ETH zürich
Departement Informatik

Programmieren und Problemlösen

Dr. Dennis Komm
https://lec.inf.ethz.ch/ppl/
19. April 2021

Übungsblatt 3
Lösungsvorschläge

Aufgabe 1

Geben Sie die kleinste Klasse in Gross-$\mathcal{O}$-Notation an, zu der die folgenden Funktionen gehören. Mit „klein“ ist hier gemeint, dass beispielsweise $3 \cdot n^2$ in $\mathcal{O}(n^2)$, $\mathcal{O}(n^3)$ oder in $\mathcal{O}(2^n)$ ist. In diesem Fall wäre die Antwort „$3 \cdot n^2 \in \mathcal{O}(n^2)$“.

Sortieren Sie die Funktionen anschliessend ihrer Grösse nach.

Tipp: Die nachfolgende Tabelle rechts enthält einige Klassen der Gross-$\mathcal{O}$-Notation in aufsteigender Komplexität. Eine umfassendere Erklärung kann auf Wikipedia gefunden werden:

https://en.wikipedia.org/wiki/Big_O_notation.

(a) $100 \cdot n + 200 \cdot n + \sqrt{5 \cdot n} + \log n$

(b) $2 \cdot n^3 + n^3 \cdot n^2 + n^4 + n^2 \cdot \log n$

(c) $\log(\log n) + 100 \cdot \log n + \sin(n)$

(d) $n \cdot \log n^3 + 2 \cdot n + 3 \cdot \log n$

(e) 3^{n-4}

(f) $n^{1/2} + n^{1/3}$

(g) $\log^2 n + 100 + 5 \cdot 20$

Gross-$\mathcal{O}$-Notation	Beispiele
$\mathcal{O}(1)$	5, 200, 2^{100}
$\mathcal{O}(\log(\log n))$	$\log(\log n) + 5$
$\mathcal{O}(\log n)$	$\log n + \log^2 n + 100$
$\mathcal{O}(\sqrt{n})$	$5 \cdot \sqrt{n} + 100$
$\mathcal{O}(n)$	$4 \cdot n + 100$
$\mathcal{O}(n \log n)$	$n \cdot \log n^5$
$\mathcal{O}(n^2)$	$n^2 + 100 \cdot n$
$\mathcal{O}(n^7)$	$n^7 + 10 \cdot n^5 + 2 \cdot n^3$
$\mathcal{O}(2^n)$	$2^n + n^{1000}$
$\mathcal{O}(5^n)$	$5^n + n^{1000}$
$\mathcal{O}(n!)$	$9^n + n!$

Lösung

Wir erhalten folgende Zuordnungen.

(a) $100 \cdot n + 200 \cdot n + \sqrt{5 \cdot n} + \log n \in \mathcal{O}(n)$

(b) $2 \cdot n^3 + n^3 \cdot n^2 + n^4 + n^2 \cdot \log n = n^5 + n^4 + n^3 \cdot 2 + n^2 \cdot \log n \in \mathcal{O}(n^5)$

(c) $\log(\log n) + 100 \cdot \log n + \sin(n) \leq \log(\log n) + 100 \cdot \log n + 1 \in \mathcal{O}(\log n)$

(d) $n \cdot \log n^3 + 2 \cdot n + 3 \cdot \log n = 3 \cdot n \cdot \log n + 2 \cdot n + 3 \log n \in \mathcal{O}(n \cdot \log n)$

(e) $3^{n-4} = 3^n / 3^4 \in \mathcal{O}(3^n)$

(f) $n^{1/2} + n^{1/3} \in \mathcal{O}(\sqrt{n})$

(g) $\log(\log n) + 100 + 5 \cdot 20 \in \mathcal{O}(\log(\log n))$

Daraus resultiert die Reihenfolge (g), (c), (f), (a), (d), (b) und (e).

Aufgabe 2

Bestimmen Sie die asymptotische Laufzeit der folgenden Funktion in Abhängingkeit von n. Diese Funktion implementiert die Multiplikation von zwei Matrizen der Grösse $n \times n$. Bitte beachten Sie, dass die Grösse der Eingabe dieser Funktion eigentlich $2 \cdot n^2$ entspricht. In diesem Kontext ist es allerdings üblich, die Laufzeit in Abhängigkeit von n zu bestimmen.

```
1  def multiply(A, B):
2      # Erinnerung: A ist eine Liste von Listen.
3      # len(A) gibt die Anzahl Listen in der Liste A zurueck.
4      # Da A nach Voraussetzung quadratisch ist,
5      # entspricht n genau len(A).
6      n = len(A)
7
8      # C initialisieren
9      C = []
10     for i in range(0, n):
11         tmp = [0] * n
12         C.append(tmp)
13
14     for i in range(0, n):
15         for j in range (0, n):
16             # calculate C[i][j]
17             tmp = 0
18             for k in range(0, n):
19                 tmp += A[i][k] * B[k][j]
20             C[i][j] = tmp
21     return C
```

Lösung

Wir fügen dem Code Kommentare hinzu.

```
1  def multiply(A, B):
2      # Erinnerung: A ist eine Liste von Listen.
3      # len(A) gibt die Anzahl Listen in der Liste A zurueck.
4      # Da A nach Voraussetzung quadratisch ist,
5      # entspricht n genau len(A).
6      n = len(A)
7
8      # C zu initialisieren dauert n^2 Schritte
```

```
    C = []
    for i in range(0, n):
        tmp = [0] * n
        C.append(tmp)

    # Die folgende Schleife wird n-mal ausgefuehrt
    for i in range(0, n):
        # Die folgende Schleife wird fuer jedes i n-mal ausgefuehrt
        for j in range (0, n):
            # Berechne C[i][j]
            tmp = 0
            # Die folgende Schleifen wird wiederum wird fuer jede
            # Kombination von i und j n-mal ausgefuehrt
            for k in range(0, n):
                tmp += A[i][k] * B[k][j]
            C[i][j] = tmp
    return C
```

Die Laufzeit von `multiply(A, B, C, n)` ist demnach in $\mathcal{O}(n^3)$.

Aufgabe 3

Bestimmen Sie die asymptotische Laufzeit der folgenden Funktion. Die Eingabe der Funktion ist eine Liste der Länge n. Für die Elemente in der Liste kann eine konstante Grösse angenommen werden, womit n als Eingabegrösse betrachtet werden kann.

```
def get_largest_element(l):
    tmp = l[0]
    for number in l:
        if tmp < number:
            tmp = number
    return tmp
```

Lösung

Wir fügen dem Code Kommentare hinzu.

```
def get_largest_element(l):
    tmp = l[0]
    # Die folgende Schleife wird n-mal ausgefuehrt
    for number in l:
        if tmp < number:
            tmp = number
    return tmp
```

Die Laufzeit von `get_largest_element(l)` ist in $\mathcal{O}(n)$.

Aufgabe 4

Bestimmen Sie die asymptotische Laufzeit der folgenden Funktion. Die Eingabe der Funktion ist wieder eine Liste der Länge n. Für die Elemente in der Liste kann auch hier

eine konstante Grösse angenommen werden, womit n wieder als Eingabegrösse betrachtet werden kann.

```
def descending_sort(l):
    for i in range(0, len(l)):
        for j in range(0, len(l)-1):
            if l[j] < l[j+1]:
                tmp = l[j]
                l[j] = l[j+1]
                l[j+1] = tmp
    return l
```

Lösung

Wir fügen dem Code Kommentare hinzu.

```
def descending_sort(l):
    # Die folgende Schleife wird n-mal ausgefuehrt
    for i in range(0, len(l)):
        # Die folgende Schleife wird fuer jedes i n-mal ausgefuehrt
        for j in range(0, len(l)-1):
            if l[j] < l[j+1]:
                tmp = l[j]
                l[j] = l[j+1]
                l[j+1] = tmp
    return l
```

Die Laufzeit von `descending_sort(l)` ist in $\mathcal{O}(n^2)$.

Aufgabe 5

Kopieren Sie das untenstehende Programm in eine Sandbox in *Code-Expert* und führen Sie es aus. Haben Sie Geduld, dies kann einige Zeit dauern. Stellen Sie eine Theorie auf, wieso diese Funktion so langsam ist.

Bei dieser Aufgabe geht es nicht um die Gross-$\mathcal{O}$-Notation, sondern um die tatsächliche Zeit, die das Programm für die konkrete gegebene Eingabe braucht.

Tipp 1: `expensive_function(l)` erhält eine Liste, quadriert alle Zahlen darin und addiert sie dann zusammen. Anschliessend prüft die Funktion, ob das Resultat eine Primzahl ist. Falls das Resultat eine Primzahl ist, wird es zurückgegeben, sonst gibt die Funktion -1 zurück.

Tipp 2: `func(l)` nimmt eine Liste `l` und zählt wie oft das erste Element addiert werden kann, bis das Resultat grösser ist als der Rückgabewert von `expensive_function(l)`.

```
l = [2, 5, 6, 7, 10, 200, 17, 1]

def expensive_function(l):
    # Quadriert alle Zahlen in l und addiert sie zusammen
    square_sum = 0
    for i in l:
        square_sum += i * i
```

```
8     # Findet heraus, ob diese Zahl eine Primzahl ist
9     is_prime = True
10    for i in range(2, square_sum):
11        if square_sum % i == 0:
12            is_prime = False
13    # Falls square_sum eine Primzahl ist, gib sie zurueck
14    if is_prime:
15        return square_sum
16    # Sonst gib 10000 zurueck
17    else:
18        return 10000
19
20 def func(l):
21    s = l[0]
22    i = 0
23    while s < expensive_function(l):
24        s += l[0]
25        i += 1
26    print(i)
27
28 func(l)
```

Lösung

Das Programm ist so langsam, weil die `while`-Schleife in Zeile 23 in jeder Runde die Funktion `expensive_function(l)` aufruft. Ersetzt man die Funktion `func(l)` durch die untenstehende Implementierung, wird zusätzlich noch die Zeit gemessen und ausgegeben. Damit kann beobachtet werden, wie lange die Funktion tatsächlich benötigt.

```
1  # Das Modul time wird importiert, um dessen Funktionen nutzen zu
2  # koennen. Wir tun dies immer ganz zu Anfang des Codes, um die
3  # UEbersicht zu behalten
4  import time
5
6  def func(l):
7     # Die aktuelle Zeit auslesen
8     t1 = time.time()
9
10    # Berechnungen durchfuehren
11    s = l[0]
12    i = 0
13    while s < expensive_function(l):
14        s += l[0]
15        i += 1
16    print(i)
17
18    # Die neue Zeit auslesen
19    t2 = time.time()
20    # Die Differnez ausgeben
21    print(t2 - t1)
```

Aufgabe 6

Vergleichen Sie das Programm aus dieser Aufgabe mit dem aus Aufgabe 5. Was ist anders? Wie könnte sich diese Veränderung auf die Laufzeit des Programms auswirken?

Kopieren Sie das untenstehende Programm in eine Sandbox in *Code-Expert* und führen Sie es aus.

```
l = [2, 5, 6, 7, 10, 200, 17, 1]

def expensive_function(l):
    # Quadriert alle Zahlen in l und addiert sie zusammen
    square_sum = 0
    for i in l:
        square_sum += i * i
    # Findet heraus, ob diese Zahl ein Primzahl ist
    is_prime = True
    for i in range(2, square_sum):
        if square_sum % i == 0:
            is_prime = False
    # Falls square_sum eine Primzahl ist, gib sie zurueck
    if is_prime:
        return square_sum
    # Sonst gib 10000 zurueck
    else:
        return 10000

def fast_func(l):
    limit = expensive_function(l)
    s = l[0]
    i = 0
    while s < limit:
        s += l[0]
        i += 1
    print(i)

fast_func(l)
```

Lösung

Sie werden festgestellt haben, dass dieses Programm viel schneller läuft als das aus Aufgabe 5. Der Grund dafür ist, dass nur ein einziges Mal `expensive_function(l)` aufgerufen wird. Anschliessend wird immer der Wert verwendet, der am Anfang berechnet wurde. Dies ist hier sinnvoll, da sich die Liste nicht verändert. Würde die Liste sich unter gewissen Bedingungen ändern, dürfte man diese Optimierung nicht machen.

Mit der untenstehenden Funktion können Sie noch die Zeit ansehen, die für die Ausführung der Funktion gebraucht wird.

```
# Das Modul time wird importiert, um dessen Funktionen nutzen zu
# koennen. Wir tun dies immer ganz zu Anfang des Codes, um die
# UEbersicht zu behalten
```

```python
import time

def fast_func(l):
    # Die aktuelle Zeit auslesen
    t1 = time.time()

    # Berechnungen durchfuehren
    limit = expensive_function(l)
    s = l[0]
    i = 0
    while s < limit:
        s += l[0]
        i += 1
    print(i)

    # Die neue Zeit auslesen
    t2 = time.time()
    # Die Differenez ausgeben
    print(t2 - t1)
```

Aufgabe 7

Die Funktion `print_ind_div_by_3(D)` soll eine Matrix `D` erhalten und alle Elemente von `D` ausgeben, bei welchen der 2. Index durch 3 teilbar ist, es werden also alle `D[i][j]` ausgegeben, für die `j` durch 3 teilbar ist.

```python
D = [ [1, 2, 3, 4, 5],
      [6, 7, 8, 9, 10],
      [1, 2, 3, 4, 5],
      [6, 7, 8, 9, 10],
      [1, 2, 3, 4, 5],
      [6, 7, 8, 9, 10] ]

def print_ind_div_by_3(D):
    for row in D:
        for j in row:
            if j % 3 == 0:
                print(row[j], end = " ")
        print()

print_ind_div_by_3(D)
```

Leider ist dabei ein Fehler passiert. Finden Sie den Fehler und korrigieren Sie ihn, so dass die Funktion `print_ind_div_by_3(D)` das Folgende ausgibt.

```
1 4
6 9
1 4
6 9
1 4
6 9
```

Lösung

Das `for row in D` in Zeile 2 bedeutet, dass `row` einmal jeder Zeile in `D` entspricht. Um nun den 2. Index zu überprüfen, braucht man eine Laufvariable, mit der man durch diese Reihe gehen kann. Diese bekommt man mit `for j in range(0, len(row))`.

In der fehlerhaften Funktion wird stattdessen `for j in row` verwendet, was bedeutet, dass `j` einmal den Wert jedes Elements in `row` annimmt, was zu einer falschen Ausgabe führt.

```
1 def print_ind_div_by_3(D):
2     for row in D:
3         for j in range(0, len(row)):
4             if j % 3 == 0:
5                 print(row[j], end = " ")
6         print()
```