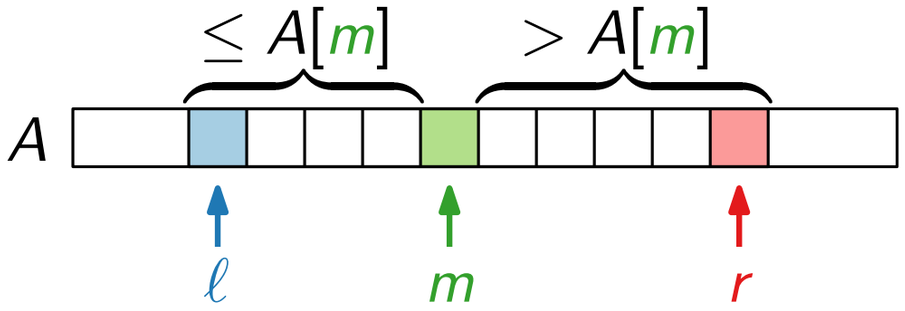
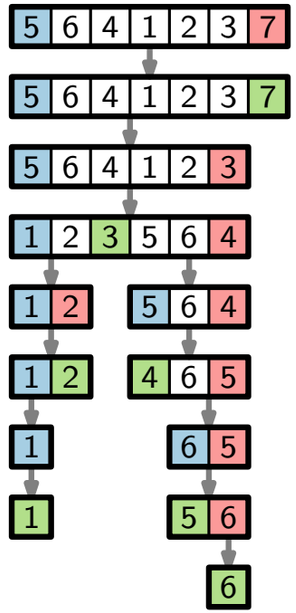
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen

Vorlesung 8: QuickSort





Alexander Wolff

Wintersemester 2025

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel:

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Und noch einmal: Sortieren!

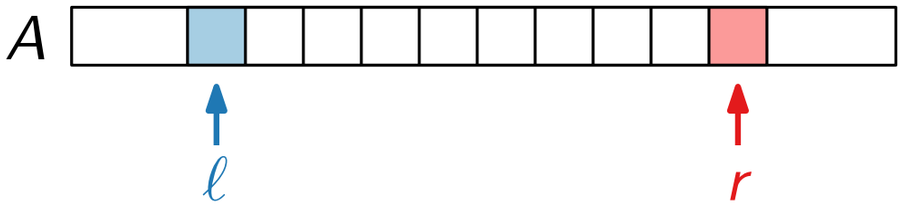
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \ldots r]$ wie folgt:



Und noch einmal: Sortieren!

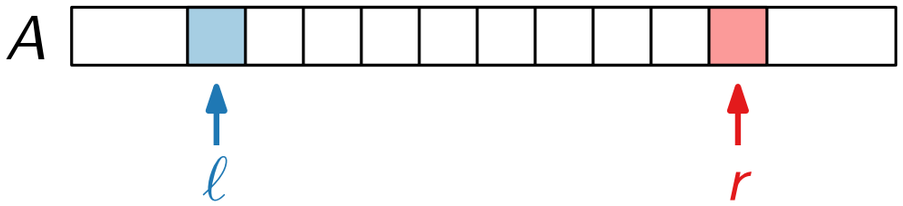
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \ldots r]$ wie folgt:



Teile:

Herrsche:

Kombiniere:

Und noch einmal: Sortieren!

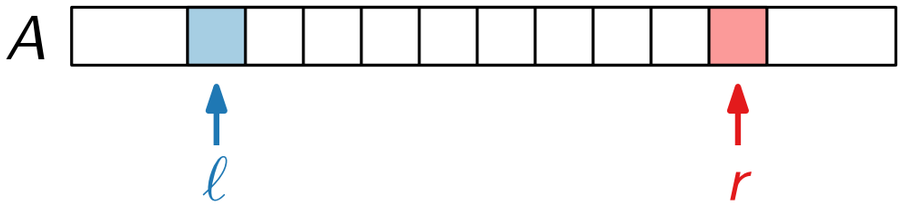
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \dots r]$ wie folgt:



Teile: Bestimme einen Index $m \in \{\ell, \dots, r\}$

Herrsche:

Kombiniere:

Und noch einmal: Sortieren!

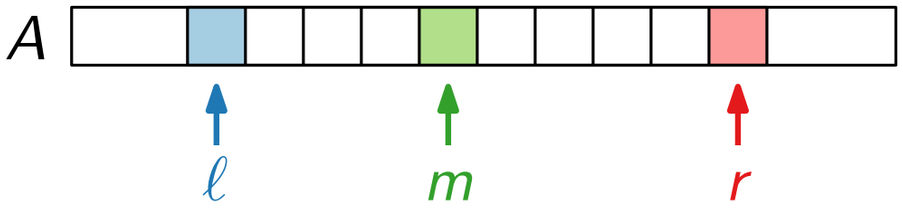
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \dots r]$ wie folgt:



Teile: Bestimme einen Index $m \in \{\ell, \dots, r\}$

Herrsche:

Kombiniere:

Und noch einmal: Sortieren!

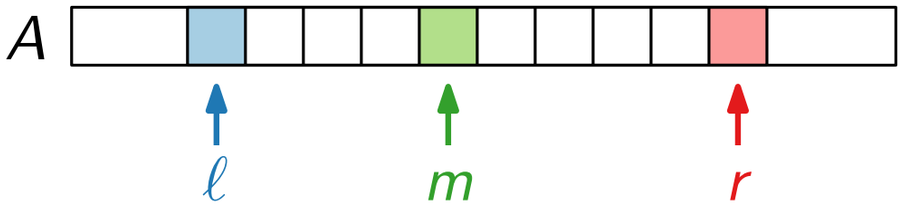
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \ldots r]$ wie folgt:



Teile: Bestimme einen Index $m \in \{\ell, \ldots, r\}$ und teile $A[\ell \ldots r]$ so auf, dass

Herrsche:

Kombiniere:

Und noch einmal: Sortieren!

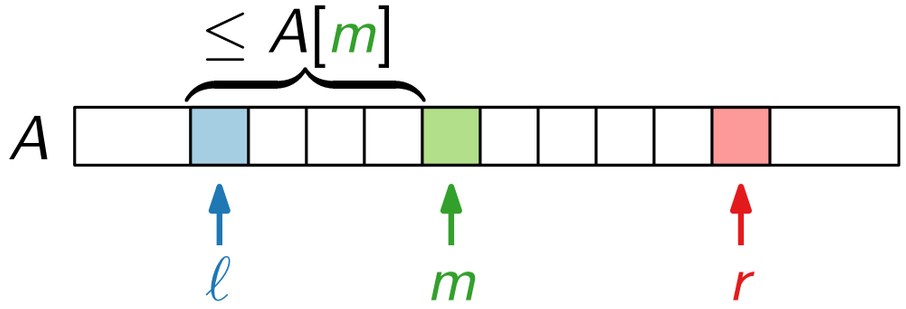
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \dots r]$ wie folgt:



Teile: Bestimme einen Index $m \in \{\ell, \dots, r\}$
und teile $A[\ell \dots r]$ so auf, dass
alle Elemente in $A[\ell \dots m-1]$ kleiner gleich $A[m]$ sind und

Herrsche:

Kombiniere:

Und noch einmal: Sortieren!

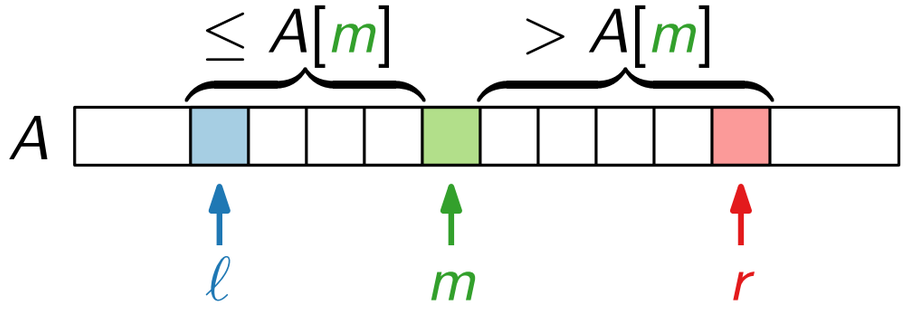
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \dots r]$ wie folgt:



Teile: Bestimme einen Index $m \in \{\ell, \dots, r\}$
und teile $A[\ell \dots r]$ so auf, dass
alle Elemente in $A[\ell \dots m-1]$ kleiner gleich $A[m]$ sind und
alle Elemente in $A[m+1 \dots r]$ größer als $A[m]$ sind.

Herrsche:

Kombiniere:

Und noch einmal: Sortieren!

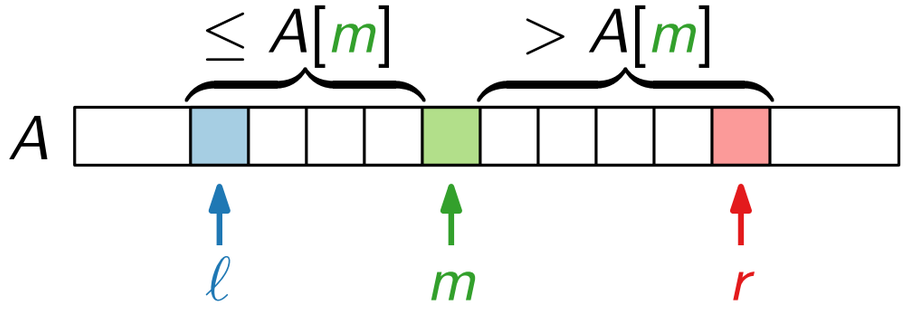
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \ldots r]$ wie folgt:



Teile: Bestimme einen Index $m \in \{\ell, \ldots, r\}$
und teile $A[\ell \ldots r]$ so auf, dass
alle Elemente in $A[\ell \ldots m-1]$ kleiner gleich $A[m]$ sind und
alle Elemente in $A[m+1 \ldots r]$ größer als $A[m]$ sind.

Herrsche: durch rekursives Sortieren der beiden Teilfelder.

Kombiniere:

Und noch einmal: Sortieren!

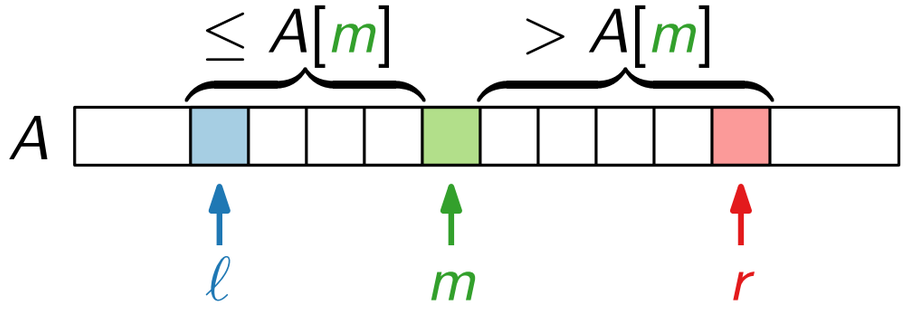
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \ldots r]$ wie folgt:



Teile: Bestimme einen Index $m \in \{\ell, \ldots, r\}$
und teile $A[\ell \ldots r]$ so auf, dass
alle Elemente in $A[\ell \ldots m-1]$ kleiner gleich $A[m]$ sind und
alle Elemente in $A[m+1 \ldots r]$ größer als $A[m]$ sind.

Herrsche: durch rekursives Sortieren der beiden Teilfelder.

Kombiniere: —

Und noch einmal: Sortieren!

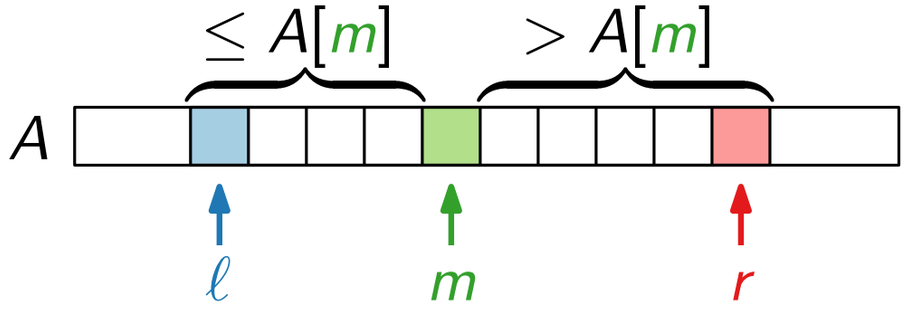
Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

Sortiere ein Teilfeld $A[\ell \ldots r]$ wie folgt:



Teile:

int Partition(A, ℓ, r) (liefert m zurück)

Bestimme einen Index $m \in \{\ell, \ldots, r\}$
und teile $A[\ell \ldots r]$ so auf, dass
alle Elemente in $A[\ell \ldots m-1]$ kleiner gleich $A[m]$ sind und
alle Elemente in $A[m+1 \ldots r]$ größer als $A[m]$ sind.

Herrsche: durch rekursives Sortieren der beiden Teilfelder.

Kombiniere: —

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

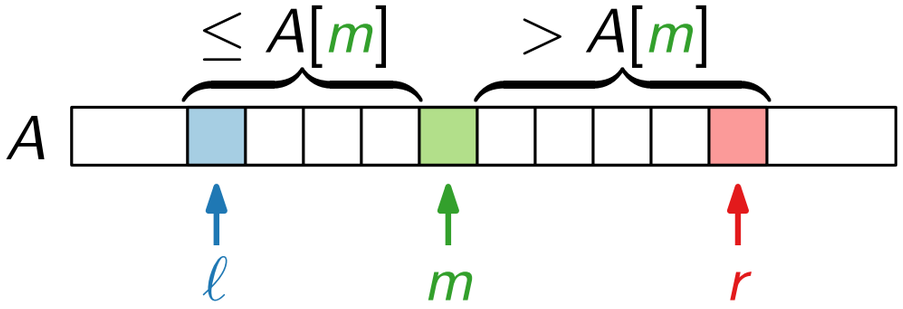
QuickSort(int[] A, int ℓ, r)

Sortiere ein Teilfeld $A[\ell \dots r]$ wie folgt:

Teile:

int Partition(A, ℓ, r) (liefert m zurück)

Bestimme einen Index $m \in \{\ell, \dots, r\}$
und teile $A[\ell \dots r]$ so auf, dass
alle Elemente in $A[\ell \dots m-1]$ kleiner gleich $A[m]$ sind und
alle Elemente in $A[m+1 \dots r]$ größer als $A[m]$ sind.



Herrsche: durch rekursives Sortieren der beiden Teilfelder.

Kombiniere: —

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

⊕ gute Worst-Case-Laufzeit (durch **Teile-und-Herrsche**)

⊖ kein **in-situ**-Verfahren (benötigt extra Felder beim Mergen)

Ziel: **Teile-und-Herrsche**-Verfahren, das trotzdem **in-situ** sortiert!

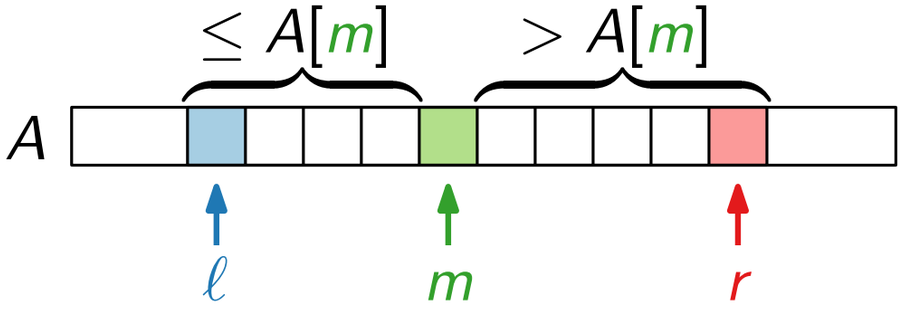
QuickSort(int[] A, int ℓ, r)

Sortiere ein Teilfeld $A[\ell \dots r]$ wie folgt:

Teile:

int Partition(A, ℓ, r) (liefert m zurück)

Bestimme einen Index $m \in \{\ell, \dots, r\}$
und teile $A[\ell \dots r]$ so auf, dass
alle Elemente in $A[\ell \dots m-1]$ kleiner gleich $A[m]$ sind und
alle Elemente in $A[m+1 \dots r]$ größer als $A[m]$ sind.



Herrsche: durch rekursives Sortieren der beiden Teilfelder.

Kombiniere: —

Aufgabe.
Schreiben Sie QuickSort in Pseudocode unter Verwendung von Partition(A, ℓ, r)!

QuickSort

$\textsc{QuickSort}(A, \ell = 1, r = A.length)$

QuickSort

$\textsc{QuickSort}(A, \ell = 1, r = A.length)$

if $\ell < r$ **then**

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
```

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
```

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
```

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

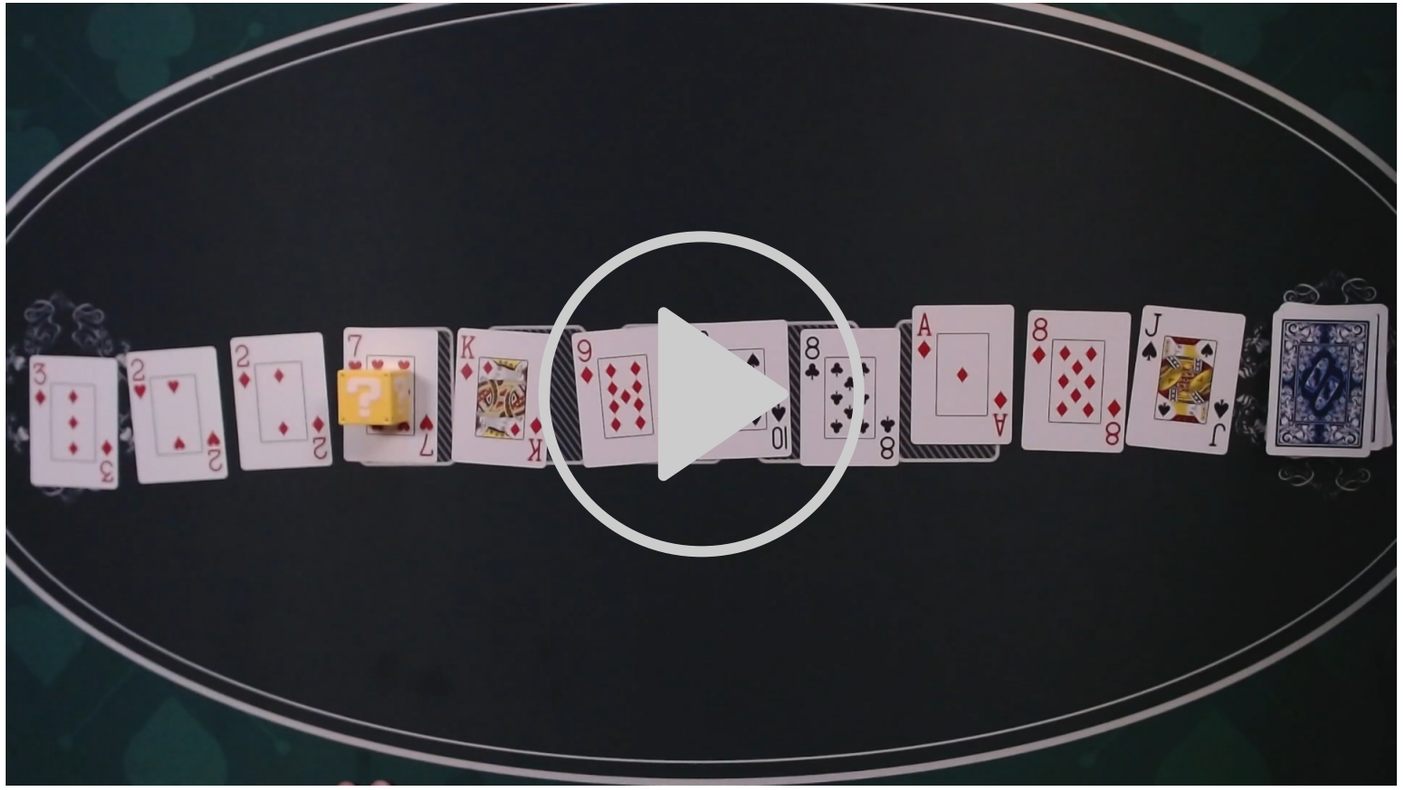
Was passiert hier?

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Was passiert hier?

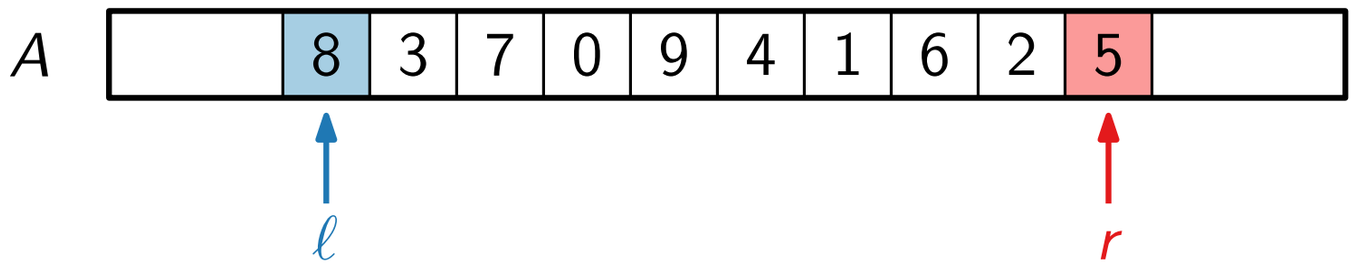


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

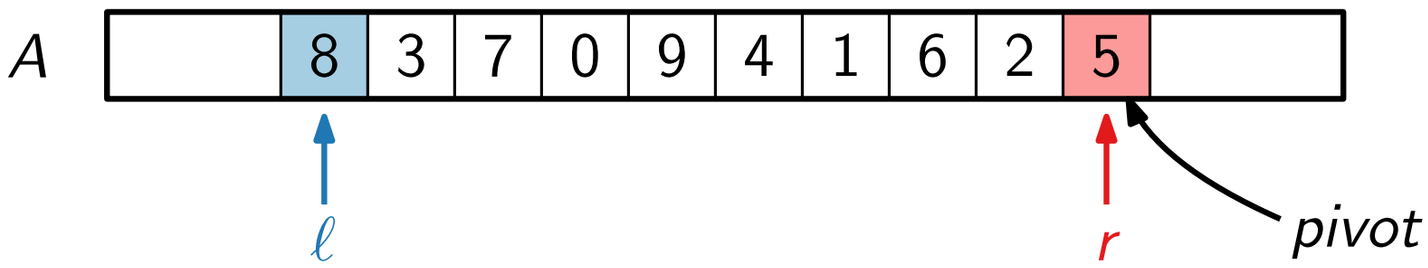


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

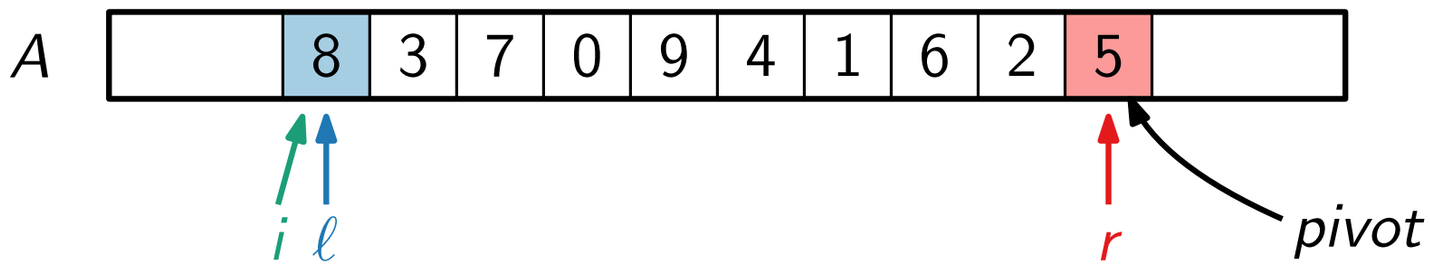


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

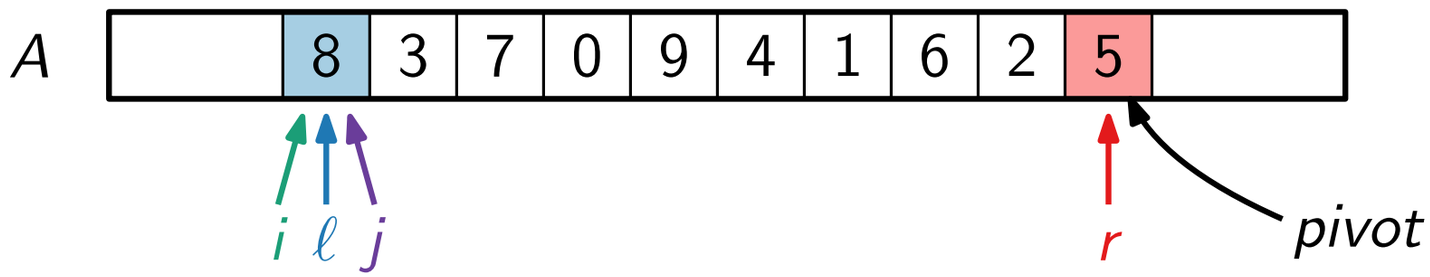


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

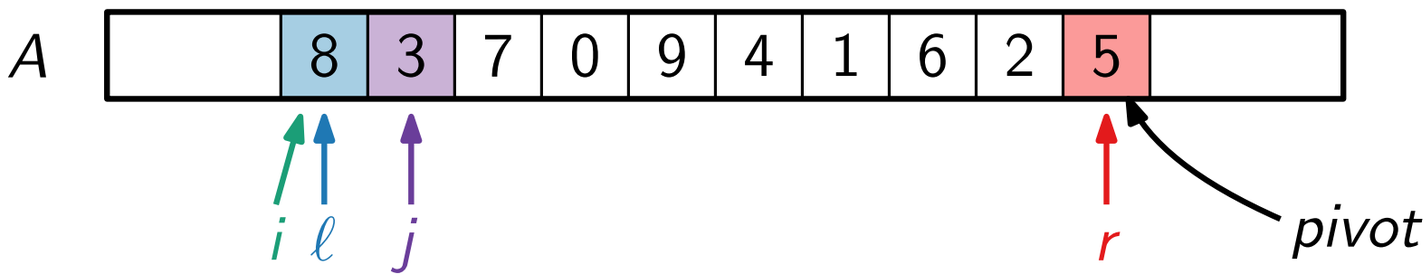


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

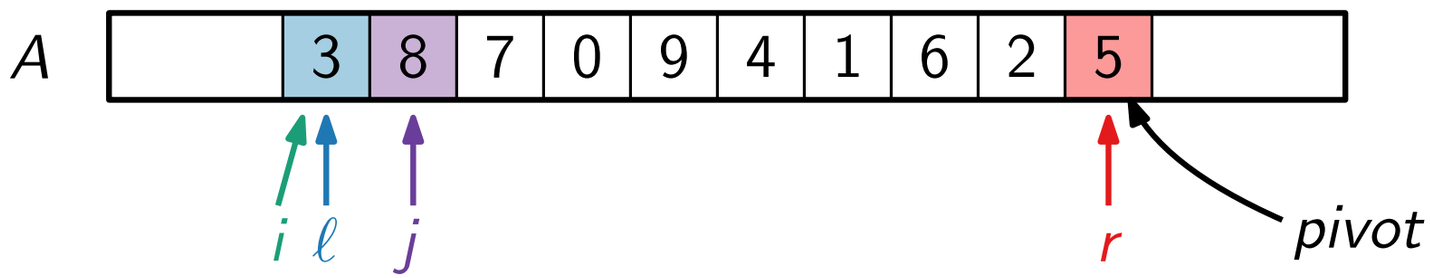


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

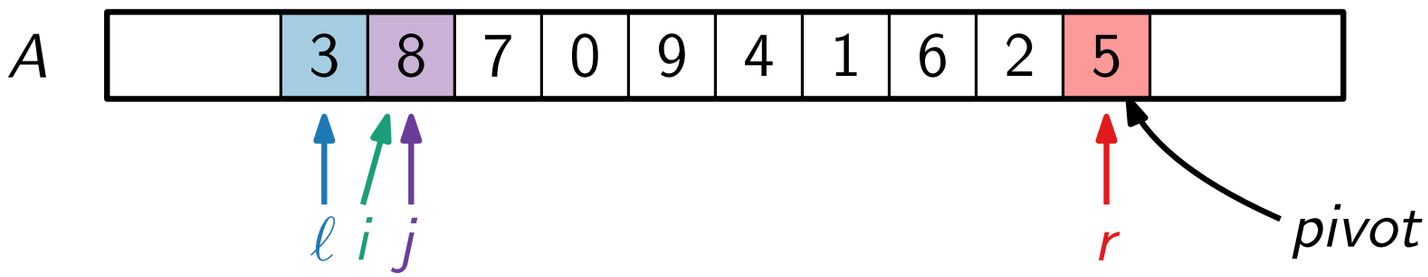


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

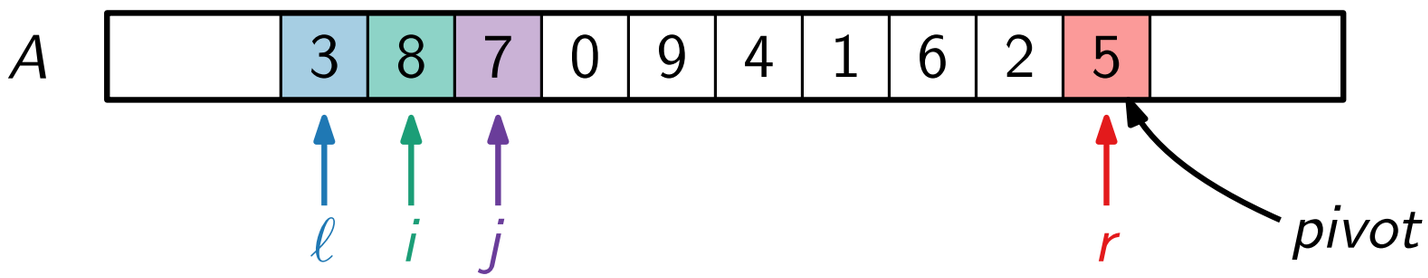


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

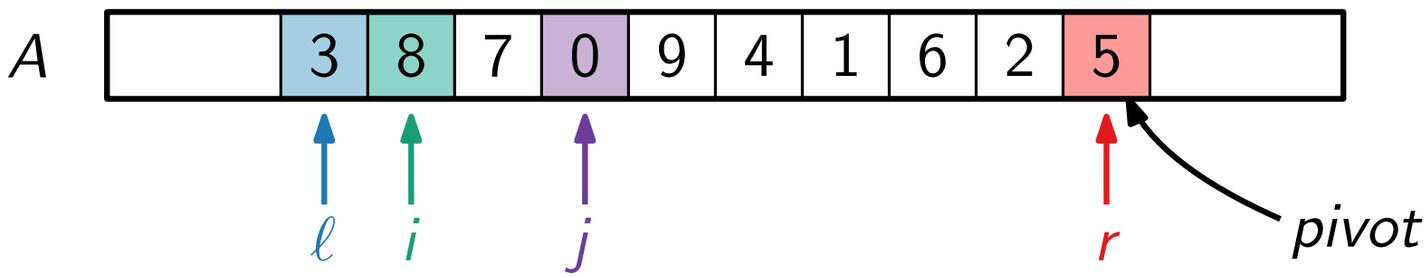


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

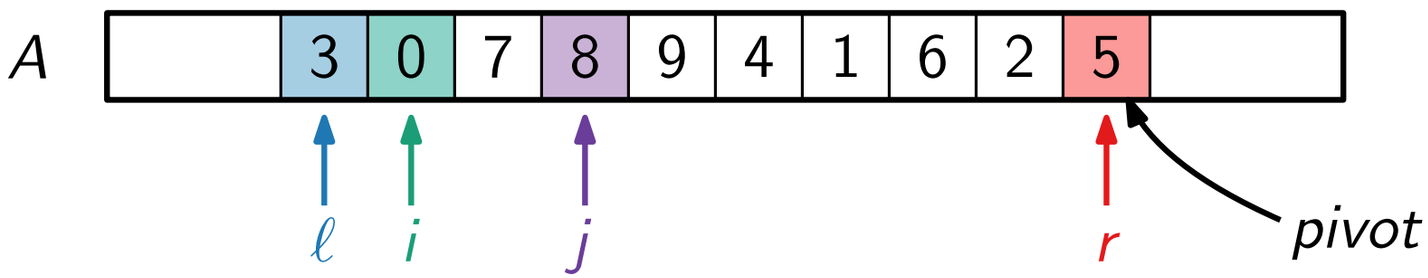


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

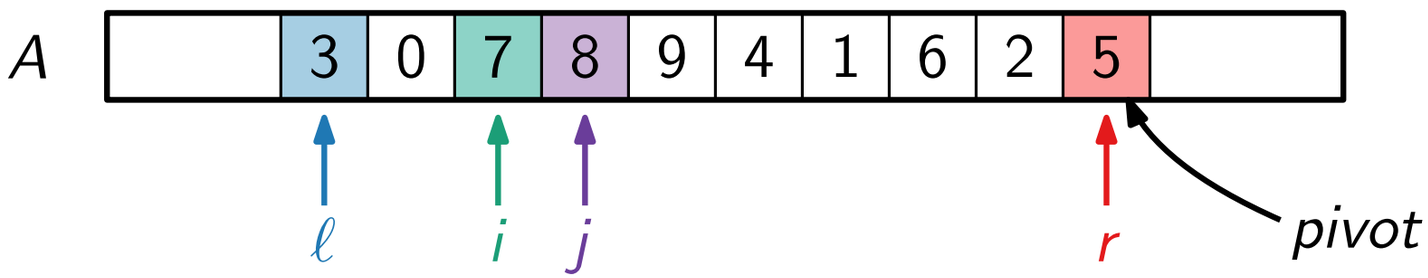


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

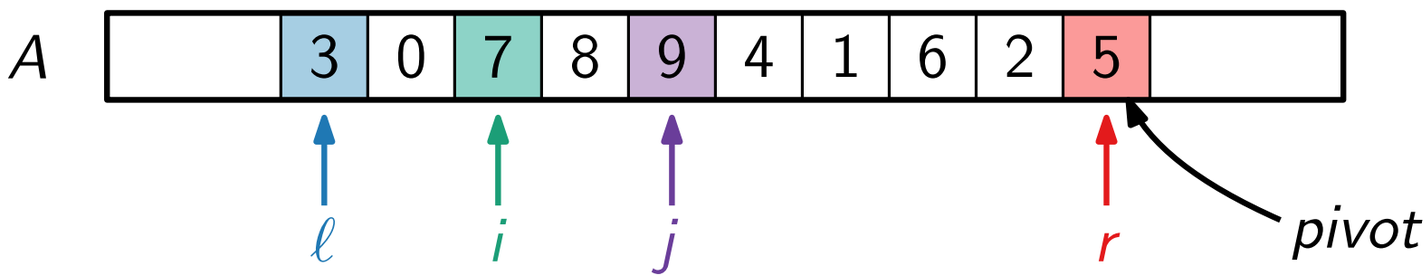


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

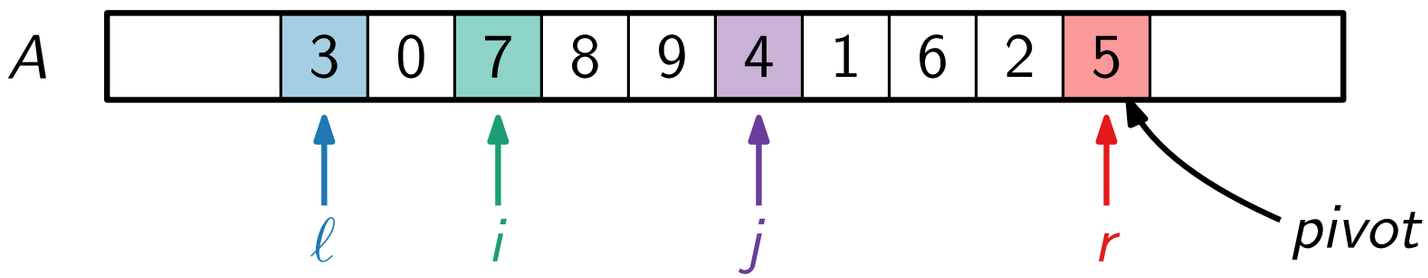


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

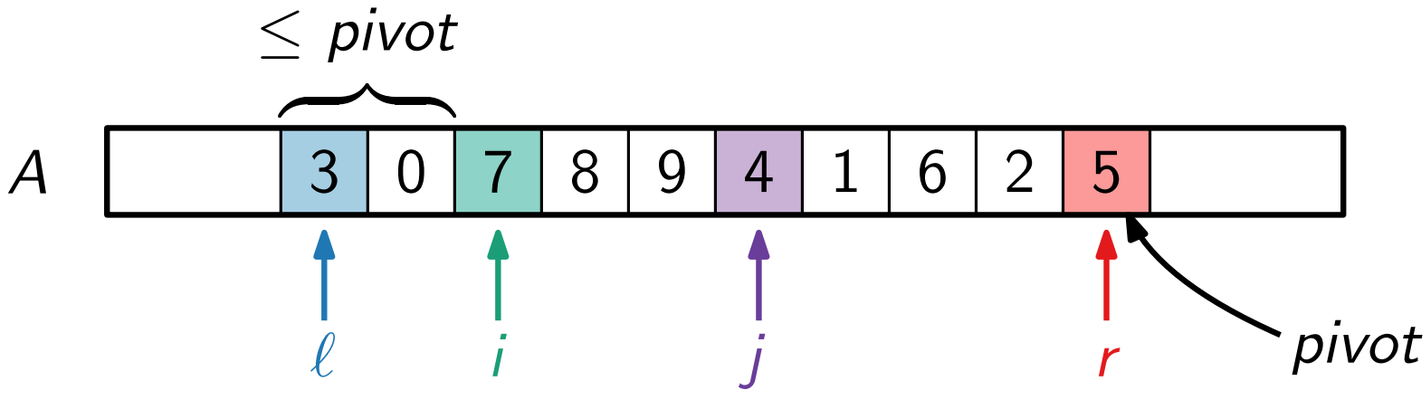


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Was passiert hier?

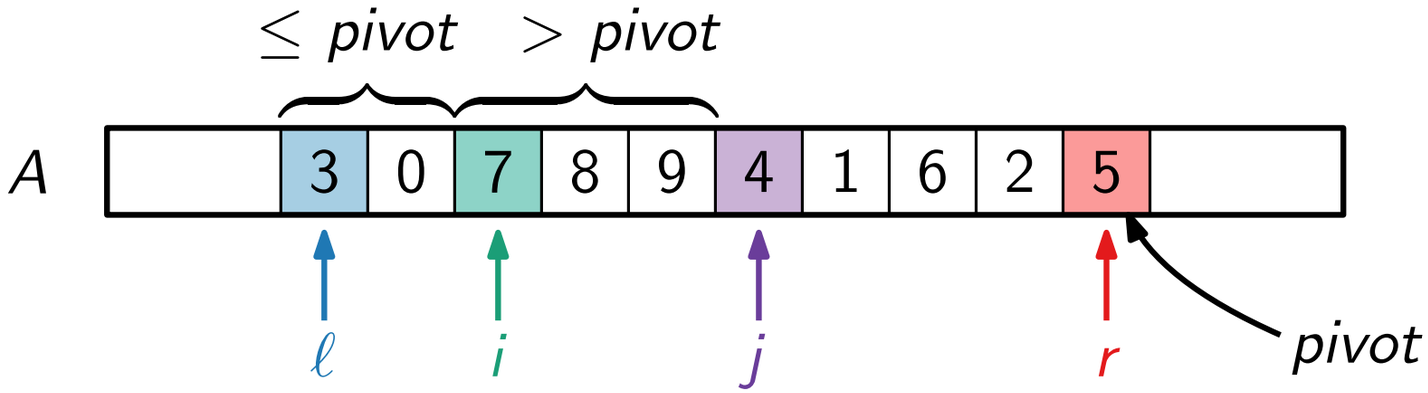


QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

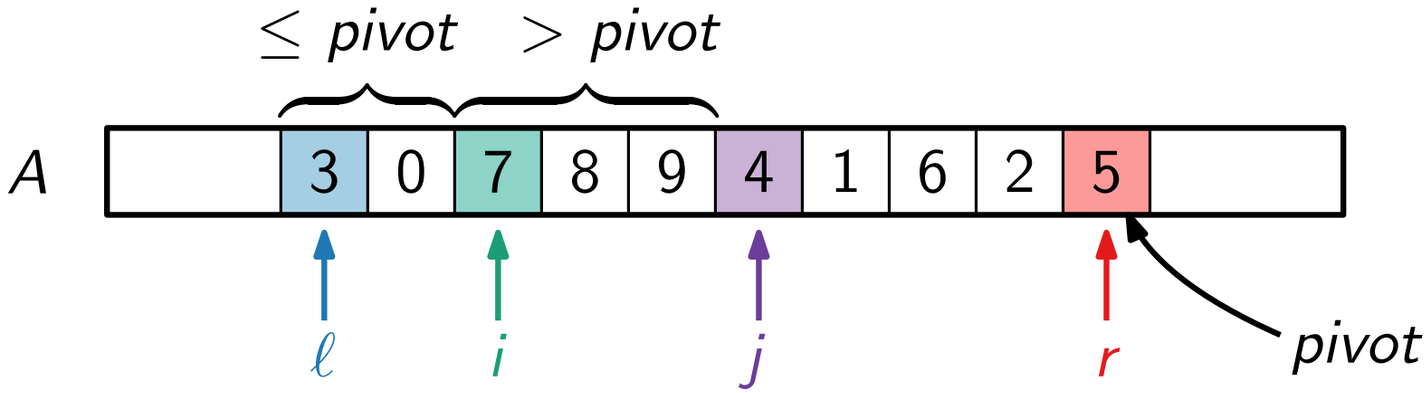
Was passiert hier?



QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

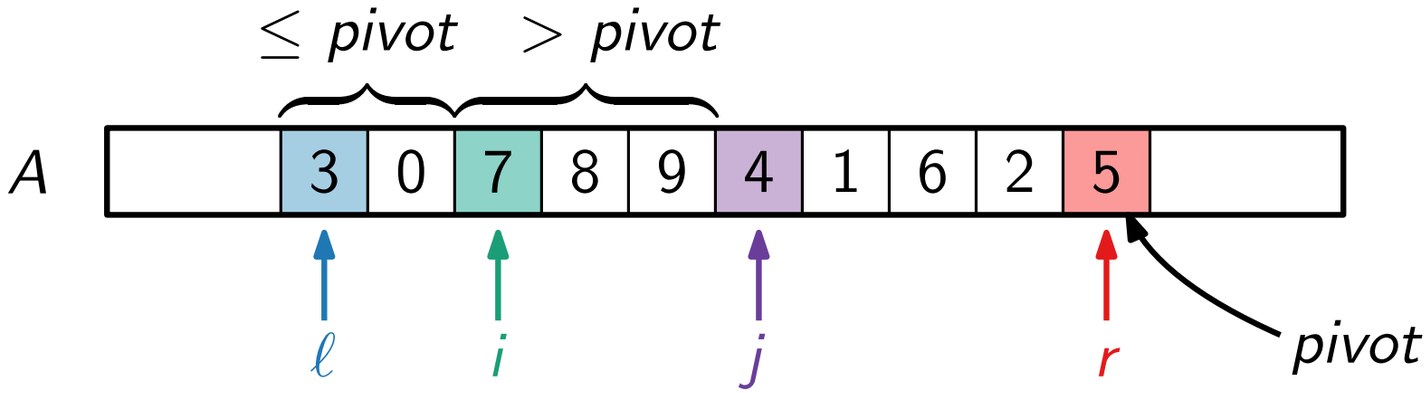


Schleifeninvariante:

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



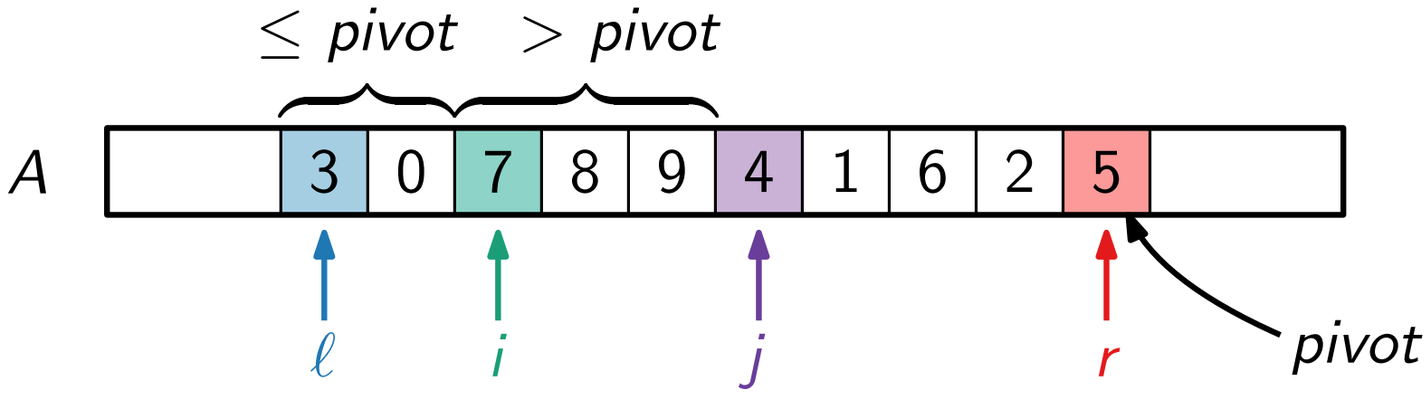
Schleifeninvariante:

(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Schleifeninvariante:

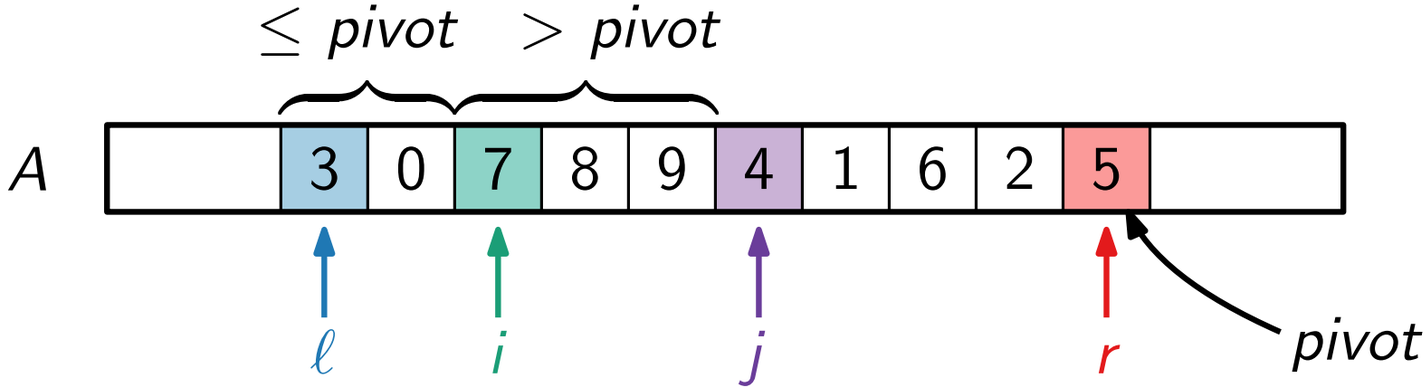
(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

(ii) Für $k = i, \ldots, j-1$ gilt: $A[k] > pivot$.

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Schleifeninvariante:

(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

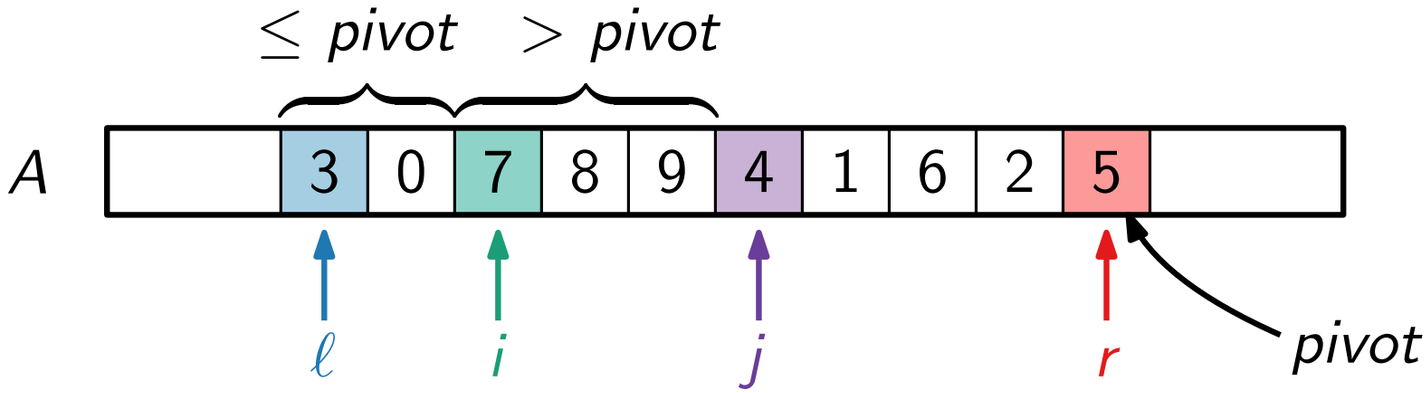
(ii) Für $k = i, \ldots, j-1$ gilt: $A[k] > pivot$.

(iii) $A[r] = pivot$.

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Schleifeninvariante:

(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

(ii) Für $k = i, \ldots, j-1$ gilt: $A[k] > pivot$.

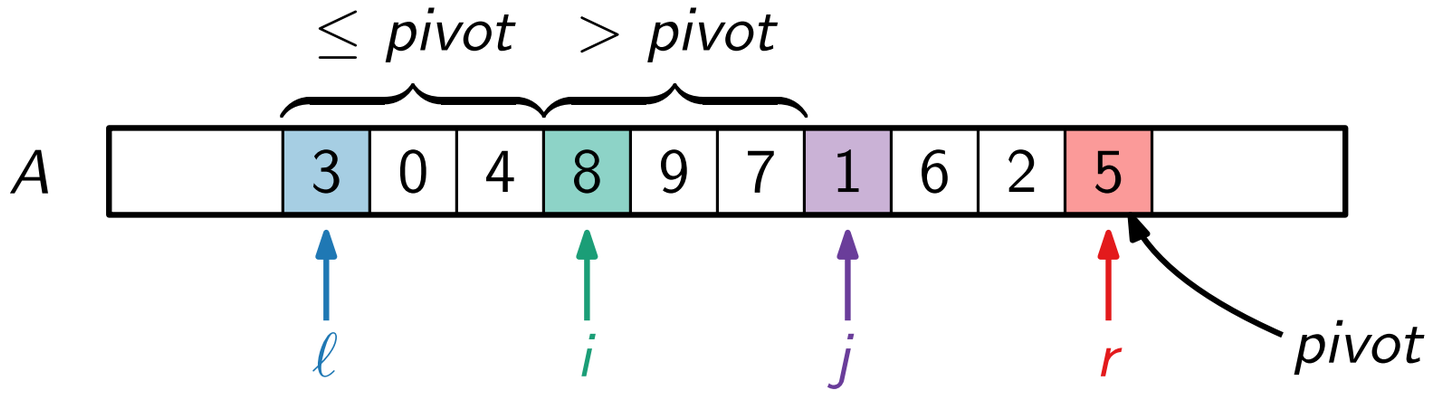
(iii) $A[r] = pivot$.

(iv) $A[\ell \ldots j-1]$ enthält die gleichen Elemente wie zu Beginn.

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Schleifeninvariante:

(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

(ii) Für $k = i, \ldots, j-1$ gilt: $A[k] > pivot$.

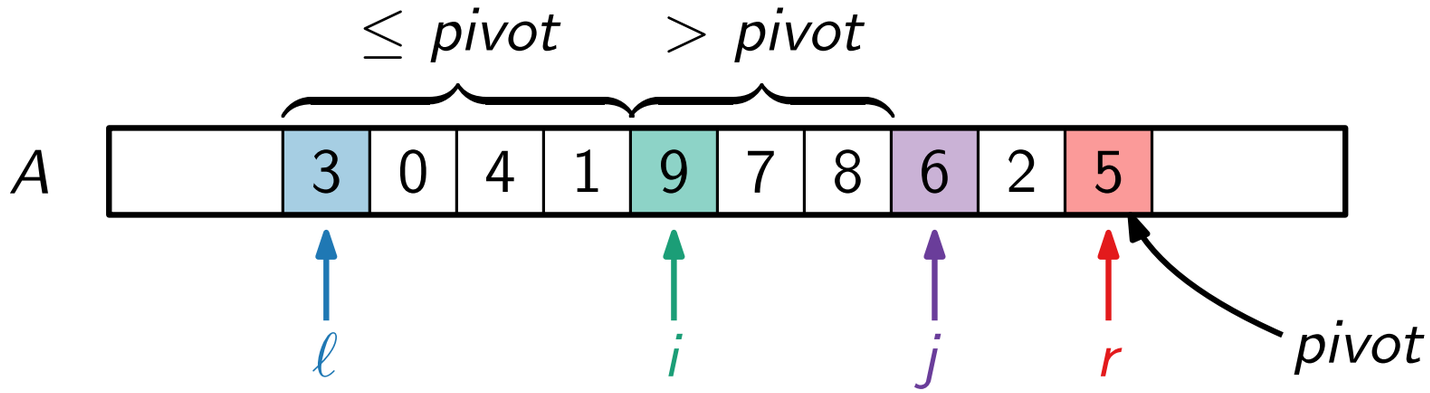
(iii) $A[r] = pivot$.

(iv) $A[\ell \ldots j-1]$ enthält die gleichen Elemente wie zu Beginn.

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
      m = Partition(A, ℓ, r)
      QuickSort(A, ℓ, m − 1)
      QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
      if A[j] ≤ pivot then
          A[i] ↔ A[j]
          i = i + 1
  A[i] ↔ A[r]
  return i
```



Schleifeninvariante:

(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

(ii) Für $k = i, \ldots, j-1$ gilt: $A[k] > pivot$.

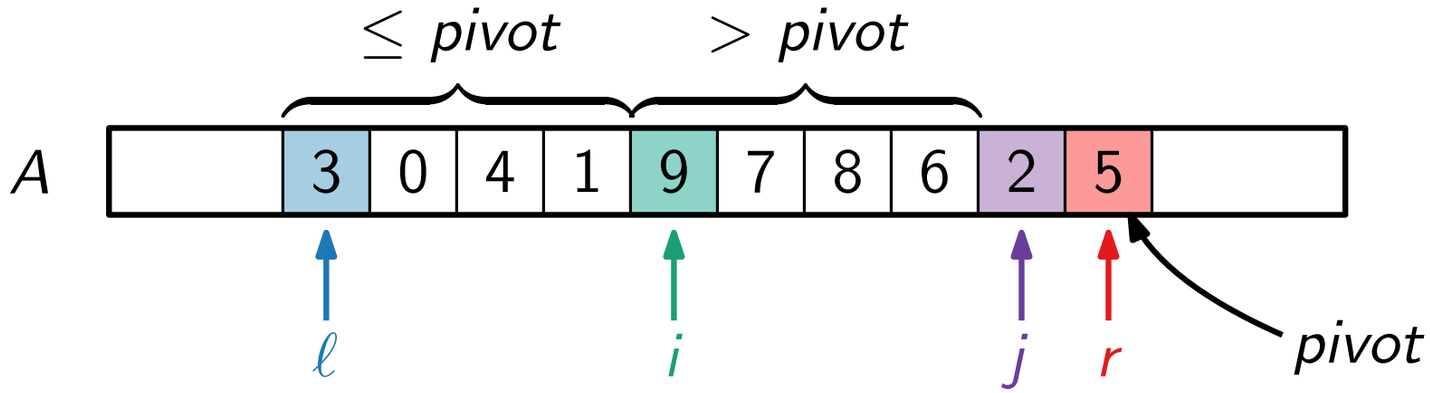
(iii) $A[r] = pivot$.

(iv) $A[\ell \ldots j-1]$ enthält die gleichen Elemente wie zu Beginn.

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Schleifeninvariante:

(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

(ii) Für $k = i, \ldots, j-1$ gilt: $A[k] > pivot$.

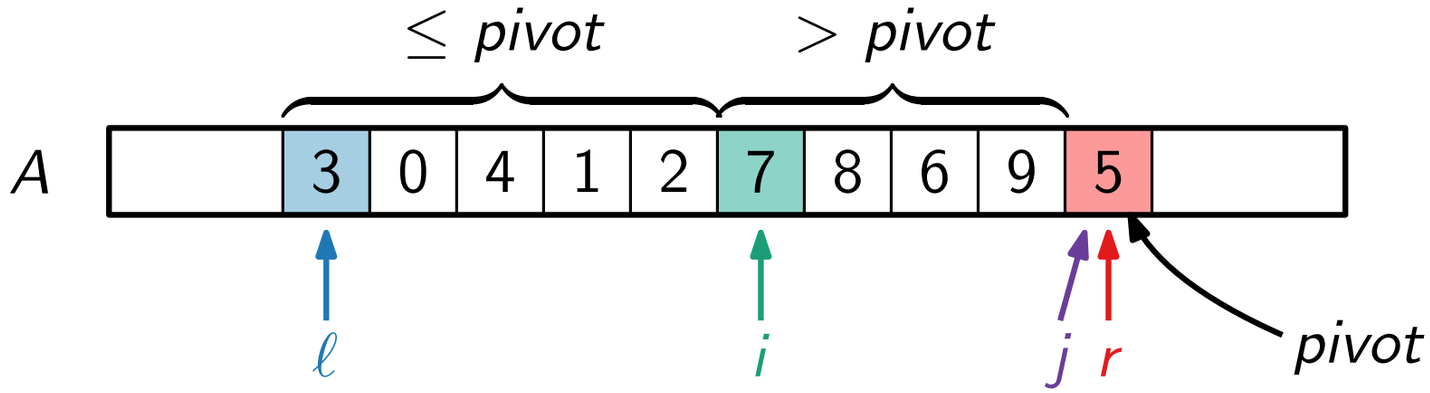
(iii) $A[r] = pivot$.

(iv) $A[\ell \ldots j-1]$ enthält die gleichen Elemente wie zu Beginn.

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Schleifeninvariante:

(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

(ii) Für $k = i, \ldots, j-1$ gilt: $A[k] > pivot$.

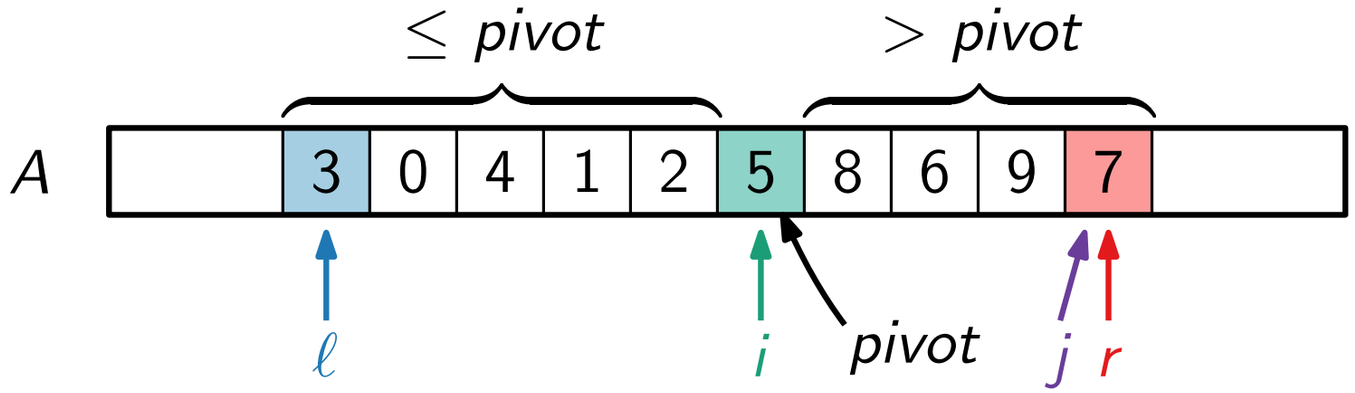
(iii) $A[r] = pivot$.

(iv) $A[\ell \ldots j-1]$ enthält die gleichen Elemente wie zu Beginn.

QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Schleifeninvariante:

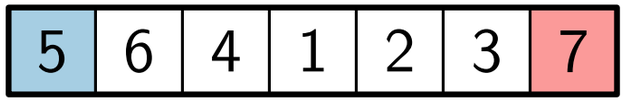
(i) Für $k = \ell, \ldots, i-1$ gilt: $A[k] \leq pivot$.

(ii) Für $k = i, \ldots, j-1$ gilt: $A[k] > pivot$.

(iii) $A[r] = pivot$.

(iv) $A[\ell \ldots j-1]$ enthält die gleichen Elemente wie zu Beginn.

Ein Beispiel



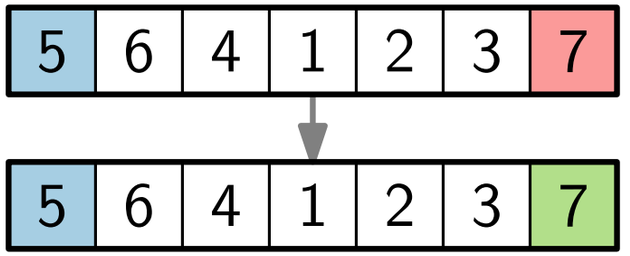

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

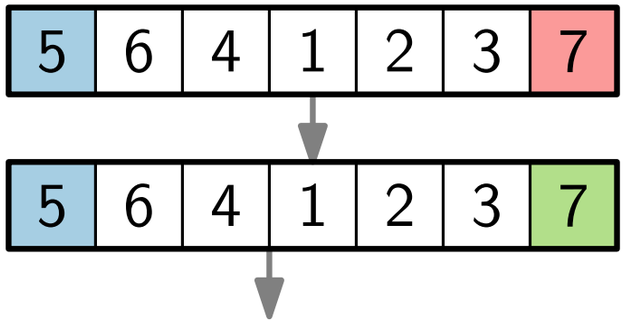
```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

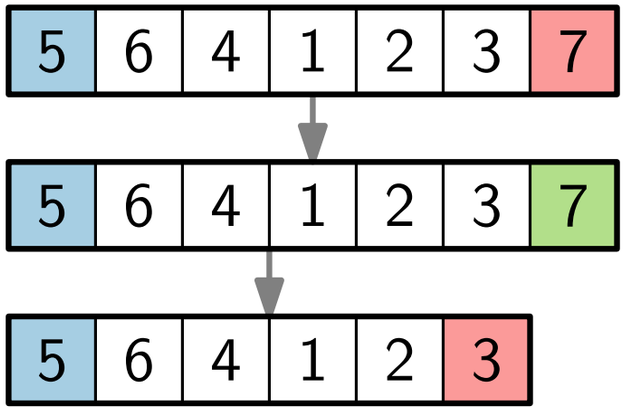
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

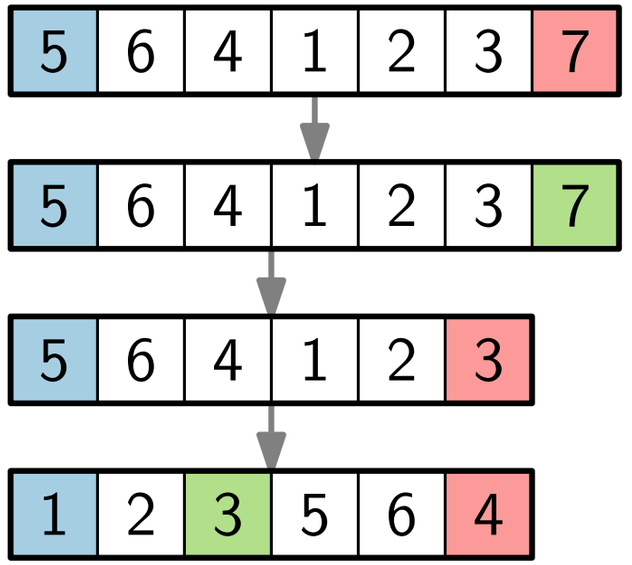
```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

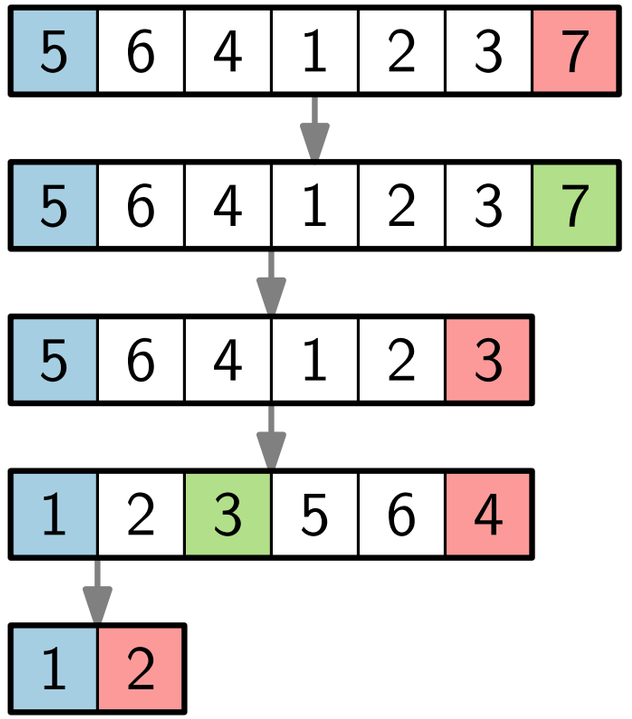
```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

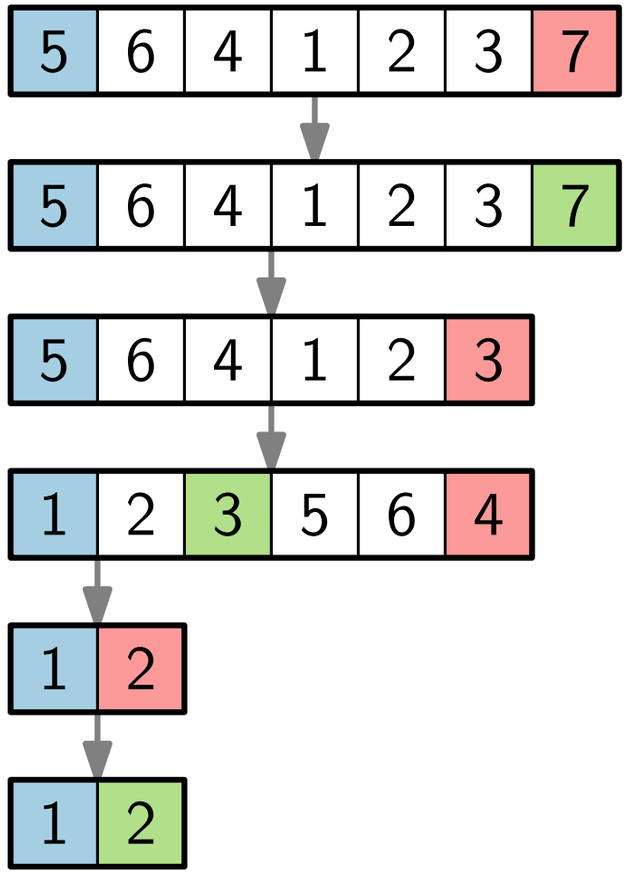
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

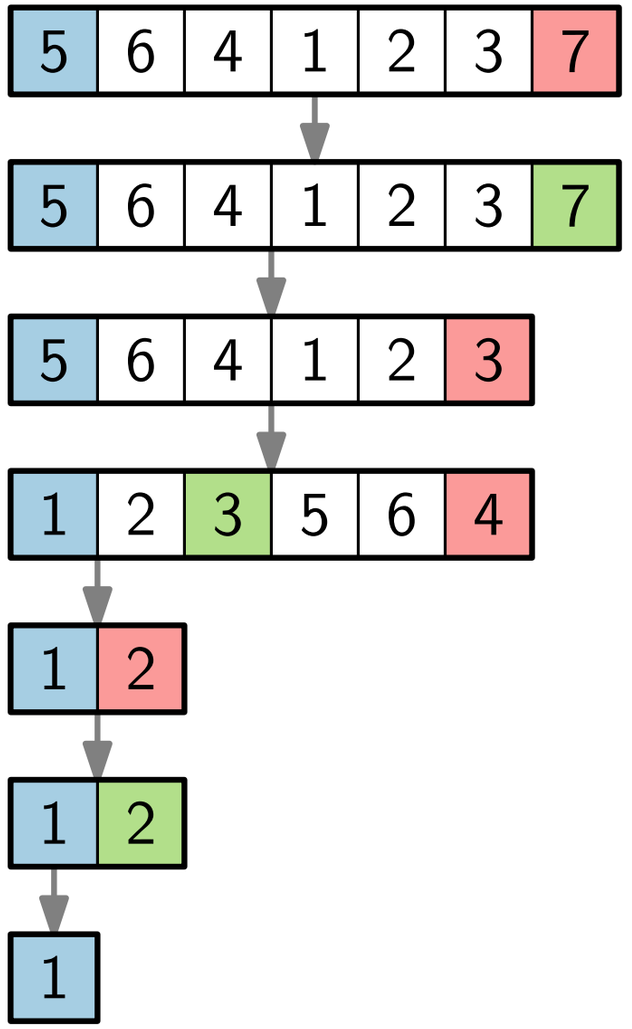
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

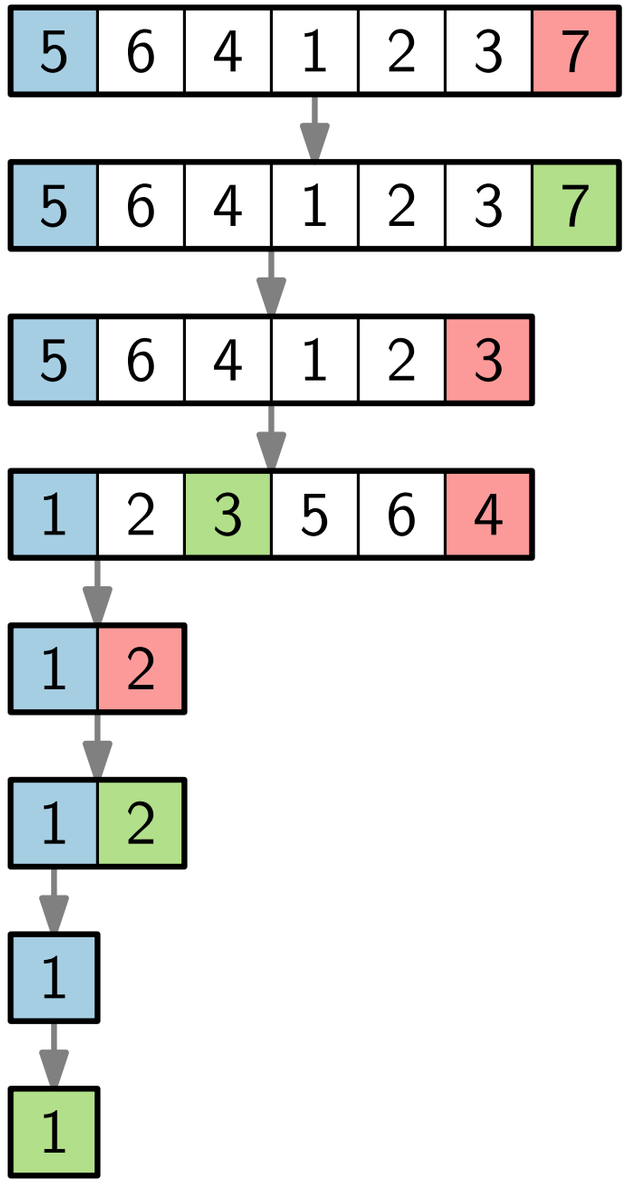
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

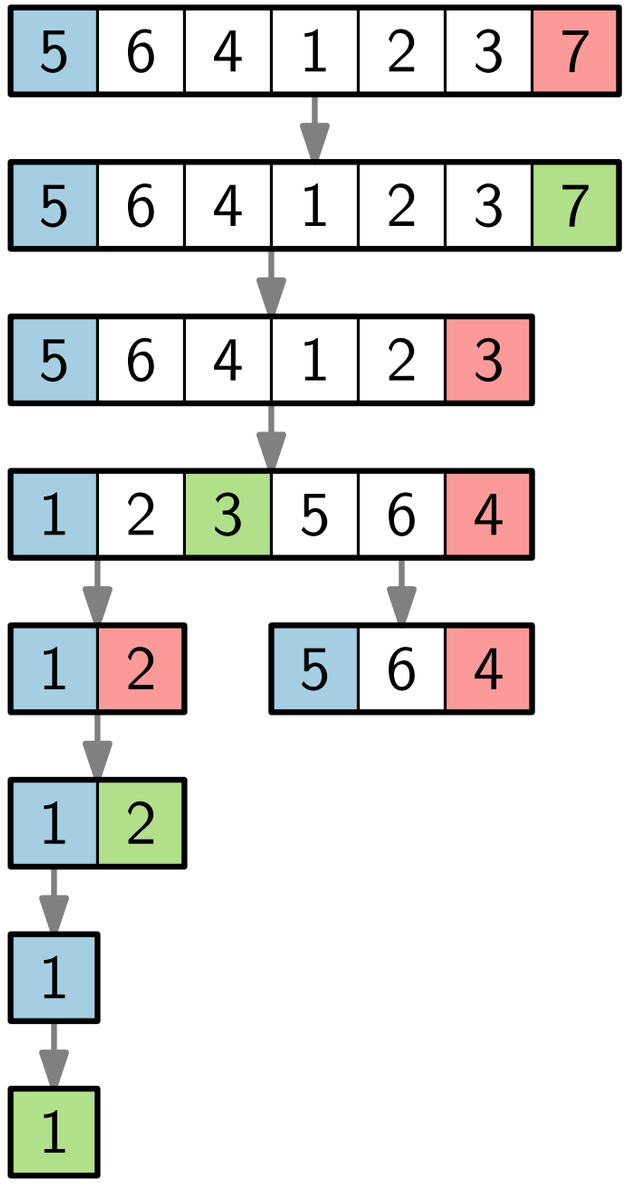
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

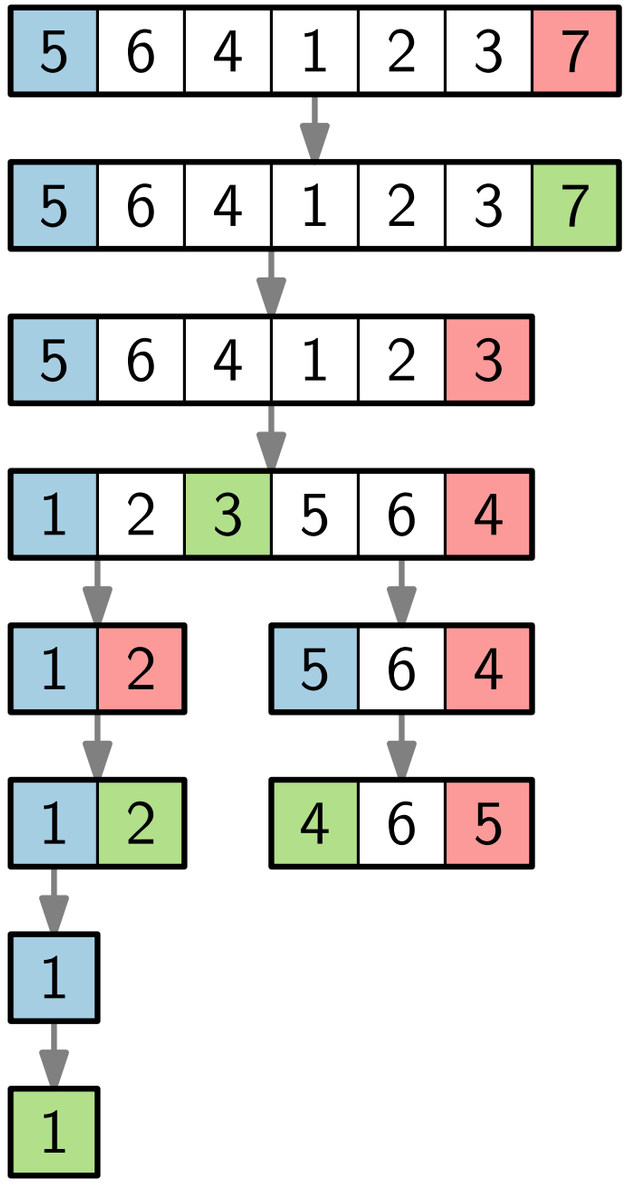
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

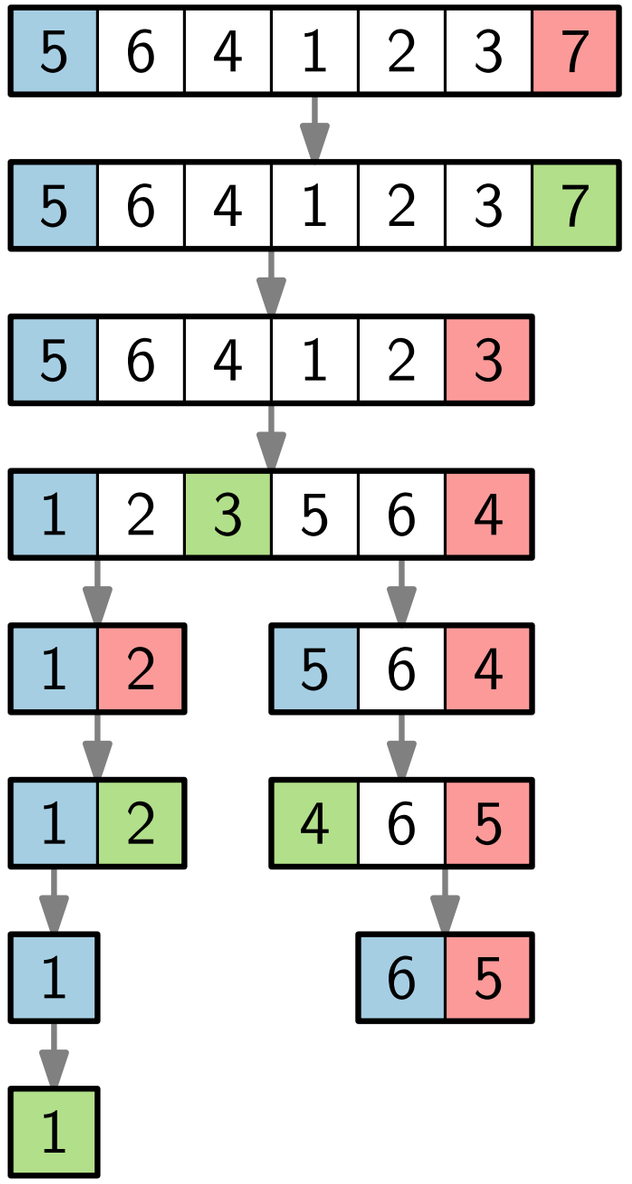
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

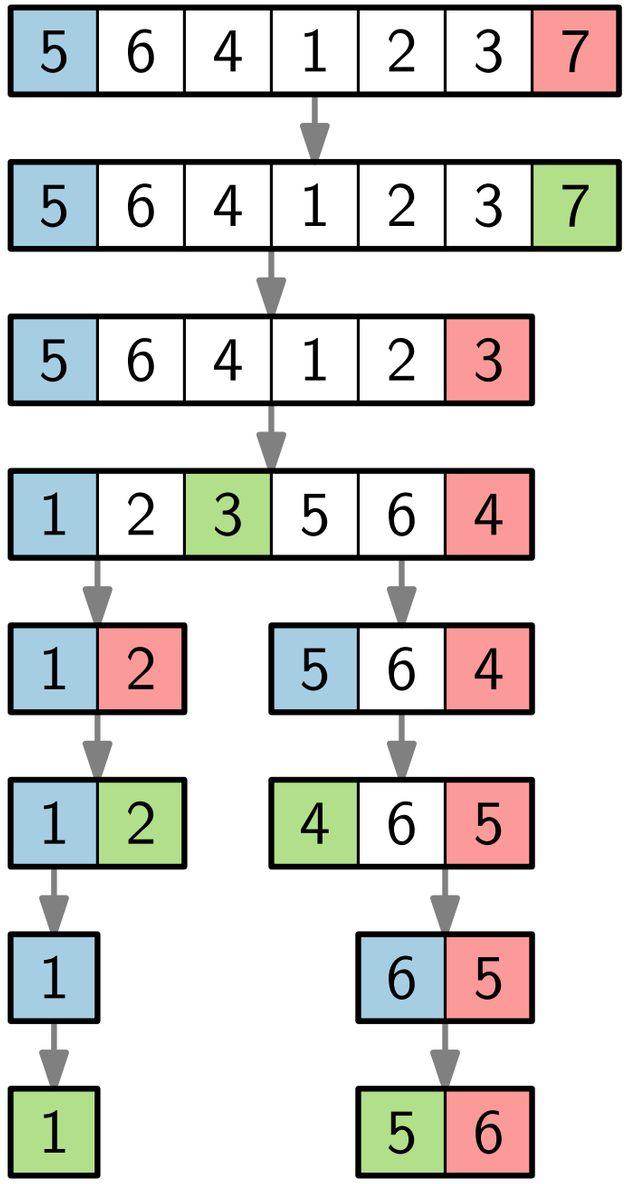
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

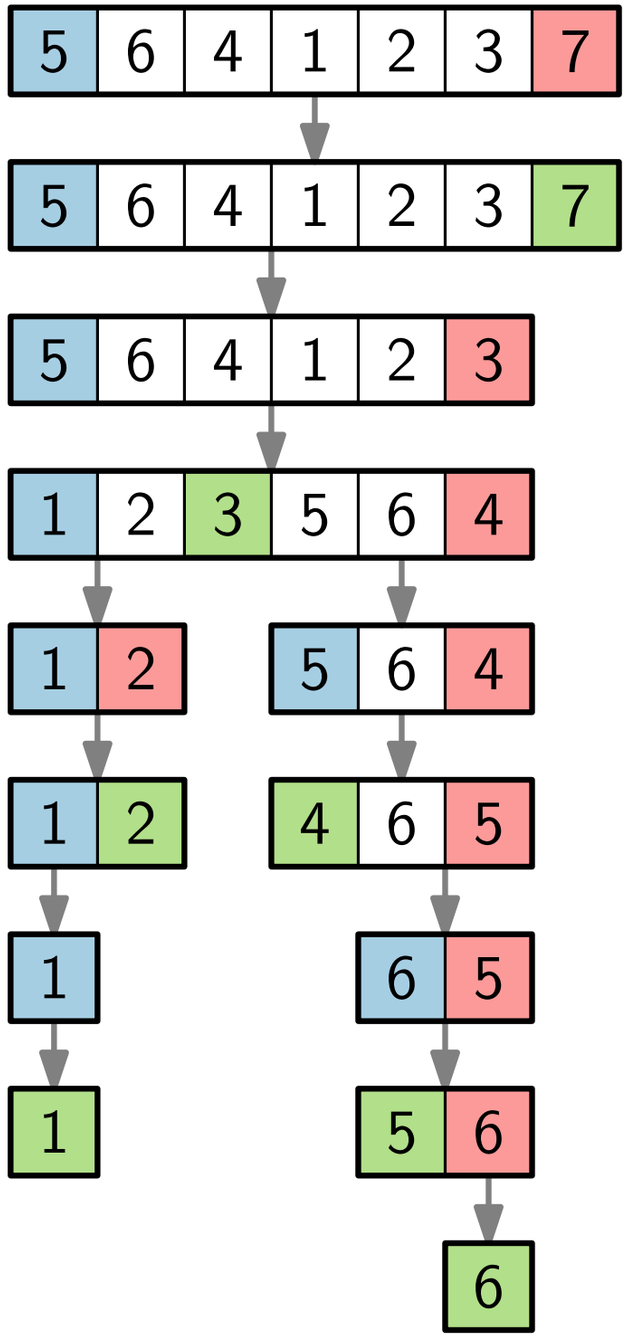
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

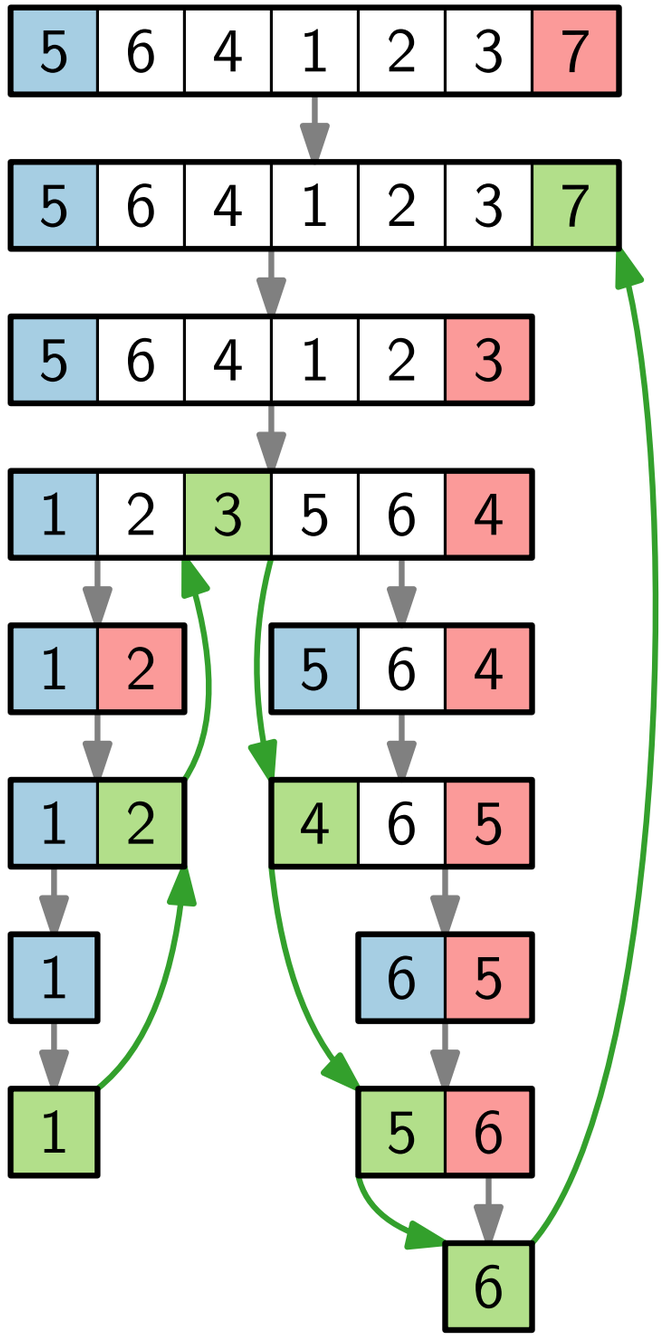
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

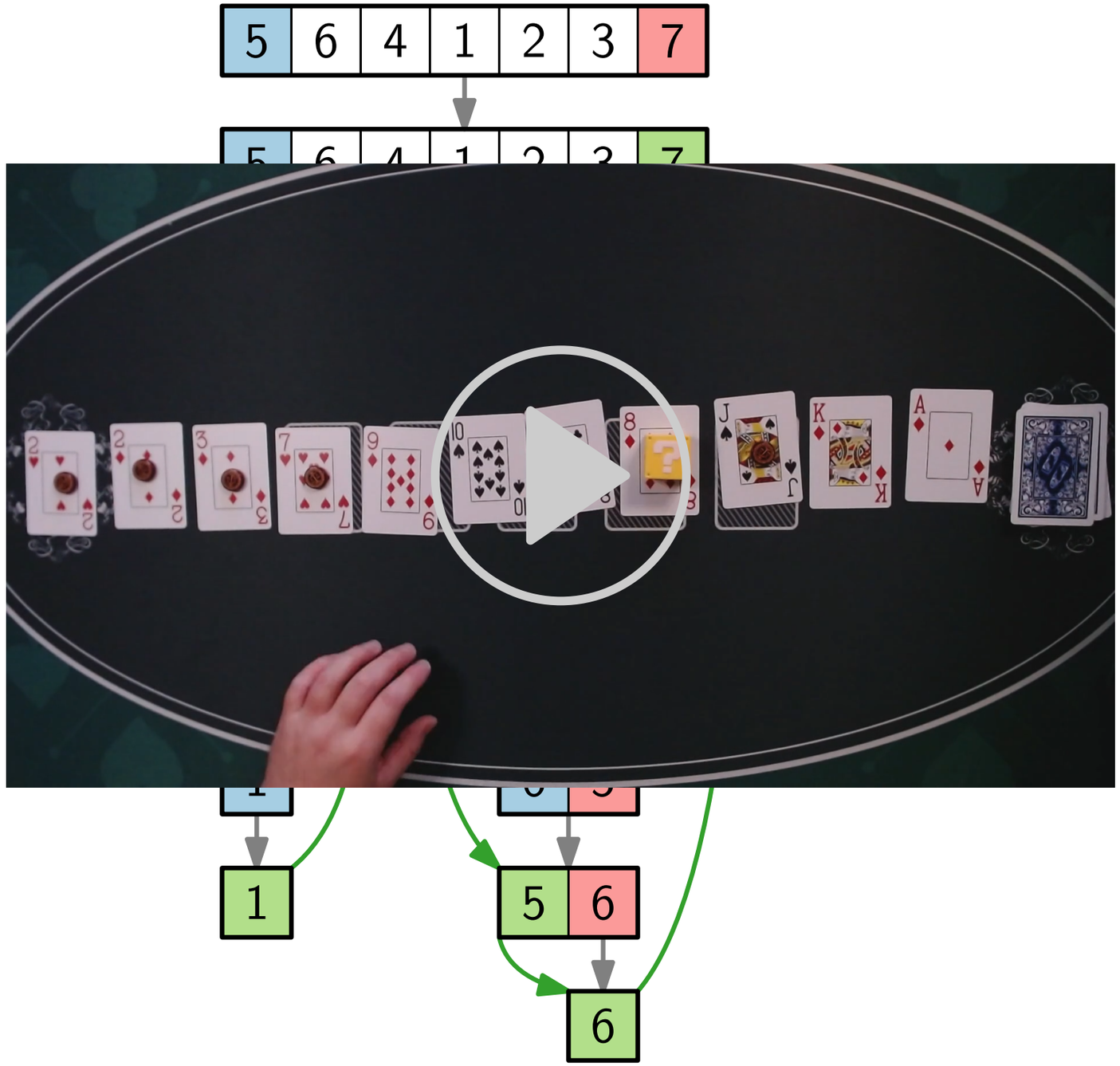
```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
    if ℓ < r then
        m = Partition(A, ℓ, r)
        QuickSort(A, ℓ, m − 1)
        QuickSort(A, m + 1, r)
```

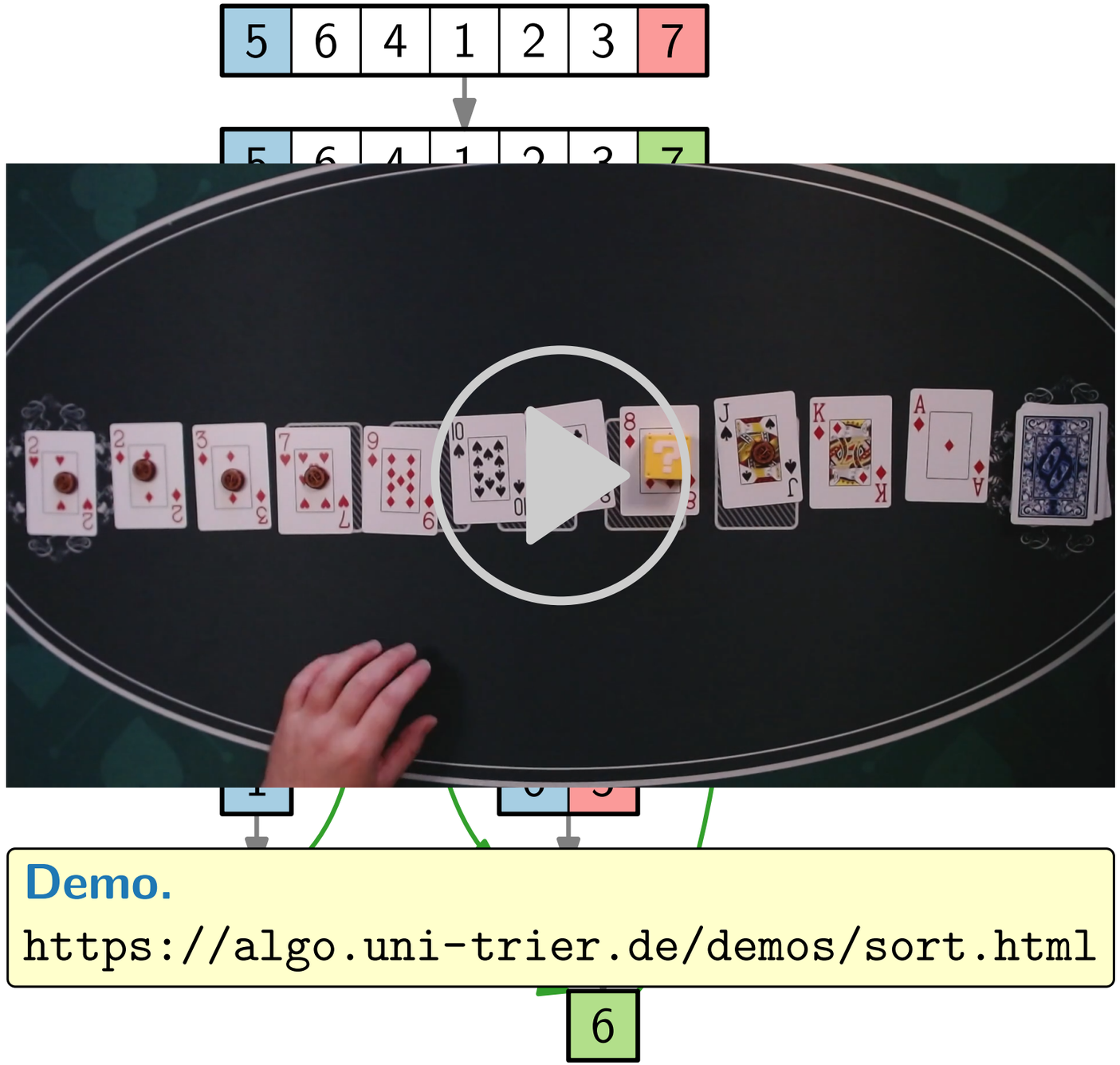
```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```



Ein Beispiel

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```



Demo.

`https://algo.uni-trier.de/demos/sort.html`

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** Vergleiche.

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$T_{\mathsf{QS}}(n) =$

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$T_{QS}(n) = T_{QS}(m-1)$

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m)$$

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$T_{QS}(n) = T_{QS}(0) + T_{QS}(n-1) + n - 1$$

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m 0 mmer erstes Element

$$T_{QS}(n) = T_{QS}(0) + T_{QS}(n-1) + n - 1$$

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m 0 immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \end{aligned}$$

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \end{aligned}$$

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n - 2 + n - 1 \end{aligned}$$

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n - 2 + n - 1 \end{aligned}$$

0

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n - 2 + n - 1 \\ &= \sum_{i=0}^{n-1} i \end{aligned}$$

0

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n - 2 + n - 1 \\ &= \sum_{i=0}^{n-1} i \in \Theta(n^2) \end{aligned}$$

2. Extremfall: m immer mittleres Element

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n - 2 + n - 1 \\ &= \sum_{i=0}^{n-1} i \in \Theta(n^2) \end{aligned}$$

2. Extremfall: m immer mittleres Element

$T_{QS}(n) \approx$

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n - 2 + n - 1 \\ &= \sum_{i=0}^{n-1} i \in \Theta(n^2) \end{aligned}$$

2. Extremfall: m immer mittleres Element

$$T_{QS}(n) \approx 2T_{QS}(n/2) + n - 1$$

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n - 2 + n - 1 \\ &= \sum_{i=0}^{n-1} i \in \Theta(n^2) \end{aligned}$$

2. Extremfall: m immer mittleres Element

$$T_{QS}(n) \approx 2T_{QS}(n/2) + n - 1 \in$$

siehe MergeSort

Laufzeit

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(int[] A, int ℓ, int r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      A[i] ↔ A[j]
      i = i + 1
  A[i] ↔ A[r]
  return i
```

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt **immer** $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n - 1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n - 1 \\ &= (T_{QS}(n-2) + n - 2) + n - 1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n - 2 + n - 1 \\ &= \sum_{i=0}^{n-1} i \in \Theta(n^2) \end{aligned}$$

2. Extremfall: m immer mittleres Element

$$T_{QS}(n) \approx 2T_{QS}(n/2) + n - 1 \in \Theta(n \log n)$$

siehe MergeSort

Wo ist die Wahrheit?

Wo ist die Wahrheit?

M.a.W. was passiert im Durchschnittsfall (**average case**)?

Wo ist die Wahrheit?

M.a.W. was passiert im Durchschnittsfall (**average case**)?

Vgl. InsertionSort:

Wo ist die Wahrheit?

M.a.W. was passiert im Durchschnittsfall (**average case**)?

Vgl. InsertionSort:

$$\begin{aligned}
\text{Bester Fall} &= n-1 && \in \Theta(n) \text{ Vergleiche} \\
\text{Schlechtester Fall} &= \binom{n}{2} && \in \Theta(n^2) \text{ Vergleiche} \\
\text{Durchschnittsfall} &= \binom{n}{2}/2 && \in \Theta(n^2) \text{ Vergleiche}
\end{aligned}$$

Wo ist die Wahrheit?

M.a.W. was passiert im Durchschnittsfall (**average case**)?

Vgl. InsertionSort:

$$\text{Bester Fall} = n - 1 \quad \in \Theta(n) \text{ Vergleiche}$$

$$\text{Schlechtester Fall} = \binom{n}{2} \quad \in \Theta(n^2) \text{ Vergleiche}$$

$$\text{Durchschnittsfall} = \binom{n}{2}/2 \quad \in \Theta(n^2) \text{ Vergleiche}$$

QuickSort:

$$\text{Bester Fall} = n \log_2 n \quad \in \Theta(n \log n) \text{ Vergleiche}$$

$$\text{Schlechtester Fall} = \binom{n}{2} \quad \in \Theta(n^2) \text{ Vergleiche}$$

$$\text{Durchschnittsfall} = ?$$

Zurück zu QuickSort

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Zurück zu QuickSort

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

RandomizedPartition(A, ℓ, r)

int Partition(int[] A, int ℓ, int r)

```
pivot = A[r]
i = ℓ
for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
        A[i] ↔ A[j]
        i = i + 1
A[i] ↔ A[r]
return i
```

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

RandomizedPartition(A, ℓ, r)

$k = \text{Random}(\ell, r)$

int Partition(int[] A, int ℓ, int r)

$pivot = A[r]$
$i = \ell$
for $j = \ell$ **to** $r - 1$ **do**
  if $A[j] \leq pivot$ **then**
    $A[i] \leftrightarrow A[j]$
    $i = i + 1$
$A[i] \leftrightarrow A[r]$
return i

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

RandomizedPartition(A, ℓ, r)

$k = \text{Random}(\ell, r)$ Liefert Zufallszahl $\in \{\ell, \ldots, r\}$.

int Partition(int[] A, int ℓ, int r)

```
pivot = A[r]
i = ℓ
for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
        A[i] ↔ A[j]
        i = i + 1
A[i] ↔ A[r]
return i
```

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

RandomizedPartition(A, ℓ, r)

$k = \text{Random}(\ell, r)$ Liefert Zufallszahl $\in \{\ell, \ldots, r\}$.

$A[k] \leftrightarrow A[r]$

int Partition(int[] A, int ℓ, int r)

$pivot = A[r]$

$i = \ell$

for $j = \ell$ **to** $r - 1$ **do**

  if $A[j] \leq pivot$ **then**

    $A[i] \leftrightarrow A[j]$

    $i = i + 1$

$A[i] \leftrightarrow A[r]$

return i

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

RandomizedPartition(A, ℓ, r)

$k =$ Random(ℓ, r) Liefert Zufallszahl $\in \{\ell, \ldots, r\}$.
$A[k] \leftrightarrow A[r]$
return Partition(A, ℓ, r)

int Partition(int[] A, int ℓ, int r)

$pivot = A[r]$
$i = \ell$
for $j = \ell$ **to** $r - 1$ **do**
  if $A[j] \leq pivot$ **then**
    $A[i] \leftrightarrow A[j]$
    $i = i + 1$
$A[i] \leftrightarrow A[r]$
return i

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Demo.

`https://algo.uni-trier.de/demos/sort.html`

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal:

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

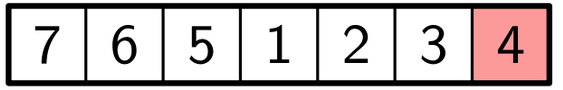
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

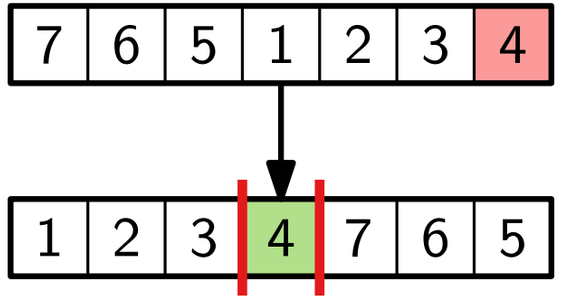
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

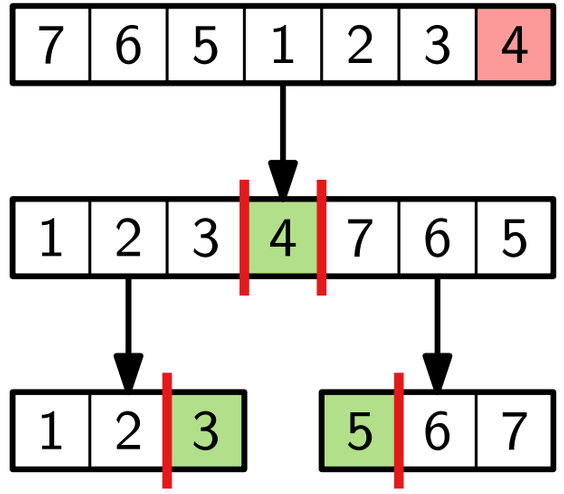
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

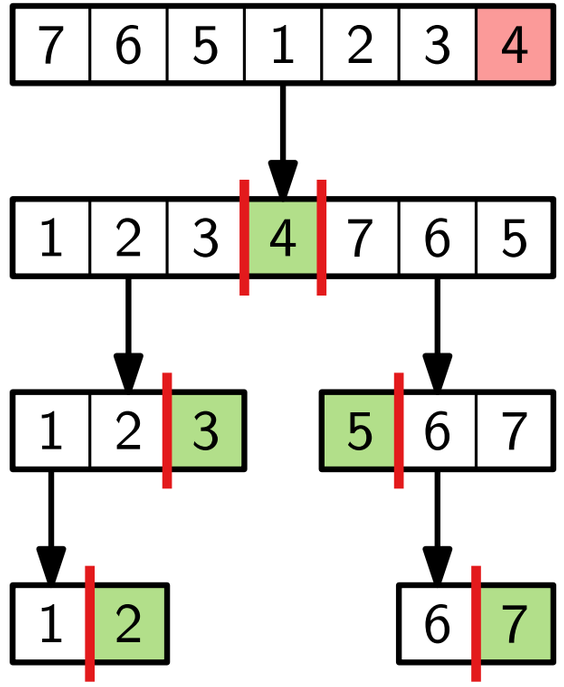
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

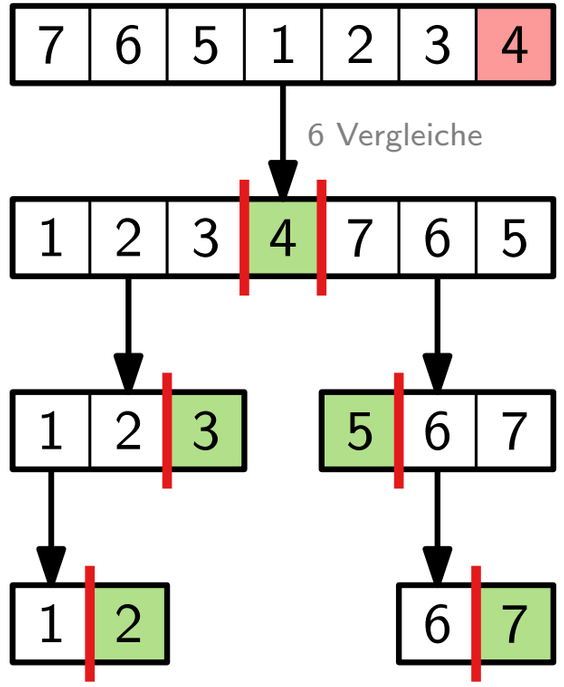
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

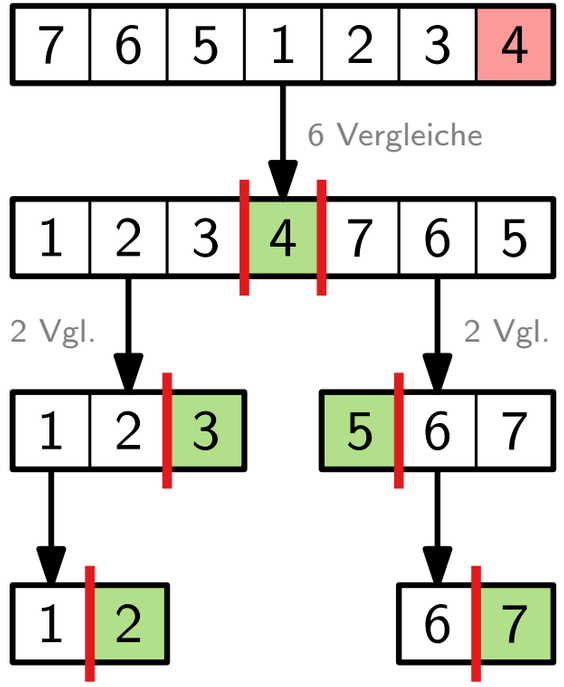
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

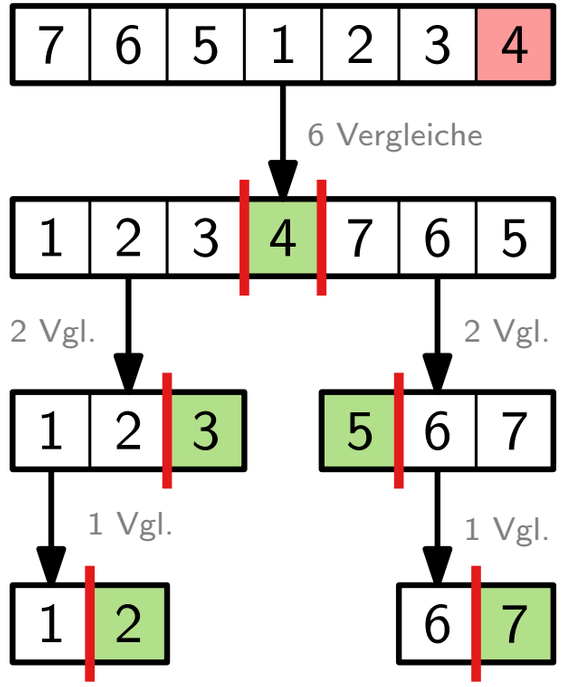
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

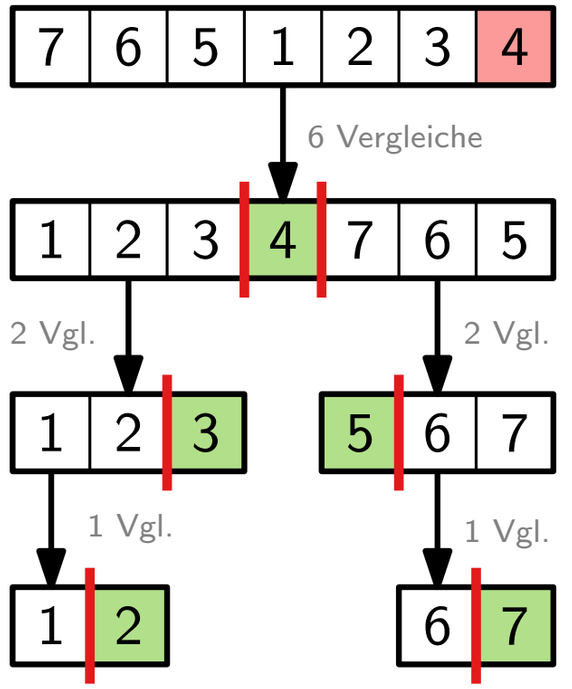
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Wie viele Vergleiche bräuchte InsertionSort für diese Instanz?

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

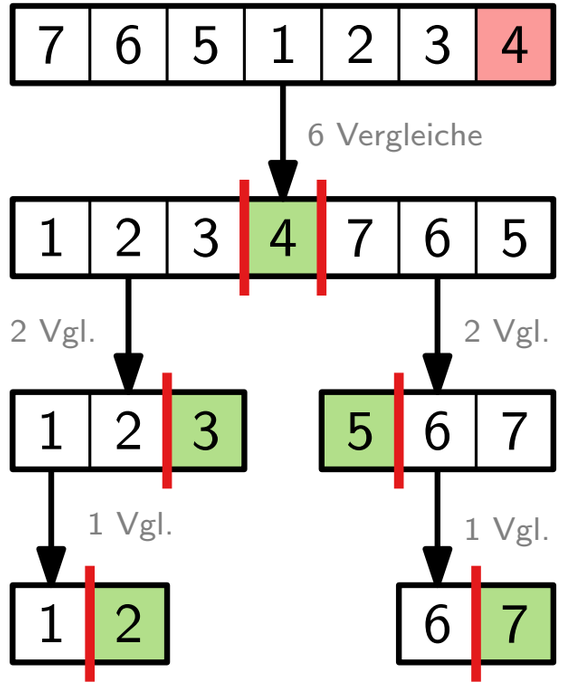
```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.



Wie viele Vergleiche bräuchte InsertionSort für diese Instanz?

$0 + 1 + 2 + 3 + 4 + 4 + 4 = 18$

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Definiere Indikator-Zufallsvariable:

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Definiere Indikator-Zufallsvariable:

$$V_{ij} = \begin{cases} 1, & \text{falls Alg. } z_i \text{ und } z_j \text{ vergleicht,} \\ 0 & \text{sonst.} \end{cases}$$

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Definiere Indikator-Zufallsvariable:

$$V_{ij} = \begin{cases} 1, & \text{falls Alg. } z_i \text{ und } z_j \text{ vergleicht,} \\ 0 & \text{sonst.} \end{cases}$$

Sei V Zufallsvar. für Gesamtanzahl der Vergleiche.

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Definiere Indikator-Zufallsvariable:

$$V_{ij} = \begin{cases} 1, & \text{falls Alg. } z_i \text{ und } z_j \text{ vergleicht,} \\ 0 & \text{sonst.} \end{cases}$$

Sei V Zufallsvar. für Gesamtanzahl der Vergleiche.

Dann gilt $V =$

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Definiere Indikator-Zufallsvariable:

$$V_{ij} = \begin{cases} 1, & \text{falls Alg. } z_i \text{ und } z_j \text{ vergleicht,} \\ 0 & \text{sonst.} \end{cases}$$

Sei V Zufallsvar. für Gesamtanzahl der Vergleiche.

Dann gilt $V = \sum_{1 \le i < j \le n} V_{ij}$.

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Definiere Indikator-Zufallsvariable:

$$V_{ij} = \begin{cases} 1, & \text{falls Alg. } z_i \text{ und } z_j \text{ vergleicht,} \\ 0 & \text{sonst.} \end{cases}$$

Sei V Zufallsvar. für Gesamtanzahl der Vergleiche.

Dann gilt $V = \sum_{1 \le i < j \le n} V_{ij}$.

$\Rightarrow E[V] =$

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Definiere Indikator-Zufallsvariable:

$$V_{ij} = \begin{cases} 1, & \text{falls Alg. } z_i \text{ und } z_j \text{ vergleicht,} \\ 0 & \text{sonst.} \end{cases}$$

Sei V Zufallsvar. für Gesamtanzahl der Vergleiche.

Dann gilt $V = \sum_{1 \le i < j \le n} V_{ij}$.

$\Rightarrow E[V] =$

Linearität des Erwartungswerts!

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

```
RandomizedPartition(A, ℓ, r)
    k = Random(ℓ, r)        // Liefert Zufallszahl ∈ {ℓ, ..., r}.
    A[k] ↔ A[r]
    return Partition(A, ℓ, r)
```

```
int Partition(int[] A, int ℓ, int r)
    pivot = A[r]
    i = ℓ
    for j = ℓ to r − 1 do
        if A[j] ≤ pivot then
            A[i] ↔ A[j]
            i = i + 1
    A[i] ↔ A[r]
    return i
```

Seien $z_1, z_2, \ldots, z_n$ die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j?

Höchstens ein Mal: wenn eins von beiden $pivot$ ist.

Definiere Indikator-Zufallsvariable:

$$V_{ij} = \begin{cases} 1, & \text{falls Alg. } z_i \text{ und } z_j \text{ vergleicht,} \\ 0 & \text{sonst.} \end{cases}$$

Sei V Zufallsvar. für Gesamtanzahl der Vergleiche.

Dann gilt $V = \sum_{1 \le i < j \le n} V_{ij}$.

$$\Rightarrow E[V] = \sum_{1 \le i < j \le n} E[V_{ij}]$$

Linearität des Erwartungswerts!

First come, first served

$E[V_{ij}] =$

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j]

First come, first served

$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] =$

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

Es gilt: Alg. vergleicht z_i und $z_j \quad \Leftrightarrow$

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

Es gilt: Alg. vergleicht z_i und $z_j \quad \Leftrightarrow \quad z^\star = z_i$ oder $z^\star = z_j$.

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

Es gilt: Alg. vergleicht z_i und $z_j \quad \Leftrightarrow \quad z^\star = z_i$ oder $z^\star = z_j$.

$\Rightarrow$ Pr[Alg. vergleicht z_i und z_j] $=$

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

Es gilt: Alg. vergleicht z_i und $z_j \quad \Leftrightarrow \quad z^\star = z_i$ oder $z^\star = z_j$.

$\Rightarrow$ Pr[Alg. vergleicht z_i und z_j] $=$ Pr[$z^\star = z_i$ oder $z^\star = z_j$]

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

Es gilt: Alg. vergleicht z_i und $z_j \quad \Leftrightarrow \quad z^\star = z_i$ oder $z^\star = z_j$.

$$\Rightarrow \text{Pr[Alg. vergleicht } z_i \text{ und } z_j] = \text{Pr}[z^\star = z_i \text{ oder } z^\star = z_j]$$

$$\overset{i \neq j}{=} \text{Pr}[z^\star = z_i] + \text{Pr}[z^\star = z_j]$$

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

Es gilt: Alg. vergleicht z_i und $z_j \quad \Leftrightarrow \quad z^\star = z_i$ oder $z^\star = z_j$.

$$\Rightarrow \text{Pr[Alg. vergleicht } z_i \text{ und } z_j] = \text{Pr}[z^\star = z_i \text{ oder } z^\star = z_j]$$

$$\overset{i \neq j}{=} \text{Pr}[z^\star = z_i] + \text{Pr}[z^\star = z_j]$$

$$= \frac{1}{|Z_{ij}|} + \frac{1}{|Z_{ij}|}$$

First come, first served

$E[V_{ij}] =$ Pr[Alg. vergleicht z_i und z_j] $=$ **?**

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

Es gilt: Alg. vergleicht z_i und $z_j \quad \Leftrightarrow \quad z^\star = z_i$ oder $z^\star = z_j$.

$$\begin{aligned}
\Rightarrow \text{ Pr[Alg. vergleicht } z_i \text{ und } z_j] &= \Pr[z^\star = z_i \text{ oder } z^\star = z_j] \\
&\overset{i \neq j}{=} \Pr[z^\star = z_i] + \Pr[z^\star = z_j] \\
&= \frac{1}{|Z_{ij}|} + \frac{1}{|Z_{ij}|} \\
&= \frac{2}{j - i + 1}
\end{aligned}$$

First come, first served

$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \ldots, z_j\}$.

Sei $z^\star$ die erste Zahl in Z_{ij}, die *pivot* wird.

Es gilt: Alg. vergleicht z_i und $z_j \quad \Leftrightarrow \quad z^\star = z_i$ oder $z^\star = z_j$.

$$\begin{aligned}
\Rightarrow \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] &= \Pr[z^\star = z_i \text{ oder } z^\star = z_j] \\
&\overset{i \neq j}{=} \Pr[z^\star = z_i] + \Pr[z^\star = z_j] \\
&= \frac{1}{|Z_{ij}|} + \frac{1}{|Z_{ij}|} \\
&= \frac{2}{j-i+1}
\end{aligned}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] =$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}]$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j - i$ durch k

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j - i$ durch k

$$= \sum_{i=1}^{n-1}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j - i$ durch k

$$= \sum_{i=1}^{n-1} \sum \frac{2}{k+1}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum \frac{2}{k+1}$$

Trick: ersetze $j-i$ durch k

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum \frac{2}{k+1}$$

$-i$

Trick: ersetze $j-i$ durch k

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

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

Trick: ersetze $j-i$ durch k

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j - i$ durch k

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j - i$ durch k

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} <$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j - i$ durch k

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j - i$ durch k

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \in \mathcal{O}(\quad)$$

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j - i$ durch k

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \in \mathcal{O}(\quad)$$

3) $\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$ **harmonische Reihe**

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j-i$ durch k

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \in \mathcal{O}(n \log n)$$

3) $\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$ **harmonische Reihe**

Auf zum letzten Gefecht...

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{\frac{2}{j-i+1}}$$

Wir wissen:

$$E[V] = \sum_{1 \le i < j \le n} E[V_{ij}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Trick: ersetze $j-i$ durch k

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \in \mathcal{O}(n \log n)$$

3) $\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$ **harmonische Reihe**

Satz. RandomizedQuickSort sortiert n Zahlen in $\mathcal{O}(n \log n)$ erwarteter Zeit.

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$		$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$		$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$		$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$		$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$		$\Theta(n \log n)$	✓	✗

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$		$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$		$\Theta(n \log n)$	✓	✗

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
QuickSort					

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
QuickSort	$\Theta(n \log n)$				

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
QuickSort	$\Theta(n \log n)$		$\Theta(n^2)$		

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$		

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	✗	✗

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	(✓)	✗

Vergleich Laufzeiten

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	(✓)	✗

QuickSort muss für jeden rekursiven Aufruf die Variable m zwischenspeichern. Dafür wird im worst case $\Omega(n)$ zusätzlicher Speicherplatz benötigt. Mit Tricks kann man dieses Problem umgehen und so QuickSort in-situ machen.