Sortieralgorithmen

Theorie

Inhaltsverzeichnis

Version vom 11. November 2025

1 Selection Sort

Beschreibung

Zuerst wird das kleinste Element im gesamten Array gesucht und mit dem Element an der ersten Stelle vertauscht, selbst wenn das kleinste Element schon an der ersten Stelle steht.

Danach sucht man in der Teilliste ab dem zweiten Element das kleinste Element und tauscht es mit dem Element an der zweiten Stelle.

Auf die gleiche Weise fahren wir fort, bis an der zweiletzten Position das richtige Element steht. Das letzte Element ist dann automatisch am richtigen Platz. Warum?

Beispiel

13	5	2	21	8	Vergleiche	Vertauschungen

Python-Implementierung

```
1  def selectionsort(A):
2      '''Sortiert das Array A aufsteigend/inplace'''
3      n = len(A)
4      # nach dem i-ten Durchlauf sind die Elemente an
5      # den Positionen 0, 1, ..., i sortiert.
6      for i in range(0, n-1):
7          minpos = i # Positionskandidat
8          # Bestimme den Index des kleinsten Elements
9          # aus A[i], A[i+1], ..., A[n-1]:
10         for j in range(i+1, n):
11             if A[j] < A[minpos]:
12                 minpos = j
13         # Vertausche A[i] mit dem kleinsten Element:
14         A[minpos], A[i] = A[i], A[minpos]
```

Laufzeitanalyse

Vergleiche: $(n-1)+(n-2)+\cdots+1=\dfrac{n(n-1)}{2}$

Vertauschungen: $n-1$

Mit gleichen Kosten $T(1)=c$ für Vergleiche und Vertauschungen erhalten wir:

$T(n)=c\left(\frac{1}{2}n^2-\frac{1}{2}\right)+c(n-1)=c\frac{1}{2}n^2+c\frac{1}{2}n-c\in O(n^2)$

2 Insertion Sort

Beschreibung

Die erste Zahl links ist ein sortiertes Array der Länge 1.

Dieses Array wird um das nächste rechts stehende Element x erweitert. Ist es grösser als sein linker Nachbar, ist dase erweiterte Array sortiert, Andernfalls wird das links stehende Element um eine Position nach rechts verschoben und das Element x wird an die Anfangsposition gesetzt.

Allgemein wird ein bereits sortiertes Array mit k Elementen um das $(k+1)$-te Element x erweitert. Dieses Element wird der Reihe nach mit den davor liegenden Elementen verglichen. Ist eines davon grösser als x, wird es um eine Position nach rechts verschoben. Andernfalls wird x an die freigewordene Stelle gesetzt und ist an seiner richtigen Position. So fahren wir fort, bis das letzte Elemente wie oben beschrieben einsortiert ist.

Beispiel

13	5	2	21	8	Vergleiche	Verschiebungen

Python-Implementierung

```
1  def insertionsort(A):
2      '''Sortiert das Array A aufsteigend/inplace '''
3      # Beginne mit A[1]
4      for i in range(1, len(A)):
5          x = A[i] # das einzusortierende Element
6          j = i-1  # aktuelle Position von 'cand'
7          # So lange die Elemente an den Positionen
8          # j, j-1, ..., 0 grösser als x sind, schiebe
9          # sie nach rechts:
10         while(j >= 0 and A[j] > x):
11             A[j+1] = A[j]
12             j = j-1
13         A[j+1] = x # Fülle die Lücke
```

Laufzeitanalyse (Worst Case)

Gegeben: Array mit n Elementen, das umgekehrt sortiert ist.

Vergleiche: $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$

Verschiebungen: $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$

Mit gleichen Kosten $T(1) = c$ für Vergleiche und Verschiebungen erhalten wir:

$T(n) = c\frac{1}{2}n^2 - c\frac{1}{2}n + c\frac{1}{2}n^2 - c\frac{1}{2}n = cn^2 - cn \in O(n^2)$

Laufzeitanalyse (Best Case)

Gegeben: Array mit n Elementen, das bereits sortiert ist

Vergleiche: $1 + 1 + \cdots + 1 = n - 1$

Verschiebungen: $0 + 0 + \cdots + 0 = 0$

Insgesamt: $T(n) = c \cdot (n-1) \in O(n)$

3 Bubble Sort

Beschreibung

Zuerst vergleicht man die ersten beiden Elemente und bringt sie, falls nötig in die richtige Reihenfolge. Dann vergleicht man das zweite und dritte Element und bringt sie, falls nötig, in die richtige Reihenfolge. Auf diese Weise „treibt“ man durch fortgesetzte Vertauschungen das grösste Element ans Ende des Arrays. Danach beginnt man wieder von vorne und treibt durch Vertauschungen das zweitgrösste Element an die zweitletzte Stelle. Auf diese Weise fährt man fort, bis man am Schluss noch die ersten beiden Elemente in die richtige Reihenfolge bringen muss.

Die Bezeichnung Bubblesort rührt daher, weil die grösseren Elemente, wie Luftblasen im Wasser, ans Ende des Arrays „aufsteigen“.

Beispiel

13	5	2	21	8	Vergleiche	Vertauschungen

Python-Implementation

```
1  def bubblesort(A):
2      '''Sortiert ein Array A aufsteigend/inplace'''
3      n = len(A)
4      # nach dem i-ten Durchlauf steht das grösste
5      # Element von A[0], A[1], ... A[n-i-1]) an
6      # Position n-i-1:
7      for i in range(0, n-1):
8          # durchlaufe j=0, j=1, ..., j=n-i-2 und
9          # tausche das Element A[j] 'nach oben',
10         # falls A[j] > A[j+1]:
11         for j in range(0, n-i-1):
12             if (A[j] > A[j+1]):
13                 (A[j], A[j+1]) = (A[j+1], A[j])
```

Laufzeitanalyse (Worst Case)

Gegeben: Array mit n Elementen, das umgekehrt sortiert ist.

Vergleiche: $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$

Vertauschungen: $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$

Mit gleichen Kosten $T(1) = c$ für Vergleiche und Verschiebungen erhalten wir:

$T(n) = c\frac{1}{2}n^2 - c\frac{1}{2}n + c\frac{1}{2}n^2 - c\frac{1}{2}n = cn^2 - cn \in O(n^2)$

Laufzeitanalyse (Best Case)

Gegeben: Array mit n Elementen, das bereits sortiert ist

Vergleiche: $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$

Vertauschungen: $0 + 0 + \cdots + 0 = 0$

Insgesamt: $T(n) = c\frac{1}{2}n^2 - c\frac{1}{2}n \in O(n^2)$

4 Quicksort

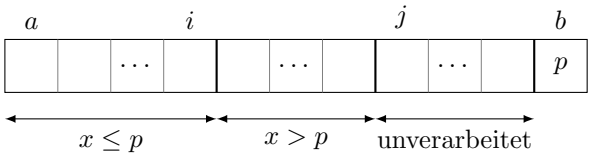
Quicksort wurde von C. Antony R. Hoare zu Beginn der 60er-Jahre entwickelt [The Computer Journal (1962) 5, Seiten 10–15].

Es basiert auf dem Prinzip von *divide and conquer* (*Teile und herrsche*) und sortiert in den meisten Fällen sehr schnell.

Der Partitionierungsschritt

Im zu sortierenden Teilarray wird das letzte Element als Pivot („Scharnier“) gewählt und die davor liegenden Elemente durch die Indizes i, j mit $a \leq i \leq j < b$ in drei Bereiche zerlegt:

- Elemente $A[k]$ mit $a \leq k \leq i$ sind nicht grösser als das Pivot p
- Elemente $A[k]$ mit $i < k < j$ sind grösser als das Pivot
- Elemente $A[k]$ mit $j \leq k < b$ sind unbearbeitet

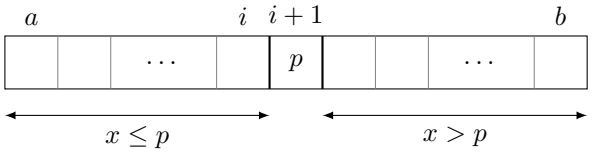


Zu Beginn sind die ersten beiden Bereiche leer ($i = a - 1$, $j = a$).

Wähle für $j = a, \ldots, b - 1$ das Element $x = A[j]$ und vergleiche es mit dem Pivot p:

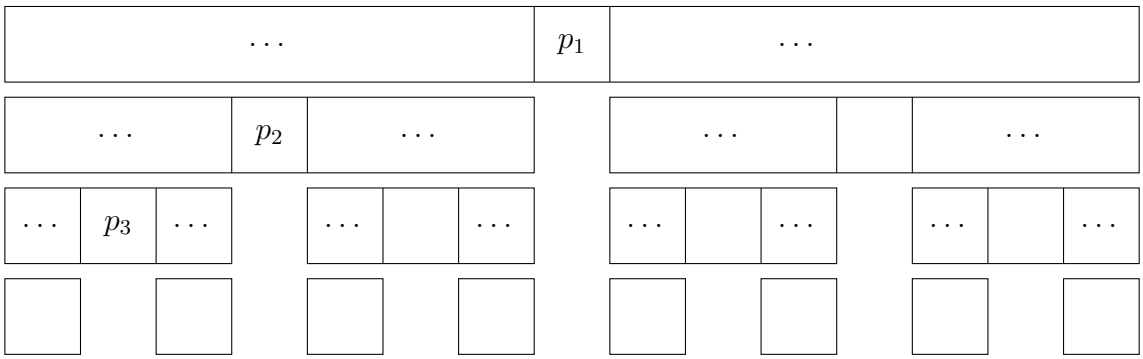
- Ist $x \leq p$, wird x mit dem Element $A[i + 1]$ vertauscht und der Index i um eine Position nach rechts verschoben.
- Ist $x > p$, bleibt das Element an seinem Platz.

Nachdem das letzte Element $A[b-1]$ verarbeitet wurde, wird das Pivot mit dem Element and der Position $i+1$ vertauscht. Somit steht das Pivotelement an der richtigen Position und muss später nicht mehr weiterverarbeitet werden.



Der Rekursionschritt

Wende die Partitionierung rekursiv auf den Teilarrays vor und nach der Position des Pivotelements an. Die Rekursion bricht bei Arrays der Länge 1 ab.



Beispiel

8	9	2	1	5	4

Code

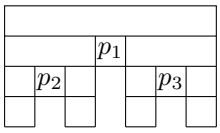
```
def quicksort(A):
    # quicksort_helper(...) ermöglicht es, quicksort(...)
    # nur mit dem Parameter 'A' und ohne Angabe der
    # Grenzen 'a' und 'b' aufzurufen.
    quicksort_helper(A, 0, len(A))

def quicksort_helper(A, a, b):
    if a < b: # solange es noch Elemente zwischen a und b gibt:
        m = partition(A, a, b)
        # Rekursive Anwendung auf unteren/oberen Teil
        quicksort_helper(A, a, m)
        quicksort_helper(A, m+1, b)
```

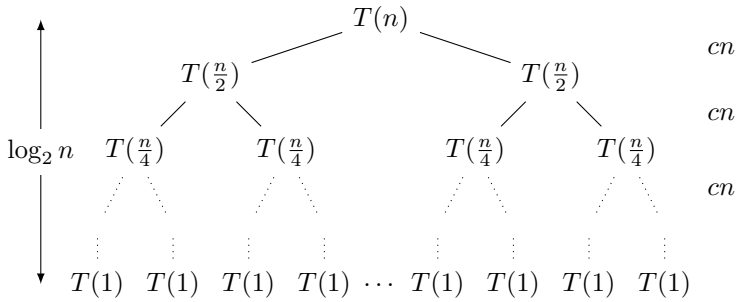
```
14 def partition(A, a, b):
15     pivot = A[b-1]
16     i = a-1
17     # vergleiche alle Elemente mit dem Pivotelement:
18     for j in range(a, b-1):
19         # verschiebe Bereichsgrenze nach oben und
20         # tausche A[j]<=p in den unteren Bereich:
21         if A[j] <= pivot:
22             i += 1
23             A[i], A[j] = A[j], A[i]
24     # tausche Pivot an die richtige Stelle:
25     A[i+1], A[b-1] = A[b-1], A[i+1]
26     # gib die Position m=i+1 des Pivots zurück:
27     return i+1
```

Best Case

Quicksort ist optimal, wenn das Pivotelement das Array in zwei gleich grosse Teile zerlegt.



Wenn die Rekursion auf zwei gleich grosse Teilarrays angewendet werden kann, ergibt sich vereinfacht folgender Rekursionsbaum:

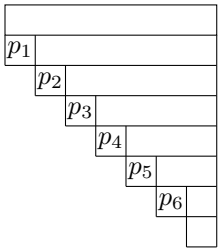


Jede Partitionierung eines Arrays mit n Elementen kann überschlagsmässig in cn Zeiteinheiten bewältigt werden.

Da ein perfekter Binärbaum die Höhe $\log_2 n$ hat, ergibt dies eine Laufzeitkomplexität $T(n) = cn \cdot \log_2 n \in \mathcal{O}(n \log_2 n)$.

Worst Case

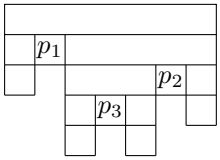
Dieser Fall tritt ein, wenn das Array bereits auf- oder absteigend sortiert ist. Dann ist ein Teilarray leer und das andere besteht aus $n-1$ Elementen.



Quicksort betreibt dann einen ähnlichen Aufwand wie Selectionsort, indem es Arrays der Länge $n-1$, $n-2$, …, 2, 1 in jeweils linearer Zeit verarbeitet und so eine Laufzeitkomplexität von $\mathcal{O}(n^2)$ hat.

Average Case

Der Average Case ist mathematisch schwierig zu behandeln, da die Wahrscheinlichkeitsverteilung der zu sortierenden Daten in der Regel unbekannt ist.



Dennoch lässt sich zeigen, dass Quicksort auch in diesem Fall – selbst wenn einige der Partitionierungen ungünstig sind – die Laufzeitkomplexität $O(n \log n)$ hat.

Der Worst Case kann durch kleine Modifikationen im Algorithmus vermieden werden.

- *Randomized Quicksort:* Wir wählen zufällig einen Index i mit $p \leq i \leq r$ im Teilarray $A[p..r]$ und vertauschen die Elemente $A[r]$ und $A[i]$. Danach führen wir den oben beschriebenen Partionierungsschritt aus. In Pseudocode:

  RANDOMIZED-PARTITION(A, p, r)
  1 $i \leftarrow$ RANDOM(p, r)
  2 vertausche $A[r]$ mit $A[i]$
  3 **return** PARTITION(A,p,r)

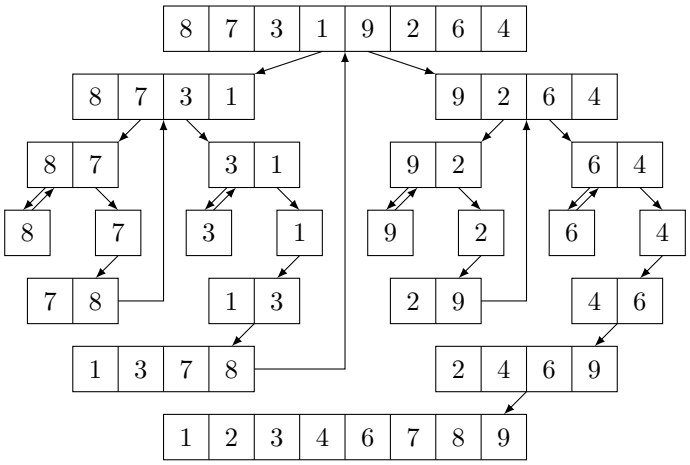
- *Median of three:* Wähle zufällig drei Elemente aus dem Array aus, bestimme ihren Median und vertausche dieses Element mit demjenigen an der Pivotposition am Ende des (Teil-)Arrays.

5 Mergesort

Mergesort ist ein Sortieralgorithmus, der nach dem Prinzip *divide and conquer* arbeitet.

1. *Divide:* Zerlege ein Array mit n Elementen in zwei Arrays mit jeweils $n/2$ Elementen
2. *Conquer:* Wenn das resultierende Array aus weniger als zwei Elementen besteht, dann ist es sortiert. Ansonsten wende Mergesort rekursiv an.
3. *Combine:* Mische die zwei sortierten Teilarrays zu einem sortierten Array zusammen.

Beispiel



Python-Code für die Rekursion

```
1  def mergesort_recursive(A):
2      mergesort_helper(A, 0, len(A))
3
4  def mergesort_helper(A, left, right):
5      if left < right-1:
6          mid = (left + right) // 2
7          mergesort_helper(A, left, mid)
8          mergesort_helper(A, mid, right)
9          merge(A, left, mid, right)
```

Python-Code für die Merge-Operation

```
11 def merge(A, left, mid, right):
12     L = A[left:mid] + [float('inf')]
13     R = A[mid:right] + [float('inf')]
14     i = 0
15     j = 0
16     for k in range(left, right):
17         if L[i] <= R[j]:
18             A[k] = L[i]
19             i = i+1
20         else:
21             A[k] = R[j]
22             j = j+1
```

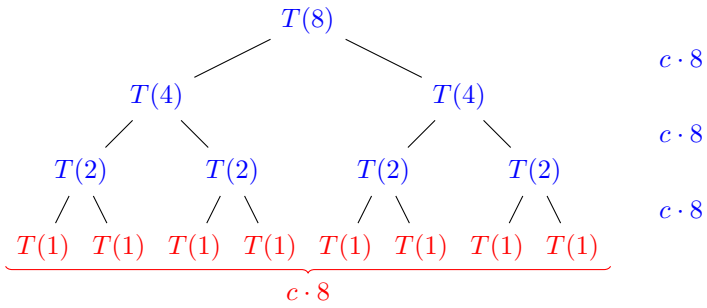
Die Analyse von Mergesort

Geht man bei der Berechnung davon aus, dass die Anzahl der zu sortierenden Elemente eine Zweierpotenz ist, So lässt sich der Berechnungsaufwand durch die *Rekursionsgleichung*

$$T(n) = T(n/2) + T(n/2) + c_1 \cdot 1 + c_2 \cdot n$$

ausdrücken. Die ersten zwei Summanden rechts besagen, dass die Aufgabe in zwei halb so grosse Teile zerlegt wird. Der dritte Summand stellt den Aufwand dar, die Mitte des ursprünglichen Arrays zu bestimmen und der letzte, die sortierten Teillisten wieder zusammenzufügen. Für ein Problem der Grösse 1 berechnen wir am Ende den konstanten Aufwand $T(1) = c_3$.

Zur Vereinfachung ersetzen wir nun alle Konstanten durch $c = \max\{c_1, c_2, c_3\}$ und ignorieren $c \cdot 1$, da $c \cdot 1 + c \cdot n \in O(n)$.



$T(8) = c \cdot 8 \cdot 3 + c \cdot 8$

oder allgemein:

$T(n) = c \cdot n \cdot \log_2 n + c \cdot n \in O(n \log n)$

Im Vergleich zu Quicksort hat Mergesort eine garantierte Laufzeit von $O(n \log_2 n)$.

Jedoch müssen beim Merge-Schritt, die Arrays vorgängig kopiert werden, was bedeutet, dass das Verfahren zusätzlichen Speicher in der Grösse des zu sortierenden Arrays benötigt.

Eine iterative Variante von Mergesort

```
def mergesort_iterative(A):
    n = len(A)
    i = 1
    while i < n:
        j = 0
        while j < n:
            left = j
            mid = j+i
            right = min(j+2*i, n) # len(A) ist nicht immer Zweierpotenz
            merge(A, left, mid, right)
            j = j + 2*i
        i = 2*i
```

Die Merge-Funktion ist dieselbe wie bei der rekursiven Version.

Beispiel

7	3	4	1	8	2	5

6 Counting Sort

Bemerkung

Es lässt sich beweisen, dass jedes Sortierverfahren, welches auf Vergleichen beruht, eine Worst Case-Laufzeitkomplexität von mindestens $O(n \log_2 n)$ haben muss.

Wir werden nun ein Verfahren kennen lernen, das nicht vergleichsbasiert ist und in $O(n)$ sortiert.

Voraussetzungen

Counting Sort setzt voraus, dass die Elemente ganze Zahlen im Bereich von 0 bis m mit $m \in \mathbb{N}$ sind.

Um ein Array A mit n Elementen zu sortieren, wird ein Hilfsarray B mit $m+1$ Elementen benötigt.

Beispiel

Array A:

3	2	3	4	0	0	2	3	3	3	2	3
0	1	2	3	4	5	6	7	8	9	10	11

Array B:

0	1	2	3	4

Array A':

0	1	2	3	4	5	6	7	8	9	10	11

Python-Implementierung

```
1  def countingsort(A):
2      m = max(A)
3
4      # Array mit den Häufigkeiten:
5      B = [0 for i in range(0, m+1)]
6      for k in A:
7          B[k] += 1
8
9      # Indexpositionen von B in A einsortieren:
10     s = 0
11     for i in range(0, len(B)):
12         for j in range(0, B[i]):
13             A[s] = i
14             s += 1
```

Laufzeitanalyse

- Bestimmung des Maximums m von A: cn
- Das Initalisieren der 0-Werte in B: $c(m+1)$.
- Bestimmen der Häufigkeiten der Elemente in A: cn.
- Bestimmen des sortierten Arrays A' aus B: cn.

Daraus ergibt sich für $(m+1) \leq n$:

$T(n) = cn + c(m+1) + cn + cn \leq 4cn \in O(n)$