Programmiertechnik++

Ralf Herbrich, Christoph Lippert

Sortieren – Quicksort

slides credit: Felix Naumann

HPI Hasso Plattner Institut
Digital Engineering • Universität Potsdam

Überblick

- Elementare Sortierverfahren (letzte Woche)
  - Selectionsort
  - Bubblesort
  - Insertionsort
  - Shellsort
- Mergesort (Montag)
- **Quicksort** (heute)
- Countingsort

Quicksort
Historie

- Erfunden von C.A.R. (Tony) Hoare
  - Sortieren von russischen Wörtern für ein maschinelle Übersetzung
  - 1980: Turing Award für Programmiersprachen (Occam)
- Analysiert und um Varianten erweitert von Robert Sedgewick

R. Sedgewick

C.A.R. Hoare

ALGORITHM 64 Commun. ACM 4(7): 321 (1961)

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
```

There are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies, and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult.

Vergleich
Mergesort und Quicksort

```
template <typename T>
void mergesort(T* a, T* aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (hi + lo) / 2;
    mergesort(a, aux, lo, mid);
    mergesort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}
```

```
template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

mergesort

- **sortiere** zwei unsortierten Sub-arrays der Länge n/2 (rekursiv)
- **merge** der zwei sortierten sub-arrays in einen sortierten array (O(n) Operationen)

quicksort

- **partitioniere** a in zwei subarrays, so dass alle Elemente des linken subarrays kleiner sind als alle Elemente des rechten sub-arrays (O(n) Operationen)

$$a[lo..p-1] \leq a[p+1..hi]$$

- **sortiere** die beiden sub-ararys (rekursiv)

Quicksort
Prinzip

Idee:

- rekursive Aufteilung (Divide-and-Conquer)
- Vermeidung des Mergevorgangs durch Partition des Arrays in zwei Teile bezüglich eines **Pivot-Elementes** *p*, wobei
  - pivot-Element wird an die richtige Stelle getauscht
  - links von *p* sind alle Elemente kleiner-gleich *p*
  - rechts von *p* sind größer-gleich *p*
- Sortierung der linken und rechten Teilarrays

```
template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

Pivot-Element

in-place

in-place

QuickSort: Zerlegen (partition) beim QuickSort

Eingabe: Array a, linke/rechte Grenze lo, hi

Ausgabe:

- Neue Pivotposition *p* Mitte der Zerlegung
- a modifiziert, so dass:
  $\forall\, i \leq p$: $a[i] \leq a[p]$
  $\forall\, j > p$: $a[j] > a[p]$

```cpp
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}
```

QuickSort: Vertauschen von Elementen

- Auswahl des Pivot-Elements *p*:
  - Kann beliebig gewählt werden, z.B. das erste Element der zu sortierenden Teilfolge
  - Aber: “Ungünstige” Wahl des Pivotelements führt zu längerer Laufzeit
- Pivot-Element *p*
  - Folge von links durchsuchen, bis Element gefunden, das größer oder gleich *p* ist
  - Folge von rechts durchsuchen, bis Element gefunden, das kleiner *p* ist
- Elemente ggf. tauschen

2	4	3	**6**	7	9	8

$\leq a[p]$ $\quad a[p] \quad$ $> a[p]$

```cpp
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}
```

Beispielablauf für Partition

pivot-element $a[p]$

$a[i] > a[p]$

$a[j] \leq a[p]$

$\text{swap}(a[i], a[j])$

```cpp
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}
```

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14
6	2	4	5	3	9	12	1	15	7	10	13	8	11	16	14
6	2	4	5	3	1	12	9	15	7	10	13	8	11	16	14
1	2	4	5	3	6	12	9	15	7	10	13	8	11	16	14

Quicksort Beispiel

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
1	2	4	5	3	6	12							11	16	14
1	2	4	5	3	6	12							11	16	14
1	2	4	5	3	6	12	9	15	7	10	13	8	11	16	14
1	2	3	4	5	6	12	9	15	7	10	13	8	11	16	14
1	2	3	4	5	6	8	9	11	7	10	12	13	15	16	14
1	2	3	4	5	6	7	8	11	9	10	12	13	15	16	14
1	2	3	4	5	6	7	8	10	9	11	12	13	15	16	14
1	2	3	4	5	6	7	8	9	10	11	12	13	15	16	14
1	2	3	4	5	6	7	8	9	10	11	12	13	15	16	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Erst linker Zweig
dann rechter
Zweig der Rekursion

Quicksort
Korrektheit

Hasso Plattner Institut
Digital Engineering • Universität Potsdam

```cpp
template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

Induktionsannahme:

- Arrays a der Länge < *n* können sortiert werden.

Induktionsanfang: $n = 1$:

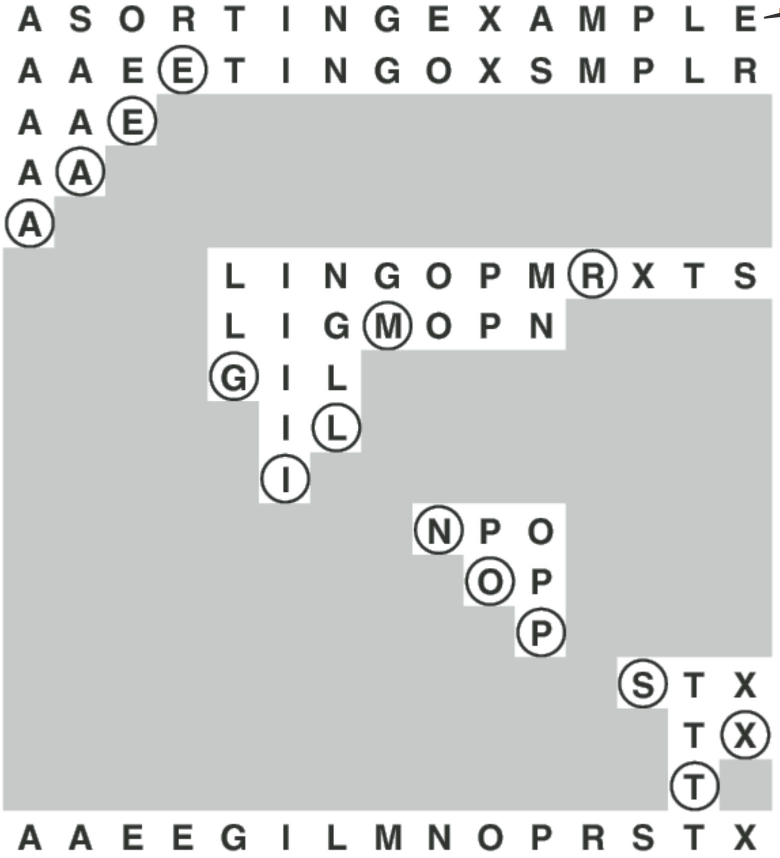
- sind immer sortiert

Induktionsschritt: $< n \rightsquigarrow n$:

- Wähle ein beliebiges Pivot-Element
- Verschiebe alle Elemente, die kleiner als das Pivot-Element sind, in ein erstes Teilarray L
- Verschiebe alle Elemente, die größer als das Pivot-Element sind, in ein zweites Teilarray R
- Sortiere beide Teilfelder mit < *n* Elementen nach Induktionsannahme
- Füge die sortierten Teilarrays zusammen: L + Pivot-Element + R

Quicksort
Anschauung

Wahl des Pivot-Elements hier jeweils: *hi*



Quicksort ist ein rekursiver Zerlegungsprozess: Wir teilen eine Datei, indem wir irgendein Element (das Trennelement) an seine Position bringen und dann das Array neu anordnen, sodass kleinere Elemente links vom Trennelement und größere Elemente rechts davon stehen. Dann sortieren wir den linken und den rechten Teil rekursiv. Jede Zeile in diesem Diagramm stellt das Ergebnis der Zerlegung der angezeigten Teildatei nach dem eingekreisten Element dar. Das Endergebnis ist eine vollständig sortierte Datei.

Stabilität

- Ist Quicksort stabil?
  - D.h. relative Reihenfolge gleicher Schlüssel bleibt erhalten
- Im Gegensatz zur MergeSort ist QuickSort durch Vorgehensweise bei Vertauschungen instabil
  - Stabilität ist herstellbar

Ja	Nein	Weiß nicht	Kommt drauf an
	X		

```
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}

template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

Analyse Quicksort 1961

Table 1

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.

sorting N 6-word items with 1-word keys

Elliott 405 magnetic disc (16K words)

Analyse Quicksort

- Laufzeitschätzung
  - Laptop: 10^8 Vergleiche / Sekunde
  - Supercomputer: 10^{12} Vergleiche / Sekunde

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

- Schlussfolgerung 1: Gute Algorithmen sind (immer noch) besser als gute Computer!
- Schlussfolgerung 2: Tolle Algorithmen sind besser als gute Algorithmen!

Inputvarianten

- Sortierter Input
- Umgekehrt sortierter Input
- Unsortierter Input
- Viele gleiche Werte
- http://www.sorting-algorithms.com/quick-sort

Problemfall:
Hintereinander gleiche Pivotelemente

Besonders günstig	Besonders ungünstig	Egal	Günstig
	Umgekehrt sortierter Input Sortierter Input	Viele gleiche Werte	Unsortierter Input

Falls erstes Element als Pivotelement gewählt wird

```cpp
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}

template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

Aufwand

- Best case
- Average case
- Worst case

$O(n)$	$O(n \log n)$	$O(n^2)$	$> O(n^2)$
	Best case Average case	Worst case	

```
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}

template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

Quicksort – Worst Case

- Werte: A … O

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

```cpp
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}

template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

Quicksort – Worst Case

```cpp
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}

template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			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	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

Analyse QuickSort worst case

- **Schlechtester Fall:** n^2 (Pivot-element immer an 1./n. Stelle)

$C_n = n + 1 + (C_1 + C_{n-1})$ für $n \geq 2$, mit $C_0 = C_1 = 0$.

Vergleiche restlichen n-1 Elemente mit Pivot-Element

\# Vergleiche für Teilarray der Länge n-1

$$C_n = (n+1) + (n) + (n-1) + \cdots + 4 + 3 = \frac{(n+1)(n+2)}{2} - 2 - 1 \approx \frac{n^2}{2}$$

- Tritt bei Wahl des Pivot-Elements ganz links dann auf, wenn die Folge schon sortiert ist (ungünstig aber unwahrscheinlich!)
- Die Wahrscheinlichkeit, dass dieser Fall auftritt, kann durch **zufällige Wahl** eines Elements der Teilfolge und Vertauschen mit dem ganz linken Element vor Partition gesenkt werden.

  Idee: mische den Array zu Beginn, um zu vermeiden, dass wiederholte Aufrufe von quicksort systematisch den worst case verursachen.

```
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}

template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

```
template<typename T>
void quicksort(T a[], int n) {
    shuffle(a, n);
    quicksort(a, 0, n-1);
}
```

Quicksort – Best Case

- Werte: A ... O

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	A	C	B	F	E	G	D	L	I	K	J	N	M	O

```cpp
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}

template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

Quicksort – Best Case

```cpp
template<typename T>
int partition(T a[], int lo, int hi) {
    T pivot = a[lo];  // first element as pivot
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;

        swap(a[i], a[j]);
    }

    swap(a[lo], a[j]);
    return j;
}

template<typename T>
void quicksort(T a[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quicksort(a, lo, p - 1);
        quicksort(a, p + 1, hi);
    }
}
```

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			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

Analyse Best Case

- Jede Zerlegung halbiert den zu durchsuchenden array genau.
- Anzahl der Vergleiche bei einem Array der Größe *n*

$$C_n = \underbrace{2C_{\frac{n-1}{2}}}_{\text{\# Vergleiche für 2 Arrays mit je } \frac{n-1}{2} \text{ Elementen + swap}} + \underbrace{n + 1}_{\text{Vergleiche restlichen n-1 Elemente mit Pivot-Element}}$$

- Ähnlich zu der Analyse von Mergesort können wir diese rekurrente Formel auflösen

$$C_n \approx n \log_2(n)$$

Analyse Average Case

$C_n = n + 1 + \frac{1}{n}\sum_{1 \le k \le n}(C_{k-1} + C_{n-k})$ für $n \ge 2$, mit $C_0 = C_1 = 0$.

Vergleiche restlichen n-1 Elemente mit Pivot-Element

Pivot-element an Position *k* gleich wahrscheinlich für jedes *k*

Vergleiche für 2 Teilarrays für *k*

Der Term in der Summe lässt sich auflösen

$$C_n = n + 1 + \frac{2}{n} \sum_{1 \le k \le n} C_{k-1}$$

$$nC_n = n(n+1) + 2 \sum_{1 \le k \le n} C_{k-1}$$

$$nC_n - (n-1)C_{n-1} = n(n+1) + 2 \sum_{1 \le k \le n} C_{k-1} - \left(n(n-1) + 2 \sum_{1 \le k \le n-1} C_{k-1} \right)$$

$$nC_n - (n-1)C_{n-1} = 2n + 2C_{n-1}$$

$$nC_n = 2n + (n+1)C_{n-1}$$

Analyse Average Case

$C_n = n + 1 + \frac{1}{n}\sum_{1\le k\le n}(C_{k-1} + C_{n-k})$ für $n \ge 2$, mit $C_0 = C_1 = 0$.

Vergleiche restlichen n-1 Elemente mit Pivot-Element

Pivot-element an Position *k* gleich wahrscheinlich für jedes *k*

Vergleiche für 2 Teilarrays für *k*

$$nC_n = 2n + (n+1)C_{n-1}$$
$$\frac{C_n}{n+1} = \frac{2}{n+1} + \frac{C_{n-1}}{n}$$
$$= \frac{2}{n+1} + \frac{2}{n} + \frac{C_{n-2}}{n-1}$$
$$= \sum_{1\le k\le n} \frac{2}{k+1}$$
$$\approx 2 \ln n$$
$$\Rightarrow C_n \approx 2n \ln n$$

Zum Vergleich: best case $C_n \approx n \log_2 n$ nur ca. 38% schneller (konstanter Faktor)

Quicksort Verbesserungen median-of-3

- Wie zuvor: Auf Insertionsort ausweichen bei < 10 Elementen
- Beste Wahl des Pivotelements: Median
  - Suche aufwändig (Sortieren!)
  - Deshalb: Median aus einem (Random) Sample (z.B. der Größe 3)
- Bei vielen gleichen Schlüsselwerten: 3-Wege-Partitionierung
  - Mittlere Partition mit nur gleichen Werten

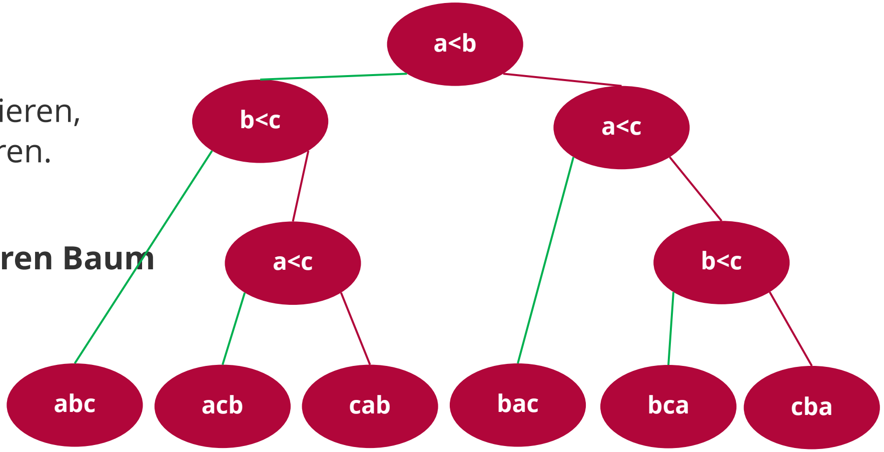
```cpp
template<typename T>
T medianOfThree(T a[], int lo, int hi) {
    int mid = lo + (hi - lo) / 2;

    if ((a[lo] - a[mid]) * (a[hi] - a[lo]) >= 0)
        return lo;
    else if ((a[mid] - a[lo]) * (a[hi] - a[mid]) >= 0)
        return mid;
    else
        return hi;
}

template<typename T>
int partition(T a[], int lo, int hi) {
    int medianIndex = medianOfThree(a, lo, hi);
    swap(a[lo], a[medianIndex]);  // Place median at first position

    T pivot = a[lo];
    int i = lo + 1;
    int j = hi;
    while (true) {
        while (i <= j && a[i] <= pivot) i++;
        while (i <= j && a[j] > pivot) j--;
        if (i >= j) break;
        swap(a[i], a[j]);
    }
    swap(a[lo], a[j]);
    return j;
}
```

Sortierverfahren im Vergleich

Verfahren	Stabilität	In-place	Vergleiche im Mittel
InsertionSort	stabil	Ja	$n^2/4$
ShellSort	instabil	Ja	Je nach Sequenz unterschiedliche Beweise, z.B. $n^{3/2}$
SelectionSort	stabil	Ja	$n^2/2$
BubbleSort	stabil	Ja	$n^2/2$
MergeSort	stabil	Nein	$n \log_2 n$
QuickSort	instabil	Ja	ca. $2n \ln n$ ($\approx 1{,}38\, n \log_2 n$)

Sortierung in O(n log n)

Beispiel: 3 Werte: a,b,c sollen sortiert werden



- Hier: Komplexität = Anzahl Vergleiche
- oBdA: Nur unterschiedliche Werte

Satz: Kein Vergleichs-basierter Sortieralgorithmus kann garantieren, N Elemente mit weniger als $\lg(N!) \sim N\ lgN$ Vergleichen zu sortieren.

Beweisidee

- Modelliere Folge der Vergleiche eines Algorithmus als **binären Baum**
- Jedes Blatt entspricht einer Permutation
  - $\Rightarrow$ **mindestens** $N!$ Blätter

Stirlingformel ab hinreichend großem *N*

- Höhe des Baums entspricht Anzahl der Vergleiche
  - Gesucht ist also der **längste Pfad** (=worst case)
- **Bestmöglicher Baum:** Balanciert mit Höhe *h*
- **Maximale Blatt-Anzahl** eines Baums der Höhe *h:* 2^h
- Zusammen: $N! \leq$ Anzahl Blätter $\leq 2^h$
- Beide Seiten logarithmieren: $\lg(N!) \sim N\ lgN \leq h$

Überblick

- Elementare Sortierverfahren
  - Selectionsort
  - Bubblesort
  - Insertionsort
  - Shellsort
- Mergesort
- Quicksort
- **Countingsort**

Countingsort

- Sortierung in linearer Zeit ist möglich, wenn man Annahmen über die Verteiliung der Werte macht und somit auf vergleiche verzichtet.
- **Beispiel:** Nur int-Werte im Bereich min..max in einem Array der Länge N

Idee:

- Zähle, wie oft jede Zahl vorkommt

- **Laufzeit** $O(\text{range} + n)$
  - effizient wenn range klein ist.
- Countingsort ist stabil

```cpp
void countingSort(int arr[], int n) {
    int max_val = *std::max_element(arr, arr + n);
    int min_val = *std::min_element(arr, arr + n);

    int range = max_val - min_val + 1;

    std::vector<int> count(range), output(n);

    for(int i = 0; i < n; i++)
        count[arr[i] - min_val]++;

    for(int i = 1; i < count.size(); i++)
        count[i] += count[i - 1];

    for(int i = n - 1; i >= 0; i--) {
        output[count[arr[i] - min_val] - 1] = arr[i];
        count[arr[i] - min_val]--;
    }

    for(int i = 0; i < n; i++)
        arr[i] = output[i];
}
```

Überblick

- Elementare Sortierverfahren
  - Selectionsort
  - Bubblesort
  - Insertionsort
  - Shellsort
- Mergesort
- **Quicksort**
- **Countingsort**

Viel Spaß bis zur nächsten Vorlesung!