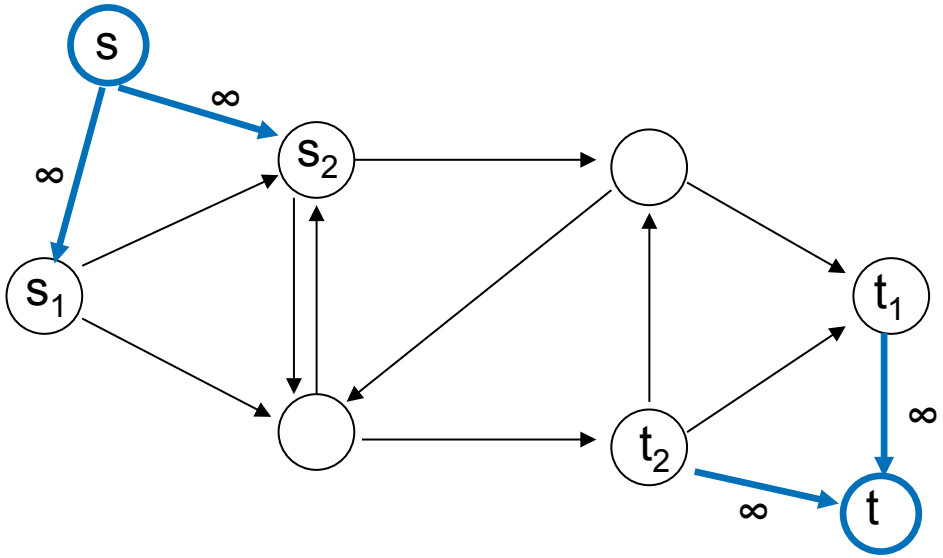
Verwandte Probleme des Maximalen-Fluss-Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

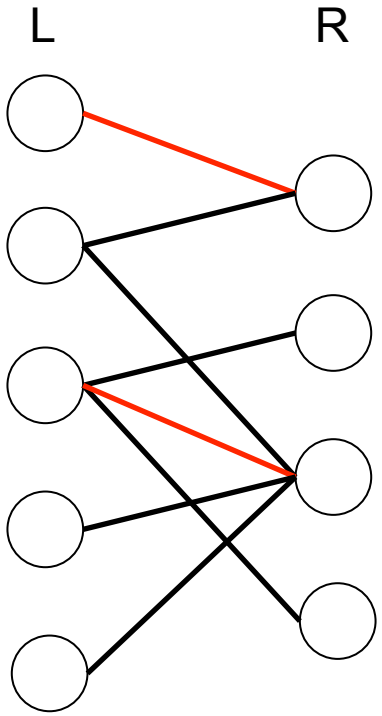
1. Mehrere Quellen und/oder Senken



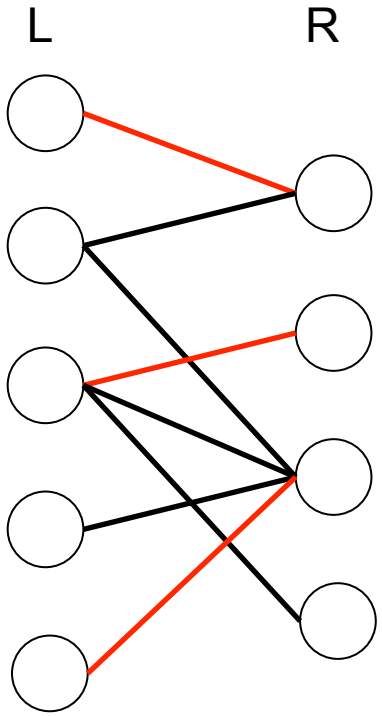
Verwandte Probleme des Maximalen-Fluss-Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

2. Maximum Matching in bipartiten Graphen



Matching der Größe 2

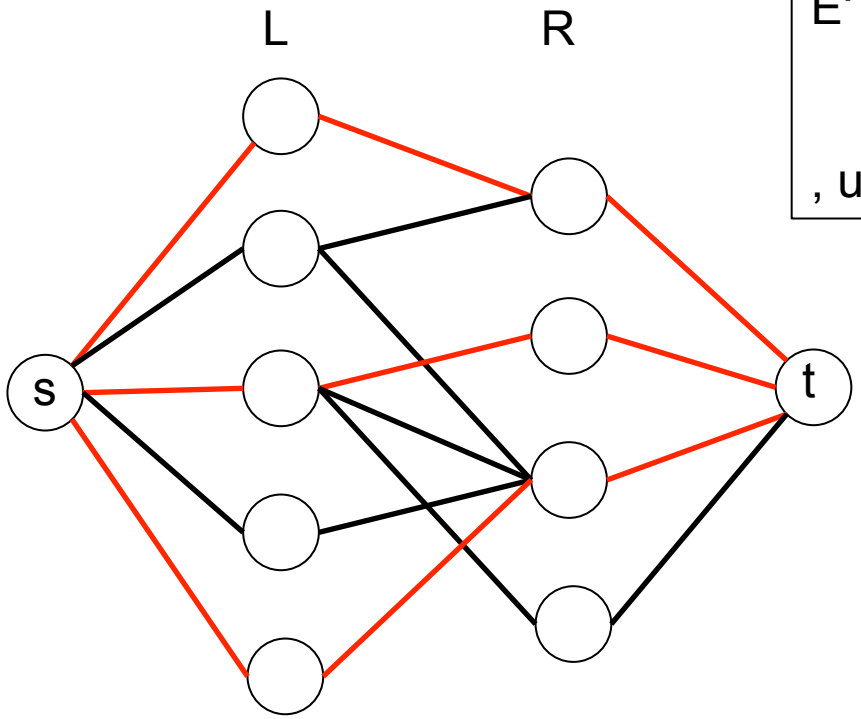


Maximum-Matching der Größe 3

Verwandte Probleme des Maximalen-Fluss-Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

2. Maximum Matching in bipartiten Graphen



$E' = \{ (s,u) : u \in L \}$
$\cup \{(u,v) : u \in L, v \in R, (u,v) \in E\}$
$\cup \{ (u,t) : u \in R \}$
, und Einheitsgewichte für die Kanten

Verwandte Probleme des Maximalen-Fluss-Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

2. Maximum Matching in bipartiten Graphen

Satz MaxBiMa: Sei G=(V,E) ein bipartiter Graph mit Knotenpartitionierung $V = L \cup R$. Sei G‘=(V‘,E‘) das zugehörige Flussnetzwerk. Dann gilt:

Wenn M ein Matching in G ist, dann gibt es einen ganzzahligen Fluss in G‘ mit $|f| = |M|$. Wenn andersherum f ein ganzzahliger Fluss in G‘ ist, dann gibt es ein Matching M in G mit $|f|=|M|$.

Beweis: Übungsaufgabe

Sortieren

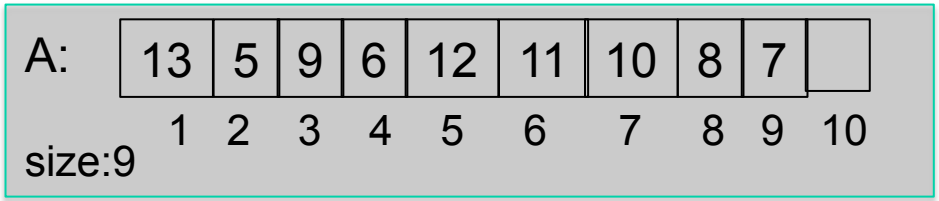
TECHNISCHE UNIVERSITÄT DARMSTADT

Input: Folge von Zahlen $<a_1,...,a_n>$
Gesucht: Permutation $<a'_1,...,a'_n>$, so dass $a'_1 \leq ... \leq a'_n$

Dass wir „nur“ Zahlenfolgen sortieren wollen, ist willkürlich, und macht für die Algorithmik keinen Unterschied.

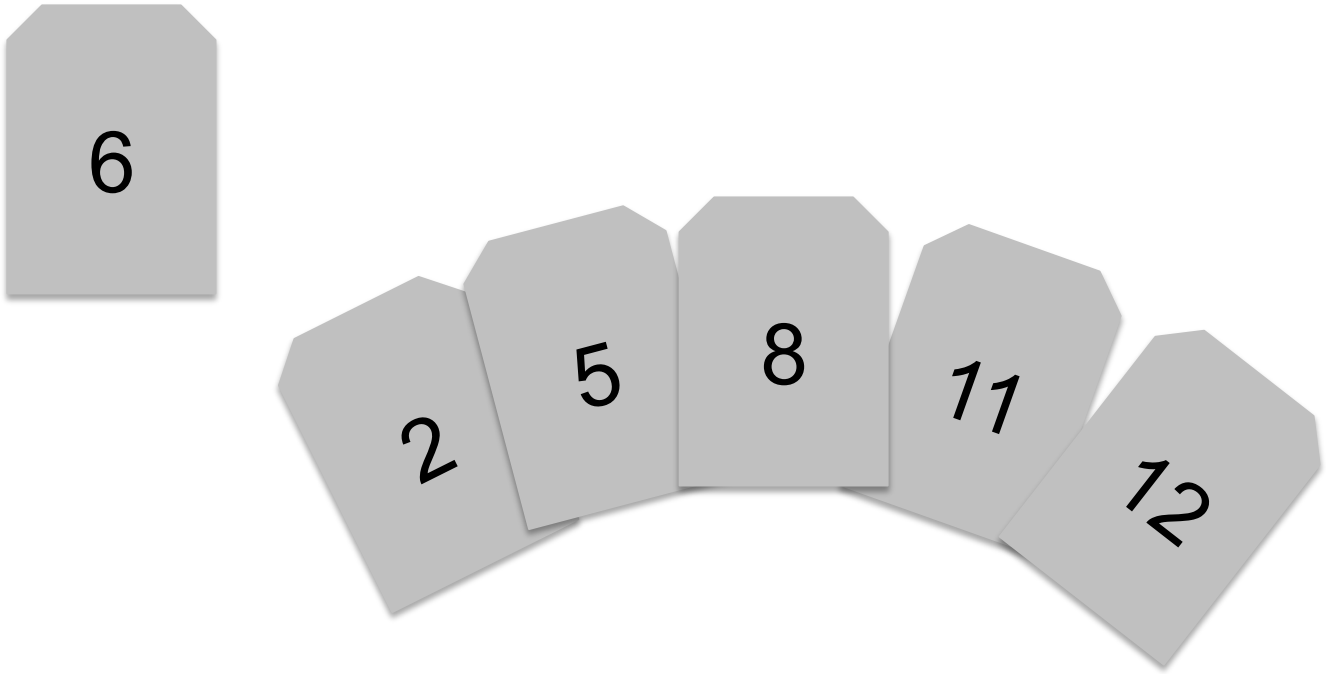
Allgemeiner können wir davon ausgehen, dass eine Menge von Objekten vorliegt, für die eine Ordnungdrelation definiert ist.
Das kann ebenfalls ein Zahlenschlüssel sein, kann aber auch z.B. eine Lexikographische Ordnung sein, wie man sie zum Sortieren von Namen Verwendet.

Daten liegen typischerweise in Arrays:
in von 1 bis n indizierbaren Zellen hintereinander



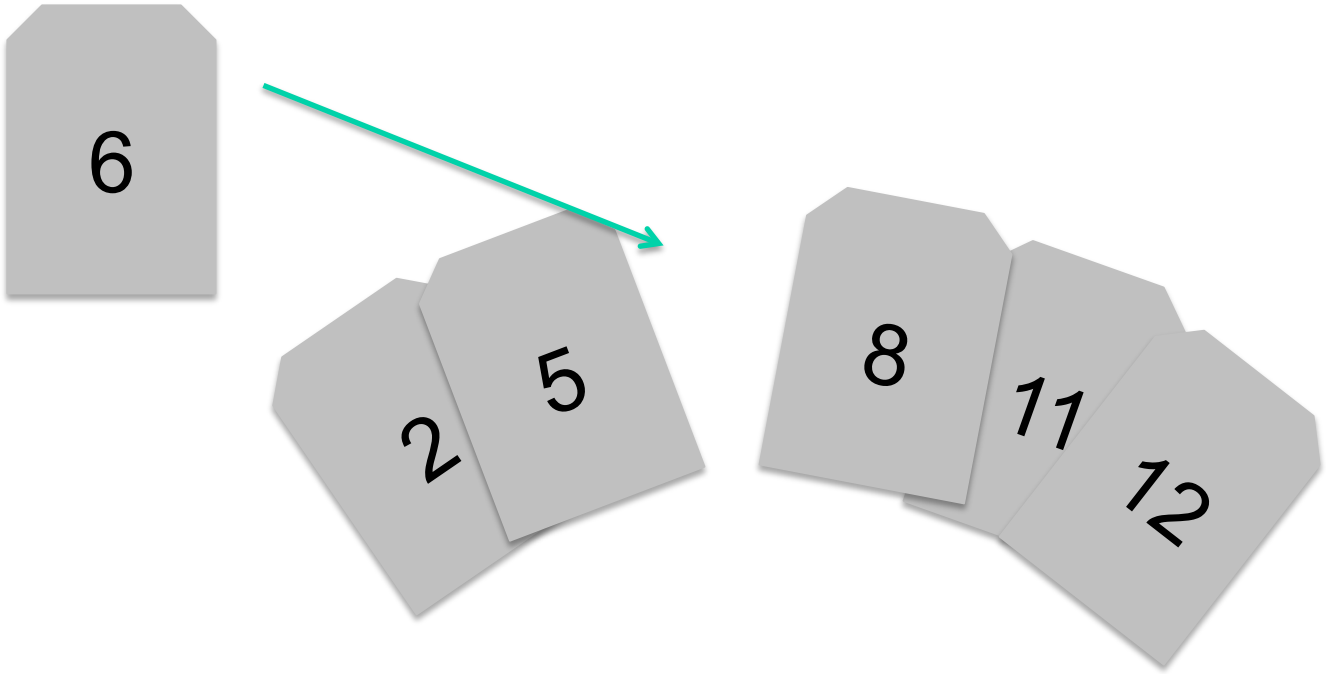
Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT



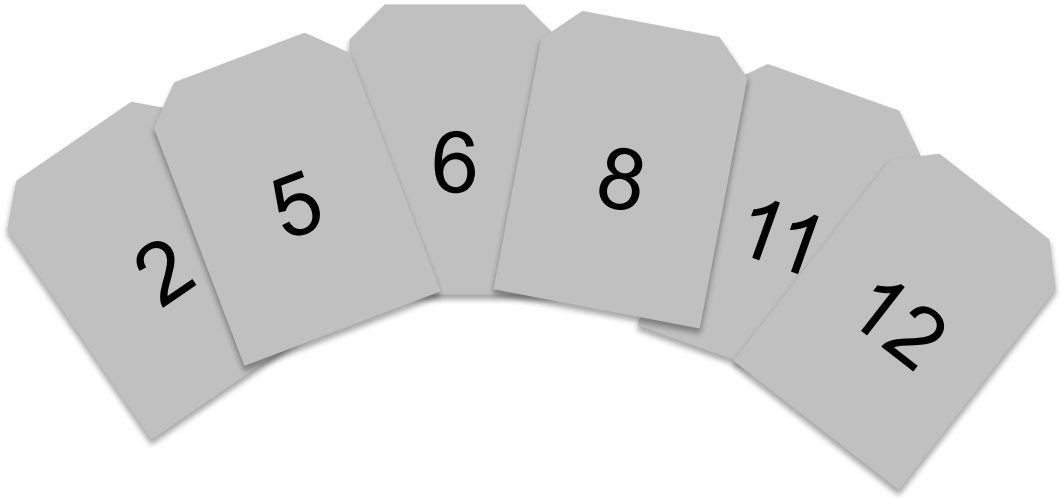
Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT



Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT

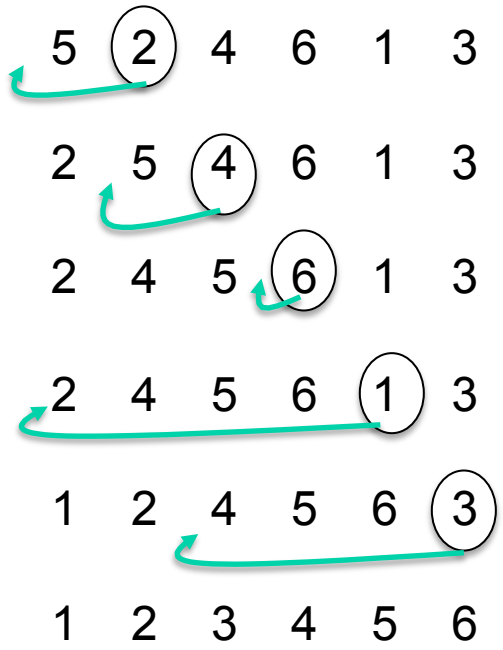


Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT

Insertion-Sort

```
Insertion-Sort(A)
 1. for j := 2 to length(A) do
 2.       key := A[j]
 3.       i := j – 1
 4.       while i > 0 and A[i] > key do
 5.             A[i+1] := A[i]
 6.             i := i-1
 7.       A[i+1] := key
```



Aufwand: $O(n^2)$

Merge-Sort

Die Merge-Operation:
Seien A_1 und A_2 zwei sortierte Arrays der Längen m und n. Diese sollen zu einem neuen sortierten Array A der Länge m+n verschmolzen werden.
Dazu durchlaufe A_1 und A_2 von „links“ nach „rechts“ mit zwei Indexzeigern i und j und führe eine Art „Reißverschlußverfahren“ aus:

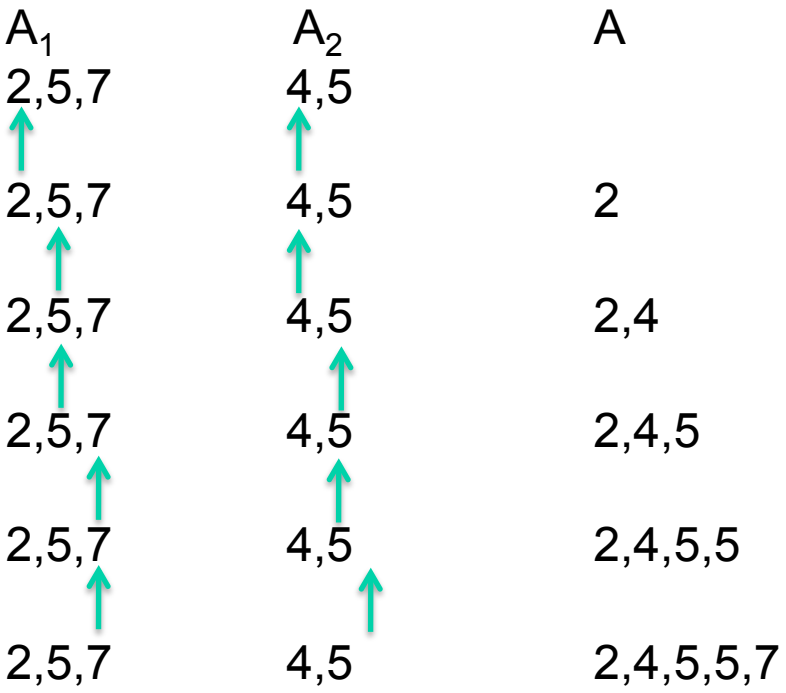
1. i := 1; j := 1; k := 1;
2. **while** i <= m **and** j <= n do
3. **if** A_1[i] < A_2[j] **then** A[k] := A_1[i]; k := k+1; i := i+1;
4. **else** A[k] := A_2[j]; k := k+1; j := j+1;
5. **if** i==m+1 **and** j <= n **then** hänge die restlichen Elemente von A_2 an A;
6. **if** j==n+1 **and** i <= m **then** hänge die restlichen Elemente von A_1 an A;

Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT

Merge-Sort

Die Merge-Operation:
Beispiel:
A_1 = (2,5,7) und A_2 = (4,5)



Korrektheit: klar; Aufwand: O(n+m)

Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT

Merge-Sort

```
Merge-Sort(A,l,r)              // sortiere im Array A von Index l bis Index r
 1. if l < r then
 2.     m := ⌊(l+r)/2⌋
 3.     Merge-Sort(A,l,m)
 4.     Merge-Sort(A,m+1,r)
 5.     Merge(A,l,m,r)          // merge das Array A[l..m] und das Array A[m+1,r]
```

Korrektheit: Induktion nach Anzahl der zu sortierenden Komponenten
Laufzeit: Sei n die Länge der zu sortierenden Zahlenfolge. Dann ist

$$T(n) \begin{cases} \Theta(1), & \text{falls } n = 1 \\ 2T(n/2) + \Theta(n), & \text{falls } n > 1 \end{cases}$$

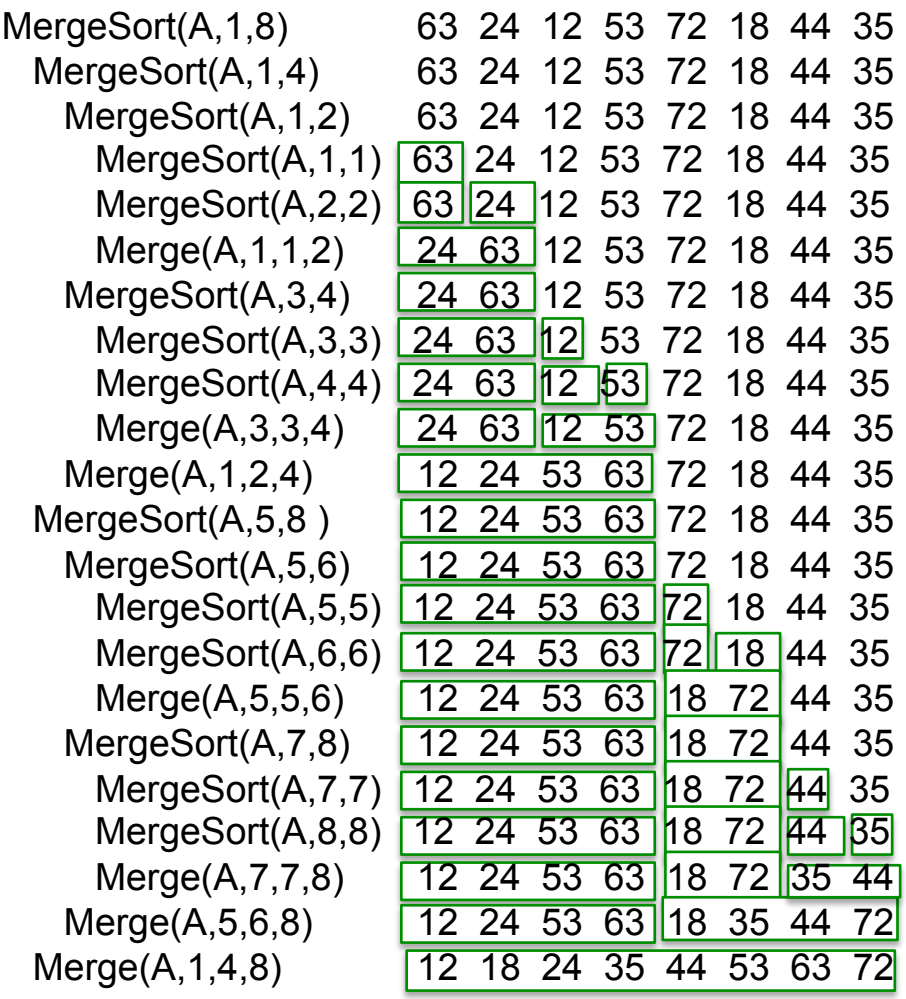
mit Mastertheorem folgt: $O(n \log(n))$

Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT

Merge-Sort

Beispiel:



Sortieren: Heapsort

TECHNISCHE UNIVERSITÄT DARMSTADT

Heapsort:

1. build-Heap
2. for i := 1 to n do
3. tausche erstes und letztes Element im Heaparray
4. verringere die Heapgröße um 1
5. Heapify

- Aufwand: O(n log(n))

- erfahrungsgemäß ab ca. 600 Elementen bestbekannter Sortieralgorithmus, der auf Vergleichen und Vertauschen von Elementpaaren beruht.

- bei weniger Elementen ist meist Quicksort schneller

Sortieren: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

Quicksort

- basiert auf einer variablen Aufteilung des Eingabearrays
- wurde 1962 von Hoare entwickelt
- benötigt im Worst Case $\Omega(n^2)$ Vergleiche, im Mittel aber nur $O(n \log(n))$ Vergleiche und damit Vertauschungen, die den Aufwand bestimmen

Sortieren: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

Grobbeschreibung

1. Gegeben sei ein Array A mit n Komponenten
2. Wähle ein Pivotelement A[pivot]
3. Zerlege A in zwei Teilbereiche A[0],...A[k-1] und A[k+1],...A[n], so dass
   a) A[i] < A[pivot] für alle i=0,...,k-1
   b) A[i] = A[pivot]
   c) A[i] ≥ A[pivot] für alle i=k+1,...,n
4. Sofern ein Teilbereich aus mehr als einer Komponente besteht, so wende Quicksort rekursiv auf ihn an. (also evtl. auch für beide Teilbereiche)

Die Korrektheit ergibt sich leicht: Durch die Zerlegung in den Zeilen 3a-3c befinden sich alle Elemente, die kleiner als A[pivot] sind in einem Teilarray und alle die größer sind im anderen. Wenn beide Teilbereiche sortiert sind, ist also auch das Gesamtarray sortiert. Rest folgt induktiv.

Sortieren: Quicksort, Zerlegungsprozedur

TECHNISCHE UNIVERSITÄT DARMSTADT

pivot *:= i;* ***l*** *:= 1;* ***r*** *:= n;*
while ***l*** < ***r***

- Suche kleinstes ***l'*** ≥ ***l*** mit *A* [***l'***] ≥ *A* [***pivot***] und
  größtes ***r'***≤ ***r*** mit {*A* [***r'***] < *A* [***pivot***] oder ***r'*** = ***pivot***}
  /* **Beachte:** ***l'***≤ ***pivot*** und ***r'***≥ ***pivot*** */
- Falls ***l'*** = ***r'*** dann *STOP*
  /* **Beachte:** dann gilt ***l'*** = ***r'*** = ***pivot*** */
  sonst vertausche *A*[***l'***] und *A* [***r'***]
  /* **Beachte:** dabei kann das Split-Element verschoben werden */
  S := ***pivot****;*
- **if** *S ==* ***l'*** /* **Beachte:** dann ist *A*[***r'***] Split-Element */
  then ***pivot*** *:=* ***r'*** *;* ***r****:=* ***r'***
  else ***r****:=* ***r'*** *- 1 ;*
- **if** *S ==* ***r'*** /* **Beachte:** dann ist *A*[***l'***] Split-Element */
  then ***pivot*** *:=* ***l'*** *,* ***l*** *:=* ***l'***
  else ***l*** *:=* ***l'*** *+ 1*

Kosten: Vergleiche und Vertauschungen: maximal n-1

Beispiel für Schritt 2 (n=13, i=7)

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
15	**47**	**33**	**87**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**44**

Beispiel für Schritt 2 (n=13, i=7)

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
15	**47**	**33**	**87**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**44**

Beispiel für Schritt 2 (n=13, i=7)

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
15	**47**	**33**	**87**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**44**
15	**47**	**33**	**44**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**87**

Beispiel für Schritt 2 (n=13, i=7)

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
15	**47**	**33**	**87**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**44**
15	**47**	**33**	**44**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**87**
15	**47**	**33**	**44**	**27**	**17**	**53**	**76**	**83**	**2**	**53**	**98**	**87**

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel für Schritt 2 (n=13, i=7)

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
15	**47**	**33**	**87**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**44**
15	**47**	**33**	**44**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**87**
15	**47**	**33**	**44**	**27**	**17**	**53**	**76**	**83**	**2**	**53**	**98**	**87**
15	**47**	**33**	**44**	**27**	**17**	**2**	**76**	**83**	**53**	**53**	**98**	**87**

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel für Schritt 2 (n=13, i=7)

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
15	**47**	**33**	**87**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**44**
15	**47**	**33**	**44**	**98**	**17**	**53**	**76**	**83**	**2**	**53**	**27**	**87**
15	**47**	**33**	**44**	**27**	**17**	**53**	**76**	**83**	**2**	**53**	**98**	**87**
15	**47**	**33**	**44**	**27**	**17**	**2**	**76**	**83**	**53**	**53**	**98**	**87**
15	**47**	**33**	**44**	**27**	**17**	**2**	**53**	**83**	**76**	**53**	**98**	**87**

Sortieren: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

Im Worst-Case wird das Array der Länge n in zwei Teilbereiche der Länge 1 und n-1 aufgeteilt. Die Anzahl der Rekursionsaufrufe ist dann gleich n.
Da in jeder Rekursionsebene für einen Teilbereich der Länge r gerade r-1 Vergleiche statt finden, erhält man

$$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}=\Theta(n^2)$$

Vergleiche. Das ist schlecht.

Gerechter wird man bei der Analyse des Quicksort-Algorithmus, wenn man sich die durchschnittliche Laufzeit bei Gleichverteilung aller n! vielen möglichen Reihenfolgen der Elemente A[1],...,A[n] ansieht, anstatt die schlechtest mögliche Laufzeit.
„Gleichverteilung“ bedeutet, dass jede mögliche Permutation mit dem gleichen Gewicht 1/n! eingehen soll.

Sortieren: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

Sei Π die Menge aller Permutationen von 1,...,n. Für $\pi \in \Pi$ sei C(π) die Anzahl von Vergleichen, die Quicksort benötigt, um π zu sortieren. Dann ist die durchschnittliche Anzahl Vergleiche

$$V(n) = \frac{1}{n!} \sum_{\pi \in \Pi} C(\pi)$$

Im Folgenden schätzen wir V(n) nach oben ab.

Teile die Menge Π aller Permutationen in die Mengen $\Pi_1,...,\Pi_n$, wobei

$\Pi_k := \{\pi \in \Pi \mid$ Pivotelement hat Wert k$\}$

→ In so einer Menge Π_k ist also ein Element fixiert, alle anderen können in beliebiger Reihenfolge auftreten. Also ist

$$|\Pi_k| = (n-1)! \text{ für } k = 1,...,n$$

Sortieren: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

Für alle $\pi \in \Pi_k$ ergibt sich die Zerlegung in Quicksort so, dass zwei Teilarrays π_1 (von 1,2,...,k-1) und π_2 (von k+1,...,n) entstehen.

Die Anzahl der Vergleiche, mit denen π in π_1 und π_2 zerlegt wird ist ≤ n. Also:

$$C(\pi) \leq n + C(\pi_1) + C(\pi_2)$$

Summiert man über alle $\pi \in \Pi_k$, so ergibt sich wegen $|\Pi_k| = (n-1)!$

$$\sum_{\pi \in \Pi_k} C(\pi) \leq \sum_{\pi \in \Pi_k} n + \sum_{\pi \in \Pi_k} C(\pi_1) + \sum_{\pi \in \Pi_k} C(\pi_2) =: S_1 + S_2 + S_3$$

Sortieren: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

Hierbei ist $S_1 = \sum_{\pi \in \Pi_k} n = (n-1)! \cdot n = n!$

Wenn π alle Permutationen von Π_k durchläuft, entstehen bei π_1 alle Permutationen Von 1,...,k-1, und zwar jede (n-1)! / (k-1)! Mal, da Π_k ja insgesamt (n-1)! Permutationen enthält.

Also ist

$$S_2 = \frac{(n-1)!}{(k-1)!} \sum_{\pi_1 \text{ Permutationen von } 1,\ldots,k-1} C(\pi_1)$$

$$= (n-1)! \cdot V(k-1).$$

Analog gilt

$$S_3 = (n-1)! \cdot V(n-k)$$

Sortieren: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

Durch Zusammensetzen aller Gleichungen bzw. Ungleichungen erhalten wir:

$$V(n) = \frac{1}{n!}\sum_{\pi\in\Pi} C(\pi) = \frac{1}{n!}\sum_{k=1}^{n}\sum_{\pi\in\Pi_k} C(\pi)$$

$$\leq \frac{1}{n!}\sum_{k=1}^{n}[n!+(n-1)!\cdot V(k-1)+(n-1)!\cdot V(n-k)]$$

$$= \frac{n!}{n!}\sum_{k=1}^{n}1+\frac{(n-1)!}{n!}\sum_{k=1}^{n}V(k-1)+\frac{(n-1)!}{n!}\sum_{k=1}^{n}V(n-k)$$

$$= n+\frac{1}{n}\sum_{k=1}^{n}V(k-1)+\frac{1}{n}\sum_{k=1}^{n}V(k-1)$$

$$= n+\frac{2}{n}\sum_{k=1}^{n}V(k-1)$$

Sortieren: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

Beachtet man noch die Anfangswerte der Rekursionsgleichung, so gilt:

$$V(0) = V(1) = 0, V(2) = 1,$$

$$V(n) \leq n + \frac{2}{n} \sum_{k=2}^{n-1} V(k), \quad \textit{für } n \geq 2$$

Die Lösung der Rekursionsgleichung ergibt

$$V(n) \leq 2 \cdot n \cdot \ln(n) = O(n \log(n))$$

Da die Anzahl der Vergleiche auch die Anzahl der Vertauschungen bestimmt, und die Anzahl Vertauschungen plus die Anzahl der Vergleiche die Laufzeit bestimmen, läuft der Quicksort-Algorithmus in Laufzeit O(n log(n)).

Sortieren: Bucket Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Bisher hatten unsere Sortieralgorithmen eine Laufzeit von O(n log(n)).
Geht das schneller?

Betrachte folgenden Sortieralgorithmus

Einschränkung: $a_1, ..., a_n \in \{1, ..., M\}$
Nutze Array L [1:M] von linearen Listen.

BucketSort (Eingabe $a_1, ..., a_n \in \{1, ..., M\}$)
zu Beginn: $L[i]$ ist leer für alle $i = 1, ..., M$.

(1) Für $j = 1, ..., n$:
- Hänge a_j an Liste $L[a_j]$ an.

(2) Hänge die Listen $L[1], ..., L[M]$ zur Liste L hintereinander.

(3) Durchlaufe L von vorne nach hinten und gebe die gelesenen Werte aus.

Zeit: O (n + M)
Platz: O (n + M)

Sortieren: Bucket Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Bisher hatten unsere Sortieralgorithmen eine Laufzeit von O(n log(n)).
Geht das schneller?

Antwort 1: ja, wenn man Sortieralgorithmen entwirft, die nicht auf paarweisen Schlüsselvergleichen beruhen. Das erfordert spezielle Annahmen, wie z.B. beim Bucket-Sort

Antwort 2: nein, wenn man sich auf Algorithmen beschränkt, die paarweise Schlüssel vergleichen. In diesem Sinne sind also Heapsort und Mergesort Optimale Algorithmen.

Sortieren: untere Schranke

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz: Jeder deterministische Sortieralgorithmus, der auf paarweisen Vergleichen von Schlüsseln basiert, braucht zum Sortieren eines n-elementigen Arrays Sowohl im Worst-Case also auch im Mittel (bei Gleichverteilung) $\Omega(n \log(n))$ Vergleiche.

Bemerkungen dazu:

- Der Satz zeigt, dass Mergesort und Heapsort bzgl. Der Größenordnung ihrer Laufzeiten optimal sind, und dass Laufzeitunterschiede lediglich den O-Konstanten zuzuschreiben sind.
- Man beachte den Unterschied zu den bisherigen O(..) Abschätzungen für ein Problem:
  - Jene erhielten wir, indem ein **konkreter** Algorithmus, der das Problem löst, analysiert wurde.
  - Die im Satz formulierte $\Omega(...)$-Abschätzung bezieht sich jedoch auf **alle möglichen** Sortierverfahren (bekannte und unbekannte)!

Sortieren: untere Schranke

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweisidee:

- für ein Array mit n Elementen gibt es n! verschiedene Anordnungen.

- wenn man ein Array nur durch paarweise Vergleiche und paarweise Vertauschungen sortiert, lassen sich die Datenbewegungen mit protokollieren und man bekommt heraus, welche Permutation man auf die Ausgangsfolge anwenden muss, um eine sortierte Folge zu erhalten.

- pro Vergleichsabfrage läßt sich jeweils die Hälfte der noch verbleibenden Permutationsmöglichkeiten ausschließen.

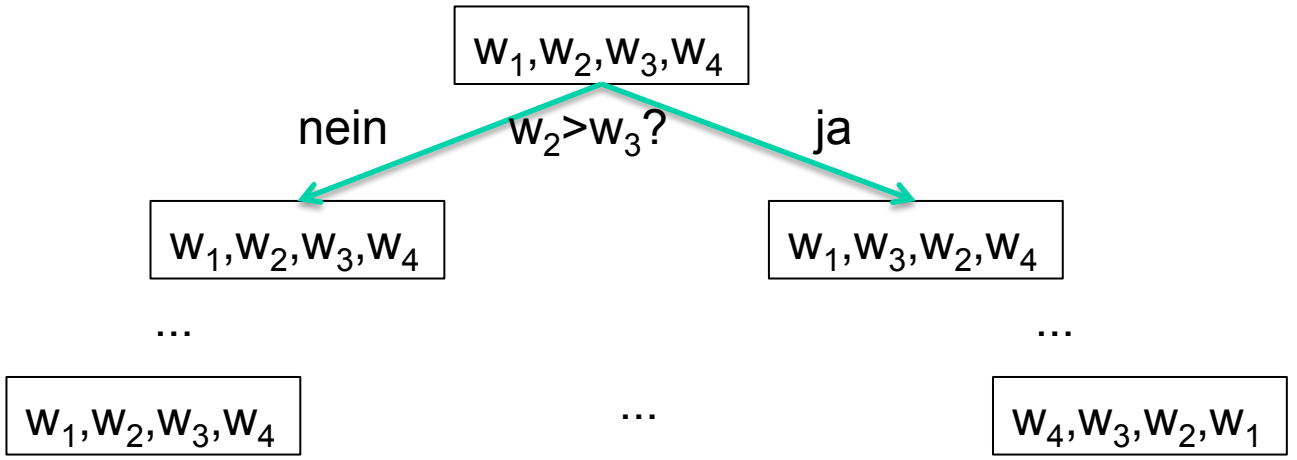
  Bsp.: Wenn man erfährt, dass A[5] >= A[6] ist, lassen sich alle Permutationen, die zu A[5] < A[6] führen, ausschliessen.

Sortieren: untere Schranke

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweisidee (Fortsetzung):

- betrachtet man nun einen Algorithmus, der durch Vergleiche und Vertauschungen die sortierte Folge erzeugt, bewegt sich dieser durch einen Baum von möglichen Ergebnissen und Abfragen:



- binärer Baum mit n! Blättern hat eine Höhe von $\log(n!) \in \Omega(n \log(n))$.

Das TSP mit Dreiecksungleichung

TECHNISCHE UNIVERSITÄT DARMSTADT

Geg.: vollständiger Graph G=(V,E), Kostenfunktion c: $V \times V \rightarrow \mathbb{Z}$, $k \in \mathbb{Z}$.
Zusätzlich gilt die Dreiecksungleichung:
$c(u,w) \leq c(u,v) + c(v,w)$
Frage: Gibt es eine Rundtour, die jeden Knoten genau einmal besucht mit Kosten höchstens k?

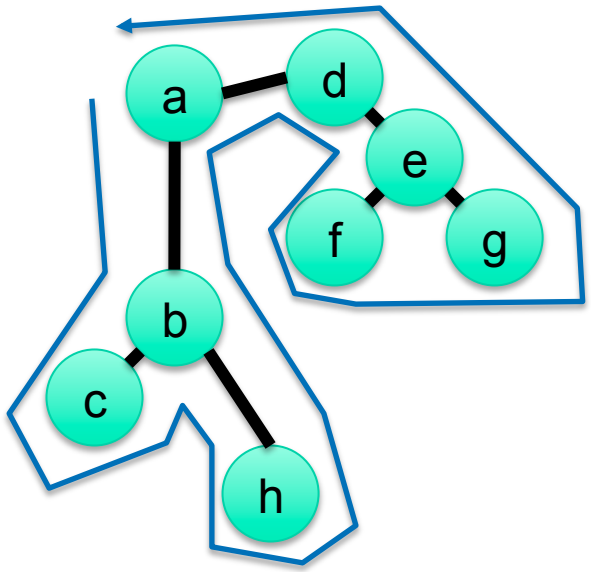
Das Problem ist NP-vollständig.

Betrachte folgenden Algorithmus:
1. wähle einen Knoten r als Startknoten
2. erzeuge einen minimalen Spannbaum mittels Algorithmus von Prim.
3. durchlaufe mittels Tiefensuche den minimalen Spannbaum

Das TSP mit Dreiecksungleichung

TECHNISCHE UNIVERSITÄT DARMSTADT

Bsp.:

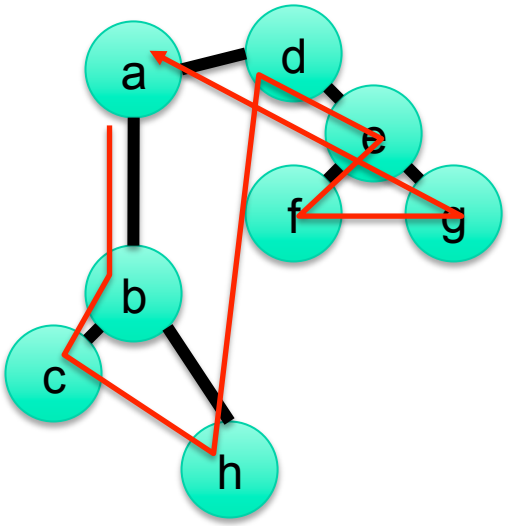


Beobachtung: Dieser Algorithmus läuft eine Tour, deren Kosten höchstens So gross sind, wie die das Doppelte der Kosten einer minimalen Rundreise. Allerdings ist unsere bisherige Tour noch keine Rundreise. Das ändern wir wie folgt:

Das TSP mit Dreiecksungleichung

TECHNISCHE UNIVERSITÄT DARMSTADT

Bsp.:



Beobachtung: Wenn ein Knoten auf unserer Tour bereits besucht wurde, besuche diesen nicht nochmal, sondern gehe zum nächsten auf der Tour liegenden Knoten direkt hin. Kürze also ab. Die neue Tour ist eine Rundtour, und ihre Kosten sind nicht größer als die Kosten der Tour, die zweimal den minimalen Spannbaum abläuft.

Fazit: Wir haben einen Näherungsalgorithmus mit Güte 2 gefunden.

Das TSP mit Dreiecksungleichung
Branch and Bound

TECHNISCHE UNIVERSITÄT DARMSTADT

Vermutlich gibt es keinen effizienten Algorithmus für das Problem. Aber, wir können verschiedene Möglichkeiten, Kanten zu Kreisen zu fügen ausprobieren und bewerten. Bekannt sei, dass es eine Tour mit Kosten y gibt. $x_{(a,b)}=1$ bedeute, Kante (a,b) wird hinzugenommen, $c_{(a,b)}$ seien die Kosten der Kante (a,b).