HPI Hasso Plattner Institut
Digital Engineering • Universität Potsdam

Programmiertechnik II

Ralf Herbrich

Sortieren: *Quick Sort*

Überblick

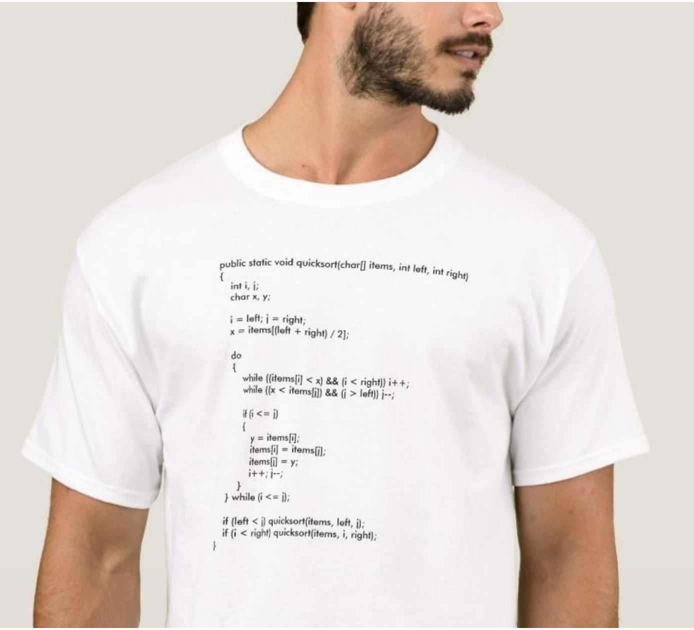
1. *Quick Sort*
2. Analyse
3. Algorithmische Verbesserungen

Merge Sort und *Quick Sort*

Algorithmus	***Best Case***	***Average Case***	***Worst Case***
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Bubble Sort	n	n^2	n^2
Shell Sort $(3x+1)$	$n \cdot \log_2(n)$	?	$n^{1.5}$
Merge Sort	${}^1/_2 \cdot n \cdot \log_2(n)$	$n \cdot \log_2(n)$	$n \cdot \log_2(n)$
Quick Sort	$n \cdot \log_2(n)$	$2n \cdot \ln(n)$	${}^1/_2 \cdot n^2$

- *Merge Sort* und *Quick Sort* sind kritische Komponenten in heutiger digitaler Infrastruktur
  - Praktisch die am meisten benutzten Sortierverfahren
  - *Quick Sort* als einer der Top-10 Algorithmen aller Zeiten ausgezeichnet

Quick Sort T-Shirt

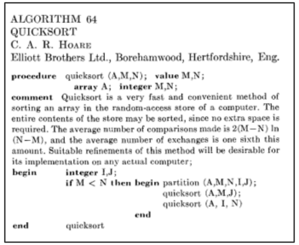


https://www.zazzle.de/quicksort_algorithmus_t_shirt-235914162256526017

Überblick

1. ***Quick Sort***
2. Analyse
3. Algorithmische Verbesserungen

- **Grundidee**: Array in zwei Teile zerlegen, so dass alle Elemente in dem linken Teil kleiner als alle Elemente in dem rechten Teil sind
  1. Ein spezielles Element `a[j]` (Pivotelement) teilt das Array in zwei Teile
  2. Für alle `a[i]` links von `j` gilt `a[i]<=a[j]`
  3. Für alle `a[i]` rechts von `j` gilt `a[i]>=a[j]`
  4. Sortierte beide Teilarrays rekursiv



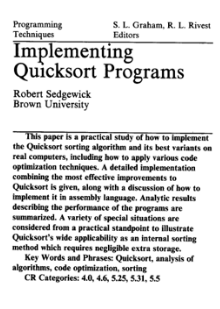
ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure quicksort (A,M,N); value M,N;
          array A; integer M,N;
comment Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin     integer I,J;
          if M < N then begin partition (A,M,N,I,J);
                              quicksort (A,M,J);
                              quicksort (A, I, N)
                        end
end       quicksort
```

Sir Tony Hoare
(1934 –)

Eingabe	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
Gemischt	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
Partitionierung	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
Links Sortieren	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
Rechts Sortieren	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
Resultat	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Pivotelement



Programming Techniques — S. L. Graham, R. L. Rivest, Editors

Implementing Quicksort Programs

Robert Sedgewick
Brown University

This paper is a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quicksort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quicksort's wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quicksort, analysis of algorithms, code optimization, sorting

CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5

Robert Sedgewick
(1946 –)

Quick Sort Partitionierung

- **Problem**: Partitioniere das Array `a` vom Index `lo` bis `hi` mit `a[lo]` als Pivotelement
- **Initialisierung**: `i=lo` und `j=hi+1`
- **Schleife**: Solange wie `i<=j` (d.h. die Zeiger sich nicht überschneiden)
  - Erhöhe `i` so lange wie `a[i] < a[lo]`
  - Verringere `j` so lange wie `a[j] > a[lo]`
  - Tausche `a[i]` und `a[j]`
- **Pivotplazierung**: Tausche `a[lo]` und `a[j]`

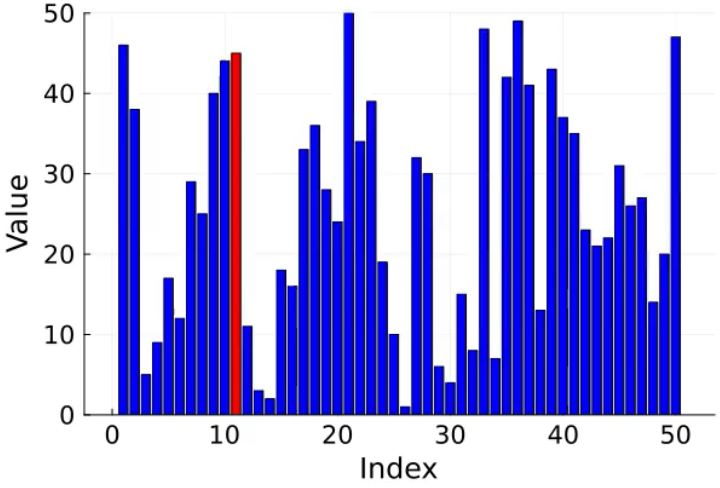
```cpp
// Implements the partition function of quick sort
template <typename Value>
int partition(Value* a, const int lo, const int hi) {
    auto i = lo, j = hi+1;
    while (true) {
        while (less(a, ++i, lo)) { if (i == hi) { break; } }
        while (less(a, lo, --j)) { if (j == lo) { break; } }
        if (i >= j) { break; }
        swap(a, i, j);
    }
    swap(a, lo, j);
    return j;
}
```

	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initialisierung	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
Linke/Rechte Suche	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
Tausch	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
Linke/Rechte Suche	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
Tausch	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
Linke/Rechte Suche	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
Tausch	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
Linke/Rechte Suche	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
Pivotplatzierung	0	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Quick Sort Partitionierung

```cpp
// Implements the partition function of quick sort
template <typename Value>
int partition(Value* a, const int lo, const int hi) {
    auto i = lo, j = hi+1;
    while (true) {
        while (less(a, ++i, lo)) { if (i == hi) { break; } }
        while (less(a, lo, --j)) { if (j == lo) { break; } }
        if (i >= j) { break; }
        swap(a, i, j);
    }
    swap(a, lo, j);
    return j;
}
```

```cpp
// Implements quick sort
template <typename Value>
void quick_sort(Value* a, const int lo, const int hi) {
    if (hi <= lo) { return; }
    auto j = partition(a, lo, hi);
    quick_sort(a, lo, j-1);
    quick_sort(a, j+1, hi);
    return;
}
```



In-place Algorithmus bei dem der Split in Teilarrays von den Daten abhängt

Quick Sort: Ausführung

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

```cpp
// Implements quick sort
template <typename Value>
void quick_sort(Value* a, const int lo, const int hi) {
    if (hi <= lo) { return; }
    auto j = partition(a, lo, hi);
    quick_sort(a, lo, j-1);
    quick_sort(a, j+1, hi);
    return;
}
```

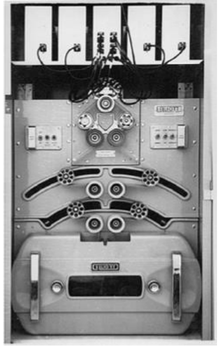
Kein Partitionieren für Teilarrays der Länge 1

Quick Sort: Empirische Analyse

Tony Hoare (1961)

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.



Elliott 405 magnetic disc

Robert Sedgewick (2020)

	Merge Sort			*Quick Sort*		
Computer	$O(n^3)$	$O(n^6)$	$O(n^9)$	$O(n^3)$	$O(n^6)$	$O(n^9)$
PC (10^8 Vergleiche/s)	instant	1 s	18 min	instant	0.6 s	12 min
Supercomputer (10^{12} Vergleiche/s)	instant	instant	instant	instant	instant	instant

Überblick

1. *Quick Sort*
2. **Analyse**
3. Algorithmische Verbesserungen

Quick Sort: *Best-Case* Analyse

- **Satz**: *Quick Sort* benötigt im besten Fall $\sim n \cdot \log_2(n)$ Vergleiche.

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

```
// Implements quick sort
template <typename Value>
void quick_sort(Value* a, const int lo, const int hi) {
    if (hi <= lo) { return; }
    auto j = partition(a, lo, hi);
    quick_sort(a, lo, j-1);
    quick_sort(a, j+1, hi);
    return;
}
```

Quick Sort: *Worst-Case* Analyse

- **Satz**: *Quick Sort* benötigt im schlechtesten Fall $\sim \frac{1}{2}n^2$ Vergleiche.

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14	14	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

```cpp
// Implements quick sort
template <typename Value>
void quick_sort(Value* a, const int lo, const int hi) {
    if (hi <= lo) { return; }
    auto j = partition(a, lo, hi);
    quick_sort(a, lo, j-1);
    quick_sort(a, j+1, hi);
    return;
}
```

Quick Sort: *Average Case* Analyse

- **Satz**: *Quick Sort* benötigt für ein zufällig sortiertes Array der Länge n mit eindeutigen Schlüsseln im Durchschnitt $\sim 2n \cdot \ln(n)$ Vergleiche.
  - **Beweisskizze**: Die erwartete Anzahl der Vergleiche $C(n)$ erfüllt $C(0) = C(1) = 0$ und für $n \geq 2$

$$C(n) = \underbrace{(n+1)}_{\text{Partitionierung}} + \underbrace{\frac{1}{n}}_{\text{Gleichverteilung über Position des Pivotelements}} \cdot \left[\left(C(0) + C(n-1)\right) + \left(C(1) + C(n-2) + \cdots + \left(\underbrace{C(n-1)}_{\text{Linkes Teilarray}} + \underbrace{C(0)}_{\text{Rechtes Teilarray}}\right)\right)\right]$$

Beide Seiten mit n multiplizieren gibt

$$n \cdot C(n) = n \cdot (n+1) + 2 \cdot \left(C(0) + C(1) + \cdots + C(n-1)\right)$$

Damit folgt für $n \cdot C(n) - (n-1) \cdot C(n-1)$ direkt

$$n \cdot C(n) - (n-1) \cdot C(n-1) = 2n + 2 \cdot C(n-1)$$

$$n \cdot C(n) = 2n + (n+1) \cdot C(n-1)$$

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

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

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

$$= \underbrace{\boxed{\frac{C(1)}{2}}}_{=0} + \frac{2}{3} + \cdots + \frac{2}{n} + \frac{2}{n+1}$$

$$C(n) = 2 \cdot (n+1) \cdot \left(\frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n+1}\right) \sim 2 \cdot (n+1) \cdot \int_3^{n+1} \frac{1}{x}\,dx$$

$\sim 2n \cdot \ln(n)$

Stabilität: *Quick Sort*

- **Satz**: *Quick Sort* ist nicht stabil.
- **Beweis**: Durch Vertauschen über weite „Distanzen“ kann die Reihenfolge innerhalb einer Gruppe sich ändern.

i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

```
// Implements the partition function of quick sort
template <typename Value>
int partition(Value* a, const int lo, const int hi) {
    auto i = lo, j = hi+1;
    while (true) {
        while (less(a, ++i, lo)) { if (i == hi) { break; } }
        while (less(a, lo, --j)) { if (j == lo) { break; } }
        if (i >= j) { break; }
        swap(a, i, j);
    }
    swap(a, lo, j);
    return j;
}
```

Überblick

1. *Quick Sort*
2. Analyse
3. **Algorithmische Verbesserungen**

Quick Sort in der Praxis

1. ***Insertion Sort*** **für kleine Arrays benutzen**
   - *Quick Sort* hat zu viel Kopierkosten für kleine Arrays
   - Typischer *cutoff* bei Array der Länge 10

2. **Median als Pivotelement benutzen**
   - Optimal wäre der Median weil das zur exakten Teilung führt
   - Schätzung des Medians durch drei Beispiele aus dem Array
   - Führt für zufällig sortierte Arrays zu 14% weniger Vergleichen

```cpp
// Implements quick sort with optimizations
template <typename Value>
void quick_sort(Value* a, const int lo, const int hi) {
    if (hi <= lo + CUTOFF - 1) insertion_sort(a, lo, hi);

    int median = median_of_three(a, lo, lo + (hi - lo) / 2, hi);
    swap(a, lo, median);

    auto j = partition(a, lo, hi);
    quick_sort(a, lo, j-1);
    quick_sort(a, j+1, hi);
    return;
}
```

Quick Sort: Gleiche Schlüssel

- Gleiche Schlüssel können *Quick Sort* verlangsamen.

- **Satz**: Die Partitionierung von *Quick Sort* zerlegt ein Array mit gleichen Schlüsseln in zwei gleich große Teilarrays.

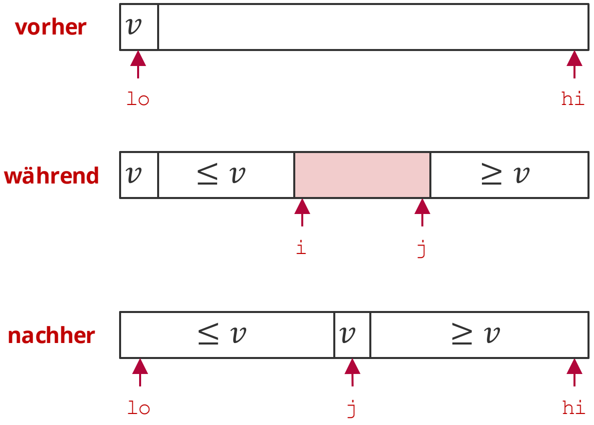
```
// Implements the partition function of quick sort
template <typename Value>
int partition(Value* a, const int lo, const int hi) {
    auto i = lo, j = hi+1;
    while (true) {
        while (less(a, ++i, lo)) { if (i == hi) { break; } }
        while (less(a, lo, --j)) { if (j == lo) { break; } }
        if (i >= j) { break; }
        swap(a, i, j);
    }
    swap(a, lo, j);
    return j;
}
```

	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initialisierung	0	16	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Linke/Rechte Suche	1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Tausch	1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Linke/Rechte Suche	2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Tausch	2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Linke/Rechte Suche	3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Tausch	3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Linke/Rechte Suche	4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Tausch	4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Linke/Rechte Suche	5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Tausch	5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Linke/Rechte Suche	6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Tausch	6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Linke/Rechte Suche	7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Tausch	7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Linke/Rechte Suche	8	8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Tausch	8	8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

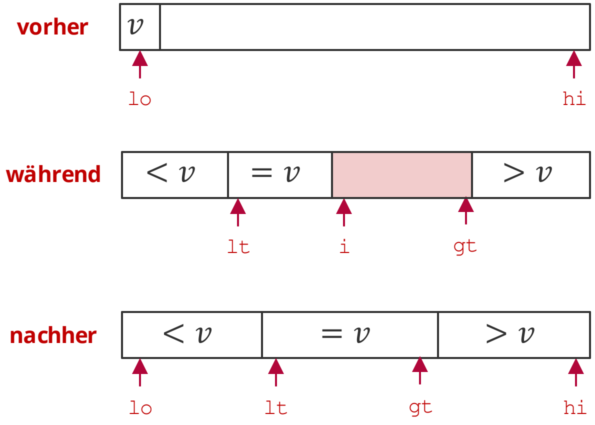
3-way Quick Sort: Gleiche Schlüssel

- Geht es auch schneller, Arrays mit gleichen Schlüsseln zu sortieren?
  - Ja, indem die drei Fälle `a[i] < a[j]`, `a[i] = a[j]` und `a[i]>a[j]` separat behandelt werden!
- Grundidee der Partitionierung:

2-way Quick Sort



3-way Quick Sort

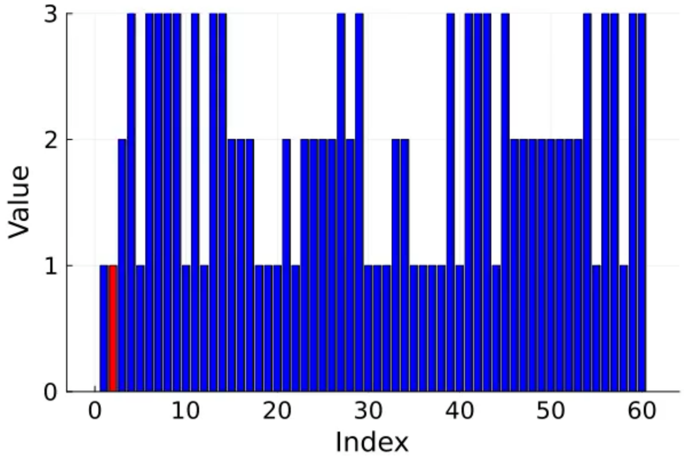


Edsger Dijkstra (1930 – 2002)

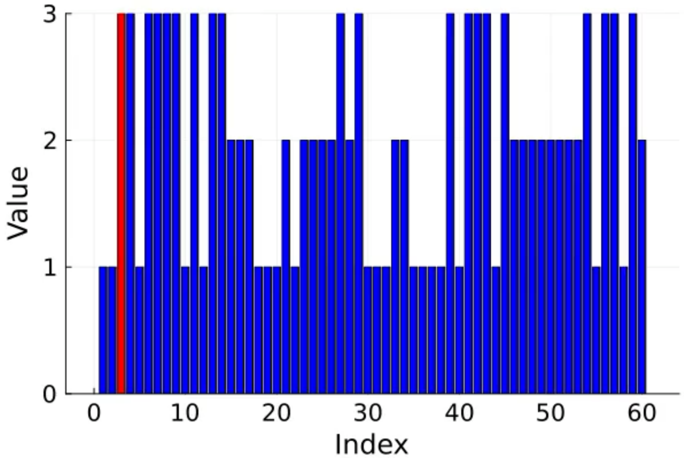
```
// Implements 3-way quick sort
template <typename Value>
void quick_sort_3way(Value* a, const int lo, const int hi) {
    if (hi <= lo) { return; }
    auto lt = lo, i = lo+1, gt = hi;
    while (i <= gt) {
        if (less(a, i, lt))      { swap(a, lt++, i++);}
        else if (less(a, lt, i)) { swap(a, i, gt--); }
        else                     { i++; }
    }

    quick_sort_3way(a, lo, lt-1);
    quick_sort_3way(a, gt+1, hi);
    return;
}
```

Quick Sort mit gleichen Schlüsseln: Beispiel

2-way Quick Sort



3-way Quick Sort



Viel Spaß bis zur nächsten Vorlesung!