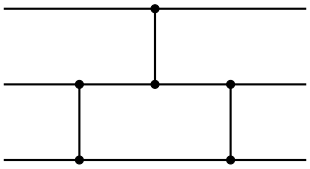
Datenstrukturen und Algorithmen, SS 2024
Prof. Dr. P. Rossmanith
D. Cloerkes, M. Gehnen, D. Mock, N. Speetzen

Abgabe: 31.05.2024

Übungsblatt mit Lösungen 06

Aufgabe T19

Manche Sortieralgorithmen lassen sich mithilfe so genannter Sortiernetzwerke[1] implementieren. In einem solchen Netzwerk repräsentieren wir die Felder des zu sortierenden Arrays als Linien, die von links nach rechts gehen. Wir können jeweils zwei dieser Linien mit einer Operation verknüpfen, dem Vergleich-Swap-Modul. Das Vergleich-Swap-Modul vertauscht die Inhalte der beiden verbunden Linien genau dann, wenn ihre Reihenfolge falsch ist. Das einfachste Sortiernetzwerk für 3 Linien sähe dann wohl so aus:



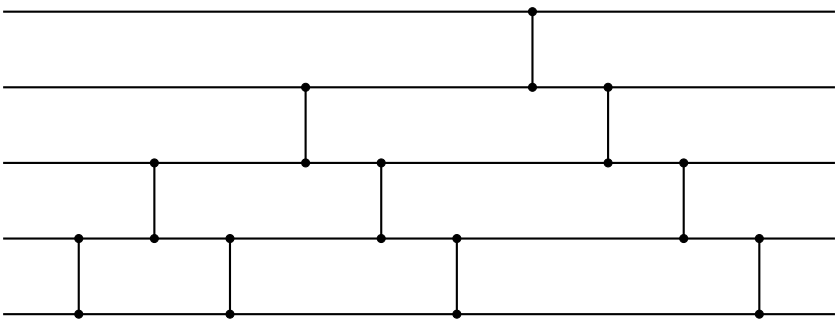
(a) Finden Sie ein Sortiernetzwerk für 5 Linien. Lässt sich Ihr Sortiernetzwerk einfach für beliebige n konstruieren?

(b) Wir definieren die Tiefe eines Sortiernetzwerkes als die maximale Anzahl an Vergleich-Swap-Modulen auf irgendeinem Pfad des Sortiernetzwerkes. Geben Sie die Tiefes ihres Sortiernetzwerkes an.

(c) Welche Vorteile sehen Sie in Sortiernetzwerken?

Lösungsvorschlag

a) Wir implementieren hier Insertionsort:



Dies lässt sich durch eine allgemeine Konstruktur für alle n verallgemeinern: Wir nehmen ein Sortiernetzwerk der Größe $n-1$ und hängen hinten $n-1$ von oben nach unten gehend an. Dadurch lassen sich beliebig große Sortiernetzwerke konstruieren.

[1] `https://www.youtube.com/watch?v=30WcPnvfiKE&t=44s`

b) In diesem Fall wäre dies 7: Wir beginnen von unten, gehen bis auf die zweit oberste und gehen dann wieder herunter zu untersten. Für die Allgemeine Konstruktion haben wir mit obiger Argumentation für n Linien eine Tiefe von $n-2+n-2+1 = 2n-3$. Es wird zwar kein Element geben, dass derart oft getauscht wird, jedoch gibt die Tiefe an, wie gut das Sortiernetzwerk parallelsiert werden kann.

c) Ein Sortiernetzwerk ist eine gute Methode, um oblivious, vergleichsbasierte Sortieralgorithmen zu visualisieren. Durch die geänderte Darstellungsmöglichkeit zeigt sich die Struktur des Algorithmus, Parallelisierungsoptionen, sowie Ähnlichkeiten zwischen Algorithmen: Nämlich wenn das Sortiernetzwerk für zwei Algorithmen identisch ist.

Aufgabe T20

	Quicksort	Heapsort	Mergesort	Insertion-Sort	Straight-Radix	Radix-Exchange
In-place?						
Stabil?						
Laufzeit (worst-case)						
Laufzeit (Durchschnitt)						
Vergleichsbasiert?						

Beantworten Sie die Fragen für alle Sortierverfahren. Gehen Sie davon aus, dass ein Vergleich in konstanter Zeit durchgeführt wird und die Anzahl der zu sortierenden Elemente n beträgt. Für Laufzeiten tragen Sie eine Funktion $f(n)$ in die Tabelle ein, um eine Laufzeit von $O(f(n))$ auszudrücken.

Lösungsvorschlag

	Quicksort	Heapsort	Mergesort	Insertion-Sort	Straight-Radix	Radix-Exchange
in-place?	??	J	N	J	N	??
stabil?	N	N	J	J	J	N
Laufzeit (worst-case)	n^2	$n \log n$	$n \log n$	n^2	nw	nw
Laufzeit (Durchschnitt)	$n \log n$	$n \log n$	$n \log n$	n^2	nw	nw
vergleichsbasiert?	J	J	J	J	N	N

Bei den Radix-Sortierverfahren bezeichne w die Wortlänge. Quicksort und Radix-Exchange-Sort sind wegen des benötigten Stacks nicht in-place. Beide lassen sich jedoch so implementieren, dass der Stack in rekursiven Funktionsaufrufen „versteckt“ wird, während jeder einzelne Aufruf nur konstant viel Platz benötigt. Bei Sortierverfahren sagen manche, dass sie nicht in-place sind, wenn sie $\Omega(n)$ viel Platz zusätzlich brauchen, andere, schon bei mehr als konstant vielem Zusatzplatz.

Aufgabe T21

	Liste	Sortierte Liste	Array	Sortiertes Array	Binärer Suchbaum	AVL-Baum	Splay-Tree	Treap	Skip-List	Hashtabelle	Min-Heap

Randomisiert											
Einfügen											
Suchen											
Löschen											
Minimum											
Maximum											
Sort. Ausgeben											

Beantworten Sie die Fragen für alle Datenstrukturen bzw. geben Sie eine obere Schranke für den Aufwand an. Gehen Sie davon aus, dass die Anzahl der enthaltenden Elemente n beträgt. Für Laufzeiten tragen Sie eine Funktion $f(n)$ in die Tabelle ein, um eine Laufzeit von $O(f(n))$ auszudrücken. Bei randomisierten Datenstrukturen ist die erwartete Laufzeit gemeint. Für Splay-Bäume tragen Sie die amortisierten Laufzeiten ein.

Lösungsvorschlag

	Liste	Sortierte Liste / Array	Sortiertes Array	Binärer Suchbaum / AVL-Baum	Splay-Tree	Treap	Skip-List	Hashtabelle	Min-Heap
Random	N	N	N	N	N	N	J	J	J	N
Einfügen	1	n	1	n	n	$\log n$	$\log n$	$\log n$	$\log n$	1	$\log n$
Suchen	n	n	n	$\log n$	n	$\log n$	$\log n$	$\log n$	$\log n$	1	n
Löschen	n	n	n	n	n	$\log n$	$\log n$	$\log n$	$\log n$	1	$\log n$
Minimum	n	1	n	1	n	$\log n$	$\log n$	$\log n$	1	n	1
Maximum	n	1	n	1	n	$\log n$	$\log n$	$\log n$	$\log n$	n	n
Sort. Aus.	$n \log n$	n	$n \log n$	n	n	n	n	n	n	$n \log n$	$n \log n$

Aufgabe T22

Entwickeln Sie eine Möglichkeit, die Adjazenzmatrix eines ungerichteten Graphens mit n Knoten in höchstens $n(n-1)/2$ Einträgen zu speichern, so dass es trotzdem in $O(1)$ Schritten möglich ist zu entscheiden, ob zwei Knoten i, j adjazent sind.

Lösungsvorschlag

Da der Graph ungerichtet ist, können Anfragen zur Adjazenz zweier Knoten i, j immer auf den Fall $i < j$ zurückgeführt werden. Den Fall $i = j$ beantworten wir immer mit "nein". Dieses entspricht genau der Darstellung in einer Adjazenzmatrix, in der nur die Dreiecksmatrix unterhalb bzw. oberhalb der Diagonalen verwendet werden. Man beachte, dass diese Dreiecksmatrix genau $n(n-1)/2$ Einträge hat, die wir nun geeignet speichern möchten.

Wir verwenden dazu ein Array, in der für die Knoten $1 \leq j \leq n$ in aufsteigender Reihenfolge die entsprechenden Zeilen der unteren Dreiecksmatrix einfach nacheinander gespeichert werden. Wir müssen nur noch effizient herausfinden können, an welcher Stelle im Array die Zeile für den Knoten j zu finden ist. Dazu beobachten wir, dass für den ersten Knoten keine Einträge, für den zweiten Knoten einen Eintrag (Adjazenz zum ersten Knoten), für den dritten Knoten zwei Einträge usw., und für den $j$ten Knoten $j-1$ Einträge gespeichert werden. Für n Knoten ergeben sich also insgesamt $\sum_{j=1}^{n}(j-1) = n(n-1)/2$ nötige Einträge. Außerdem erkennen wir

direkt, dass der Beginn der Zeile für den $j$ten Knoten nun an der Position $p(j) := (j-1)(j-2)/2$ im Array zu finden ist. Der der Eintrag zur Adjazenz der Knoten i und j für $i < j$ befindet sich daher an Position $p(j) + i - 1 = (j-1)(j-2)/2 + i - 1$ im Array. Diese Position ist durch zwei Multiplikationen und eine Addition also in konstanter Zeit zu bestimmen.

Aufgabe H19 (1 + 1 + 2 + 6 Punkte)

Gegeben seien zwei $n \times n$-Matrizen A und B, welche jeweils n^2 Buchstaben $A_{ij}, B_{ij} \in \{a, \ldots, z\}$ enthalten.

Ziel dieser Aufgabe ist ein schneller Algorithmus, der entscheiden kann, ob A und B genau die gleichen Zeilen (in möglicherweise anderer Reihenfolge) enthalten.

Beispiel: Hier lässt sich A durch Zeilenvertauschungen in B verwandeln

$$A = \begin{pmatrix} e & t & y & o & u & d \\ g & o & n & n & a & l \\ p & n & e & v & e & r \\ v & e & y & o & u & u \\ o & n & n & a & g & i \\ n & e & v & e & r & g \end{pmatrix} \qquad B = \begin{pmatrix} p & n & e & v & e & r \\ e & t & y & o & u & d \\ g & o & n & n & a & l \\ o & n & n & a & g & i \\ n & e & v & e & r & g \\ v & e & y & o & u & u \end{pmatrix}$$

(a) Ein Lösungsansatz ist es, die Matrizen zu sortieren und anschließend zu vergleichen. Wie lange dauert es, zwei Zeilen nach lexikographischer Ordnung zu vergleichen?

(b) Wie lange würde Heapsort benötigen, die Zeilen einer $n \times n$-Matrix lexikographisch zu sortieren?

(c) Jemand schlägt vor, zunächst jede Zeile in eine (sehr große) Zahl zu verwandeln (indem die Binärkodierungen der Buchstaben konkateniert werden) und diese n Zahlen dann in $O(n \log n)$ Schritten zu sortieren. Wo steckt der Fehler in dieser Idee?

(d) Wie kann man die Aufgabe in $O(n^2)$ Schritten lösen? Beschreiben Sie Ihren Algorithmus verständlich und begründen Sie seine Korrektheit und die Laufzeitschranke.

Hinweis: Wenn Sie Teil (d) ohne Sortieren lösen, dann müssen Sie (a)–(c) nicht für die volle Punktzahl abgeben.

Lösungsvorschlag

(a) $O(n)$

(b) $O(n^2 \log(n))$

(c) Unser Algorithmus basiert auf dem Vertauschen von Zeilen. Naiv dauert es $O(n)$ Zeit zwei Zeilen einer Matrix zu vertauschen. Daher bauen wir zunächst eine Datenstruktur, mit Spaltenvertauschungen in Zeit $O(1)$ gemacht werden können. Dies können wir zum Beispiel machen, indem jede Zeile einen Pointer auf ein Spaltenarray enthält. Dann kann man Spalten vertauschen, indem man zwei Pointer in $O(1)$ Zeit vertauscht.

Nun sortieren wir die Zeilen mithilfe von Radix-Sort nach ihrer lexikographischen Ordnung. Wir haben n Zeilen der Länge n und Vertauschungen dauern $O(1)$. Daher dauert die Prozedur $O(n^2)$. Jetzt enthalten A und B die gleichen Zeilen genau dann wenn beide Matrizen nach dem Sortieren identisch sind. Wir prüfen letzteres Zeit $O(n^2)$.

Ein Programm in Java, welches auf diese Weise eine Matrix sortieren kann ist besonders einfach, da in Java Matrizen als ein Array von Arrays implementiert sind und ein Array selbst eine Referenz ist, welche in konstanter Zeit zugewiesen werden kann.

```
static void sort(char A[][]) { // O(n^2) steps
  int n = A.length;
  for(int j = n − 1; j ≥ 0; j−−) {
    sortByColumn(A, j); // O(n) steps
  }
}

static void sortByColumn(char A[][], int j) {
  int n = A.length;
  int count[] = new int['z' + 1]; // O(1) steps
  int target[] = new int['z' + 1]; // O(1) steps
  for(int i = 0; i < n; i++) { // O(n) steps
    count[A[i][j]]++;
  }
  for(int i = 'a'; i ≤ 'z'; i++) { // O(1) steps
    target[i] = target[i − 1] + count[i − 1];
  }
  char B[][] = new char[n][]; // O(n) steps
  for(int i = 0; i < n; i++) { // O(n) steps
    B[target[A[i][j]]++] = A[i];
  }
  for(int i = 0; i < n; i++) { // O(n) steps
    A[i] = B[i];
  }
}
```

Alternativer Lösungsvorschlag mit Hashing: Wir berechnen den Hashwert jeder Zeile. Danach sortieren wir die Zeilen der beiden Matrizen bezüglich ihres Hashes in Zeit $O(n \log(n))$. Wir testen, ob nach dem Sortieren beide Matrizen die gleichen Hashwerte in den Zeilen haben. Falls dies nicht der Fall ist, so können wir *nein* zurück geben.

Falls die Hashwerte identisch sind, dann haben die Matrizen höchstwahrscheinlich die gleichen Zeilen. Es könnte jedoch auch (sehr selten) passieren, dass zwei unterschiedliche Zeilen auf den gleichen Wert gehasht werden. Wir testen in Zeit $O(n^2)$ ob beide Matrizen identisch sind, indem wir paarweise die Zeilen mit dem gleichen Hashwert vergleichen. Meistens ist dies der Fall und wir geben *ja* zurück.

Falls die Hashwerte gleich sind und die Matrizen unterschiedlich sind, hat es eine Kollision gegeben. Da dies sehr selten auftritt können wir in $O(n^2 \log(n))$ Zeit wie in Aufgabenteil *b*) testen, ob die Matrizen die gleichen Zeilen enthalten. Die erwartete Laufzeit des Algorithmus ist auf jeder Eingabe $O(n^2)$.

Welche Hashfunktion können wir verwenden? In der Vorlesung wurde eine Familie von universellen Hashfunktionen vorgestellt, welche ein Universum $\{0, \ldots, p-1\}$ auf $\{0, \ldots, m-1\}$ abbildet. Hier haben wir aber keine Zahlen, sondern ein Array mit n Symbolen, welche wir aber als Zahl zwischen 0 und 255 auffassen können. Wenn wir also eine Zeile als $(z_1, \ldots, z_n)$ mit $z_i \in \{0, \ldots, 255\}$ auffassen, dann können wir sie als eine Zahl

$$z = \sum_{i=1}^{n} 256^i z_i$$

interpretieren. Dann ist

$$h_{a,b}(z) = ((az + b) \bmod p) \bmod m = \left(a \sum_{i=1}^{n} 256^i z_i + b\right) \bmod p \bmod m,$$

welche wir mithilfe des Hornerschemas in $O(n)$ Schritten berechnen können. Wir müssen nicht mit Zahlen rechnen, die grösser als p sind.

```
static int hashValue(char x[], int a, int b, int p) {
  long h = 0;
  for(int i = 0; i < x.length; i++) {
    h = (256 * h + x[i])%p;
  }
  return(a * h + b)%p;
}
```

Die Wahrscheinlichkeit, dass zwei verschiedene Zeilen denselben Hashwert erhalten ist jetzt höchstens $1/m$ und die Wahrscheinlichkeit, dass irgendwelche verschiedene Zeilen auf den selben Wert abgebildet werden höchstens n^2/m. Wählen wir nun zum Beispiel $m = n^3$, dann müssen wir eine teure Überprüfung nur mit Wahrscheinlichkeit $1/n$ durchführen. Der Erwartungswert der Laufzeit der teuren Überprüfung ist dann nur $O(n^2 \log(n))/n = O(n^2)$. Da es keinen Grund gibt, $m \neq p$ zu wählen, können wir für $p = m$ einfach eine Primzahl wählen, die mindestens n^3 ist.

Aufgabe H20 (3 + 4 + 3 Punkte)
Gegeben sei ein Integer-Array der Größe m, welches jede Zahl von 1 bis n *genau einmal* enthält, wobei $n \leq m$. Die restlichen $m - n$ Positionen sind mit 0 gefüllt.

(a) Implementieren Sie einen Algorithmus, der ein gegebenes solches Array in $O(m)$ Schritten sortiert. Achten Sie insbesondere darauf, dass ein Fehler gemeldet wird, wenn das Array nicht dem gewünschten Format entspricht. Geben Sie eine kurze Beschreibung an. Implementieren Sie Ihren Algorithmus in Python/Java/C++/Rust.

(b) Wie kann man die obige Aufgabe *in place* lösen, also nur mit konstant viel Speicher zusätzlich zur Eingabe? Beschreiben Sie wieder Ihren Algorithmus und implementieren Sie diesen in Python/Java/C++/Rust.

(c) Geben Sie die gemessene Laufzeit Ihrer Programme (ohne die Zeit um die Datei einzulesen) in Sekunden an für die Listen, die Sie als Archiv `arrays.zip` auf unserer Website `https://tcs.rwth-aachen.de/lehre/DA/SS2024/arrays.zip` finden können.[2] Die Listen sind einfache ASCII Dateien `arrayX.txt`, in der jede Zeile eine Zahl enthält.

Lösungsvorschlag
Folgende Javafunktion löst die Aufgabe *in-place* in linearer Zeit.

```
void orderArray(int[] a) {
    int numZeros = 0;
    for (int i = 0; i < a.length; i++) {
        if (a[i] == 0) {
        numZeros++;
        }
    }

    int c = 0;
    while (c < a.length) {
        if (a[c] == 0 || c == a[c] + numZeros - 1) {
```

[2] In Moodle kann man nur Dateien hochladen, die nur höchstens 250 MB groß sind.

```
        c++;
        continue;
        }
        if (a[c] < 0 || a[c] + numZeros - 1 >= a.length) {
        throw new RuntimeException("Value in Array outside"
                                   + "of allowed range.");
        }
        if (a[c] == a[a[c] + numZeros - 1]) {
        throw new RuntimeException("Array contains " + a[c]
                                   + " more than once.");
        }
        int temp = a[c];
        a[c] = a[temp + numZeros - 1];
        a[temp + numZeros - 1] = temp;
    }
}
```

Zuerst zeigen wir, dass das Ergebnis richtig ist, wenn das Array das gewünschte Format hat. Das erste was die Funktion macht ist, die Anzahl an Nullen $numZeros$ im Array zu zählen. Dieser Schritt braucht $O(m)$ Zeit. Wenn man die Anzahl der Nullen kennt, dann weiß man, dass ein Wert w an der Position $w + numZeros - 1$ vom Array stehen muss. Der Zähler c gibt uns die jetzige Position im Array. Wir vertauschen dann den Wert an der jetzigen Position mit dem Wert an der Position wo er sein sollte $(a[c] + numZeros - 1)$. Wenn in der jetzigen Position c eine Null oder der richtige Wert steht, wird der Zähler um Eins erhöht. Von *0* bis $numZeros - 1$ wird der Zähler nur erhöht, wenn eine Null an dieser Position steht. Danach wird der Zähler nur erhöht, wenn der richtige Wert in der Aufzählung nach den Nullen steht. Da wir bei jedem Vertausch einen Wert richtig setzen, können maximal m Vertauschungen stattfinden und der Algorithmus muss terminieren. Somit ist auch gezeigt, dass der Algorithmus in $O(m)$ Zeit läuft.
Es fehlt nur noch zu zeigen, dass ein Fehler ausgegeben wird, wenn die Eingabe nicht das richtige Format hat. Wenn ein Wert mehr als einmal vorkommt, wird solange vertauscht bis dieser Wert zum zweiten mal gefunden wird. Hier wird dann die zweite *RuntimeException* geworfen. Wenn einer der Werte nicht erlaubt ist, also nicht zwischen 0 und m minus die Anzahl an Nullen beträgt, dann wird die erste *RuntimeException* geworfen. Wenn in dem Array einer der Werte ab Eins fehlt, dann ist ein anderer Wert zu groß, oder ein anderer Wert wird wiederholt. Hiermit ist gezeigt, dass wenn das Array nicht das richtige Format hat, ein Fehler ausgegeben wird.

Aufgabe H21 (10 Punkte)
Implementieren Sie ein Programm, welches folgende Funktion berechnen kann. Dabei sollen Sie aber sehr sparsam vorgehen: Sie dürfen ein vorinitialisiertes Array mit 17 Zahlen verwenden, eine XOR-Operation, eine MOD-Operation und einen lesenden Zugriff auf Ihr Array.

n	9689	9941	11213	19937	21701	23209	44497	86243	110503	132049
$f(n)$	3	5	7	13	17	19	23	37	47	59

n	216091	756839	859433	1257787	1398269	2976221	3021377
$f(n)$	61	67	71	79	89	101	103

Wenn Ihr Programm eine hier nicht aufgeführte Zahl als Eingabe bekommt, darf die Ausgabe beliebig sein. Das Programm darf aber auch dann nicht „abstürzen“.

Lösungsvorschlag
Wir versuchen eine Funktion der Form $k \mapsto (k \oplus z) \bmod 17$ zu finden, welche keine Kollisionen verursacht. Das Programm in Abbildung 1 ist dafür geeignet und es findet ein erfolgreiches $z = 481268$.

```
#include <bits/stdc++.h>

using namespace std;

vector<int> val { 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243,
                  110503, 132049, 216091, 756839, 859433, 1257787,
                  1398269, 2976221, 3021377};

int z;

int h(int k) {
  return abs(k^z)%17;
}

int has_collision() {
  vector<int> p(val.size());
  for(int n : val) {
    int s = h(n);
    if(p[s]) return true;
    p[s] = n;
  }
  return false;
}

int main() {
    for(int i = 0; i < 1000000; i++) {
      z = random()%4000000;
      if(!has_collision()) {
        cout << z << endl;
        return 0;
      }
    }
  return 100;
}
```

Abb. 1: Finden einer geeigneten Hashfunktion durch Probieren.

Jetzt können wir $f(k) = res[(k \oplus z) \bmod 17]$ festlegen und müssen nur noch das Array *res* festlegen. Zum Beispiel ist $(23209 \oplus 481268) \bmod 17 = 11$ und wir müssen daher $res[11] = 19$ setzen.

Das Programm in Abbildung 2 automatisiert dieses und berechnet für uns das Array *res*. Man kann es aber natürlich auch selbst machen.

Die Lösung ist jetzt einfach und kurz:

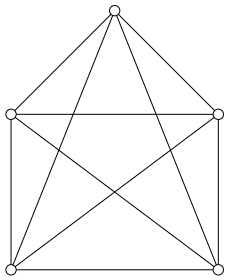
```
res = [5, 7, 37, 13, 67, 23, 79, 59, 17, 101, 103, 19, 89, 3, 47, 61, 71]

def f(x) :
  return res[(x^481268)%17]

#test f(x) on all relevant values
values = [9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503,
          132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377] :
for x in values :
  print(x, f(x))
```

```
val = [9689, 9941, 11213, 19937, 21701, 23209, 44497,
       86243, 110503, 132049, 216091, 756839, 859433,
       1257787, 1398269, 2976221, 3021377]

a = [3, 5, 7, 13, 17, 19, 23, 37, 47, 59, 61, 67, 71, 79, 89, 101, 103]
res = [0] * 17

i = 0
for x in val :
  res[(x^481268)%17] = a[i]
  i += 1

print(res)
```

Abb. 2: Finden des Arrays $res = [5, 7, 37, 13, 67, 23, 79, 59, 17, 101, 103, 19, 89, 3, 47, 61, 71]$

Aufgabe H22 (10 Punkte)

Ein *Dreieck*[3] in einem ungerichteten Graphen ist ein Untergraph, der aus drei Knoten besteht, welche paarweise miteinander verbunden sind.
Entwerfen Sie einen Algorithmus, der für einen Graphen mit n Knoten in $O(n^3)$ Schritten herausfinden kann, ob er ein Dreieck enthält.
Wie viele Dreiecke gibt es im rechten Graphen?



Lösungsvorschlag

Ein einfacher Algorithmus mit Laufzeit $O(n^3)$, der Dreiecke findet:

Betrachte alle Tripel $(u, v, w) \in V^3$ von Knoten und teste jeweils ob alle drei unterschiedlich und es alle drei möglichen Kanten zwischen ihnen gibt. Wenn wir eine Adjazenzmatrix haben (oder sie in quadratischer Zeit selbst aus einer Adjazenzliste berechnen), dann können wir diesen Test in konstanter Zeit durchführen, was zu insgesamt kubischer Laufzeit führt.

Der abgebildete Graph hat fünf Knoten. Er besitzt alle möglichen Kanten. Daher gibt es soviele Dreiecke, wie es Mengen von drei Knoten gibt, also $\binom{5}{3} = 10$.

[3] `https://www.youtube.com/watch?v=dQw4w9WgXcQ`