Leibniz Universität Hannover
FG Graphische Datenverarbeitung
Prof. Dr. F.-E. Wolter
M.Sc Maximilian Klein

Datenstrukturen und Algorithmen

Stundenübung 1 – Pseudocode und Grundlagen

Aufgabe 1.1 – Rekursion und Iteration

Gegeben ist die rekursive Implementation zur Berechnung der Fibonacci Zahlen.

```
function fibonacci(n){
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

(a) Geben Sie eine nicht rekursive Implementierung an! Dazu müssen Sie über die Fibonacci Zahlen iterieren bis Sie zu der gewünschten Zahl kommen.

(b) Berechnen Sie die Anzahl an Additionen in der iterativen Implementation

Lösung

(a)

```
function fibonaccilter(n){
 if(n==0) return 0;
 fn = 1; // = f1
 flastn = 0; // = f0

 for(i = 2; i<=n; i=i+1){
     newfn = fn + flastn;
     flastn = fn;
     fn = newfn;
 }
 return fn;
}
```

(b) Es gibt im iterativen Programm zwei Stellen an denen addiert wird. Einmal wird das `i` in der for-Schleife hochgezählt und einmal wird `newfn` über die Summe der beiden Vorgänger berechnet. Daraus ergibt sich die Folgende Laufzeitfunktion.

$$f_{fib+}(n) = \sum_{2}^{n} 2 = 2 \cdot (n - 2 + 1)$$
$$= 2n - 2$$

Aufgabe 1.2 – Suche in Listen

Schreiben Sie einen Algorithmus, der in einem gegebenen unsortierten Array A nach dem m-ten vorkommen des Zeichens c sucht. Geben Sie die Stelle aus, an der das Zeichen gefunden wurde. Sollte das Zeichen nicht mindestens m Mal in dem Array vorkommen soll -1 ausgegeben werden.

Lösung

Der Algorithmus könnte wir folgt aussehen

```
function searchCharacter(A,n,m,c){
    for(i=0; i<n; i++){
        if(A[i] == c){
            m--;
            if(m == 0){
                return i;
            }
        }
    }
    return -1;
}
```


Aufgabe 1.3 – Optimale Matrizenmultiplikation

Im Folgenden versuchen wir die optimale Reihenfolge zum Multiplizieren von Matrizen zu finden.

(a) Gegeben seien die vier Matrizen $A \in \mathbb{R}^{n\times m}, B \in \mathbb{R}^{m\times k}, C \in \mathbb{R}^{k\times l}$ und $D \in \mathbb{R}^{l\times r}$. Es soll das Produkt:

$$A \cdot B \cdot C \cdot D$$

berechnet werden. Geben Sie eine Klammerung an, die für $n = 10, m = 5, k = 20, l = 8$ und $r = 4$ die Anzahl an skalaren Multiplikationen minimiert.

Hinweis: Bei der Multiplikation von $A \cdot B$ werden $n \cdot m \cdot k$ skalare Multiplikationen durchgeführt.

(b) Geben Sie eine Implementation von `minCost` an, die alle Möglichkeiten für die Matrizenmultiplikation durchtestet und die Kosten für die beste Klammerung ausgibt. Testen Sie dafür alle möglichen Kombinationen. Als Eingabe wird ein Array übergeben, das die einzelnen Matrixdimensionen beinhaltet. Für das obige Beispiel sähe das Array z.B. wie folgt aus:

```
d = [ 10, 5, 20, 8, 4 ]
```

Hinweis: Verwenden Sie `remove(d,i)`, um eine Kopie vom Array d zu erhalten in der das i-te Element gelöscht wurde. Bsp: `remove([1,2,3],0) = [2,3]` oder `remove([1,2,3],1) = [1,3]`.

(c) Geben Sie die Rekursionsformel $r_{\text{minCost}}(n)$ an, die die Anzahl der Rekursionen von `minCost` angibt.

Lösung

(a) Es gibt die folgenden 5 Möglichkeiten:

- $(A \cdot B) \cdot (C \cdot D)$: $nmk + klr + nkr = 1000 + 640 + 800 = 2440$
- $((A \cdot B) \cdot C) \cdot D$: $nmk + nkl + nlr = 1000 + 1600 + 320 = 2920$
- $A \cdot (B \cdot (C \cdot D))$: $klr + mkr + nmr = 640 + 400 + 200 = 1240$
- $(A \cdot (B \cdot C)) \cdot D$: $mkl + nml + nlr = 800 + 400 + 320 = 1520$
- $A \cdot ((B \cdot C) \cdot D)$: $mkl + mlr + nmr = 800 + 160 + 200 = 1160$

(b) Dauer: ca. 10Minuten

```
function minCost(d,n){
    if(n <= 2) return 0;
    minCosts = Infinity;
    // bei jeder Multiplikation fällt die mittlere Dimensionszahl weg
    for(i=1; i<n-1; i++){
        // entferne die mittlere Dimension der Matrizen die
        // miteinander multipliziert werden sollen.
        dRemove = remove(d,i);
        // Kosten für die Multiplikation
        costsI = d[i-1]*d[i]*d[i+1];
        // Gesamtkosten rekursiv + diese Multiplikation
        costs = minCost(dRemove,n-1)+costsI;
        minCosts = min(costs,minCosts);
    }
    return minCosts;
}
```

(c) Es gilt:

$$r_{\text{minCost}}(n) = \begin{cases} 1 & \text{für } n \leqslant 2 \\ \sum_{k=1}^{n-1} r_{\text{minCost}}(n-1) = (n-2) r_{\text{minCost}}(n-1) & \text{sonst} \end{cases}$$
$$= (n-2)!$$

Wobei gilt:

$$n! \geqslant 2^n \ \forall n > 4$$

Für Zuhause

Die folgende Aufgabe behandelt die Methode der dynamischen Programmierung. Diese Methode ist eine häufig verwendete Strategie in vielen Algorithmen. Im Rahmen dieser Vorlesung ist sie aber nicht klausurrelevant.

Aufgabe 1.4 – Dynamische Programmierung

(a) Das Prinzip der dynamischen Programmierung funktioniert bei Problemen, deren Lösung sich durch die Lösung der Teilprobleme zusammensetzen lässt. In unserem Fall heißt das, dass es für eine Matrixmultiplikationskette mindestens eine optimale Teilung gibt. Wobei eine Teilung bedeutet, dass die beiden Teile die entstehen, die Matrizen ergeben, die als letztes multipliziert werden. Im Beispiel von drei Matrizen A_1, A_2, A_3 könnte so ein Teilung bei $k = 1$ oder $k = 2$ liegen, dann würde die Klammerung so aussehen:

$$A_1(A_2A_3) \qquad \text{oder} \qquad (A_1A_2)A_3$$

Für eine Multiplikationskette der Länge n, also $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ führt eine Teilung an der Stelle k zu

$$(A_1A_2 \ldots A_k)(A_{k+1}A_{k+2} \ldots A_n)$$

Um nun das optimale k zu finden, also das k für das die skalaren Multiplikationen minimiert werden, muss man alle kleineren Matrizenketten berechnen, oder berechnet haben. Das Prinzip der dynamischen Programmierung macht genau das. Es fängt bei den kleinsten Teilketten an und berechnet deren Kosten. Die kleinsten Teilketten sind die der Länge 1, also solche die nur eine Matrix enthalten und somit keine Kosten verursachen. Darauf folgen alle Ketten der Länge 2, danach die Ketten der Länge drei.

All diese Kosten werden in eine Matrix M der Größe $n \times n$ gespeichert. Dabei gibt $m_{i,j}$, die Zahl in der Zelle i, j, die minimalen Kosten für die Berechnung der Teilkette $A_{i,j} = A_iA_{i+1} \ldots A_j$ an. Die folgende Rekursionsformel gibt dabei an, wie man aus bereits berechneten Kosten die Kosten größerer Ketten berechnet:

$$m_{i,i} = 0$$
$$m_{i,j} = \min_{i \le k < j}(m_{i,k} + m_{k+1,j} + d_{i-1}d_kd_j) \qquad \text{für } i < j$$

Mit den Dimensionen $d_0 \ldots d_n$ wobei die Matrix A_i die Dimension $d_{i-1} \times d_i$ hat und die Kette $A_{i,j}$ hat die Dimension $d_{i-1} \times d_j$.

Berechnen Sie die Matrix M für das Beispiel aus Aufgabe 1.3.

(b) Schreiben Sie das Programm `minCostDP`, dass die beste Matrixkette mittels dynamischer Programmierung berechnet. Als Eingabe erhalten Sie wieder ein Array der Dimensionen.

(c) Schätzen Sie die Anzahl der Multiplikation von `minCostDP` großzügig nach oben ab und vergleichen Sie diese mit der naiven Implementation.

Lösung

(a) Zunächst sollten die Diagonalelemente eingetragen werden.

$$M = \begin{matrix} 0 & (1) & (4) & (6) \\ . & 0 & (2) & (5) \\ . & . & 0 & (3) \\ . & . & . & 0 \end{matrix}$$

Als nächstes geht man die Nebendiagonalen durch. Im obigen Beispiel ist eine mögliche Reihenfolge durch die geklammerten Zahlen angedeutet. Wir berechnen zunächst (1) bzw. $m_{1,2}$ was der Verkettung $A_{1,2} = A_1 A_2$ entspricht. Hier ist das Minimum, genauso wie bei (2), (3) einfach die Kosten für die Multiplikation der entsprechenden Matrizen, also:

$$M = \begin{matrix} 0 & 1000 & (4) & (6) \\ . & 0 & 800 & (5) \\ . & . & 0 & 640 \\ . & . & . & 0 \end{matrix}$$

Damit haben wir alle möglichen Paare von Matrizenmultiplikationen. Nun folgen die beiden Tripel (4) und (5), die errechnen sich aus der Rekursionsformel $m_{i,j} = \min_{i \leqslant k < j}(m_{i,k} + m_{k+1,j} + d_{i-1} d_k d_j)$. Für (4), also der Multiplikation $A_1 A_2 A_3$ gibt es zwei Teilungspunkte $k = 1$ und $k = 2$, aus den beiden berechnen wir nun die minimalen Kosten für das Tripel (4).

$$m_{1,3} = \min_{1 \leqslant k < 3}(m_{1,k} + m_{k+1,3} + d_0 d_k d_3)$$

$$m_{1,3} = \min\left\{ \underbrace{m_{1,1} + m_{2,3} + 10 \cdot 5 \cdot 8}_{k=1} \;,\; \underbrace{m_{1,2} + m_{3,3} + 10 \cdot 20 \cdot 8}_{k=2} \right\}$$

$$m_{1,3} = \min\{800 + 400, 1000 + 1600\} = 1200$$

Analog für $m_{2,4}$

$$m_{2,4} = \min_{2 \leqslant k < 4}(m_{2,k} + m_{k+1,4} + d_1 d_k d_4)$$

$$m_{2,4} = \min\left\{ \underbrace{m_{2,2} + m_{3,4} + 5 \cdot 20 \cdot 4}_{k=2} \;,\; \underbrace{m_{2,3} + m_{4,4} + 5 \cdot 8 \cdot 4}_{k=3} \right\}$$

$$m_{2,4} = \min\{640 + 400, 800 + 160\} = 960$$

Zwischenstand:

$$M = \begin{matrix} 0 & 1000 & 1200 & (6) \\ . & 0 & 800 & 960 \\ . & . & 0 & 640 \\ . & . & . & 0 \end{matrix}$$

Und für die letzte Zahl werden es ein paar mehr Berechnungen...

$$m_{1,4} = \min_{1 \leqslant k < 4} (m_{1,k} + m_{k+1,4} + d_0 d_k d_4)$$

$$m_{1,4} = \min \Bigg\{ \underbrace{m_{1,1} + m_{2,4} + 10 \cdot 5 \cdot 4}_{k=1} \; , \; \underbrace{m_{1,2} + m_{3,4} + 10 \cdot 20 \cdot 4}_{k=2} \; , \; \underbrace{m_{1,3} + m_{4,4} + 10 \cdot 8 \cdot 4}_{k=3} \Bigg\}$$

$$m_{1,4} = \min\{960 + 200, 1000 + 640 + 800, 1200 + 320\} = 1160$$

Also:

$$M = \begin{matrix} 0 & 1000 & 1200 & 1160 \\ . & 0 & 800 & 960 \\ . & . & 0 & 640 \\ . & . & . & 0 \end{matrix}$$

(b) Iterativ

```
function minCostDP(d,n) {
 // erstelle Matrix mit nur Nullen, der Größe nxn
 m = (n,n)*[0];
 for(L = 2;L < n; L++) { // L = Länge der Kette
     for(i = 1; i<n-L+1; i++) {
         j = i + L - 1;
         m[i,j] = Infinity;
         for(k = i; k<j-1; k++){ // prüfe alle Teilungen
             q = m[i, k] + m[k+1, j] + d[i-1]*d[k]*d[j]
             if (q < m[i, j]) {
                 m[i,j] = q;
             }
         }
     }
 }
 return m[1,n];
}
```

Oder gern auch rekursiv (wahrscheinlich einfacher):

```
function minCostDP(d,n) {
 // erstelle Matrix mit nur -1en, der Größe nxn
 m = -1*[n,n]
 // verwende Memoisation und berechne rekursiv den
 // Eintrag an der Stelle 1,n
 return minCostDP_M(d,n,1,n,m);
}
```

```
function minCostDP_M(d,n,i,j,m){
 // Memoisation
 if(m[i,j] != -1) return m[i,j];
 // Diagonalelement
 if(i == j) return 0;
 // Minimum nach Rekursionsformel berechnen
 m[i,j] = Infinity;
 // Alle Teilungen beachten
 for(k=i,k<j-1;k++){
      q = minCostDP_M(d,n,i,k,m) + minCostDP_M(d,n,k+1,j,m)
          + d[i-1]*d[k]*d[j];
      if(q < m[i,j]){
          m[i,j] = q;
      }
 }
 return m[i,j];
}
```

(c) An der iterativen Variante sieht man sehr schön, das für jede Zelle höchstens $n \cdot 2$ Multiplikationen geschehen. Die Matrix hat n^2 Elemente von denen wir zwar nicht alle berechnen, aber das ist großzügig genug. Also haben wir höchstens

$$2n^3$$

Multiplikationen, was normalerweise deutlich kleiner ist als die 2^n von Aufgabenteil 1.3.