Prof. Dr. Stefan Göller
Florian Bruse
Dr. Norbert Hundeshagen

Einführung in die Informatik

WS 2018/2019

Übungsblatt 7
6.12.2018-13.12.2018

Abgabe: Bis zum 13.12. 18:00 Uhr über moodle. Reichen Sie pro Aufgabe, die Sie bearbeitet haben, die vorgebenene Rumpfdatei ein. Verändern Sie diese Datei nicht weiter, als angegeben. Andere Dateien oder unzulässig veränderte Rumpfdateien werden im Zweifelsfall nicht korrigiert.

Aufgabe 1 (Laufzeit von Programmen) (5+5+5+5 Punkte):

Geben Sie jeweils die best-case-Laufzeit (also die am schwächsten wachsende Funktion, f, so dass das Programm in $\mathcal{O}(f)$ läuft) sowie die worst-case-Laufzeit der folgenden Programme in Abhängigkeit von der Eingabe $n \in \mathbb{N}$, bzw. der Länge $n \in \mathbb{N}$ der Eingabe an. Geben Sie auch an, welche Funktion das Programm genau berechnet. Begründen Sie Ihre Angaben wie im folgenden Beispiel:

```
def programm0(n):
        if n % 2 == 0:
                return -1
        j = 0
        for i in range(n):
                j += i
        return j
```

Der best case für dieses Programm sind Eingaben von geraden Zahlen. In diesem Fall benötigt das Programm eine konstante Anzahl von Operationen, denn es werden nur die ersten beiden Zeilen je einmal ausgeführt. Der best case ist also in $\mathcal{O}(1)$. Der worst case

für das Programm sind ungerade Zahlen, die Laufzeit ist dann in $\mathcal{O}(n)$. In diesem Fall werden die beiden ersten Zeilen einmal, und die Zeile in der Schleife n mal ausgeführt. Das Programm berechnet die Funktion

$$f(n) = \begin{cases} -1 & \text{falls } n \text{ gerade} \\ \Sigma_{i=0}^{n-1} i = \frac{1}{2}(n-1)n & \text{sonst} \end{cases}$$

Gehen Sie davon aus, dass eine einzelne Ausführung jeder Zeile eine konstante Anzahl von Operationen benötigt.

a)
```
def programm1(n):  # Eingabegroesse: n
        if n < 2:
                return False
        divisor = 2
        while divisor * 2 < n:
                divisor += 1
                if n % divisor == 0:
                        return False
        return True
```

b)
```
def programm2(n):  # Eingabegroesse: n
        i = 0
        while i <= n:
                i += 1
                if i ** 2 > 0:
                         break
        return i
```

c)
```
def programm3(L):  # Eingabegroesse: len(L)
          for i in range(len(L)):
                sortiert = True
                for j in range(1,len(L)-i):
                        if L[j] < L[j-1]:
                                L[j], L[j-1] = L[j-1], L[j]
                                sortiert = False
                if sortiert == True:
                      break
          return L
```

d)
```
def programm4(n):  # Eingabegroesse: n
        i = 1
        for j in range(n):
                i *= 2
        j = 1
        for k in range(i):
                k *= 2
        return k
```

Aufgabe 2 (Programmoptimierung) (5+5+5 Punkte):

Gegeben ist das folgende Programm, wobei `maxmult(L)` die Hauptfunktion ist:

```
def max(L):
        if len(L) == 0:
                return 0

        a  = max(L[1:])
        return a if a > L[0] else L[0]


def maxmult(L):
        if len(L) < 2:
                return 0

        a = maxmult(L[1:])
        b = max(L[1:]) * L[0]
        return a if a > b else b
```

Bei Eingabe einer Liste aus positiven Integern L gibt `maxmult`(L) die größte Zahl zurück, welche sich durch Multiplikation zweier Zahlen die an verschiedenen Positionen in L stehen, ergeben kann, falls die Liste mindestens Länge 2 hat. Ansonsten wird 0 zurückgegeben.

Beispiel: Bei Eingabe `[2,6,4]` gibt `maxmult` die Zahl `24` zurück und bei Eingabe `[2,6,8,1,8]` gibt `maxmult` die Zahl `64` zurück.

a) Beschreiben Sie kurz, wie das Programm diese Zahl rekursiv berechnet.

b) Geben Sie die Laufzeit des Programms mit Begründung an.

c) Schreiben Sie eine Funktion `maxmult_lin(L)`, welche die gleiche Funktion wie `maxmult(L)` berechnet, aber nur lineare Laufzeit benötigt.

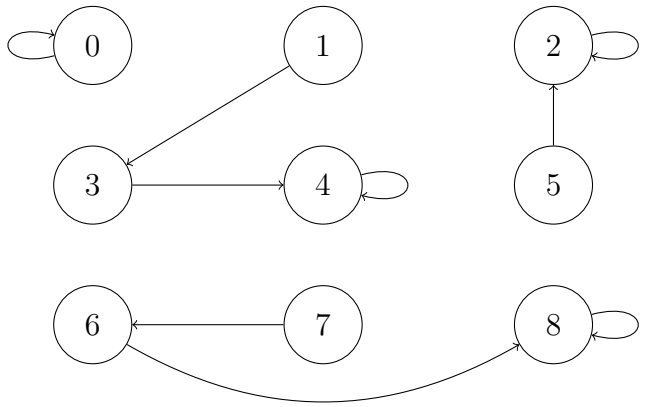
Aufgabe 3 (Wörterbücher als Graphen) (5+5+5+5+5 Punkte):

Wir betrachten Wörterbücher, deren Schlüsselmenge und Wertebereich beide die Menge $\{0, \ldots, n\}$ für ein $n \in \mathbb{N}$ sind. So ein Wörterbuch kann man sich als Kodierung eines Graphen vorstellen. Wenn die Schlüsselmenge des Wörterbuchs die Menge $\{0, \ldots, n\}$ ist, dann hat der Graph die Knoten $\{0, \ldots, n\}$. Der (eindeutige) Nachfolger eines Knotens ist der Knoten, der sich ergibt, wenn man im Wörterbuch den Wert sucht, der diesem Knoten zugeordnet ist.

Als Beispiel betrachten wir das folgende Wörterbuch:

Schlüssel	0	1	2	3	4	5	6	7	8
Wert	0	3	2	4	4	2	8	6	8

Bildlich kann man sich den zugehörigen Graphen so vorstellen:



Wie man sieht, kann man jedem Knoten einen *Pfad* zuordnen, also die Folge der Knoten, die man erhält, wenn man von diesem Knoten aus die Nachfolger abläuft, bis man an einer Schlaufe ankommt. Ein Knoten hat eine Schlaufe, wenn er sein eigener Nachfolger ist (so ein Knoten bildet dann einen Kreis der Länge 1). Dem Knoten 1 ist beispielsweise der Pfad $1 \to 3 \to 4$ zugeordnet, dem Knoten 0 der Pfad 0, und dem Knoten 3 der Pfad $3 \to 4$.

Für die folgenden Python-Funktionen geben Sie bitte jeweils als Kommentar auch die worst-case-Laufzeit in Abhängigkeit von den angegebenen Parametern an. Sie können davon ausgehen, dass ein übergebener Pfad kein "Lasso" (also keinen Kreis der Länge ≥ 2, etwa $1 \to 2 \to 3 \to 2 \to 3 \to \cdots$, enthält.) ist. Sie brauchen auch nicht zu testen, ob ein übergebener Knoten im Wörterbuch enthalten ist.

a) Schreiben Sie eine Python-Funktion `hat_schlaufe(buch, i)`, welche ein Wörterbuch wie oben beschrieben und einen Knoten i entgegennimmt, und testet, ob dieser eine Schlaufe hat, d.h., ob er sein eigener Nachfolger ist. In diesem Falls soll `True` zurückgegeben werden, falls nicht, soll `False` zurückgegeben werden. Die Eingabegröße ist die Größe n des Wörterbuchs `buch`.

b) Schreiben Sie eine Python-Funktion `laenge(buch, i)`, welche ein Wörterbuch wie oben beschrieben und einen Pfad (als Startknoten i) entgegennimmt. Ihre Funktion soll nun die Länge des von Knoten i ausgehenden Pfades berechnen. Beispielsweise würde `laenge(buch, 0)` für den obigen Graphen den Wert 1 zurückliefern, während `laenge(buch, 6)` den Wert 2 liefert. Die Eingabegröße ist die Länge n des von Knoten i ausgehenden Pfades.

c) Schreiben Sie eine Python-Funktion `eintrag(buch, i, j)`, welche ein Wörterbuch wie oben beschrieben und einen Pfad (als Startknoten i) und eine natürliche Zahl j entgegennimmt. Ihre Funktion soll nun den `j`-ten Knoten auf dem durch i kodierten Pfad zurückgeben, oder -1, falls der Pfad nicht lang genug ist. Im obigen Graph würde `eintrag(buch, 1, 1)` beispielsweise 3 zurück geben, während `eintrag(buch, 5, 2)` den Wert -1 zurück gibt. Die Eingabegröße ist die Länge n des von Knoten i ausgehenden Pfades.

d) Schreiben Sie eine Python-Funktion `verlaengere(buch, i, j)`, welche ein Wörterbuch wie oben beschrieben und einen Pfad (als Startknoten i) und einen Knoten j entgegennimmt. Ihre Funktion soll nun prüfen, ob j bereits im von Knoten i beginnenden Pfad vorkommt. Falls ja, soll `False` zurück gegeben werden, falls nein, soll der Knoten j an den von i ausgehenden Pfad angehängt werden, d.h. der letzte Knoten im vom Knoten i beginnenden Pfad soll nun den Nachfolger j haben. In diesem Fall gibt Ihre Funktion `True` zurück, und das Wörterbuch ändert sich als Nebeneffekt Ihres Programms. Die Eingabegrößen sind die Längen n und m der von den Knoten i und j ausgehenden Pfade.

e) Schreiben Sie eine Python-Funktion `disjunkt(buch, i, j)`, welche ein Wörterbuch wie oben beschrieben und Knoten i und j entgegennimmt, und testet, ob die von den beiden Knoten ausgehenden Pfade disjunkt sind, d.h., ob kein Knoten auf beiden Pfaden vorkommt. Das Ergebnis soll als `True` oder `False` zurückgegeben werden. Beispielsweise liefert `disjunkt(buch, 6, 7)` den Wert `False`, während `disjunkt(buch, 5, 8)` den Wert `True` liefert. Die Eingabegrößen sind die Längen n und m der von den Knoten i und j ausgehenden Pfade.