Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. C. Böhm, G. Zhang

Übungen zu Einführung in die Informatik

Hinweis: Freiwillige Abgabe vom Mittwoch, den 12.12.07 bis zum Montag, den 17.12.07, 12.00

Aufgabe 9-1 **Vererbung**

Dr. Matthias Hölzl will Vererbungspolymorphie in Java lernen. Er hat eine einfache Vererbungshierarchie mit vier Klassen `A`, `B`, `C` und `D` implementiert und in der Klasse `Inheritance` einige Tests geschrieben.

a) Laden Sie die Klassen von unserer Web-Seite herunter und führen Sie das Programm aus. Welches Ergebnis erhalten Sie? Können Sie Dr. Hölzl erklären, warum sein Programm dieses Ergebnis produziert?

b) Kommentieren Sie die auskommentierten Methodenaufruf wieder ein. Welche Fehler meldet der Java-Compiler? Können Sie Dr. Hölzl erklären, warum der Compiler diese Methodenaufrufe nicht akzeptiert?

Aufgabe 9-2 **Maximale Summe Zusammenhängender Teilfolgen** (keine Abgabe)

Die maximale Summe zusammenhängender Teilfolgen (MSzT) einer Folge (möglicherweise negativer) ganzer Zahlen $A_1, A_2, \ldots, A_N$ ist der maximale Wert von $\Sigma_{k=i}^{j} A_k, 1 \leq i \leq j \leq N$. Bei $A = \{-2, \mathbf{11, -4, 13}, -5, 2\}$ gilt also $MSzT(A) = 20$ und bei $A = \{1, -3, \mathbf{4, -2, -1, 6}\}$ gilt $MSzT(A) = 7$. Falls eine Folge nur aus negativen Zahlen besteht, ist ihre MSzT per definitionem 0.

In seinem Buch [1] stellt Mark Allen Weiss insgesamt vier Algorithmen verschiedener Komplexitäten vor, die die MSzT einer gegebenen Folge berechnen. Wir betrachten in dieser Aufgabe drei dieser vier Algorithmen. Die Hilfsklasse `MCSSResult` können Sie von unserer Web-Seite herunterladen.

```
MCSSResult mcss1(int[] a)
{
  int sum = 0;
  int start = -1;
  int end = -1;

  for (int i = 0; i< a.length; i++)
    for (int j = i; j< a.length; j++)
    {
      int thisSum = 0;
      for (int k=i; k<=j; k++)
      {
        thisSum += a[k];
        if (thisSum > sum)
        {
          sum = thisSum;
          start = i;
          end = j;
        }
      }
    }
  MCSSResult res =
     new MCSSResult();
  res.sum = sum;
  res.start = start;
  res.end = end;
  return res;
}
```

```
MCSSResult mcss2(int[] a)
{
  int sum = 0;
  int start = -1;
  int end = -1;

  for (int i = 0; i< a.length; i++)
  {
    int thisSum = 0;
    for (int j = i; j< a.length; j++)
    {
      thisSum += a[j];
      if (thisSum > sum)
      {
        sum = thisSum;
        start = i;
        end = j;
      }
    }
  }
  MCSSResult res = new
      MCSSResult();
  res.sum = sum;
  res.start = start;
  res.end = end;
  return res;
}
```

```
MCSSResult mcss3(int[] a)
{
  int sum = 0;
  int thisSum = 0;
  int start = -1;
  int end = -1;
  int i = 0;
  int j = 0;
  while (j< a.length)
  {
    thisSum += a[j];
    if (thisSum > sum)
    {
      sum = thisSum;
      start = i;
      end = j;
    } else if (thisSum < 0)
    {
      i = j + 1;
      thisSum = 0;
    }
    j++;
  }
  MCSSResult res =
      new MCSSResult();
  res.sum = sum;
  res.start = start;
  res.end = end;
  return res;
}
```

a) Geben Sie die Zeit-Komplexität der drei oben vorgestellten Algorithmen in der O-Notation an. Geben Sie eine kurz Begründung an.
Hinweis: Es gibt insgesamt $N(N+1)(N+2)/6$ mögliche Zahlenkombinationen für i, j, und k, so dass $1 \leq i \leq k \leq j \leq N$ gilt.

b) Laden Sie das Programm `MCSS.java` von unserer Web-Seite herunter und implementieren Sie eine Methode `generateNumbers`, die einen Parameter `n` bekommt und `n` ganze Zahlen zwischen -50 und 50 zufallsgeneriert und in einem `int` Array zurückgibt.
Hinweis: `Math.random()`

c) Lassen Sie 10, 100 bzw. 500 ganze Zahlen generieren und testen Sie die drei Methoden. Lassen Sie 1000, 10000, 20000 bzw. 30000 ganze Zahlen generieren und testen Sie die Methoden `mcss2` und `mcss3`. Insbesondere sollten die Methoden bei gleicher Eingabe die gleiche MSzT zurückgeben.
Achtung: `mcss1` ist bei großen Zahlenmengen sehr langsam!

Literatur

[1] Mark Allen Weiss. *Data Structures & Problem Solving Using Java.* Addison-Wesley, 1997.