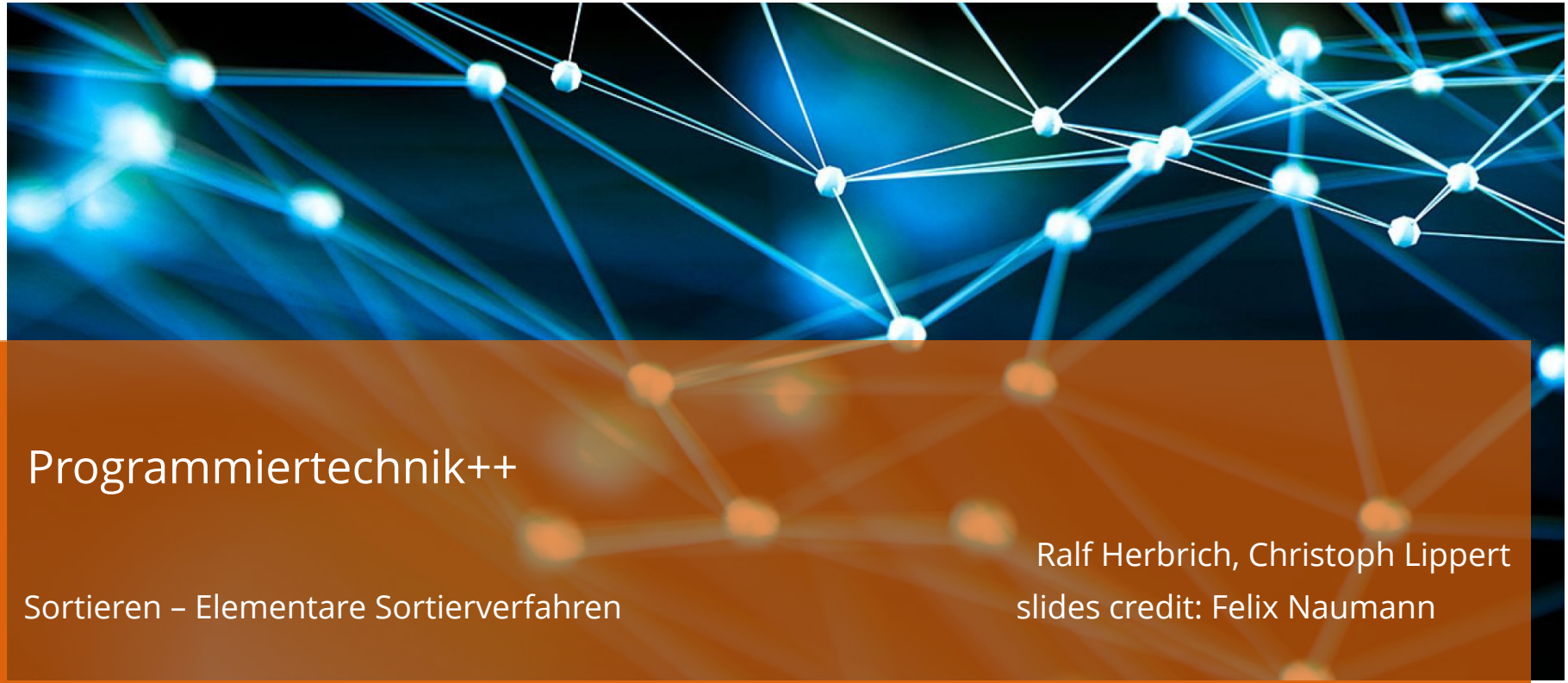


Programmiertechnik++

Sortieren – Elementare Sortierverfahren

Ralf Herbrich, Christoph Lippert
slides credit: Felix Naumann

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Sortieren

- Aufgabe
  - Ordnen von Dateien mit Datensätzen, die Schlüssel enthalten
  - Umordnen der Datensätze, so dass klar definierte Ordnung der Schlüssel (numerisch/alphabetisch) besteht
- ¼ aller Rechenzeit entfällt auf Sortiervorgänge (nach Ottmann, Widmayer)
- Vereinfachung
  - Nur Betrachtung der Schlüssel, z.B. Array von int-Werten
- **Interne** Sortierverfahren: in Hauptspeicherstrukturen (Array, Listen)
- **Externe** Sortierverfahren: Datensätze auf externen Medien (Festplatte, Magnetband)
  - Wichtig in Datenbanksystemen

Überblick

1. Selectionsort
2. Bubblesort
3. Insertionsort
4. Shellsort

Überblick

1. **Selectionsort**
2. Bubblesort
3. Insertionsort
4. Shellsort

Sortieren durch Selektion - Selectionsort

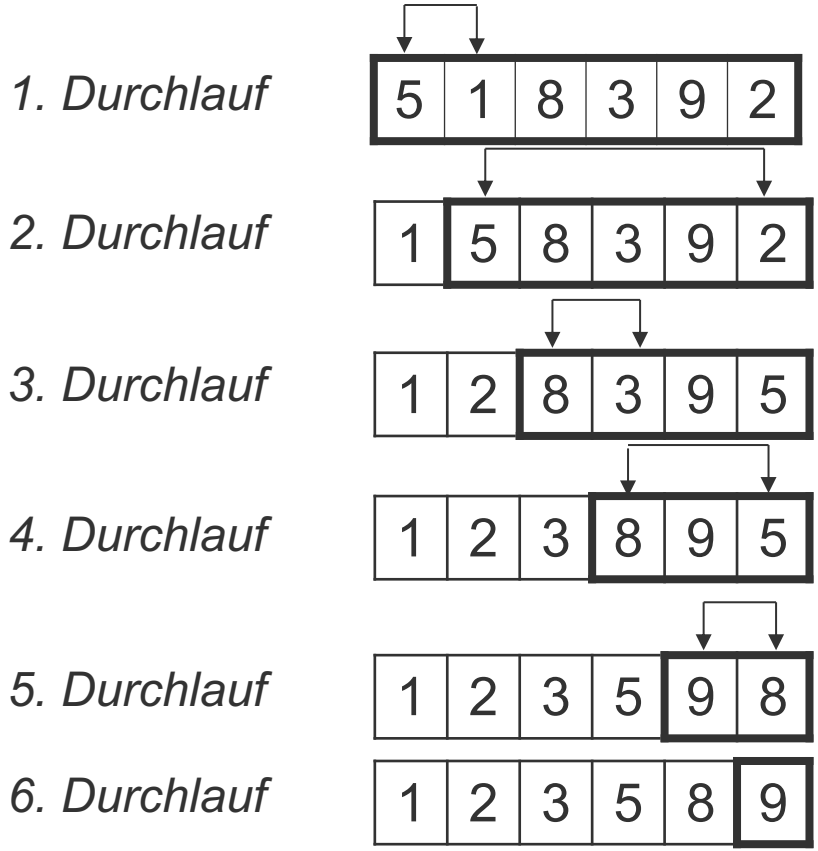
Idee:

Suche jeweils den kleinsten Wert des unsortierten Teilarrays. Tausche diesen an die erste Stelle des unsortierten Teilarrays und füge diese dem sortierten Teilarray zu; fahre dann mit dem restlichen unsortierten Teilarray fort.

```
template <typename T>
void selectionSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        swap(arr[min_index], arr[i]);
    }
}
```

```
template <typename T>
void swap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}
```

Selectionsort
Beispiel




```
template <typename T>
void selectionSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        swap(arr[min_index], arr[i]);
    }
}
```

Selectionsort
Beweis der Korrektheit

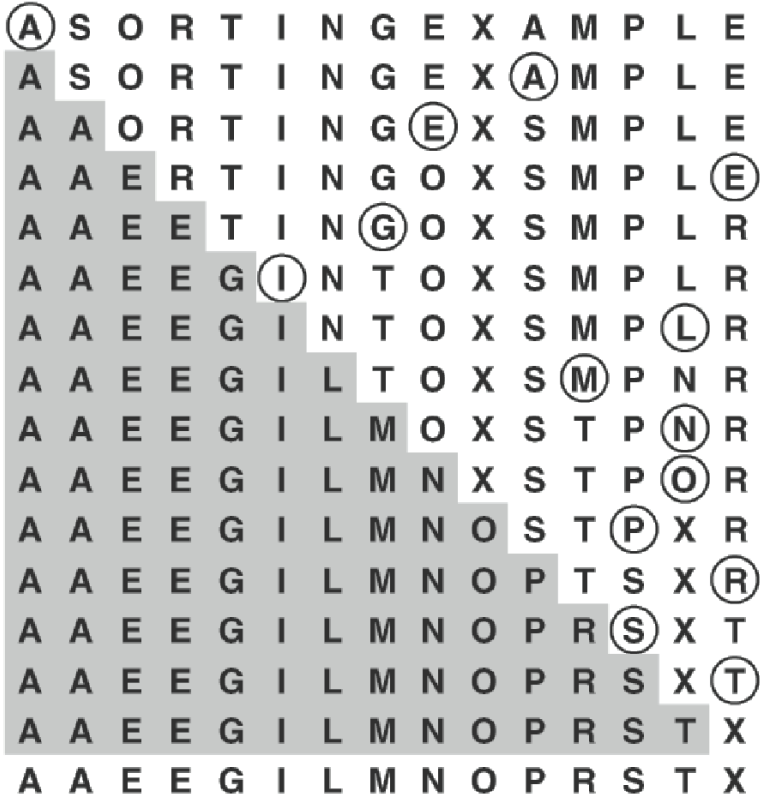
- **Induktionsannahme:**

  Arrays der Länge n−1 können sortiert werden.
- **Induktionsanfang:**

  Arrays der Länge n = 1 sind sortiert.
- **Induktionsschritt:** n−1 ⇝ n :
  - Suche und entnehme das kleinste Element im (Teil-)Array
  - Füge das im ersten Schritt entnommene Element in den sortierten Teilarray hinter der letzten Stelle ein
  - Sortiere den verbleibenden Teilarray mit n−1 Elementen nach Induktionsannahme

```
template <typename T>
void selectionSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        swap(arr[min_index], arr[i]);
    }
}
```

Selectionsort Anschauung



Der erste Durchgang hat in diesem Beispiel keine Wirkung, weil das Array kein Element enthält, das kleiner als das links stehende A *ist. Im zweiten Durchgang ist das andere* A *das kleinste verbliebene Element, sodass es mit* S *an der zweiten Position ausgetauscht wird. Dann wird das* E *in der Mitte mit dem* O *an der dritten Position getauscht, anschließend – im vierten Durchgang – das andere* E *mit dem* R *an der vierten Position usw.*

Aufwand

- Best case
- Average case
- Worst case

```
template <typename T>
void selectionSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        swap(arr[min_index], arr[i]);
    }
}
```

$O(n)$	$O(n \log n)$	$O(n^2)$	$> O(n^2)$
		Best case Average case Worst case	

Runtime Analyse Selectionsort

- In jedem Iteration *i* das Element von *arr*[*i*] mit dem kleinsten Element **tauschen**
- *i* läuft von 0 . . . *n*-1
  - => *n* **Vertauschungen** (Aufrufe von swap)
- In jeder Iteration *i* das **kleinste Element** aus *i* . . . *n-1* **ermitteln**
  - *(n* − 1)− *i* **Vergleiche** pro Iteration

$$(n-1)+(n-2)+(n-3)+\ldots+2+1=\frac{n(n-1)}{2}\approx\frac{n^2}{2}$$

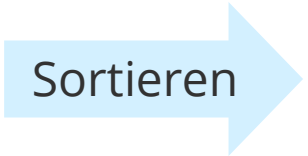
- Anzahl Vergleiche identisch für
  - besten Fall
  - mittleren Fall
  - schlechtesten Fall

```
template <typename T>
void selectionSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        swap(arr[min_index], arr[i]);
    }
}
```

Stabilität von Sortierverfahren

- Relative Reihenfolge gleicher Schlüssel bleibt erhalten.
  - Beispiel:
    alphabetisch geordnete Liste von Personen soll nach Alter sortiert werden

Name	Alter
Endig, Martin	30
Geist, Ingolf	28
Höpfner, Hagen	24
Schallehn, Eike	28



Name	Alter
Höpfner, Hagen	24
Geist, Ingolf	28
Schallehn, Eike	28
Endig, Martin	30

Selectionsort
Stabilität

- Ist Selectionsort stabil?
  - D.h. relative Reihenfolge gleicher Schlüssel bleibt erhalten

```
template <typename T>
void selectionSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        swap(arr[min_index], arr[i]);
    }
}
```

Ja	Nein	Weiß nicht	Kommt drauf an
X			

Überblick

1. Selectionsort
2. **Bubblesort**
3. Insertionsort
4. Shellsort

BubbleSort

Idee:

Verschieden große aufsteigende Blasen („Bubbles“) in einer Flüssigkeit sortieren sich quasi von allein, da größere Blasen die kleineren „überholen“.

Beobachtung:

- Größte Zahl rutscht in jedem Durchlauf automatisch an das Ende der Liste.
- Im Durchlauf *i* Untersuchung bis Position $n - i - 1$

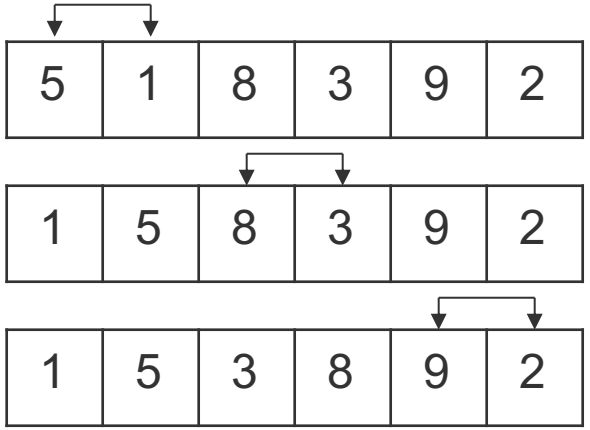
```
template <typename T>
void bubbleSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

```
template <typename T>
void swap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}
```

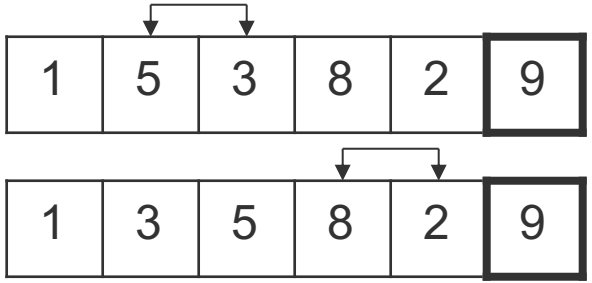
Bubblesort Beispiel

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

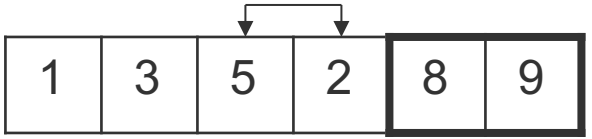
1. Durchlauf



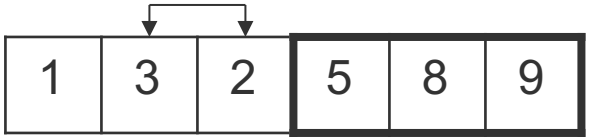
2. Durchlauf



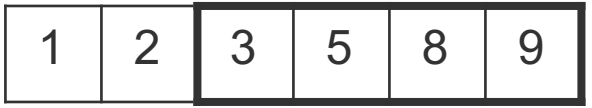
3. Durchlauf



4. Durchlauf



5. Durchlauf




```
template <typename T>
void bubbleSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

BubbleSort

Kleine Verbesserung:

Abbrechen wenn keine Vertauschung mehr geschieht.

```
template <typename T>
void bubbleSort(T arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Stabilität

- Ist Bubblesort stabil?
  - D.h. relative Reihenfolge gleicher Schlüssel bleibt erhalten

```
template <typename T>
void bubbleSort(T arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Ja	Nein	Weiß nicht	Kommt drauf an
X			

Inputvarianten

- Sortierter Input
- Umgekehrt sortierter Input
- Unsortierter Input
- Viele gleiche Werte
- http://www.sorting-algorithms.com/bubble-sort

```
template <typename T>
void bubbleSort(T arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Besonders günstig	Besonders ungünstig	Egal	Günstig
Sortierter Input	Umgekehrt sortierter Input	Unsortierter Input	Viele gleiche Werte

Aufwand

- Best case
- Average case
- Worst case

```
template <typename T>
void bubbleSort(T arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

$O(n)$	$O(n \log n)$	$O(n^2)$	$> O(n^2)$
Best case		Worst case Average case	

Bubblesort Variante: Cocktail sort

- Abwechselnd auf- und absteigend sortieren
  - Nutzt Lokalität der Daten aus
    - Falls Daten nicht in RAM bzw. CPU Cache passen
  - Verschiedene Namen:

    Cocktail shaker sort, bidirectional bubble sort, cocktail sort, shaker sort, ripple sort, shuffle sort, shuttle sort
- Knuth:

  But none of these refinements leads to an algorithm better than straight insertion [that is, insertion sort]; and we already know that straight insertion isn't suitable for large N. [...] In short, the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.

```cpp
template <typename T>
void cocktailSort(T arr[], int n) {
    bool swapped = true;
    int start = 0;
    int end = n - 1;

    while (swapped) {
        swapped = false;

        // Von links nach rechts
        for (int i = start; i < end; ++i) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                swapped = true;
            }
        }

        if (!swapped)
            break;

        swapped = false;

        --end;

        // Von rechts nach links
        for (int i = end - 1; i >= start; --i) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                swapped = true;
            }
        }

        ++start;
    }
}
```

Überblick

1. Selectionsort
2. Bubblesort
3. **Insertionsort**
4. Shellsort

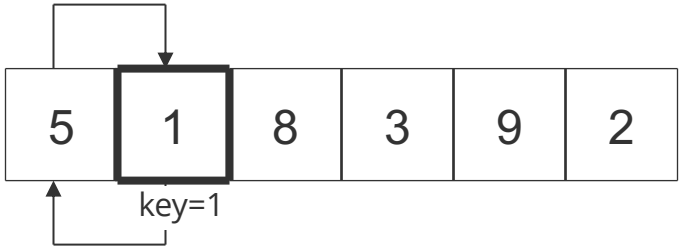
Sortieren durch Einfügen
Insertionsort

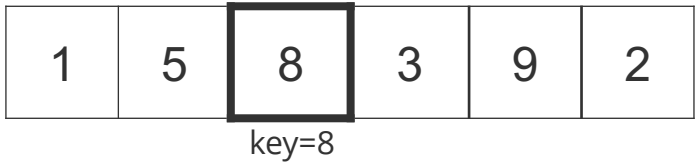
Idee:

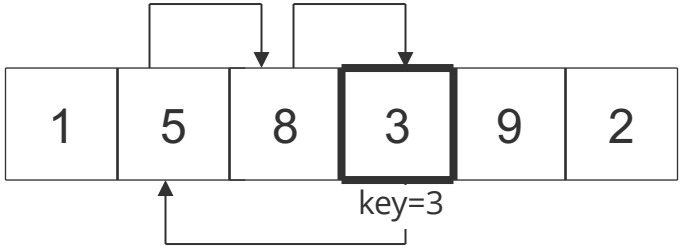
- Umsetzung der typischen (?) menschlichen Vorgehensweise, z.B. beim Sortieren eines Kartenstapels:
  - Starte mit der ersten Karte einen neuen Stapel.
  - Nimm jeweils nächste Karte des Originalstapels und füge sie an der richtigen Stelle in den neuen Stapel ein.

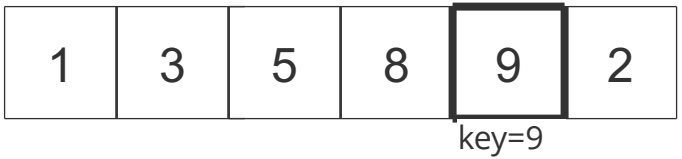
```
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

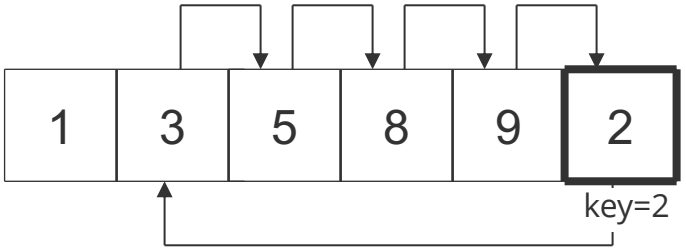
Beispiel Insertionsort












```cpp
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

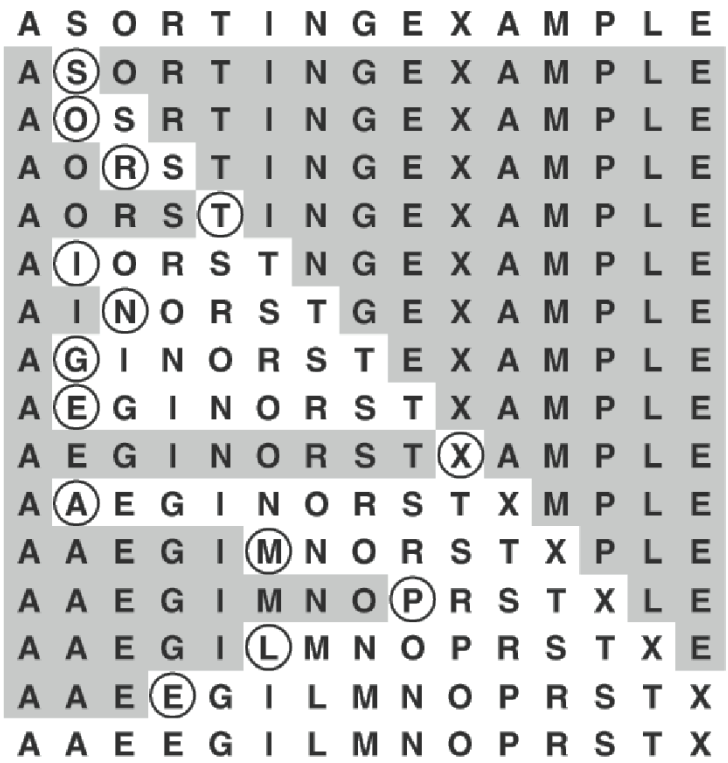
Korrektheit Insertionsort

Induktionsbeweis:

- **Induktionsannahme:** Arrays arr der Länge n−1 können sortiert werden.
- **Induktionsanfang:** n = 1: Arrays der Länge 1 sind sortiert.
- **Induktionsschritt:** n−1 ⇝ n
  - Entnehme dem unsortierten (Teil-)Array von arr das erste Element
  - Füge das im ersten Schritt entnommene Element in den sortierten Teilarray von arr an der richtigen Stelle ein
  - Sortiere den verbleibenden Teilarray mit n−1 Elementen nach Induktionsannahme

```
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Insertionsort: Anschauung



Quelle: Sedgewick, Algorithmen in Java

```cpp
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Stabilität

- Ist Insertionsort stabil?
  - D.h. relative Reihenfolge gleicher Schlüssel bleibt erhalten

```
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Ja	Nein	Weiß nicht	Kommt drauf an
X			

Laufzeit abhängig von Inputvarianten

- Sortierter Input
- Umgekehrt sortierter Input
- Unsortierter Input
- Viele gleiche Werte
- http://www.sorting-algorithms.com/insertion-sort

```
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Besonders günstig	Besonders ungünstig	Egal	Kommt drauf an
Viele gleiche Werte Sortierter Input	Umgekehrt sortierter Input	Unsortierter Input	

Aufwand

- Best case
- Average case
- Worst case

```
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

$O(n)$	$O(n \log n)$	$O(n^2)$	$> O(n^2)$
Best case		Worst case Average case	

Analyse: Insertionsort 1/4
Überblick

- Aufwand
  - Anzahl der Vertauschungen
  - Anzahl der Vergleiche
  - Anzahl Vergleiche dominieren Anzahl Vertauschungen, d.h. es werden mehr Vergleiche als Vertauschungen benötigt
- Außerdem Unterscheidung
  - Bester Fall: Liste ist schon sortiert
  - Mittlerer (zu erwartender) Fall: Liste ist unsortiert
  - Schlechtester Fall: z.B. Liste ist absteigend sortiert

```cpp
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Analyse: Insertionsort 2/4
Best Case

- Wir müssen in jedem Fall alle Elemente $i = 1$ bis $n-1$ durchgehen
  - D. h. immer Faktor $n - 1$
- Für jedes Element zur korrekten Einfügeposition zurückgehen

- Bester Fall: Liste sortiert
  - Einfügeposition ist gleich nach einem Schritt an Position $i - 1$ bei jedem Rückweg 1 Vergleich
  - Gesamtanzahl der Vergleiche: $(n - 1) * 1 = n - 1$
  - Für große Listen abgeschätzt: $n - 1 \approx n$
  - ”Linearer Aufwand”

```
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Analyse: Insertionsort 3/4
Average Case

- Mittlerer (zu erwartender) Fall: Liste unsortiert
  - Einfügeposition wahrscheinlich auf der Hälfte des Rückwegs bei jedem der $n - 1$ Rückwege
    - Faktor $(i - 1)/2$
  - Gesamtanzahl der Vergleiche:

$$\frac{1}{2} + \frac{2}{2} + \cdots + \frac{n-3}{2} + \frac{n-2}{2} + \frac{n-1}{2} = \frac{1+2+\cdots+(n-2)+(n-1)}{2}$$

$$= \frac{1}{2} * n * \frac{(n-1)}{2} = \frac{n*(n-1)}{4} \approx \frac{n^2}{4}$$

```
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

- Mit jedem Vergleich „halbe“ Vertauschung
  - Halb: Nur ein Element wird verschoben; neues Element wird erst am Schluss eingefügt
  - Im Durchschnitt $\frac{n^2}{8}$ Vertauschungen

Analyse: Insertionsort 4/4
Worst Case

- Schlechtester Fall: Liste umgekehrt sortiert
  - Einfügeposition am Ende des Rückwegs bei Position 1
  - D.h. bei jedem der $n - 1$ Rückwege Faktor $i - 1$
  - Analog zu vorhergehenden Überlegungen, aber doppelte Rückweglänge
  - Gesamtanzahl der Vergleiche: $\frac{n*(n-1)}{2} \approx \frac{n^2}{2}$
- $\frac{n^2}{4}$ und $\frac{n^2}{2}$: "quadratischer Aufwand"
  - Konstante Faktoren (1/4 bzw. 1/2) werden nicht berücksichtigt.
- Variante: Einfügestelle mit binärer Suche bestimmen
  - Dadurch nur log i Vergleiche
  - Aber immer noch durchschnittlich $i/2$ Vertauschungen
  - Noch besser: Einfügen in Baumstrukturen (später)

```
template <typename T>
void insertionSort(T arr[], int n) {
    for (int i = 1; i < n; i++) {
        T key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Überblick

1. Selectionsort
2. Bubblesort
3. Insertionsort
4. **Shellsort**

Shellsort

- Nach Donald L. Shell
- CACM 2(7), 1959

IBM type 704 electronic data processing machine

SCIENTIFIC AND BUSINESS APPLICATIONS

A High-Speed Sorting Procedure

D. L. SHELL, *General Electric Company, Cincinnati, Ohio*

```
*  CALLING SEQUENCE
*       SXJ  SORT,4
*       ZER  IA,0,N[a]
*         RETURN
*
SORT    CAL  1,4
        SXD  IA,4
        SXD  B1,1
        STA  S
        STA  A
        STA  Y-1
        STA  IA
        STD  M
        CPL[b]
        ADD  DEC1
        STD  B
C       CLA   M
        ARS   1
        STD  M
        CLA  M
        ZEJ  E
        STD  W
        STD  X
        ADD  B
        STD  K
        CLA  M
        ARS  18
        ADD  IA
        STA  U
        STA  Y
              LXA  M,5
U       CLA  --,4
S       SGA  --,4
        UNJ  V
        UNJ  V
        LDQ  --,4
Y       STQ  --,4
A       STO  --,4
W       RXJ  *+1,4,--
X       HXJ  S,4,--
        LXJ  S,4,0
V       LXA  M,4
        RXJ  *+1,5,-1
K       HXJ  U,4,--
        UNJ  C
E       LXD  IA,4
        LXD  B1,1
        UNJ  2,4
DEC1    ZER  0,0,1
IA      ZER  --,0,--
M       ZER  0,0,--
B       ZER  --,0,--
B1      ZER  0,0,--
*       LAST CARD OF SORT PROGRAM
```

[a] IA is address of first element; N is the number of elements to sort.

[b] Twos complement of N + 1.

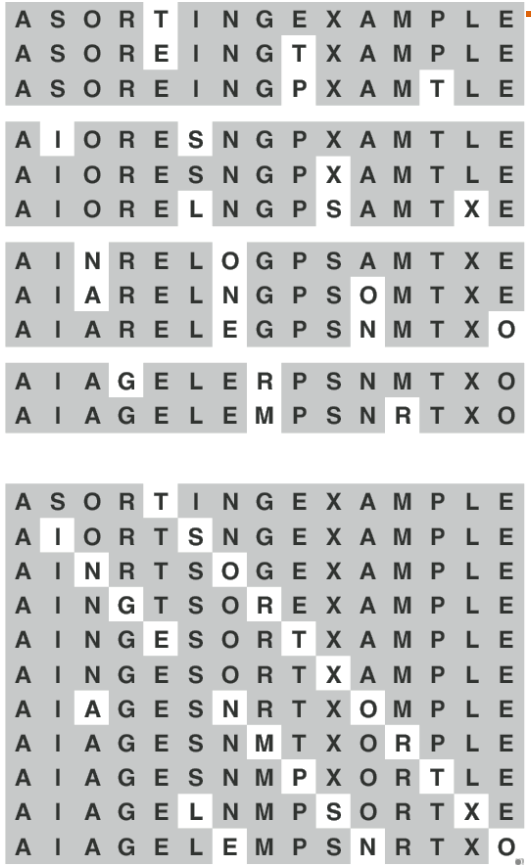
Shellsort

Erweiterung von Insertionsort

- Einfügen eines Elements ineffektiv, weil viele (bis zu n) Elemente bewegt werden müssen
- Zerlegen der Eingabe in h Teile der Größe n/h, separates Sortieren der Teile
  - Dadurch größere Sprünge einzelner Elemente
- Wiederholen für immer kleinere h.
- h-Sortierung: Eingabe ist überlappend in h sortierten Teilen
  - arr[0] <= arr[h] <= arr[2*h] ...
  - arr[1] <= arr[h+1] <= arr[2*h+1] ...
  - ...
- Sortierung zunächst für große Werte von h, danach für immer kleinere Werte
  - h = 1: Insertionsort
  - im Algorithmus: $h_{i+1} = 2 \cdot h_i$ (Originalsequenz nach Shell, 1959)
  - Alternative: $h_{i+1} = 3 \cdot h_i + 1$ (Knuth, 1969)
    - 1 4 13 40 121 364 1093 3280 9841

```cpp
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

Sortieren mit Sprungweite h=4



Der obere Teil der Abbildung zeigt den Ablauf beim 4-Sortieren einer Datei von 15 Elementen: Zuerst wird die Teildatei an den Positionen 0, 4, 8, 12 sortiert, danach die Teildatei an den Positionen 1, 5, 9, 13 , dann die Teildatei an den Positionen 2, 6, 10, 14 und schließlich die Teildatei an den Positionen 3, 7, 11 (jeweils mit Sortieren durch Einfügen). Da die vier Teildateien unabhängig sind, können wir das gleiche Ergebnis erreichen, indem wir jedes Element an seine Position in seiner Teildatei bringen und jeweils um vier Elemente zurückgehen (unten). Entnimmt man aus der oberen Darstellung die erste Zeile aus jedem Abschnitt, dann die zweite Zeile aus jedem Abschnitt usw., gelangt man zur unteren Darstellung.

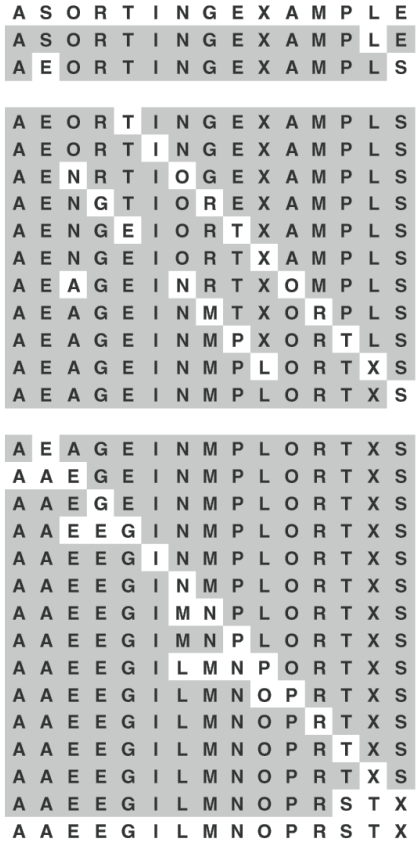
```cpp
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

Insertionsort vs. Shellsort

```cpp
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

```cpp
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

Shellsort Beispiel



Sortiert man eine Datei mit 13-Sortieren (oben), dann 4-Sortieren (Mitte), schließlich 1-Sortieren (unten), sind nicht viele Vergleiche notwendig (wie es die nicht schattierten Elemente anzeigen). Der letzte Durchgang ist lediglich ein Sortieren durch Einfügen, wobei aber kein Element weit zu verschieben ist, weil die beiden ersten Durchläufe bereits eine gewisse Ordnung in die Datei gebracht haben.

```cpp
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```


Quelle: Sedgewick, Algorithmen in Java

Shellsort
Stabilität

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

- Ist ShellSort stabil?
  - D.h. relative Reihenfolge gleicher Schlüssel bleibt erhalten

```
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

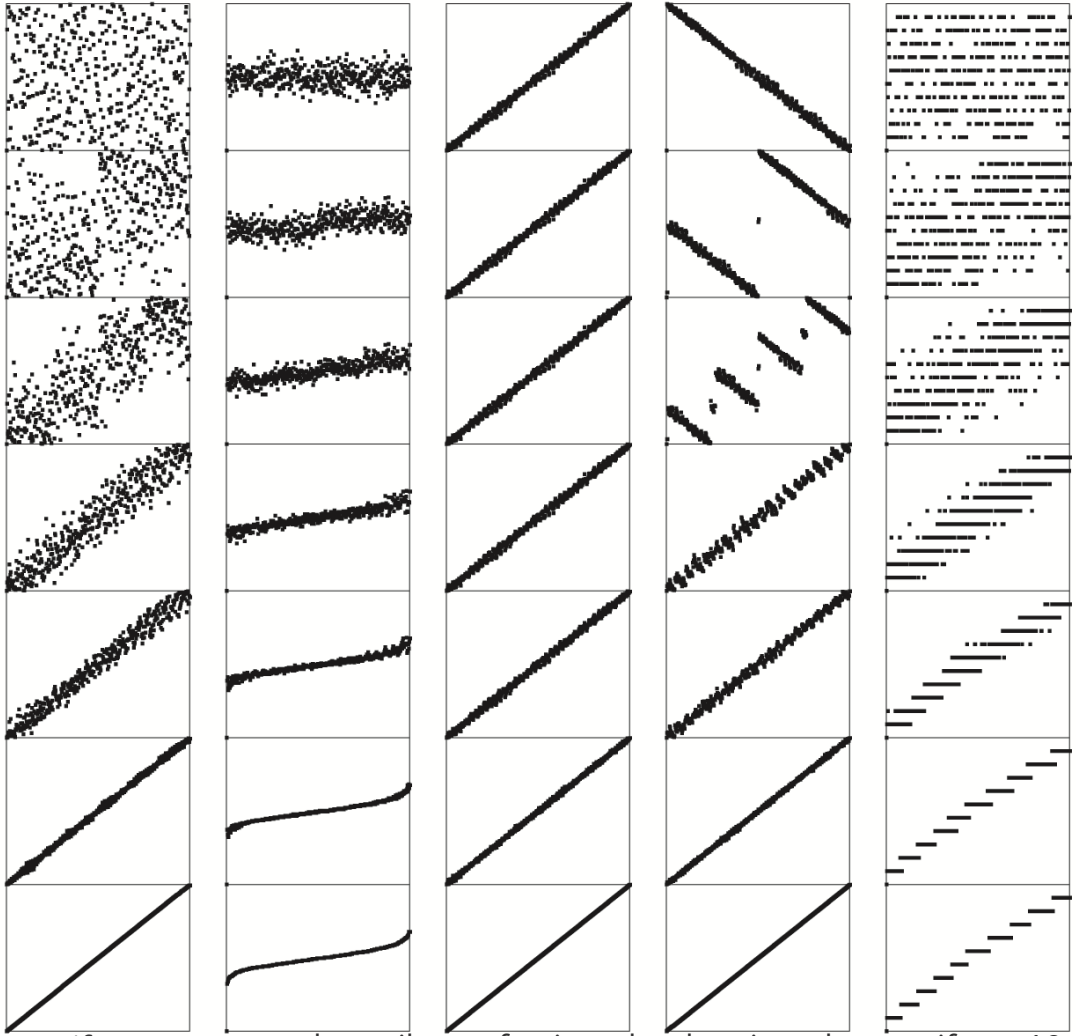
Ja	Nein	Weiß nicht	Kommt drauf an
	X		

Inputvarianten

- Sortierter Input
- Umgekehrt sortierter Input
- Unsortierter Input
- Viele gleiche Werte
- http://www.sorting-algorithms.com/shell-sort

```cpp
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

Besonders günstig	Besonders ungünstig	Egal	Günstig
		Viele gleiche Werte Unsortierter Input	Umgekehrt sortierter Input (worst case ist schlimmer) Sortierter Input (best case)




```java
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

Abbildung 6.15: Dynamische Eigenschaften von Shellsort für verschiedene Arten von Dateien

Diese Diagramme zeigen Shellsort mit den Abständen 209 109 41 19 5 1 für Dateien, die zufällig, Gauß-verteilt, nahezu sortiert, nahezu umgekehrt sortiert und zufällig geordnet mit 10 verschiedenen Schlüsselwerten sind (von links nach rechts, in der oberen Reihe). Die Laufzeit für jeden Durchgang hängt davon ab, wie gut die Datei am Anfang dieses Durchgangs sortiert ist. Nach wenigen Durchläufen sind diese Dateien ähnlich sortiert; folglich ist die Laufzeit nicht sonderlich von den Eingabedaten abhängig.

Quelle: Sedgewick, Algorithmen in Java

Aufwand

- Best case
- Average case
- Worst case

```
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

Denn: O(log n) verschiedene h-Werte in optimaler h-Sequenz

$O(n)$	$O(n \log n)$	$O(n^2)$	$> O(n^2)$
	Best case	Worst case Average case	

Shellsort
Gap (h) Sequenz

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Welche ist die beste Folge für die Größe der Lücke h?

- Originale Folge (von Donald L. Shell, 1959): 1 2 4 8 16 32 64
  - Gut? Nicht so gut? Warum?
  - Ist ineffizient, weil Elemente an geraden und ungeraden Positionen bis zur letzten Runde nie verglichen werden.
- $h_{i+1} = 3*h_i+1$, also 1 4 13 40 … wurde 1969 von Knuth vorgeschlagen
- "Optimale" Folge ist nicht bekannt
  - Exakte Komplexität hängt von Input ab und ist oft nicht bekannt.

```
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

Weitere Ergebnisse für Lückengröße:

- Weniger als $O(n^{3/2})$ Vergleiche für die h-Folge 1 4 13 40 …
- Weniger als $O(n^{4/3})$ Vergleiche für die h-Folge 1, 2, 3, 4, 6, 8, 9, 12, … ($2^p * 3^q$ mit $p, q \in N^0$) „Pratt Sequenz"
- Eine Folge mit O(n log n) average/worst case ist nicht bekannt. (wahrscheinlich unmöglich)
- Eine Folge mit $O(n^{1+e})$ ist prinzipiell möglich.

Weitere Eigenschaften von Shellsort:

- k-sortiert man eine h-sortierte Datei, so ist das Ergebnis sowohl k-sortiert als auch h-sortiert.
- Shellsort benötigt weniger als n(h-1)(k-1)/g Vergleiche, um h- und k-sortierte Daten zu g-sortieren, sofern h und k teilerfremd (ggT=1) sind.

https://en.wikipedia.org/wiki/Shellsort#Gap_sequences

Zusammenfassung Shellsort (nach Sedgewick)

- Bevorzugt, da akzeptable Laufzeiten für große Dateien
- Wenig Code
  - Leicht zu implementieren, leicht zum Laufen zu bringen

- „Wenn Sie ein Sortierproblem schnell lösen wollen und sich nicht mit der Schnittstelle zu einem Systemsortierverfahren herumschlagen wollen, *verwenden Sie Shellsort* und entscheiden Sie später selbst, ob sich der Aufwand lohnen würde, dieses Verfahren durch ein komplizierteres zu ersetzen.“

```cpp
template <typename T>
void shellSort(T arr[], int n) {
    // Starte mit einer großen Lücke und reduziere sie
    for (int h = n / 2; h > 0; h /= 2) {
        // Führe eine gapped Insertion Sort für diese Lücke durch
        for (int i = h; i < n; i++) {
            T temp = arr[i];
            int j;

            for (j = i; j >= h && arr[j - h] > temp; j -= h) {
                arr[j] = arr[j - h];
            }

            arr[j] = temp;
        }
    }
}
```

Überblick

1. Selectionsort
2. Bubblesort
3. Insertionsort
4. Shellsort

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!