Informatik II (D-ITET)
Frühjahrssemester 2016
Prof. Friedemann Mattern
Dr. Wilhelm Kleiminger
Hossein Shafagh (shafagh@inf.ethz.ch)

ETH Zürich
Institut für Pervasive Computing
Gruppe für Verteilte Systeme
http://vs.inf.ethz.ch

Übungsserie Nr. 8

Ausgabe: 27. April 2016
Abgabe: 4. Mai 2016

Hinweise

Für diese Serie benötigen Sie die folgenden Archive:
http://vs.inf.ethz.ch/edu/FS2016/I2/downloads/u8.zip
http://vs.inf.ethz.ch/edu/FS2016/I2/downloads/reversi.jar

1. Aufgabe: (11 Punkte) Binäre Suche

(1a) *(1 Punkt)* Gegeben sei das Array [3, 7, 17, 25, 33, 47, 56, 62, 65, 66, 68, 70, 78, 89, 92]. Zeichnen Sie den Entscheidungsbaum, den eine binäre Suche nach der Zahl 47 durchläuft.

Hinweis: Der Entscheidungsbaum für die Suche nach der Zahl 3 im Array [1, 2, 3] hat zwei Knoten: Die Zahl 2 als Wurzel und die Zahl 3 als rechtes Kind.

(1b) *(2 Punkte)* Legen Sie nun alle Entscheidungsbäume für die binäre Suche nach jeweils allen Zahlen übereinander. Wie sieht das Ergebnis aus? Was für eine Datenstruktur ist das?

(1c) *(2 Punkte)* Nehmen Sie nun an, die binäre Suche würde das Teilarray nicht halbieren, sondern es so in zwei Teile teilen, dass jeweils im ersten Teil das erste Drittel der Zahlen und im zweiten Teil die verbleibenden zwei Drittel der Zahlen stehen. Wie sieht nun die Überlagerung aller Enscheidungsbäume aus? Ist diese Strategie schlechter oder besser als Halbieren? Warum?

(1d) *(3 Punkte)* Schreiben Sie eine rekursive Implementierung der Schnittstelle *IBinarySearch* und implementieren Sie die Fabrikmethode *BinarySearchFactory.create*.

Hinweis: Der generische Typ *Key* implementiert die Schnittstelle *Comparable*⟨*Key*⟩. Das wird durch die Generic-Deklaration festgelegt und vom Compiler überprüft. Dadurch weiss die Implementierung, dass Objekte von diesem Typ mittels *compareTo* miteinander verglichen werden können.

(1e) *(3 Punkte)* Ihre Implementierung aus Aufgabe *d)* soll nun zusätzlich die Schnittstelle *IMeasure* implementieren. Schreiben Sie anschliessend eine *main*-Klasse, die mit Hilfe der *IMeasure*-Schnittstelle die folgenden Fragen beantwortet:

Gegeben das Array aus *a)*, wie gross ist jeweils die mittlere Anzahl von rekursiven Aufrufen für die Strategien aus *b)* und *c)*, wenn

1. man nach allen vorhandenen Zahlen sucht?
2. man nach allen Zahlen von 0 bis 99 sucht?
3. man nach allen Zahlen von 0 bis 9 sucht?

Wie würden Sie nun auf die Frage nach der besten Strategie antworten?

2. Aufgabe: (12 Punkte) Rucksackproblem und Backtracking

Ein Dieb ist in ein Haus eingebrochen und steht nun vor dem Problem, unter den K vorhandenen Gegenständen $x_1, \ldots, x_K$ eine Auswahl treffen zu müssen, so dass das zulässige Gesamtgewicht G seiner Tasche nicht überschritten wird. Gleichzeitig soll der Wert aller eingepackten Objekte möglichst gross sein. Der Dieb kennt von jedem Gegenstand dessen Wert $w_i \geq 0$ sowie dessen Gewicht $g_i \geq 0$. Formal ist also eine Selektion $b_1, \ldots, b_K$ mit $b_i \in 0, 1$ gesucht, für die gilt:

$$\sum_{i=1}^{K} b_i w_i \text{ ist maximal, und } \sum_{i=1}^{K} b_i g_i \leq G$$

(2a) *(2 Punkte)* Gibt es immer genau eine optimale Lösung? Beweisen Sie Ihre Antwort.

(2b) *(4 Punkte)* Implementieren Sie die Schnittstelle *IRucksack* mit einem Algorithmus, der alle Möglichkeiten ausprobiert und die beste davon zurück liefert. Passen Sie zum Testen die Fabrikmethode *RucksackFactor.create* an.

(2c) *(4 Punkte)* Implementieren Sie die Schnittstelle *IRucksack* mit einem Backtracking-Algorithmus. Passen Sie zum Testen die Fabrikmethode erneut an.

(2d) *(2 Punkte)* Der Test *complex* misst die Zeit, die Ihr Algorithmus braucht, um eine Lösung zu finden. Vergleichen Sie die Werte für Ihre Algorithmen aus Aufgabe *b)* und *c)*. Was fällt Ihnen auf? Wie erklären Sie sich das? Findet der Backtracking-Algorithmus immer eine optimale Lösung?

3. Aufgabe: (10 Punkte) Reversi [Teil 2]

(3a) *(*5 Punkte) Implementieren Sie die Schnittstelle *ICheckMove* anhand der vorhanden Dokumentation, ohne dabei *reversi.GameBoard.checkMove* oder *reversi.GameBoard.isMoveAvailable* zu verwenden. Die zugehörige Fabrikmethode heisst *CheckMoveFactory.create*.

(3b) *(5 Punkte)* Implementieren Sie einen Reversi-Spieler, der unter allen möglichen Zügen den Besten wählt. Dabei soll nur der nächste Zug berücksichtig werden (d.h., es soll in dieser Übung noch kein Spielbaum aufgebaut werden). Schreiben Sie eine Bewertungsfunktion, die einen Zug danach bewertet, wie viele Steine der Spieler nach dem Zug mehr hat als der Gegner.

Hinweis: Um hypothetische Züge machen zu können, müssen Sie die aktuelle Spielsituation kopieren.

Hinweis: Die Schnittstelle von *reversi.GameBoard* schreibt vor, dass man vor einem *makeMove* ein *checkMove* auf *demselben* Objekt aufrufen muss.