Algorithmen & Komplexität

Angelika Steger
Institut für Theoretische Informatik

steger@inf.ethz.ch

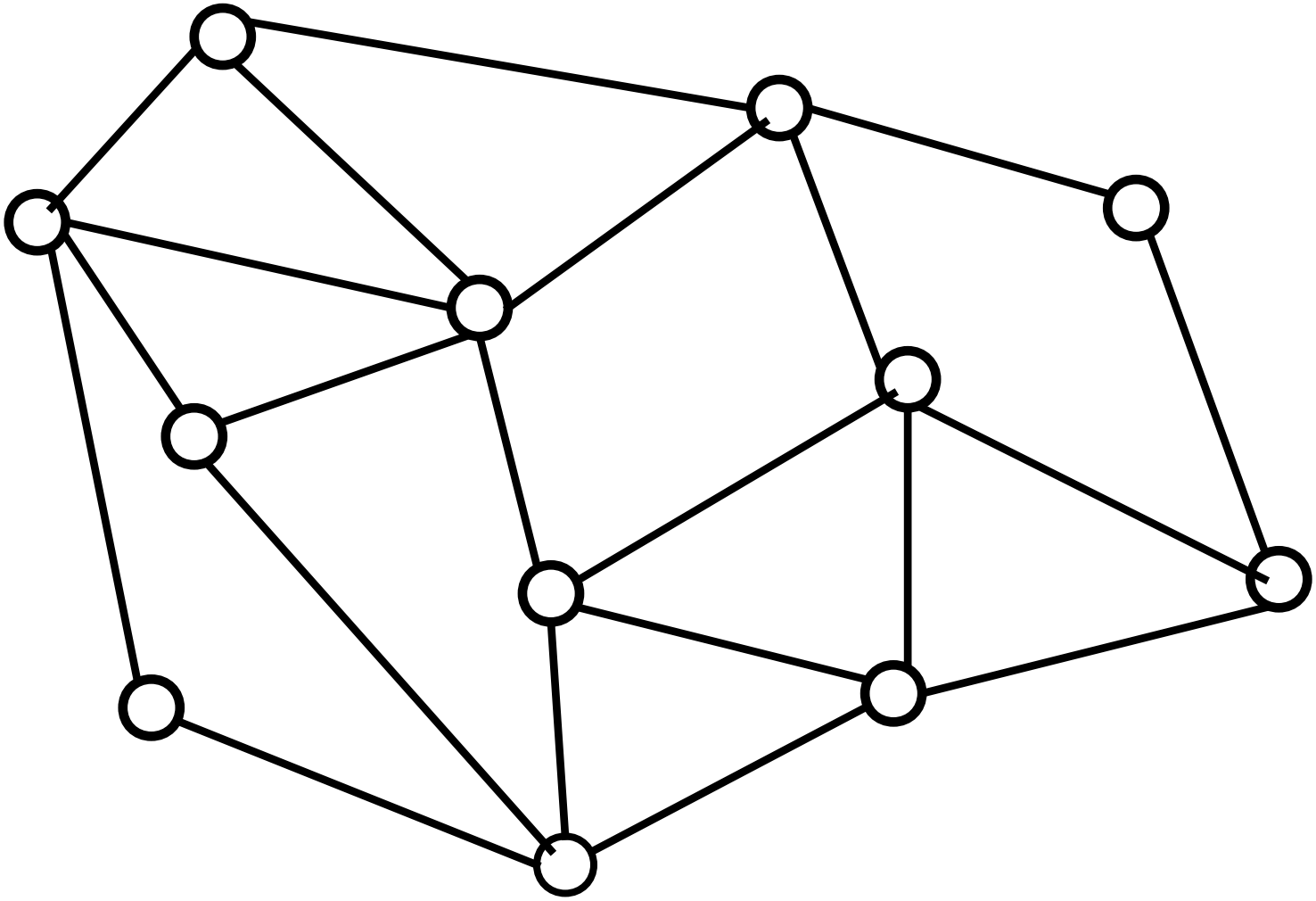
Breitensuche, Tiefensuche

Wir besprechen nun zwei grundlegende Verfahren, alle Knoten eines Graphen zu durchlaufen

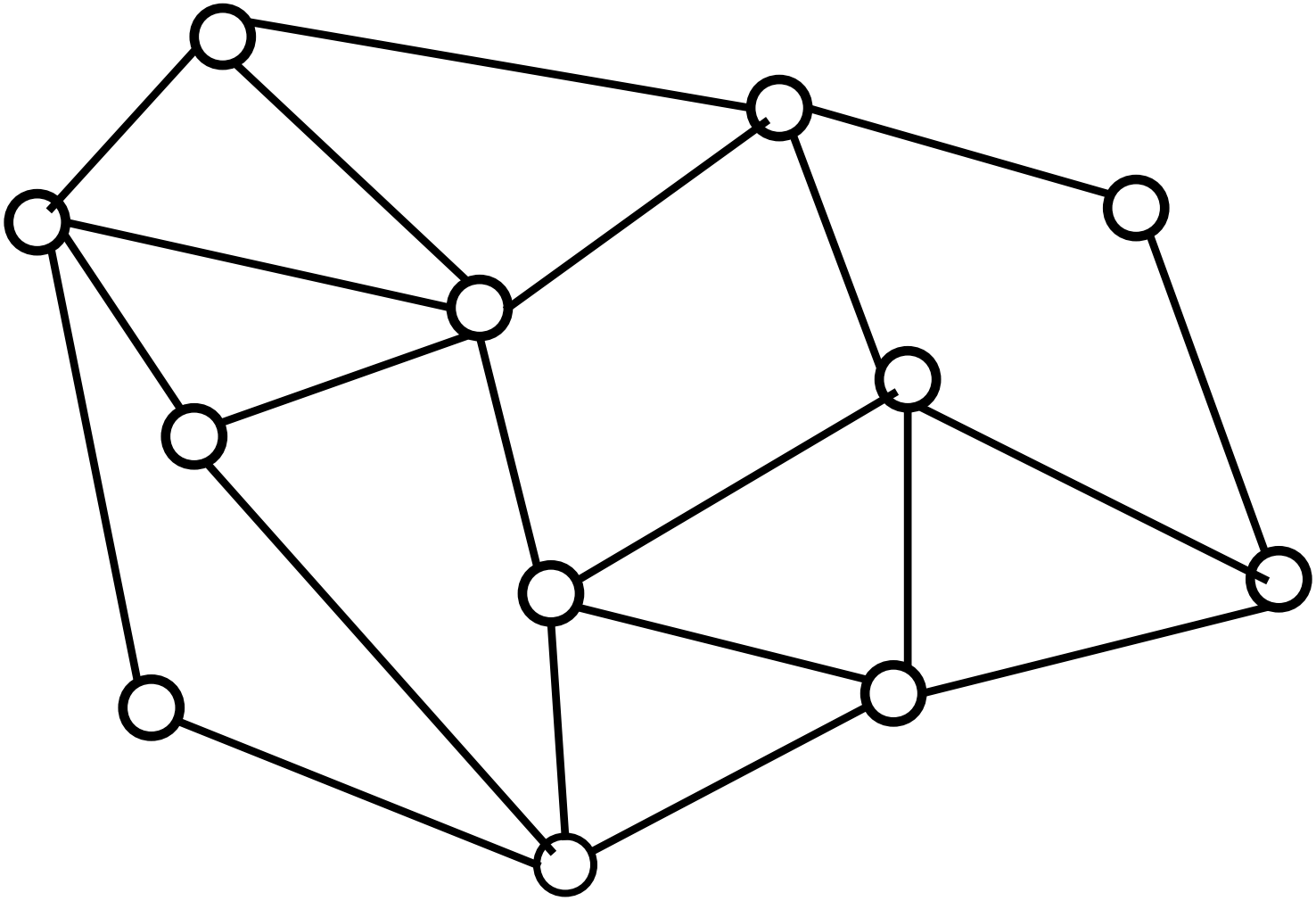
Breitensuche	Tiefensuche
(„*breadth first search*“, BFS)	(„*depth first search*“, DFS)

⇒ wichtige Bausteine von „fortgeschrittenen“ Graphenalgorithmen

BFS



DFS



Breitensuche, Tiefensuche

Wir besprechen nun zwei grundlegende Verfahren, alle Knoten eines Graphen zu durchlaufen

Breitensuche („*breadth first search*“, BFS)

Tiefensuche („*depth first search*“, DFS)

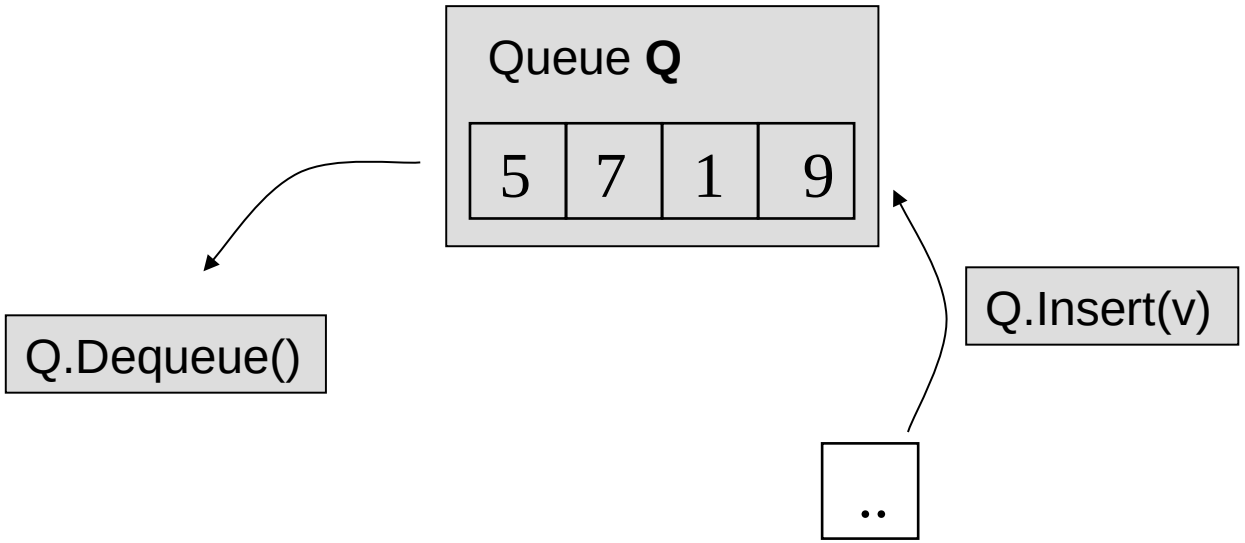
Datenstrukturen:

queue

stack

Datenstruktur „Queue“

- Queue (dt. Warteschlange)
- FIFO („*first in first out*“)



Breitensuche

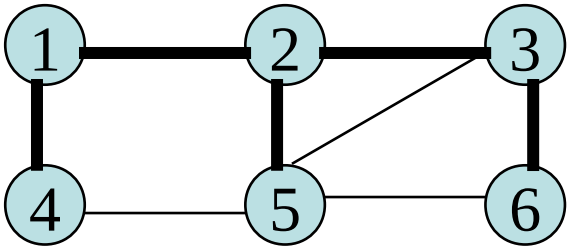
Input: Graph G=(V,E), Startknoten s∈V

Output: Felder d[v], pred[v] für v∈V

```
Setze d[v] = ∞, pred[v] = nil   ∀v∈V („unbesucht“)
d[s] = 0
Q.Insert(s)
while not Q.IsEmpty()
     v ← Q.Dequeue()
     for all u∈ Γ(v)
        if d[u]=∞ then
           d[u] = d[v]+1
           pred[u] = v
           Q.Insert(u)
         end if
     end for
```

Breitensuche - Beispiel

Startknoten: s=1



Init

d[1] = 0
d[2] = ∞
d[3] = ∞
d[4] = ∞
d[5] = ∞
d[6] = ∞

Q=(1)

①

d[1] = 0
d[2] = 1
d[3] = ∞
d[4] = 1
d[5] = ∞
d[6] = ∞

Q=(2,4)

②

d[1] = 0
d[2] = 1
d[3] = 2
d[4] = 1
d[5] = 2
d[6] = ∞

Q=(4,3,5)

④

d[1] = 0
d[2] = 1
d[3] = 2
d[4] = 1
d[5] = 2
d[6] = ∞

Q=(3,5)

③

d[1] = 0
d[2] = 1
d[3] = 2
d[4] = 1
d[5] = 2
d[6] = 3

Q=(5,6)

Breitensuche

Satz: G=(V,E), gegeben als Adjazenzlisten, Startknoten s.

Dann gilt:

- Die Breitensuche hat eine Laufzeit von $O(|V|+|E|)$.
- d[v] ist die Länge eines kürzesten s-v-Pfades bzw. $d[v]= \infty$, wenn kein solcher existiert.
- Falls G zusammenhängend, bilden die Kanten $\{ \{v,pred[v]\} \mid v \in V \setminus s \}$ einen Spannbaum T von G, mit der Eigenschaft, dass für alle $v \in V$ der eindeutige s-v-Pfad in T ein kürzester s-v-Pfad in G ist.

Beweis:

Laufzeit: ✓

Sei $v \in V \setminus s$.

- Es gilt $d[v] = d[pred[v]] + 1 \geq 1$, $\forall v \in V \setminus s$ mit $d[v] \neq \infty$.
- Starte in v und laufe entlang $\{u, pred[u]\}$ Kanten
  - Wert $d[u]$ nimmt entlang jeder Kante um 1 ab
  - Es gilt $d[s] = 0$
- Also: v-s-Pfad der Länge $d[v]$
- Noch zu zeigen: Es gibt keinen kürzeren v-s-Pfad.

Beweis:

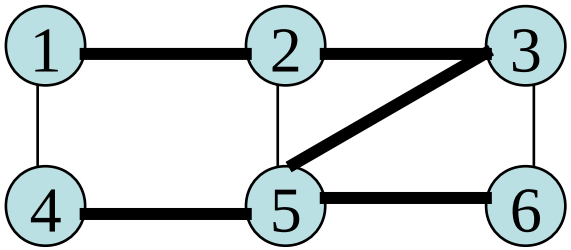
Sei $v \in V \setminus s$.

- Sei $\{w_1,w_2\} \in E$. Dann gilt $d[w_1] \leq d[w_2] + 1$.
  - Zu jedem Zeitpunkt unterscheiden sich die d-Werte von Elementen in der Queue Q um höchstens 1, und die kleineren Werte stehen am Anfang von Q
- Sei $P=(s = u_0, u_1,\ldots,u_k = v)$ ein beliebiger s-v-Pfad. Es gilt
  $d[v] = d[u_k] \leq d[u_{k-1}] + 1 \leq d[u_{k-2}] + 2 \leq \ldots \leq d[s] + k = k$
  D.h. die Länge von P ist mindestens $d[v]$.
- Da G zshgd, gilt $d[v] \neq \infty \quad \forall v \in V$
- Der aus den Kanten $\{v,pred[v]\}$ bestehende Graph besitzt also $n-1$ Kanten, ist also ein Baum, und damit ein Spannbaum von G.

Breitensuche und Zusammenhang

- Am Ende der BFS: $\{v \in V \mid d[v] < \infty \}$ bilden Zusammenhangskomponente.
- Weitere Komponenten: Iterativ mit neuen Startknoten.
- Laufzeit: $O(|V|+|E|)$

Tiefensuche - Beispiel

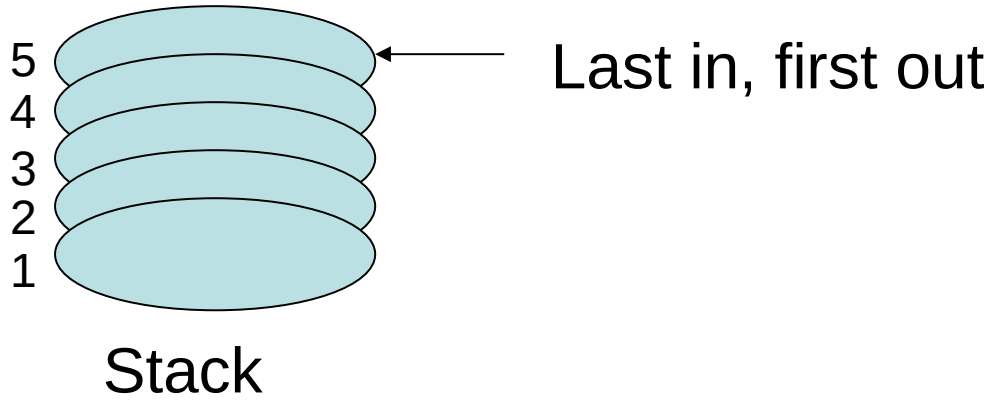


Reihenfolge: 1 2 3 5 4 6

Datenstruktur „Stack“ (Keller)

- LIFO - Queue („*last in first out*“)
- Operationen: Push(v) und Pop()

für DFS



F.L. Bauer, K. Samelson: *Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens*, Deutsches Patentamt, Auslegeschrift 1094019, B441221X/42m, 1957.

IEEE Computer Pioneer Award (1988) – „Für die Erfindung des Kellerprinzips“

Tiefensuche

Input: Graph G=(V,E), Startknoten s∈V

Output: Feld pred[v] für v∈V

```
Setze pred[v] = nil für alle v∈V („unbesucht“)
v = s
repeat
   if (∃u ∈ Γ(v) mit pred[u]=nil)  then
      Stack.Push(v)
      pred[u]=v
      v = u
   else if not Stack.IsEmpty()
      v = Stack.Pop()
   else
      v = nil
   end if
until v = nil
```

Tiefensuche

Satz: $G=(V,E)$, gegeben als Adjazenzlisten, Startknoten s.

Dann gilt:

- Die Tiefensuche hat eine Laufzeit von $O(|V|+|E|)$.
- Falls G zusammenhängend, bilden die Kanten $\{ \{v,pred[v]\} \mid v \in V \setminus s \}$ einen Spannbaum von G.

Bemerkung: Diverse Modifikationen von DFS zur Lösung anderer Graphenprobleme.

Weitere Beispiele Effizienter Algorithmen

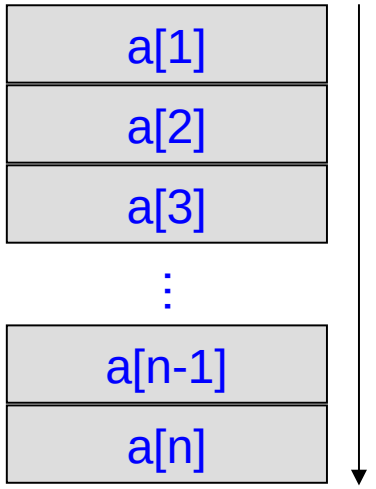
Sequentielle Suche

Gegeben: Array a[1..n]

Suche in a nach Element x

Ohne weitere Zusatzinformationen:

Sequentielle Suche

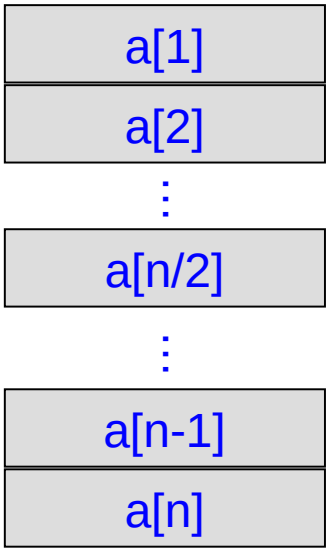


Laufzeit:

n Schritte im worst-case

Binäre Suche

Angenommen: Array a ist sortiert



Vergleiche x mit a[n/2]

$x \leq a[n/2] \Rightarrow$ x liegt in a[1..n/2]

$x \geq a[n/2] \Rightarrow$ x liegt in a[n/2..n]

Binäre Suche

Rekursive Programmierung:

```
function binarySearch (a,x,l,r)
  if (r = l)
    if (x = a[l])
      return l
    else
      return not found
  mid = floor((l+r)/2)
  if (x ≤ a[mid])
    binarySearch(a, x, l, mid)
  else
    binarySearch(a, x, mid+1, r)
```

Laufzeit Binäre Suche

Satz: Für die maximale Anzahl Vergleiche für die binäre Suche in $a[1..n]$ gilt

$$B_n = B_{\lceil n/2 \rceil} + 1 \text{ für } n \geq 2, \text{ und } B_1 = 1$$

Die Lösung dieser Rekursion lautet

$$B_n = \lceil \log_2 n \rceil + 1$$

Beweis:

- Beide Hälften von $a[1..n]$ sind höchstens $\lceil n/2 \rceil$ gross
- Es gilt die Monotonie: $B_m \leq B_n \ \forall m \leq n$
- Induktiv gilt $B_{2k} = k + 1 = \lceil \log_2(2^k) \rceil + 1$, und zusammen mit der Monotonie folgt die Behauptung.

Anzahl Vergleiche bei Binärer Suche

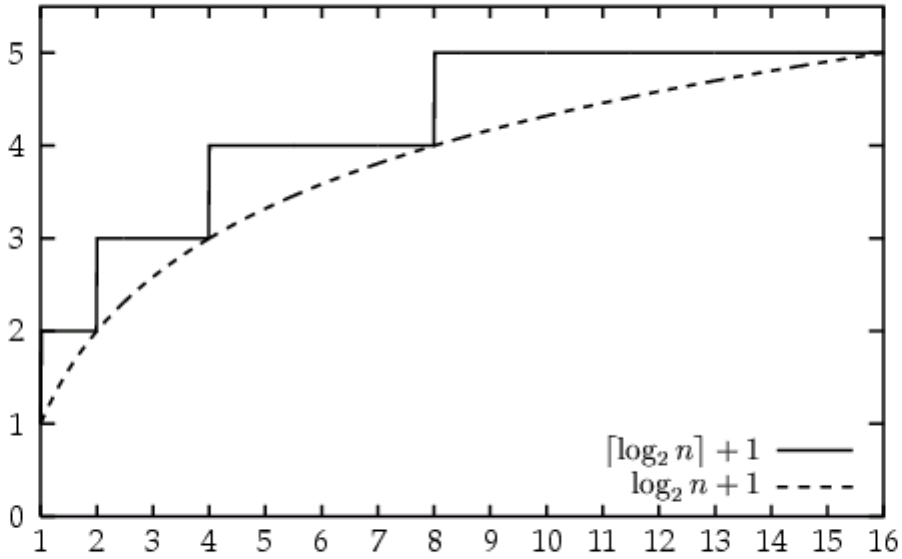


Abbildung 2.1: Anzahl Vergleiche $B_n = \lceil \log_2 n \rceil + 1$ bei einer binären Suche.

Für asymptotische Analyse:

Laufzeit von Binärer Suche ist O(log n)

2.2 Sortieralgorithmen

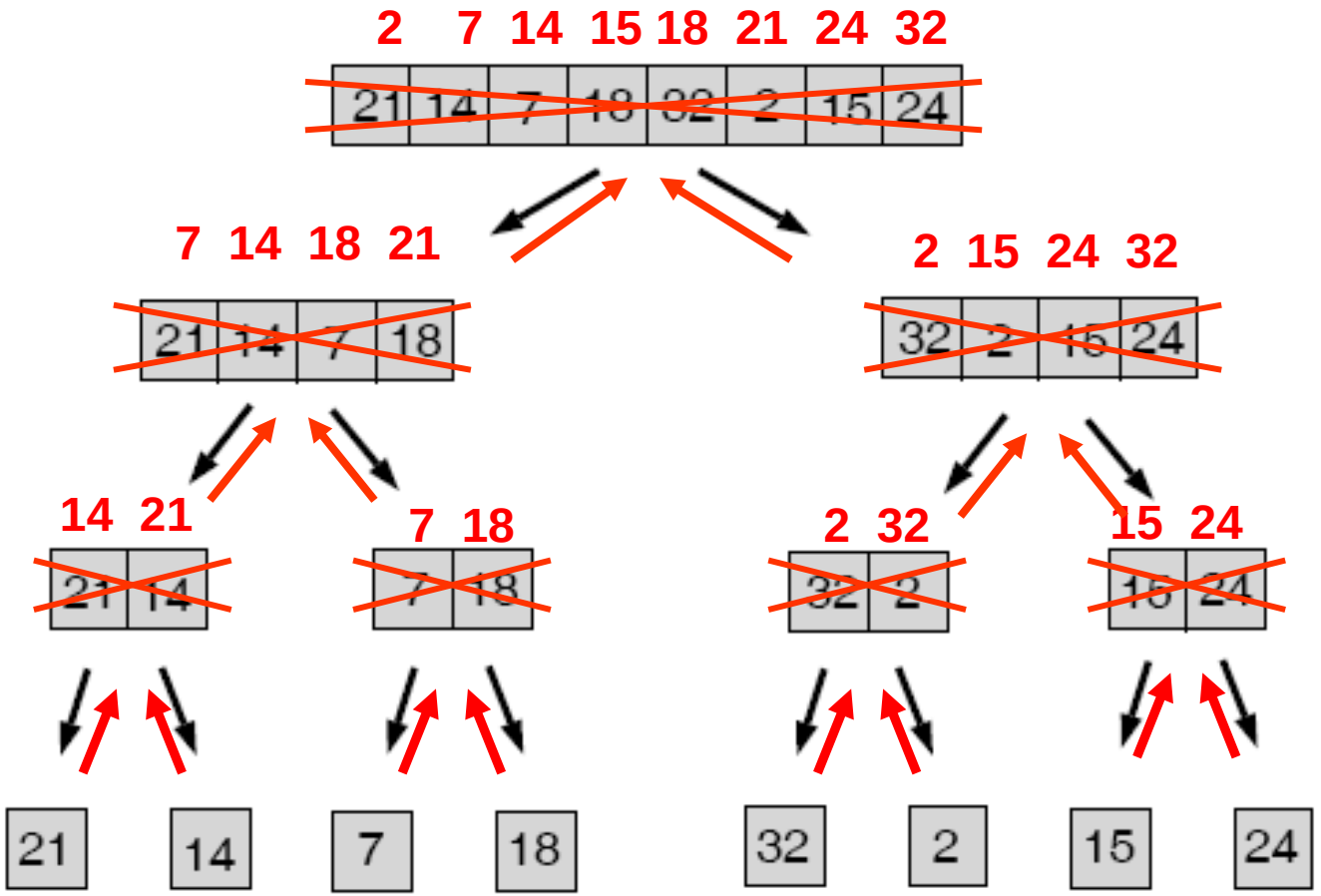
Gegeben: Zahlen $a_1,\ldots,a_n$;
Aufgabe: Sortiere sie!

Bemerkungen:

- Wir schätzen statt der Laufzeit meist die Anzahl der benötigten Vergleiche zwischen Elementen der Eingabe ab.

- *Wichtig:* Wir zählen hier wirklich nur die Anzahl Vergleiche zwischen Elementen der Eingabe (so genannte Schlüsselvergleiche) und nicht Vergleiche der Form „$i \leq n$“, die bspw. benötigt werden, um festzustellen, ob eine Laufvariable die vorgegebene Grösse eines Feldes überschreitet.

MergeSort - Beispiel



MergeSort - Analyse

C_n := Anzahl (Schlüssel-)Vergleiche, die MergeSort höchstens durchführt, wenn ein Feld der Grösse n sortiert wird

$n = 2^k$:

$$C_1 = 0$$

$$C_{2k} = 2 \cdot C_{2k-1} + 2 \cdot 2^{k-1} - 1$$

MergeSort – Analyse (2)

$n = 2^k$: $C_1 = 0$

$$C_{2k} = 2 \cdot C_{2k-1} + 2 \cdot 2^{k-1} - 1$$

$$\leq 2 \cdot C_{2k-1} + 2^k$$

$$\leq 2 \cdot (2 \cdot C_{2k-2} + 2^{k-1}) + 2^k$$

$$\leq \ldots \leq 2^k \cdot C_1 + k \cdot 2^k = k \cdot 2^k$$

n beliebig:

$$C_n \leq C_{2\lceil \log_2 n \rceil} \leq \lceil \log_2 n \rceil \cdot 2^{\lceil \log_2 n \rceil} \leq 2n \cdot (\log_2 n + 1)$$

MergeSort – Analyse (3)

Satz:

Um ein Feld der Grösse n zu sortieren genügen $2n \cdot (\log_2 n + 1)$ Vergleiche bzw. Laufzeit $O(n \cdot \log_2 n) = O(n \log(n))$.

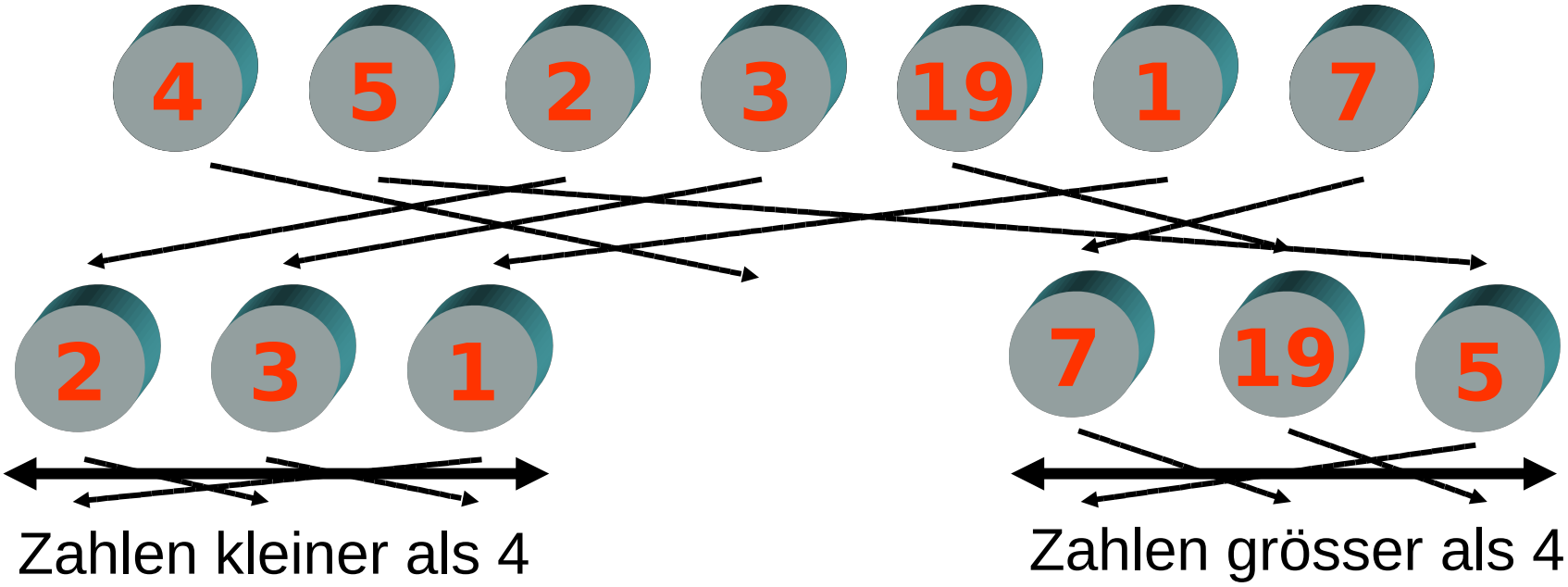
Bemerkung:

Genauere Analyse zeigt, dass sogar

$$C_n \leq n \cdot \lfloor \log_2 n \rfloor + 2n$$

gilt.

QuickSort - Beispiel



- Wähle erstes Element und teile übrige Elemente in „kleiner“ und „grösser“
- Wende analoges Verfahren auf die

- Test auf einem handelsüblichen Rechner:
- 500.000 Zahlen werden in 0.4 sec sortiert

QuickSort

- Wir testen unseren Algorithmus an einer realen Aufgabe.
- Dazu bitten wir einen Mathematiker ☺ uns 500.000 Zahlen in einer beliebigen, von ihm/ihr gewählten Reihenfolge zu liefern.
- M. liefert uns die Zahlen

  1, 2, 3, 4, 5, … , 499.999, 500.000
- QuickSort rechnet … und rechnet …

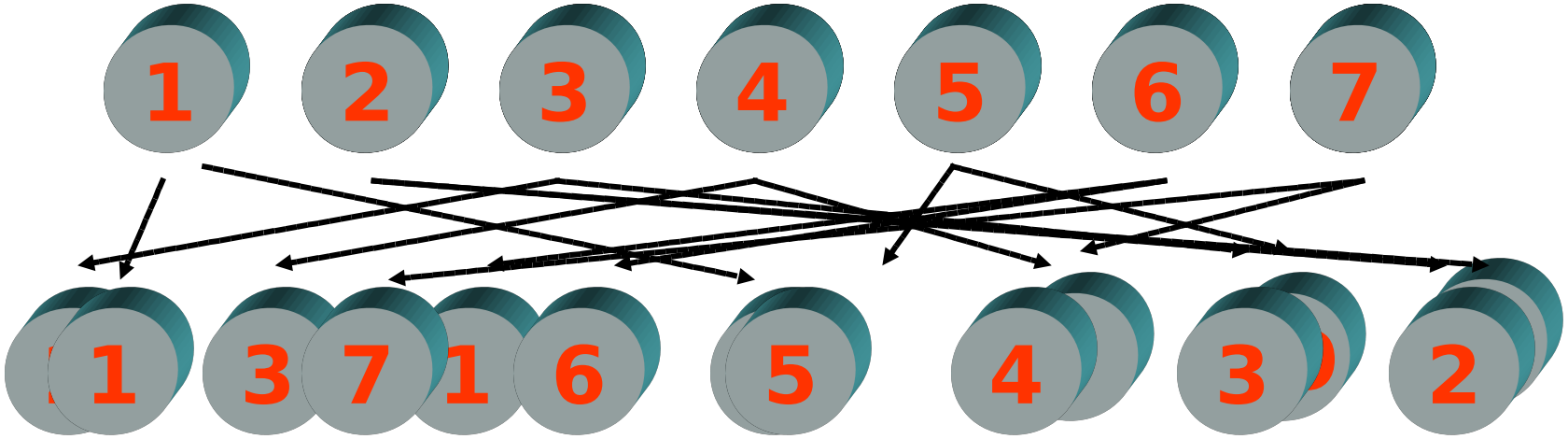
  und rechnet … und rechnet …

  und rechnet … und rechnet …

  und stürzt ab.

Idee des QuickSort-Algorithmus:

Die zu sortierenden Elemente werden in jedem Schritt in zwei etwa gleich große Mengen aufgeteilt: **für Testeingabe gilt aber**



QuickSort - Analyse

C_n := Anzahl (Schlüssel-)Vergleiche, die QuickSort höchstens durchführt, wenn ein Feld der Grösse n sortiert wird

$$C_n = \max_{0 \le k < n} (n-1 + C_k + C_{n-k-1})$$

Und somit:

$$C_n \ge n-1 + C_{n-1} \ge (n-1) + (n-2) + C_{n-2}$$
$$\ge \ldots \ge (n-1) + (n-2) + \ldots + 1 + C_1$$
$$= \tfrac{1}{2}\, n(n-1)$$

QuickSort – Analyse (2)

Satz:

Um ein Feld der Grösse n zu sortieren benötigt QuickSort im schlimmsten Fall $\frac{1}{2}\, n(n-1)$ Vergleiche bzw. Laufzeit $\Theta(n^2)$.

Bemerkung:

- Trotzdem gilt QuickSort als ein (in der Praxis und leicht abgewandelt) als sehr effizientes Sortierverfahren.
- Mathematisch exakt kann man dies begründen indem man die erwartete Laufzeit betrachtet, wobei der Erwartungswert über alle n! Reihenfolgen der Zahlen 1, …,n gebildet wird.

  Man erhält: erwartete Laufzeit = $O(n \log n)$

Sortieren – Untere Schranken

Satz:

Jeder Algorithmus, der n Elemente mit Hilfe von „if“-Abfragen sortiert, benötigt mindestens $\Theta(n \log n)$ Vergleiche.

Beweisidee:

Gegenspieler-Beweis …