Praktische Informatik 3 WS 06/07

2. Übungsblatt

Ausgabe: 14.11.06
Bearbeitungszeit: Zwei Wochen

Christoph Lüth <cxl>
Matthias Berger <tokio>
Klaus Hartke <hartke>
Cui Jian <ken>
Friederike Jolk <rikej>
Christian Maeder <maeder>
Diedrich Wolter <dwolter>

4 *Sinus* — *5 Punkte*

Trigonometrische Funktionen sind etwas Praktisches, wenn man zum Beispiel Rotationswinkel berechnen will. Im Standard-Prelude sind zwar trigonometrische Funktionen vordefiniert, aber diese benutzen natürlich Fließkommazahlen; das hat unter anderem das Problem der beschränkten Genauigkeit.

Deshalb wollen wir in diesem kurzen Beispiel eine Funktion implementieren, die den Sinus für rationale Zahlen (`Rational`) beliebig genau approximiert. Dies erfolgt durch die Reihenentwicklung nach Taylor:

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ldots$$

Implementieren Sie eine Funktion

```
sinus :: Rational-> Rational
```

welche den Sinus durch eine Reihenentwicklung approximiert. Eine globale Konstante

```
epsilon :: Rational
epsilon = 1/10^6
```

gibt an, bis zu welcher Genauigkeit gerechnet werden soll.

5 *Wörter suchen* — *5 Punkte*

Implementieren Sie eine Funktion

```
histogram :: String-> [(String, Int)]
```

welche die in einem Text vorkommenden Wörter zählt und zusammen mit ihrer Häufigkeit in absteigender Häufigkeit zurückgibt. Ein *Wort* ist dabei definiert als eine durch Leerzeichen, Sonderzeichen oder numerische Zeichen begrenzte Folge von Buchstaben. Beim Vergleich zweier Wörter sollte unterschiedliche Groß- und Kleinschreibung keine Rolle spielen.

Hinweise: Implementieren Sie der Reihe nach folgende Funktionalität:

1. Ersetzen Sie alle Sonderzeichen und numerischen Zeichen durch Leerzeichen.
2. Nutzen Sie die vordefinierte Funktion `words`.
3. Definieren Sie eine Funktion

   ```
   ins :: String-> [(String, Int)]-> [(String, Int)]
   ```

   welche ein Wort in eine Liste von Häufigkeiten einfügt, indem entweder die Häufigkeit für dieses Wort erhöht wird oder das Wort neu mit der Häufigkeit 1 eingefügt wird.
4. Fügen Sie rekursiv alle Worte aus Schritt 2 ein, beginnend mit der leeren Liste.
5. Sortieren Sie die Liste mit Ihrem Lieblingssortieralgorithmus, indem Sie nur die Häufigkeiten vergleichen.

6 *Hallo, Nachbar!* *10 Punkte*

Manch ein Punkt fühlt sich verlassen und einsam in den Euklidischen Weiten. Kein anderer Punkt in der Nähe! Wirklich nicht?

Diese Aufgabe will einsamen Punkten helfen, ihre Nachbarn zu finden. Genauer gesagt geht es um das *closest-pair-problem*, welches darin besteht, aus einer Menge P von Punkten die zwei Punkte $p, q \in P$ zu finden, die den geringsten Abstand haben, d.h. für alle anderen $x, y \in P$ ist $|x - y| \geq |p - q|$. Hierbei ist der Abstand definiert als die Länge des Distanzvektors, i.e.

$$|(p_x, p_y) - (q_x, q_y)| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Punkte sind für uns einfach Tupel von Fließkommazahlen

```
type Point = (Double, Double)
```

Die Signatur der zu implementierenden Funktion ist damit

```
closestPair :: [Point]-> (Point, Point, Double)
```

Das allereinfachste Verfahren zur Berechnung des *closest pair* (engsten Paares) ist einfach, alle Paare durchzuprobieren (*brute force*). Diese Funktion nennen wir

```
simpleCP :: [Point]-> (Point, Point, Double)
```

Der Aufwand ist allerdings quadratisch $\Theta(n^2)$, was für große Punktmengen unpraktikabel sein kann. In dieser Aufgabe lernen wir einen Algorithmus kennen, der nur Aufwand $O(n \log n)$ hat. Dieser Algorithmus funktioniert rekursiv:

- Die Menge der Punkte wird geteilt, in dem wir eine vertikale Linie l so durch die Punkte legen, dass die eine Hälfte links und die andere Hälfte rechts zu liegen kommt.
- Danach werden rekursiv die engsten Paare (p_1, q_1, δ_1) und (p_2, q_2, δ_2) der beiden Teilmengen berechnet.
- Jetzt ist entweder das Minimum δ von δ_1 und δ_2 das engste Paar, oder das engste Paar besteht aus je einem Punkt links und rechts von l. Um diesen Fall zu prüfen, ist es ausreichend, alle Punkte, die innerhalb des Intervalls von δ rechts und links von l liegen, zu betrachten.

  Eine einfache Näherungslösung ist hier, auf alle diese Punkte das brute-force-Verfahren `simpleCP` anzuwenden (Variante A). Allerdings kann dies in ungünstigen Fällen wieder quadratischen Aufwand bedeuten.

  Eine bessere, aber aufwändigere Variante (Variante B) ist folgende: für jeden einzelnen dieser Punkte kann ein potentielles engeres Paar nur noch innerhalb eines Rechteckes liegen, welches aus zwei Quadraten der Kantenlänge δ entlang der Linie l besteht. Dieses Rechteck kann höchstens 8 Punkte enthalten; deshalb genügt es, für jeden Punkt p die nächsten sieben Punkte zu prüfen, die innerhalb des Intervalls liegen.
- Die Rekursion terminiert, wenn wir nur noch drei Punkte oder weniger haben; in diesem Fall wenden wir die brute-force-Methode `simpleCP` von oben an.

Algorithmisch implementieren wir die Teilung entlang der vertikalen Linie l, indem wir eine nach den X-Koordinaten aufsteigend sortierte Liste in der Mitte teilen. Entscheidend hierbei ist, dass nur *einmal* sortiert wird, und *nicht* bei jedem rekursiven Aufruf! Daher hat die rekursive Funktion die Signatur

```
cp :: [Point]-> [Point]-> (Point, Point, Double)
```

wobei die beiden Argumentlisten die gleiche Menge von Punkten enthalten, aber einmal sortiert nach aufsteigenden X-Koordinaten und einmal sortiert nach aufsteigenden Y-Koordinaten. Aus diesen werden immer nur Punkte entfernt, nie wieder hinzugefügt; daher bleibt die Sortierung erhalten.

Punkteverteilung für die Aufgabe:

1. Implementierung der brute-force-Methode `simpleCP` (*3 Punkte*)
2. *Entweder* Implementierung von `cp` mit Variante A (*4 Punkte*)
3. *Oder* Implementierung von `cp` mit Variante B (*7 Punkte*)

Bitte begründen Sie, ob und warum Ihre Lösung den gewünschten Aufwand $O(n \log n)$ hat.

Dies ist Version 1.1 vom 2006/11/18 11:17:18.