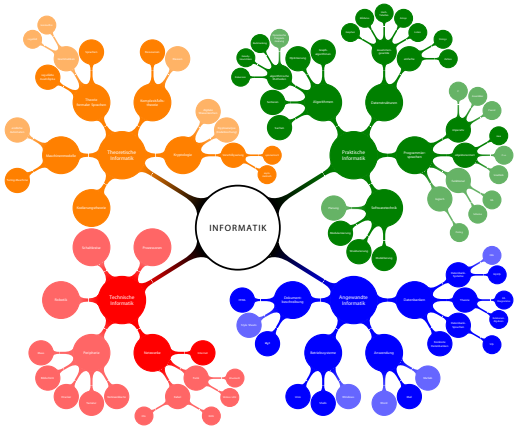
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Kapitel 17

Sortieralgorithmen

Skat, das Telefonbuch und Atome

Vorlesung Einführung in die Informatik 1 vom 19. Dezember 2013 von Till Tantau



Lernziele von Kapitel 17

1. Arten von Sortierproblemen kennen
2. Bubble-Sort, Insertion-Sort, Selection-Sort verstehen und implementieren können

Gliederung von Kapitel 17

Wo taucht Sortieren überall auf?

Das *abstrakte* Sortierproblem taucht allein in der Molekularbiologie in vielen Kontexten auf:

- Eine Liste von Genen soll alphabetisch sortiert angezeigt werden.
- Ein Molekül soll gezeichnet werden. Dazu sollen die Atome »von hinten nach vorne« gezeichnet werden.
- Gene sollen nach Entfernung von einer Site sortiert werden.
- Vorhersagen sollen nach Wahrscheinlichkeit sortieren werden.
- In Gen- oder Proteindatenbanken sollen die Daten vorsortiert werden.

Was meint man mit »Sortieren« eigentlich?

Das einfachste Sortierproblem

Eingabe	Array von Zahlen
Ausgabe	Array mit denselben Zahlen, aber in der Reihenfolge so verändert, dass jede Zahl höchstens so groß wie ihr Nachfolger ist.

Eine »Veränderung in der Reihenfolge« nennt man auch *Permutation*.

Definition des allgemeinen Sortierproblems.

Das allgemeine Sortierproblem

Eingabe Array von Objekten, die sich vergleichen lassen

Ausgabe Eine Permutation der Objekte, so dass jedes Objekt in der neuen Reihenfolge kleiner oder gleich dem nachfolgenden ist.

Wünschenswerte Eigenschaften von Sortieralgorithmen.

Folgende Eigenschaften sind bei Sortieralgorithmen besonders wünschenswert:

1. Ein Verfahren ist *stabil*, falls sich die Reihenfolge von gleichen Elementen nicht ändert.
   Beispiel: Eine Adressliste wird nach Namen sortiert. Kommt vor der Sortierung Peter Müller aus Berlin vor Peter Müller aus Aachen, so soll dies nach der Sortierung immernoch der Fall sein.
2. Ein Verfahren ist *in-place*, falls es lediglich eine kleine Menge extra Speicher benötigt, falls es also keine Kopie des Arrays benötigt.
3. Das Verfahren sollte mit *möglichst wenigen* Vergleichen, Vertauschungen und Verschiebungen auskommen.

Der Bubble-Sort-Algorithmus

Idee

- Wir sind fertig, wenn für je zwei aufeinanderfolgende Objekte gilt, dass das erste kleiner oder gleich dem zweiten ist.
- Also suchen wir nach Paaren, bei denen dies nicht der Fall ist, und tauschen sie aus.

Erste Version von Bubble-Sort


```
static void stupidSort (int[] array)
{
  int i = 0;
  while (i < array.length-1) {
    if (array[i] > array[i+1]) {
      // Korrigiere die Reihenfolge:
      swap(array, i, i+1);

      // Neustart
      i = 0;
    }
    else {
      i = i+1;
    }
  }
}

static void swap (int[] array, int i, int j)
{
  // Vertausche array[i] und array[j]
  int temp = array[i];
  array[i] = array[j];
  array[j] = temp;
}
```

Verbesserungen der ersten Version

- Es macht keinen Sinn, nach jeder Vertauschung wieder am Anfang zu beginnen.
- Stattdessen macht man einfach mit dem nächsten Element weiter.
- Dann ist am Ende eines Durchgangs das größte Element am Ende.
- Jede folgende Runde kann dann eins früher enden.

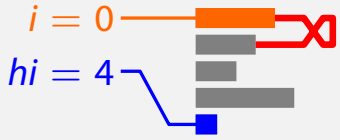
Zweite Version von Bubble-Sort


```
static void bubbleSort (int[] array)
{
  for (int hi = array.length-1; hi >= 0; hi = hi-1) {
    for (int i = 0; i < hi; i = i+1) {
      if (array[i] > array[i+1]) {
        swap(array, i, i+1);
      }
    }
  }
}
```

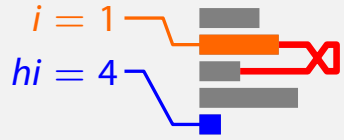
Ablauf von Bubble-Sort an einem Beispiel

Der zu sortierende Array sei `array == {4,3,2,5,1}`.

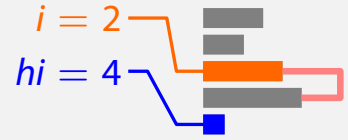
1



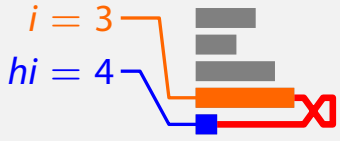
2



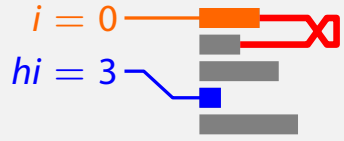
3



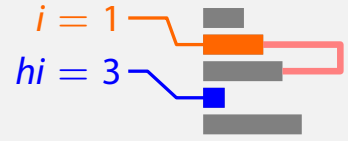
4



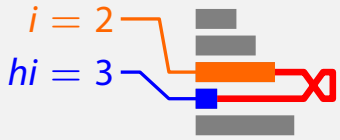
5



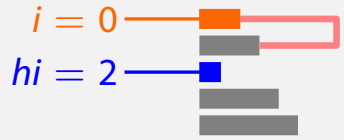
6



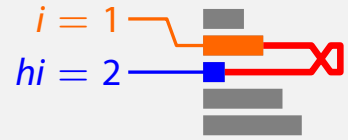
7



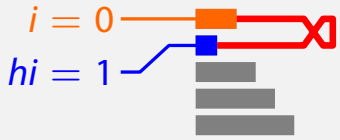
8



9



10



11

5-Minuten-Aufgabe

Zur Übung

Bei einem Array der Länge *n*, wie viele

1. Vergleiche macht Bubble-Sort mindestens (grob)?
2. Vergleiche macht Bubble-Sort höchstens (grob)?
3. Vertauschungen macht Bubble-Sort mindestens (grob)?
4. Vertauschungen macht Bubble-Sort höchstens (grob)?

Vor- und Nachteile von Bubble-Sort

Vorteile

+ Einfach zu programmieren.
+ Einfach zu verstehen.
+ Kann leicht modifiziert werden, so dass er bei sortierten Daten sehr schnell ist.
+ In-place und stabil.

Nachteile

– Bei zufälligen Daten langsam, da viele Vergleiche.
– Bei fast sortierten Daten trotzdem langsam.

Der Selection-Sort-Algorithmus

Idee

- Wir wollen möglichst wenig vertauschen.
- Deshalb suchen wir zunächst das kleinste Element im Array und tauschen es an die erste Stelle.
- Im Rest suchen wir dann wieder das kleinste und tauschen es an die zweite Stelle, und so weiter.

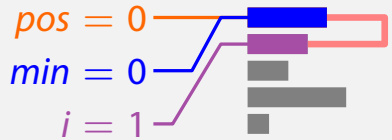
Selection-Sort


```
static void selectionSort (int[] array)
{
  for (int pos = 0; pos < array.length-1; pos++) {
    // Finde erstes Minimum ab Position pos
    int min = pos;

    for (int i = pos+1; i<array.length; i++) {
      if (array[min] > array[i]) {
        min = i;
      }
    }

    swap(array,pos,min);
  }
}
```

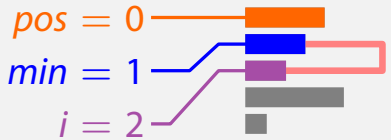
Ablauf von Selection-Sort an einem Beispiel

Der zu sortierende Array sei `array == {4,3,2,5,1}`.

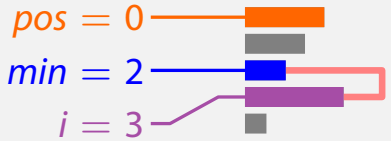
1



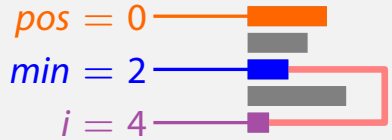
2



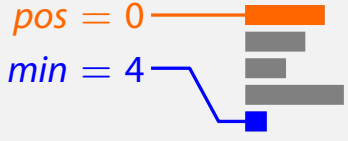
3



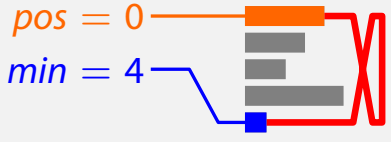
4



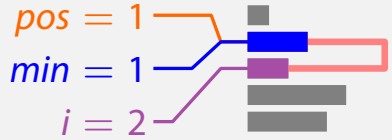
5



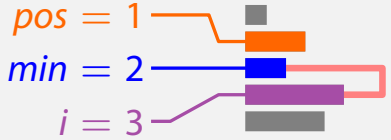
6



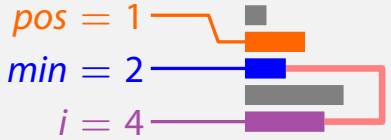
7



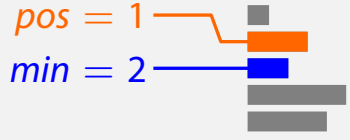
8



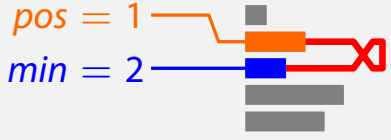
9



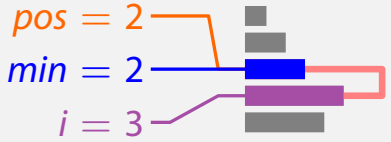
10



11



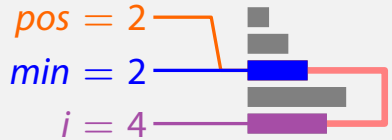
12



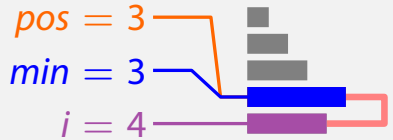
Ablauf von Selection-Sort an einem Beispiel

Der zu sortierende Array sei `array == {4,3,2,5,1}`.

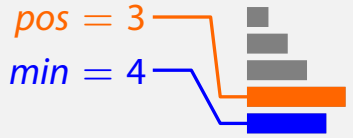
13



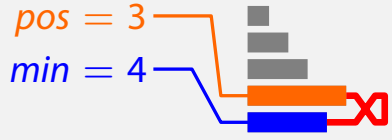
14



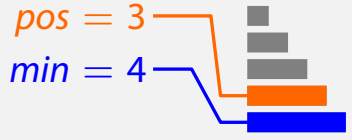
15



16



17



Vor- und Nachteile von Selection-Sort

Vorteile

+ Einfach zu verstehen.
+ Minimale Anzahl an Vertauschungen.
+ In-place.

Nachteile

– Etwas aufwändiger zu programmieren.
– Immer viele Vergleiche, selbst bei sortierten Daten.
– Nicht stabil.

Der Insertion-Sort-Algorithmus

Idee

- Wir halten den ersten Teil des Arrays immer sortiert.
- Um den sortierten Teil des Arrays um ein Element zu erweitern, tauschen wir dies so lange nach links, bis es am Ziel angekommen ist.

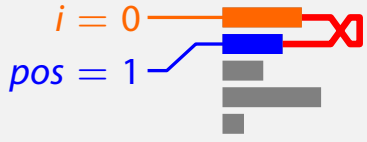
Insertion-Sort


```
static void insertionSort (int[] array)
{
  for (int pos = 1; pos < array.length; pos++) {
    int i = pos-1;
    while (i >= 0 && array[i] > array[i+1])
    {
      swap(array,i,i+1);
      i = i - 1;
    }
  }
}
```

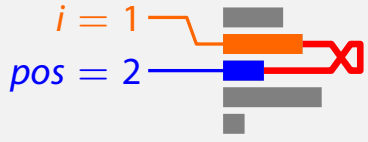
Ablauf von Insertion-Sort an einem Beispiel

Der zu sortierende Array sei `array == {4,3,2,5,1}`.

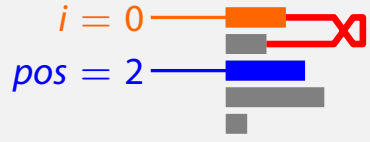
1



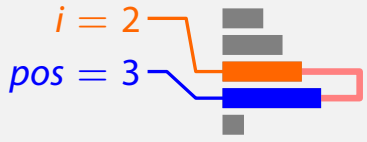
2



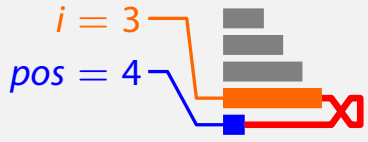
3



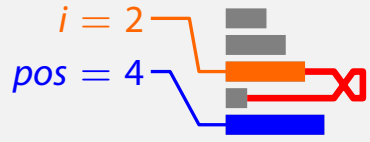
4



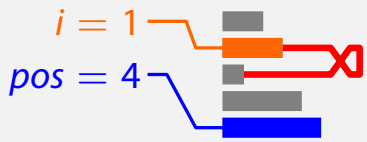
5



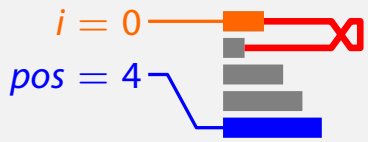
6



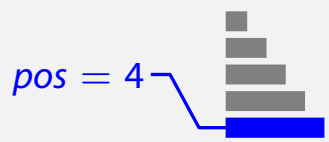
7



8



9



Vor- und Nachteile von Insertion-Sort

Vorteile

+ Einfach zu verstehen.
+ Sehr schnell bei sortierten und fast sortierten Daten.
+ In-place und stabil.

Nachteile

– Bei zufälligen Daten viele Vergleiche und Vertauschungen.

5-Minuten-Aufgabe

Zur Übung

Bei einem Array der Länge n, wie viele

1. Vergleiche macht Insertion-Sort mindestens (grob)?
2. Vergleiche macht Insertion-Sort höchstens (grob)?
3. Vertauschungen macht Insertion-Sort mindestens (grob)?
4. Vertauschungen macht Insertion-Sort höchstens (grob)?

Wie viele Vergleiche werden mindestens benötigt?

- Bubble-, Insertion- und Selection-Sort benötigten *grob* $n^2/2$ Vergleiche im schlimmsten Fall.
- Dies ist nicht optimal, Merge-Sort benötigt lediglich *grob* $n \log_2 n$ Vergleiche.
- Man kann sicherlich nicht mit weniger als mit $n - 1$ Vergleichen auskommen.
- Wie viele Vergleiche benötigt also ein *optimaler* Algorithmus?

Die untere Schranke für die Anzahl der Vergleiche.

Satz

Jeder Sortieralgorithmus, der lediglich Vergleiche zum Sortieren nutzt, benötigt mindestens $n \log_2 \frac{n}{e}$ *Vergleiche, um n Werte zu sortieren.*

Sortieralgorithmen wie Merge-Sort sind also optimal.

Beweis der unteren Schranke

- Sei A ein beliebiges Sortieralgorithmus.
- Für jede Permutation π der Zahlen von 1 bis n betrachten wir die Folge der Vergleiche, die der Algorithmus durchführt.
- Jeder Vergleich liefert entweder »kleiner« oder »größer«; jede Antwortfolge entspricht also einem Bitstring.
- Sind zwei Permutationen unterschiedlich, so muss die Folge der gelieferten Antworten auf die Vergleiche unterschiedlich sein.

Beweis der unteren Schranke

- Es gibt $n!$ Permutationen,
- aber nur $2^{l+1} - 1$ Bitstrings der Länge höchstens l.
- Es muss also, damit alle Permutationen durch einen anderen Bitstring repräsentiert werden, gelten:

$$2^{l+1} - 1 \geq n!.$$

- Dies liefert

$$l \geq \log_2(n! + 1) - 1.$$

- Die Stirling-Approximation der Fakultät liefert nun

$$l \geq \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n + 1\right) - 1 \geq n \log_2 \frac{n}{e}.$$

Zusammenfassung

Sortieren

- *Sortieren* ist das Problem, für eine Liste von Objekten eine *Permutation* zu finden, so dass sie in monoton wachsender Reihenfolge sind.
- Man muss mindestens $n \log_2(n/e)$ Vergleiche durchführen, um n Objekte zu sortieren.
- Die Algorithmen Bubble-Sort, Selection-Sort und Insertion-Sort benötigen alle schlimmstenfalls mindestens $n^2/2$ Vergleiche.

Zusammenfassung

Bubble-Sort

So lange noch zwei Objekte nebeneinander in der falschen Reihenfolge sind, vertausche sie.

Selection-Sort

Für alle Positionen *i* tue nacheinander: Finde das kleinste Element ab Position *i* und tausche es an Stelle *i*.

Insertion-Sort

Für alle Positionen *i* tue nacheinander: Tausche das Element an Stelle *i* so lange mit seinen Vorgängern, bis die Element von 1 bis *i* in sortierter Reihenfolge sind.