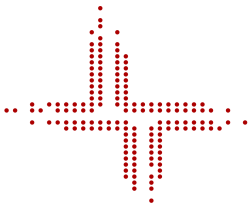

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Sortieralgorithmen

Tobias Schneider, Fatih Kahraman

30. Mai 2017

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Abstract . . . 1
- 1.2 Leser Fangen . . . 1
- 1.3 Problem und Relevanz . . . 1

2 Grundlagen **1**
- 2.1 O-Notation . . . 1
- 2.2 In-Place . . . 2
- 2.3 Stabilität . . . 2
- 2.4 Testumgebung . . . 2

3 Sortieralgorithmen **2**
- 3.1 BubbleSort . . . 2
  - 3.1.1 Geschichte . . . 2
  - 3.1.2 Pseudo-Code . . . 2
  - 3.1.3 Vorgehensweise . . . 3
  - 3.1.4 Eigenschaften . . . 3
  - 3.1.5 Testfälle . . . 3
- 3.2 InsertionSort . . . 4
  - 3.2.1 Geschichte . . . 4
  - 3.2.2 Pseudo-Code . . . 4
  - 3.2.3 Vorgehensweise . . . 4
  - 3.2.4 Eigenschaften . . . 4
  - 3.2.5 Testfälle . . . 5
- 3.3 SelectionSort . . . 5
  - 3.3.1 Geschichte . . . 5
  - 3.3.2 Pseudo-Code . . . 5
  - 3.3.3 Vorgehensweise . . . 5
  - 3.3.4 Eigenschaften . . . 5
  - 3.3.5 Testfälle . . . 5
- 3.4 MergeSort . . . 5
  - 3.4.1 Geschichte . . . 5
  - 3.4.2 Pseudo-Code . . . 5
  - 3.4.3 Vorgehensweise . . . 5
  - 3.4.4 Eigenschaften . . . 5
  - 3.4.5 Testfälle . . . 6
- 3.5 MergeSort . . . 6
  - 3.5.1 Geschichte . . . 6
  - 3.5.2 Pseudo-Code . . . 6
  - 3.5.3 Vorgehensweise . . . 6
  - 3.5.4 Eigenschaften . . . 6
  - 3.5.5 Testfälle . . . 6
- 3.6 HeapSort . . . 6
  - 3.6.1 Geschichte . . . 6

3.6.2 Pseudo-Code 6
3.6.3 Vorgehensweise 6
3.6.4 Eigenschaften 6
3.6.5 Testfälle 7
3.6.6 Pseudo Code 7
3.6.7 Eigenschaften 7
3.6.8 Testfälle 7
3.7 Evaluierung 7
3.7.1 Vergleich der Ergebnisse 7
3.7.2 Relevanz der O-Notation 7
3.7.3 Wie wichtig sind weitere Kriterien 7

4 Schluss **7**
4.1 Zusammenfassung und Ausblick 7
4.1.1 Fazit 7
4.1.2 Anwendungstipps 7

5 Literaturverzeichnis **7**

1 Einleitung

1.1 Abstract

Sortieralgorithmen sind heutzutage nicht mehr wegzudenken. Ihre Benutzung ist essentiell für die Verwaltung von vielen Dateien. Sichtbar für den Benutzer wird es z.B. auf Shopping Webseiten, auf welchem man gewisse Artikel nach Wunsch anordnen kann. Wie und welches Sortieralgorithmus hier verwendet wurde, bleibt dem Kunden unbewusst. Interessant für den Entwickler ist die Stabilität und die Schnelligkeit des Sortieralgorithmusses. In dieser Seminararbeit werden wir uns auf ein Verfahren konzentrieren, mit dem die Laufzeit berechnet werden kann. Ebenso werden wir verschiedene Experimente durchführen, in dem wir unterschiedliche Testumgebungen(IDE) benutzen. Um einen Überblick zu erschaffen, müssen wir uns mit den zahlreichen Eigenschaften eines Sortieralgorithmusses bekanntlich machen. Im Quelltext wird nachgeforscht, ob zusätzlicher Speicher benötigt wird und ob dies die Performance beeinträchtigt. Die umfangreichen Versuche zeigen darauf, dass es keinen perfekten Sortieralgorithmus existiert. Wir veranschaulichen die Stärken und Schwächen der genannten Verfahren in unterschiedlichen Umgebungen.

1.2 Leser Fangen

1.3 Problem und Relevanz

2 Grundlagen

2.1 O-Notation

Mithilfe der O-Notation, auch Landau-Notation gennant, wird eine Laufzeitberechnung anhand der gegebenen Eingabelänge n bestimmt. Dabei wird ein ungefähres Wachstumsverhalten des Algorithmus als mathematische Funktion definiert. Dies geschieht durch Analyse des Quellcodes und wird üblicherweise für drei Fälle durchgeführt: [**?**]

Worst-Case: Es wird nach der maximalen Laufzeit des Algorithmus gesucht. Dies geschieht indem eine obere Schranke aufgestellt wird, über die die Laufzeit nicht steigt. Dieser Fall ist Sortieralgorithmus abhängig und kann nicht immer eine in umgekehrter Reihenfolge sortierte Liste sein.
Best-Case: Stellt die minimale Laufzeit da und wird durch eine untere Schranke realisiert. Auch hier fällt die Laufzeit nicht unter die Schranke. Der Best-Case ist nicht immer eine bereits aufsteigend sortierte Liste sein.
Average-Case: Es wird eine durchschnittliche Laufzeit aufgestellt.

Vorteil der Landau-Notation ist, dass sie komplett unabhängig von Hardware und Betriebssystem ist. Mit ihr kann man die Laufzeiten von Algorithmen miteinander vergleichen. Ein **großer Nachteil** ist das die Funktionen nur angenähert sind und so nur eine grobe Einschätzung liefern. Weiterhin existiert keine Berücksichtigung auf Speicher Allokationen oder Zeitdauer von rekursive aufrufe.

2.2 In-Place

Diese Eigenschaft beschreibt ob der Algorithmus neben der zu sortierenden Liste noch weiteren Speicherplatz benötigt oder nicht. Eine Ausnahme ist der Tausch-Speicher der beim tausch zweier Werte benötigt wird. [**?**]

2.3 Stabilität

Ein stabiler Algorithmus behält die Reihenfolge von äquivalenten Werten bei. Diese Eigenschaft ist bei Sortierung von Zahlen nicht relevant, aber sobald mehr Dateien damit zusammenhängen gewinnt diese Eigenschaft an relevanz. [**?**]

2.4 Testumgebung

SSD vs HDD , CPU, Compiler, Sprache ...

3 Sortieralgorithmen

Vorstellungen von unterschiedlichen SA. min 5 bis (Textlimit erreicht ;D)

3.1 BubbleSort

3.1.1 Geschichte

3.1.2 Pseudo-Code

[**?**]

```
void bubbleSort(T liste[], int anzahl) {
    boolean swapped;
    for (int i = 1; i < anzahl; i++) {
        bool swap = false;
        for (int j = 0; j < anzahl - i; j++) {
            if (liste[j] > liste[j + 1]) {
                    swap(j, j+1);
                    swapped = true;}
        }
        if (!swapped) {
            return;}
}}
```

Code von [**?**].

Best Case	Average Case	Worst Case
O(n)	O(n^2)	O(n^2)

Tabelle 1: O(Notation) des BubbleSorts

3.1.3 Vorgehensweise

Idee: Größere Elemente steigen im Array nach rechts auf.

Der BubbleSort ist ein Algorithmus der mit zwei ineinander verschachtelten Schleifen arbeitet. Die äußere Schleife definiert bis zu welchem Punkt die innere Schleife läuft. Die innere Schleife vergleicht das aktuelle Element mit dem darauf folgenden, und falls das aktuelle größer ist, werden diese beiden Elemente vertauscht.
Wenn die innere Schleife einen Durchlauf abgeschlossen hat, steht das größte Element am Ende und wird danach nicht mehr berücksichtigt.
Eine Besonderheit wurde noch hinzugefügt, es wird in einem Schleifendurchlauf mittels „swapped“ kontrolliert ob getauscht wurde. Falls nicht getauscht wurde, ist die Liste schon fertig sortiert und der BubbleSort wird abgebrochen. Mit diesem Trick ist der Best-Case O(n) möglich.

3.1.4 Eigenschaften

O-Notation: Da bei einer bereits sortierten Liste keine vertauschungungen erfolgen , ist der BubbleSort nach einem Schleifendurchlauf fertig und beendet sich mit dem break. Beim Worst-Case werden beide Schleifen komplett durchlaufen was zu einer O-Notation von n^2 führt.

Best-Case Der Best-Case des BubbleSorts ist eine bereits sortierte Liste. Der Algorithmus geht die Liste einmal durch und merkt es wurde nicht getauscht. Es erfolgt ein break;. Die O-Notation ist deshalb O(n).
Worst-Case Eine in falscher Reihenfolge sortierte Liste ist der Worst-Case des BubbleSorts. Da dadurch beide Schleifen komplett durchlaufen werden ist die O-Notation O(n^2)
Stabilität Da nur jeweils benachbarte Elemente mittels echt größer als verglichen und vertauscht werden, ist der BubbleSort ein stabiler Algorithmus

In-Place Da der Algorithmus keine rekursiven aufrufe durchführt und lediglich zum vertauschen einen temporären Speicherplatz benötigt, arbeitet der Algorithmus mit O(1).

3.1.5 Testfälle

kommen bald!

Best Case	Average Case	Worst Case
O(n^2)	O(n^2)	O(n^2)

Tabelle 2: O(Notation) des InsertionSorts

3.2 InsertionSort

3.2.1 Geschichte

3.2.2 Pseudo-Code

```
void insertionSort(T arr[], int length) {
    int i, j;
    T tmp;
    for (i = 1; i < length; i++) {
        j = i;
        while (j > 0 && arr[j - 1] > arr[j]) {
            swap(j, j-1);
            j--;
        }
    }
}
```

Code von [**?**]

3.2.3 Vorgehensweise

Idee: Einsortieren von Elementen in eine bereits sortierte Liste.

Der InsertionSort arbeitet wie der BubbleSort mit zwei ineinander verschachtelten Schleifen.Die Anzahl der Elemente ist am anfang auf 2 Elemente festgelegt. Bei jedem äußeren Durchlauf wird die größe des Arrays um eins erhöht, bis zu seiner maximal größe. Dadurch wird immer ein Element zur bereits sortierten Liste hinzugefügt und an die richtige stelle einsortiert. Die bereits sortierten Elemente werde falls nötig nach hinten verschoben.

3.2.4 Eigenschaften

O-Notation:

Best-Case Der Best-Case des InsertionSorts ist eine bereits sortierte Liste. Da dann die Anzahl der verschiebungen null ist. Da die Äußere Schleife aber n mal aufgerufen wird und die innere auch ist hier die O-Notation O(n^2).
Worst-Case Eine falsch herum sortierte Liste ist der Worst-Case. Ähnlich wie beim Best-Case werden die Schleifen durchlaufen. Es wird aber bei jedem durchgang maximal verschoben. Auch hier ist die O-Notation O(n^2).
Stabilität: Es werden nur benachbarte Elemente miteinander verschoben falls ein Element echt größer ist. Es werden außerdem nur Elemente von links in das Array eingefügt. Dadurch ist die Stabilität gegeben.

In-Place: Der Algorithmus arbeitet nur auf der übergebenden Liste. Es wird nur ein zusätzlicher Speicher für das Vertauschen benötigt deshalb ist der

Best Case	Average Case	Worst Case
O(n)	O(n^2)	O(n^2)

Tabelle 3: O(Notation) des InsertionSorts

InsertionSort O(1).

3.2.5 Testfälle

3.3 SelectionSort

3.3.1 Geschichte

3.3.2 Pseudo-Code

```
void selectionSort(T liste[], int anzahl) {
    int k;
    T temp;
    for (int i = 0; i < anzahl; i++) {
        k = i;
        for (int j = i + 1; j < anzahl; j++) {
            if (liste[j] < liste[k]) {
                k = j;
            } }
        swap(i,k)
            }}
```

3.3.3 Vorgehensweise

Idee: Suche das kleinste Element und stelle es nach vorne.

3.3.4 Eigenschaften

O-Notation:
Best-Case Der Best-Case des .
Worst-Case
Stabilität

In-Place

Best Case	Average Case	Worst Case
O(n)	O(n^2)	O(n^2)

Tabelle 4: O(Notation) des InsertionSorts

Best Case	Average Case	Worst Case
O(n)	O(n^2)	O(n^2)

Tabelle 5: O(Notation) des InsertionSorts

3.3.5 Testfälle

3.4 MergeSort

3.4.1 Geschichte

3.4.2 Pseudo-Code

3.4.3 Vorgehensweise

3.4.4 Eigenschaften

O-Notation:
Best-Case Der Best-Case des .
Worst-Case
Stabilität

In-Place

3.4.5 Testfälle

3.5 MergeSort

3.5.1 Geschichte

3.5.2 Pseudo-Code

3.5.3 Vorgehensweise

3.5.4 Eigenschaften

O-Notation:
Best-Case Der Best-Case des .
Worst-Case
Stabilität

In-Place

Best Case	Average Case	Worst Case
O(n)	O(n^2)	O(n^2)

Tabelle 6: O(Notation) des InsertionSorts

3.5.5 Testfälle

3.6 HeapSort

3.6.1 Geschichte

3.6.2 Pseudo-Code

3.6.3 Vorgehensweise

3.6.4 Eigenschaften

O-Notation:
Best-Case Der Best-Case des .
Worst-Case
Stabilität

In-Place

3.6.5 Testfälle

3.6.6 Pseudo Code

3.6.7 Eigenschaften

Wie sind die O-Notation, In-Place, Stabilität zu diesem SA

3.6.8 Testfälle

Worst Case, Average Case, Best Case, nearly sorted, festplattenart, genug Speicher, zuwenig Speicher, unterschiedliche Datentypen - Integer versus Klassenobjekte

3.7 Evaluierung

3.7.1 Vergleich der Ergebnisse

3.7.2 Relevanz der O-Notation

3.7.3 Wie wichtig sind weitere Kriterien

In-Place, Stabilität, Speicherplatz ...

4 Schluss

4.1 Zusammenfassung und Ausblick

4.1.1 Fazit

4.1.2 Anwendungstipps

5 Literaturverzeichnis

Literatur

[1] C. Rehn, “Sortieralgorithmen unter mathematischen gesichtspunkten.”