Übung Algorithmen I
20.5.15

Christoph Striecks
Christoph.Striecks@kit.edu

(Mit Folien von Julian Arz, Timo Bingmann und Sebastian Schlag.)

Roadmap

- Organisation
- Mergesort, Quicksort
- Dual Pivot Quicksort
- Vorsortiertheit und adaptive Sortieren

Organisation

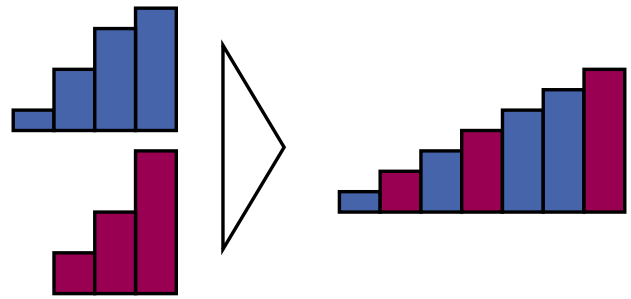
- Übungsblatt 5, Aufgabe 1c):
  $\Sigma = \{A, b, e, g, h, i, l, m, n, o, r, t\}$

Sortieren durch Mischen

Idee: Teile und Herrsche

Function mergeSort($\langle e_1, \ldots, e_n \rangle$) : Sequence **of** Element
  if $n = 1$ **then return** $\langle e_1 \rangle$ // base case
  else return merge(mergeSort($\langle e_1, \ldots, e_{\lfloor n/2 \rfloor} \rangle$),
    mergeSort($\langle e_{\lfloor n/2 \rfloor + 1}, \ldots, e_n \rangle$))

Gegeben:
zwei sortierte Folgen a und b
Berechne:
sortierte Folge der Elemente aus a und b



Merge Sort

```
Function mergeSort(A : Array of Element; lo, hi : ℕ)
    if hi − lo ≤ 1 then return                                  // Basisfall
    mid := (lo + hi)/2                                          // mittleres Element
    mergeSort(lo, mid), mergeSort(mid, hi)                      // Sortiere Hälften
    T := allocate (Array of Element size hi − lo)
    i := lo,   j := mid,   k := 0                               // Laufindizes
    while i < mid ∧ j < hi
        if A[i] < A[j]  T[k++] := A[i++]                        // Mische!
        else            T[k++] := A[j++]
    endwhile
    while i < mid  do  T[k++] := A[i++]                         // Kopiere Reste
    while j < hi   do  T[k++] := A[j++]
    A[lo, ..., hi − 1] := T[0, ..., (hi − lo) − 1]              // Kopiere zurück
    dispose (T)
```

Worst case: $\Theta(n \log n)$, average case $\Theta(n \log n)$.

Tatsächliche Laufzeit einer Implementierung

- Average case: $\Theta(n \log n)$
- $n = 100$, $100 \log_2 100 \approx 664$
- (Demo)

Quicksort – erster Versuch

Idee: Teile-und-Herrsche aber verglichen mit mergesort „andersrum“.
Leiste Arbeit vor rekursivem Aufruf

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
    if $|s| \leq 1$ **then return** s
    pick “some” $p \in s$
    $a := \langle e \in s : e < p \rangle$
    $b := \langle e \in s : e = p \rangle$
    $c := \langle e \in s : e > p \rangle$
    return concatenation of quickSort(a), b, and quickSort(c)

Quicksort – Analyse im schlechtesten Fall

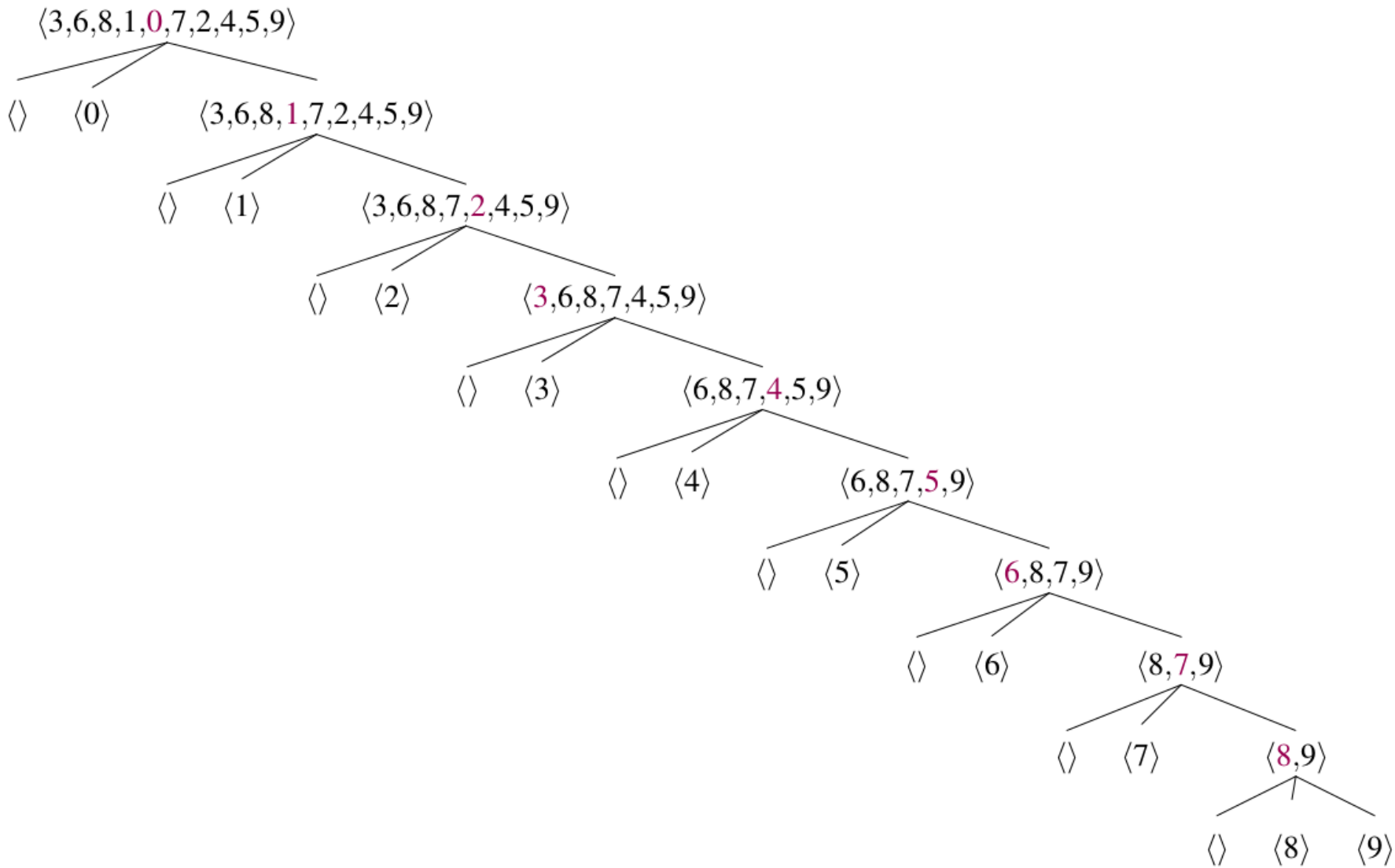
Annahme: Pivot ist immer Minimum (oder Max.) der Eingabe

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \Theta(n) + T(n-1) & \text{if } n \geq 2. \end{cases}$$

$\Rightarrow$

$T(n) = \Theta(n + (n-1) + \cdots + 1) = \Theta(n^2)$

Schlechtester Fall: Beispiel

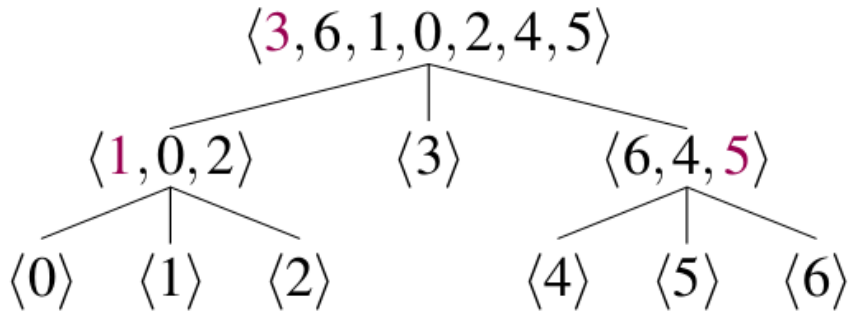


Quicksort – Analyse im besten Fall

Annahme: Pivot ist immer Median der Eingabe

$$T(n) \leq \begin{cases} O(1) & \text{if } n = 1, \\ O(n) + 2T(\lfloor n/2 \rfloor) & \text{if } n \geq 2. \end{cases}$$

$\Rightarrow$ (Master-Theorem)
$T(n) = O(n \log n)$
Problem: Median bestimmen ist nicht so einfach



Satz: $\bar{C}(n) \leq 2n \ln n \leq 1.45 n \log n$

$$\begin{aligned}
\bar{C}(n) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&\leq \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} \\
&= 2n \sum_{k=2}^{n} \frac{1}{k} \\
&= 2n(H_n - 1) \leq 2n(1 + \ln n - 1) = 2n \ln n \ .
\end{aligned}$$

i	j	$\overbrace{j-i+1}^{=:k}$
1	$2..n$	$2..n$
2	$3..n$	$2..n-1$
3	$4..n$	$2..n-2$
$\vdots$	$\vdots$	$\vdots$
$n-1$	$n..n$	$2..2$
n	$\emptyset$	$\emptyset$

(harmonische Summe)

□

Einige Quicksort-Analysen

Hoare (1962)
"Ur"-Quicksort – Average Case für 1 zufällig gewähltes Pivot

		für $n = 100$
erwartete Vergleiche:	$= 2n(H_n - 1)$	$837,5$
	$\approx 2n \log_e n$	$921,0$

(Demo.)

Wild, Nebel (2012)
"Yaroslavskiy"-Quicksort – Average Case für 2 zufällig gewählte Pivots

		für $n = 100$
erwartete Vergleiche:		
	$= 1.9n \log_e n - 2.46n + O(\log n)$	$629 + O(\log 100)$

Vorgefertigte Sortieralgorithmen in aktuellen Programmiersprachen

Hinweis: Verwenden Sie diese Sortieralgorithmen anstatt eigene zu implementieren!

C++

- **Zahlen:** Variante von Quick-Sort

```
#include <algorithm>
int numbers[] = {42, 7, 9, 18, 1, 123};
std::vector<int> vec(numbers, numbers + 6);
std::sort(vec.begin(), vec.end())
```

- **Allgemeine Elemente:** Variante von Merge-Sort

```
#include <algorithm>
bool less_than(Elements& a, Elements& b) {
/*...*/ }
...
Elements* elements = createElements(n);
std::sort(elements, elements + n, less_than)
...oder...
std::stable_sort(elements, elements + n,
less_than)
```

Java

- **Zahlen:** Variante von Quick-Sort

```
int[] numbers = {42, 7, 9, 18, 1, 123};
java.util.Arrays.sort(numbers)
```

- **Allgemeine Elemente:** Variante von Merge-Sort

```
Comparator<Elem> comparator = new
Comparator<Elem> {
    int compare(Elem a, Elem b) { /*...*/ }
}
Elem[] elements = { /*...*/ }
java.util.Arrays.sort(elements, 3, 15,
comparator)
```

Dual Pivot Quicksort

Dual Pivot Quicksort

- **Idee:** Partitioniere Eingabe in **3** Teile durch 2 Pivot-Elemente $p \leq q$
- **Historisch:**

  [Sedgewick 1975], [Hennequin 1991]
  $\Rightarrow$ **keine** Verbesserung durch Multi-Pivot-Ansatz in Theorie und Praxis

Sind die nächsten 15 Minuten also reine Zeitverschwendung? **Nein.**

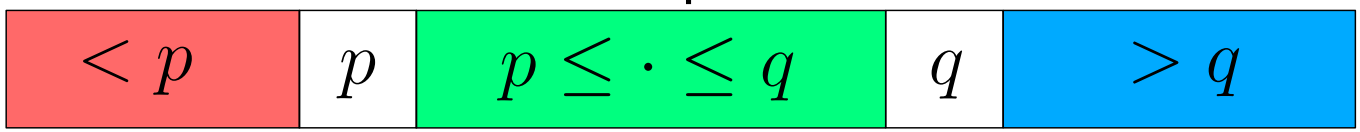
- **2009:** [Yaroslavskiy 2009]
  - praktisch & theoretisch **besser** als Implementierung der Bibliothek
- **2011:** Dual Pivot Quicksort wird Standard in Java 7
- **2012:** Average-case-Analyse [Wild, Nebel 2012]

Dual Pivot Quicksort

1. Wähle 2 Pivotelemente $p \leq q$

2. Klassifiziere Elemente in:
   - klein wenn $\cdot < p$
   - mittel wenn $p \leq \cdot \leq q$
   - groß wenn $q < \cdot$

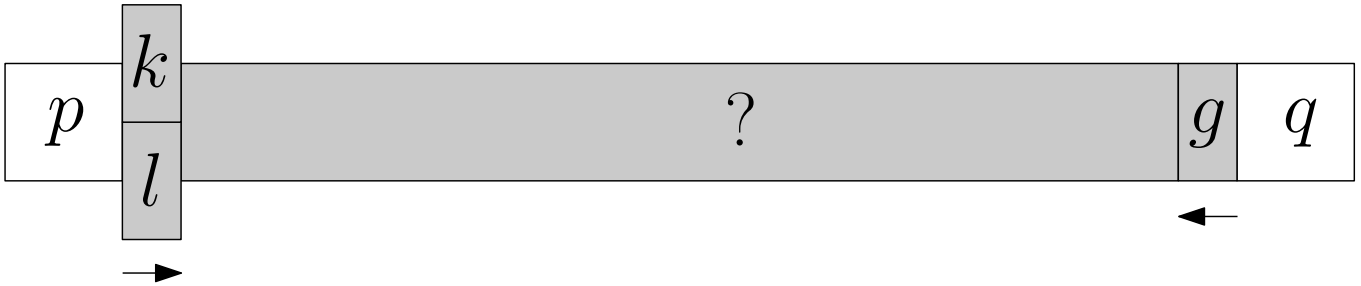
3. Ordne alle Elemente entsprechend ihrer Klasse:



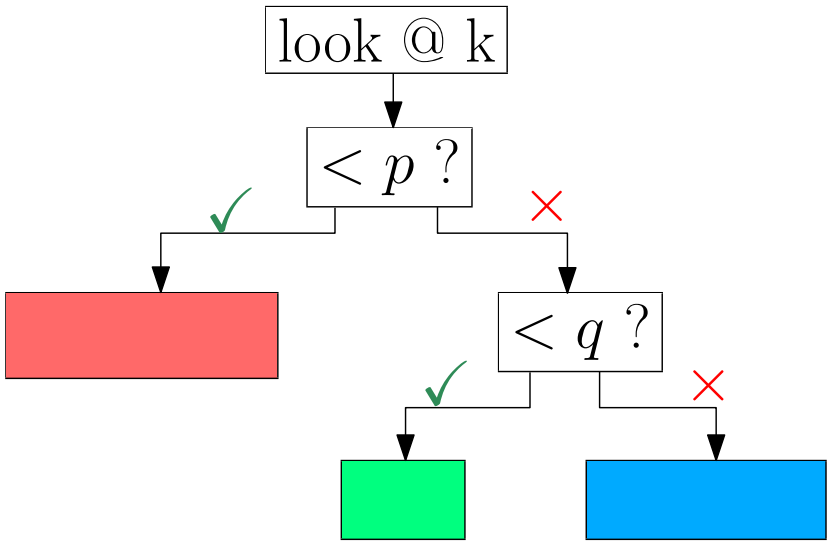
4. Sortierte die 3 Teilbereiche rekursiv

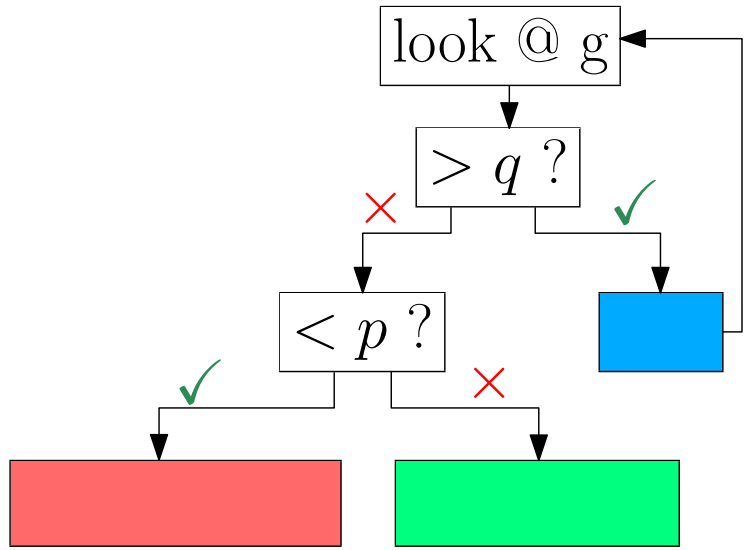
Partitionierung mit 2 Pivots

Fall $a[k] < p$:



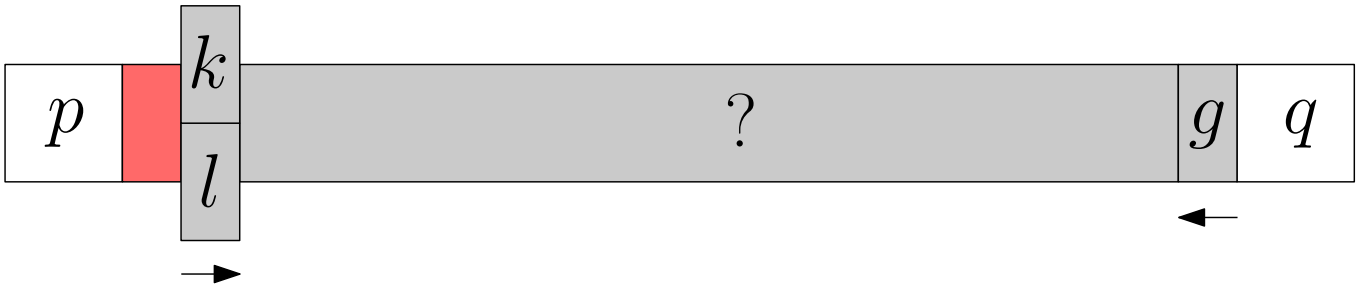
Partitionierung:



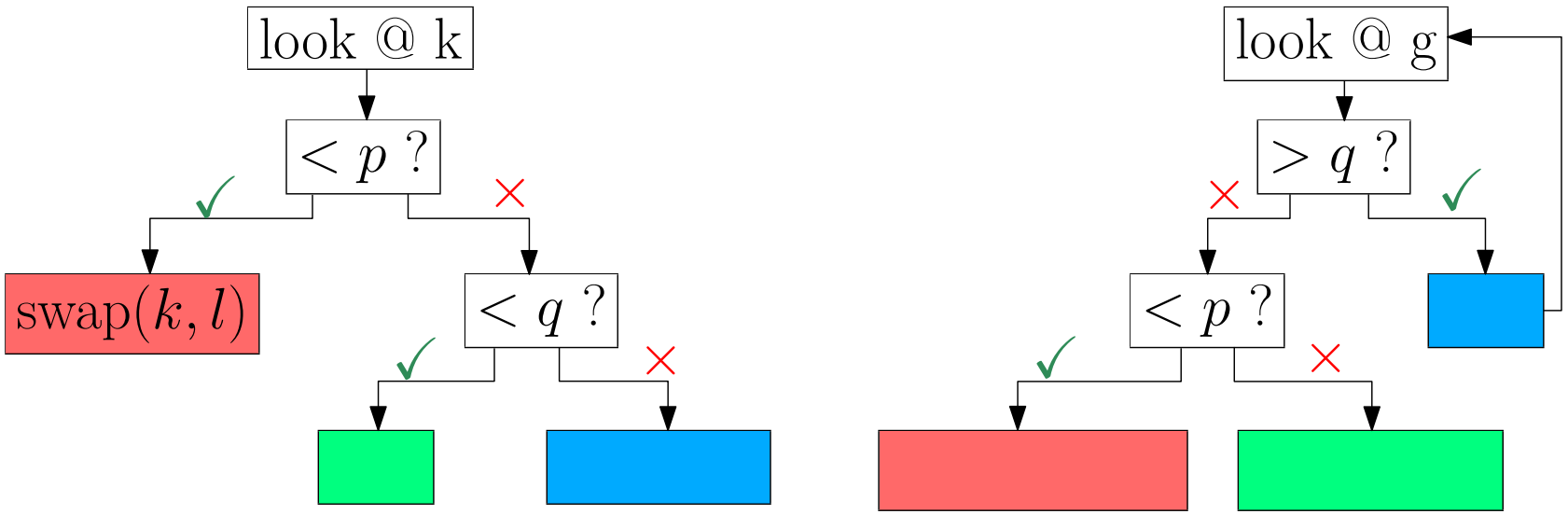


Partitionierung mit 2 Pivots

Fall $a[k] < p$:

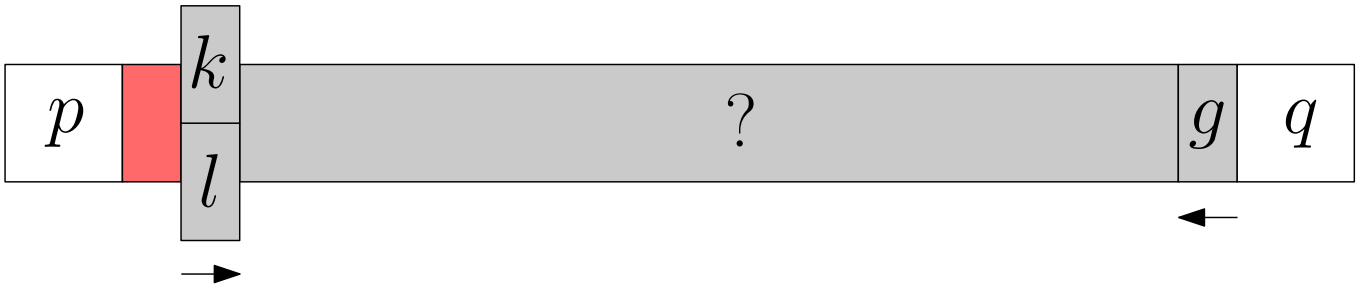


Partitionierung:

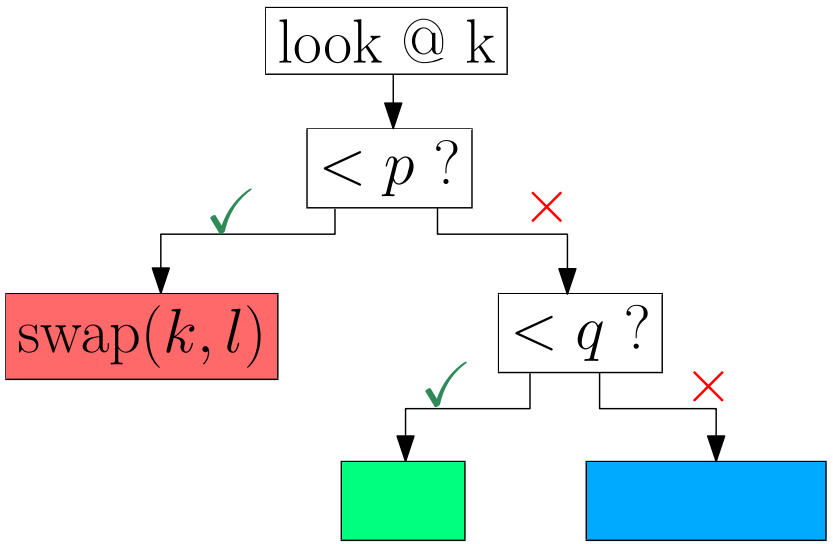


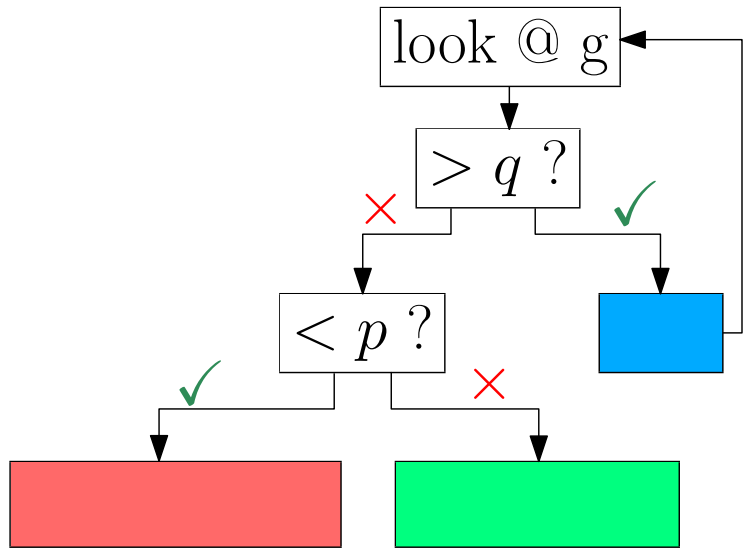
Partitionierung mit 2 Pivots

Fall $p \leq a[k] \leq q$:



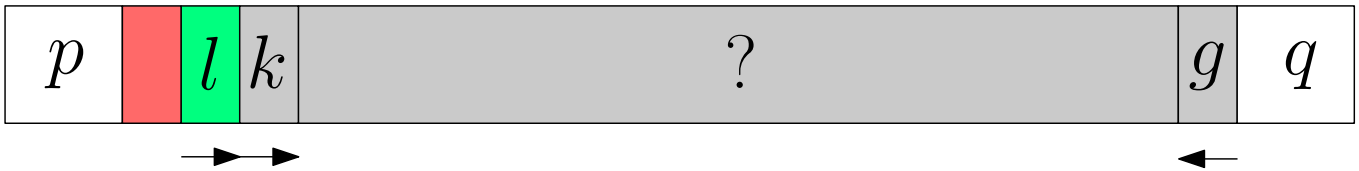
Partitionierung:



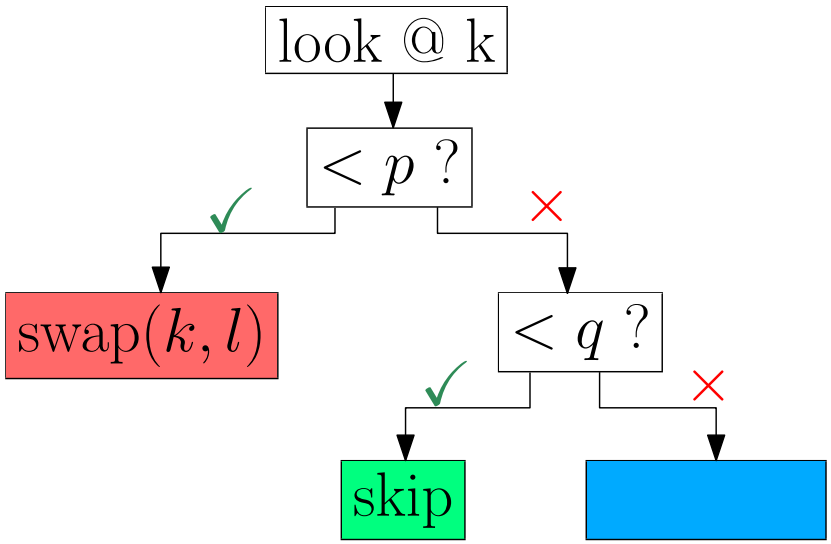


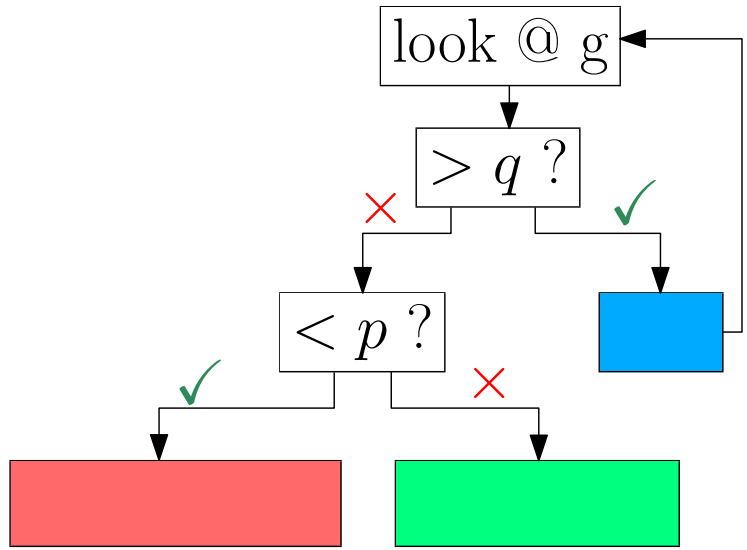
Partitionierung mit 2 Pivots

Fall $p \leq a[k] \leq q$:



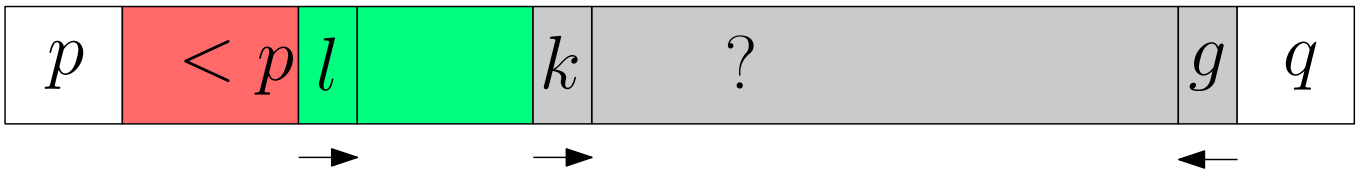
Partitionierung:



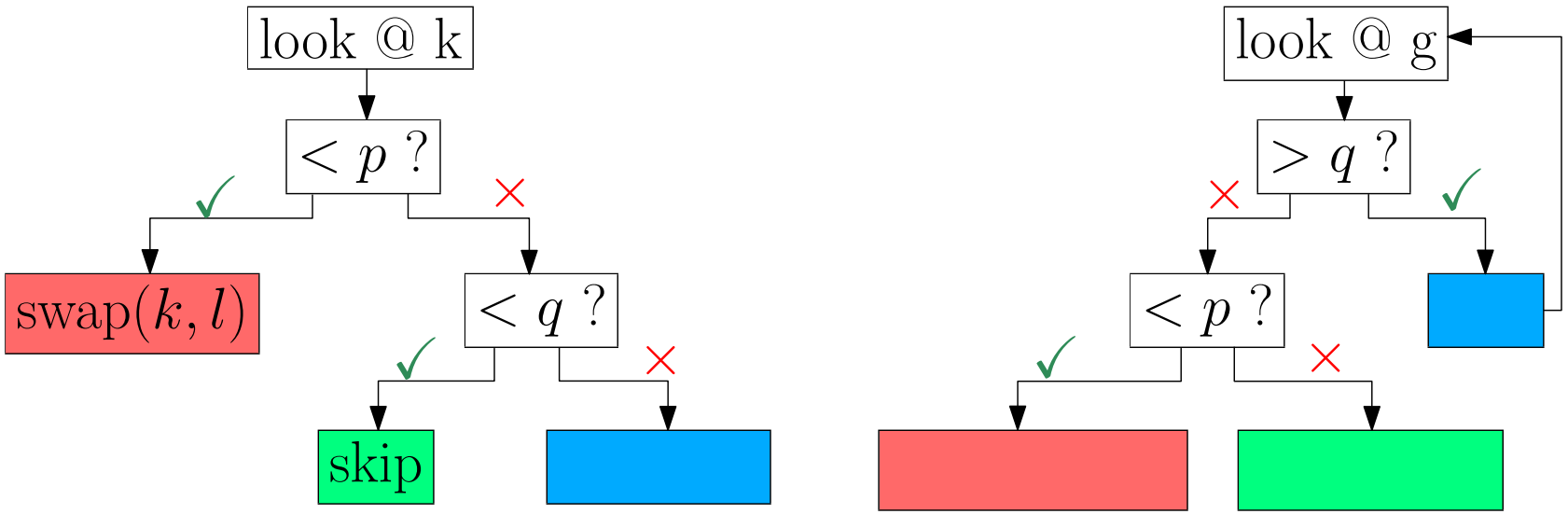


Partitionierung mit 2 Pivots

...

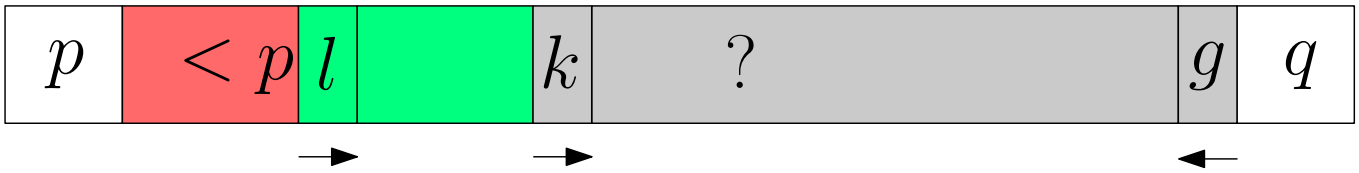


Partitionierung:

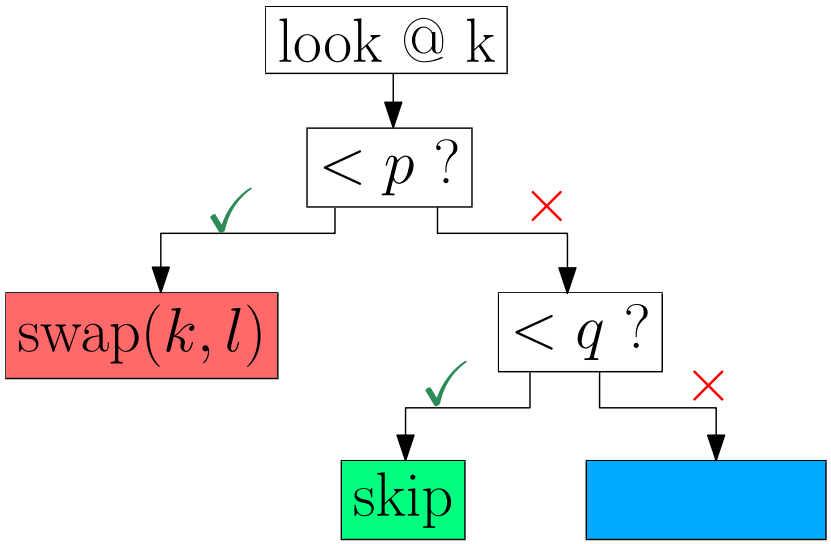


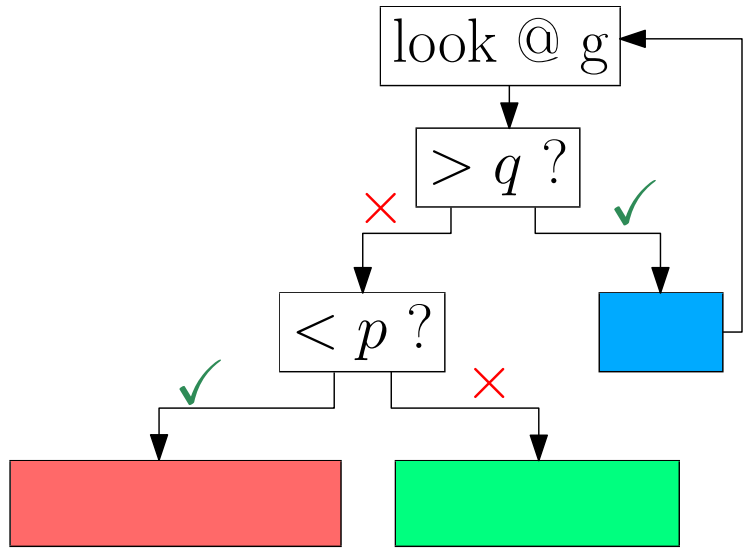
Partitionierung mit 2 Pivots

Fall $q \leq a[k]$:



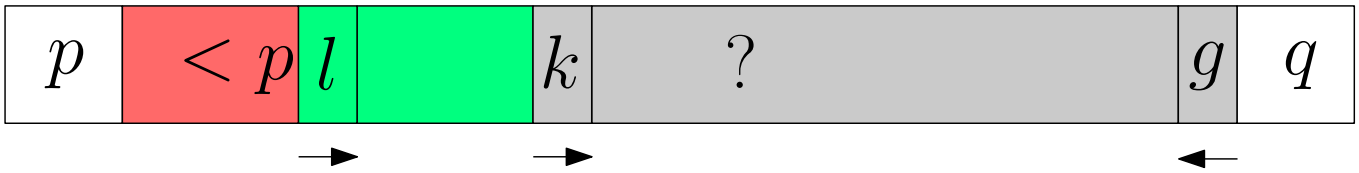
Partitionierung:



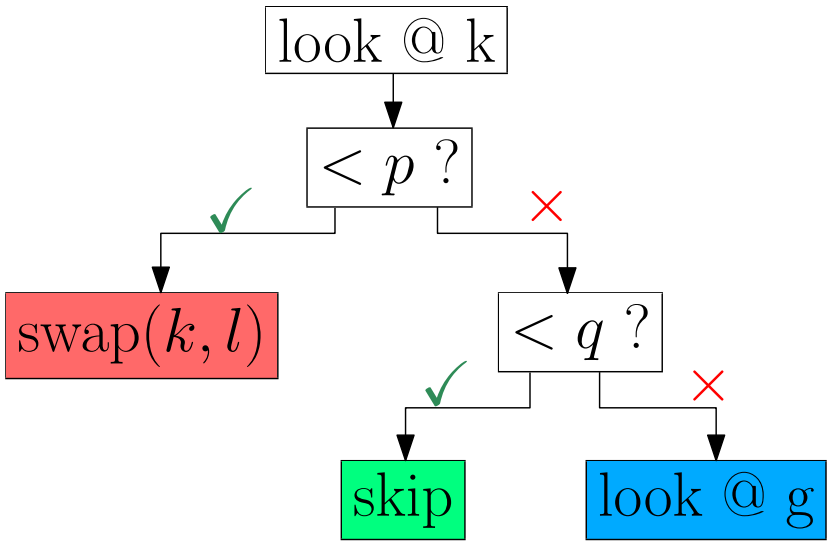


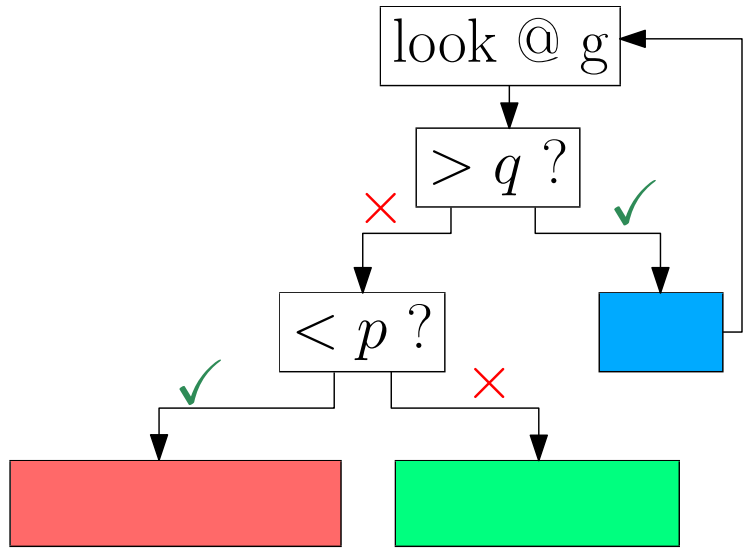
Partitionierung mit 2 Pivots

Fall $q \leq a[k]$:



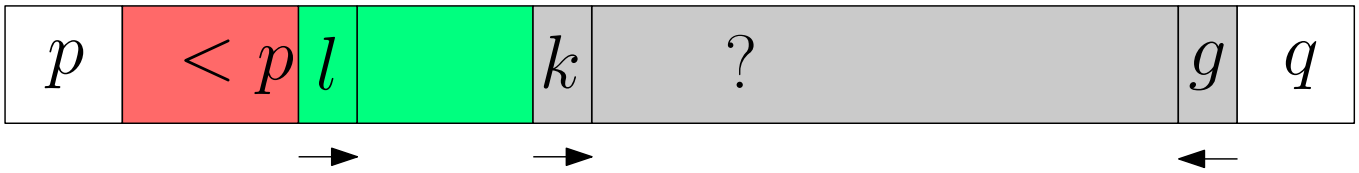
Partitionierung:



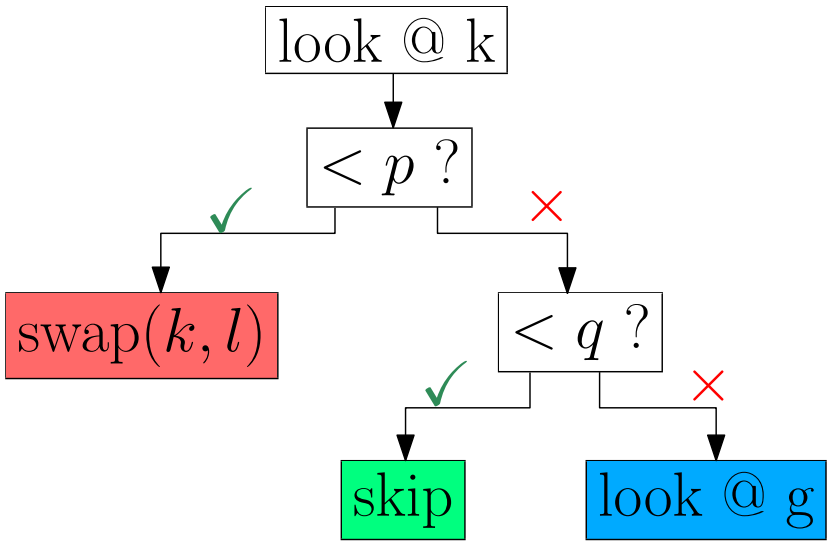


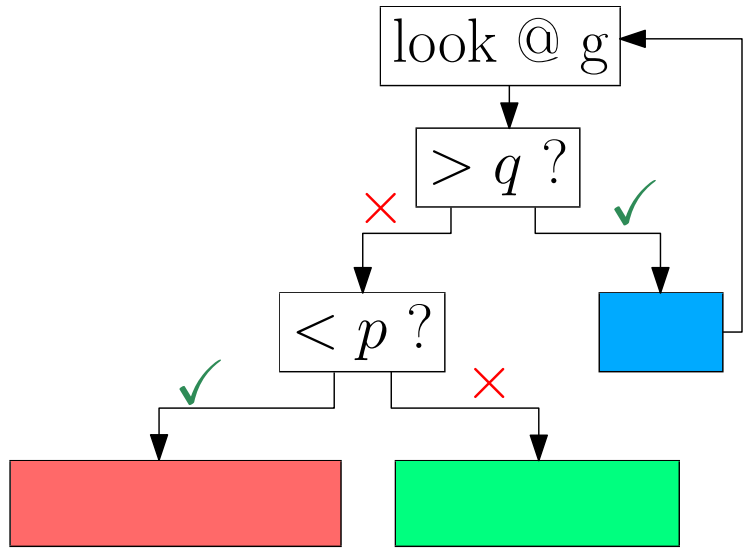
Partitionierung mit 2 Pivots

Fall $q \leq a[k] \wedge a[g] > q$:



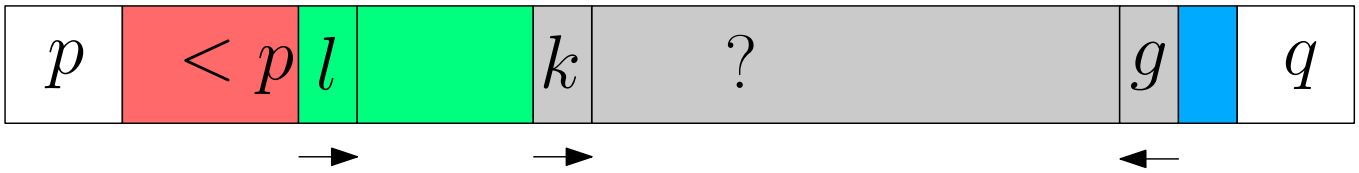
Partitionierung:



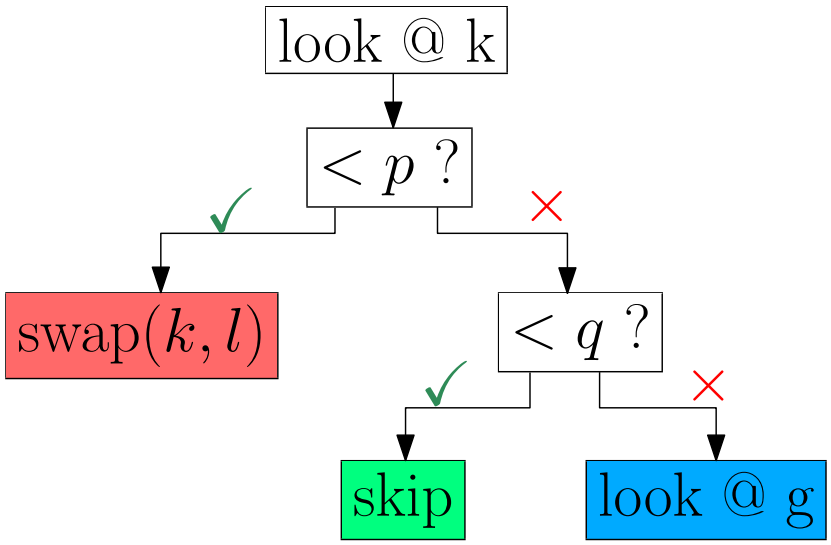


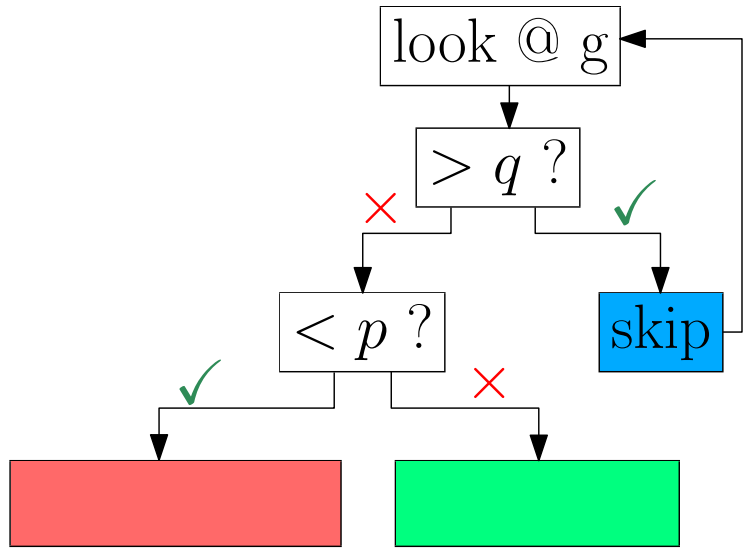
Partitionierung mit 2 Pivots

Fall $q \leq a[k] \wedge a[g] > q$:



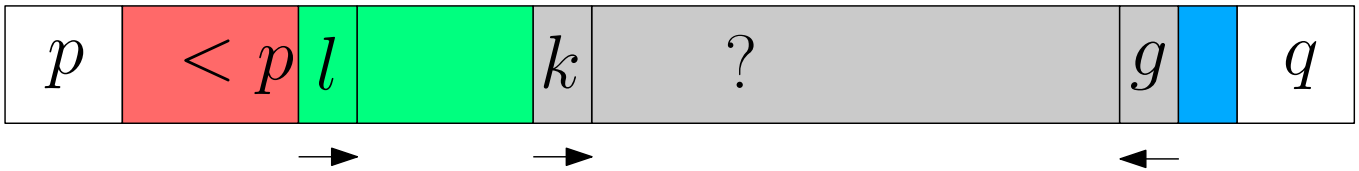
Partitionierung:



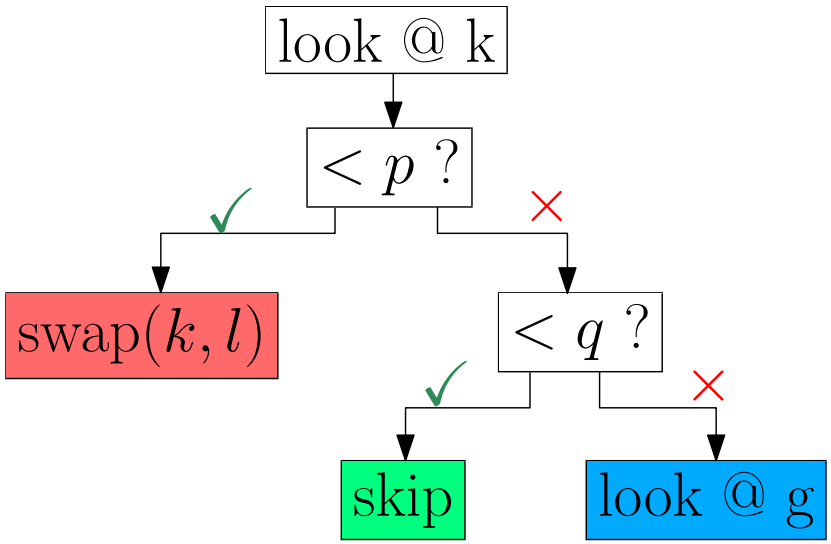


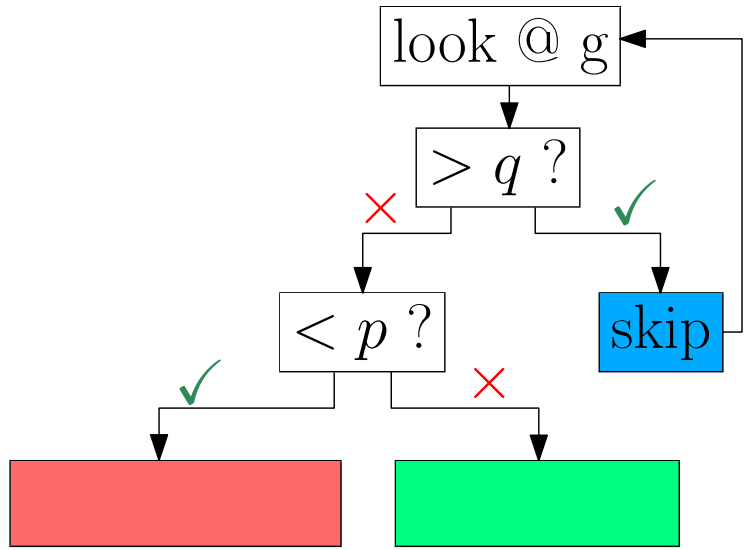
Partitionierung mit 2 Pivots

Fall $q \leq a[k] \wedge p \leq a[g] \leq q$:



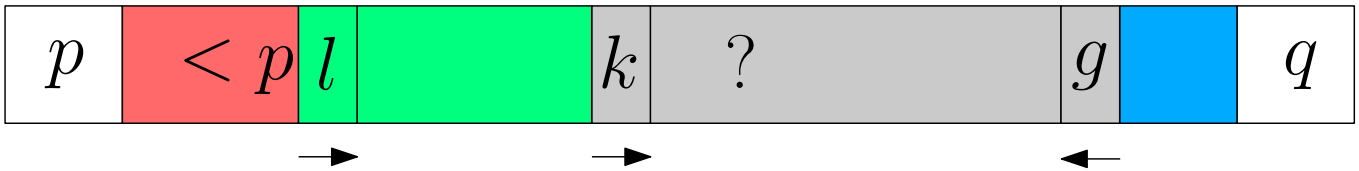
Partitionierung:



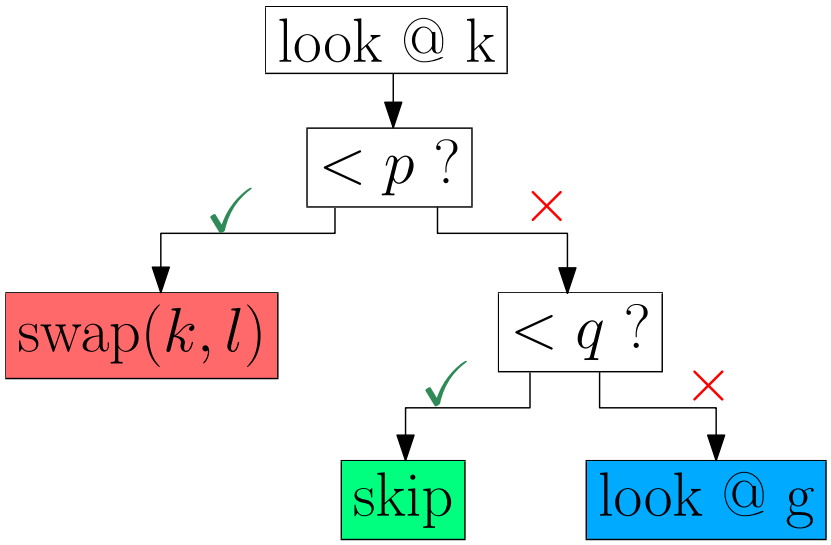


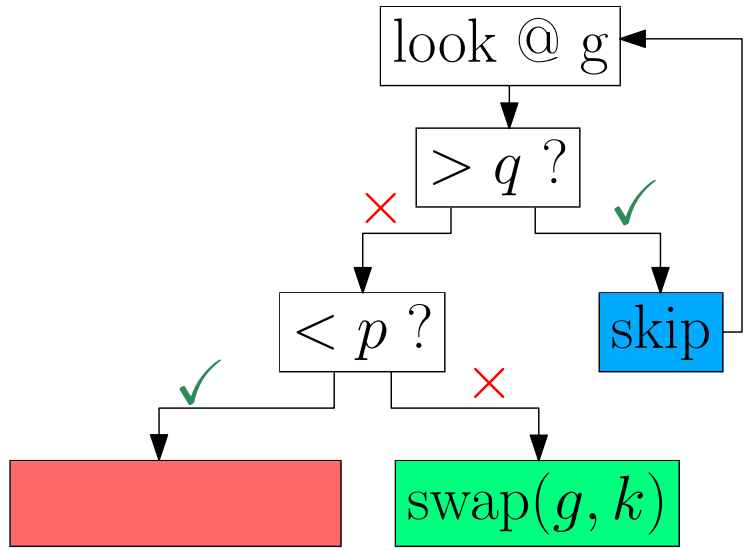
Partitionierung mit 2 Pivots

Fall $q \leq a[k] \wedge p \leq a[g] \leq q$:



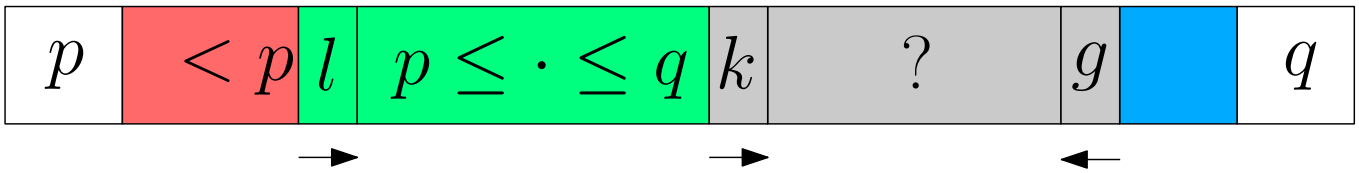
Partitionierung:



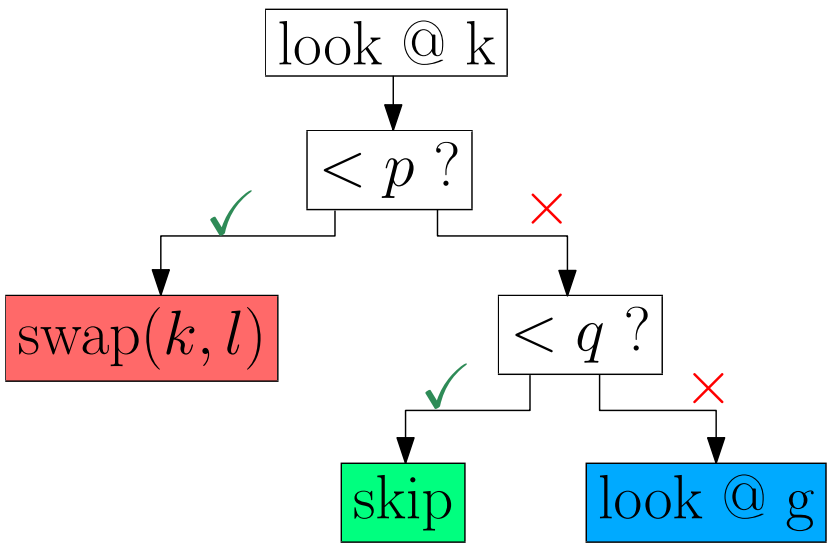


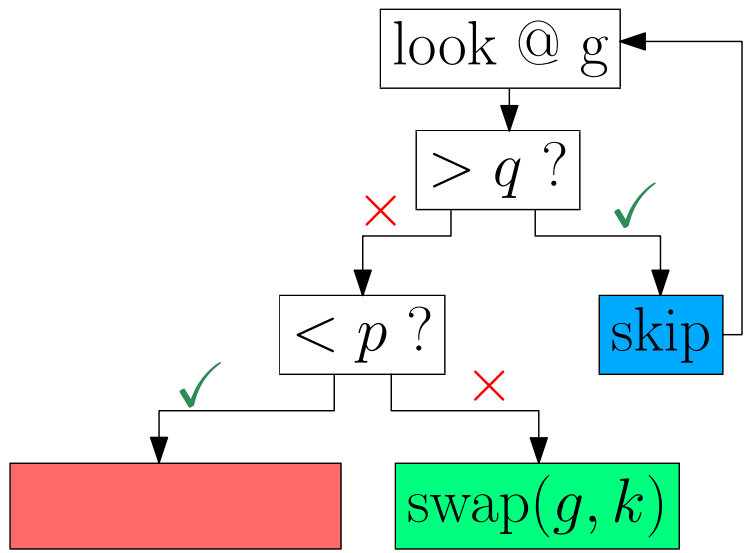
Partitionierung mit 2 Pivots

...



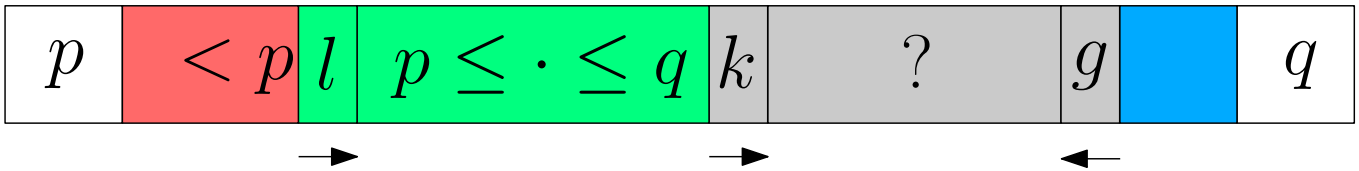
Partitionierung:



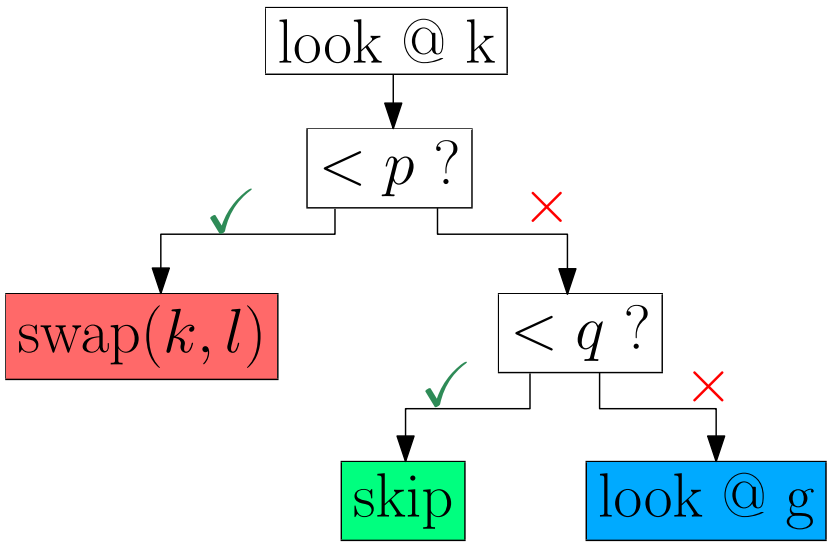


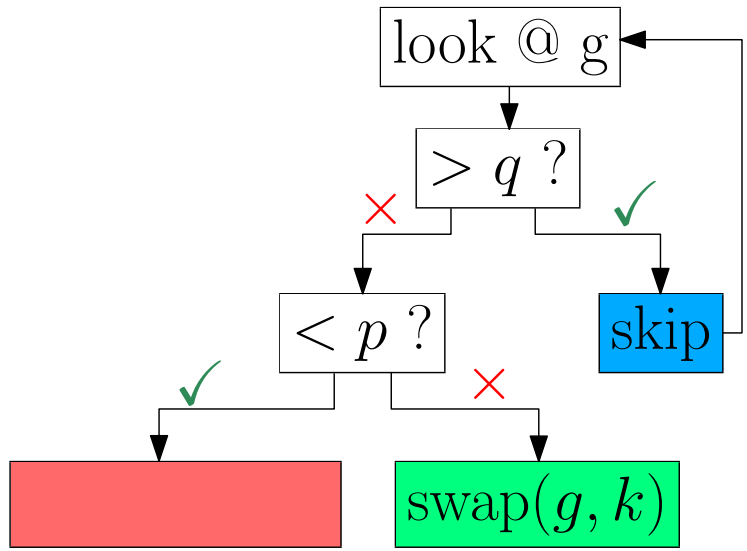
Partitionierung mit 2 Pivots

Fall $q \leq a[k] \wedge a[g] < p$:



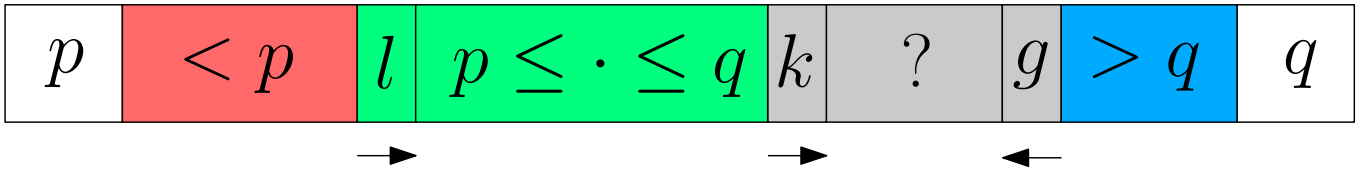
Partitionierung:



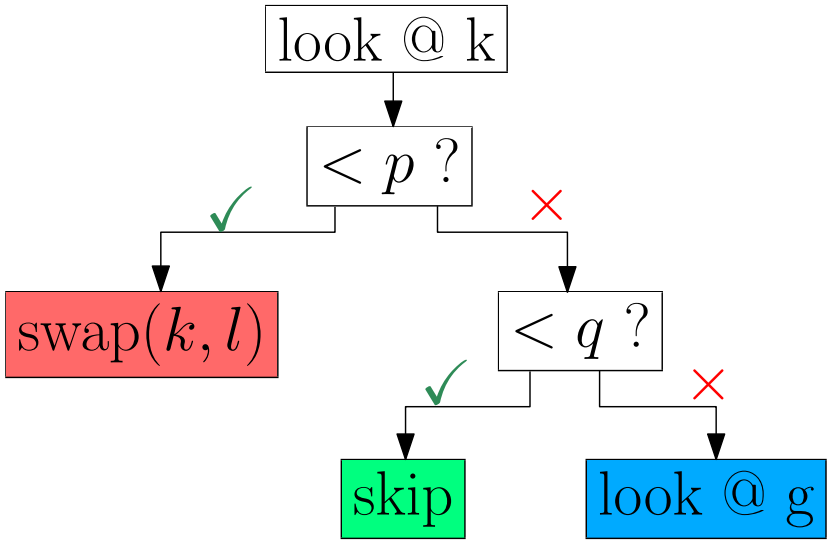


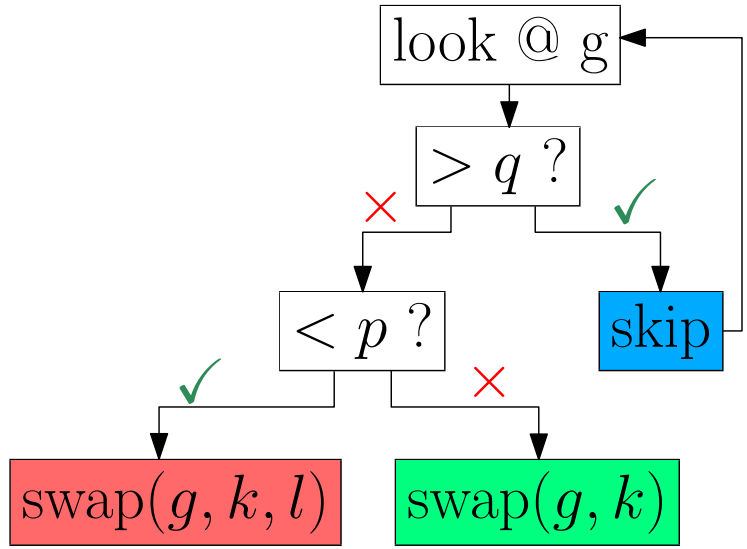
Partitionierung mit 2 Pivots

Fall $q \leq a[k] \wedge a[g] < p$:



Partitionierung:





Einige Quicksort-Analysen

Hoare (1962)
"Ur"-Quicksort – Average Case für 1 zufällig gewähltes Pivot

		für $n = 100$
erwartete Vergleiche:	$= 2n(H_n - 1)$	$837,5$
	$\approx 2n \log_e n$	$921,0$

Wild, Nebel (2012)
"Yaroslavskiy"-Quicksort – Average Case für 2 zufällig gewählte Pivots

	für $n = 100$
erwartete Vergleiche:	
$= 1.9n \log_e n - 2.46n + O(\log n)$	$629 + O(\log 100)$

(Demo.)

Kennzahlen der Vorsortiertheit und adaptive Sortierverfahren

Adaptives Sortieren

Warm-up

Sortieren Sie diese Zahlenfolgen:

1 3 2 4 6 5 7 8

8 2 7 1 5 6 3 4

Adaptives Sortieren

Warm-up

Sortieren Sie diese Zahlenfolgen:

1 3 2 4 6 5 7 8

Nur zwei Inversionen $\rightarrow$ Einfach!

8 2 7 1 5 6 3 4

Nicht so einfach! (17 Inversionen)

Adaptives Sortieren

Sortieren

- Worst und average case: $\Omega(n \log n)$ ist untere Schranke
- Für zufällige Permutationen
- Aber: nutze bestimmte Eigenschaften der Folge aus
- Adaptives Sortieren: Laufzeit steigt mit Länge n und Chaos m

Chaos?

- Unsortiertheit? Vorsortiertheit? Ist dieses Chaos messbar?
- Ja. $\rightarrow$ Kennzahlen der Vorsortiertheit
- Englisch: *measures of presortedness* oder *measures of disorder*

Adaptives Sortieren

Kennzahlen der Vorsortiertheit

- Inversionen
- Runs
- Größte Distanz zwischen Inversionen
- Größte Distanz zur korrekten Position
- Vertauschungen
- Verschachtelte Listen
- Removals
- SUS
- SMS
- Oszillationen
- Alle anderen zusammen

Adaptives Sortieren

Kennzahlen der Vorsortiertheit

- **Inversionen**
- **Runs**
- Größte Distanz zwischen Inversionen
- Größte Distanz zur korrekten Position
- Vertauschungen
- Verschachtelte Listen
- **Removals**
- SUS
- SMS
- Oszillationen
- Alle anderen zusammen

Adaptives Sortieren

Kennzahlen der Vorsortiertheit

Inversionen

- Inversion: Paar $(i, j) \in \mathbb{N}^2$ mit $i < j$ und $\sigma(i) > \sigma(j)$
- Siehe Übung letzte Woche
- Best case $m_{\text{inv}} = 0$, worst case $m_{\text{inv}} = \frac{n(n-1)}{2} = \Theta\left(n^2\right)$
- Average case $m_{\text{inv}} = \binom{n}{2} \cdot \frac{1}{2} = \Theta\left(n^2\right)$

Adaptiv bzgl. m_{inv}: Insertion Sort

- $O(m_{\text{inv}})$ Vertauschungen, $O(n + m_{\text{inv}})$ Vergleiche

Adaptives Sortieren

Kennzahlen der Vorsortiertheit

Inversionen

- Nachteil: Lokale Sortiertheit wird nicht erkannt.
- Beispiel: 5 6 7 8 9 0 1 2 3 4
- Quadratische Anzahl von Inversionen, aber einfach zu sortieren

Runs

- Ein Run ist eine zusammenhängende Teilfolge aufsteigend sortierter Elemente
- Das obige Beispiel hat 2 Runs.
- Best case $m_{\text{runs}} = 1$, worst case $m_{\text{runs}} = n$
- Average case?

Runs

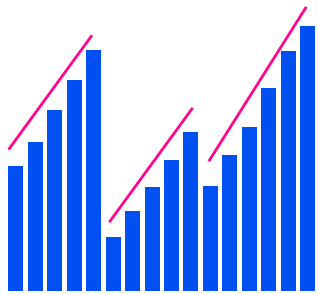
Erwartete Anzahl von Runs

- Für festes n sei $\#_k$ die Anzahl der Permutationen mit k Runs.
- Beobachtung: Permutation mit k Runs hat $k-1$ Abstiege

Runs

Erwartete Anzahl von Runs

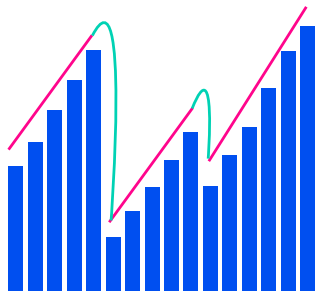
- Für festes n sei $\#_k$ die Anzahl der Permutationen mit k Runs.
- Beobachtung: Permutation mit k Runs hat $k-1$ Abstiege



Runs

Erwartete Anzahl von Runs

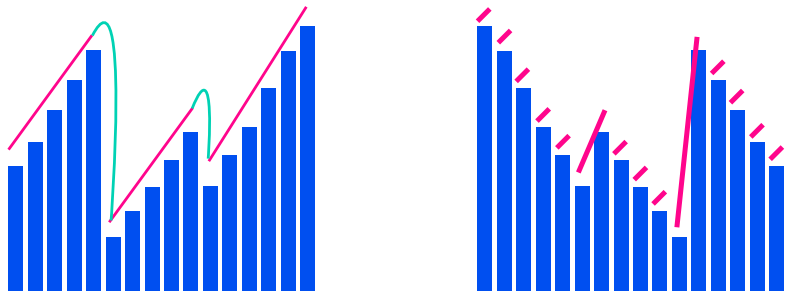
- Für festes n sei $\#_k$ die Anzahl der Permutationen mit k Runs.
- Beobachtung: Permutation mit k Runs hat $k-1$ Abstiege



Runs

Erwartete Anzahl von Runs

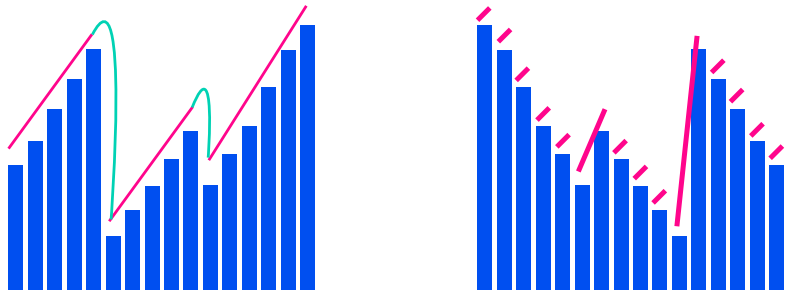
- Für festes n sei $\#_k$ die Anzahl der Permutationen mit k Runs.
- Beobachtung: Permutation mit k Runs hat $k-1$ Abstiege



Runs

Erwartete Anzahl von Runs

- Für festes n sei $\#_k$ die Anzahl der Permutationen mit k Runs.
- Beobachtung: Permutation mit k Runs hat $k-1$ Abstiege



- Rückwärts gelesen hat sie $n-(k-1) = n-k+1$ Runs.
- $\rightarrow$ Bijektion von Permutationen mit k Runs auf Permutationen mit $n-k+1$ Runs
- $\rightarrow \#_k = \#_{n-k+1}$

$$\mathrm{E}(R(\sigma)) = \sum_{\sigma \in S_n} p(\sigma) \cdot R(\sigma) = \sum_{k=1}^{n} \frac{\#_k}{n!} k$$

Runs

Erwartete Anzahl von Runs

$$2 \cdot \mathsf{E}(R(\sigma)) = 2\sum_{k=1}^{n} \frac{\#_k}{n!} k = \sum_{k=1}^{n} \frac{\#_k}{n!} k + \sum_{k=1}^{n} \overbrace{\frac{\#_{n-k+1}}{n!}(n-k+1)}^{\text{Indexvertauschung}}$$

$$= \sum_{k=1}^{n} \frac{\#_k}{n!} k + \sum_{k=1}^{n} \overbrace{\frac{\#_k}{n!}}^{\text{s. letzte Folie}} (n-k+1)$$

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

Daraus folgt:

$$\mathsf{E}(R(\sigma)) = \frac{n+1}{2}$$

Runs

Adaptives Sortieren

3 8 4 7 2 6 1 8

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8
3	8	4	7	2	6	1	8

Runs

Adaptives Sortieren

3 8 4 7 2 6 1 8

3 8 | 4 7 | 2 6 | 1 8 Idee: Mergesort!

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!

Runs

Adaptives Sortieren

3 8 4 7 2 6 1 8

3 8 | 4 7 | 2 6 | 1 8 Idee: Mergesort!

3

Runs

Adaptives Sortieren

3 8 4 7 2 6 1 8

3 8 | 4 7 | 2 6 | 1 8 Idee: Mergesort!

3 4

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!
3	4	7						

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!
3	4	7	8					

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!
3	4	7	8	1	2	6	8	

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!
3	4	7	8	1	2	6	8	

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!
3	4	7	8	1	2	6	8	
1								

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!
3	4	7	8	1	2	6	8	
1	2							

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!
3	4	7	8	1	2	6	8	
1	2	3	4	5	6	7	8	

Runs

Adaptives Sortieren

3	8	4	7	2	6	1	8	
3	8	4	7	2	6	1	8	Idee: Mergesort!
3	4	7	8	1	2	6	8	
1	2	3	4	5	6	7	8	

- Natürlicher Mergesort
- Laufzeit $O(n + n \log m_{\text{runs}})$

Adaptives Sortieren

Kennzahlen der Vorsortiertheit

Runs

- Nachteil: Globale Sortiertheit wird nicht erkannt
- Beispiel: 1 0 3 2 5 4 7 6 8
- Viele Runs, aber nur zwei ineinander verschränkte aufsteigende Teilsequenzen

Removals

- Removals: minimale Anzahl von Elementen, deren Löschen eine sortierte Folge erzeugt
- Das Beispiel hat eine längste aufsteigende Teilsequenz der Länge 5 $\rightarrow m_{\mathrm{rem}} = 4$
- Best case: $m_{\mathrm{rem}} = 0$; worst case: $m_{\mathrm{rem}} = n - 1$