



---

Wintersemester 2017/2018

---

## Lineare und kombinatorische Optimierung

### Übungsblatt 2

#### Gruppenübungen

##### Aufgabe 1. (Knotengrad)

(0 Punkte)

1. Ein Graph ist  $k$ -regulär, wenn alle seine Knoten Grad  $k$  haben. Es sei  $G$  ein  $k$ -regulärer Graph, wobei  $k$  eine ungerade Zahl ist. Man beweise, dass die Anzahl der Kanten in  $G$  ein Vielfaches von  $k$  ist.
2. Es sei  $G$  ein Graph mit  $n$  Knoten und  $e$  Kanten, und es sei  $m$  die kleinste positive Zahl, so dass  $m \geq 2e/n$ . Es ist zu beweisen, dass  $G$  einen Knoten enthält, der mindestens den Grad  $m$  hat.

##### Lösung:

1. Sei  $n := |V(G)|$  und  $e := |E(G)|$ . Es ist  $\deg(v) = k$  für alle  $v \in V(G)$ . Dann ist

$$2e = \sum_{v \in V(G)} \deg(v) = k \cdot n.$$

Da  $2e$  gerade ist und  $k$  ungerade, muss  $n$  auch gerade sein. Also ist 2 ein Teiler von  $n$  und es gilt  $e = k \cdot n/2 \in \mathbb{N}$ , d.h.  $n/2$  ist das gesuchte Vielfache von  $k$ .

2. Angenommen, für alle Knoten  $v \in V(G)$  ist  $\deg(v) < m$ , also  $\deg(v) \leq m - 1$ . Dann gilt

$$\sum_{v \in V(G)} \deg(v) \leq n \cdot (m - 1).$$

Andererseits gilt

$$\sum_{v \in V(G)} \deg(v) = 2e.$$

Also ist  $2e/n \leq m - 1$ .

1. Fall,  $n$  teilt  $2e$ . Dann ist  $m = 2e/n$ , und es folgt durch Einsetzen:  $m \leq m - 1$ , bzw.  $0 \leq -1$ , also ein Widerspruch.
2. Fall,  $n$  teilt  $2e$  nicht. Dann ist  $m = \lfloor 2e/n \rfloor + 1$ , und es folgt durch Einsetzen:  $2e/n \leq \lfloor 2e/n \rfloor + 1 - 1$ , also  $2e/n \leq \lfloor 2e/n \rfloor$ , also auch hier ein Widerspruch.

**Aufgabe 2.** (Pfade und Wege)

(0 Punkte)

Es sei  $G = (V, E)$  ein einfacher ungerichteter Graph. Beweisen Sie die folgenden Aussagen:

1. Jeder  $[u, v]$ -Pfad enthält einen  $[u, v]$ -Weg. Ein kürzester (bzgl. einer Gewichtung  $c_e > 0, e \in E$ )  $[u, v]$ -Pfad ist ein  $[u, v]$ -Weg.
2. Ist  $\deg(v) \geq \left\lfloor \frac{|V|}{2} \right\rfloor$  für alle  $v \in V$ , dann existiert zwischen je zwei Knoten ein Weg, der aus höchstens 2 Kanten besteht.

**Lösung:**

1. Zunächst sei nochmal kurz der Unterschied zwischen Pfad und Weg erläutert: In einem Pfad darf jede Kante nur einmal enthalten sein, Knoten dürfen jedoch mehrmals enthalten sein. In einem Weg darf jeder Knoten nur einmal enthalten sein, damit können auch Kanten nicht mehrmals auftreten.

Wenn der  $[u, v]$ -Pfad  $P$  keinen Kreis enthält, tritt jeder Knoten im Pfad nur einmal auf und  $P$  ist ein Weg.

Wenn  $P$  einen Kreis enthält, so hat  $P$  z.B. die folgende Form:  $P = [u, v_1, \dots, v_k, v_{k+1}, \dots, v_k, v_l, \dots, v]$ . Entfernt man den Kreis  $[v_k, v_{k+1}, \dots, v_k]$ , so erhält man  $P' = [u, v_1, \dots, v_k, v_l, \dots, v]$ . Eliminiert man auf diese Weise alle Kreise aus  $P$ , so erhält man einen  $[u, v]$ -Weg  $W$  in  $P$ .

Der in  $P$  enthaltene Weg  $W$  ist kürzer als  $P$  (da  $c_e > 0$ ), also ist  $W$  ein kürzester  $[u, v]$ -Pfad und ein  $[u, v]$ -Weg.

2. Angenommen, es existieren Knoten  $u, v$ , sodass es keinen  $[u, v]$ -Weg über höchstens zwei Kanten gibt. Da  $G$  einfach ist, hat  $u$  nach Voraussetzung insgesamt mindestens  $\left\lfloor \frac{|V|}{2} \right\rfloor$  Nachbarn, die alle verschieden von  $u$  sind und die auch paarweise verschieden sind. Ebenso hat  $v$  mindestens  $\left\lfloor \frac{|V|}{2} \right\rfloor$  Nachbarn, die auch verschieden sind von den Nachbarn von  $u$ , weil es sonst einen  $[u, v]$ -Weg der Länge zwei in  $G$  gäbe. Insgesamt haben wir  $\left\lfloor \frac{|V|}{2} \right\rfloor + \left\lfloor \frac{|V|}{2} \right\rfloor + 2 \geq |V| - 1 + 2 = |V| + 1$  verschiedene Knoten, ein Widerspruch.

**Aufgabe 3.** (Laufzeit von Algorithmen)

(0 Punkte)

1. Zeigen Sie: Seien  $f : \mathbb{N} \rightarrow \mathbb{R}$  und  $g : \mathbb{N} \rightarrow \mathbb{R}$  zwei Funktionen mit

$$f(n) = 8n^3 + n^2 + 76n \quad \text{und} \quad g(n) = n^3.$$

Dann gilt  $f(n) \in O(g(n))$ .

2. Berechne die Laufzeiten folgender Anweisungen in Abhängigkeit von  $n$ . Eine elementare Rechenoperationen (z. B. Addition oder auch ein Vergleich) habe dabei konstante Laufzeit.

---

For-Schleife

---

```
for  $i = 1, \dots, n$  do
   $m = m + 1$ 
```

---



---

Geschachtelte For-Schleifen

---

```
for  $i = 1, \dots, n$  do
  for  $j = 1, \dots, n$  do
     $m = m + 1$ 
```

---

---

**If-Anweisungen**

---

```
if  $n \leq 100$  then
     $m = m + 100$ 
else
     $m = m - 1$ 
```

---

---

**Aneinanderreihung von Anweisungen**

---

```
 $x = x - 20$ 
for  $i = 1, \dots, n$  do
     $m = m + 10$ 
for  $i = 1, \dots, n$  do
    for  $j = 1, \dots, n - 1$  do
         $m = m + 1$ 
         $x = x + 1$ 
```

---

**Lösung:**

1.  $f(n) \in O(g(n))$  genau dann, wenn es positive Konstanten  $c$  und  $n_0$  gibt, so dass

$$|f(n)| \leq c|g(n)| \quad \forall n \geq n_0.$$

Dies in der Tat der Fall, da

$$f(n) = 8n^3 + n^2 + 76n \leq 8n^3 + n^3 + 76n^3 = 85n^3 \quad \forall n \geq 1$$

2. For-Schleife:  $n$  Durchläufe mit jeweils konstanter Laufzeit, also  $c \cdot n \in O(n)$ , also lineare Laufzeit.  
Geschachtelte For-Schleifen:  $n$  äußere und  $n$  innere Durchläufe, also  $c \cdot n^2 \in O(n^2)$ , also quadratische Laufzeit.  
If-Abfrage: Sowohl der If-Teil als auch der Else-Teil sind konstant, also  $c \cdot 2 \in O(1)$ .  
Aneinanderreihung von Anweisungen:  $(c_0 + n \cdot c_1 + n(n-1) \cdot 2c_3) \in O(n^2)$ , also quadratische Laufzeit.

**Hausübungen****Aufgabe 4.** (Bipartite Graphen)

(4 Punkte)

Zeigen Sie: Ein Graph  $G = (V, E)$  ist bipartit genau dann, wenn er keine Kreise ungerader Länge enthält.

**Lösung:**

( $\Rightarrow$ ): Sei  $G$  bipartit mit Knotenpartition  $V = S \dot{\cup} T$ . Sei  $C = (v_1, v_2, \dots, v_n, v_1)$  ein geschlossener Pfad in  $G$ , und sei o.B.d.A.  $v_1 \in S$ . Weil es innerhalb von  $S$  bzw.  $T$  keine Kanten gibt, muss  $v_{2k} \in T$  und  $v_{2k+1} \in S$  für  $k = 1, \dots, \lfloor \frac{n}{2} \rfloor$  gelten. Um  $v_1 \in S$  zu erhalten, muss  $v_n \in T$  gelten, also muss  $n$  gerade sein. Folglich kann  $G$  keine Kreise ungerader Länge enthalten.

( $\Leftarrow$ ): Sei  $G$  ein Graph, welcher keinen ungeraden Kreis enthält. Wir können o.B.d.A. annehmen, dass  $G$  zusammenhängend ist. Für einen Knoten  $x_0 \in V$  sei  $S$  die Menge aller Knoten  $x$  mit geradem Abstand von  $x_0$  und  $T = V \setminus S$ . Angenommen, es gäbe eine Kante  $\{x, y\} \in E$ , so dass  $x, y \in S$ . Seien  $W_x$  und  $W_y$  kürzeste Wege von  $x_0$  nach  $x$  bzw.  $y$ . Aufgrund der Definition von  $S$  haben  $W_x$  und  $W_y$  gerade Länge. Sei ferner  $z$  der letzte gemeinsame Knoten von  $W_x$  und  $W_y$  (welche beide in  $x_0$  beginnen), und bezeichne die verbleibenden Wege, welche von  $z$  nach  $x$  bzw.  $y$  führen, mit  $W'_x$  bzw.  $W'_y$ . Dann ist der Weg  $x - W'_x \rightarrow z - W'_y \rightarrow y - \{x, y\} \rightarrow x$  ein Kreis ungerader Länge in  $G$ , da entweder  $W'_x$  und  $W'_y$  beide gerade Länge oder beide ungerade Länge haben. Widerspruch zur Annahme. Analog lässt sich zeigen, dass  $G$  keine Kante  $\{x, y\}$  mit  $x, y \in T$  enthält. Somit ist  $V = S \dot{\cup} T$  eine Partition von  $V$ , wobei es keine Kanten innerhalb von  $S$  und  $T$  gibt, und folglich ist  $G$  bipartit.

**Aufgabe 5.** (Eulerformel)

(4+2 Punkte)

1. Sei  $G$  ein zusammenhängender planarer Graph. Ferner seien  $n, e, f$  die Anzahl der Knoten, Kanten bzw. Flächen von  $G$ . Zeigen Sie mittels vollständiger Induktion über die Anzahl der Kanten, dass

$$n - e + f = 2$$

gilt.

2. Zeigen Sie mithilfe von Eulers Formel, dass  $K_{3,3}$  nicht planar ist.

### Lösung:

1. *Induktionsanfang:* Sei  $e = 0$ . Da  $G$  zusammenhängend ist, gibt es genau einen Knoten,  $n = 1$ . Damit gibt es genau eine Fläche, die äußere Fläche, also  $f = 1$ . Es gilt  $n - e + f = 1 - 0 + 1 = 2$ .

*Induktionsannahme:* Eulers Formel gelte für zusammenhängende planare Graphen mit  $e$  Kanten.

*Induktionsschritt:* Wir fügen eine weitere Kante  $k$  hinzu, so dass der neue Graph  $G'$  planar ist. Seien  $n', e', f'$  die Anzahl der Knoten, Kanten bzw. Flächen von  $G'$ .

Es ist  $e' = e + 1$ . Wir unterscheiden zwei Fälle:

Fall 1:  $k$  verbindet zwei Knoten von  $G$ .

Dann wird eine Fläche von  $G$  in zwei Flächen aufgeteilt,  $f' = f + 1$ .

Die Anzahl der Knoten bleibt gleich,  $n' = n$ .

Also gilt  $n' - e' + f' = n - (e + 1) + (f + 1) = n - e + f = 2$ .

Fall 2:  $k$  ist nur mit einem Knoten inzident.

Dann muss ein weiterer Knoten hinzugefügt werden,  $n' = n + 1$ .

Die Anzahl der Flächen bleibt unverändert,  $f' = f$ .

Also gilt  $n' - e' + f' = n + 1 - (e + 1) + f = n - e + f = 2$ .

Da  $G'$  zusammenhängend ist, sind dies die einzigen Fälle. Wir haben also Eulers Formel per Induktion bewiesen.

2. Angenommen,  $K_{3,3}$  ist planar.  $K_{3,3}$  hat  $n = 6$  Knoten und  $e = 9$  Kanten. Nach Eulers Formel gibt es dann  $f = 9 + 2 - 6 = 5$  Flächen. Andererseits hat der Graph keine Kreise der Länge drei (vergl. vorherige Aufgabe). Also hat jede Fläche (mindestens) vier Kanten. Jede Kante ist in (höchstens) zwei Flächen. Um fünf Flächen abzugrenzen, braucht es daher (mindestens)  $4 \cdot 5/2 = 10$  Kanten. Widerspruch.

### Aufgabe 6. ( $O$ -Notation)

(2+2 Punkte)

- i) Sortieren Sie für die unten gegebenen Funktionen die  $O$ -Klassen  $O(a(n))$ ,  $O(b(n))$ ,  $O(c(n))$ ,  $O(d(n))$  und  $O(e(n))$  bezüglich ihrer Teilmengenbeziehung. Nutzen Sie ausschließlich die echte Teilmenge sowie die Gleichheit für die Beziehungen zwischen den Mengen.

$$a(n) = n^2 \cdot \log_2 n + 12$$

$$b(n) = 2^n + n^3$$

$$c(n) = 2^{2 \cdot n}$$

$$d(n) = 2^{n+4}$$

$$e(n) = \sqrt{n^3}$$

- ii) Beweisen oder widerlegen Sie  $(\log_2 n)^2 \in O(n)$ .

### Lösung:

- i)  $O(e(n)) \subset O(a(n)) \subset O(b(n)) = O(d(n)) \subset O(c(n))$

ii) Es gilt

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{(\log_2 n)^2}{n} &\stackrel{=l'Hopital}{=} \lim_{n \rightarrow \infty} \frac{2 \cdot \log_2 n \cdot \frac{1}{n \cdot \log(2)}}{1} \\ &= \lim_{n \rightarrow \infty} \frac{2 \cdot \log_2 n}{n \cdot \log(2)} \\ &\stackrel{=l'Hopital}{=} \lim_{n \rightarrow \infty} \frac{2 \cdot \frac{1}{n \cdot \log(2)}}{\log(2)} \\ &= \lim_{n \rightarrow \infty} \frac{2}{n \cdot (\log(2))^2} \\ &= 0.\end{aligned}$$

**Aufgabe 7.** (Laufzeiten von Algorithmen)

(1+3 Punkte)

Betrachten Sie die folgenden zwei Algorithmen. Eingabewerte sind jeweils ein Array  $a$  der Länge  $n$ , das nach Größe sortierte Zahlen enthalten soll, und eine Zahl  $value$ . versuchen Sie zu verstehen, was die Algorithmen tun und beschreiben Sie dies in Worten. Welche Laufzeiten haben diese Algorithmen? Sind die Laufzeiten polynomiell in der Länge des Arrays? Was sind Vor- und Nachteile der jeweiligen Algorithmen?

---

Alg1

---

```
Eingabe:  $a[1, \dots, n]$ ,  $value$ 
for  $i = 1, \dots, n$  do
  if  $a[i] == value$  then
    return  $i$  // found // STOP
  else
     $i = i + 1$ 
return -1 // not found
```

---

---

Alg2

---

```
Eingabe:  $a[1, \dots, n]$ ,  $value$ 
 $low = 1$ 
 $high = n$ 
while  $low \leq high$  do
   $mid = \lfloor (low + high) / 2 \rfloor$ 
  if  $a[mid] > value$  then
     $high = mid - 1$ 
  if  $a[mid] < value$  then
     $low = mid + 1$ 
  if  $a[mid] == value$  then
    return  $mid$  // found // STOP
return -1 // not found
```

---

**Lösung:**

Die Lineare Suche (Alg1) durchsucht das Array von vorne nach hinten, stoppt, wenn das gesuchte Element gefunden wird, und gibt die Fundstelle aus. Die Laufzeit ist  $O(n)$  und damit polynomial in der Länge des Arrays.

Die binäre Suche (Alg2) ist ein Algorithmus, der auf einem Array recht schnell ein gesuchtes Element findet bzw. eine zuverlässige Aussage über das Fehlen dieses Elementes liefert. Voraussetzung ist, dass die Elemente des Arrays in einer dem Suchbegriff entsprechenden Weise sortiert sind. Der Algorithmus basiert auf dem Schema Teile und Herrsche. Zuerst wird das mittlere Element des Arrays überprüft. Es kann kleiner, größer oder gleich dem gesuchten Element sein. Ist es kleiner als das gesuchte Element, muss

das gesuchte Element in der hinteren Hälfte stecken, falls es sich dort überhaupt befindet. Ist es hingegen größer, muss nur in der vorderen Hälfte weitergesucht werden. Die jeweils andere Hälfte muss nicht mehr betrachtet werden. Ist es gleich dem gesuchten Element, ist die Suche (vorzeitig) beendet.

Jede weiterhin zu untersuchende Hälfte wird wieder gleich behandelt: Das mittlere Element liefert wieder die Entscheidung darüber, wo bzw. ob weitergesucht werden muss.

Die Länge des Suchbereiches wird von Schritt zu Schritt halbiert. Spätestens, wenn der Suchbereich auf 1 Element geschrumpft ist, ist die Suche beendet. Dieses eine Element ist entweder das gesuchte Element, oder das gesuchte Element kommt nicht vor.

Um in einem Array mit  $n$  Einträgen ein Element zu finden bzw. festzustellen, dass dieses sich nicht im Array befindet, werden maximal  $\log_2(n)$  Schritte benötigt. Somit liegt die binäre Suche, in der Landau-Notation ausgedrückt, in der Komplexitätsklasse  $O(\log n)$ . Damit ist sie deutlich schneller als die lineare Suche mit Laufzeit  $O(n)$ , welche allerdings den Vorteil hat, auch in unsortierten Arrays zu funktionieren.

Die Laufzeit der binären Suche ist somit polynomial in der Länge des Arrays, und sogar noch schneller!

**Aufgabe 8.** (Programmieraufgabe: Fakultät, Binomialkoeff., Fibonacci-Zahlen)(1+1+1+1+2+2 Punkte)

Laden Sie die Datei `ProgAufgabe02.py` aus StudOn herunter und fügen Sie sie in ihr PyCharm ein.

Im ersten Teil dieser Programmieraufgabe werden Sie zwei verschiedene Berechnungen der Fakultät und eine Berechnung von Binomialkoeffizienten implementieren.

1. Zunächst implementieren Sie die rekursive Funktion `factorialRecursive` zur Berechnung der Fakultät  $n!$ . Eine rekursive Methode ist eine Methode die sich in irgendeiner Form selber aufruft. Hier haben wir einmal den Basisfall

$$0! = 1$$

und den rekursiven Aufruf

$$n! = n \cdot (n-1)!, \quad n \neq 0.$$

2. Implementieren Sie nun die Methode `factorialIterative`, die die Fakultät iterativ berechnet, d.h. ohne sich selbst aufzurufen (beispielsweise mit einer `for`-Schleife).
3. Zuletzt implementieren Sie die Methode `binomialCoefficient`, die den Binomialkoeffizienten  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$  berechnet. Verwenden sie hierfür eine der beiden von ihnen implementieren Methoden für die Fakultät.

Der zweite Teil der Aufgabe dreht sich um die Fibonacci-Zahlen  $\text{fib}(n)$ , die definiert sind durch folgende rekursive Vorschrift:

$$\begin{aligned} \text{fib}(0) &= 0, \\ \text{fib}(1) &= 1, \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2), \quad n \geq 2. \end{aligned}$$

1. Implementieren Sie zunächst die Methode `fibonacciA`, die die Fibonacci-Zahlen rekursiv berechnet. Setzen Sie dafür einfach die Rekursionsvorschrift von oben um. Diese Art der Rekursion nennt man *kaskadenartig*, da pro Rekursionsschritt zwei rekursive Aufrufe stattfinden, welche jeweils wiederum zwei rekursive Aufrufe starten, usw.. Diese Implementierung ist daher extrem langsam für große  $n$ .
2. Nun wenden wir uns einer effizienteren rekursiven Berechnung der Fibonacci-Zahlen zu. Die Idee ist dabei folgende Darstellung der Fibonacci-Zahlen

$$\begin{pmatrix} \text{fib}(n+1) \\ \text{fib}(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \text{fib}(n) \\ \text{fib}(n-1) \end{pmatrix},$$

Diese Form kann man jetzt schon für eine lineare, statt kaskadenartige Rekursion verwenden. Hierfür benötigen wir nun das Modul `numpy`, das am Anfang der Datei `ProgAufgabe02.py` mit `import numpy as np` eingefügt wurde. Mit `A = np.matrix('1 2 3; 4 5 6')` erstellen Sie beispielsweise die Matrix  $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ . Ein Vektor ist einfach eine  $2 \times 1$ -Matrix. Auf numpy-Matrizen können arithmetische Operatoren (+, -, \*) angewandt werden. Implementieren Sie nun die Methode `fibonacciB`, die obige, vektorielle Rekursion umsetzt. Wichtig: Die Methode soll bei Eingabe  $n$  einen 2-dim. numpy-Vektor ausgeben, der übereinander die Zahlen  $\text{fib}(n+1)$  und  $\text{fib}(n)$  beinhaltet!

3. Zuletzt gehen wir noch einen Schritt weiter und lösen die vektorielle Rekursion zu folgender expliziter Form auf:

$$\begin{pmatrix} \text{fib}(n+1) \\ \text{fib}(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \forall n \in \mathbb{N}_0.$$

Das Ziel ist es nun, die Matrixpotenz  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$  effizient zu berechnen. Nehmen wir zunächst an,  $n$  wäre eine 2er-Potenz, das heißt  $n = 2^k$ . Dann lässt sich  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$  durch  $k$ -faches Quadrieren von  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$  berechnen. Für allgemeine  $n$  betrachten wir deren Binärdarstellung

$$n = x_0 \cdot 2^0 + x_1 \cdot 2^1 + x_2 \cdot 2^2 + \dots + x_k \cdot 2^k,$$

wobei die  $x_i \in \{0, 1\}$  binär sind. Mit dieser Darstellung ist

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \prod_{i=0}^k \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{x_i \cdot 2^i}.$$

Implementieren Sie nun die Methode `fibonacciC`, die genauso wie `fibonacciB` bei Eingabe  $n$  die Zahlen  $\text{fib}(n+1)$  und  $\text{fib}(n)$  ausgibt. Als Hilfestellung ist in der Methode bereits eine `while`-Schleife implementiert, mit der Sie über alle Binärstellen iterieren und jeweils überprüfen, ob  $x_i = 0$  oder  $x_i = 1$  gilt.

### Lösung:

Die Musterlösung der Programmieraufgabe finden Sie im StudOn in der Datei `ProgAufgabe02_Loesung.py`.

Abgabe der Übung am 08.11.2017.