Datenstrukturen und Algorithmen

Vorlesungen 6-7: Suchen und Sortieren für lineare Datenstrukturen (K2,6,7,8)

Prof. Dr. Erika Ábrahám

Theorie Hybrider Systeme
Informatik 2

http://ths.rwth-aachen.de/teaching/ss-14/datenstrukturen-und-algorithmen/

Diese Präsentation verwendet in Teilen Folien von Joost-Pieter Katoen.

08. Mai 2014 und 09. Mai 2014

RWTH AACHEN UNIVERSITY

Übersicht

Annahmen

Annahmen dieser Vorlesung

- Die Daten sind als Array organisiert.
- Elementaroperationen sind Vergleiche von Schlüsseln.

Die meisten Algorithmen können auf andere linearen Datenstrukturen (wie z.B. Listen) angepasst werden.

Übersicht

Das Suchproblem

Rechenproblem

Eingabe: Array `E` mit `n` Einträgen, sowie das gesuchte Element `K`.

Ausgabe: Ist `K` in `E` enthalten?

Übersicht

Lineare Suche

```
1 bool linSearch(int E[], int n, int K) {
2   for (int index = 0; index < n; index ++) {
3     if (E[index] == K) {
4       return true; // oder: return index;
5     }
6   }
7   return false; // nicht gefunden
8 }
```

41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

Lineare Suche – Komplexitätsanalyse

Elementare Operation

Vergleich einer ganzen Zahl `K` mit Element `E[index]`.

Zeitkomplexität

- $W(n) = n \in \Theta(n)$, da n Vergleiche notwendig sind, falls `K` nicht in `E` vorkommt (oder wenn `K == E[n]`).
- $B(n) = 1 \in \Omega(1)$, da ein Vergleich ausreicht, wenn `K` gleich `E[1]` ist.
- $A(n) \approx \frac{1}{2}n$?, da im Schnitt `K` mit etwa der Hälfte der Array `E` verglichen werden muss? – Nein.

Intermezzo: Erwartungswerte

Quelle: Wikipedia

Betrachte einen einarmigen Banditen. Er hat 3 Räder und jedes Rad ist beschriftet mit Herz- und Karo-Symbolen.

Jedes Rad wird unabhängig von allen anderen Rädern angestoßen; jedes liefert Herz oder Karo, beides mit der Wahrscheinlichkeit $\frac{1}{2}$.

Gewinne:

- Alle Räder zeigen Herz: ganzer Jackpot
- Alle Räder zeigen Karo: die Hälfte des Jackpots
- Sonst: nichts.

Frage: Wieviel Prozent des Jackpots gewinnt man im Schnitt?

Antwort: $\frac{1}{8} \times 1 + \frac{1}{8} \times \frac{1}{2} + \frac{6}{8} \times 0 \;=\; \frac{3}{16}$.

Lineare Suche – Average-Case Analyse

Zwei Szenarien

1. K kommt nicht in E vor.
2. K kommt in E vor.

Zwei Definitionen

1. Sei $A_{K \notin E}(n)$ die (Average-Case) Laufzeit für den Fall "K nicht in E".
2. Sei $A_{K \in E}(n)$ die Average-Case Laufzeit für den Fall "K in E".

$$A(n) = \Pr\{\text{K in E}\} \cdot A_{K \in E}(n) + \Pr\{\text{K nicht in E}\} \cdot A_{K \notin E}(n)$$

Der Fall "K in E"

- Nehme an, dass alle Elemente in E unterschiedlich sind.
- Damit ist die Wahrscheinlichkeit für K == E[i] gleich $\frac{1}{n}$.
- Die Anzahl benötigter Vergleiche im Fall K == E[i] ist $i+1$.
- Damit ergibt sich:

$$\begin{aligned} A_{K \in E}(n) &= \sum_{i=0}^{n-1} \left(\frac{1}{n}\right) \cdot (i+1) \\ &= \left(\frac{1}{n}\right) \cdot \sum_{i=0}^{n-1} (i+1) \\ &= \left(\tfrac{1}{n}\right) \cdot \tfrac{n(n+1)}{2} \\ &= \tfrac{n+1}{2}. \end{aligned}$$

Herleitung

$$\begin{aligned} A(n) &= \Pr\{\text{K in E}\} \cdot A_{K \in E}(n) + \Pr\{\text{K nicht in E}\} \cdot A_{K \notin E}(n) \\ &\quad \mid A_{K \in E}(n) = \frac{n+1}{2} \\ &= \Pr\{\text{K in E}\} \cdot \frac{n+1}{2} + \Pr\{\text{K nicht in E}\} \cdot A_{K \notin E}(n) \\ &\quad \mid \Pr\{\text{nicht } B\} = 1 - \Pr\{B\} \\ &= \Pr\{\text{K in E}\} \cdot \frac{n+1}{2} + (1 - \Pr\{\text{K in E}\}) \cdot A_{K \notin E}(n) \\ &\quad \mid A_{K \notin E}(n) = n \\ &= \Pr\{\text{K in E}\} \cdot \frac{n+1}{2} + (1 - \Pr\{\text{K in E}\}) \cdot n \\ &= \left(1 - \frac{1}{2}\Pr\{\text{K in E}\}\right) \cdot n + \frac{1}{2}\Pr\{\text{K in E}\} \end{aligned}$$

Lineare Suche – Average-Case Analyse

Endergebnis

Die Average-Case Zeitkomplexität der linearen Suche ist:

$$A(n) = \left(1 - \frac{1}{2}\Pr\{\texttt{K in E}\}\right) \cdot n + \frac{1}{2}\Pr\{\texttt{K in E}\}$$

Beispiel

Wenn $\Pr\{K \text{ in } E\}$

- $= 1$, dann $A(n) = \frac{n+1}{2}$, d. h. etwa 50% von E ist überprüft.
- $= 0$, dann $A(n) = n = W(n)$, d. h. E wird komplett überprüft.
- $= \frac{1}{2}$, dann $A(n) = \frac{3 \cdot n}{4} + \frac{1}{4}$, d. h. etwa 75% von E wird überprüft.

Übersicht

Bilineare Suche

Statt eine Reihe in einer Richtung zu durchsuchen, kann man dies auch in beide Richtungen "zeitgleich".

Dies führt zur bilineare Suche.

```
1 bool bilinSearch(int E[], int n, int K) {
2   int left = 0, right = n - 1;
3   while (left <= right) {
4     if (E[left] == K || E[right] == K) {
5       return true;
6     }
7     left = left + 1;
8     right = right - 1;
9   }
10  return false; // nicht gefunden
11 }
```

41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

Bilineare Suche – Komplexitätsanalyse

Worst-Case Zeitkomplexität

Im schlimmsten Fall, wird die Schleife $\lceil \frac{n}{2} \rceil$ mal durchlaufen.

Pro Schleife finden zwei Vergleiche `E[i] == K` statt.

Also $W(n) = 2 \lceil \frac{n}{2} \rceil \in \Theta(n)$.

Best-Case Zeitkomplexität

$B(n) = 2 \in \Omega(1)$, da zwei Vergleiche reichen, wenn `K == E[1]` oder `K == E[n]`.

Average-Case Zeitkomplexität

Ähnlich wie für die lineare Suche.

Übersicht

Binäre Suche

Suchen in einem sortierten Array

Eingabe: Sortiertes Array `E` mit `n` Einträgen, und das gesuchte Element `K`.

Ausgabe: Ist `K` in `E` enthalten?

Idee

Da `E` sortiert ist, können wir das gesuchte Element `K` schneller suchen.
Liegt `K` nicht in der Mitte von `E`, dann:

1. suche in der linken Hälfte von `E`, falls `K` $<$ `E[mid]`
2. suche in der rechten Hälfte von `E`, falls `K` $>$ `E[mid]`

Fazit:

Wir halbieren den Suchraum in jedem Durchlauf.

Binäre Suche

```
1  bool binSearch(int E[], int n, int K) {
2    int left = 0, right = n - 1;
3    while (left <= right) {
4      int mid = floor((left + right) / 2); // runde ab
5      if (E[mid] == K) { return true; }
6      if (E[mid] > K) { right = mid - 1; }
7      if (E[mid] < K) { left = mid + 1; }
8    }
9    return false;
10 }
```

0	2	3	4	6	8	12	13	17	19	25	26	34	41	69

Binäre Suche – Komplexitätsanalyse

Rekursionsgleichung

Rekursionsgleichung für Worst-Case Laufzeit:

$$T(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 + T(\lceil (n-1)/2 \rceil) & \text{falls } n > 0 \end{cases}$$

Theorem

Die Worst-Case Laufzeit der binären Suche ist

$$\lfloor \log n \rfloor + 1 \in \Theta(\log n)\,.$$

Anwendung: Partyadresse

Problem

- n Freunde, die in n unterschiedlichen Wohnungen wohnen, beschließen n Parties zu geben.
- Jeder Freund gibt genau eine Party bei sich zu Hause in einer festgelegten Reihenfolge.
- Einer der Freunde lädt Sie zu einer der Parties ein.
- Sie haben eine Liste der Adressen aller n Freunde.
- Wo gehen Sie hin?
- Annahme: Wenn Sie zu eine Adresse hingehen, können Sie durchs Fenster schauen und feststellen, ob dort bereits eine Party stattgefunden hat oder gerade stattfindet.

Anwendung: Partyadresse

Lineare Suche

Besuchen Sie alle Adressen in der Ordnung, wie sie auf Ihrer Liste stehen, bis Sie die Party gefunden haben.

Bilineare Suche (parallelisiert)

Nehmen Sie eine Freundin mit. Besuchen Sie die Adressen auf der Liste von oben nach unten, die Freundin von unten nach oben. Sie rufen einander an, wenn Sie oder die Freundin die Party gefunden hat.

Anwendung: Partyadresse

Binäre Suche

Annahme: Sie wissen, dass die i-te Party an der i-ten Adresse auf Ihrer Liste stattfindet.

1. Gehen Sie zu der Adresse in der Mitte der Liste.
2. Wenn dort gerade eine Party läuft, dann freuen Sie sich und beenden Sie die Suche.
3. Wenn dort eine Party schon stattgefunden hat, dann schränken Sie die Liste auf die untere Listenhälfte ein.
4. Wenn dort noch keine Party stattgefunden hat, dann schränken Sie die Liste auf die obere Listenhälfte ein.
5. Machen Sie mit 1) weiter.

Übersicht

Die Bedeutung des Sortierens

Sortieren ist ein wichtiges Thema

- Sortieren wird häufig benutzt und hat viele Anwendungen.
- Sortierverfahren geben Ideen, wie man Algorithmen verbessern kann.
- Geniale und optimale Algorithmen wurden gefunden.

- Neben der Funktionsweise der Algorithmen widmen wir uns auch der Laufzeitanalyse.

Anwendungen des Sortierens

Beispiel (Suchen)

- Schnellere Suche ist die wohl häufigste Anwendung des Sortierens.
- Binäre Suche findet ein Element in $\mathcal{O}(\log n)$.

Beispiel (Engstes Paar (closest pair))

- Gegeben seien n Zahlen. Finde das Paar mit dem geringstem Abstand.
- Nach dem Sortieren liegen die Paare nebeneinander.
  Der Aufwand ist dann noch $\mathcal{O}(n)$.

Noch einige Anwendungen

Beispiel (Eigenschaften von Datenobjekten)

- Sind alle n Elemente einzigartig oder gibt es Duplikate?
- Welches Element einer Menge ist das k-größte?

Beispiel (Textkompression/Entropiekodierung)

- Sortiere die Buchstaben nach Häufigkeit des Auftretens um sie dann effizient zu kodieren (d.h. mit möglichst kurzen Bitfolgen).

Einige Hilfsbegriffe

Permutation

Eine Permutation einer Menge $A = \{a_1, \ldots, a_n\}$ ist eine bijektive Abbildung $\pi : A \to A$.

Totale Ordnung

Sei $A = \{a_1, \ldots, a_n\}$ eine Menge. Eine Relation $\leqslant \subseteq A \times A$ ist eine totale Ordnung (auf A) wenn für alle $a_i, a_j, a_k \in A$ gilt:

1. Antisymmetrie: $a_i \leqslant a_j$ und $a_j \leqslant a_i$ impliziert $a_i = a_j$.
2. Transitivität: $a_i \leqslant a_j$ und $a_j \leqslant a_k$ impliziert $a_i \leqslant a_k$.
3. Totalität: $a_i \leqslant a_j$ oder $a_j \leqslant a_i$.

Beispiel

Die lexikographische Ordnung von Zeichenketten und die numerische Ordnung von Zahlen sind totale Ordnungen.

Sortierproblem

Das Sortier-Problem

Eingabe:
1. Ein Array `E` mit `n` Einträgen.
2. Die Einträge gehören zu einer Menge A mit totaler Ordnung $\leqslant$.

Ausgabe: Ein Array `F` mit `n` Einträgen, so dass
1. `F[1]`, ..., `F[n]` eine Permutation von `E[1]`, ..., `E[n]` ist
2. Für alle $0 < i < n$ gilt: `F[i]` $\leqslant$ `F[i+1]`.

Sortieralgorithmen – Beispiele

Beispiel (Einige Sortieralgorithmen)

- Vergleichende Verfahren:
  Selectionsort, Bubblesort, Insertionsort, Quicksort, Mergesort, Heapsort, Blocksort, Coctailsort, Combsort, Cyclesort, Gnomesort, Introsort, Librarysort, Patiencesorting, Shellsort, Smoothsort, Stoogesort, Strandsort, Timsort, Turnamentsort, Unshufflesort, usw.
- Andere Verfahren:
  Countingsort, Bucketsort, Radixsort, Pigeonholesort, Spreadsort, usw.

Sortieralgorithmen – Begriffe

Begriffe

- Ein Sortieralgorithmus ist stabil wenn er die Reihenfolge der Elemente, deren Sortierschlüssel gleich sind, bewahrt.

  Wenn z.B. eine Liste alphabetisch sortierter Personendateien nach dem Geburtsdatum neu sortiert wird, dann bleiben unter einem stabilen Sortierverfahren alle Personen mit gleichem Geburtsdatum alphabetisch sortiert.
- Ein Sortieralgorithmus ist in-place wenn er keinen zur Eingabelänge proportionalen Anteil der Eingabe kopiert.

Übersicht

Selectionsort (auch Minsort/Maxsort genannt)

Idee: Suche kleinstes Element und platziere ganz links, suche zweitkleinstes Element und platziere zweitlinks, ...

```
1 void selectionSort(int E[], int lengthOfE) {
2   for (int left=0; left<lengthOfE-1; left++) {
3     int min = left;
4     for (int i=left+1; i<lengthOfE; i++)
5       if (E[i] < E[min])
6         min = i;
7     swap(E[min],E[left]);
8   }
9 }
10 //swap(E[min],E[left]) steht fuer
11 // int tmp = E[min]; E[min] = E[left]; E[left] = tmp;
```

3	8	17	17	19	25	26	41

Selectionsort - Eigenschaften

Eigenschaften

- Zeitkomplexität (Best-Case, Average-Case, worst-Case) in $\Theta(n^2)$.
- In-place, Speicherplatzbedarf in $\Theta(1)$.
- Nicht stabil, es gibt aber auch stabile Varianten.

Übersicht

Bubblesort - Idee

Idee

- Gehe durch die Liste und vertausche alle nebeneinanderstehenden Paare, die in falscher Ordnung stehen.
- Die Liste rechts von der letzten Tauschposition ist bereits sortiert.
- Wenn keine Änderung, dann fertig.
- Sonst gehe wieder durch die Liste...

Beim Sortieren steigen/sinken die Elemente zu ihrem Platz in der Ordnung wie Blasen im Wasser auf (daher der Name).

Bubblesort

```
1 void bubbleSort(int E[], int lengthOfE) {
2   while (lengthOfE > 1) {
3     int n = 1;
4     for (int i=0; i<lengthOfE-1; i++) {
5       if (E[i] > E[i+1]){
6         swap(E[i],E[i+1]);
7         n = i+1;
8       }
9     }
10    lengthOfE = n;
11  }
12 }
13 //swap (E[i],E[i+1]) steht fuer
14 // int tmp = Ei]; E[i] = E[i+1]; E[i+1] = tmp;
```

3	8	17	17	19	25	26	41

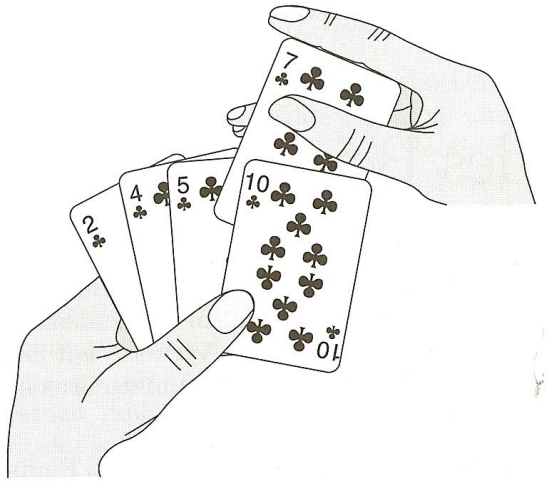
Bubblesort - Eigenschaften

Eigenschaften

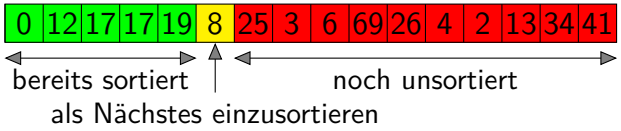
- Zeitkomplexität: Best-Case in $\Theta(n)$, Worst-Case und Average-Case in $\Theta(n^2)$
- In-place, Speicherplatzbedarf in $\Theta(1)$.
- Stabil.

Übersicht

Sortieren durch Einfügen – Insertionsort

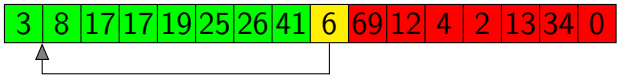


Sortieren durch Einfügen – Insertionsort



- Durchlaufen des (unsortierten) Arrays von links nach rechts.
- Gehe zum ersten bisher noch nicht berücksichtigten Element.
- Füge es im sortierten Teil (links) nach elementweisem Vergleichen ein.

Insertionsort – Animation und Algorithmus




```
1 void insertionSort(int E[], int lengthOfE) {
2   for (int i = 1; i < lengthOfE; i++) {
3     int v = E[i]; // speichere E[i]
4     int j = i-1;
5     while (j >= 0 && E[j] > v) {
6       E[j+1] = E[j]; // schiebe Element j-1 nach rechts
7       j = j-1;
8     }
9     E[j+1] = v; // füge E[i] an der richtigen Stelle ein
10  }
11 }
```


Insertionsort – Eigenschaften

Eigenschaften

- Zeitkomplexität: Best-Case in $\Theta(n)$, Average-Case und Worst-Case in $\Theta(n^2)$.
- In-place, Speicherplatzbedarf in $\Theta(1)$. Speicherplatz.
- Stabil.

Übersicht

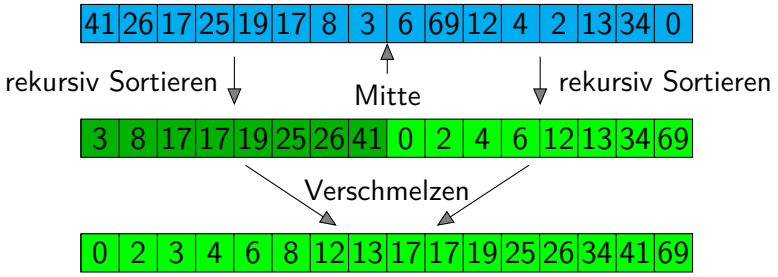
Mergesort: Divide-and-Conquer

Vorgestellt durch John von Neumann 1945.

Erinnerung: Teile-und-Beherrsche (divide-and-conquer)

Teile	das Problem in eine Anzahl von Teilproblemen auf.
Beherrsche	die Teilprobleme durch rekursives Lösen. Hinreichend kleine Teilprobleme werden direkt gelöst.
Verbinde	die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems.

Mergesort – Strategie



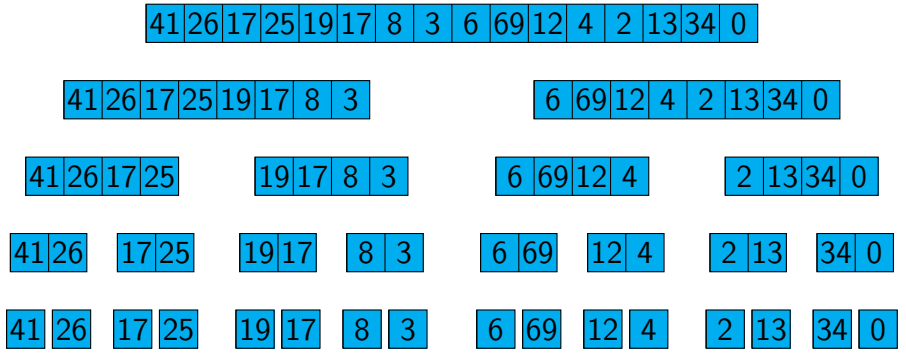
Teile	das Array in zwei – möglichst gleichgroße – Hälften.
Beherrsche:	Sortiere die Teile durch rekursive Mergesort-Aufrufe.
Verbinde:	Mische je 2 sortierte Teilsequenzen zu einem einzigen, sortierten Array.

Mergesort – Algorithm

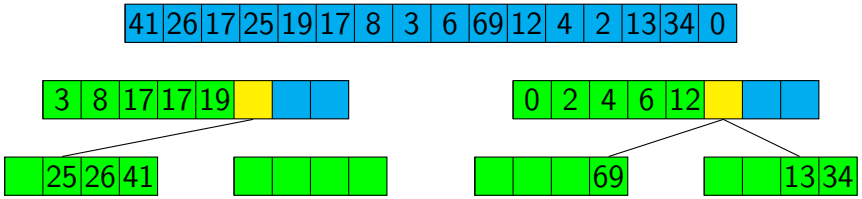
```
1 void mergeSort(int E[], int left, int right) {
2   if (left < right) {
3     int mid = (left + right) / 2; // finde Mitte
4     mergeSort(E, left, mid);      // sortiere linke Hälfte
5     mergeSort(E, mid + 1, right); // sortiere rechte Hälfte
6     // Verschmelzen der sortierten Hälften
7     merge(E, left, mid, right);
8   }
9 }
10 // Aufruf: mergeSort(E, 0, lengthOfE-1);
```

- Verschmelzen kann man in Linearzeit. – Wie?

Mergesort – Animation



Mergesort – Animation



Mergesort – Verschmelzen in Linearzeit

```
1  void merge(int E[], int left, int mid, int right) {
2    int a = left, b = mid + 1;
3    int Eold[] = E;
4    for (; left <= right; left++) {
5      if (a > mid) { // Wir wissen (Widerspruch): b <= right
6        E[left] = Eold[b];
7        b++;
8      } else if (b > right || Eold[a] <= Eold[b]) { // stabil: <=
9        E[left] = Eold[a];
10       a++;
11     } else { // Eold[a] > Eold[b]
12       E[left] = Eold[b];
13       b++;
14     }
15   }
16 }
```

- Mergesort ist stabil (vgl. Zeile 8).

Mergesort – Komplexitätsanalyse

Worst-Case

Wir erhalten:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \quad \text{mit} \quad W(1) = 0.$$

Mit Hilfe des Mastertheorems ergibt sich: $W(n) \in \Theta(n \cdot \log n)$.

Best-Case, Average-Case

Das Worst-Case Ergebnis gilt genauso im Best-Case, und damit auch im Average-Case: $W(n) = B(n) = A(n) \in \Theta(n \cdot \log n)$.

Speicherbedarf

$\Theta(n)$ für die Kopie des Arrays beim Mergen. $\Theta(\log n)$ für den Stack.

- Mergesort ist nicht in-place.
- Mit zusätzlichen Verschiebungen ist die Kopie des Arrays nicht nötig.

Übersicht

Quicksort – Idee

Quicksort

- Divide-and-conquer: Mergesort teilt das Array in zwei Hälften, sortiert rekursiv die Teilarrays und fügt sie dann zu einem sortierten Gesamtarray zusammen.
- Quicksort arbeitet ähnlich, aber die Aufteilung in zwei Hälften ist anders.
- Quicksort wurde 1961 von Tony Hoare (Großbritanien) entwickelt.

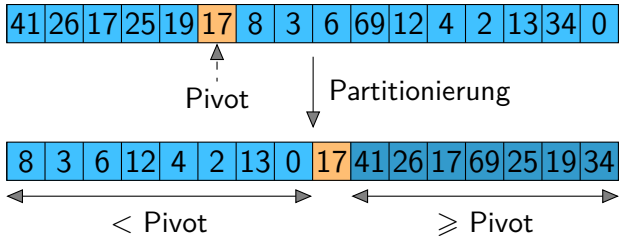
Quicksort – Tony Hoare

Charles Antony Richard Hoare (geb. 1934)

"I conclude that 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. "
(Dankesrede für den Turing-Preis 1980)

"I think that Quicksort is the only really interesting algorithm I've ever developed. "

Quicksort – Strategie



Teile Wähle ein Pivotelement aus dem zu sortierenden Array und partitioniere das zu sortierende Array in zwei Teile auf:

1. Kleiner als das Pivotelement, sowie
2. mindestens so groß wie das Pivotelement.

Beherrsche: Sortiere die Teile rekursiv und setze dann das Pivotelement zwischen die sortierten Teile.

Verbinde: Da die Teilfelder in-place sortiert werden ist keine Arbeit nötig, um sie zu verbinden.

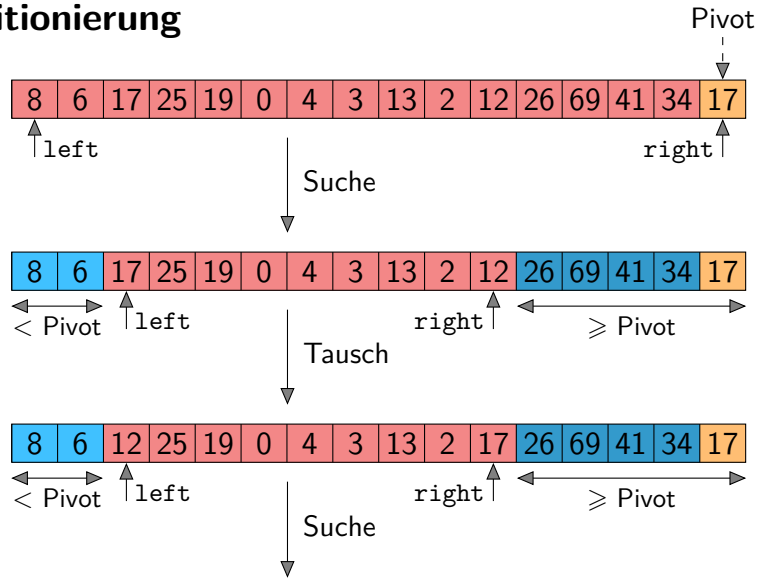
Partitionierung

Sobald ein Pivot ausgewählt ist, kann das Array in $\mathcal{O}(n)$ partitioniert werden, z. B. folgendermaßen:

- Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $<$ Pivot ist.
- Schiebe die rechte Grenze nach links, solange das zusätzliche Element $\geqslant$ Pivot ist.
- Tausche das links gefundene mit dem rechts gefundenen Element.
- Fahre fort, bis sich die Grenzen treffen.

(Es gibt auch andere Verfahren.)

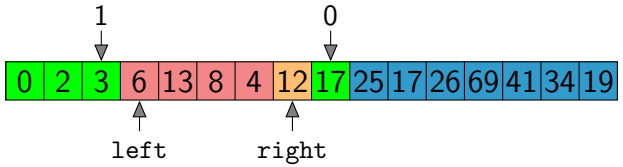
Partitionierung



Partitionierung – Algorithmus

```
1 int partition(int E[], int left, int right) {
2   // Wähle einfaches Pivotelement
3   int ppos = right, pivot = E[ppos];
4   while (left < right) {
5     // Bilineare Suche
6     while (left < right && E[left] < pivot) left++;
7     while (left < right && E[right] >= pivot) right--;
8     if (left < right) {
9       swap(E[left], E[right]);
10    }
11  }
12  swap(E[left], E[ppos]);
13  return left; // gib neue Pivotposition als Splitpunkt zurück
14 }
```


Quicksort – Algorithmus und Animation




```
1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     int i = partition(E, left, right);
4       // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }
```


Quicksort - Illustration

Quicksort – Speicherkomplexität

Quicksort ist nicht stabil aber in-place, da Array-Elemente nicht kopiert werden. – Aber:

- Die rekursiven Aufrufe benötigen Speicherplatz für alle `left` und `right` Parameter.
- Im Worst-Case wird durch die Partitionierung nur ein Element abgespalten.
- Dann wird für diese n Elemente $\Theta(n)$ Platz auf dem Stack benötigt.
- Man kann den Platzbedarf aber auf $\Theta(\log n)$ reduzieren (siehe Aufgabe 7-4 im Buch).
- Hauptidee: sortiere nur große Teilarrays rekursiv, die kleineren iterativ.

Quicksort – Worst-Case Zeitkomplexität

- Auf jede Ebene des Rekursionsbaums werden $\Theta(n)$ Schritte benötigt (zum Teilen).
- Gesamtzeitaufwand: $\Theta(n)$ mal die Höhe des Rekursionsbaumes.
- Worst-Case: Pivot ist das kleinste oder größte Element.
  Dadurch ist das Partitionieren maximal unbalanciert:
  Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.
  (Komischerweise ist das der Fall, wenn das Array bereits sortiert ist.)
  $\Rightarrow \Theta(n)$ Ebenen im Rekursionsbaum.
  $\Rightarrow W(n) \in \Theta(n^2)$

Das ist aber genauso schlecht, wie Insertionsort, Mergesort, usw.
Woher also das „quick“ in Quicksort?

Quicksort – Best-Case Zeitkomplexität

- Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so gleichmäßig wie möglich geschieht.
- Wenn das Pivot-Element ist immer der Median, dann teilen wir die Arrays immer in zwei Teilen der gleichen Größe und erhalten $\log n$ Ebenen im Rekursionsbaum.
  $\Rightarrow B(n) \in \Theta(n \cdot \log n)$.

Die Ausbalanciertheit der beiden Hälften der Zerlegung in jeder Rekursionsstufe erzeugt also einen asymptotisch schnelleren Algorithmus.

Fazit: Wenn man eine Aufgabe zerlegt, ist es am Besten, in gleich große Teile zu teilen.

Quicksort – Balancierte Zerlegung

- Die mittlere Laufzeit von Quicksort ist viel näher an der des besten Falls als an der schlechtesten Falls.
- Schlüssel zum Verständnis: wie schlägt sich die Balanciertheit der Zerlegung in der Rekursionsgleichung nieder?
- Betrachte z. B. eine Zerlegung im Verhältnis 9:1.
  $\Rightarrow \Theta(\log n)$ Ebenen im Rekursionsbaum.
  $\Rightarrow T(n) \in \Theta(n \cdot \log n)$
- Diese 9:1 “unbalancierte” Zerlegung liefert asymptotisch die gleiche Zeit wie bei einer Aufteilung zu gleichen Teilen!
- Eine Aufteilung im Verhältnis 99:1 liefert ebenso: $T(n) \in \Theta(n \cdot \log n)$.

Quicksort – Average-Case Zeitkomplexität

Idee: Wenn alle Eingaben der gleichen Länge gleich wahrscheinlich sind, dann ist die Länge der jeweils längeren Teilliste beim rekursiven Aufruf im Schnitt

$$2 \cdot \sum_{i=n/2}^{n-1} \frac{1}{n} \cdot i = \frac{2}{n} \sum_{i=n/2}^{n-1} i = \frac{3}{4}n + c\,.$$

D.h., die Tiefe der Rekursion ist in $\Theta(\log n)$ und die Zeitaufwand in $\Theta(n \cdot \log n)$.

Übersicht

Heaps

Heap (Haufen)

Ein Heap ist ein Binärbaum, der Elemente mit Schlüsseln enthält und in ein Array eingebettet ist. Die Heap-Bedingung für *Max-Heaps* fordert:

- Der Schlüssel eines Knotens ist stets größer als (bzw. mindestens so groß wie) die Schlüssel seiner Kinder.

Weiter gilt:

- Alle Ebenen, abgesehen von evtl. der untersten, sind komplett gefüllt.
- Die Blätter befinden sich damit alle auf einer (höchstens zwei) Ebene(n).
- Die Blätter der untersten Ebene sind linksbündig angeordnet.

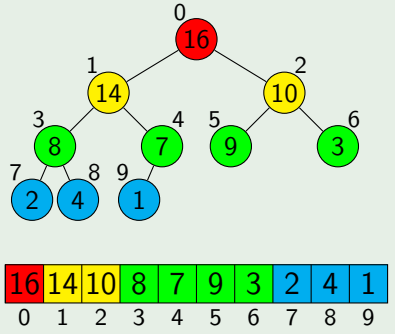
Arrayeinbettung eines Heaps

Arrayeinbettung

Das Array `a` wird wie folgt als Binärbaum aufgefasst:

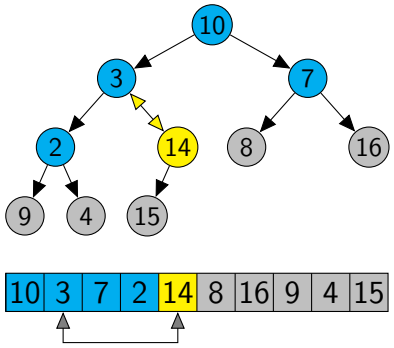
- Die Wurzel liegt in `a[0]`.
- `left(i)=2*i+1`: Das linke Kind von `a[i]` liegt in `a[2*i+1]`.
- `rigth(i)=2*i+2`: Das rechte Kind von `a[i]` liegt in `a[2*i+2]`.

Beispiel



- Durch die möglichst vollständige Füllung der Ebenen werden „Löcher“ im Array vermieden.
- Vergrößert man den Baum um ein Element, so wird das Array gerade um ein Element länger.

Naiver Heapaufbau



Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau – Algorithmus und Analyse

```
1 void bubble(int E[], int pos) {
2   while (pos > 0) {
3     int parent = (pos - 1) / 2;
4     if (E[parent] > E[pos]) {
5       break;
6     }
7     swap(E[parent], E[pos]);
8     pos = parent;
9   }
10 }
```

Die Höhe k eines Heaps mit n Elementen ist beschränkt durch:

$$n \leqslant 2^{k+1} - 1 \quad \Rightarrow \quad k = \lfloor \log n \rfloor$$

- Damit kostet jedes Einfügen $k \approx \log n$ Vergleiche.

$\Rightarrow$ Zum Aufbau eines Heaps mit n Elementen benötigt man $\Theta(n \cdot \log n)$ Vergleiche.

Es geht effizienter: `heapify` (auch: sink, fixheap) [Floyd 1964]

Heapify – Strategie

Betrachte `E[i]` unter der Annahme, der rechte und linke Teilbaum ist bereits ein Heap.

- `E[i]` kann kleiner als seine Kinder sein.
- Wir wollen die beiden Teilbäume – Heaps – zusammen mit `E[i]` zu einem (Gesamt-)Heap verschmelzen.
- Dazu lassen wir `E[i]` in den Heap hineinsinken, so dass der Teilbaum mit Wurzel `E[i]` ein Heap ist.

Heapify

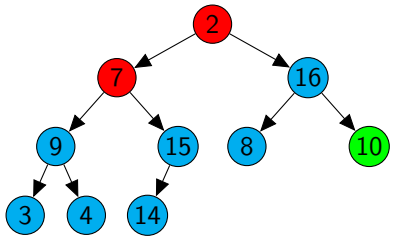
- Finde das Maximum der Werte `E[i]` und seiner Kinder.
- Ist `E[i]` bereits das größte Element, dann ist dieser gesamte Teilbaum auch ein Heap. Fertig.
- Andernfalls tausche `E[i]` mit dem größten Element und führe Heapify in diesem Unterbaum weiter aus.

Heapify – Algorithmus und Beispiel

```
1 void heapify(int E[], int n, int pos) {
2   int max = left(pos);
3   while (max < n) {
4     if (right(pos) < n && E[right(pos)] > E[max]) {
5       max = right(pos);
6     }
7     if (E[pos] >= E[max]) {
8       break;
9     }
10    swap(E[pos], E[max]);
11    pos = max;
12    max = left(pos);
13  }
14 }
```

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.




```
1 void buildHeap(int E[], int lengthOfE) {
2   for (int i = lengthOfE / 2 - 1; i >= 0; i--) {
3     heapify(E, lengthOfE, i);
4   }
5 }
```

Nach jedem Aufruf von `heapify(E, lengthOfE, i)` sind die Knoten `i`, `...`, `lengthOfE - 1` schon Wurzeln von Heaps.

Konstruktion eines Heaps

Lemma

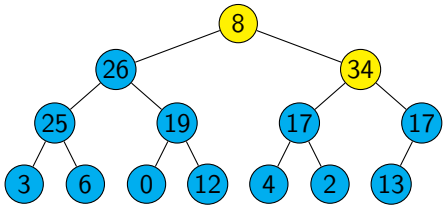
Der Algorithmus `buildHeap` *ist korrekt und terminiert.*

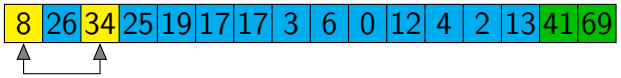
- Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor + 1$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
- Schleifeninvariante: Zu Beginn der `for`-Schleife ist jeder Knoten $i{+}1, \ldots,$ *lengthOfE* die Wurzel eines Heaps.
- In jeder Iteration sind alle Kinder des Knotens i bereits Wurzeln eines Heaps (Schleifeninvariante).

$\Rightarrow$ Bedingung für den Aufruf von `heapify` ist erfüllt.

- Dekrementierung von i stellt Schleifeninvariante wieder her.
- Terminierung: Bei $i = 0$ ist gemäß Schleifeninvariante jeder Knoten $1, 2, \ldots, n$ die Wurzel eines Heaps.

Heapsort – Algorithmus und Beispiel






```
1 void heapSort(int E[], int lengthOfE) {
2   buildHeap(E, lengthOfE);
3   for (int i = lengthOfE - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
```


Heapsort - Illustration

Heapsort – Komplexitätsanalyse

```
1 void heapSort(int E[], int lengthOfE) {
2   buildHeap(E, lengthOfE);
3   for (int i = lengthOfE - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
```

- Die Worst-Case Komplexität von `Heapify` ist in $\Theta(\log n)$ für n Knoten.
- Die Worst-Case Komplexität von `buildHeap` ist in $\Theta(n)$
- Für `Heapsort` erhalten wir somit $W(n) \in \Theta(n \cdot \log n)$
- Es wird kein zusätzlicher Speicherplatz benötigt.

Heapsort – Zusammenfassung

- Heapsort sortiert in $\mathcal{O}(n \cdot \log n)$
- Heapsort ist ein in-place Algorithmus mit Speicherkomplexität $\Theta(1)$.
- Heapsort ist nicht stabil.

Übersicht

Komplexität von vergleichenden Sortieralgorithmen

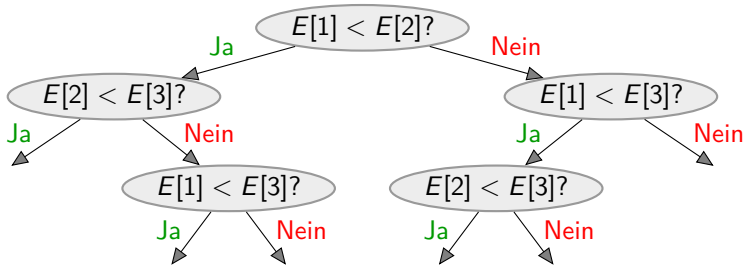
	Zeitbedarf			
	Worst-Case	*Average-Case*	*Platzbedarf*	*Stabil*
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	N*
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	J
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	J
Mergesort	$\Theta(n \cdot \log n)$	$\Theta(n \cdot \log n)$	$\Theta(n)$	J
Quicksort	$\Theta(n^2)$	$\Theta(n \cdot \log n)$	$\Theta(\log n)$	N*
Heapsort	$\Theta(n \cdot \log n)$	$\Theta(n \cdot \log n)$	$\Theta(1)$	N

* es gibt Varianten die stabil sind.

Einige Bemerkungen

- Insertion Sort ist einfach und ziemlich effizient für kleinere Arrays und fast sortierte Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- Mergesort ist ein sehr effizientes Sortierverfahren und wird u.a. benutzt in Perl, Python, und Java. Ist leicht anpassbar für Listen und ist stabil.
- Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log n)$ braucht für fast sortierte Eingaben.
- Quicksort ist typischerweise ein effizientes Verfahren. Die Wahl des Pivots ist wichtig. Nicht stabil, und nicht so effizient auf fast sortierte Eingaben.
- Kopieren haben wir in der Analyse vernachlässigt (da Anzahl Kopieren kleiner als Anzahl Vergleiche). Große Elemente können zu langen Kopierzeiten führen, d.h., in der Praxis kann die Elementgröße relevant sein.

Wie effizient kann man sortieren?



Betrachte vergleichende Verfahren als Entscheidungsbaum:

- Dieser beschreibt die Abfolge der durchgeführten Vergleiche.
- Sortieren verschiedener Eingabepermutationen ergibt also verschiedene Pfade im Baum.

Wie effizient kann man sortieren?

Theorem

Vergleichende Verfahren benötigen im Worst-Case mindestens $\mathcal{O}(n \log n)$ Vergleiche.

Beweis.

Es gilt:

- Anzahl Vergleiche im Worst-Case = Länge des längsten Pfades = Baumhöhe k.
- Da wir binäre Vergleiche verwenden, ergibt sich ein Binärbaum mit $n!$ Blättern.

$\Rightarrow$ Mit $n! \leqslant 2^k$ erhält man $k \geqslant \lceil \log(n!) \rceil$ Vergleiche im Worst-Case.

Da $\lceil \log(n!) \rceil \approx n \cdot \log n - 1.4 \cdot n$, es geht nicht besser als $\mathcal{O}(n \cdot \log n)$! □

Übersicht

Countingsort

Countingsort: Idee

- Annahme: Die Schlüssel aller Elemente des Arrays E sind aus $\{0, \ldots, k\}$ für ein bekanntes k.
- Idee: Zähle wie oft die einzelnen Schlüssel auftreten und berechne daraus die Positionen der Arrayelemente nach dem Sortieren.
- Benötigt über Vergleich/Kopieren hinaus weitere Berechnungen
  Benötigt zusätzlichen Speicher

Countingsort

```
1  void countingSort(int[] E, int lengthOfE, int nrOfKeys){
2    int sorted[lengthOfE], index[nrOfKeys];
3    for (int i=0; i<nrOfKeys; i++)
4      index[i] = 0;
5    for (int j=0; j<lengthOfE; j++) //index[i] = Anzahl Key i in
         E
6      index[E[j]]++;
7    for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i
         in E
8      index[i] += index[i-1];
9    for (j=lengthOfE-1; j>=0; j--){
10     sorted[index[E[j]]-1] = E[j];
11     index[E[j]]--;
12   }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2

index	0	1	3	6	9

sorted	0	1	1	2	2	2	3	3	3	4	4

Countingsort - Komplexitätsanalyse

Komplexität von Countingsort

- Laufzeit in $\mathcal{O}(n + \text{number of keys})$
- Speicherbedarf in $\mathcal{O}(n + \text{number of keys})$
- Stabil

Übersicht

Radixsort – Historisches

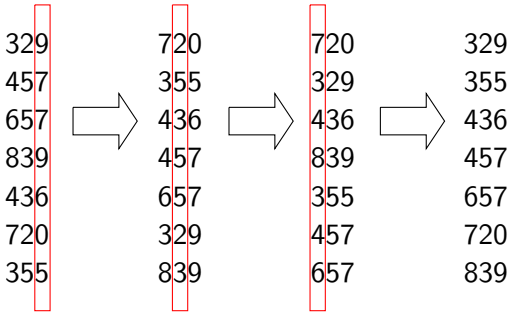
Radixsort

Radixsort wurde von Lochkartensortiermaschinen verwendet.

- Lochkarte: d Spalten mit je einem Loch an einem von k möglichen Positionen
- Bezeiche $1 \leqslant L(i) \leqslant k$ die Position des Loches in Spalte i auf Lochkarte L
- Ordnung auf Lochkarten: $L_1 < L_2$ wenn
  - $L_1(1) < L_2(1)$ oder
  - $L_1(1) = L_2(1)$ und $L_1(2) < L_2(2)$ oder
  - $L_1(1) = L_2(1)$ und $L_1(2) = L_2(2)$ und $L_1(3) < L_2(3)$ oder ...

$\Rightarrow$ Lexikographische Ordnung.

Radixsort auf Dezimalzahlen



Allgemeine Anwendung: Sortieren von Datensätzen mit Schlüsseln, die aus mehreren Komponenten bestehen.

Radixsort – Algorithmus

Algorithmus

Verwende einen stabilen Algorithmus um die Elemente eines Arrays iterativ nach allen Schlüsselkomponenten (in einer gegebenen Prioritätsordnung) zu sortieren.

```
1 void radixSort(Element[] E, int lengthOfE, int keyKomponents) {
2   for (int i=keyComponents; i>=1; i--)
3     sortiere E nach der i-ten Schluesselkomponente;
4 }
```


Radixsort – Zeitkomplexität

Zeitkomplexität

- Wir wollen Arrays mit n Elementen sortieren.
- Die Elemente haben d Schlüsselkomponenten.
- Jede Schlüsselkomponente hat k mögliche (geordnete) Werte.

Wenn das verwendete stabile Sortierverfahren $\Theta(n+k)$ Zeit benötigt, dann ist Radixsort in $\Theta(d \cdot (n+k))$.

Übersicht

Bucketsort

Idee

- Voraussetzung:
  - Die Elemente des Arrays sind aus dem Intervall $[0, 1)$.
  - Sie werden von einem zufälligen Prozess gleichmäßig und unabhängig voneinander verteilt.
- Idee: Teile $[0, 1)$ in n gleich große Teilintervalle (Buckets) auf.
- Erwartung: Nur wenige Zahlen fallen auf das gleiche Bucket.
- Algorithmus:
  - Ordne die Elemente in die Buckets.
  - Sortiere die Buckets.
  - Gebe die Elemente der Buckets in aufsteigender Reihenfolge aus.

Bucketsort – Algorithmus

```
1 void bucketSort(float E[], int lengthOfE) {
2   for (int i=0; i<lengthOfE; i++)
3     add E[i] to bucket B[n*E[i]]; //n*E[i] nach unten runden
4   for (int i=0; i<lengthOfE; i++)
5     sort B[i];
6   connect buckets to B[0]...B[n-1];
7 }
```

Average-Case Laufzeit in $\mathcal{O}(n)$.

Bucketsort – Beispiel

E:	0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
B:		0.17 0.12	0.26 0.21 0.23	0.39			0.68	0.78 0.72		0.94
sorted B:		0.12 0.17	0.21 0.23 0.26	0.39			0.68	0.72 0.78		0.94
sorted E:	0.12	0.17	0.21	0.23	0.26	0.39	0.68	0.72	0.78	0.94

Übersicht

Komplexität von Sortieralgorithmen

Zeitbedarf

	Worst-Case	*Average-Case*	*Platzbedarf*	*Stabil*
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	N*
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	J
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	J
Mergesort	$\Theta(n \cdot \log n)$	$\Theta(n \cdot \log n)$	$\Theta(n)$	J
Quicksort	$\Theta(n^2)$	$\Theta(n \cdot \log n)$	$\Theta(\log n)$	N*
Heapsort	$\Theta(n \cdot \log n)$	$\Theta(n \cdot \log n)$	$\Theta(1)$	N
Countingsort	$\Theta(n + r)$	$\Theta(n + r)$	$\Theta(n + r)$	J
Radixsort	$\Theta(n \cdot \frac{k}{d})$	$\Theta(n \cdot \frac{k}{d})$	$\Theta(n)$	J
Bucketsort	$\Theta(n^2 \cdot k)$	$\Theta(n + k)$	$n \cdot k$	J

* es gibt Varianten die stabil sind.