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

Übungen zu Einführung in die Informatik (Lösungsvorschlag)

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

a) Bestimmen Sie die Zeit-Komplexität der drei oben vorgestellten Algorithmen. in der O-Notation.
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.

In seinem Buch [1] analysiert Mark Allen Weiss die Algorithmen wie folgt:

```
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++){  // (1)
        thisSum += a[k];
        if (thisSum > sum){      // (2)
          sum = thisSum;
          start = i;
          end = j;
        }
      }
    }
  MCSSResult res = new MCSSResult();
  res.sum = sum;
  res.start = start;
  res.end = end;
  return res;
}
```

In *mcss1* ist es entscheidend, wie oft der Rumpf der innersten Schleife ausgeführt wird. Denn in jeder Iteration von `(1)` wird ein mal verglichen (`2`). Sei n die Eingabegröße (die Länge der Folge `a`), so laufen i von 0 bis $n-1$, j von i bis $n-1$, und k von i bis j. Die Anzahl der Ausführungen des Rumpfs von der Schleife `(1)` ist also die Anzahl aller möglichen Zahlenkombinationen für i, j, und k, so dass $0 \leq i \leq k \leq j \leq n-1$ gilt. Diese Zahl, wie in [1] bewiesen, ist $n(n+1)(n+2)/6$. Damit hat der Algorithmus eine Komplexität von $O(n^3)$.

```
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++){ //(3)
      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;
}
```

mcss2 optimiert *mcss1*, indem die innerste Schleife `(1)` gespart wird: wenn $\Sigma_{k=i}^{j-1} A_k$ schon berechnet ist, dann braucht man nicht wieder von vorne anfangen, um $\Sigma_{k=i}^{j} A_k$ zu berechnen, sondern nur A_j auf die alte Summe zu addieren. Die Größenordnung der Zeitkomplexität des Rumpfs der Schleife `(3)` ist $O(1)$. Da der Rumpf insgesamt $n + (n-1) + (n-2) + \cdots + 1 + 0 = n(n+1)/2$ mal ausführt wird, haben wir $T_{mcss2}(n) \in O(n^2)$. Im Allgemeinen gilt, dass im Programm p

```
for (int i=0; i<n; i++)
  for (int j=i; j<n; j++)
    S;
```

S $n(n+1)/2$ mal ausgeführt wird. Gilt zusätzlich $T_S \in O(1)$, so hat man $T_p \in O(n^2)$.

```
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;
}
```

Schließlich geht *mcss3* noch einen Schritt weiter. Es wird die Tatsache berücksichtigt, dass eine Teilfolge, deren Summe negativ ist, nicht am Anfang der Teilfolge stehen kann, deren Summe die MSzT der gegebenden Folge ist. Der Algorithmus hat folgende Form:

```
for (j=0; j<n; j++)
  S;
```

mit $T_S(n) \in O(1)$. Damit haben wir $T_{mcss3}(n) \in O(n)$.

Literatur

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