Aufgabenblatt
Eigenschaften von Algorithmen

- Abgabetermin: **Samstag, 04.06.2016 23:55 Uhr**
- Zur Prüfungszulassung müssen in einem Aufgabenblatt mind. 25% der Punkte erreicht werden und alle weiteren Aufgabenblätter mit mindestens 50% der Punkte bestanden werden.
- Die Aufgaben müssen in *Zweiergruppen* bearbeitet werden.
- Jede Hausaufgabe enthält eine optionale Zusatzaufgabe, die bei erfolgreicher Bearbeitung 1 Klausurpunkt zählt.
- Abgabe:
  - Die Abgabe erfolgt über CodeOcean[1]. CodeOcean ermöglicht die Bearbeitung und Ausführung der Quelldateien im Browser. Bei Bedarf können die Sources (Quelltexte) und Unit Tests von der Übungs-Webseite[2] heruntergeladen werden, um lokal zu entwickeln.
  - CodeOcean benutzt Java Version 8.
  - Alle Aufgaben sind über das CodeOcean einzureichen. CodeOcean ermöglicht das Anfügen von Kommentaren bei der Abgabe, darüber hinaus können auch Kommentare in den Source-Code geschrieben werden.
  - Geben Sie bei **jeder** Aufgabe beide Namen Ihrer Gruppe an.
  - Achten Sie darauf, dass Ihr Quelltext hinreichend kommentiert ist.

Aufgabe 1: Algorithmische Korrektheit

6 P

a) In die Methode `SquareMatrix multiply(double alpha)` der Klasse `assignment3.SquareMatrix`, mittels derer eine Skalarmultiplikation auf einer quadratischen Matrix ausgeführt werden kann, hat sich ein kleiner Fehler eingeschlichen. Korrigieren Sie diesen syntaktischen oder semantischen Defekt!
Hinweis: Fehler können auf verschiedene Arten gefunden werden - Analyse des Quellcodes, Debugging, Ausgabe von Zwischenergebnissen, Tests oder einer Kombination davon.

b) Die formale Sicherstellung der algorithmischen Korrektheit ist überaus aufwändig und kann demnach nicht für sämtliche Methoden einer komplexen Softwarelösung durchgeführt werden. Aus diesem Grund haben sich in der gängigen Praxis erprobte Softwaretests anhand umfassender Rahmenstrukturen wie etwa `JUnit` durchgesetzt. Bisher wurden Tests für die Aufgaben bereitgestellt - nun haben Sie die Aufgabe, selber Tests zu schreiben!

- Informieren Sie sich über die unterschiedlichen Testarten[3] zur empirischen Messung der verhältnismäßigen Testabdeckung[4]!
- Implementieren Sie die definierten Rümpfe und – falls nötig – zusätzliche Methoden der Klasse `assignment3.SquareMatrixTest`, sodass in Bezug auf die Methoden `double min()`, `double max()`, `SquareMatrix multiply(double alpha)` wie auch `SquareMatrix multiply(SquareMatrix other)` der Klasse `assignment3.SquareMatrix` eine vollständige Zweigüberdeckung erreicht wird!

[1] https://open.hpi.de/courses/pt2-2016
[2] https://hpi.de/naumann/teaching/current-courses/ss-16/uebung-programmierechnik-ii.html
[3] https://de.wikipedia.org/wiki/Kontrollflussorientierte_Testverfahren
[4] https://de.wikipedia.org/wiki/Testabdeckung

Aufgabe 2: Algorithmische Komplexität

5 P

Beantworten Sie die folgenden Aufgaben in einem Quelltextkommentar.

a) Ordnen Sie die folgenden)-Notationen vom langsamsten zum schnellsten Wachstum: O(2^n), O(n^3), O($\sqrt{n}$), O($n\sqrt{n}$), O(n^n), O(n^{logn}), O(n^2), O(n^2 log n), O(n), O(n log n), O(log n)

b) Bestimmen Sie, ob die folgenden Aussagen wahr sind und begründen Sie Ihre Antwort.

- $O(100 * N^2) \subseteq O(N^3)$
- $10 * N + 20 \in O(N)$
- $10 * N^{\frac{3}{2}} \in O(N \log N)$

Aufgabe 3: Autoboxing in Java

6 P

Java kennt sogenannte *primitive Datentypen* und *Wrapperklassen* für diese Datentypen - so gibt es `int` und `Integer`. In dieser Aufgabe sollen Sie die Unterschiede zwischen den beiden Typen analysieren. Implementieren Sie dazu Methoden der Klasse `assignment3.Autoboxing` und dokumentieren Sie Ihre gewählten Werte und Erkenntnisse in einem Quelltextkommentar.

a) Fügen Sie in der Methode `compareIntIntegerRuntime` 10.000.000 verschiedene Ganzzahlen in ein Array ein und messen Sie die Laufzeit hierfür. Erstellen Sie `int-Arrays` und `Integer-Arrays` und fügen Sie jeweils `int` oder `Integer` ein. Sie vergleichen also 4 verschiedene Settings. Messen Sie die Zeit für das Einfügen der Daten und die Zeit für das Erstellen der Arrays separat und berichten Sie über beide Zahlen. Damit Nebeneinflüsse wie z.B. CPU-Benutzung anderer Programme weniger Einfluss haben stellen Sie – wie im Plenum beschrieben – sicher, dass die *schnellste* der *Einfüge*-Messungen mindestens 500 ms dauert.

b) In der Methode `compareIntIntegerSize` soll die Größe von Arrays verglichen werden. Dies ist in Java nicht direkt über z.B. ein `sizeOf(int[])` möglich - allerdings bietet Java die Methoden `Runtime.getRuntime().freeMemory()` und `Runtime.getRuntime().totalMemory()` an, mit denen man den aktuell verbrauchten Speicher schätzen kann. Vergleichen Sie nun den verbrauchten Speicher für ein `int[]`, `Integer[]` und `ArrayList<Integer>`, die jeweils `arraySize` verschiedene Werte beinhalten. Berichten Sie für jede der 3 Möglichkeiten den belegten Speicher und den belegten Speicher pro int-Wert (Wählen Sie eine geeignete Einheit).

c) Die Methode `compareIntIntegerSize` liefert für `arraySize` = 16000 vernünftig aussehende Werte, diese variieren aber, wenn das Programm erneut ausgeführt wird. Für sehr kleine Werte (z.B. `arraySize` = 200) oder große Werte (z.B. `arraySize` = 500000) scheinen die gemeldeten Werte nicht stimmen zu können. Können Sie dieses Verhalten erklären und vielleicht sogar eine Lösung finden?

Aufgabe 4: Vergleich von Sortierverfahren

8 P

Um verschiedene Sortierverfahren miteinander zu vergleichen, können die durchgeführten Tausch- und Vergleichsopperationen gezählt und verglichen werden. Finden Sie herraus, wie verschiedene Sortierverfahren auf allen möglichen Permutationen der Zahlen 1 bis 10 abschneiden. Schreiben Sie Ihre Ergebnisse als Quelltextkommentar in `assignment3.CountingSortComparison`.

a) Implementieren Sie die Funktion `Vector<int[]> permutations(int[] a)` der Klasse `assignment3.Permutation`. Diese Funktion soll alle Permutationen des Arrays `a` zurückgeben. Sie dürfen davon ausgehen, dass alle Werte in `a` paarweise verschieden sind (d.h., die Anzahl der Unique-Werte in `a` ist die Anzahl der Werte in `a`). Stellen sie sicher, dass alle Tests erfolgreich behandelt werden.

b) Betrachten Sie die Klasse `assignment3.CountingSorter` und die gegebenen Algorithmen-Klassen `assignment3.CountingBubbleSorter`, `assignment3.CountingInsertionSorter` und `assignment3.CountingQuickSorter`. Wenn die gegebenen Sortieralgorithmen Vergleiche oder Vertauschoperationen durchführen werden diese in eine Methode ausgelagert, um diese besser analysieren zu können. Vervollständigen Sie dazu die Methoden in `assignment3.CountingSorter`, sodass Vergleichs- und Vertauschoperationen ausgeführt und gezählt werden und Statistiken über den best, worst und average case gesammelt werden.

c) Wenn Sie das Programm ausführen sollte Ihnen die maximale, minimale und gesamte Anzahl der durchgeführten Vergleiche und Vertauschungen verschiedener Sortieralgorithmen auf allen Permutationen der Zahlen 1 bis 10 ausgegeben werden. Welches Sortierverfahren benötigt...

- ...im besten Fall die wenigsten Vertauschungen.
- ...im schlechtesten Fall die meisten Vertauschungen.
- ...insgesamt die meisten Vertauschungen.
- ...im besten Fall die wenigsten Vergleiche.
- ...im schlechtesten Fall die meisten Vergleiche.
- ...insgesamt die meisten Vergleiche.

d) Benötigen Vergleichs- und Vertauschoperationen die gleiche Laufzeit? Erweitern Sie Ihren Code, messen Sie die Laufzeit, und kommentieren Sie Ihre Ergebnisse. Erläutern Sie Ihre Messmethode kurz.

e) Die gegebene Implementierung von Quicksort benutzt immer das linkeste Element als Pivotelement. Warum ist davon abzuraten?

Zusatzaufgabe: Rocket Science

1 P

Sie arbeiten bei einem großen Raumfahrtunternehmen und transportieren für Ihre Kunden Lasten ins Weltall. Ihr neuester Auftraggeber möchte, dass Sie für ihn eine Reihe von bis zu 1000 Satelliten mit je 1 Tonne Gewicht in den Orbit bringen.

Für die nächste Zeit ergibt sich genau ein Startfenster, daher können Sie nur genau eine große Rakete, Rocky McRocketface, starten. Leider ist Ihr Chief Executive Scientist of Rocket Calculations nach einem Pubcrawl in Las Vegas verschwunden und Sie können deshalb die Tragfähigkeit Ihrer Rakete nicht mehr zuverlässig berechnen. Daher probieren Sie es einfach aus und bauen Attrappen, mit denen Sie Testläufe durchführen. Ziel ist es, die Tragfähigkeit zu ermitteln, und zu erkennen, wie viele Satelliten Sie maximal mitführen können.

Gehen Sie davon aus, dass an Bord Ihrer Test-Raketen Platz für eine gewisse Anzahl n ($n \in \mathbb{N}_+$) an Satelliten-Nachbildungen ist. Ab einer Beladung von k Tonnen ($k \in \mathbb{N}_+$) und darüber sind die Raketen überladen und der Start endet äußerst spektakulär (was sehr schön anzusehen ist, aber leider auch sehr teuer). Raketen, die beim Start mit weniger als k Tonnen beladen waren, teilen dieses Schicksal nicht und können wiederverwendet werden.

a) Implementieren Sie die Methode `fewestFailures` der Klasse `RocketScience`, in der Sie die Tragfähigkeit unter Verlust so weniger Test-Raketen wie möglich ermitteln.
Hinweis: Dies sollte in k Versuchen möglich sein.

b) Nehmen Sie an Geld spielt keine Rolle, sondern Zeit. Implementieren Sie die Methode `fewestAttemptsLogN`, in der Sie die Anzahl der benötigten Versuche minimieren.
Hinweis: Sie sollten $O(\log n)$ Versuche benötigen.

c) Es geht auch anders! Finden Sie in der Methode `fewestAttemptsLogK` einen Weg, die Anzahl benötigter Versuche auf $O(\log k)$ zu reduzieren.

d) Stellen Sie sich vor, Sie haben nur genau zwei Test-Raketen. Finden Sie in `fewestAttempts TwoRockets` die Tragfähigkeit unter dieser Annahme mit möglichst wenigen Versuchen heraus.
Hinweis: Eine Lösung ist in $O(\sqrt{n})$ möglich.

Wie üblich: Denken Sie daran, Ihre Lösung im Quelltext gut nachvollziehbar zu kommentieren.