Heapsort

Prioritätswarteschlangen

Datenstruktur zum Speichern einer Menge S, deren Elemente einen Schlüssel zugeordnet haben.
Max-PWS: großer Schlüssel heißt hohe Priorität
Operationen: Einfügen, Maximum von S, Entfernen des Maximums, Erhöhen eines Schlüssels

Algorithm PWS-MAX(A)

```
1: return A[1]
```

Algorithm EXTRACT-MAX(A)

```
1: heap_size:= |A|
2: if heap_size < 1 then error
3: end if
4: max := A[1]
5: A[1] ⟷ A[max]
6: heap_size := heap_size−1
7: MAX-HEAPIFY(A, 1)
8: return max
```

Heapsort

Algorithm INCREASE-KEY(A, i, p)

1: **if** $p < A[i]$ **then** error
2: **end if**
3: $A[i] := p$
4: **while** $i > 1$ **and** $A[\text{PARENT}(i)] < A[i]$ **do**
5: $\quad A[i] \longleftrightarrow A[\text{PARENT}(i)]$
6: $\quad i := \text{PARENT}(i)$
7: **end while**

Algorithm PWS-INSERT(A, S)

1: heap_size:= $|A| + 1$
2: $A[\text{heap\_size}] := -\infty$
3: INCREASE-KEY(A, heap_size, S)

Heapsort

Kosten

- PWS-MAX: $\Theta(1)$;
- EXTRACT-MAX: Schleifenrumpf wie bei Heapsort, daher $\mathcal{O}(\log n)$ wegen MAX-HEAPIFY;
- INCREASE-KEY: $\mathcal{O}(\log n)$ (Pfadlänge: $h(\text{Wurzel}) - h(A[i])$;
- PWS-INSERT: $\mathcal{O}(\log n)$.

7) Quicksort

Quicksort

In-place Sortierverfahren, in der Praxis häufig verwendet,
Divide&Conquer-Strategie

- Divide: $A[p..r]$ zerlegen: Wähle Pivot-Element $A[q]$ und bringe alle kleineren Elemente nach $A[p..q-1]$, alle größeren nach $A[q+1..r]$;
- Conquer: sortiere $A[p..q-1]$ und $A[q+1..r]$ mittels Quicksort;
- Merge: nichts zu tun.

Partitionierung: zB $x = A[r]$ als Pivot-Element wählen.

Quicksort

Algorithm QUICKSORT(A, p, r)

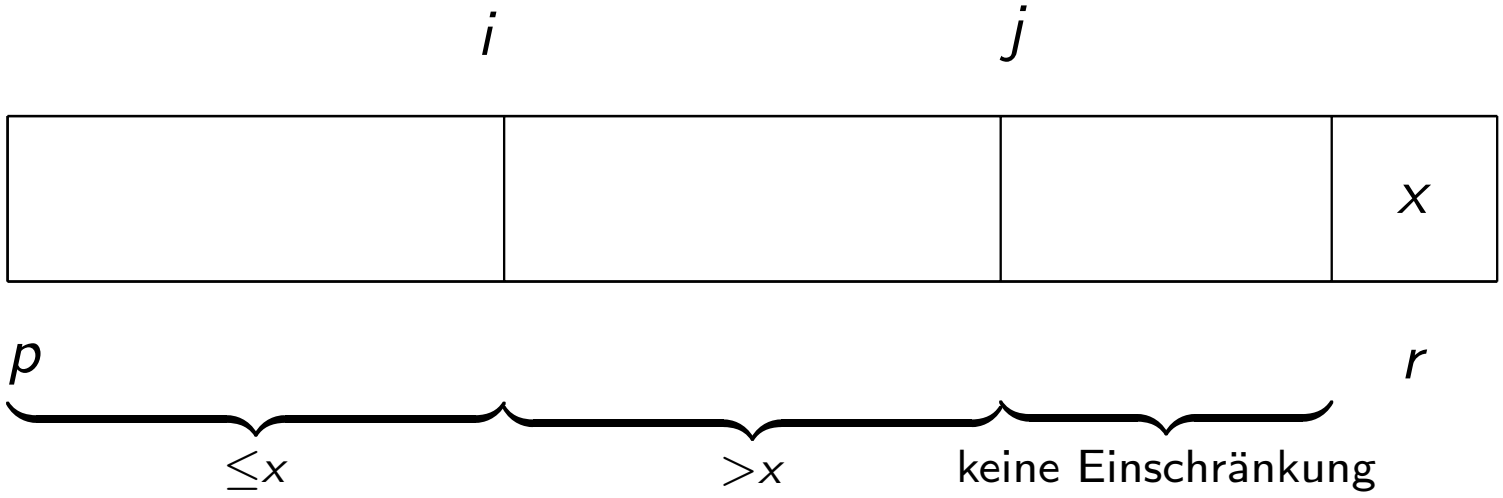
```
1: if p < r then
2:     q :=PARTITION(A, p, r)
3:     QUICKSORT(A, p, q − 1)
4:     QUICKSORT(A, q + 1, r)
5: end if
```

Algorithm PARTITION(A, p, r)

```
1: x = A[r]
2: i := p − 1
3: for j = p to r − 1 do
4:     if A[j] ≤ x then
5:         i := i + 1
6:         A[i] ⟷ A[j]
7:     end if
8: end for
9: A[i + 1] ⟷ A[r]
10: return i + 1
```

Quicksort

Partitionierung mit $x = A[r]$ als Pivot-Element erzeugt durch p, i, j, r vier Teilfelder:

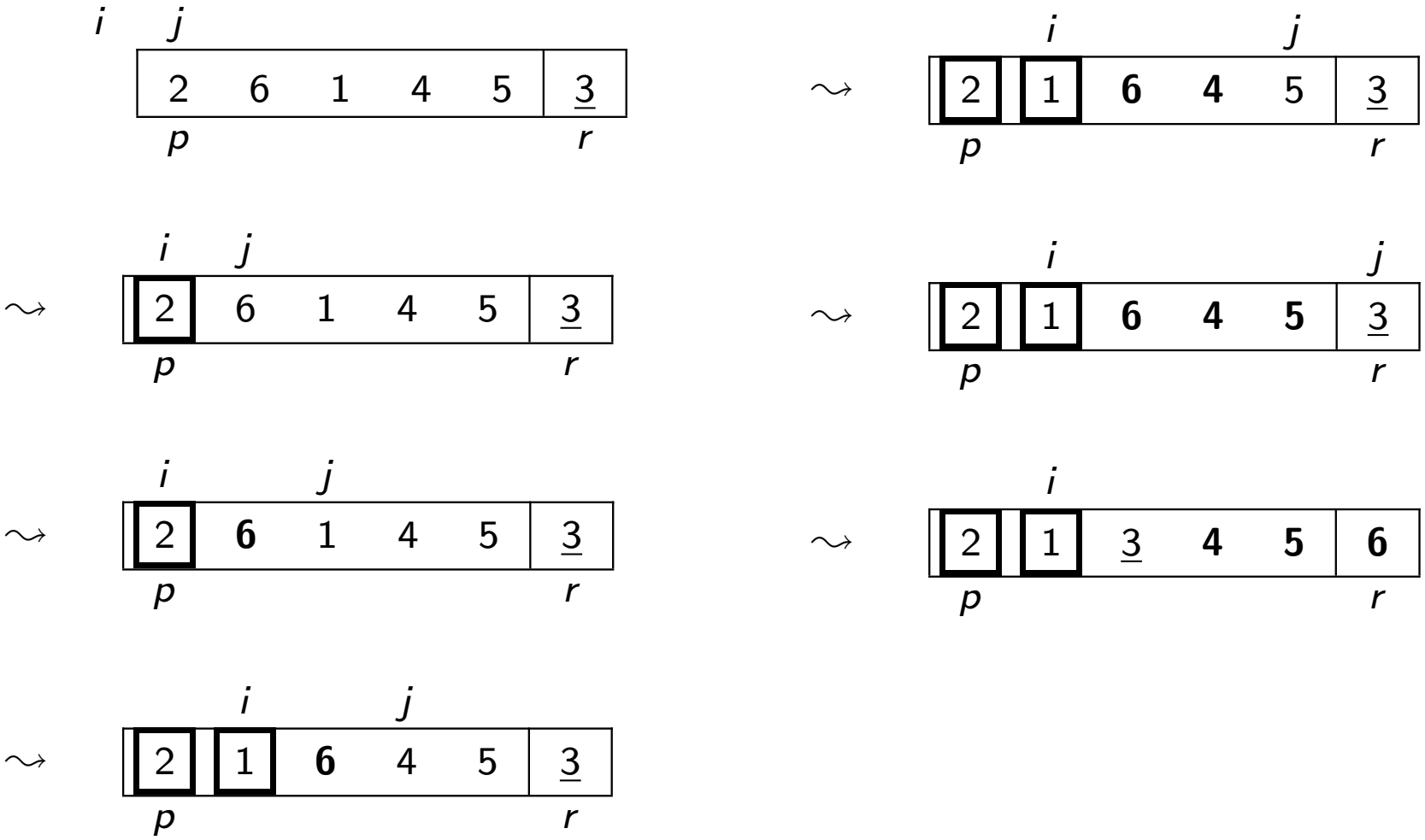


Schleifeninvariante!

Laufzeit: $n + 1$ Vertauschungen, n Vergleiche $\rightsquigarrow \mathcal{O}(n)$

Quicksort

Beispiel: $A = [2, 6, 1, 4, 5, 3]$, Pivot-Element: 3.



Analyse von Quicksort

Worst Case: Partitionierung erzeugt Felder der Größen $n-1$ und 0;

$$T(n) = T(n-1)+T(0)+\Theta(n) = T(n-1)+\Theta(n) \implies T(n) = \Theta(n^2);$$

tritt ein, wenn $A[1..n]$ sortiert (auf- oder absteigend)

Best Case:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \implies T(n) = \Theta(n \log n)$$

Average Case: Wir probieren Teilung im Verhältnis 9:1:

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n) \implies T(n) = \Theta(n \log n)$$

Das gilt auch, wenn Zerlegung im Verhältnis $kn : (1-k)n$.

Randomisiertes Quicksort

Nachteil in der Praxis: Datensätze oft teilsortiert

Randomisieren durch

1. Permutieren der Eingabe;
2. zufällige Wahl des Pivot-Elements.

Algorithm RANDOMIZED-QUICKSORT(A, p, r)

```
1: if p < r then
2:     q :=RANDOMIZED-PARTITION(A, p, r)
3:     RANDOMIZED-QUICKSORT(A, p, q − 1)
4:     RANDOMIZED-QUICKSORT(A, q + 1, r)
5: end if
```

Algorithm RANDOMIZED-PARTITION(A, p, r)

```
1: i :=RANDOM(p, r)
2: A[i] ⟷ A[r]
3: return PARTITION(A, p, r)
```

Quicksort

Analyse

Worst Case:

$$T(n) = \max_{1 \le q \le n} (T(q-1) + T(n-q)) + \Theta(n) \quad \Longrightarrow \quad T(n) = \Theta(n^2)$$

Best Case:

$$T(n) = \min_{1 \le q \le n} (T(q-1) + T(n-q)) + \Theta(n) \quad \Longrightarrow \quad T(n) = \Theta(n \log n)$$

Average Case: X = Anzahl der Schlüsselvergleiche
(1 Vergleich pro Iteration der **for**-Schleife von PARTITION)

Höchstens n Aufrufe von PARTITION
(jedes Pivot-Element nur einmal)

$\Longrightarrow \quad T(n) = \mathcal{O}(n + X)$,
denn PARTITION hat Aufwand
$\mathcal{O}(1) + \mathcal{O}(\#$ Iterationen der **for**-Schleife).

Quicksort

X ist Zufallsvariable
n Datenelemente $z_1 < z_2 < \cdots < z_n$, $\quad Z_{i,j} := \{z_i, z_{i+1}, \ldots, z_j\}$
$X_{i,j} = \mathbb{1}_{[\text{Vergleich } z_i : z_j]}$

Jedes Paar z_i, z_j wird höchstens einmal verglichen, da Vergleiche nur mit dem Pivot-Element erfolgen.

Daher gilt:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}.$$

$$\mathbb{E}X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}X_{i,j} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{P}\{z_i : z_j\}$$

$z_i : z_j$ passiert, wenn z_i oder z_j in $Z_{i,j}$ als erstes Pivot gewählt werden, denn andernfalls: z_i und z_j in verschiedenen Teilfeldern.

Quicksort

$$\mathbb{P}\{z_i : z_j\} = \frac{2}{j-i+1}$$

$$\begin{aligned}
\mathbb{E}X &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1} \\
&= \sum_{k=1}^{n-1}\frac{2}{k+1}\sum_{i=1}^{n-k}1 = \sum_{k=1}^{n-1}\frac{2}{k+1}(n+1-(k+1)) \\
&= (n+1)2(H_n-1) - 2(n-1) \\
&= 2(n+1)H_n - 4n \\
&\sim 2n\log n + \mathcal{O}(n)
\end{aligned}$$

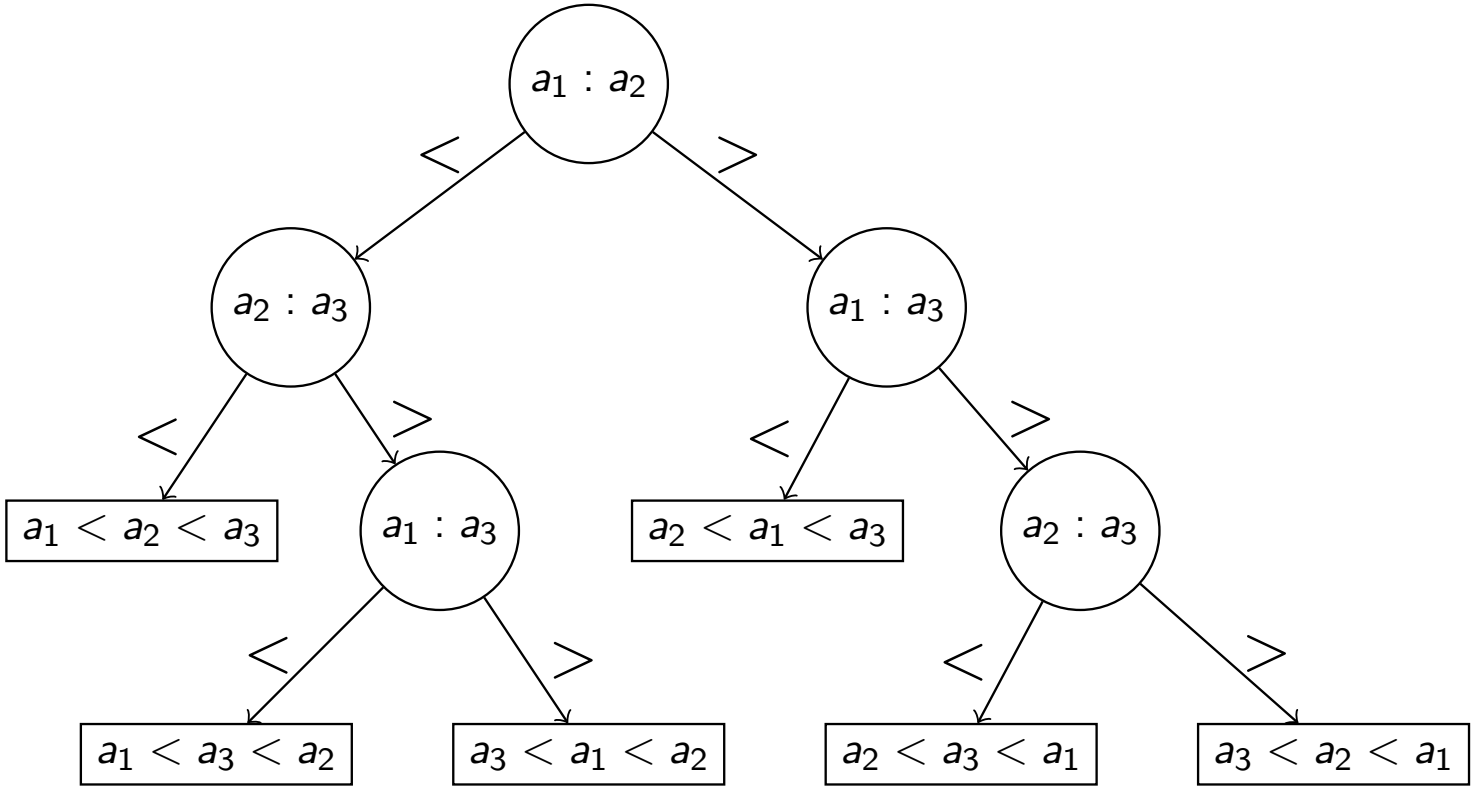
8) Sortieren in linearer Zeit

Sortieren in linearer Zeit

Untere Schranke für vergleichsbasierte Sortierverfahren

Annahme: $a_1, a_2, \ldots, a_n$ paarweise verschieden.
Daher gilt:

- Abfragen der Form $a_i = a_j$ unnötig;
- Abfragen der Form $a_i < a_j, a_i \leq a_j, a_i > a_j, a_i \geq a_j$ äquivalent.



Sortieren in linearer Zeit

- Algorithmus muss alle $n!$ Permutation erzeugen können, $\rightsquigarrow$ $n!$ Blätter;
- Längster Wurzel-Blatt-Pfad entspricht Worst-Case.

Satz

Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im Worst-Case.

Beweis: Ein Baum der Höhe h hat höchstens 2^h Blätter.
Daher muss folgendes gelten:

$$h \geq \log_2(n!) \sim n \log_2 n + \mathcal{O}(n).$$

Folgerung

Heap- und Mergesort sind asymptotisch optimal.

Sortieren in linearer Zeit

Sortieren durch Zählen

Annahme: $a_1, a_2, \ldots, a_n \in \{0, 1, \ldots, k\}$

- Strategie: Bestimme $|\{a_j | \, a_j < a_i\}|$ für jedes a_i.
- Eingabe: $A[1..n]$, Ausgabe: $B[1..n]$, Hilfsfeld $C[0..k]$.

Algorithm COUNTING-SORT(A, k)

```
 1: Seien B[1..n] und C[0..k] neue Datenfelder
 2: for i = 0 to k do
 3:     C[i] := 0
 4: end for
 5: for j = 1 to |A| do
 6:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 7: end for
 8: for i = 1 to k do
 9:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
10: end for
11: for j = |A| downto 1 do
12:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
13:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
14: end for
```

Sortieren in linearer Zeit

Kosten: Initialisierung: $\Theta(k)$,
eigentliches Sortieren: $\Theta(n), \Theta(k)$ bzw. $\Theta(n)$,
also insgesamt: $\Theta(k+n)$

Definition

Ein Algorithmus heißt stabil, wenn gleiche Elemente ihre Reihenfolge nicht ändern.

Bemerkung: Sortieren durch Zählen ist stabil (ebenso Insertion-Sort und Merge-Sort) im Gegensatz zu Heapsort und Quicksort.

Sortieren in linearer Zeit

6: **for** $j = 1$ **to** $|A|$ **do**
7: $\quad C[A[j]] := C[A[j]] + 1 \qquad \%\ C[i] = \#\{j : A[j] = i\}$
8: **end for**
9: **for** $i = 1$ **to** k **do**
10: $\quad C[i] := C[i] + C[i-1] \qquad \%\ C[i] = \#\{j : A[j] \leq i\}$
11: **end for**
12: **for** $j = |A|$ **downto** 1 **do**
13: $\quad B[C[A[j]]] := A[j] \qquad \%\ C[A[j]] = \#\{k : A[k] \leq A[j]\}$ =korrekte Position.
14: $\quad C[A[j]] := C[A[j]] - 1$ % $B \leftarrow A[j] \implies \#\{k : A[k] \leq A[j]\}$ um eins kleiner
15: **end for**

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

Sortieren in linearer Zeit

6: **for** $j = 1$ **to** $|A|$ **do**
7: $\quad C[A[j]] := C[A[j]] + 1 \qquad \%\ C[i] = \#\{j : A[j] = i\}$
8: **end for**
9: **for** $i = 1$ **to** k **do**
10: $\quad C[i] := C[i] + C[i-1] \qquad \%\ C[i] = \#\{j : A[j] \leq i\}$
11: **end for**
12: **for** $j = |A|$ **downto** 1 **do**
13: $\quad B[C[A[j]]] := A[j] \qquad \%\ C[A[j]] = \#\{k : A[k] \leq A[j]\}$ =korrekte Position.
14: $\quad C[A[j]] := C[A[j]] - 1$ % $B \leftarrow A[j] \implies \#\{k : A[k] \leq A[j]\}$ um eins kleiner
15: **end for**

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [2, 0, 2, 4, 0, 1]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [2, 0, 2, 4, 0, 1] \longrightarrow C = [2, 2, 4, 8, 8, 9]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [2, 2, 4, 8, 8, 9]$

$B = [\_, \_, \_, \_, \_, \_, \_, \_, \_]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [2, 2, 4, 8, 8, 9]$

$B = [\_, \_, \_, \_, \_, \_, \_, 3, \_]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [2, 2, 4, 7, 8, 9]$

$B = [\_, 0, \_, \_, \_, \_, \_, 3, \_]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [1, 2, 4, 7, 8, 9]$

$B = [\_, 0, \_, 2, \_, \_, \_, 3, \_]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [1, 2, 3, 7, 8, 9]$

$B = [\_, 0, \_, 2, \_, \_, 3, 3, \_]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [1, 2, 3, 6, 8, 9]$

$B = [\_, 0, 2, 2, \_, \_, 3, 3, \_]$

Sortieren in linearer Zeit

6: **for** $j = 1$ **to** $|A|$ **do**
7: $\quad C[A[j]] := C[A[j]] + 1 \qquad \%\ C[i] = \#\{j : A[j] = i\}$
8: **end for**
9: **for** $i = 1$ **to** k **do**
10: $\quad C[i] := C[i] + C[i-1] \qquad \%\ C[i] = \#\{j : A[j] \leq i\}$
11: **end for**
12: **for** $j = |A|$ **downto** 1 **do**
13: $\quad B[C[A[j]]] := A[j] \qquad \%\ C[A[j]] = \#\{k : A[k] \leq A[j]\}$ =korrekte Position.
14: $\quad C[A[j]] := C[A[j]] - 1$ % $B \leftarrow A[j] \implies \#\{k : A[k] \leq A[j]\}$ um eins kleiner
15: **end for**

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [1, 2, 2, 6, 8, 9]$

$B = [0, 0, 2, 2, \_, \_, 3, 3, \_]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [0, 2, 2, 6, 8, 9]$

$B = [0, 0, 2, 2, \_, 3, 3, 3, \_]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [0, 2, 2, 5, 8, 9]$

$B = [0, 0, 2, 2, \_, 3, 3, 3, 5]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [0, 2, 2, 5, 8, 8]$

$B = [0, 0, 2, 2, 3, 3, 3, 3, 5]$

Sortieren in linearer Zeit

```
 6: for j = 1 to |A| do
 7:     C[A[j]] := C[A[j]] + 1        % C[i] = #{j : A[j] = i}
 8: end for
 9: for i = 1 to k do
10:     C[i] := C[i] + C[i − 1]       % C[i] = #{j : A[j] ≤ i}
11: end for
12: for j = |A| downto 1 do
13:     B[C[A[j]]] := A[j]            % C[A[j]] = #{k : A[k] ≤ A[j]} =korrekte Position.
14:     C[A[j]] := C[A[j]] − 1 % B ← A[j] ⟹ #{k : A[k] ≤ A[j]} um eins kleiner
15: end for
```

Beispiel:

$A = [3, 5, 3, 0, 2, 3, 2, 0, 3]$

$C = [0, 2, 2, 4, 8, 8]$

$B = [0, 0, 2, 2, 3, 3, 3, 3, 5]$

Sortieren in linearer Zeit

Radix-Sort

Sortieren von d-stelligen Zahlen, zuerst nach der letzten Stelle, dann nach der vorletzten u.s.w.. . .

Für die Korrektheit müssen die Teilsortierungen stabil sein

Algorithm RADIX-SORT(A, d)

```
1: for i = 0 to d do
2:     COUNTING-SORT(A bzgl. i)
3: end for
```

Beispiel:

432		631		631		123		123		123	
827		151		151		827		827		151	
631		432		432		628		628		239	
477		123		123		631		631		432	
628		444		444		432		432		444	
239	$\rightarrow$	827	$\rightarrow$	827	$\rightarrow$	239	$\rightarrow$	239	$\rightarrow$	477	
151		477		477		444		444		628	
649		628		628		649		649		631	
123		239		239		151		151		649	
444		649		649		477		477		827	

Sortieren in linearer Zeit

Lemma

Gegeben seien n d-stellige Zahlen, jede Stelle in $\{0, 1, \ldots, k\}$. Dann sortiert Radix-Sort korrekt in der Laufzeit von $\Theta(d(n+k))$.

Beweis: Korrektheit: Induktion nach der Stelle. Laufzeit: klar.

Folgerung

Gegeben seien n b-Bit-Zahlen und $r \in \mathbb{N}$ mit $r \leq b$. Dann sortiert Radix-Sort diese Zahlen korrekt in der Laufzeit von $\Theta\left(\frac{b}{r}(n+2^r)\right)$.

Beweis: b Bits auffassen als d-stellige Zahl,
jede Stelle r Bits,
$\Longrightarrow \quad d = \left\lceil \frac{b}{r} \right\rceil$.
Daher ist jede Ziffer in $\{0, 1, \ldots, 2^r - 1\}$. □

Sortieren in linearer Zeit

Bucket-Sort

Annahme: Die Eingabe besteht aus unabhängigen und über $[0, 1)$ gleichverteilten Zufallsvariablen $X_1, \ldots, X_n$.
Idee: Teile $[0, 1)$ in n Buckets (Teilintervalle); pro Teilintervall wenige Zahlen.

Algorithm BUCKET-SORT(A)

```
 1: n := |A|
 2: Sei B[0..n − 1] ein neues Feld
 3: for i = 0 to n − 1 do
 4:     B[i] := [ ]
 5: end for
 6: for i = 1 to n do
 7:     B [⌊nA[i]⌋] := [B [⌊nA[i]⌋] , A[i]]
 8: end for
 9: for i = 0 to n − 1 do
10:     INSERTION-SORT(B[i])
11: end for
12: Hänge B[0], B[1], . . . , B[n − 1] zusammen.
```

Sortieren in linearer Zeit

Korrektheit

$A[i] \leq A[j]$. Zwei Fälle möglich:

1. verschiedene Buckets, dann $\lfloor nA[i] \rfloor < \lfloor nA[j] \rfloor$;
2. gleiches Bucket, aber dann Insertion-Sort.

Kosten

$N_i =$ Anzahl der Zahlen in $B[i]$ (Zufallsvariable!)
Daher:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}\left(N_i^2\right),$$

und

$$\mathbb{E}T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}\left(\mathbb{E}N_i^2\right).$$

Sortieren in linearer Zeit

$N_i =$ Anzahl der Zahlen in $B[i]$

$X_{i,j} := \mathbb{1}_{[A[j] \text{ fällt in } B[i]]}$, dann gilt: $N_i = \sum_{j=1}^{n} X_{i,j}$.

$$\mathbb{E}[N_i^2] = \mathbb{E}\left[\left(\sum_{j=1}^{n} X_{i,j}\right)^2\right] = \mathbb{E}\left[\sum_{j=1}^{n} X_{i,j}^2 + 2\sum_{1\le j<k\le n} X_{i,j}X_{i,k}\right]$$

$$= \sum_{j=1}^{n} \mathbb{E}\left[X_{i,j}^2\right] + 2\sum_{1\le j<k\le n} \mathbb{E}\left[X_{i,j}X_{i,k}\right]$$

$$= \sum_{j=1}^{n} \mathbb{E}X_{i,j} + 2\sum_{1\le j<k\le n} \mathbb{P}\left\{X_{i,j} = X_{i,k} = 1\right\}$$

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

Daraus folgt $\mathbb{E}T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}\left(\mathbb{E}N_i^2\right) = \Theta(n)$.

9) Ordnungsstatistiken

Ordnungsstatistiken

Minimum und Maximum

Definition

Die i-te Ordnungsstatistik von $a_1, \ldots, a_n$ ist jenes a_j, für das $|\{a_\ell | \, a_\ell < a_j\}| = i - 1$.
Die erste Ordnungsstatistik heißt Minimum, die n-te Ordnungsstatistik Maximum.
Für $i = \left\lfloor \frac{n+1}{2} \right\rfloor$ nennen wir die i-te Ordnungsstatistik auch unteren Median, für $i = \left\lceil \frac{n+1}{2} \right\rceil$ oberen Median.

Auswahlproblem:
Eingabe: Menge A von n paarweise verschiedenen Zahlen und $i \in \mathbb{Z} : 1 \leq i \leq n$
Ausgabe: $x \in A$ mit $|\{y \in A | \, y < x\}| = i - 1$
Möglich in $\Theta(n \log n)$ Schritten: Sortieren.

Ordnungsstatistiken

Algorithm MIN(A)

```
1: min := A[1]
2: for j = 2 to n do
3:     if A[j] < min then
4:         min := A[j]
5:     end if
6: end for
```

Maximum: analog.

Anzahl der Vergleiche: $n - 1$ (bestmöglich!)

Ordnungsstatistiken

Minimum und Maximum gleichzeitig: Sei $|A|$ gerade.

Algorithm MINMAX(A)

```
1: n := |A|
2: M := max(A[1], A[2])
3: m := min(A[1], A[2])
4: for i = 2 to n/2 do
5:     m = min(min(A[2i − 1], A[2i]), m)
6:     M = max(max(A[2i − 1], A[2i]), M)
7: end for
```

Falls $|A|$ ungerade, dann zuerst sowohl Minimum als auch Maximum auf $A[1]$ setzen.
Anzahl der Vergleiche:

$$T(n) = \begin{cases} 3\frac{n-1}{2} = 3\left\lfloor \frac{n}{2} \right\rfloor & \text{falls } n \text{ ungerade;} \\ 1 + 3\frac{n-2}{2} = \frac{3n}{2} - 2 \leq 3\left\lfloor \frac{n}{2} \right\rfloor & \text{falls } n \text{ gerade.} \end{cases}$$

Auswahl in erwarteter linearer Zeit

Divide & Conquer (Quickselect, Hoare's Find-Algorithm)
zur Bestimmung der i-ten Ordnungsstatistik von $A[p..r]$

Algorithm RANDOMIZED-SELECT(A, p, r, i)

```
 1: if p = r then
 2:     return A[p]
 3: end if
 4: q :=RANDOMIZED-PARTITION(A, p, r)
 5: k := q − p + 1
 6: if i = k then
 7:     return A[q]
 8: else if i < k then
 9:     return RANDOMIZED-SELECT(A, p, q − 1, i)
10: else
11:     return RANDOMIZED-SELECT(A, q + 1, r, i − k)
12: end if
```

Ordnungsstatistiken

Analyse: $n = r - p + 1 \ldots$ # Elemente, paarweise verschieden.
RANDOMIZED-PARTITION: $\mathcal{O}(n)$,
$\mathbb{P}\{|A[p..q]| = k\} = \frac{1}{n}$ für $1 \leq k \leq n$.

Sei $B_k = [|A[p \ldots q]| = k]$ und $X_k = \mathbb{1}_{B_k}$. Dann $\mathbb{E}X_k = \frac{1}{n}$.
Annahme: $T(n)$ monoton wachsend. Dann folgt

$$\begin{aligned} T(n) &\leq \sum_{k=1}^{n} X_k \cdot \max(T(k-1), T(n-k)) + \mathcal{O}(n) \\ &= \sum_{k=1}^{n} X_k T(\max(k-1, n-k)) + \mathcal{O}(n) \end{aligned}$$

und daher gilt

$$\mathbb{E}T(n) \leq \sum_{k=1}^{n} \mathbb{E}X_k \cdot \mathbb{E}T(\max(k-1, n-k)) + \mathcal{O}(n).$$

Ordnungsstatistiken

$$\mathbb{E}T(n) \leq \sum_{k=1}^{n} \mathbb{E}X_k \cdot \mathbb{E}T(\max(k-1, n-k)) + \mathcal{O}(n)$$

und $\mathbb{E}X_k = \frac{1}{n}$ sowie

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{für } k > \left\lceil \frac{n}{2} \right\rceil; \\ n-k & \text{für } k \leq \left\lceil \frac{n}{2} \right\rceil. \end{cases}$$

n gerade: $T\left(\left\lceil \frac{n}{2} \right\rceil\right), \ldots, T(n-1)$ genau 2-mal in der Summe;

n ungerade: $T\left(\left\lceil \frac{n}{2} \right\rceil\right), \ldots, T(n-1)$ genau 2-mal, $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ einmal.

Daraus folgt

$$\mathbb{E}T(n) \leq \frac{2}{n} \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} \mathbb{E}T(k) + \mathcal{O}(n).$$

Ordnungsstatistiken

Wir haben also $\mathbb{E}T(n) \leq \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} \mathbb{E}T(k) + \mathcal{O}(n)$.

Noch zu zeigen: $\mathbb{E}T(n) = \mathcal{O}(n)$; mittels Substitutionsmethode.

Annahme: $\mathbb{E}T(n) \leq cn$, $T(n) = \mathcal{O}(1)$ für $n \leq N$, $\mathcal{O}(n) \leq an$.

$$\begin{aligned}
\mathbb{E}T(n) &\leq \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} ck + an = \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor - 1} k \right) + an \\
&\leq \frac{c}{n} \left(n(n-1) - \left(\frac{n}{2} - 1 \right) \left(\frac{n}{2} - 2 \right) \right) + an \\
&= c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right)
\end{aligned}$$

Wähle $c > 4a$, $n \geq \frac{2c}{c-4a} =: N$

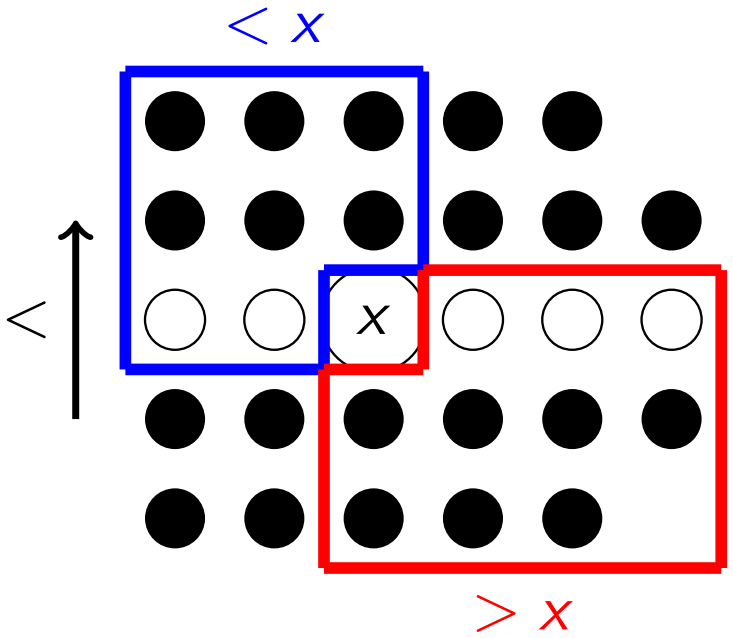
Auswählen in linearer Zeit (Worst-Case)

PARTITION(A, x) modifizieren, $x \in \{A[1], A[2], \ldots, A[n]\}$
Algorithmus (SELECT):

1. Elemente in $\lfloor \frac{n}{5} \rfloor$ 5er-Gruppen und eine ($n \mod 5$)-Gruppe teilen;
2. Median jeder dieser $\lceil \frac{n}{5} \rceil$ Gruppen mit INSERTION-SORT bestimmen;
3. Median x dieser $\lceil \frac{n}{5} \rceil$ Mediane mit SELECT bestimmen;
4. PARTITION(A, x) liefert $A[1..k-1]$, $A[k+1..n]$, $x = A[k]$;
5. Wenn $i = k$, dann x ausgeben, sonst:

$$\begin{array}{ll} \text{SELECT}(A[1..k-1], i), & \text{falls } i < k, \\ \text{SELECT}(A[k-1..n], i-k), & \text{falls } i > k. \end{array}$$

Ordnungsstatistiken



- Es gibt mindestens $\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil - 2$ Gruppen mit 3 Elementen, die größer als x sind.
- Daher gibt es mindestens $3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6$ Elemente, die größer als x sind (und mindestens $\frac{3n}{10} - 6$, die kleiner als x sind).
- Der rekursive Aufruf von SELECT erfolgt auf höchstens $\frac{7n}{10} + 6$ Elemente.

Ordnungsstatistiken

Kosten:

- Schritte (1), (2) und (4): $\mathcal{O}(n)$;
- Schritt (3): $T\left(\left\lceil \frac{n}{5} \right\rceil\right)$;
- Schritt (5): höchstens $T\left(\frac{7n}{10} + 6\right)$.

$$T(n) \leq \begin{cases} \mathcal{O}(1), & \text{falls } n < N; \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + \mathcal{O}(n) & \text{falls } n \geq N. \end{cases}$$

Mit der Substitutionsmethode lässt sich für $N \geq 140$ zeigen, dass $T(n) = \mathcal{O}(n)$.