Sortieren

1 Dumme Verfahren

1.1 Selection sort

```
Vergleiche: O( n^2 )          bestcase: O( n^2 )
Bewegungen: O( n )            bestcase: 0
in situ:    ja
stabil:     ja
```

Selection-Sort wählt aufsteigend jeweils das *i*-te Element der sortierten Folge aus, und tauscht es mit dem Element an Position *i*.

1.2 Insertion sort

```
Vergleiche: O( n^2 )          bestcase: O( n )
Bewegungen: O( n^2 )          bestcase: 0
in situ:    ja
stabil:     möglich
```

Per Induktion wird von links her angefangen, wobei der linke Teil der Folge bereits sortiert ist. Jedes zusätzliche Element das beim Erweitern des linken Teils hinzukommt, wird dabei an die richtige Stelle gerückt. Dabei müssen möglicherweise bis zu O(n) Elemente um platziert werden, nämlich wenn das neue Element ganz nach links muss.

1.3 Bubble sort

```
Vergleiche: O( n^2 )          bestcase: O( n )
Bewegungen: O( n^2 )          bestcase: 0
in situ:    ja
stabil:     ja
```

Die unsortierte Liste wird durchlaufen, wobei alle benachbarten Elemente, die in der falschen Reihenfolge vorliegen, vertauscht werden. Das wird so lange wiederholt, bis keine Vertauschungen mehr nötig sind.

2 Verfahren mit Schlüsselvergleichen

2.1 Mergesort

```
Vergleiche: O( n log(n) )     bestcase: O( n log(n) )
Bewegungen: O( n log(n) )     bestcase: O( n log(n) )
in situ:    nein, O(n) Speicher (in situ möglich, aber komplex)
stabil:     ja
```

Mergesort verwendet die Idee des divide & conquer: Die zu sortierende Liste wird in zwei Teile aufgetrennt, wobei jede einzeln sortiert wird. Danach werden die beiden sortierten Listen in einer Schleife in linearer Zeit zusammengeführt. Die Laufzeit ist garantiert O(n logn), aber es wird linear viel zusätzlicher Speicher benötigt.

2.1.1 Straight mergesort

Es wird dasselbe Verfahren wie bei normalem Mergesort verwendet, jedoch nicht rekursiv, sondern induktiv. Dadurch wird der Algorithmus in der Praxis etwas schneller, asymptotisch ändert sich jedoch nichts.

2.1.2 Natürliches 2-Wege Mergesort

Mergesort provitiert nicht wirklich von bereits vorsortierten Teilen in der Eingabe, das will dieses Verfahren ändern. Es wird wieder induktiv vorgegangen, wobei nicht etwa mit Blöcken fester Grösse begonnen wird, sondern es werden sogenannte Runs betrachtet: Ist eine Teilfolge bereits vorsortiert, wird diese Run genannt. Der Algorithmus geht jetzt durch die Liste, und verschmilzt alle benachbarten Runs, wobei die Runs bei jedem Durchgang halbiert werden. Da zu Beginn zwischen 0 und n Runs vorliegen, kann dies in der Praxis eine deutliche Performancesteigerung darstellen mit einer Laufzeit von O(n log(r)).

2.2 Quicksort

```
Vergleiche: O( n^2 )      best- und average-case: O( n log(n) )
in situ:    ja, aber O(n) Speicher für Rekursion (optimiert O(log n) )
stabil:     nein
```

Es wird ein Pivotelement p ausgewählt, und die Liste wird in linearer Zeit folgendermassen umstrukturiert: Alle Elemente die kleiner sind als p kommen nach links, dann kommt p und rechts davon alle anderen Elemente. Dann wird rekursiv wieder gleich vorgegangen, bis die zu sortierende Teilliste nur noch ein Element enthält.

Die Laufzeit hängt nun stark von der Wahl von p ab, kann aber im schlechtesten Fall quadratisch werden.

2.2.1 Randomisiertes Quicksort

Das Pivotelement wird zufällig ausgewählt, was zu einer erwarteten Laufzeit von O(n logn) führt.

2.2.2 Median of three

Der Algorithmus betrachtet das erste, das letze und ungefähr das mittlere Element (von der Position her), und wählt das in der sortieren Reihenfolge dieser drei in der Mitte liegende. Das kommt in der Praxis fast an randomisiertes Quicksort, und benötigt keinen Zufallsgenerator.

2.3 Heapsort

```
Vergleiche: immer O( n log n )
in situ:    ja
stabil:     nein
```

Heapsort verwendet zum Sortieren einen Maximums-Heap: Das heisst, ein Baum, in dessen Wurzel das grösste Element der Liste gespeichert ist, und bei dem für jeden inneren Knoten gilt, dass die beiden Kinder kleiner oder höchstens gleich gross sind.

Zu Beginn muss natürlich erst ein solcher Heap aufgebaut werden, was sich in linearer Zeit erledigen lässt: Der Baum wird einfach mit den Elementen der Eingabe gefüllt und dann wird von unten her begonnen: Die Teilbäume auf der untersten Ebene sind bereits Heaps (da sie nur ein Element haben),

und dann kann für jeden inneren Knoten die Prozedur *versickern* aufgerufen werden, damit die Heapbedingung überall erfüllt ist.

Danach beginnt das eigentliche Sortieren: Die Wurzel wird entfernt und mit dem letzen Element des Heaps ausgetauscht. Dann wird die Wurzel versickert und das ganze beginnt von vorne.

Der Heap kann dabei immer implizit gespeichert werden, also direkt im Array. Die Kinder des Elementes an Position *i* sind dann bei *2i* und *2i+1* zu finden.

3 Weitere Verfahren

3.1 Radix-Exchange-Sort

```
Laufzeit:   O( l * n ) = O( n log n )
in situ:    ja
stabil:     nein (? nicht ganz sicher)
```

Dieses Verfahren macht sich zunutze, dass die Schlüssel meistens in binärform Vorliegen: Seien die Schlüssel also alle Element des Alphabetes {0,1} mit zwei Elementen, und sei l weiter die Anzahl Zeichen im längsten Schlüssel. Radix-Exchange-Sort geht nun folgendermassen vor: Zuerst wird die erste, also höchstwertige Position im Schlüssel betrachtet. Alle Elemente die dort eine 0 haben, kommen nach links, alle anderen nach rechts (wie bei Quicksort, jedoch ohne Pivot). Dann wird Rekursiv auf beiden Teilen weitergemacht, mit dem nächsten Bit.

Sind alle Schlüssel verschieden, so ist $l \geq \log(n)$, was dieses Verfahren nur Effizient macht, wenn es nicht nur wenige, sehr lange Schlüssel gibt.

3.2 LSD-Radixsort

```
Laufzeit:   O( l * n )
in situ:    nein
stabil:     ja
```

Die Anzahl Zeichen im Alphabet sei nun m, und beliebig. Das Sortieren geschieht nun durch „Fachverteilung“, wobei es mehrere Durchgänge gibt, die alle aus einer „Verteil-„ und „Sammelphase“ bestehen. Es wird wieder jeweils ein Bit betrachtet, jedoch wird beim niederwertigsten (least-significant-digit, LSD) gestartet.

Beim Verteilen werden die Elemente gemäss dem gerade betrachteten Bit in m Fächer verteilt, wobei die ursprüngliche Ordnung erhalten bleiben muss. Dann wird alles wieder eingesammelt, wobei FIFO-mässig vorgegangen wird, um die ursprüngliche Ordnung nicht zu zerstören.

Die Repräsentation der Fächer kann dabei auf zwei verschiedene Arten geschehen:

1. Array der Länge n, wobei die Fächer einfach hintereinander zu finden sind. Dazu braucht es ein weiteres Array der Länge m, in welchem diese Grenzen zu finden sind, wobei diese Grenzen in einem ersten Durchgang bestimmt werden müssen.
2. Verkettete Listen haben den Vorteil, dass kein 2. Durchgang nötig ist.

3.3 Bucketsort

Laufzeit: zwischen O(n) und $O(n^2)$
in situ: nein
stabil: ja

Es werden n gleich grosse „Buckets“ erstellt, wobei dann die Elemente in den entsprechenden Bucket verteilt werden. Danach wird jeder nichtleere Bucket mit einem herkömmlichen Verfahren sortiert, und dann alle Buckets wieder zurück in den Speicher geschrieben. Oft werden die Elemente auch schon vor dem Sortieren in den Buckets wieder zurückgeschrieben, um dann Insertionsort auf das gesamte Array anzuwenden.

Sind die Schlüssel annähernd gleichverteilt, so hat Bucketsort eine lineare Laufzeit. Diese kann jedoch sehr viel schlechter werden, wenn viele Schlüssel nahe beieinander liegen.