technische universität dortmund

Fakultät für Informatik
Lehrstuhl 11

Mart Hagedoorn
Guangping Li

Sommersemester 2024
26. April

# DAP2 Praktikum – Blatt 3

Abgabe: KW 18

## Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer*innen die folgenden Leistungen erbringen:
  - Es müssen mindestens 50 Prozent der Punkte in den Kurzaufgaben erreicht werden.
  - Es müssen mindestens 50 Prozent der Punkte in den Langaufgaben erreicht werden.
- Im Krankheitsfall kann ein Testat bei Vorlage eines Attests in der folgenden Woche nachgeholt werden.
- Wenn ein Praktikumstermin auf einen Feiertag fällt, müssen Sie sich an einem beliebigen anderen Praktikumstermin in der gleichen Woche testieren lassen.
- **Hinweis:** Notieren Sie sich Ihre Punkte nach jedem Testat! Dies dient der eigenen Kontrolle. (Ihr Punktestand kann Ihnen während des Semesters nicht genannt werden.)

## Wichtige Information (im Moodle verfügbar)

- Beachten Sie die Erklärung des **Ablaufs (Blatt A)**.
- Beachten Sie die **Regeln und Hinweise (Blatt R)** in der aktuellsten Version!
- Beachten Sie die **Hilfestellungen (Blatt H)** in der aktuellsten Version!

## Kurzaufgabe 3.1: Enumerierung von Teilmengen (5 Punkte)

Schreiben Sie ein Programm, das eine Ganzzahl $n$ und eine Liste von Ganzzahlen $a_0, a_1, \ldots, a_{n-1}$ via Standard-In (Nach jedem Wert bestätigen Sie mit der Enter-Taste), sowie eine Ganzzahl $k$ als Argument erhält, und alle $k$-Elementigen Teilmengen ausgibt. Dies soll in folgenden Schritten umgesetzt werden:

- Zuerst werden die Zahlen $a_0, a_1, \ldots, a_{n-1}$ in ein Array `data` eingelesen.
- Anschließend werden Duplikate (also doppelte Einträge) aus dem Array entfernt. Implementieren sie eine Methode

  `public static int removeDuplicates(int [] data)`,

  welche die Anzahl $n' \leq n$ der unterschiedlichen Werte in `data` zurückgibt. Zusätzlich soll die Methode die Werte im Array umordnen, sodass die $n'$ unterschiedlichen Werte genau im Prefix `data`$[0, n'-1]$ des Arrays stehen. Also es ist nicht wichtig, was in dem Suffix `data`$[n', n-1]$ steht.

  **Hinweis:** Sie dürfen für diesen Schritt die Methode `Arrays.sort(...)` verwenden. Für die volle Punktzahl dürfen Sie keine Hilfsarrays verwenden, und abgesehen von dem Aufruf von `Arrays.sort(...)` nur $\mathcal{O}(n)$ Zeit benötigen!
- Wenn $n' < k$, dann setzen Sie $k$ auf $n'$. Geben Sie alle $k$-elementigen Teilmengen der $n'$ verschiedenen Elemente aus. Abschließend geben Sie außerdem an, wie viele Teilmengen dies sind (berechnen Sie dies, indem Sie zählen, wie oft Sie eine Teilmenge drucken). Für die volle Punktzahl sollten Sie dafür (abgsehen von der Ausgabe) nicht mehr als $\mathcal{O}(\binom{n'}{k})$ Rechenschritte benötigen (dabei ist $\binom{n'}{k}$ der Binomialkoeffizient). Das gelingt am leichtesten mit einer rekursiven Implementierung.

  **Hinweis:** Für die Struktur der Rekursion können Sie sich von Pascal's Dreieck inspirieren lassen.

Wie immer können Sie davon ausgehen, dass die Eingabe korrekt ist und müssen der Code nicht um diese Eingabe Robust gestalten. Beispielausgaben des Programms:

```
java B3A1.java 2 <Enter>
5 2 2 4 6 8 <Enter>
Before removing duplicates: [2, 2, 4, 6, 8]
After removing duplicates:  [2, 4, 6, 8]
[2, 4]
[2, 6]
[2, 8]
[4, 6]
[4, 8]
[6, 8]
Number of subsets: 6
```

```
java B3A1.java 6 <Enter>
5 2 2 4 6 8 <Enter>
Before removing duplicates: [2, 2, 4, 6, 8]
After removing duplicates:  [2, 4, 6, 8]
[2, 4, 6, 8]
Number of subsets: 1
```

## Kurzaufgabe 3.2: Mehr Permutationen & Lex. Ordnung (3 Punkte)

Für dieses Problem erhalten sie eine Ganzzahl $n$ und eine Liste von Ganzzahlen $a_0, a_1, \ldots, a_{n-1}$ via Standard-In, sowie eine positive Ganzzahl $k$ als Argument. Dieses mal dürfen Sie einfach davon ausgehen, dass die Liste der Ganzzahlen keine Duplikate enthält.

Seien $p_0, p_1, \ldots, p_{n-1}$ und $q_0, q_1, \ldots, q_{n-1}$ unterschiedliche Permutationen von $a_0, a_1, \ldots, a_{n-1}$. Wir sagen, dass $p_0, p_1, \ldots, p_{n-1}$ lexikographisch kleiner als $q_0, q_1, \ldots, q_{n-1}$ ist, genau dann wenn es einen index $i \in \{0, \ldots, n-1\}$ gibt, sodass $\forall j \in \{0, \ldots, i-1\} : p_j = q_j$ und $p_i < q_i$. Beispiele:

- $1, 4, 3, 7, 5$ ist lex. kleiner als $4, 1, 3, 7, 5$
- $3, 4, 1, 5, 7$ ist lex. kleiner als $3, 4, 1, 7, 5$

Schreiben Sie ein Programm, das die lexikographisch $k$-kleinste Permutation von $a_0, a_1, \ldots, a_{n-1}$ ausgibt. Sie dürfen die Methode `Arrays.sort(..)` verwenden, um die Liste der Zahlen einmalig zu sortieren. Ihr Programm sollte nur $\mathcal{O}(n^2)$ Rechenschritte benötigen. Beispiele:

```
java B3A2.java 17 <Enter>
4 4 3 1 2 <Enter>
Sorted input:
[1, 2, 3, 4]
The 17-smallest permutation is:
[3, 4, 1, 2]
```

**Tipp:** Für jedes $i \in \{0, n-1\}$ gibt es $(n-1)!$ Permutationen von $a_0, a_1, \ldots, a_{n-1}$, die mit $a_i$ anfangen. Wenn die Elemente aufsteigend sortiert sind, also $a_0 < a_1 < \cdots < a_{n-1}$, dann gilt:

- Die $(1 + 0 \cdot (n-1)!)$-kleinste Permutation (also die kleinste Permutation) beginnt mit $a_0$.
- Die $(1 + 1 \cdot (n-1)!)$-kleinste Permutation beginnt mit $a_1$.
- Die $(1 + 2 \cdot (n-1)!)$-kleinste Permutation beginnt mit $a_2$.
- usw...

Allgemein ausgedrück ist die $(1+i \cdot (n-1)!)$ kleinste Permutation genau die kleinste Permutation die mit $a_i$ anfängt. Um die $k$-kleinste Permutation von $a_0, a_1, \ldots, a_{n-1}$ zu finden, können sie eine einfache rekursive Regel anwenden:

- $\mathit{find}(a_0, 1) = a_0$, (Basisfall für eine 1-elementige Liste) und rekursiv
- $\mathit{find}(a_0, a_1, \ldots, a_{n-1}, k) = a_i, \mathit{find}(a_0, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1}, k')$,
  wobei $i = \left\lfloor \frac{k-1}{(n-1)!} \right\rfloor$ und $k' = k - i \cdot ((n-1)!)$.