Programmiertechnik II

Ralf Herbrich

Sortieren: Elementare Sortierverfahren

Überblick

1. Sortierprobleme
2. *Selection Sort*
3. *Insertion* und *Bubble Sort*
4. *Shell Sort*

Überblick

1. **Sortierprobleme**
2. *Selection Sort*
3. *Insertion* und *Bubble Sort*
4. *Shell Sort*

Sortierprobleme

- **Beispiel**: Kontaktlisten
  - Originale Liste von Kontakten

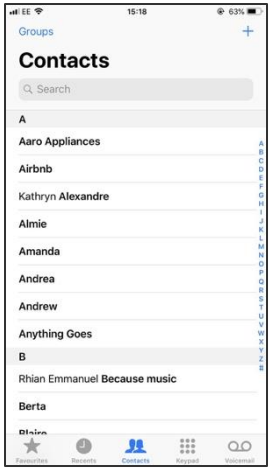
Schlüssel

Eintrag

Name	Telefonnummer
Berta	0151-49205030
Dirk	0176-93544523
Celine	0152-49502663
Andrew	0172-34290034

  - Sortierte Liste von Kontakten

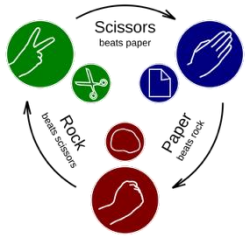
Name	Telefonnummer
Andrew	0172-34290034
Berta	0151-49205030
Celine	0152-49502663
Dirk	0176-93544523



Totale Ordnung

- Um Einträge zu sortieren, brauchen wir eine Vergleichsoperation $\leq$, die eine totale Ordnung definiert:
  1. **Asymmetrie**: Wenn $v \leq w$ und $w \leq v$ dann gilt $v = w$
  2. **Transitivität**: Wenn $u \leq v$ und $v \leq w$ dann gilt $u \leq w$
  3. **Total**: Entweder gilt $v \leq w$ oder $w \leq v$ oder beides
- **Beispiele**:
  - Vergleichsoperator für Ganzzahlen und Gleitkommazahlen (z.B., $4 \leq 6$)
  - Lexikographische Ordnung von Zeichenketten (z.B., “algol” $\leq$ “algorithms”)
  - **Aber**: Stein-Papier-Schere ist nicht transitiv weil $\text{Papier} \leq \text{Schere}$, $\text{Schere} \leq \text{Stein}$, aber $\text{Stein} \leq \text{Papier}$
- Implementierung durch Funktion `bool less(const V* a, int i, int j)`

```
template <typename Value>
bool less(const Value* a, const int i, const int j) {
    return (a[i] < a[j]);
}
```



Überblick

1. Sortierprobleme
2. ***Selection Sort***
3. *Insertion* und *Bubble Sort*
4. *Shell Sort*

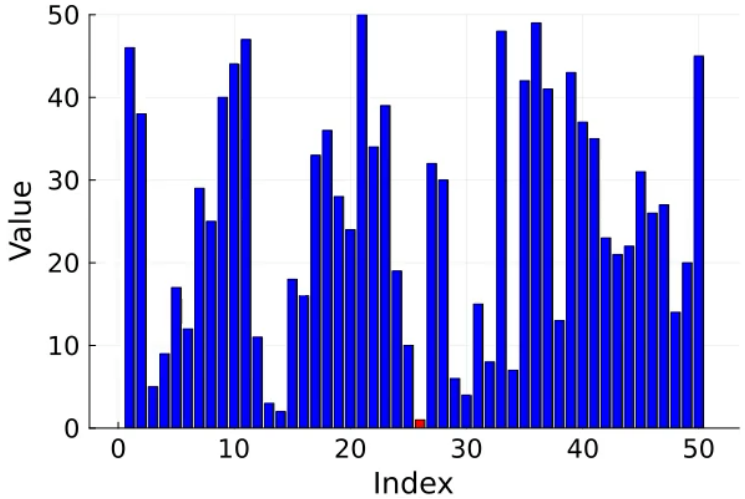
Selection Sort

- **Grundidee**: Kleinstes, noch nicht sortiertes Element, an das Ende der sortierten Elemente tauschen
  - Für jeden Index i
    1. Finde den Index $\min \in \{i+1, \ldots, n\}$ so dass $a_{\min} \leq a_j$ mit $j \in \{i+1, \ldots, n\}$
    2. Tausche a_i und $a_{\min}$ so, dass a_i danach das i-kleinste Element enthält

```
// Implements a swap of element i and j in an array
template <typename Value>
void swap(Value* a, const int i, const int j) {
    const Value tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
    return;
}
```

Selection Sort Algorithmus

```
// Implements selection sort
template <typename Value>
void selection_sort(Value* a, const int n) {
    for (auto i = 0; i < n; i++) {
        auto min = i;
        for (auto j = i+1; j < n; j++) {
            if (less(a, j, min)) {
                min = j;
            }
        }
        swap(a, i, min);
    }
    return;
}
```



Finde den Index des kleinsten Elements in allen nicht-sortierten Elementen

Selection Sort: Mathematische Analyse

- **Satz**: *Selection Sort* benutzt $(n-1)+(n-2)+\cdots+1 \sim n^2/2$ Vergleiche und n Vertauschungen
- **Beweis**:

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	**A**	M	P	L	E
1	4	A	O	R	T	**E**	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	**E**
3	9	A	E	E	T	O	X	S	M	P	**L**	R
4	7	A	E	E	L	O	X	S	**M**	P	T	R
5	7	A	E	E	L	M	X	S	**O**	P	T	R
6	8	A	E	E	L	M	O	S	X	**P**	T	R
7	10	A	E	E	L	M	O	P	X	S	T	**R**
8	8	A	E	E	L	M	O	P	R	**S**	T	X
9	9	A	E	E	L	M	O	P	R	S	**T**	X
10	10	A	E	E	L	M	O	P	R	S	T	**X**

Laufzeit ist unabhängig von der Eingabe (selbst wenn die Eingabe sortiert ist)

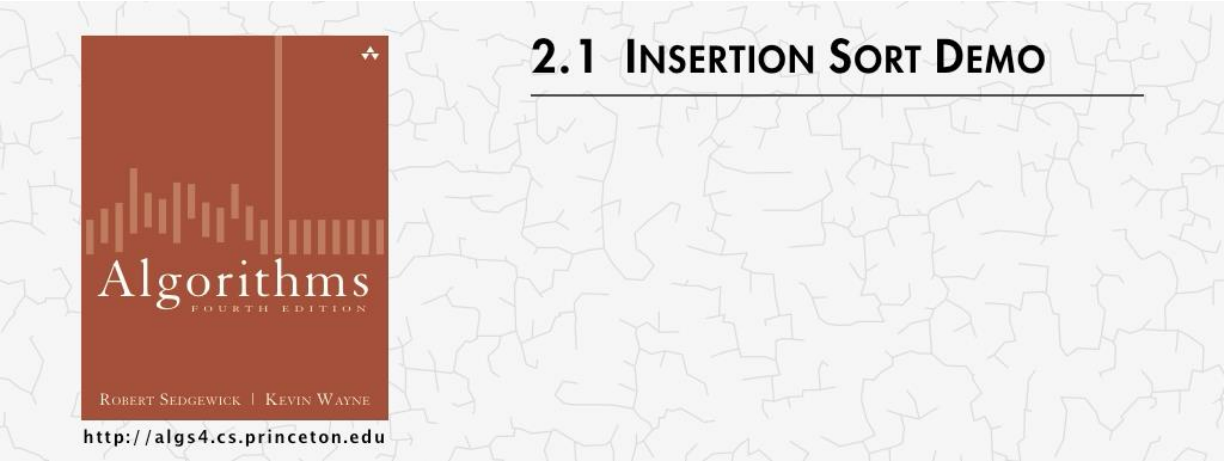
Anzahl Vertauschungen sind minimal (immer linear in der Eingabelänge)

Überblick

1. Sortierprobleme
2. *Selection Sort*
3. ***Insertion* und *Bubble Sort***
4. *Shell Sort*

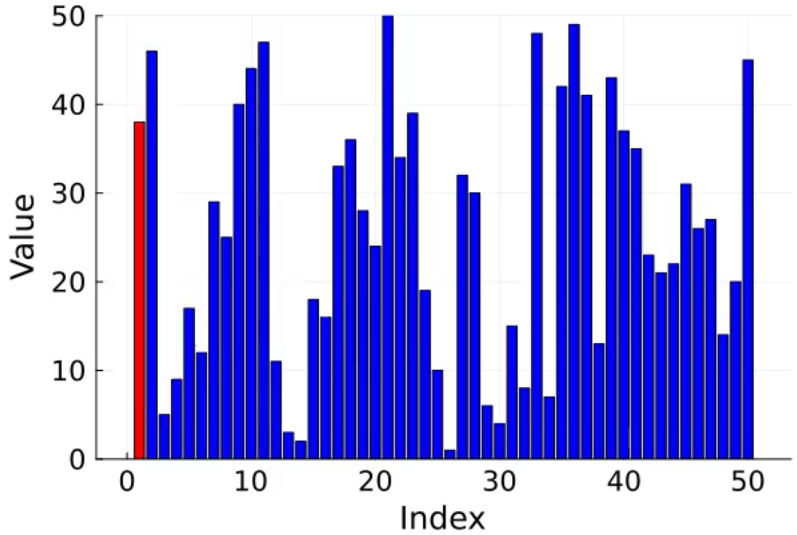
Insertion Sort

- **Grundidee**: Das nächste, noch nicht sortiertes Element, an die korrekte Stelle in die sortierten Elemente tauschen
  - Für jeden Index i
    1. Starte bei $j = i$
    2. Tausche a_j und a_{j-1}, so lange wie $a_j < a_{j-1}$ und zähle j um eins runter



Insertion Sort Algorithmus

```cpp
// Implements insertion sort
template <typename Value>
void insertion_sort(Value* a, const int n) {
    for (auto i = 1; i < n; i++) {
        for (auto j = i; j > 0 && less(a, j, j - 1); j--) {
            swap(a, j, j - 1);
        }
    }
    return;
}
```



Tausche in die schon sortierten Elemente, bis das Element `a[i]` an der richtigen Stelle ist

Insertion Sort: Mathematische Analyse

- **Satz**: In einem zufällig-sortierten Array benutzt *Insertion Sort* $\sim n^2/4$ Vergleiche und $\sim n^2/4$ Vertauschungen
- **Beweis**: In Schritt i sind im Erwartungswert $i/2$ Vertauschungen notwendig

i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X

- ***Best Case***: Wenn die Eingabe aufsteigend sortiert ist, braucht *Insertion Sort* $n - 1$ Vergleiche und 0 Vertauschungen.
- ***Worst Case***: Wenn die Eingabe absteigend sortiert ist, braucht *Insertion Sort* $\sim 1/2\, n^2$ Vergleiche und $\sim 1/2\, n^2$ Vertauschungen.

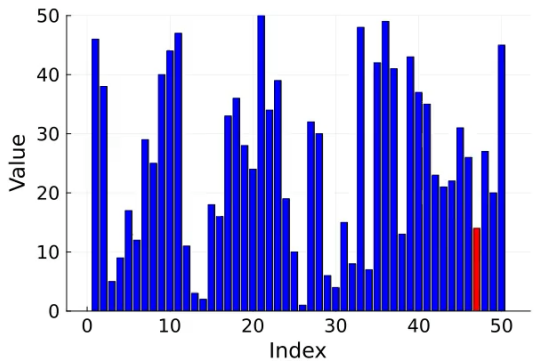
Bubble Sort

- Sehr ähnlich zu *Insertion Sort*: Nach dem i-ten Schritt sind immer alle Elemente $a_0, \dots, a_i$ sortiert aber $a_{i+1}, \dots, a_{n-1}$ noch unsortiert
  - *Insertion Sort*: Im i-ten Schritt wird a_i an die richtige Stelle in $a_0, \dots, a_i$ getauscht
  - *Bubble Sort*: Im i-ten Schritt wird das kleinste Element von $a_i, \dots, a_{n-1}$ zu a_i getauscht

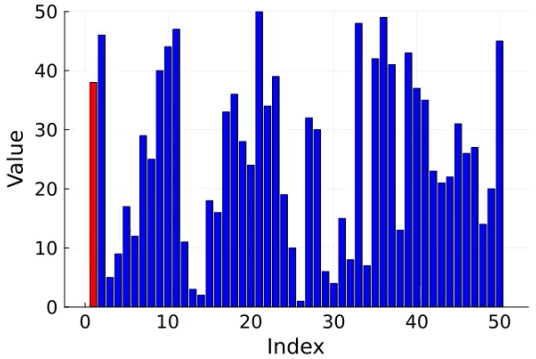
Finde das kleinste Element in `a[i]` bis `a[n-1]` durch paarweises Vertauschen

```
// Implements bubble sort
template <typename Value>
void bubble_sort(Value* a, const int n) {
    for (auto i = 0; i < n; i++) {
        for (auto j = n-1; j > i; j--) {
            if (less(a, j, j-1)) {
                swap(a, j, j-1);
            }
        }
    }
    return;
}
```

Bubble Sort



Insertion Sort



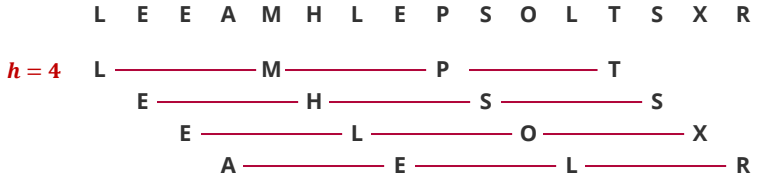
- *Bubble* und *Insertion Sort* haben die gleichen Laufzeitkomplexitäten!

Überblick

1. Sortierprobleme
2. *Selection Sort*
3. *Insertion* und *Bubble Sort*
4. ***Shell Sort***

Shell Sort

- **Idee**: Vertausche weit entfernte Elemente durch h-Sortierung das Array
- ***h*-Sortierung**: Ein Array ist h-sortiert wenn $\{a_i, a_{i+h}, \dots, a_{i+k \cdot h}\}$ für alle $i \in \{0, \dots, h-1\}$ sortiert sind.
- ***Shell Sort***: h-Sortiere ein Array für absteigende Werte von h



Donald Shell
(1924 – 2015)

Programmiertechnik II

Unit 5a – Elementare Sortierverfahren

h-Sortierung

- **Wie soll man ein Array h-sortieren**? Mit *Insertion Sort*!
  - Großes h: Schnell, weil kleines Teilarray!
  - Kleines h: Schnell, weil das Array schon fast sortiert ist!

7-Sortierung

S	O	R	T	E	X	A	M	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	L	T	E	X	A	S	P	R	E
M	O	L	E	E	X	A	S	P	R	T

3-Sortierung

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

1-Sortierung

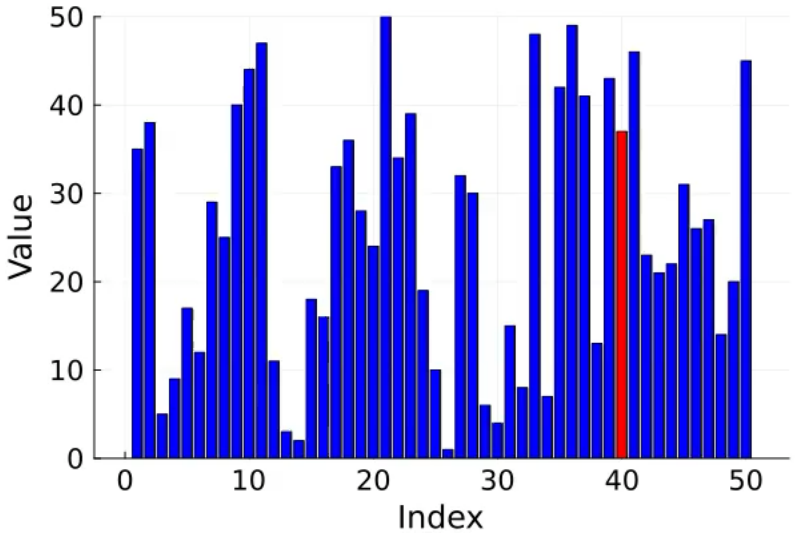
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	R	S	X	T
A	E	E	L	M	O	P	R	S	T	X

Shell Sort Algorithmus

```
// Implements shell sort
template <typename Value>
void shell_sort(Value* a, const int n) {

    // 3x+1 increment sequence:  1, 4, 13, 40, 121, 364, 1093, ...
    int h = 1;
    while (h < n/3) {
        h = 3*h + 1;
    }

    while (h >= 1) {
        // h-sort the array
        for (auto i = h; i < n; i++) {
            for (auto j = i; j >= h && less(a, j, j-h); j -= h) {
                swap(a, j, j-h);
            }
        }
        h /= 3;
    }
    return;
}
```



Geeigneten Startwert für h finden

Insertion Sort mit *h-strides*

Shell Sort: Mathematische Analyse

- **Welche Sequenz von h-Werten?**
  - Potenzen von zwei: 1, 2, 4, 8, 16, 32, …
  - Potenzen von zwei minus eins: 1, 3, 7, 15, 31, 63, …
  - $3x + 1$: 1, 4, 13, 40, 121, 364, …
  - Sedgewick: 1, 5, 19, 41, 109, 209, 505 ← Vereinigung von $(9 \cdot 4^i) - (9 \cdot 2^i) + 1$ und $4^i - (3 \cdot 2^i) + 1$
- $3x + 1$ ist leicht zu berechnen aber Sedgewick ist empirisch am besten!

- ***Worst Case***: Im schlimmsten Fall braucht *Shell Sort* mit dem $3x + 1$ Werten $\sim \Theta(n^{1.5})$ Vergleiche und $\Theta(n^{1.5})$ Vertauschungen.
- ***Best Case***: Im besten Fall braucht *Shell Sort* mit dem $3x + 1$ Werten $\sim n \cdot \log_2(n)$ Vergleiche und Vertauschungen
- ***Average Case***: Ein genaues Laufzeitmodell und optimale h-Wertsequenzen von *Shell Sort* für zufällig-sortierte Arrays ist noch ein ungelöstes Problem!
  - **Bemerkung**: Empirisch braucht *Shell Sort* mit dem $3x + 1$ Werten $\approx n \cdot \log_2(n)$ Vergleiche und Vertauschungen.

Zusammenfassung

- *Shell Sort* ist ein gutes Beispiel, wie eine einfache Idee zu signifikaten algorithmischen Verbesserungen führen kann.
  - Wird in der Praxis benutzt für kleine Arrays (z.B., bzip2, /linux/kernel/groups.c)
  - Wir auch in eingebetten Systemen benutzt

Algorithmus	Best Case	Average Case	Worst Case
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Bubble Sort	n	n^2	n^2
Shell Sort $(3x + 1)$	$n \cdot \log_2(n)$	?	$n^{1.5}$
Ziel	n	$n \cdot \log_2(n)$	$n \cdot \log_2(n)$

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!