# Algorithmen & Datenstrukturen

Woche 8

Marius Tomek, Nicolas Wehrli, Tim Rieder

14. November 2022

ETH Zürich

# Inhalt

Kurze Kommentare zur letzten Serie

Datenstrukturen

Homework 07

Peergrading 7.3

# Kurze Kommentare zur letzten Serie

## Serie 06

Häufige Fehler:

- 6.2.c: base case ist $n = 0$ not $n = 1$
  vollständige Induktion, normale Induktion genügt nicht
- 6.4.b: Solche Induktionsaufgaben über 2 Variablen genauer anschauen, nicht so intuitiv wie welche über 1 Variable,
  wenn man unten rechts ankommt terminiert die Rekursion nicht, sondern die Funktion macht einfach keine weiteren rekursiven Aufrufe
- 6.4.c: Bei $A[x][y] =' #'$ nicht $-1$ verwenden und weiterrechnen, sondern entweder $-\infty$ und weiterrechnen oder 0 und sofort zurückgeben
- 6.4.d: gegebenes Schema verwenden für bessere Struktur, sonst vergisst man schnell was

# Datenstrukturen

# Datenstrukturen

Wir betrachten folgende Abstrakte Datentypen (ADT's).

- Stack
- Queue
- Priority Queue
- Dictionary

Die Implementation davon sind Datenstrukturen.

# Stack – LIFO

- push(x, S):
  legt x auf den Stack
- pop(x, S):
  entfernt das oberste Element vom Stack und gibt es zurück
- top(x, S):
  gibt das oberste Element zurück, ohne dies zu entfernen
- isEmpty(S), ...

Wir können diesen ADT zum Beispiel mit einer Linked-List implementieren.
(Array würde auch gehen, aber wir müssen die maximale Grösse kennen)

## Queue – FIFO

- enqueue(x, S):
  x wird hinten angefügt
- dequeue(S):
  entfernt das vorderste Element und gibt es zurück

Implementation mit doubly-linked-list + pointer auf das letzte Element

# Priority Queue

- insert(x, P):
  fügt x in P ein
- extractMax(P):
  entfernt das Maxmimum und gibt es zurück

Dies kann mit einem Heap implementiert werden.

# Dictionary

- search(x, T)
  Gibt zurück ob x in T ist
- insert(x, T)
  fügt x in T ein
- remove(x, T)
  entfernt x von T

Dies kann mit einem Heap, Arrays, Linked-Lists, etc. implementiert werden.
Wir sind aber an einer besonderen Implementation interessiert: Suchbäume

## Binary Search Tree (BST)

Baum mit der Bedingung, dass für alle Knoten $x$ im Baum folgendes gilt:
Für alle Knoten $a$ im linken Teilbaum von $x$ und alle Knoten $b$ im rechten Teilbaum von $x$ gilt:

$$a < x < b$$

(Wir benutzen eine vereinfachende Annahme, dass alle Zahlen unterschiedlich sind)

- Suchen wie Binary search
- Einfügen von $c$ mit Suchen nach $c$, dann das Blatt durch $c$ ersetzen.
- Entfernen von $d$: Suchen nach $d$, im generellen Fall mit dem symmetrischen Nachfolger tauschen und dann löschen, in speziellen Fällen direkt löschen und evtl. Teilbaum umhängen

Problem der möglichen Unbalanciertheit $\Longrightarrow$ AVL-Bäume

# AVL-Baum

AVL-Bedingung:
Für alle Knoten $x$ im Baum gilt:

$$|h_l - h_r| \leq 1$$

wobei $h_l$ die Höhe des linken und $h_r$ die Höhe des rechten Teilbaums ist.

Jetzt müssen wir aber bei jedem Einfügen oder Entfernen, evtl. die AVL-Bedingung wiederherstellen.

Dabei müssen wir von der manipulierten Stelle aus alle Vorgänger betrachten.

AVL-Baum Visualisierung:
https://www.cs.usfca.edu/ galles/visualization/AVLtree.html

# Homework 07

**Exercise 7.1** *$k$-sums* **(1 point)**.

We say that an integer $n \in \mathbb{N}$ is a *$k$-sum* if it can be written as a sum $n = a_1^k + \cdots + a_p^k$ where $a_1, \ldots, a_p$ are distinct natural numbers, for some arbitrary $p \in \mathbb{N}$.

For example, 36 is a 3-*sum*, since it can be written as $36 = 1^3 + 2^3 + 3^3$.

Describe a DP algorithm that, given two integers $n$ and $k$, returns True if and only if $n$ is a $k$-sum. Your algorithm should have asymptotic runtime complexity at most $O(n^{1+\frac{2}{k}})$.

***Hint:*** *The intended solution has complexity* $O(n^{1+\frac{1}{k}})$.

# Exercise 7.1

In your solution, address the following aspects:

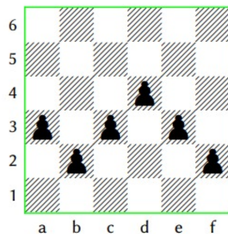
1. *Dimensions of the DP table*: What are the dimensions of the $DP$ table?
2. *Definition of the DP table*: What is the meaning of each entry?
3. *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

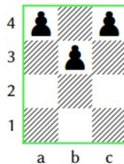
# Exercise 7.3

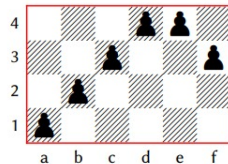
**Exercise 7.3** *Safe pawn lines* **(1 point).**

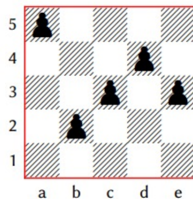
On an $N \times M$ chessboard ($N$ being the number of rows and $M$ the number of columns), a *safe pawn line* is a set of $M$ pawns with exactly one pawn per column of the chessboard, and such that every two pawns from adjacent columns are located diagonally to each other. When a pawn line is not safe, it is called *unsafe*.

The first two chessboards below show safe pawn lines, the latter two unsafe ones. The line on the third chessboard is unsafe because pawns d4 and e4 are located on the same row (rather than diagonally); the line on the fourth chessboard is unsafe because pawn a5 has no diagonal neighbor at all.









# Exercise 7.4

**Exercise 7.4** *String Counting* **(1 point).**

Given a binary string $S \in \{0,1\}^n$ of length $n$, let $f(S)$ be the length of the longest substring of consecutive 1s. For example $f(\text{"}01100011010\underline{111}0001\text{"}) = 3$ because the string contains "111" (underlined) but not "1111". Given $n$ and $k$, the goal is to count the number of binary strings $S$ of length $n$ where $f(S) = k$.

Write the **pseudocode** of an algorithm that, given positive integers $n$ and $k$ where $k \leq n$, reports the required answer. For full points, the running time of your solution can be any polynomial in $n$ and $k$ (e.g., even $O(n^{11}k^{20})$ is acceptable).

***Hint:*** *The intended solution has complexity* $O(nk^2)$.

In your solution, address the same six aspects as in Exercise 7.1.

# Exercise 7.4

**Algorithm 1**

```
1:  Input: integers n, k.
2:  Define dp[1...n][0...k][0...k], initialized to 0.
3:  dp[1][0][0] ← 1
4:  dp[1][1][1] ← 1
5:  for len ∈ {1, ..., n − 1} do
6:      for maks ∈ {0, ..., k} do
7:          for curr ∈ {0, ..., k} do
8:              val ← dp[len][maks][curr]
9:              if val ≠ 0 then                                  ▷ Prevents going out-of-bounds.
10:                 (Note: let a +← b be the shorthand for a ← a + b.)
11:                 dp[len + 1][maks][0] +← val                              ▷ Append 0.
12:                 dp[len + 1][max(maks, curr + 1)][curr + 1] +← val       ▷ Append 1.
13: sol ← 0
14: for curr ∈ {0, 1, ..., k} do
15:     sol ← sol + dp[n][k][curr]
16: Print("solution = ", sol)
```

# Peergrading 7.3