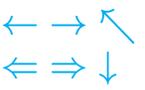


Algorithmen und Datenstrukturen (Th. Ottmann und P. Widmayer)

Folien: Einfache Sortierverfahren

Autor: Stefan Edelkamp

Institut für Informatik
Georges-Köhler-Allee
Albert-Ludwigs-Universität Freiburg



Überblick

Das Sortierproblem

Grundlegende Implementation

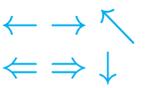
Auswahlort

Sortieren durch Einfügen

Shellsort

Mergesort

Quicksort



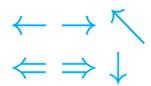
- **Gegeben:** Folge $s[1], \dots, s[n]$ von n Sätzen; jeder Satz $s[i]$ hat einen Schlüssel k_i .
- **Gesucht:** Permutation π , die die Anordnung der Sätze gemäß ihrer Schlüssel in aufsteigende Reihenfolge bringt:

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \dots \leq k_{\pi(n)}.$$

Ausgabe: Sätze in aufsteigender Reihenfolge

- in einem Array (**internes Sortieren**)
- in einer Datei auf Festplatte oder Magnetband (**externes Sortieren**)

In situ-Verfahren: Verfahren, die wenig zusätzlichen Speicherplatz benötigen.



Schlüsselvergleiche ($<$, $=$, $>$) sind die einzige verwendbare Information zur Lösung des Sortierproblems.

1. Beispiele:

- (a) Bubblesort, Auswahlort Sortieren durch Einfügen, Shellsort
- (b) Mergesort
- (c) Quicksort
- (d) Clever-Quicksort
- (e) Heapsort
- (f) Weak-Heapsort

2. Untere Schranken

3. Spezielle Sortierverfahren:

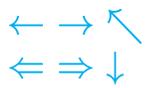
Schlüssel sind ganze Zahlen

Verwendung von $*$, $/$, shift, mod, etc.

- (a) Radix (exchange-) Sort
- (b) Sortieren durch Fachverteilung

4. Sortieren vorsortierter Daten:

Verfahren für besondere Eingaben



Laufzeitmessung:

a. # Schlüsselvergleiche (comparisons)

$C_{min}(n)$ (best case)

$C_{max}(n)$ (worst case)

$C_{av}(n)$ (average case)*

b. # Datenbewegungen (movements) $M_{min}, M_{max}, M_{av}^*$

* Im average case: Mittelung über alle $n!$ Anordnungen von n verschiedenen Sätzen

Messung der Anzahl der Vergleiche

1. Für die meisten Sortierverfahren: $M_x \in O(C_x)$,
 $x = max, min, av$

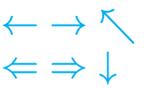
2. Datenbewegung: Pointerzuweisung,
Schlüsselvergleich: Vergleich der einzelnen Zeichen

Beispiel: Bubblesort

$$C_{min}(n) = n - 1, \quad M_{min} = 0$$

$$C_{max}(n) = \frac{1}{2}n(n - 1) = M_{max}$$

$$C_{av}(n) \in O(n^2), \quad M_{av}(n) \in O(n^2)$$



Rahmenprogramm: SortAlgTest.java: Programm zum Testen von Sortieralgorithmen

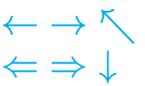
```
public class SortAlgTest {
    public static void main(String args[]){
        int vec[] = {15, 2, 43, 17, 4, 8, 47};
        if (args.length != 0) {
            vec = new int [args.length];
            for (int j=0; j<args.length;j++) {
                vec[j] = Integer.valueOf(args[j]).intValue();
            }
        }
        /* t[0] wird als Stopper verwendet */
        OrderableInt t[] = OrderableInt.array (vec);
        SortAlgorithm.printArray(t);
        SortAlgorithm.sort(t);
        SortAlgorithm.printArray (t);
    }
}
```

Interface: Orderable.java: Beschreibt das Verhalten vergleichbarer Objekte

```
interface Orderable {
    public boolean equal      (Orderable o);
    public boolean greater   (Orderable o);
    public boolean greaterEqual (Orderable o);
    public boolean less      (Orderable o);
    public boolean lessEqual  (Orderable o);
    public Orderable minKey  ();
}
```

Sortierbasisklasse: SortAlgorithm.java bietet abzuleitende Klasse für die Sortieralgorithmen

```
class SortAlgorithm {
    static void swap(Object A[], int i, int j){
        Object o = A[i]; A[i] = A[j]; A[j] = o;
    }
    static void sort (Orderable A[]) {}
    static void printArray (Orderable A[]) {
        for (int i = 1; i < A.length; i++)
            System.out.print(A[i].toString()+" ");
        System.out.println();
    }
}
```



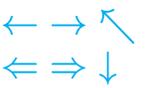
Idee: Bestimme der Reihe nach das i -kleinste Element im Rest der Liste ($1 \leq i \leq n$) und tausche es an die Position i

Implementation

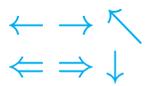
```
class AuswahlSort extends SortAlgorithm {
    static void sort (Orderable A[]) {
        for (int i = 1; i < A.length-1; i++) {
            int min = i;
            for (int j = i+1; j <= A.length-1; j++) {
                if (A[j].less(A[min])) {
                    min = j;
                }
            }
            swap(A, i, min);
        }
    }
}
```

Analyse: Übungsaufgabe

5 Sortieren durch Einfügen



```
class EinfuegeSort extends SortAlgorithm {
    static void sort (Orderable A[]) {
        for (int i = 2; i < A.length; i++) {
            Orderable temp = A[i];
            int j = i - 1;
            while (j >= 1 && A[j].greater(temp)) {
                A[j+1] = A[j];
                j--;
            }
            A[j+1] = temp;
        }
    }
}
```

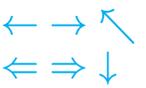


Einfügen des i -ten Elementes benötigt
mindestens 1 , höchstens $i - 1$ Vergleiche und
mindestens 2 , höchstens $i + 1$ Bewegungen
($i = 2, \dots, n$)

$$C_{min}(n) = n - 1 \quad C_{max} = \sum_{i=2}^n (i - 1) = \frac{n(n - 1)}{2}$$
$$M_{min}(n) = 2(n - 1)$$
$$M_{max}(n) = \sum_{i=2}^n (i + 1) = \frac{n(n - 1)}{2} + 2(n - 1)$$

Im Mittel: Die Hälfte von s_0, \dots, s_{i-1} ist größer als s_i

$$C_{av}(n) \approx \sum_{i=1}^{n-1} \frac{i + 1}{2} = \Theta(n^2)$$
$$M_{av}(n) \approx \sum_{i=1}^{n-1} \frac{i - 1}{2} + 2 = \Theta(n^2)$$



Folge $1 = h_1 < h_2 < h_3 < \dots < h_k$

1. Zu $h_i > 0$, $j = 0, \dots, h_i - 1$, betrachte Teilfolgen

$$s_j, s_{j+h_i}, s_{j+2h_i}, \dots$$

2. Sortiere jeder dieser Teilfolgen durch Einfügen

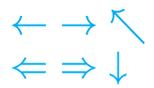
3. Falls $h_i > 1$, erniedrige h_i auf einen Wert h_{i-1} und gehe zu 1, sonst fertig;

Satz: Worst case Laufzeit: $O(n \log^2 n)$, falls

$$h_i \in \{2^p 3^q < N\}$$

Satz: (Li and Vitányi 1999): Average case für alle Sequenzen (k pass) in $\Omega(kn^{1+1/k})$.

Beispiel zu Shellsort



$$h_3 = 5, h_2 = 3, h_1 = 1$$

Eingabefolge: 15 2 43 17 4 8 47

$h_3 = 5$ 15 2 43 17 4 8 47

8 2 43 17 4 15 47

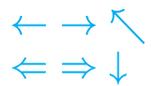
$h_2 = 3$ 8 2 43 17 4 15 47

8 2 15 17 4 43 47

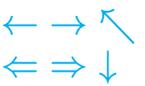
$h_1 = 1$ 8 2 15 17 4 43 47

2 4 8 15 17 43 47

Implementation: Shellsort



```
class ShellSort extends SortAlgorithm {  
  
    /* Sortierverfahren nach D.L. Shell */  
  
    static void sort (Orderable A[]) {  
        int n = A.length - 1;  
        int h = n / 2;  
  
        while (h > 1) {  
            for (int j = 1; j <= h; j++) {  
                sort (A, h, j);  
            }  
            h = h / 2;  
        }  
        sort(A, 1, 1);  
    }  
  
    static void sort (Orderable A[],int h,int k) {  
        for (int i = k + h; i < A.length; i = i + h) {  
            Orderable temp = A[i];  
            int j = i - h;  
            while (j >= 1 && temp < A[j]) {  
                A[j+h] = A[j];  
                j = j - h;  
            }  
            A[j+h] = temp;  
        }  
    }  
}
```



Mergesort—Sortieren durch Verschmelzen

Prinzip von Mergesort

Sortieren durch rekursives Verschmelzen von sortierten Teilfolgen

Eingabe: unsortierte Folge L

Ausgabe: sortierte Folge

```
if |L| = 1
```

```
    return L
```

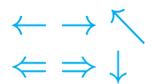
```
L1 = erste Haelfte von L
```

```
L2 = zweite Haelfte von L
```

```
Mergesort(L1); Mergesort(L2)
```

```
return Merge(L1, L2)
```

Verschmelzen zweier Teilfolgen



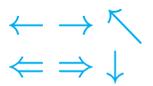
Merge Verschmelze die sortierten Folgen $A[l..m]$ und $A[m+1..r]$ zu einer Folge

```
static void merge(Orderable A[],int l,int m,int r) {
    comparable B [] = new Orderable[A.length];
    int i = l;          // Zeiger in A[l],...,A[m]
    int j = m + 1;     // Zeiger in A[m+1],...,A[r]
    int k = l;         // Zeiger in B[l],...,B[r]
    while (i <= m && j <= r) {
        if (A[i].less(A[j])) {
            B[k] = A[i]; i++;
        }
        else {
            B[k] = A[j]; j++;
        }
        k++;
    }
    if (i > m) { // erste Teilfolge erschöpft
        for (int h = j; h <= r; h++, k++) B[k] = A[h];
    }
    else { // zweite Teilfolge erschöpft
        for (int h = i; h <= m; h++, k++) B[k] = A[h];
    }
    for (int h = l; h <= r; h++) A[h] = B[h];
}
```

Beispiel:

2 4 8 15 17 25 43 47 62

9 12 15 16 28 33 37



```
class mergeSort {
  /* sortiert das ganze Array */
  public static void sort (Orderable A[]){
    sort(A,1,A.length-1);
  }
  static void merge ...
  static void sort (Orderable A[], int l, int r){
    if (r > l) {
      int m = (l + r) / 2;
      sort(A, l, m);
      sort(A, m+1, r);
      merge(A, l, m, r);
    }
  }
}
```

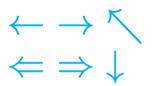
Beispiel: 8 6 7 3 4 5 2 1

divide 8 6 7 3 | 4 5 2 1

divide 8 6 | 7 3 | 4 5 | 2 1

divide 8 | 6 | 7 | 3 | 4 | 5 | 2 | 1

merge



Schlüsselvergleiche für Merge:

worst-case: $n_1 + n_2 - 1$

best-case: $\min(n_1, n_2)$

$$n_2 = r - m + 1 =$$

$n_1 = m - l =$ Länge der linken Teilfolge

$n_2 = r - m + 1 =$ Länge der rechten Teilfolge

Rekursionsgleichung für Laufzeit von Mergesort

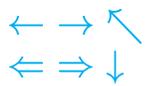
$$T(n) = 2T(n/2) + O(n)$$

$$O(n) = \begin{matrix} n/2 + n/2 - 1 & \text{worst case} \\ n/2 & \text{best case} \end{matrix}$$

$$\Rightarrow T(n) = n \log n - (n - 1)$$

zusätzlicher Speicher: $O(n)$

Reines 2-Wege-Mergesort



Ansatz:

for $i = 0$ to $\log n$

verschmelze Teilfolgen der Länge 2^i

8	6	9	3	4	7	2	1
6	8	3	9	4	7	1	2
3	6	8	9	1	2	4	7
1	2	3	4	6	7	8	9

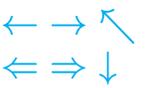
Natürliches 2-Wege-Mergesort

Ansatz: Wie reines Mergesort mit Ausnutzung vorsortierter Teilfolgen

8	6	9	3	4	7	2	1
6	8	9	2	3	4	7	1
2	3	4	6	7	8	9	1
1	2	3	4	6	7	8	9

$$C_{min} = n - 1 \quad C_{av}, C_{max} = O(n \log n)$$

8 Quicksort

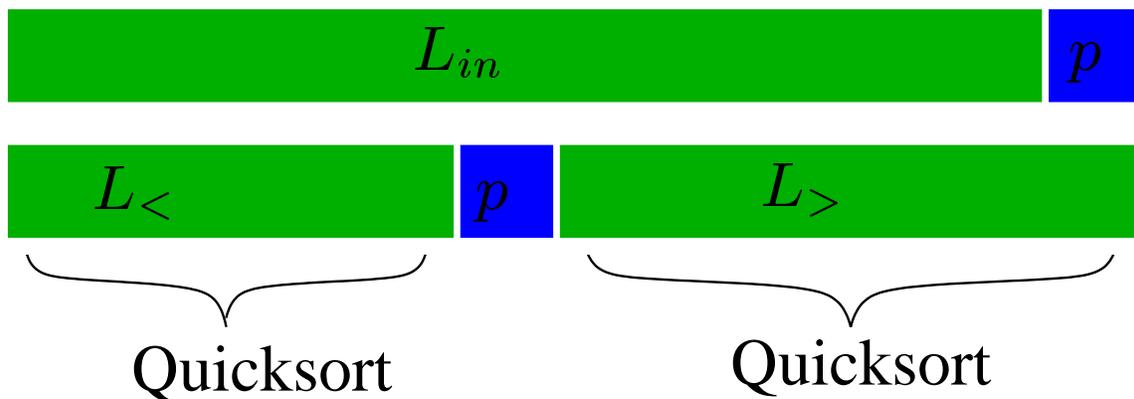


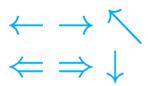
Sortieren durch Teilen

- Divide-&-Conquer Prinzip
- Sehr gute Laufzeit im Mittel, schlechte Laufzeit im worst case.

Quicksort Eingabe: unsortierte Liste L
sortierte Liste

```
if ( |L| <= 1 )
  return L
else waehle Pivotelement p aus L
  L< = {a in L | a < p}
  L> = {a in L | a > p}
  return
    [Quicksort(L<)] + [p] + [Quicksort(L>)]
```





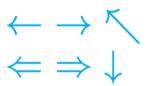
Eingliederung in das Rahmenprogramm:

```
class QuickSort extends SortAlgorithm {

    static void sort (Orderable A[]){
        /* sortiert das ganze Array */
        sort (A, 1, A.length-1);
    }

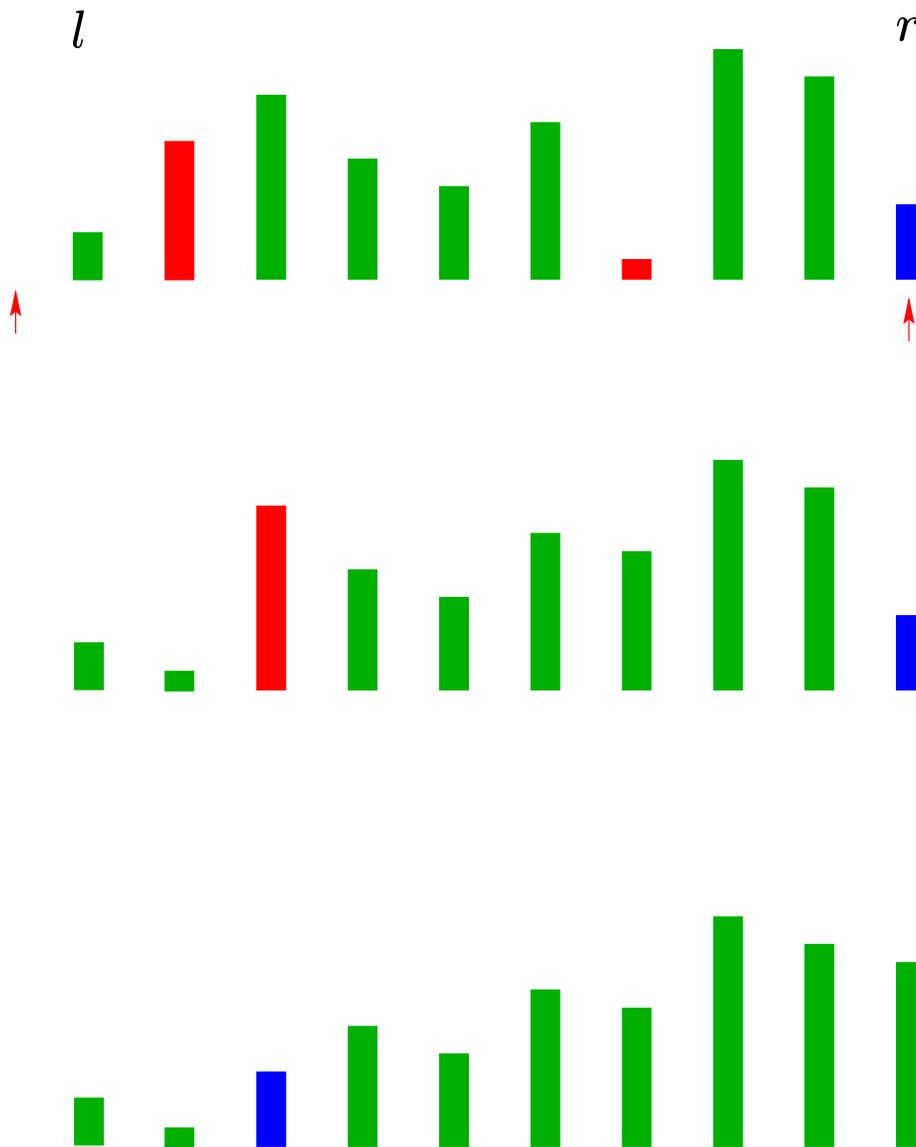
    static void sort (Orderable A[], int l, int r){
        /* sortiert das Array zwischen Grenzen l und r */
        if (r > l) { // mind. 2 Elemente in A[l..r]
            int i = divide(A, l, r);
            sort (A, l, i-1);
            sort (A, i+1, r);
        }
    }

    static int divide (Orderable A [], int l, int r) {
        /* teilt das Array zwischen l und r mit Hilfe
           des Pivot-Elements in zwei Teile auf und gibt
           die Position des Pivot-Elementes zurueck */
        ...
    }
}
```

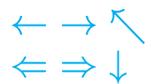


$\text{divide}(A, l, r)$:

- liefert den Index des Pivotelements in A
- ausführbar in Zeit $O(r - l)$



Implementation: Aufteilungsschritt



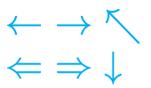
```
static int divide (Orderable A [], int l, int r) {

    // teilt das Array zwischen l und r mit Hilfe
    // des Pivot-Elements in zwei Teile auf und gibt
    // die Position des Pivot-Elementes zurueck

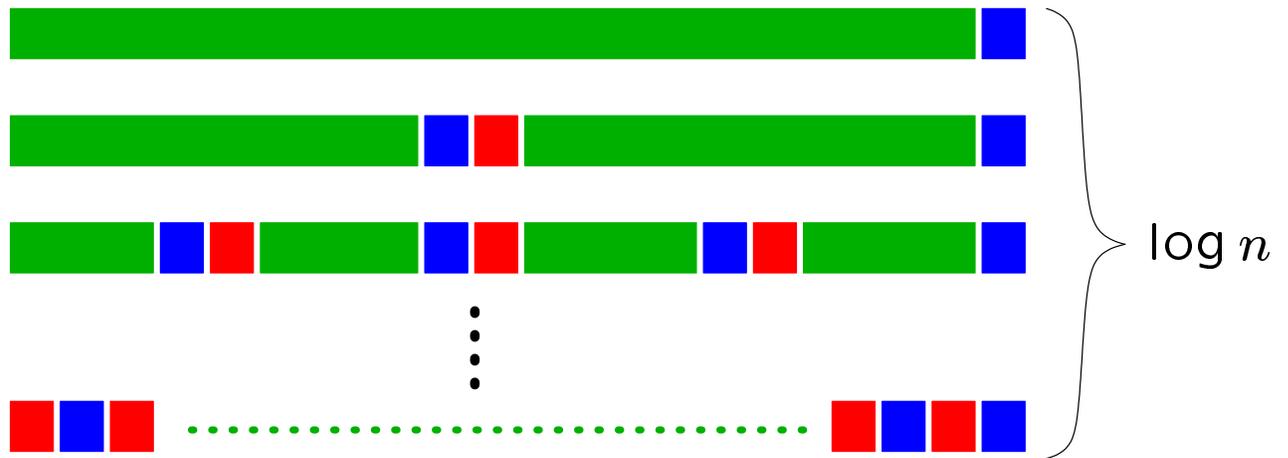
    int i = l-1;           // linker Zeiger auf Array
    int j = r;            // rechter Zeiger auf Array

    Orderable pivot = A [r]; // das Pivot-Element

    while (true){        // "Endlos"-Schleife
        do i++; while (i < j && A[i].less(pivot));
        do j--; while (i < j && A[j].greater(pivot));
        if (i >= j) {
            swap (A, i, r);
            return i;      // Abbruch der Schleife
        }
        swap (A, i, j);
    }
}
```

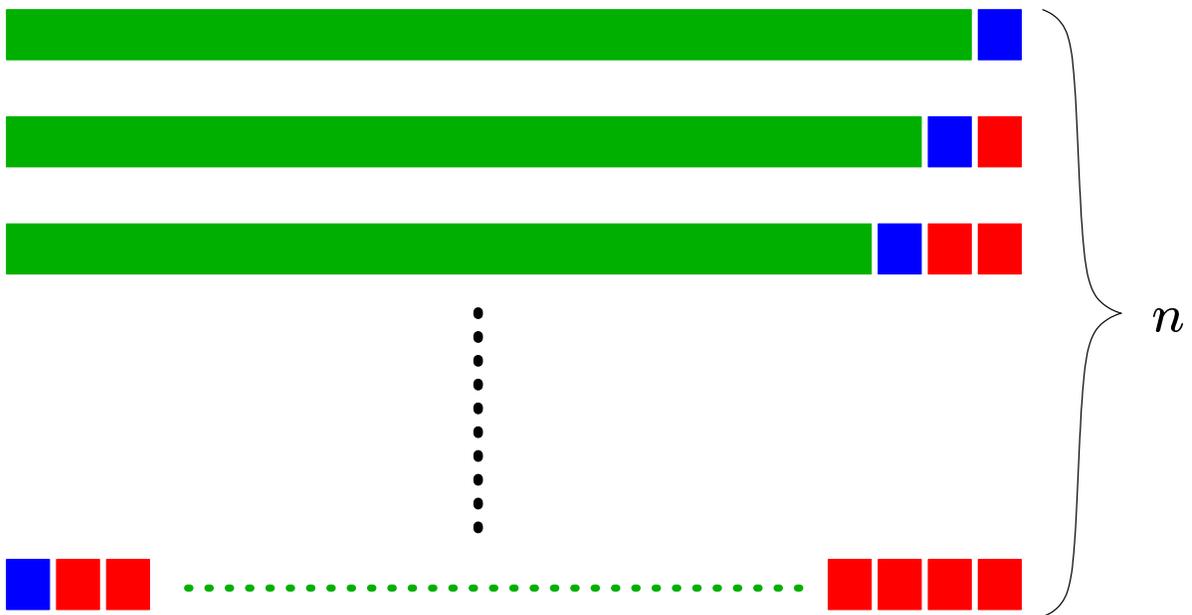


Günstigster Fall:

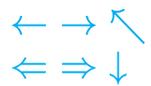


$$T_{min}(n) = O(n \log n)$$

Schlechtester Fall:



$$T_{max}(n) = O(n^2)$$



Best Case: Pivotelement teilt Folge jeweils genau zur Hälfte auf

Aufteilungsaufwand für Folge der Länge n : $n - 1$

$T_{min}(n)$ = Schlüsselvergleiche von Quicksort im besten Fall (C_{min})

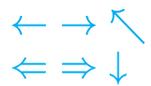
$$\begin{aligned}T_{min}(1) &= 0 \\T_{min}(n) &\leq 2T_{min}(n/2) + n - 1 \\ \Rightarrow T_{min}(n) &\leq n \log n - (n - 1)\end{aligned}$$

Worst Case: Eine der beiden durch Aufteilung entstehenden Folgen ist leer

(z.B. aufsteigend sortierte Folge von Schlüsseln)

$T_{max}(n)$ = Schlüsselvergleiche von Quicksort im schlechtesten Fall (C_{max})

$$\begin{aligned}T_{max}(1) &= 0 \\T_{max}(n) &= T_{max}(n - 1) + n - 1 \\ \Rightarrow T_{max}(n) &= T_{max}(n - 2) + (n - 2) + n - 1 \\ &\vdots \\ &= T(1) + \cdot 1 + \dots + (n - 2) + (n - 1) \\ &= \frac{n(n - 1)}{2} = O(n^2)\end{aligned}$$



Anahmen:

- n paarweise verschiedene Schlüssel
- Alle $n!$ Permutationen gleichwahrscheinlich

Fakt:

Teilfolgen des Aufteilungsschritts wieder zufällig

$$\Rightarrow P(\text{Pivotelement hat Rang } k) = 1/n.$$

Rekursionsgleichung:

$$C_{av}(0) = 0$$

$$C_{av}(1) = 0$$

$$C_{av}(n) = n - 1 + \frac{1}{n} \sum_{k=1}^n (C_{av}(k-1) + C_{av}(n-k))$$

Satz (Geschlossene Form)

$$C_{av}(n) \approx 1.386n \log n - 2.846n + O(\log n)$$

Beweis:

Exemplarisch lösen wir die Rekursionsgleichung, wobei wir C_{av} durch C abkürzen. Durch Multiplikation mit n

und Zusammenfassen der beiden Summanden (gleicher Indexbereich) gilt

$$nC(n) = n(n-1) + 2 \sum_{i=1}^n C(i-1)$$

In dieser Gleichung kommen noch $C(0), \dots, C(n)$ vor. Um die Zahl verschiedener Werte auf 2 zu senken, betrachten wir die oben gewonnene Gleichung auch für $n-1$ und subtrahieren diese Gleichung von der Gleichung für n .

$$(n-1)C(n-1) = (n-1)(n-2) + 2 \sum_{i=1}^{n-1} C(i-1)$$

Demnach ist

$$\begin{aligned} nC(n) - (n-1)C(n-1) &= \\ n(n-1) - (n-1)(n-2) + 2C(n-1) \end{aligned}$$

Vereinfachung und Division durch $(n+1)n$ liefert

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

Es ist nun naheliegend, $Z(n) = C(n)/(n+1)$ zu untersuchen.

$$\begin{aligned} Z(n) &= Z(n-1) + \frac{2(n-1)}{n(n+1)} \\ &= Z(n-2) + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} = \dots \\ &= Z(1) + 2 \sum_{i=2}^n \frac{i-1}{i(i+1)} \end{aligned}$$

Da $C(1) = 0$, ist $Z(1) = 0$. Es ist gut zu wissen, daß

$$\frac{1}{i(i+1)} = \frac{1}{i} - \frac{1}{i+1}$$

gilt. Also ist

$$\begin{aligned}
Z(n) &= 2 \sum_{i=2}^n \frac{i-1}{i(i+1)} \\
&= 2 \sum_{i=2}^n \frac{i-1}{i} - 2 \sum_{i=3}^{n+1} \frac{i-2}{i} \\
&= 1 + 2 \sum_{i=3}^n \frac{1}{i} - 2 \frac{n-1}{n+1}
\end{aligned}$$

Die Reihe $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots$ tritt so häufig auf, daß sie einen Namen, nämlich Harmonische Reihe $H(n)$, erhalten hat. Es folgt

$$Z(n) = 2H(n) - 2 - 2 \frac{n-1}{n+1} = 2H(n) - 4 + \frac{4}{n+1}$$

und

$$\begin{aligned}
C(n) &= (n+1)Z(n) = 2(n+1) \left(H(n) - 2 + \frac{2}{n+1} \right) \\
&= 2(n+1)H(n) - 4n
\end{aligned}$$

Was können wir über $H(n)$ aussagen? Aus der Betrachtung der Riemannschen Summen folgt

$$\text{int}_1^{n+1} \frac{1}{x} \leq H(n) \leq \text{int}_1^n \frac{1}{x}$$

und damit

$$\ln(n + 1) \leq H(n) \leq 1 + \ln n$$

Ohne größere Probleme läßt sich sogar zeigen, daß $H(n) - \ln n$ konvergiert. Der Grenzwert γ heißt

Eulersche Konstante, es gilt $\gamma \approx 0.57721 \dots$

Schließlich ist $\ln n = (\log n) \ln 2$. Insgesamt gilt also

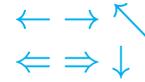
$$C(n) = 2(n + 1)H(n) - 4n$$

$$\approx 1.386n \log n - 2.846n + O(\log n)$$

Gleiche Analyse wie **Randomisiertes Quicksort**

- Pivot = zufälliges Element aus $A[l..r]$
- vertausche $A[r]$ mit Pivot
- weiter wie bisher

Worst-case: Laufzeit jetzt zufällig $C_{ex}(n)$



Überblick, **2**

Allgemeine Sortierverfahren, **4**

Analyse von Insertion Sort, **10**

Analyse von Mergesort, **17**

Analyse von Quicksort, **23**

Auswahlsort, **8**

Average Case Analyse Quicksort, **25**

Beispiel zu Shellsort, **12**

Best und Worst Case Quicksort, **24**

Das Sortierproblem, **3**

Der Aufteilungsschritt, **21**

Elementare Sortierverfahren, **5**

Grundlegende Implementation, **6**

Implementation, **20**

Implementation: Aufteilungsschritt, **22**

Implementation: Shellsort, **13**

Mergesort, **14, 16**

Quicksort, **19**

Reines 2-Wege-Mergesort, **18**

Shellsort, **11**

Verschmelzen zweier Teilfolgen, **15**