Algorithmen und Datenstrukturen
Aufgabenblatt 3

Abgabe: 13.05.2021 22:00
Abnahme: 14.05.2021

1. Komplexitätsbestimmung

Entwickeln Sie die Regel, nach der sich die Laufzeit-Komplexität im Kalkül von Schleifen bestimmen lässt. Sie können sich hier auf While-Schleifen beschränken. Orientieren Sie sich dabei am Beispiel aus der Vorlesung, in dem if-Ausdrücke betrachtet wurden.

Wenden Sie Ihre Regel an auf das folgende Programmfragment:

```
eingabe(n) //integer, O(1)
i = 0
zahl = 0
while i <= n*n
 if i gerade
  zahl += i
 else
  zahl += 2*i
 i++
```

2. Komplexitätsbestimmung

Berechnen Sie die Laufzeit der Funktion *fkt*. Erweitern Sie die Regel von if-Anweisungen aus der Vorlesung auf if-else-Anweisungen und begründen Sie Ihre Lösung. Bestimmen und erklären Sie anschließend die Laufzeit des folgenden Programmfragments:

```
def fkt(i):
  j = 1
  r = 1
  while j <= i;
    r+=1
    j+=1
  return(r)

eingabe(n)
if fkt(n) > 10:
  m = fkt(n -1)
else
  m = 0
```

3. Aufwandsabschätzungen

Hier sollen Sie Regeln entwickeln zur Aufwandsabschätzung von rekursiven Funktionen im allgemeinen Fall. Dazu schätzen Sie sukzessive die Komplexität des in der Vorlesung vorgestellten rekursiven Algorithmus für die Berechnung der maximalen Teilsumme mit Hilfe der Strategie **Teile-und-herrsche** ab. Begründen Sie Ihre Antwort.

Beantworten Sie dazu zunächst folgende Fragen, die Ihnen helfen sollen, die Regeln zu finden:

Teilaufgabe 3.1: Welche Komplexität haben die Funktionen `rechtesRandmax` und `linkesRandmax`?

Teilaufgabe 3.2: Welche Komplexität hat die Funktion `maxTeilsummerek`, wenn man die Zeile mit den rekursiven Aufrufen nicht berücksichtigt?

Teilaufgabe 3.3: Welche Komplexität hat die Funktion `maxTeilsummerek`, wenn man annimmt, dass sich die Anzahl der rekursiven Aufrufe mit $f(n)$ berechnen lässt.

Teilaufgabe 3.4: Wie oft kann man eine Folge der Länge 2, der Länge 4, der Länge 8, der Länge 16, ... in zwei gleiche Hälften teilen, d. h. wie oft kann man diese Länge halbieren, bis man (in einem Ast) bei 1 angelangt ist?

Teilaufgabe 3.5: Wie oft kann man eine Folge der Länge 27, der Länge 173 oder der Länge 291 in zwei möglichst gleichlange Teilfolgen aufteilen?

Teilaufgabe 3.6: Wie oft kann man eine Folge der Länge n in zwei möglichst gleichlange Folgen aufteilen?

Teilaufgabe 3.7: Wie oft wird *maxTeilsummerek* bei einer Folgenlänge von n aufgerufen? bitte beachten: Zwei "gleichwertige" Aufrufe hintereinander bedeuten zwar doppelten Aufwand, der Faktor 2 wird aber nicht berücksichtigt bei der Worst-Case-Betrachtung, wie Sie wissen.

Teilaufgabe 3.8: Welche Komplexität hat *maxTeilsummerek*? Begründen Sie Ihre Antwort mit Hilfe der Antworten auf die vorangehenden Fragen.

Teilaufgabe 3.9: Wie berechnet man die Laufzeit im O-Kalkül für rekursive Funktionen? Begründen Sie Ihre Antwort mit Hilfe der Antworten auf die vorangehenden Fragen.

Ziel: Komplexitätsabschätzung von rekursiven Funktionen.

Aufgabe 1)

Betrachtet werden muss der Funktionsteil der am schnellsten in Abhängigkeit der Eingabedaten wächst, also meistens der "Körper" der innersten Schleife.
Von dort muss aus den Laufzeiten der Schleifen ein Produkt erstellt werden.
Bei einer Fallunterscheidung werden die Laufzeit vor der Fallunterscheidung, die Bedingung der Fallunterscheidung und der "Körper" der Fallunterscheidung betrachtet und von diesem der Maximalwert genommen.

Im Beispiel ist die Laufzeit der Schleife quadratisch abhängig von der Eingabe also $O(n^2)$, die Bedingung der Fallunterscheidung und der "Körper" sind Konstant also $O(1)$, daher ist die Laufzeit des Beispiels $Max(O(n^2), O(1), O(1)) = O(n^2)$

Aufgabe 2)

Die Regel zur Berechnung von if-Anweisungen sieht folgendermaßen aus:

```
<A>
if <condition>
    <B>
```

Betrachtet wird die Laufzeit im schlechtesten Fall, daher ist die Gesamtlaufzeit Max(O(A), O(condition), O(B)).

Im Falle einer if-else-Anweisung muss dann zusätzlich der "Körper" des else-Zweiges betrachtet werden.

```
<A>
if <condition>
    <B>
else
    <C>
```

So ist die Gesamtlaufzeit in diesem Fall Max(O(A), O(condition), O(B), O(C)).

Im Beispiel ist die Laufzeit von <A>, also der Eingabe, Konstant also O(1).
In der Bedingung wird die Funktion fkt aufgerufen, diese Funktion hat eine lineare Laufzeit also O(n).
<B> ruft ebenfalls die Funktion fkt auf, deshalb ist hier ebenfalls eine lineare Laufzeit O(n).
Die Laufzeit des else-Zweig <C> ist Konstant also O(1).

Daher ist die Gesamtlaufzeit:
Max(O(<A>), O(<condition>), O(<B>), O(<C>))
= Max(O(1), O(n), O(n), O(1))
= O(n)

Aufgabe 3)

3.1)
Beide Funktionen haben die Laufzeit O(n).

3.2)
Ohne die rekursiven Aufrufe hat die Funktion die größte Laufzeit in den Aufrufen
von rechtesRandmax und linkesRandmax (jeweils O(n)).
Daher hat die Funktion die Laufzeit O(n).

3.3)
Die Komplexität der Funktion ohne den rekursiven Aufruf beträgt O(n),
durch den rekursiven Aufruf wird diese Komplexität f(n) mal wiederholt.
Daher ist die Komplexität O(n * f(n)).

3.4)
Bei n = 2: (2 -> 1) => Einmal
Bei n = 4: (4 -> 2 -> 1) => Zweimal
Bei n = 8: (8 -> 4 -> 2 -> 1) => Dreimal
Bei n = 16: (16 -> 8 -> ...) => Viermal

3.5)
Bei n = 27: (27 -> 14 -> 7 -> 4 -> 2 -> 1) => Fünfmal
Bei n = 173: (173 -> 87 -> 44 -> 22 -> 11 -> 6 -> 3 -> 2 -> 1) => Achtmal
Bei n = 291: (291 -> 146 -> 73 -> 37 -> 19 -> 10 -> 5 -> 3 -> 2 -> 1) => Neunmal

3.6)
Betrachten wir Folge dessen Länge eine Zahl der Basis 2 ist, also 2^x,
dann lässt sich die Folge x-mal teilen. Bei einer Länge die keine Zahl der Basis 2 ist
kann man diese Folge so lange teilen wie es bei einer Folge, die als Länge die nächstgrößere
Zahl zur Basis 2 hat.

3.7)
Bei den Rekursiven Aufrufen wird jedes mal die Datenmenge halbiert.
Ist also beim ersten Aufruf z.B. n = 8 (n = links - rechts),
ist beim zweiten Aufruf n = 4,
beim dritten Aufruf n = 2
und der vierte Aufruf wäre dann (n=1), dieser bricht aber an dem "Anker" ab.
Daher gilt hier die Funktion f(n) = ld(n) (Logarithmus zur Basis 2) aufgerundet.

3.8)
Die Komplexität beträgt damit O(n * log(n))

3.9)
Die Komplexität einer rekursiven Funktion lässt sich folgendermaßen verallgemeinern:
O(A): Komplexität der Funktion ohne die rekursiven Aufrufe
f(n): Funktion zur Berechnung der Anzahl der rekursiven Aufrufe
dann ist die Gesamtkomplexität:
O(O(A) * f(n))