Prof. Dr. Gerd Stumme, Folke Eisterlehner, Dominik Benz
Fachgebiet Wissensverarbeitung

11.05.2009

3. Hausübung „Algorithmen und Datenstrukturen“

Sommersemester 2009

Abgabetermin: Montag, 18.05.2009, 10:00 Uhr

1 Theorie

1.1 Sortieralgorithmen.

1. In der Vorlesung wurde der Sortieralgorithmus „Bubblesort“ vorgestellt. Eine alternative zu diesem Sortieralgorithmus durch Vertauschen ist der in der Literatur häufig erwähnte Algorithmus „Shakersort“. Hierbei werden abwechselnd in aufsteigender und absteigender Feldindexfolge fehlstehende benachbarte Elemente vertauscht (siehe auch Präsenzübung). Welchen Vorteil mag dieses Verfahren gegenüber dem klassischen Bubblesort haben? Verändert sich dadurch die worst-case Laufzeit im Vergleich zu BubbleSort? (5 Punkte)

2. Betrachten Sie folgenden Algorithmus in Pseudocode:

```
Eingabe: Folge F der Länge n
   for i := n − 1 downto 1 do
      m := F[i];
      p := i;
      while p < n do
         if m > F[p + 1] then
            F[p] := F[p + 1];
            p := p + 1
         else
            Verlasse innere Schleife
         fi;
      od
      F[p] := m
   od
```

   a) Was berechnet obiger Algorithmus bei Eingabe einer Folge F der Länge n?

   b) Bestimmen Sie in O-Notation die Laufzeit des Algorithmus bezogen auf die Anzahl der Vergleiche für den besten, schlechtesten und mittleren (zu erwartenden) Fall.

   c) In der **while**-Schleife wird der kleinste Index p gesucht, so dass $F[p] < m \leq F[p+1]$ oder $p = n$ gilt um dann den Wert $F[n-i]$ nach Position p zu verschieben. Welche

Laufzeit ergäbe sich, wenn diese Position mit Hilfe einer binären Suche (die Folge $F[n-i+1], \ldots, F[n]$ ist aufsteigend sortiert) bestimmt und anschließend $F[n-i]$ zu der gefundenen Position verschoben würde? (15 Punkte)

1.
- Bei ShakerSort verringert sich die Größe des noch zu sortierenden Feldes mit jedem Durchlauf. Im Average-Case ergibt sich so eine bessere Laufzeit.
- Die Worst-Case-Laufzeit bleibt unverändert.

2. *a*) Der Algorithmus sortiert die Folge F.

b) In jedem Fall müssen (1) alle Elemente $i = n-1$ bis 1 betrachtet und (2) jeweils die richtige Position zum Einfügen bestimmt werden.

best-case: Die Folge ist bereits sortiert. Dann ist in Runde i die Einfügeposition gleich $n-i$ und wird in einem Schritt gefunden (O(1)). Dies wiederholt sich $n-1$-Mal.
$\rightsquigarrow$ Gesamt: O(n)

worst-case: Die Folge ist umgekehrt sortiert. Dann ist in Runde i die Einfügeposition gleich n und wird in $i-1$ Schritten gefunden (O(n)). Dies wiederholt sich $n-1$-Mal.
$\rightsquigarrow$ Gesamt: O(n^2)

average-case: Die Folge ist unsortiert (uniform zufällig). Dann ist in Runde i die Einfügeposition wahrscheinlich $\frac{2n-i+1}{2}$ und wird in $\frac{i+1}{2}$ Schritten gefunden. Gesamtanzahl der Vergleiche ist somit:

$$
\begin{aligned}
&(n)/2 + (n-1)/2 + (n-2)/2 + \cdots + 3/2 + 2/2 \\
&= \frac{(n) + (n-1) + \cdots + 2 + 1}{2} - \frac{1}{2} \\
&= \frac{1}{2} * \frac{n * (n-1)}{2} - \frac{1}{2} \\
&= \frac{n * (n-1)}{4} - \frac{1}{2} \\
&\approx \frac{n^2}{4}
\end{aligned}
$$

$\rightsquigarrow$ Gesamt: O(n^2)

c) An der Laufzeit würde sich asymptotisch (in O-Notation) nichts ändern, da nach Auffinden der Einfügeposition p in Runde i noch anschließend die Elemente an Positionen $n-i+1, \ldots, p$ jeweils um ein Feld nach „links“ verschoben werden müssen (im besten Fall sind das keine Elemente, im schlechtesten Fall $i-1$ und im erwarteten Fall $\frac{i-1}{2}$).

2 Programmierung

2.1 Java Collection Framework

Zur effektiven Abwicklung von Produktionsabläufen möchte die Firma EffectiveSolutions ihren Mitarbeitern ein System zur automatischen Sortierung aller anstehenden Aufgaben nach ihrer Dringlichkeit anbieten. Eine Aufgabe hat bei EffectiveSolutions folgende Eigenschaften:

- **jobName**: Titel der Aufgabe (z.B. „Projektbericht schreiben“)
- **initiator**: Der Name des Auftraggebers (z.B. „John“)
- **initiatorPosition**: Die Position des Auftraggebers (z.B. „Chef“)
- **daysLeft**: die Anzahl der Tage, die noch bis zur Fälligkeit der Aufgabe verbleiben (z.B. 10)
- **numberOfParticipants**: Die Anzahl der beteiligten Personen (z.B. 3)
- **funFactor**: Der „Spassfaktor“ in Prozent, wieviel Spass die Aufgabe macht (z.B. 20%)
- **reward**: Die Belohnung - wieviel Geld der Angestellte für die Erfüllung der Aufgabe bekommt (z.B. 100 Euro), begrenzt auf minimal 0 und maximal 10.000 Euro.

Bei der Sortierung der Aufgaben nach Priorität soll wie folgt vorgegangen werden (die Reihenfolge der Abarbeitung ist einzuhalten):

1. Aufgaben, deren Position des Auftraggebers „Chef“ lautet, sollen höhere Priorität haben als alle anderen Aufgaben.
2. Wenn es mehrere Aufträge vom Chef gibt, so soll die Aufgabe zuerst bearbeitet werden, die mehr Belohnung einbringt.
3. Aufträge, die *nicht* vom Chef kommen und bis zu deren Fälligkeitsdatum noch mehr als 20 Tage verbleiben, sollen die minimale Prioriät bekommen.
4. Für alle anderen Aufträge wird die Priorität p berechnet nach der Formel

$$p = \frac{funFactor + reward}{numberOfParticipants + daysLeft}$$

Die Klasse `Job.java` von der Vorlesungsseite beschreibt eine solche Aufgabe.

1. Erweitern Sie die Klasse in der Form, dass sie das Interface `Comparable`, das in der Vorlesung vorgestellt wurde, implementiert. Die Methode `compareTo` soll hierbei die Priorität zweier Aufgaben vergleichen. Bei diesem Vergleich soll nach dem oben beschriebenen Verfahren vorgegangen werden. Erweitern Sie die Klasse des Weiteren um eine Methode `getPriority()`, die einen Zahlenwert der Priorität zurückliefert; die maximale Priorität soll hierbei `1000000` sowie die minimale `0` sein. (8 Punkte)

2. Die Klasse `ExampleJobs.java` von der Vorlesungsseite enhält die statische Methode `Jobs[] getJobs()`, die ein Array von Beispiel-Aufgaben liefert. Verwenden Sie diese Methode, um Testdaten zu erhalten; nutzen Sie anschließend die interne Java-Methode `Arrays.sort`, um die Testdaten anhand ihrer Priorität sortiert auszugeben. Die Ausgabe sollte so aussehen (nur die ersten drei Einträge sind hier aufgelistet; $< X_i >$ steht jeweils für die errechnte Priorität): (4 Punkte)

```
Job nummer 1 [Priorität <X1>]:
Name: "Tischvorlage ausarbeiten"
  Auftraggeber  : Markus (Chef)
  Restzeit      : 22
  Teilnehmer    : 2
  Spaßfaktor    : 22.0
  Belohnung     : 5194.0

Job nummer 2 [Priorität <X2>]:
Name: "Arbeitsstelle ausschreiben"
  Auftraggeber  : Norbert (Chef)
  Restzeit      : 46
  Teilnehmer    : 0
  Spaßfaktor    : 24.0
  Belohnung     : 4818.0

Job nummer 3 [Priorität <X3>]:
Name: "Hausmeister feuern"
  Auftraggeber  : Otto (Chef)
  Restzeit      : 13
  Teilnehmer    : 2
  Spaßfaktor    : 19.0
  Belohnung     : 3814.0
...
```

3. Welches Interface aus dem Java Collections Framework müssten Sie verwenden, wenn es innerhalb der Firma mehrere Berechnungsvorschriften für die Priorität einer Aufgabe gäbe? (2 Punkte)

2.2 Mergesort

Implementieren sie in geeigneter Weise den Sortieralgorithmus *MergeSort*, um die Beispiel-Aufgaben nach Priorität absteigend sortiert auszugeben. Die Ausgabe soll das gleiche Format haben wie in der vorigen Teilaufgabe. Die "richtige" Reihenfolge (die natürlich auch für die vorherige Teilaufgabe gilt) ist dabei folgende: (8 Punkte)

```
Job nummer 1: Tischvorlage ausarbeiten
Job nummer 2: Arbeitsstelle ausschreiben
Job nummer 3: Hausmeister feuern
Job nummer 4: Kaffee kochen
Job nummer 5: Flug buchen
Job nummer 6: Pause machen
Job nummer 7: Bilanz erstellen
Job nummer 8: Neuen Firmenwagen kaufen
Job nummer 9: Hausuebung bearbeiten
Job nummer 10: Projektbericht schreiben
Job nummer 11: Werbesendung schalten
Job nummer 12: Sortieralgorithmus implementieren
Job nummer 13: Website neu gestalten
Job nummer 14: Vorlesung 'Alogrithmen und Datenstrukuren' besuchen
Job nummer 15: Testphase starten
Job nummer 16: Buero reinigen
Job nummer 17: Tischfussball spielen
Job nummer 18: Abschlussbericht verfassen
Job nummer 19: Konferenzbeitrag verfassen
Job nummer 20: Tagung organisieren
```

Viel Spaß und viel Erfolg!