Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Ulf Leser
M. Bux, B. Grußien, L. Heimberg, P. Schäfer

Berlin, den 05.06.2017

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 4

Abgabe: Montag den 19.06.2017 bis 11:10 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten neben Raum 3.321, RUD25.

Die Übungsblätter sind in Gruppen von zwei (in Ausnahmen drei) Personen zu bearbeiten. Jedes Übungsblatt muss bearbeitet werden. (Sie müssen mindestens ein Blatt für wenigstens eine Aufgabe jedes Übungsblattes abgeben.) Die Lösungen sind auf nach Aufgaben getrennten Blättern abzugeben. Heften Sie bitte die zu einer Aufgabe gehörenden Blätter vor der Abgabe zusammen. Vermerken Sie auf allen Abgaben die Namen und **CMS-Benutzernamen** aller Gruppenmitglieder, Ihre Abgabegruppe (z.B. AG123) aus Moodle, und den Übungstermin (z.B. Di 13 Uhr bei Marc Bux), zu dem Sie Ihre korrigierten Blätter zurückerhalten werden.

Beachten Sie die Informationen auf der Übungswebseite (https://hu.berlin/algodat17).

Konventionen:

- Soweit nicht anders angegeben, beginnt die Indizierung aller Arrays auf diesem Blatt bei 1. Bitte beginnen Sie die Indizierung der Arrays in Ihren Lösungen auch bei 1.
- Mit log wird der Logarithmus $\log_2$ zur Basis 2 bezeichnet.

Aufgabe 1 (Fibonacci-Suche) — 5 + 6 = 11 Punkte

1. Führen Sie einen Schreibtischtest für den Algorithmus *Fibonacci-Search* aus der VL durch, bei dem das folgende Array A nach dem Wert $c = 17$ durchsucht wird. Geben Sie die aktuellen Belegungen der Variablen fib2, fib3, und m vor jedem Aufruf von Zeile 8 im Pseudocode von Folie 13 an.

$A =$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

2. Die Fibonacci-Zahlen sind wie folgt definiert: $F_1 = 1$, $F_2 = 1$ und $F_k = F_{k-2} + F_{k-1}$ für alle $k > 2$. Zeigen Sie unter Verwendung dieser Definition für Fibonacci-Zahlen, dass für $k \geq 1$ die Beziehung $F_k \geq \frac{1}{3} F_{k+2}$ gilt.

Aufgabe 2 (Suche in sortierten Arrays) **4 + 5 + 7 = 16 Punkte**

Implementieren Sie die nachfolgenden drei Suchverfahren für die Suche eines Werts k in einem aufsteigend sortierten Array A mit Werten vom Typ `Long`:

(a) *binäre Suche*,

(b) *Interpolations-Suche*,

(c) *exponentielle Suche*.

Implementieren Sie die *binäre Suche* und die *Interpolations-Suche*, wie sie in der Vorlesung vorgestellt wurden. Implementieren Sie anschließend das Suchverfahren der *exponentiellen Suche*. Bei der *exponentiellen Suche* handelt es sich um ein Suchverfahren für sortierte Arrays, das aus zwei Phasen besteht:

- In der ersten Phase wird der Suchbereich ermittelt, in dem das gesuchte Element liegen muss. Dies geschieht, indem der gesuchte Wert k nacheinander mit den Werten $A[0]$, $A[2^0]$, $A[2^1]$, $A[2^2]$, usw. verglichen wird. D.h., es wird das kleinste $i \geq 0$ bestimmt, sodass $A[2^i] \leq k < A[\min\{2^{i+1}, n\}]$ ist. Da die Indexierung von Arrays in Java bei 0 beginnt, muss außerdem zusätzlich der Suchbereich $A[0] \leq k < A[2^0]$ betrachtet werden.
- Anschließend wird in dem ermittelten Suchbereich eine *binäre Suche* nach dem Wert k durchgeführt.

Die *exponentielle Suche* wird typischerweise eingesetzt, wenn die Länge des sortierten Arrays zunächst unbekannt oder sehr groß gegenüber dem vermuteten Index von k im Array ist.
Ergänzen Sie den fehlenden Code in der Vorlage `SortedSearch.java`, welche Sie auf der Übungswebseite[1] vorfinden. Sie können beliebige neue Variablen und Hilfsmethoden zur Klasse hinzufügen, dürfen jedoch keine außer den von Java bereitgestellten Standard-Bibliotheken verwenden. Verwenden Sie für alle Vergleiche die vorgegebene Klasse `CountingComparator`, welche die Anzahl der Vergleiche zählt. Ein Beispiel für die Implementierung einer Suche in dem vorgegebenen Quellcode-Gerüst finden Sie in der Klasse `LinearSearch`.
Stellen Sie sicher, dass alle Testfälle in der main-Methode der Datei SortedSearch.java erfolgreich durchlaufen. Achten Sie außerdem auf Randbedingungen und Spezialfälle, die von den Testfällen vielleicht nicht vollständig abgedeckt werden.
Hinweis zur Abgabe: Ihr Java-Programm muss auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe des von Ihnen modifizierten Quellcodes `SortedSearch.java` erfolgt über Moodle.

[1] https://hu.berlin/algodat17

Aufgabe 3 (Algorithmenentwurf) **9 + 2 = 11 Punkte**

Zu zwei disjunkten Gruppen A und B von jeweils n Personen sei das Alter (in Jahren) der Personen jeweils in aufsteigender Reihenfolge in den Arrays L_A und L_B gegeben. Nun vereinigen sich beide Gruppen zu einer großen Gruppe C. Gesucht ist das Alter einer Person aus C, die mindestens so alt ist wie $n-1$ andere Personen aus C und höchstens so alt ist wie $n-1$ andere Personen aus C.

1. Entwerfen Sie einen möglichst effizienten Algorithmus in Pseudocode, der als Eingabe die sortierten Arrays L_A und L_B hat, und als Ausgabe ein Alter ausgibt, das den oben genannten Vorgaben entspricht.

   Hinweis: Für einen korrekten sublinearen Algorithmus gibt es 9 Punkte, ein linearer Algorithmus gibt 6 Punkte, eine superlineare Variante 3 Punkte.

2. Analysieren Sie die Laufzeit Ihres in Pseudocode vorgestellten Algorithmus.

Aufgabe 4 (Amortisierte Analyse) **8 + 4 = 12 Punkte**

In der Vorlesung wurde Ihnen die amortisierte Analyse am Beispiel eines Binärzähler mit beschränkter Länge vorgestellt.

1. Ein *Ternärzähler* ist ein Zähler, der als Basis für die Zahlendarstellung die Zahl 3 benutzt. Der Zählerstand $t_k t_{k-1} \dots t_0$ liegt also in Ternärdarstellung vor und kodiert die Dezimalzahl $\sum_{i=0}^{k} 3^i t_i$. Eine einzelne Stelle einer Zahl in Ternärdarstellung bezeichnet man als *Trit*. Wir betrachten einen Ternärzähler unbeschränkter Länge. Der Ternärzähler hat genau eine Operation *increment*, die den Zählerstand um 1 erhöht. Die Kosten einer *increment*-Operation seien dabei die Anzahl der dafür benötigter Tritänderungen.

   Zeigen Sie mittels amortisierter Analyse (Potentialmethode oder Account-Methode), dass die Worst-Case-Komplexität der amortisierten Tritwechselkosten eines Ternärzählers unbeschränkter Länge, der n mal um 1 inkrementiert wird, $\mathcal{O}(n)$ beträgt. Dabei ist der Zähler anfangs mit 0 initialisiert.

2. Gegeben sei ein Binärzähler unbeschränkter Länge, der mit 0 initialisiert wird. In der Vorlesung haben wir die amortisierten Bitwechselkosten für die n-malige Addition der Zahl 1 bestimmt. In dieser Aufgabe addieren wir n-mal zum Zähler die Zahl 3. Zeigen Sie, dass dabei amortisierte Bitwechselkosten von höchstens 4 pro Addition anfallen.