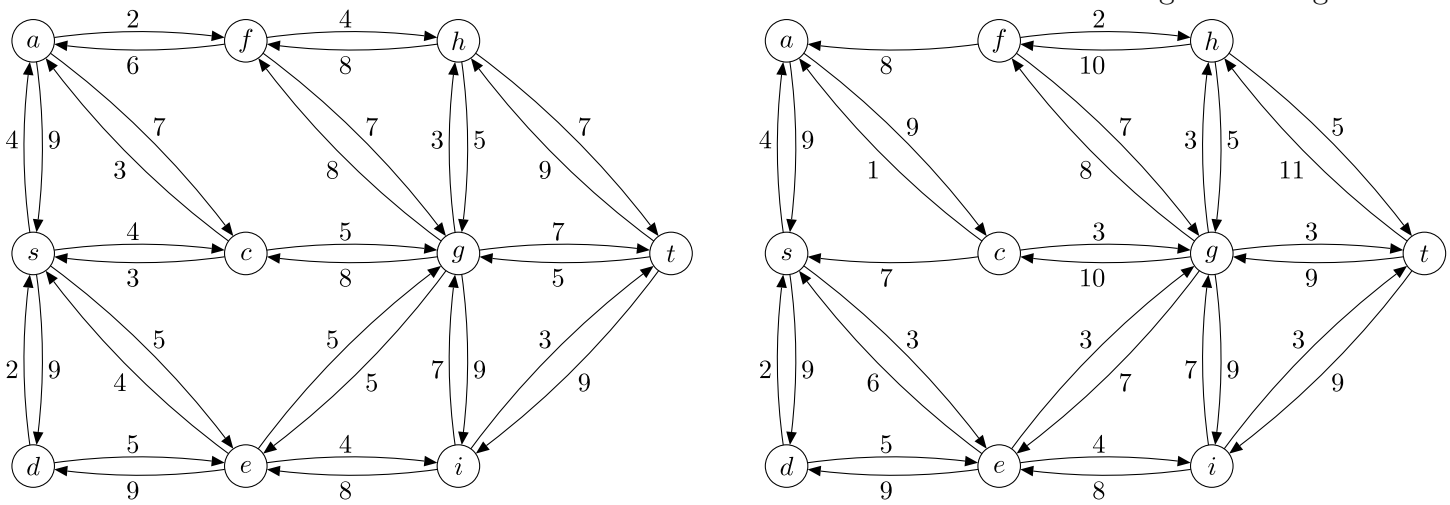
Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

Datum: 08.08.2019

Klausur mit Lösungen 01

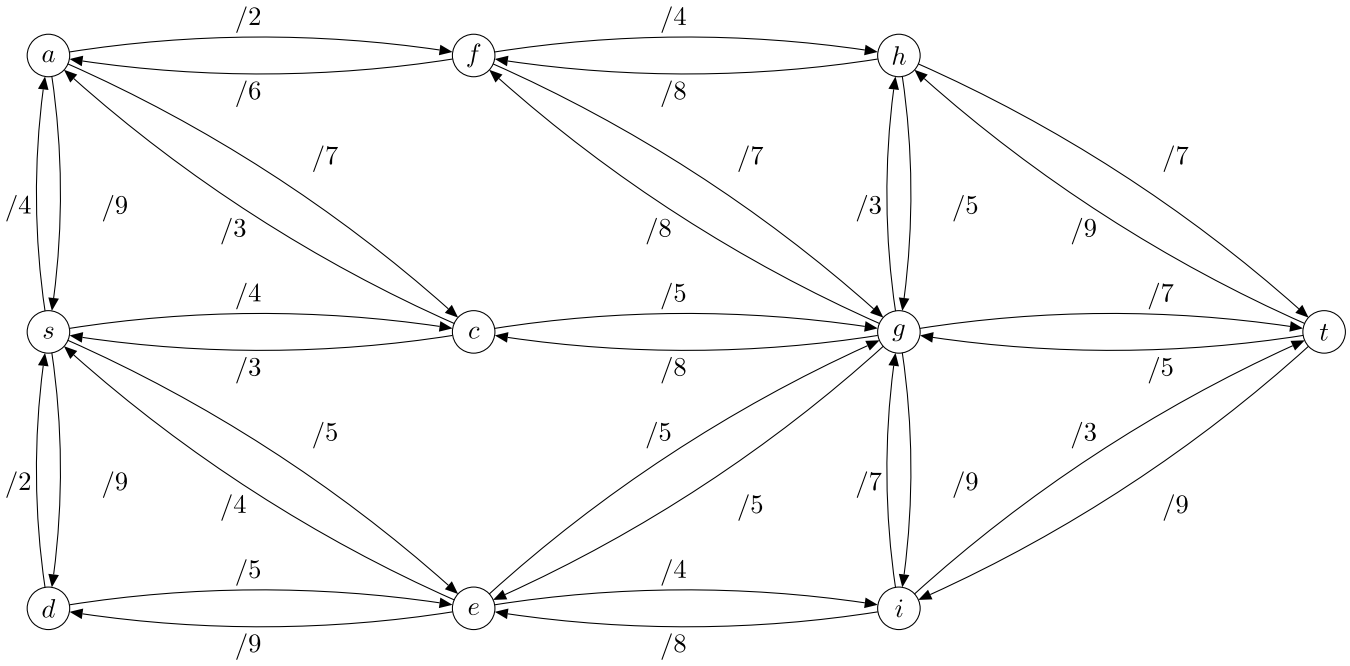
Aufgabe K1 (5+2+8+3 Punkte)
Sie finden links ein Flussnetzwerk und rechts das Residualnetzwerk nach zwei Augmentierungen:



a) Geben Sie die beiden augmentierenden Pfade an, indem Sie die zugehörigen Knoten in der entsprechenden Reihenfolge eintragen: [ ] und [ ].

b) Was ist der Wert des Flusses, der zu dem obigen Residualnetzwerk gehört? [ ]

c) Finden Sie einen maximalen Fluss zu dem Flussnetzwerk links und zeichnen Sie ihn hier ein (keine Nullen, nur positive Werte). Im Anhang dieser Klausur sind einige Kopien der unten stehenden Grafik, die sie als Schmierpapier verwenden können.

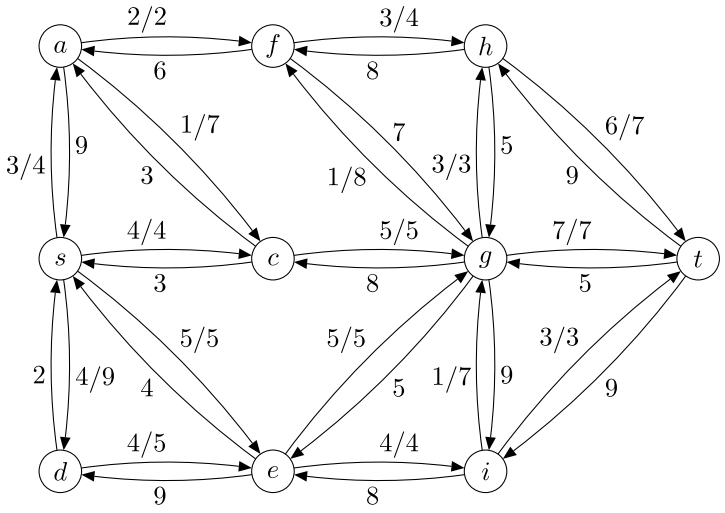


d) Geben Sie einen Schnitt (S, T) an, dessen Kapazität minimal ist:
$S = \{$ [ ] $\}$, $T = \{$ [ ] $\}$. Die Kapazität ist [ ].

Lösungsvorschlag

a) *scgt* und *segcafht*.
Typische Fehler: Es wurden mehr als zwei augmentierende Pfade gefunden, oder solche, die in Kombination nicht den gesamten Fluss abdecken.

b) Der Wert des Flusses ist 6.

c)


Typische Fehler: Es wurde ein augmentierender Pfad übersehen, es wurde mehr Fluss über eine Kante geschickt als Kapazität vorhanden war.
Typische Kuriosität, die zu keinem Punktabzug führt: Es wurde zwischen zwei Knoten in beide Richtungen Fluss geschickt.

d) (S, T) mit $S = \{s, a, c, d, e\}$ und $T = \{f, g, h, i, t\}$. Die Kapazität des Schnitts ist 16.

Aufgabe K2 (8+3+3 Punkte)
Gegeben sind folgende Schlüssel mit dazugehörigen Zugriffswahrscheinlichkeiten: A(0.25), E(0.08), I(0.12), O(0.35), U(0.20). Konstruieren Sie einen optimalen Suchbaum wie folgt:

a) Füllen Sie untenstehende Tabellen für $w_{i,j}$ und $e_{i,j}$ nach dem Verfahren aus der Vorlesung aus. Geben Sie in $e_{i,j}$ ebenfalls die Wurzel des optimalen Suchbaums für $\{i, \ldots, j\}$ an. Sie dürfen dazu die Notation aus der Übung verwenden.

$w_{i,j}$	A	E	I	O	U
A					
E	—				
I	—	—			
O	—	—	—		
U	—	—	—	—	

$e_{i,j}$	A	E	I	O	U
A	( )	( )	( )	( )	( )
E	—	( )	( )	( )	( )
I	—	—	( )	( )	( )
O	—	—	—	( )	( )
U	—	—	—	—	( )

b) Geben Sie den optimalen Suchbaum graphisch an, welcher sich aus Ihrem Ergebnis von Teilaufgabe a) ergibt.

c) Ist der optimale Suchbaum eindeutig? Geben Sie dazu eine kurze Begründung an.

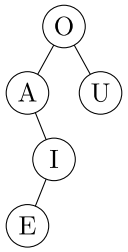
Sie finden im Anschluss an die Aufgaben weitere Kopien der obigen Tabellen für ihre Notizen.

Lösungsvorschlag

a)

$w_{i,j}$	A	E	I	O	U
A	0.25	0.33	0.45	0.80	1.00
E	0.00	0.08	0.20	0.55	0.75
I	0.00	0.00	0.12	0.47	0.67
O	0.00	0.00	0.00	0.35	0.55
U	0.00	0.00	0.00	0.00	0.20

$e_{i,j}$	A	E	I	O	U
A	0.25 (A)	0.41 (A)	0.73 (A)	1.53 (O)	1.93 (O)
E	0.00	0.08 (E)	0.28 (I)	0.83 (O)	1.23 (O)
I	0.00	0.00	0.12 (I)	0.59 (O)	0.99 (O)
O	0.00	0.00	0.00	0.35 (O)	0.75 (O)
U	0.00	0.00	0.00	0.00	0.20 (U)



b) Typische Fehler: Die lexikographische Ordnung wurde nicht eingehalten.

c) Der Optimale Suchbaum ist eindeutig, denn bei der Konstruktion von $e_{i,j}$ war jedes Minimum eindeutig.

Man darf sich auch darauf berufen, dass die Wurzel jedes Teilbaumes in der Tabelle eindeutig ist.
Typische Fehler: Die Formulierungen waren sehr ungenau, es ließ sich zwar vermuten, dass obige Antwort gemeint war, jedoch mit zu viel Interpretationsspielraum. Die Aussage, dass ein Baum uneindeutig ist, sobald ein Teilbaum der Tabelle eine mehrdeutige Wurzel hat, ist im Allgemeinen nicht korrekt.

Aufgabe K3 (2+2+9 Punkte)

a) Wann ist ein Sortierverfahren „in-place“?

b) Wann ist ein Sortierverfahren stabil?

c) Beantworten Sie die Fragen für alle Sortierverfahren in folgender Tabelle. 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. Schätzen Sie dabei die Laufzeit möglichst präzise ab. Für die Sortierverfahren, welche nicht vergleichsbasiert sind, drücken Sie die Laufzeiten durch Funktionen $f(n, w)$ aus, wobei w die Wortlänge der zu sortierenden Elemente in Bits ist. Durchschnittliche Laufzeiten beziehen sich auf n verschiedene Elemente, die zufällig permutiert sind.

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

Lösungsvorschlag

a) Ein Sortierverfahren ist „in-place“, wenn es nur konstant viel zusätzlichen Speicher zur Ausführung benötigt. Hierbei wird das unsortierte Eingabearray mit dem sortierten Ausgabearray überschrieben. Auch logarithmisch viel zusätzlicher Speicher wäre richtig. Kein zusätzlicher Speicher dagegen ist falsch.

b) Ein Sortierverfahren ist stabil, wenn die Reihenfolge gleicher Schlüssel im Ausgabearray der Reihenfolge dieser Schlüssel im Eingabearray entspricht.
Typische Fehler: Es wird verlangt, dass die Reihenfolge zwischen zwei Elementen zu jedem

Zeitpunkt gleich bleibt, dies muss jedoch nur nach der Ausführung des Algorithmus gelten. Weiterhin falsch ist es zu verlangen, dass der absolute Abstand zwischen zwei Elementen identisch bleibt.

c) Für Quicksort und Radix-Exchange gibt es jeweils Implementierungen die In-place als auch nicht In-place sind. Daher sind hier beide Antworten richtig.

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.

	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

Aufgabe K4 (10+5 Punkte)

a) Entwerfen Sie einen Algorithmus der folgendes Problem mit einer Laufzeit von $O(n)$ löst: Die Eingabe ist ein Array mit $n \geq 1$ natürlichen Zahlen $k_1, \ldots, k_n$. Die Frage ist, ob die Zahlen im Array lückenlos sind, d.h. dass das Array eine Menge $\{m, m+1, m+2, \ldots, m+l\}$ von Zahlen enthält. Das Array $3, 6, 4, 5, 3$ hat diese Eigenschaft, aber $3, 6, 4, 6, 3$ hat sie nicht. Beachten Sie, dass in einem lückenlosen Array Zahlen durchaus mehrfach vorkommen dürfen. Sie dürfen dabei beliebig viel zusätzlichen Speicher verwenden.

b) *Wir empfehlen, diese Teilaufgabe erst dann zu bearbeiten, wenn Sie bereits mit der Bearbeitung der restlichen Klausur fertig sind.* Lösen Sie das Problem aus a) in Zeit $O(n)$, jedoch mit der zusätzlichen Einschränkung, dass Sie nur konstant viel zusätzlichen Speicher zur Verfügung haben und so, dass das Array am Ende wieder den gleichen Inhalt hat, wie zu dem Zeitpunkt als der Algorithmus startete. Sollte ihre Lösung aus a) dieses Kriterium bereits erfüllen, müssen Sie diese Teilaufgabe nicht bearbeiten.

Beschreiben Sie jeweils, wie Ihr Algorithmus vorgeht und geben sie ihn zusätzlich in Pseudocode oder einer vernünftigen Programmiersprache an. Argumentieren Sie weiterhin, warum die Laufzeit Ihres Algorithmus $O(n)$ ist.

Lösungsvorschlag

a) Ein einfaches Verfahren das dieses Problem mit linear viel zusätzlichem Speicherplatz löst, arbeitet wie folgt: Man berechnet erst einmal das kleinste Element m und das größte Element M. Falls $M - m \geq n$, dann kann es keine Lösung geben und der Algorithmus sagt „nein“. Ansonsten legen wir ähnlich zum Bucket-Sort ein Array A der Länge $M - m + 1$ an und iterieren wie folgt über unser ursprüngliches Array: Von jeder Zahl z aus unserem

Array ziehen wir die kleinste Zahl m ab und fügen einen Marker in $A[z-m]$ ein. Zuletzt iterieren wir über A und überprüfen, ob an jeder Stelle ein Marker vorhanden ist. Falls dem so ist, antwortet der Algorithmus „ja“, sonst „nein“.

```
static boolean lueckenlosSimpel(int a[]) {
  int n = a.length;
  if(n ≤ 1) return true;
  int m = a[0];
  int M = a[0];
  for(int i = 0; i < n; i++) {
    if(a[i] < m) m = a[i];
    if(a[i] > M) M = a[i];
  }
  if(M − m ≥ n) return false;
  int A[M − m + 1];
  for(int i = 0; i < n; i++) {
    int z = a[i];
    A[z − m] = 1;
  }
  for(int i = 0; i < M − m + 1; i++) {
    if(A[i] ≠ 1) return false;
  }
  return true;
}
```

Typische Fehler: Standardsortierverfahren brauchen mindestens $O(n\log(n))$ Zeit, was zu langsam ist und daher null Punkte gibt. Radixsort läuft in Zeit $O(n\cdot w)$. Dies ist ebenfalls nicht ausreichend, da w nicht beschränkt ist. Wenn man vergisst zu testen, ob $M-m \geq n$, dann dauert der letzte Schritt $M-n$ Zeiteinheiten, was nicht $O(n)$ ist. Wenn man ein Array der Größe M alloziiert, dann braucht man $O(M)$ Zeiteinheiten, was nicht $O(n)$ ist.

b) Man berechnet erst einmal das kleinste Element m und das größte Element M. Falls $M-m \geq n$, dann kann es keine Lösung geben und der Algorithmus sagt „nein“. Ansonsten gehen wir einmal durch das Array durch und markieren in einem zweiten Array b jede gefundene Zahl x durch $b[x-m]=1$ (anfangs soll b nur Nullen enthalten). Dann müssen wir nun überprüfen, ob $b[i]=1$ für alle $i=0,\ldots,M-m$ gilt, denn genau dann enthält das Eingabearray alle Zahlen zwischen m und M (einschliesslich).

Wir können auf das zweite Array b verzichten, indem wir die Markierungen im Eingabearray a so anbringen, dass sie wieder entfernt werden können. Zum Beispiel durch $a[i] := a[i]+(M+1)$. (Wichtig ist natürlich, dass jetzt eine Markierung nicht zweimal angebracht werden darf.) Alternativ kann man auch einen Eintrag markeren, indem man ihn mit -1 multipliziert.

```
static boolean lueckenlos(int a[]) {
  int n = a.length;
  if(n ≤ 1) return true;
  int m = a[0];
  int M = a[0];
  for(int i = 0; i < n; i++) {
    if(a[i] < m) m = a[i];
    if(a[i] > M) M = a[i];
  }
  if(M − m ≥ n) return false;
  for(int i = 0; i < n; i++) {
    int x = a[i];
    if(x > M) x = x − (M + 1);
    if(a[x − m] ≤ M) a[x − m] += M + 1;
  }
  boolean lueckenlos = true;
  for(int i = 0; i ≤ M − m; i++) {
    if(a[i] ≤ M) lueckenlos = false;
  }
  for(int i = 0; i < n; i++) {
    if(a[i] > M) a[i] −= M + 1;
  }
  return lueckenlos;
}
```

b) Alternative Lösung: Zunächst berechnen wir in $O(n)$ den kleinsten Wert min sowie den größten Wert max im Array. Falls das Intervall $[min, max]$ mehr als n natürliche Zahlen enthält, so gibt es keine Lösung und der Algorithmus gibt $false$ zurück. Anschließend gilt also $max - min < n$.

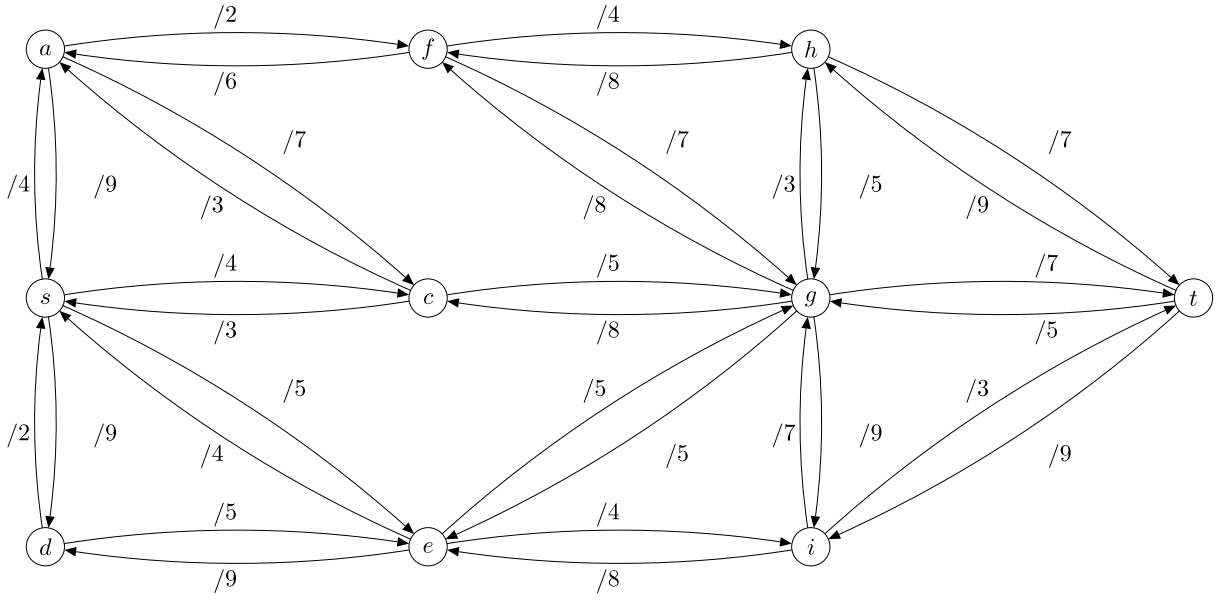
Nun zählt der Algorithmus die *duplicates* im Array. Dazu läuft er in $O(n)$ einmal über das ganze Array. Sobald er eine Zahl z das erste mal sieht, merkt er sich dies. Für jedes weitere Mal, welches er die selbe Zahl z erneut sieht, inkrementiert er *duplicates* um eins. Am Ende wurde *duplicates* also für jeden Arrayeintrag um eins erhöht, außer wenn dieser Arrayeintrag eine bisher unbekannte Zahl enthalten hat. Der Ausdruck $n - duplicates$ liefert uns nun also die Anzahl der **verschiedenen** Zahlen im Array. Der Algorithmus merkt sich ob er eine Zahl z bereits gesehen hat, indem er im Array an der Stelle $z - min$ das Vorzeichen auf $-$ verändert. Damit dies funktioniert nehmen wir hier also an, dass 0 keine natürliche Zahl ist und somit nicht im Array enthalten sein kann (0 hat kein Vorzeichen). Damit der Algorithmus weiterhin mit den korrekten, positiven Zahlen arbeitet, muss er beim Auslesen der Zahl den Betrag nehmen, also die Markierung ignorieren. Außerdem wissen wir, da $max - min < n$, dass wir für diese Markierungen im Eingabearray ausreichend Platz haben.

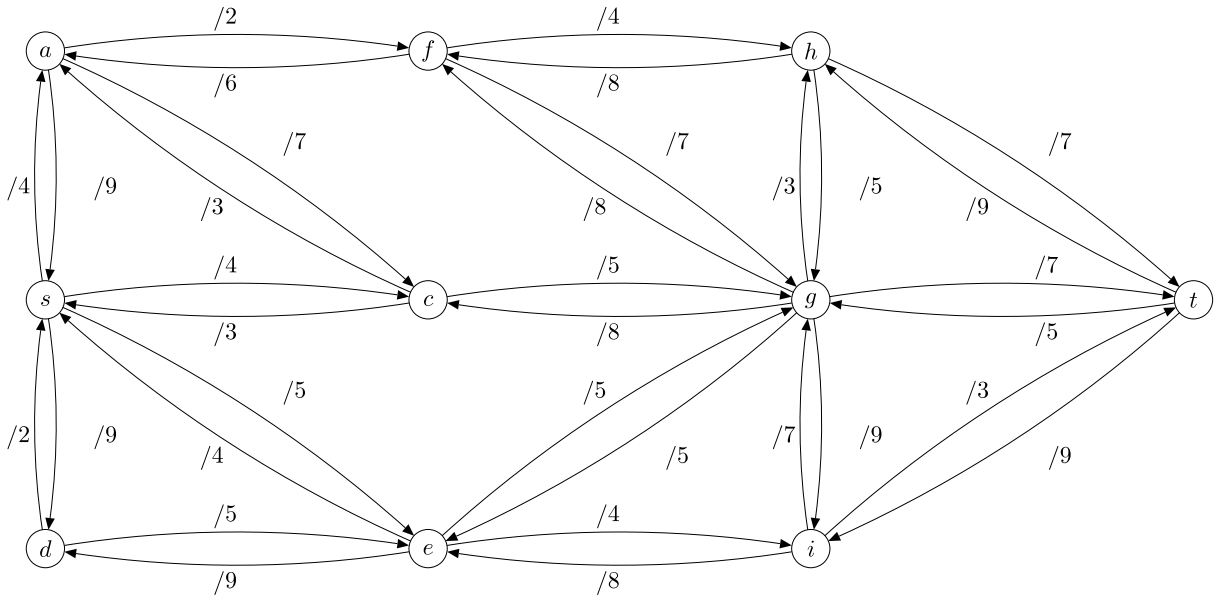
Um die Anforderung, das Eingabearray am Ende wiederherzustellen, zu erfüllen, wechseln wir nun erneut in $O(n)$ das Vorzeichen aller Arrayeinträge auf $+$.

Abschließend gibt der Algorithmus genau dann *true* zurück, falls die Anzahl der verschiedenen Zahlen im Array $n - duplicates$ gleich der Anzahl der natürlichen Zahlen im Intervall $[min, max]$ ist. Ansonsten gibt der Algorithmus $false$ zurück. Dies ist korrekt, denn das Array kann nicht mehr verschiedene Zahlen enthalten als es natürliche Zahlen im Intervall $[min, max]$ gibt. Falls das Array jedoch weniger verschiedene Zahlen enthalten würde, als es natürliche Zahlen im Intervall $[min, max]$ gibt, so gäbe es eine Lücke im Array. Die Existenz einer solchen Lücke wird durch das Prüfen dieser Gleichheit verhindert.

```
static boolean lueckenlos2(int[] a) {
  int n = a.length;
  if(n == 0) {
    return true;
  }

  int min = a[0];
  int max = a[0];
  for(int value : a) {
    if(min > value) {
      min = value;
    }
    if(max < value) {
      max = value;
    }
  }

  if(max - min + 1 > n) {
    return false;
  }

  int duplicates = 0;
  for(int i = 0; i < n; i++) {
    int ai = a[i];
    int j = (ai > 0 ? ai : -ai) - min;
    if(a[j] > 0) {
      a[j] *= -1;
    }else {
      duplicates++;
    }
  }

  for(int i = 0; i < n; i++) {
    if(a[i] < 0) {
      a[i] *= -1;
    }
  }
  return(n - duplicates) == (max - min + 1);
}
```

Sie können folgende Grafiken als Schmierpapier verwenden, um Aufgabe 1 zu lösen. Bitte beachten Sie, dass eingezeichnete Lösungswege in diesen Grafiken nicht zur Bewertung herangezogen werden. Sollten Sie ihr Endergebnis dennoch in diese Grafiken eintragen wollen, so kennzeichnen Sie dies deutlich!





Sie können folgende Tabellen als Schmierpapier verwenden, um Aufgabe 2 zu lösen. Bitte beachten Sie, dass die Einträge in diesen Tabellen nicht zur Bewertung herangezogen werden. Sollten Sie ihr Endergebnis dennoch in diese Tabellen eintragen wollen, so kennzeichnen Sie dies deutlich!

$w_{i,j}$	A	E	I	O	U
A					
E	—				
I	—	—			
O	—	—	—		
U	—	—	—	—	

$e_{i,j}$	A	E	I	O	U
A	( )	( )	( )	( )	( )
E	—	( )	( )	( )	( )
I	—	—	( )	( )	( )
O	—	—	—	( )	( )
U	—	—	—	—	( )

$w_{i,j}$	A	E	I	O	U
A					
E	—				
I	—	—			
O	—	—	—		
U	—	—	—	—	

$e_{i,j}$	A	E	I	O	U
A	( )	( )	( )	( )	( )
E	—	( )	( )	( )	( )
I	—	—	( )	( )	( )
O	—	—	—	( )	( )
U	—	—	—	—	( )

Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

RWTH AACHEN UNIVERSITY

Datum: 08.08.2019

Klausur mit Lösungen 01

Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

RWTH AACHEN UNIVERSITY

Datum: 08.08.2019

Klausur mit Lösungen 01

Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

RWTH AACHEN UNIVERSITY

Datum: 08.08.2019

Klausur mit Lösungen 01

Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

RWTH AACHEN UNIVERSITY

Datum: 08.08.2019

Klausur mit Lösungen 01

Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

RWTH AACHEN UNIVERSITY

Datum: 08.08.2019

Klausur mit Lösungen 01

Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

RWTH AACHEN UNIVERSITY

Datum: 08.08.2019

Klausur mit Lösungen 01