Aufgaben zu Algorithmen - Beispiellösungen

Aufgabe 1 - Struktogramm

Das Struktogramm zeigt einen Algorithmus, der von einer gegebenen Zahl n durch wiederholtes dekrementieren (Bis n gleich 0 ist) die Fakultät bildet. Die Komplexität des Algorithmus wächst linear mit der Größe der gegebenen Zahl n, wodurch sich die Komplexität mit $O(n)$ ergibt.

Das $\tau(n)$ zu bestimmen ist hier nicht unbedingt nötig. Auch wird das Bestimmen von $\tau(n)$ meistens nur für explizite Codeausschnitte durchgeführt, da die Anzahl der Operationen in einem Struktogramm nicht immer eindeutig bestimmbar ist. Näherungsweise lässt sich für das gegebene Struktogramm aber folgende Operationen für $\tau(n)$ ableiten:

- Einlesen von n - 1 Operation, einmal durchgeführt
- Setzen von *res* - 1 Operation, einmal durchgeführt
- Berechnung $res = res * n$ - 2 Operationen (Multiplikation und Zuweisung), n mal ausgeführt[1]
- Dekrementieren von n - 1 Operation, n mal ausgeführt
- Vergleich $n! = 0$ - 1 Operation, n mal ausgeführt
- Ausgabe *res* - 1 Operation, einmal ausgeführt

Aufgrund dieser Operationen lässt sich bestimmen:

$$\tau(n) = 1 + 1 + 2 \cdot n + n + 1 = 3 + 3 \cdot n = c_1 \cdot n + c_2$$

Nach Wegstreichen der konstanten Faktoren erkennt man, dass der skalierende Faktor n ist.

Bei genauerer Betrachtung fällt auf, dass der Algorithmus durch eine schlecht gewählte Abbruchbedingung nicht in allen Fällen korrekt funktioniert. Dadurch, dass am Ende des Schleifendurchlaufs n lediglich gegen 0 geprüft wird, würde der Algorithmus für alle $n <= 0$ in einer Endlosschleife enden. Nach Ende des ersten Schleifendurchlaufs wäre n in jedem Fall kleiner als 0 und die Abbruchbedingung könnte nicht mehr erfüllt werden (Außer ggf. durch Integer Underflow). Besser geeignet wäre für diesen Algorithmus die Nutzung einer Zählschleife (*for-loop*) oder die Abbruchbedingung als $n > 0$ zu formulieren.

Aufgabe 2

Um einen Algorithmus zu entwerfen, der den größten ganzzahligen Teiler einer gegebenen Zahl n ermittelt, gibt es verschiedene Herangehensweisen. Der „naive“ Ansatz besteht darin, über alle Zahlen von 1 bis n zu iterieren und für jede einzeln zu überprüfen, ob diese n ganzzahlig teilt. Sofern dies zutrifft wird die aktuelle Zahl als größter Teiler gespeichert. Danach wird weiter über die Zahlen iteriert. Die am Ende gespeicherte Zahl entspricht dem größten Teiler.

Dieser primitive Ansatz ist in sofern ineffizient, dass auch alle Zahlen größer als $\frac{n}{2}$ geprüft werden. Diese können aber n überhaupt nicht ganzzahlig teilen. Somit würde die erste Optimierung hierbei darin bestehen, lediglich über alle Zahlen bis $\frac{n}{2}$ zu iterieren. Eine weitere Optimierung

[1] In besonderen Fällen für n stimmt dies natürlich nicht, da der Algorithmus in eine Endlosschleife laufen würde

besteht darin, nicht von 1 aufwärts zu iterieren, sondern von $\frac{n}{2}$ abwärts. All diese Optimierungen führen jedoch zu keiner Verbesserung der Komplexität O, da es sich jeweils nur um einen konstanten Faktor handelt, der sich ändert. Die Komplexität ist in allen Fällen $O(N)$

Durch die mathematischen Eigenschaften der Multiplikation kann die Komplexität jedoch auf $O(\sqrt{N})$ reduzieren. Die Eigenschaft, die man sich in diesem Fall zunutze macht, ist die *Kommutativität*. Das bedeutet im genauen:

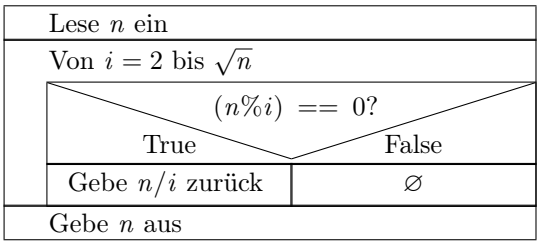
$$a = b \cdot c = c \cdot b$$

Oder kurz: Die Faktoren in der Multiplikation lassen sich vertauschen. Dadurch lässt sich die Zahl der Iterationen weiter verringern. Die höchste Zahl, die nun zu prüfen ist, ist $\sqrt{n}$, da alle danach gefundenen Teiler lediglich eine Vertauschung der Faktoren eines zuvor gefundenen Teilers repräsentieren. Beispielhaft betrachten wir die ganzzahligen Teiler von 36

- $2 \Rightarrow 2 \cdot 18 = 36$
- $3 \Rightarrow 3 \cdot 12 = 36$
- $4 \Rightarrow 4 \cdot 9 = 36$
- $6 \Rightarrow 6 \cdot 6 = 36$
- $9 \Rightarrow 9 \cdot 4 = 36 = 4 \cdot 9$
- $12 \Rightarrow 12 \cdot 3 = 36 = 3 \cdot 12$
- $18 \Rightarrow 18 \cdot 2 = 36 = 2 \cdot 18$

Zu erkennen ist, dass die nach 6 gefundenen Faktoren implizit schon einmal gefunden wurden. Das bedeutet für unseren Algorithmus, dass wir lediglich den kleinsten Teiler finden müssen. Das jeweilige Komplementär(=der andere Faktor) dazu ist unser gesuchter größter Teiler[2]. Das Komplementär lässt sich simpel über Division ermitteln.

Für den „optimalsten“ Algorithmus sieht das Struktogramm folgendermaßen aus:



Implementierungen für die verschiedenen Ansätze finden sich im beigefügten Java Projekt.

Aufgabe 3

Der in dieser Aufgabe dargestellte Algorithmus gibt von zwei Listen jedes mögliche Elementweise Paar aus(Kartesisches Produkt). Für die Komplexitätsbetrachtung für τ betrachten wir einen Funktionsaufruf als eine Operation. Dann ergeben sich für den gegebenen Codeausschnitt folgende Operationen:

[2]Hierzu habe ich zwar keine wissenschaftlich fundierte Quelle, aber das lässt sich logisch schlussfolgern

```java
public void func(List<?> in1, List<?> in2){
  for(int i=0;i<in1.size();i++){        //2 Operationen, 2(n+1) mal
      for(int j=0;in2.size();j++){      //2 Operationen, 2*(m+1)*n mal
          //4 Operationen, (m*n) mal
          String out = String.format("{\%s,\%s}", in1.get(i), in2.get(j));
          System.out.println(out);      //1 Operation, (m*n) mal
      }
  }
}
```

Damit ergibt sich $\tau(N, M)$ zu:

$$\tau(N, M) = 2 \cdot (N + 1) + 2 \cdot (M + 1) \cdot N + 4 \cdot (M \cdot N) + 1 \cdot (M \cdot N)$$

$$\tau(N, M) = 2 \cdot N + 2 + 2 \cdot M \cdot N + 2 \cdot N + 4 \cdot M \cdot N + M \cdot N$$

$$\tau(N, M) = 7 \cdot M \cdot N + 4 \cdot N + 2 = c_1 \cdot M \cdot N + c_2 \cdot N + c_3$$

Und somit $\tau(4, 2)\tau(4, 2) = 7 \cdot 2 \cdot 4 + 4 \cdot 4 + 2 = 82$

Für die Betrachtung von O zählt nur der größte dynamische Faktor, in diesem Fall gilt also $O(N \cdot M)$

Aufgabe 4

Der dargestellte Algorithmus beschreibt ein Verfahren, in dem für eine gegebenen Zahl n gegen alle Zahlen von 2 bis $n-1$ geprüft wird, ob diese n ganzzahlig teilt. Wenn dies der Fall ist, bricht der Algorithmus ab und gibt `false` zurück. Wurde über alle Zeilen iteriert, gibt der Algorithmus `true` zurück.

Zur Betrachtung von $\tau(n)$ können folgende Operationen ermittelt werden (Betrachtung, dass der Algorithmus nicht vorzeitig abgebrochen wird):

- Einlesen von $n \rightarrow 1$ Operation, Einmal ausgeführt
- Initialisieren von i auf $2 \rightarrow 1$ Operation, Einmal ausgeführt
- Prüfung $i < n \rightarrow 1$ Operation, $(n-1)$ mal(Weil für $i = 1$ nicht geprüft wird)
- Prüfung, ob i ein ganzzahliger Teiler von n ist $\rightarrow 2$ Operationen, $(n-2)$ mal ausgeführt
- Inkrementieren von $n \rightarrow 1$ Operation, $(n-2)$ mal ausgeführt

Somit ergibt sich für $\tau(n)$:

$$\tau(n) = 1 + 1 + (n-1) + 2 \cdot (n-2) + (n-2) = 5 \cdot n - 4$$

Und somit die Komplexität $O(n)$

Wie in Aufgabe 2 lässt sich dieser Algorithmus weiter optimieren, indem man die kommutativen Eigenschaften der Multiplikation nutzt. Auch hier kann die Anzahl der Schleifendurchläufe reduziert werden, indem als Abbruchbedingung nicht $i < n$ gewählt wird, sondern $i < \sqrt{n}$

Dadurch ändert sich die Komplexität zu $O(\sqrt{n}) und \tau(n)$ ergibt sich zu:

$$\tau(n) = 1 + 1 + (\sqrt{n} - 1) + 2 \cdot (\sqrt{n} - 2) + (\sqrt{n} - 2) = 5 \cdot \sqrt{n} - 4$$

Aufgabe 5

Der gezeigte Codeausschnitt iteriert über die Elemente eines Arrays und manipuliert diese. Charakteristisch ist hierbei, dass abhängig davon, ob der Wert an dem aktuellen Index des Arrays gerade ist oder nicht, eine sehr simple Operation oder eine sehr komplexe Operation ausgeführt wird.

Der Best-Case tritt hierbei auf, wenn alle Elemente des übergebenen Arrays ungerade sind, weil dann lediglich eine simple Addition auf das Arrayelement durchgeführt wird. Der Worst Case trifft analog ein, wenn alle Elemente des Arrays gerade sind. Für die Betrachtung von $\tau(n)$ lassen sich folgende Operationen ermitteln.

```
1  public void func(int[] array, int n){
2      //For Schleife wird in jedem Fall ausgef hrt
3      for(int i=0;i<n;i++){       //2 Operationen, (n+1) mal
4          //if-   berprfung    wird in jedem Fall ausgef hrt
5          if((array[i]%2)==0){    //3 Operationen, n mal
6          //Worst-Case Betrachtung: Es wird immer der if-Zweig ausgef hrt
7              array[i]+=1;        //3 Operationen, n mal
8              array[i]*=7;        //3 Operationen, n mal
9              array[i]*=array[i]; //4 Operationen, n mal
10             array[i]-=42;       //3 Operationen, n mal
11             array[0]+=array[i]; //4 Operationen, n mal
12         }else{
13             //Best-Case Betrachtung: Es wird immer der else-Zweig ausgef hrt
14             array[i]+=2;        //3 Operationen, n mal
15         }
16     }
17 }
```

Somit ergibt sich für den Best-Case ein $\tau(n)$:

$$\tau(n) = 2 \cdot (n+1) + 3 \cdot n + 3 \cdot n = 8 \cdot n + 2$$

Und für den Worst-Case ein $\tau(n)$:

$$\tau(n) = 2 \cdot (n+1) + 3 \cdot n + 3 \cdot n + 3 \cdot n + 4 \cdot n + 3 \cdot n + 4 \cdot n$$

$$\tau(n) = 22 \cdot n + 2$$

Für die Betrachtung von O gibt es in diesem Fall keine Unterscheidung, da die einzige Änderung in den konstanten Faktoren liegt. Sollte es jedoch hier eine Unterscheidung in den dynamischen Faktoren geben, so würde für die Betrachtung von O lediglich der Worst-Case herangezogen werden.

Somit ergibt sich für die Komplexität $O(n)$