

Sortierverfahren zusammengefasst

- **Insertion-Sort** (Sortieren durch Einfügen)
- **Selection-Sort** (Sortieren durch Ausschuchen)
- **Bubblesort** (Sortieren durch paarweises Austauschen)
- **Shakersort** (modifiziertes Bubblesort)
- **Shellsort** (benannt nach D.L. Shell)
- **Quicksort** (rekursives Sortieren, mit Pivot – Element)
- **Mergesort** (Sortieren durch Mischen)
- **Natural Mergesort**
- **Bucketsort** (Sortieren nach “Eigenschaften” (Eimern))
- **Fachverteilen**
- **Heapsort**
- Zusammenfassung und Vergleich

Insertion-Sort (Sortieren durch Einfügen)

Die Idee bei Sortieren durch Einfügen ist, dass wenn man ein Feld oder eine Liste ungeordnet vorliegen hat, das Feld sortiert, indem man einen Sortierten Teil schafft. In diesen Teil, werden jetzt immer nach und nach die Elemente aus dem Unsortierten Teil eingefügt, wodurch sich der Sortierte Teil vergrößert, und der unsortierte Teil verkleinert. Dieser Algorithmus wird so lange ausgeführt, bis der unsortierte Teil leer ist.

Bsp:

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

sei zu sortieren

↓ links von dem Pfeil steht der sortierte Teil der Folge

Schritt 0

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

wir nehmen das erste Element in den sortierten Teil auf

Schritt 1

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

wir nehmen die 3 in den sortierten Teil auf, und fügen ihn an der richtigen Stelle ein

Schritt 2

3	23	45	2	6	5	8	2	4	12
---	----	----	---	---	---	---	---	---	----

45 aufnehmen und einsortieren

Schritt 3

3	23	45	2	6	5	8	2	4	12
---	----	----	---	---	---	---	---	---	----

2 aufnehmen und einsortieren

Schritt 4

2	3	23	45	6	5	8	2	4	12
---	---	----	----	---	---	---	---	---	----

6 aufnehmen und einsortieren

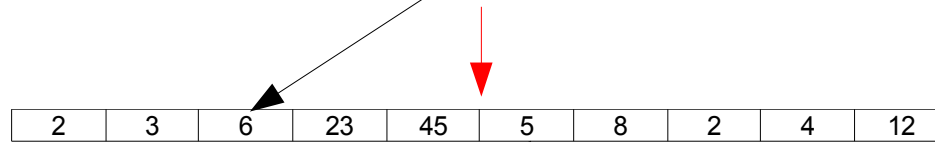
Insertion-Sort (Sortieren durch Einfügen)

Schritt 4



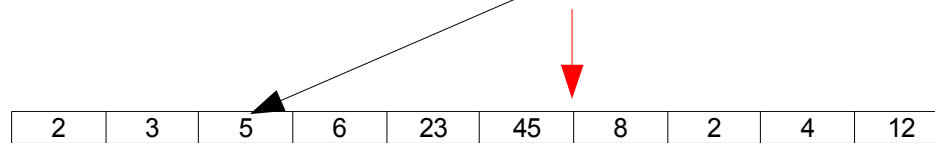
6 aufnehmen und einsortieren

Schritt 5



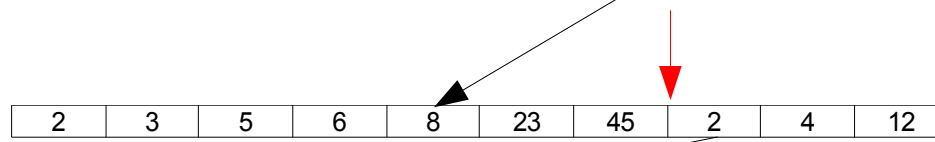
5 aufnehmen und einsortieren

Schritt 6



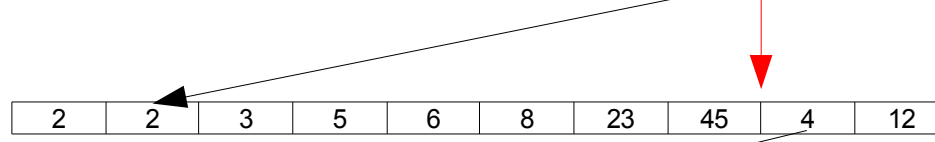
8 aufnehmen und einsortieren

Schritt 7



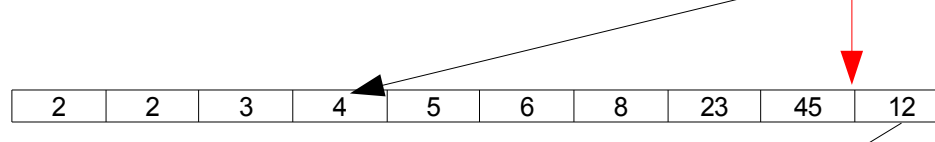
2 aufnehmen und einsortieren

Schritt 8



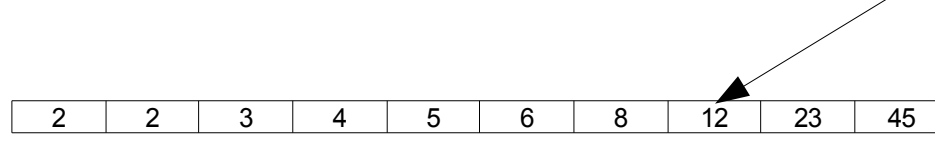
4 aufnehmen und einsortieren

Schritt 9



12 aufnehmen und einsortieren

Schritt 10



fertig sortiert

Stabil? Im allgemeinen kann man davon ausgehen, das Insertion-Sort ein stabiles Sortierverfahren ist, da man es in der Regel so implementieren wird, dass das linkeste Element des unsortierten Teiles, durch Paarweises Vergleichen, an die richtige Stelle im Sortierten Feld verschoben wird.

Komplexität? worst case $O(n^2)$ im Best Case $O(n)$. Im Bestcase is unsere Liste bereits sortiert, es wird immer nur der sortierte Teil vergrößert, also hinten an ihn angefügt durchlauf benötigt

Ordnungsverträglich? ja, denn umso sortierter die Folge, umso weniger Schritte werden für das Einsortieren im Sortierten Teil benötigt.

Selection-Sort (Sortieren durch Aussuchen)

Die Idee bei Sortieren durch Aussuchen ist, dass wenn man ein Feld oder eine Liste ungeordnet vorliegen hat, das Feld sortiert, indem man sich aus dem gegebenen unsortierten Feld immer das kleinste Element holt. Bei diesem Sortieren, muss man in Listen, normal eigentlich nur wenige Zeigerumsetzen, beim Array wird man es so implementieren, das man wieder einen sortierten Teil, und einen unsortierten Teil hat. Der sortierte Teil wird immer durch das neue Element am Ende um eins vergrößert, dafür vertauscht man immer das einzusortierende mit dem Element hinter dem bereits sortierten Teil.

Bsp:

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

sei zu sortieren

↓ links von dem Pfeil steht der sortierte Teil der Folge

Schritt 1

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

kleinstes Suchen; ist 2
einsortieren

Schritt 2

2	3	45	23	6	5	8	2	4	12
---	---	----	----	---	---	---	---	---	----

kleinstes Suchen; ist 2
einsortieren

Schritt 3

2	2	45	23	6	5	8	3	4	12
---	---	----	----	---	---	---	---	---	----

kleinstes Suchen; ist 3
einsortieren

Schritt 4

2	2	3	23	6	5	8	45	4	12
---	---	---	----	---	---	---	----	---	----

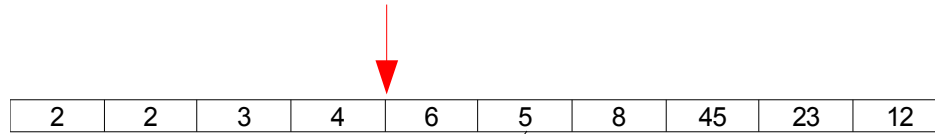
kleinstes Suchen; ist 4
einsortieren

Schritt 5

2	2	3	4	6	5	8	45	23	12
---	---	---	---	---	---	---	----	----	----

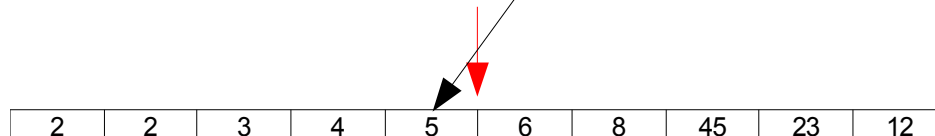
kleinstes Suchen; ist 5
einsortieren

Schritt 4



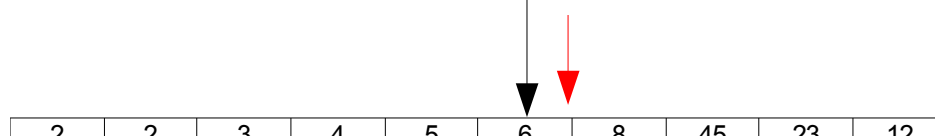
kleinstes Suchen; ist 5
einsortieren

Schritt 5



kleinstes Suchen; ist 6
einsortieren

Schritt 6



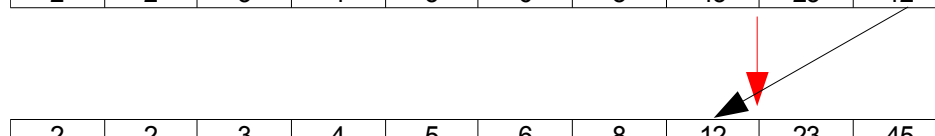
kleinstes Suchen; ist 8
einsortieren

Schritt 7



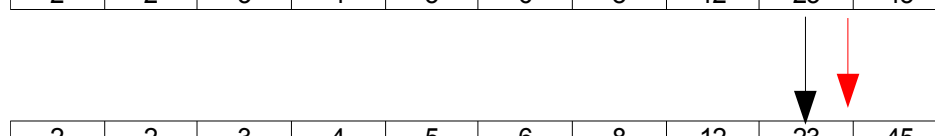
kleinstes Suchen; ist 12
einsortieren

Schritt 8



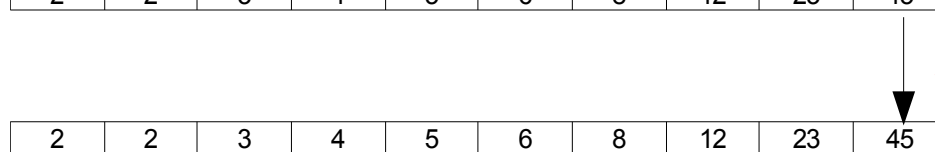
kleinstes Suchen; ist 23
einsortieren

Schritt 9



kleinstes Suchen; ist 45
einsortieren

Schritt 10



fertig sortiert

Stabil? Bei Selection-Sort ist dies sehr von der gewählten Datenstruktur und von der Implementierung abhängig. Bei zu Sortierenden Listen sollte es im allgemeinen stabil sein, da da nur wenige Zeige umgesetzt werden, und die unsortierte Liste halt kleiner wird, und die Sortierte größer, die Reihenfolge aber nicht verändert wird.

Bei Arrays ist dies sehr von der Implementierung abhängig, so wie oben beschrieben, ist Selectionsort nicht stabil, da durch das Vertauschen der Positionen im Array, aus ohne das man es will gleiche in der Reihenfolge vertauscht. Den Fehler könnte manschon so umgehen, indem man immer das Gesamte noch unsortierte Feld verschiebt, das vor der einzusortierenden Zahl steht. Dann wär es wieder stabil. oder man arbeitet auf einem Zweiten Array, dann wär es auch wieder stabil.

Komplexität? worst case $O(n^2)$ im Best Case $O(n)$.

Wir durchlaufen immer das Komplette unsortierte Feld, um das Kleinste zu finden. Und das machen wir für jedes Element. Also erhalten wir eine Komplexitätsgleichung, die lautet:

$n * (n / 2) \rightarrow n^2 / 2$. Daher ist die Zeitkomplexität in $O(n^2)$;

Wenn man dieses Sortieren mit Arrays und Stabil wie oben beschrieben implementiert, so erhält man sogar $O(n^3)$.

Ordnungsverträglich? nein, denn unabhängig davon, ob das Feld bereits sortiert ist, oder nicht, wird der Algorithmus immer das gesamte Feld durchsuchen, um das Kleinste Element zu ermitteln. es bringt also normalerweise nicht einmal eine minimale Laufzeitverbesserung, wenn ein sortiertes Feld vorliegt.

Bubble-Sort (Sortieren durch paarweises Vergleichen)

Die Idee von Bubblesort ist die, das man immer zwei nebeneinanderstehende Elemente vergleicht, und sollte das Rechte kleiner als das Linke sein, so vertausche man diese beiden Elemente. Mit diesem Vorgang wird das ganze Feld so oft von vorn nach hinten durchlaufen, bis das Feld in sortiertem Zustand vorliegt. Auch hier kann einen Sortierten Teil einführen, hierdurch wird eine Effizienzsteigerung erreicht.

Bsp:

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

sei zu sortieren

rechts von dem Pfeil steht der sortierte Teil der Folge

Felddurchlauf 1

3	23	2	6	5	8	2	4	12	45
---	----	---	---	---	---	---	---	----	----

Felddurchlauf 2

3	2	6	5	8	2	4	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 3

2	3	5	6	2	4	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 4

2	3	5	2	4	6	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 5

2	3	2	4	5	6	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 5

2	3	2	4	5	6	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 6

2	2	3	4	5	6	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 7

2	2	3	4	5	6	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Unser intelligent programmierter Algorithmus hat natürlich bemerkt, dass er im letzten durchlauf keine Vertauschung mehr durchgeführt hat, da das Feld sortiert ist. Der Algorithmus stoppt also.

Stabil? Bubblesort ist normalerweise stabil, da man beim Vergleichen benachbarter Elemente es normalerweise so implementiert, dass gleiche Objekte nicht vertauscht werden.

Komplexität? worst case $O(n^2)$ im Best Case $O(n)$.

Wenn das Feld in umgekehrter Reihenfolge vorliegt, so muss jede Zahl durch das Feld hindurch getragen werden.

Und das für jedes Element, man erhält $O(n^2)$.

Im Best Case ist das Feld sortiert, und liegt in der richtigen Reihenfolge vor. Dann wird das Feld nur einmal durchlaufen.
Ordnungsverträglich? ja, denn abhängig davon, ob das Feld bereits sortiert ist, oder nicht, wird der Algorithmus eine geringere oder höhere Laufzeit aufweisen. umso sortierter das Feld, umso geringer die Laufzeit.

Shaker-Sort

Die Idee von Shakersort ist die Laufzeit von Bubblesort dadurch zu verbessern, dass man das Feld immer abwechselnd von vorn nach hinten, und von hinten nachvorn durchläuft. Jedoch bringt diese Veränderung keine Laufzeitverbesserung. Bei der Implementierung ändert sich nun, das wir noch einen zweiten Sortierten Teil erhalten. Da einmal das kleinste Element nach vorn und dann das größte Element nach hinten transportiert werden.

Bsp:

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

sei zu sortieren

in der Mitte steht der unsortierte Teil

3	23	2	6	5	8	2	4	12	45
---	----	---	---	---	---	---	---	----	----

Felddurchlauf 1

2	3	2	6	5	8	4	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 2

2	2	3	5	6	4	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 3

2	2	3	4	5	6	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 4

2	2	3	4	5	6	8	12	23	45
---	---	---	---	---	---	---	----	----	----

Felddurchlauf 5

Unser intelligent programmierter Algorithmus hat natürlich bemerkt, das er im letzten durchlauf keine Vertauschung mehr durchgeführt hat, da das Feld sortiert ist. Der Algorithmus stoppt also.

Auch wenn es hier der Fall ist, im allgemeinen führt Shakersort nicht zu einer Laufzeitverbesserung von Bubblesort.

Für **Komplexität**, **Ordnungsverträglichkeit**, und **Stabilität** gilt das gleiche wie bei Bubblesort.

Shell-Sort (benannt nach D.L. Shell)

Die Idee des Verfahrens ist, die Daten als zweidimensionales Feld zu arrangieren und spaltenweise zu sortieren. Dadurch wird eine Grobsortierung bewirkt. Dann werden die Daten als schmaleres zweidimensionales Feld angeordnet und wiederum spaltenweise sortiert. Dann wird das Feld wiederum schmaler gemacht usw. Zum Schluss besteht das Feld nur noch aus einer Spalte.

Werden die Feldbreiten geschickt gewählt, reichen jedesmal wenige Sortierschritte aus, um die Daten spaltenweise zu sortieren bzw. am Ende, wenn nur noch eine Spalte vorhanden ist, vollständig zu sortieren. Die Effizienz von Shellsort hängt von der Folge der gewählten Feldbreiten ab.

Bsp:

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

sei zu sortieren

Schritt 1

23	3	45	2
6	5	8	2
4	12		



4	3	8	2
6	5	45	2
23	12		

Schritt 2

4	3
8	2
6	5
45	2
23	12



4	2
6	2
8	3
23	5
45	12

Schritt 3

4
2
6
2
8
3
23
5
45
12



2
2
3
4
5
6
8
12
23
45

fertig

Als erstes wird das Feld als Feld mit Vier Spalten umformatiert, in diesem werden jetzt die Spalten sortiert.

Das Feld erhaltene Feld wird umformatiert, in eines mit 2 Spalten. Diese werden jetzt sortiert.

Das Feld erhaltene Feld wird umformatiert, in eines mit einer Spalte. Diese wird jetzt sortiert.

Implementierung:

Tatsächlich werden die Daten nicht in unterschiedlich breite Felder umarrangiert, sondern die Daten stehen in einem eindimensionalen Feld, das entsprechend indiziert wird. Beispielsweise bilden die Elemente an den Indexpositionen 0, 5, 10, 15 usw. die erste Spalte eines 5-spaltigen Feldes. Die durch die jeweilige Indizierung entstehenden "Spalten" werden im Shellsort-Algorithmus mittels Insertion Sort (Sortieren durch Einfügen) sortiert, da Insertion Sort bei teilweise vorsortierten Daten recht effizient ist.

Stabil? Nein, da durch die Spaltenweise Betrachtung, die Reihenfolge nicht mehr gesichert ist.

Komplexität? Nach dem Beweis von Papernov und Stasevic liegt die Komplexität in $O(n \cdot \sqrt{n})$

Ordnungsverträglich? Da dies eine spezielle Abart von Insertionsort darstellt, ist auch Shellsort ordnungsverträglich.

Quick-Sort (rekursives Sortieren mit Pivot-Element)

Die Idee von Quicksort ist, dass man ein Element der Folge bestimmt, und alle Elemente die kleiner sind, in der Liste vor dieses Element (das Pivot – Element) bringt, und alle die Größer sind hinter das Pivot-Element. Hierdurch wissen wir, dass dieses Pivot-Element nun schon sortiert ist, und an der richtigen Stelle steht. Aber wir haben weiterhin zwei Teilfelder, die als unsortiert angesehen werden müssen. Deshalb führt man jetzt rekursiv diesen Vorgang für die beiden entstandenen Teilfelder weiter durch. Bis wir das gesamte Feld sortiert haben.

Bsp:

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

sei zu sortieren

Schritt 1

3	23	2	6	5	8	2	4	12	45
---	----	---	---	---	---	---	---	----	----

wählen des Pivot

Schritt 1 - 2

3	2	4	2	5	8	6	12	23	45
---	---	---	---	---	---	---	----	----	----

“sortieren” + 5 steht richtig

Schritt 2

3	2	4	2	5	8	6	12	23	45
---	---	---	---	---	---	---	----	----	----

wählen neuer Pivot-Elemente

Schritt 2 - 2

2	2	4	3	5	8	6	12	23	45
---	---	---	---	---	---	---	----	----	----

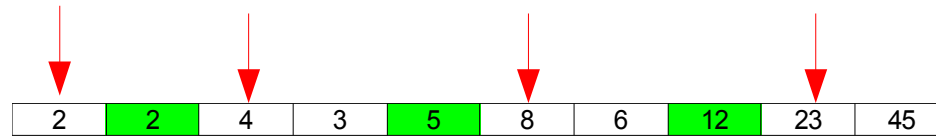
“sortieren” + 2 & 12 stehn richtig

Schritt 3

2	2	4	3	5	8	6	12	23	45
---	---	---	---	---	---	---	----	----	----

wählen neuer Pivot-Elemente

Schritt 3



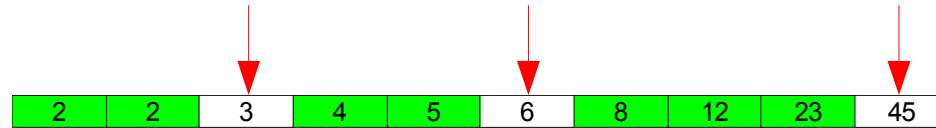
wählen neuer Pivot-Elemente

Schritt 3 - 2



“sortieren” + 2,4,8,23 stehn richtig

Schritt 4



wählen neuer Pivot-Elemente

Schritt 4 - 2



“sortieren” + 3, 6, 45 stehn richtig

Feld ist sortiert

Implementiert, arbeitet man logischerweise die Rekursionen nacheinander ab, und nicht gleichzeitig.

Stabil? Nö.**Komplexität?**

worst case: rund $\frac{1}{2} * n^2$ Vergleiche, wenn das Pivot-Element stets das kleinste oder das größte Element des Teilfelds ist. ($O(n^2)$);

best case: $n \log(n)$, wenn das Pivot-Element stets das Mittelste der Elemente des Teilfelds ist. ($O(n \log n)$)

average case: Es ist mit $1.3863 * n * \log(n) - 1.8456 * n$ Vergleichen im Mittel zu rechnen

Ordnungsverträglich? Alles hängt vom Pivot ab, Quicksort ist also ziemlich Scheißegal, ob das Feld bereits sortiert ist, die Komplexität wird dadurch nicht besser.

Merge-Sort (Sortieren durch Mischen)

Die Idee von Mergesort ist ähnlich der von Quicksort ebenfalls ein Divide & Conquer Algorithmus. Mergesort besteht im Wesentlichen aus zwei Teilen, im ersten Teil, wird das Feld zerlegt und dann wieder verschmolzen. Hierbei geht man wie bei Quicksort rekursiv vor. Wir teilen das Feld in Zwei Teilfelder. Diese sind jetzt zu sortieren. Also Teilen wir die Teilfelder wieder in Zwei zu sortierende Teilfelder usw. Die Rekursion endet, wenn die Teilfolge nur noch ein Element enthält.

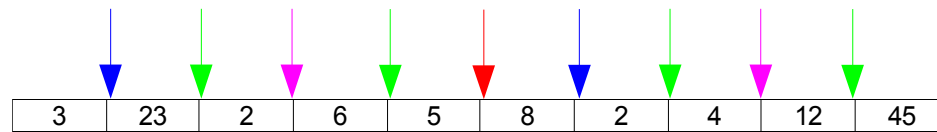
Danach werden die sortierten Hälften verschmolzen (merge). Hierbei werden zunächst die beiden Hälften hintereinander kopiert. Dann werden die beiden Hälften mit einem Index i und einem Index j Elementweise verglichen, und jeweils das nächstgrößte Element in das Originalfeld zurückkopiert.

Bsp:

23	3	45	2	6	5	8	2	4	12
----	---	----	---	---	---	---	---	---	----

sei zu sortieren

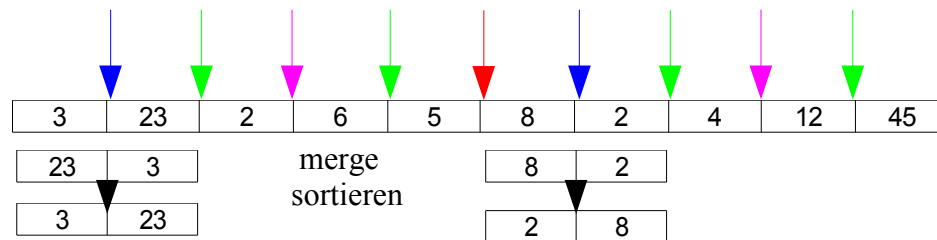
Schritt 1



Teilen des Feldes

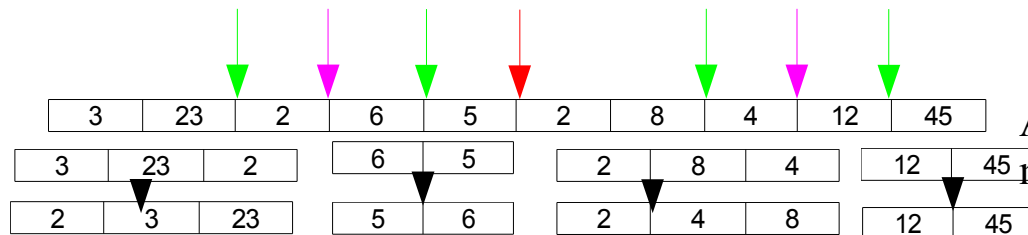
Schritt 1 = rot ; Schritt 2 = pink ;
Schritt 3 = grün ; Schritt 4 = blau

Schritt 2 - 1



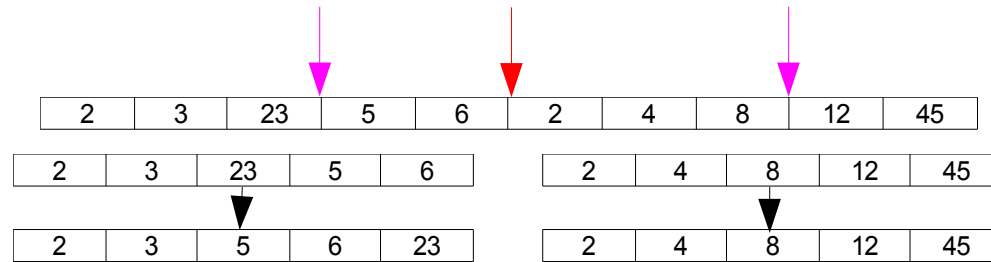
Auflösen der 4. Rekursion >
merge
blau beginnt

Schritt 2 - 2



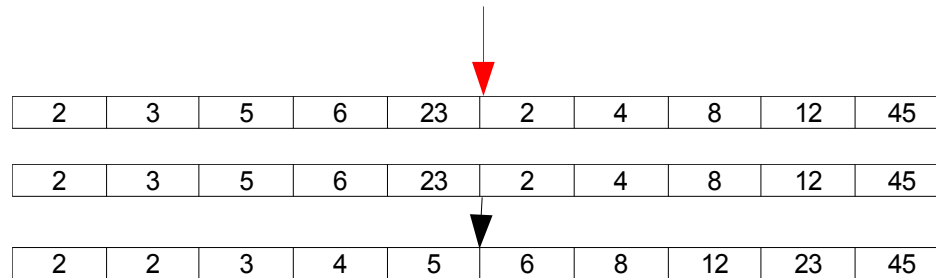
Auflösen der 3. Rekursion >
merge
(grün)

Schritt 2 - 3



Auflösen der 2. Rekursion >
merge
(pink)

Schritt 2 - 4



Auflösen der 1. Rekursion >
merge
(rot)

Komplexität $T(n) = 2n + 2 T(n/2)$ und $T(1) = 0$.
Die Auflösung dieser Rekursionsgleichung ergibt
 $T(n) = 2n \log(n)$ $O(n \log(n))$.

Das Verfahren ist somit optimal, da die untere Schranke für das Sortierproblem von $\Omega(n \log(n))$ erreicht wird. Mergesort benötigt unabhängig davon, ob die Eingabedaten sortiert, umgekehrt sortiert oder zufällig vorliegen, immer genau gleichviele Schritte.

Eine Variante von Mergesort, die bestehende Vorsortierungen in den Eingabedaten ausnutzt, ist Natural Mergesort.

Stabil? Ja. Nützlich ist, dass Mergesort einen zusätzlichen Speicherplatzbedarf von $\Theta(n)$ für das temporäre Array b hat.

Ordnungsverträglich? Nein.

Natural Merge-Sort

Natural Mergesort ist eine Variante von Mergesort. Der Grundgedanke besteht darin, in der zu sortierenden Folge "natürlich" vorkommende, bereits sortierte Teilstücke auszunutzen.

Hierbei wird das Feld anhand dieser Sortierung nun in bereits sortierte Stücke zerlegt, hierbei wird ein natürlich sortiertes Stück als bitonischer Lauf bezeichnet, wenn es entweder monoton wächst, oder monoton fällt, oder eine Zeitlang monoton wächst und dann eine Zeit lang monoton fällt. Unsere Folge wird also in bitonische Folgen zerlegt.

Bsp: fällt aus

Komplexität Bei jedem Durchlauf wird die Anzahl der bitonischen Läufe halbiert. Sind zu Anfang r natürliche Läufe vorhanden, so sind damit $\Theta(\log r)$ Aufrufe erforderlich, bis nur noch ein Lauf übrig ist. Da dennoch für jedes Objekt Vergleiche notwendig sind, dies also in Zeit $\Theta(n)$ läuft, hat Natural Mergesort eine Zeitkomplexität von $\Theta(n \log r)$.

Der schlechteste Fall tritt ein, wenn alle natürlich vorkommenden bitonischen Läufe die Länge 2 haben, z.B. wie in der Folge 1 0 1 0 1 0 1 0. Dann sind $r = n/2$ Läufe vorhanden, und Natural Mergesort benötigt genausoviel Zeit wie Mergesort, nämlich $\Theta(n \log(n))$.

Der günstigste Fall tritt ein, wenn die Folge nur aus konstant vielen bitonischen Läufen besteht. Dann ist nur $\Theta(n)$ Zeit erforderlich.

Insbesondere ist dies der Fall, wenn die Folge bereits aufsteigend oder absteigend sortiert ist, wenn sie aus zwei sortierten Teilstücken besteht, oder wenn konstant viele Elemente in eine sortierte Folge einsortiert werden sollen.

Nachteilig ist, dass Natural Mergesort ebenso wie Mergesort einen zusätzlichen Speicherplatzbedarf von $\Theta(n)$ für das temporäre Array b hat.

Ordnungsverträglich? Ja.

Bsp:

3	23	45	2	6	8	5	2	4	12
---	----	----	---	---	---	---	---	---	----

sei zu sortieren

Teilen des Feldes an hand der bitonischen Läufe

Schritt 1 = rot ; Schritt 2 = pink ;
Schritt 3 = grün ; Schritt 4 = blau

Auflösen der 4. Rekursion >
merge
blau beginnt

Auflösen der 3. Rekursion >
merge
(grün)

Schritt 1

3	23	2	6	5	8	2	4	12	45
---	----	---	---	---	---	---	---	----	----



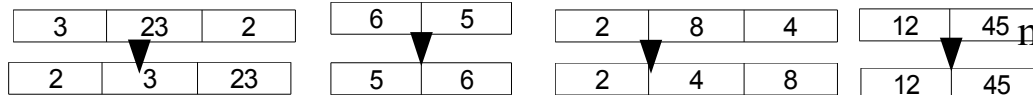
Schritt 2 - 1

3	23	2	6	5	8	2	4	12	45
---	----	---	---	---	---	---	---	----	----



Schritt 2 - 2

3	23	2	6	5	2	8	4	12	45
---	----	---	---	---	---	---	---	----	----



Bucket-Sort

BucketSort zählt die Häufigkeit jedes Schlüsselwertes in einer Liste. Daraus errechnet es die korrekte Position jedes Elements und fügt es in einer zweiten Liste dort ein.

Die Häufigkeit der Schlüsselwerte wird in einem so genannten Histogramm gespeichert. Dies wird meist als Array implementiert, das so lang ist, wie es mögliche Schlüsselwerte gibt; als Indizes werden dabei die Schlüsselwerte bzw. die ihnen zugeordneten ganzen Zahlen gewählt. Elemente mit gleichem Sortierschlüssel werden dabei in Gruppen, so genannten Buckets, zusammengefasst.

Das Histogramm wird zunächst mit Nullen initialisiert. Dann wird die zu sortierende Liste durchlaufen und bei jedem Listenelement der entsprechende Histogrammeintrag um eins erhöht.

In einem zweiten Array, das ebenso lang ist wie das Histogramm-Array und ebenfalls mit Nullen initialisiert wird, werden nun die aus dem Histogramm errechneten Einfügepositionen gespeichert.

Schließlich werden in eine Liste, die ebenso lang ist wie die zu sortierende, die Elemente der zu sortierenden Liste nacheinander an den berechneten Positionen eingefügt.

Komplexität Die asymptotische Laufzeit von BucketSort ist abhängig von der Länge n der zu sortierenden Liste (im Beispiel: $n=15$) und der Anzahl k der Werte, die von den Sortierschlüsseln angenommen werden können (im Beispiel: $k=16$).

Für die Histogramm-Erstellung wird die zu sortierende Liste einmal durchlaufen. Die Komplexität wächst also linear mit der Länge der Liste. Die Aufwandklasse der Histogramm-Erstellung ist also $O(n)$.

Um die Einfügepositionen zu berechnen, wird das Histogramm einmal durchlaufen. Da das Histogramm-Array so lang ist, wie es mögliche Sortierschlüssel-Werte gibt, ist die Aufwandklasse für die Erstellung des Einfügestellen-Arrays $O(k)$.

Anschließend wird die zu sortierende Liste erneut durchlaufen, um die Elemente im Zielarray zu speichern. Die Aufwandklasse ist erneut $O(n)$.

Insgesamt ergibt sich für BucketSort also eine Komplexität von $O(n+k)$.

Die worst case Laufzeit und die best case Laufzeit des Algorithmus unterscheiden sich nicht voneinander, da der BucketSort für alle gleichgroßen Eingabegrößen n immer gleich lange zur Berechnung braucht. Hierdurch ergibt sich, dass der BucketSort in $\Theta(n)$ liegt

Stabil? Ja.

Ordnungsverträglich? Nein.

Sortieren durch Fachverteilen

Sind die Schlüssel Wörter über einem endlichen Alphabet $A = \{a_1, a_2, \dots, a_s\}$, kann man die zu sortierenden Elemente zunächst bzgl. des letzten Zeichens an s Listen anfügen.

Diese Listen hängt man aneinander und fügt nun alle Elemente bzgl. des vorletzten Zeichens an s Listen an usw. Liegt die Länge jedes Schlüssels in der Größenordnung von $\log(n)$, dann ergibt sich ein $O(n \cdot \log(n))$ -Sortierverfahren.

Das Anhängen an die s Listen entspricht dem Ablegen in s verschiedene Fächer, weshalb dieses Verfahren als "Fachverteilen" bezeichnet wird.

Komplexität wohl n^2 ?

Stabil? Ja.

Ordnungsverträglich? Nein.

Heapsort

Das Sortierverfahren Heapsort hat eine Zeitkomplexität von $\Theta(n \log(n))$. Eine untere Schranke für die Zeitkomplexität von Sortierverfahren ist $\Omega(n \log(n))$. Heapsort ist daher optimal, d.h. es gibt kein asymptotisch schnelleres Sortierverfahren.

Heapsort verwendet eine besondere Datenstruktur, die als Heap bezeichnet wird.

Diese Datenstruktur basiert auf der Definition eines vollständigen binären Baums. Ein binärer Baum ist vollständig, wenn alle Schichten außer möglicherweise der letzten vollständig besetzt sind.

Desweiteren ist ein Heap ein Heap, wenn er die Heapeigenschaft erfüllt. Also wenn für jeden Knoten gilt, das er in einer Sortierung vorliegt. Das heißt, in einem Absteigendem Heap, gilt für jeden Knoten im Baum, das die Sohnknoten kleiner oder gleich dem Vaterknoten sind. Im Aufsteigenden Heap gilt somit logischerweise das Gegenteil, also das für jeden Sohnknoten eines Knotens gilt, dass diese größergleich dem Vaterknoten sind.

Zusammenfassung:

Mit der Zeitkomplexität von $O(n \log(n))$ im schlechtesten Fall ist Heapsort optimal. Im Gegensatz zu Mergesort benötigt Heapsort keinen zusätzlichen Speicherplatz.

Gegenüber Quicksort ist Heapsort im Durchschnitt langsamer. Die Variante Bottom-Up-Heapsort kommt jedoch der Laufzeit von Quicksort sehr nahe.

Die Datenstruktur des Heaps wird auch zur effizienten Implementation einer Prioritätenliste (priority queue) benutzt.

Komplexität

Ein vollständiger binärer Baum mit n Knoten hat die Tiefe $d \leq \log(n)$. Die Prozedur downheap benötigt daher höchstens $\log(n)$ Schritte.

Eine grobe Analyse ergibt, dass buildheap für höchstens n Knoten downheap aufruft. Heapsort ruft zusätzlich noch einmal für alle n Knoten downheap auf. Somit liegt die Zeitkomplexität von Heapsort in $O(n \cdot \log(n))$.

Eine genauere Analyse zeigt, dass buildheap sogar nur $O(n)$ Schritte benötigt. Denn im Verlauf der bottom-up-Konstruktion des Heaps wird downheap für höchstens $n/2$ Bäume der Tiefe 1, für höchstens $n/4$ Bäume der Tiefe 2 usw. und schließlich für einen Baum der Tiefe $\log(n)$ aufgerufen.

Somit benötigt buildheap höchstens

$$S = n/2 \cdot 1 + n/4 \cdot 2 + n/8 \cdot 3 + \dots + 2 \cdot (\log(n)-1) + 1 \cdot \log(n)$$

Schritte. Da

$$2S = n \cdot 1 + n/2 \cdot 2 + n/4 \cdot 3 + n/8 \cdot 4 + \dots + 2 \cdot \log(n),$$

ergibt sich durch Subtraktion $2S - S$:

$$S = n + n/2 + n/4 + n/8 + \dots + 2 - \log(n) \leq 2n \in O(n).$$

Insgesamt beträgt die Zeitkomplexität von Heapsort allerdings trotzdem noch $T(n) \in O(n \log(n))$.

Asymptotisch geht es nicht schneller, denn die untere Schranke für das Sortieren liegt bei $\Omega(n \log(n))$.

Heapsort ist damit optimal, da es die untere Schranke erreicht (s. Abschnitt).

Stabil? Ja.

Ordnungsverträglich? Nein.