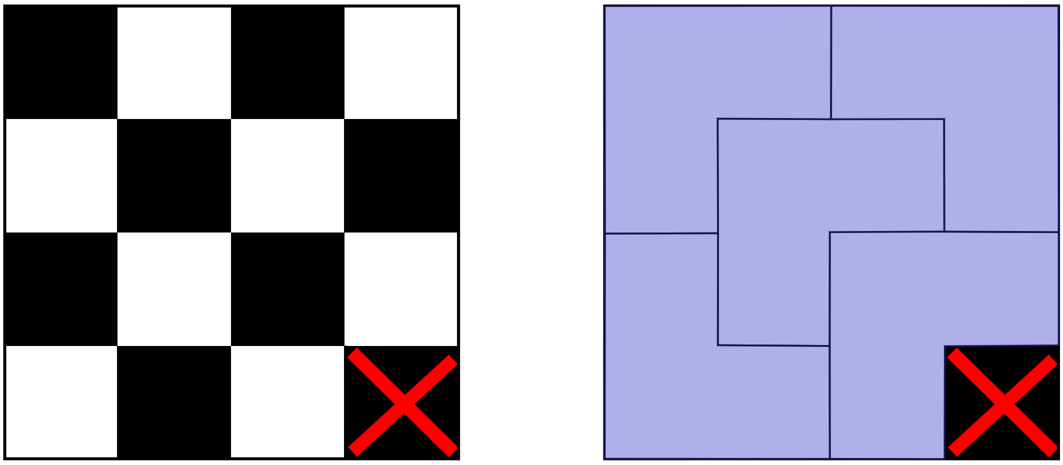
```
for j = 1, ..., n do
    for k = 1, ..., j do
        for ℓ = 1, ..., k do
            for m = ℓ, ..., n do
                f()
```

Exercise 3.2 *Divide and Conquer* **(1 Point)**.

a) List at least two algorithms from your solutions or the sample solutions of sheet 1 and sheet 2 that are divide-and-conquer algorithms.

b) Consider the following problem:

You are given a $2^k \times 2^k$ chessboard with one missing square and as many L-shaped puzzle pieces as you want. Each puzzle-piece can cover exactly three squares of the chessboard. As you will show algorithmically in this exercise, it is always possible to cover such chessboards by L-shaped puzzle pieces. An example is given in Figure 1 for $k = 2$, where the missing piece is a corner piece.



(a) 4×4 chessboard (b) covering

Figure 1: Example of a chessboard and its covering by L-shaped puzzle pieces.

1) Devise a divide-and-conquer algorithm that can cover a $2^k \times 2^k$ chessboard with one missing square at an arbitrary position for $k \in \{1, 2, 3, \dots\}$. Describe your algorithm using words. Make sure to describe how you divide the problem into *subproblems* and how you handle the *base case(s)*. Your description should be *concise* (e.g., it could have a pseudo-code-like form for readability).

   You can assume that each square is represented by its coordinates, specifically, the square in the lower left corner has coordinates $(1, 1)$ and the square in the upper right corner has coordinates $(2^k, 2^k)$. The input of your algorithm is (k, a, b), where a and b are coordinates of the missing square.

2) Determine the running time of your algorithm in terms of $n = 2^k$ in $\mathcal{O}$-notation.

Exercise 3.3* *Maximum-Submatrix-Sum.*

Provide an $\mathcal{O}(n^3)$ time algorithm which given a matrix $M \in \mathbb{Z}^{n \times n}$ outputs its maximal submatrix sum S. That is, if M has some non-negative entries,

$$S = \max_{\substack{1 \le a \le b \le n \\ 1 \le c \le d \le n}} \sum_{i=a}^{b} \sum_{j=c}^{d} M_{ij},$$

and if all entries of M are negative, $S = 0$.

Justify your answer, i.e. prove that the asymptotic runtime of your algorithm is $\mathcal{O}(n^3)$.

Hint: *You may want to start by considering the cumulative column sums*

$$C_{ij} = \sum_{k=1}^{i} M_{kj}.$$

How can you compute all C_{ij} efficiently? After you have computed C_{ij}, how you can use this to find S?