Programmiertechnik 2

Unit 8: Elementare Sortieralgorithmen

Ablauf

- Sortieren
  - Elementdarstellung, Primitive Operationen
  - Internes vs. Externes Sortieren
- Bewertung
  - Zahl Vergleiche, Austauschoperationen, Speicherplatzbedarf
- InsertionSort, SelectionSort, BubbleSort
  - Komplexität der einfachen Algorithmen
  - Sortieren von Listen
- ShellSort

Sortieren

- Abstrakte Operation
  - geg: Menge von “items” (Elemente)
  - jedes Element besitzt Sortierschlüssel
  - Schlüssel unterliegen einer Ordnung
  - eventuell sind doppelte Schlüssel erlaubt
- internes Sortieren: Elemente sind im Hauptspeicher
  - nicht geeignet für große Mengen von Schlüsseln
  - Annahme: Zugriff auf Element wahlfrei
    - aber: evtl. Sortieren von Listen
- externes Sortieren: Elemente werden vom Programm eingelesen
  - üblicherweise von Dateien
  - Algorithmus hat gleichzeitig nur wenige Elemente im Speicher

Sortieren (2)

Exkurs: Abstrakte Datentypen

- Primitive Operationen von Elementen
  - less(i1, i2): Ordnungsrelation
- Primitive Operationen des Containers
  - exch(index1, index2): Vertauschen zweier Elemente
  - compExch(index1, index2): Vertauschen zweier Elemente, falls das kleinere dem größeren folgt
    - Vertauschen: tmp=a[index1]; a[index1]=a[index2]; a[index2]=tmp;
- nicht-adaptiver Algorithmus
  - Kontrollfluss hängt nicht von Daten ab
  - einzige Operation: compExch
- adaptiver Algorithmus
  - Kontrollfluss hängt von less() ab

ITEM.java

Sort.java

Sortieren (3)

- Ergebnis des Sortierens: Elemente sind aufsteigend nach Schlüssel sortiert
  - Ziel: Effizientes Auffinden von Elementen anhand des Schlüssels
  - Ziel: Vergleich zweier Mengen auf Gleichheit
- Verhalten bei doppelten Schlüsseln?
  - stabiles Sortieren: gleiche Schlüssel sind hinterher in der gleichen Reihenfolge wie vorher
- Künstliche Stabilisierung: Explizite Integration der ursprünglichen Reihenfolge in Sortierung
  - Elemente werden nummeriert
  - Verglichen werden Paare (k_1, pos_1), (k_2, pos_2)
    - $(k_1, pos_1) < (k_2, pos_2)$ falls $k_1 < k_2$ oder $k_1 == k_2$ und $pos_1 < pos_2$
- "indirektes" Sortieren: Zum Vertauschen von Elementen werden nur Referenzen vertauscht
  - ursprünglich: Vertauschen von Indizes

myItem.java

arraySort.java

myArray.java

Sortieren (4)

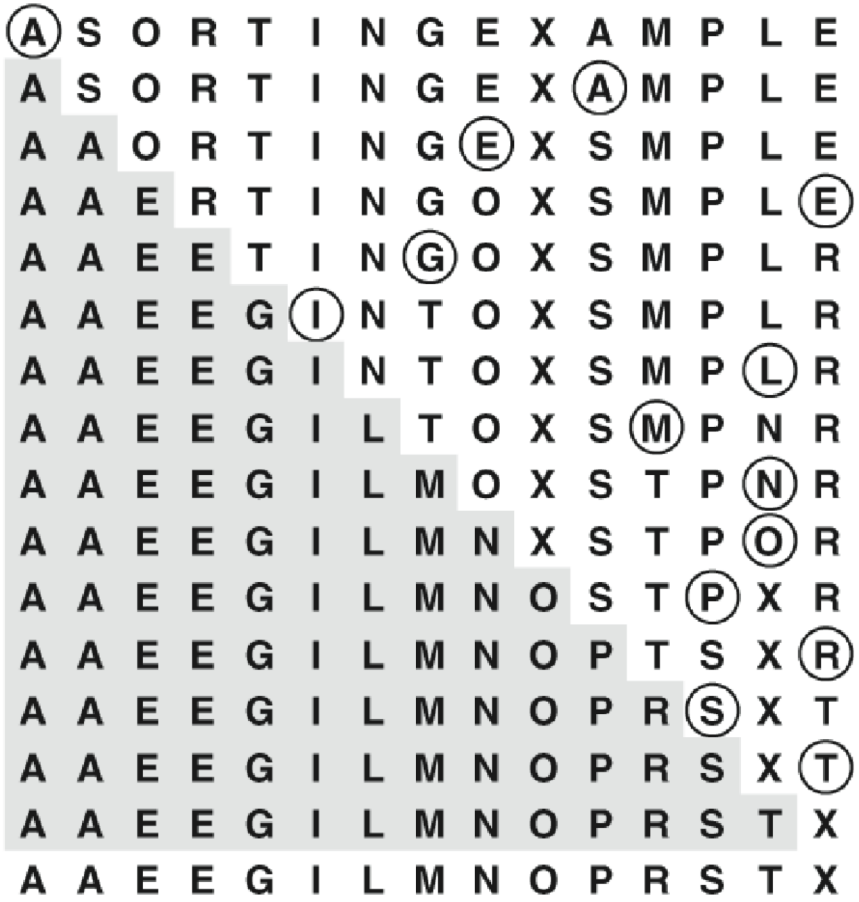
- Leistung eines Algorithmus: Laufzeit und Hauptspeicherverbrauch
- Laufzeit: Zahl der Operationen
  - compExch, less
- Hauptspeicherverbrauch:
  - in-place: Elemente werden im Container umsortiert, ohne zusätzlichen Speicher
  - Sortieren von Listen: eventuell Aufbau für “Rückgrat” einer neuen Liste
    - eventuell gleichzeitige Freigabe des Rückgrats der alten Listen
  - Sortieren mithilfe eines weiteren Felds
    - etwa: indirektes Sortieren

Selection Sort

1. Finde kleinstes Element k_0
2. Vertausche k_0 mit Element 0
   - k_0 ist dann bereits an endgültiger Position
3. Finde nächst-kleinstes Element k
4. Vertausche k mit Element mit nächst-kleinster Position n
5. Wiederhole 3 und 4, bis alle Elemente auf ihrem Platz sind

- Auffinden des minimalen Elements muss stets alle verbleibenden Elemente durchmustern
  - Algorithmus ist nicht-adaptiv, Sortieren einer sortierten Folge ist aufwendig
- Algorithmus ist instabil: vorderes Element wird ohne Berücksichtigung der Originalreihenfolge nach hinten verschoben

Selection Sort – ein Beispiel



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.*

Selection Sort (2)

```
static void selection(ITEM[] a, int L, int R)
   // nach Sedgewick
{
   for (int i = L; i < R; i++)
   {
         int min = i;
         for (int j = i+1; j <= R; j++)
            if (less (a[j], a[min])) min = j;
        exch(a, i, min);
   }
}
```

SelectionSort.java

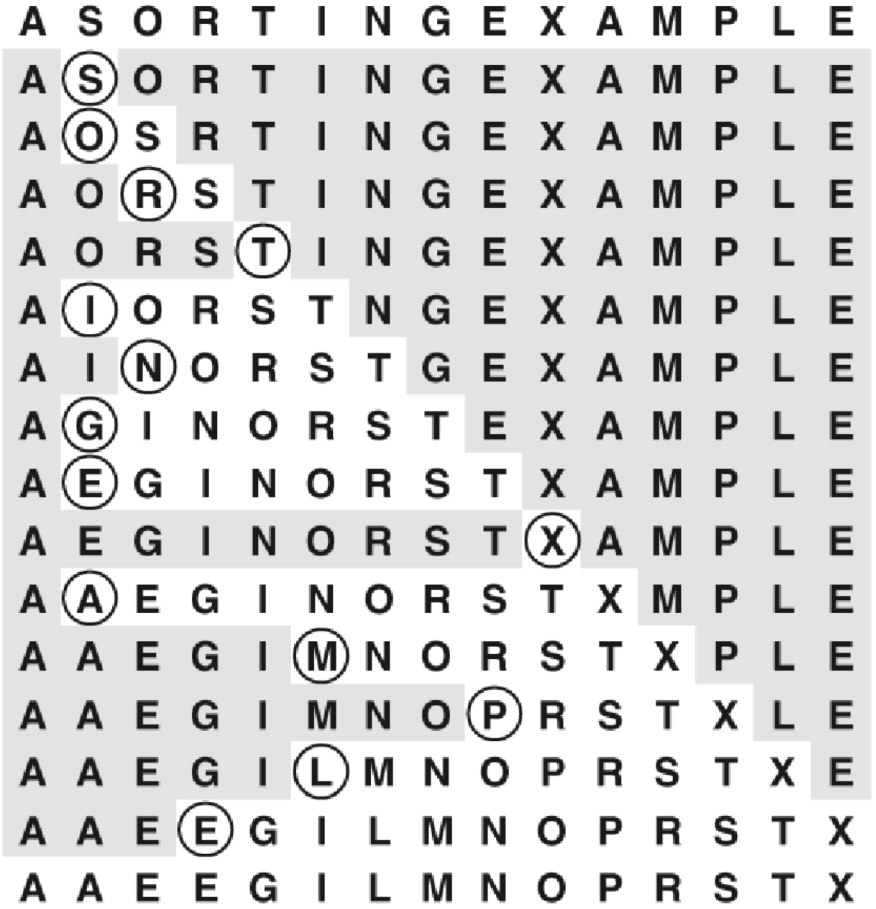
Insertion Sort

- In jedem Schritt ist ein Anfangsstück 0..k sortiert

1. Element an Position k+1 wird an richtige Stelle L (L<k) eingefügt
2. Elemente von L..k werden um 1 nach rechts verschoben
3. Schritte 1 und 2 werden wiederholt bis k=n

- Sortieren ist stabil, falls Element k hinter alle Elemente mit gleichem Schlüssel eingefügt wird
- Adaptiv: Suche nach richtiger Stelle bricht abhängig von Daten ab
  - Optimierung: zuerst kleinstes Element auf Position 0
    - "sentinel" (Wächter): Suche nach richtiger Position muss nicht Index <0 beachten
  - Optimierung: Nach-Rechts-Verschieben ohne paarweises Tauschen

Insertion Sort – ein Beispiel



Im ersten Durchgang beim Sortieren durch Einfügen ist das S an der zweiten Position größer als das A, sodass es nicht verschoben werden muss. Wenn der zweite Durchgang auf das O an der dritten Position trifft, wird es mit dem S getauscht, um A O S in die richtige Reihenfolge zu bringen, usw. Nicht schattierte Elemente ohne Kreis wurden lediglich um eine Position nach rechts geschoben.

Insertion Sort (2)

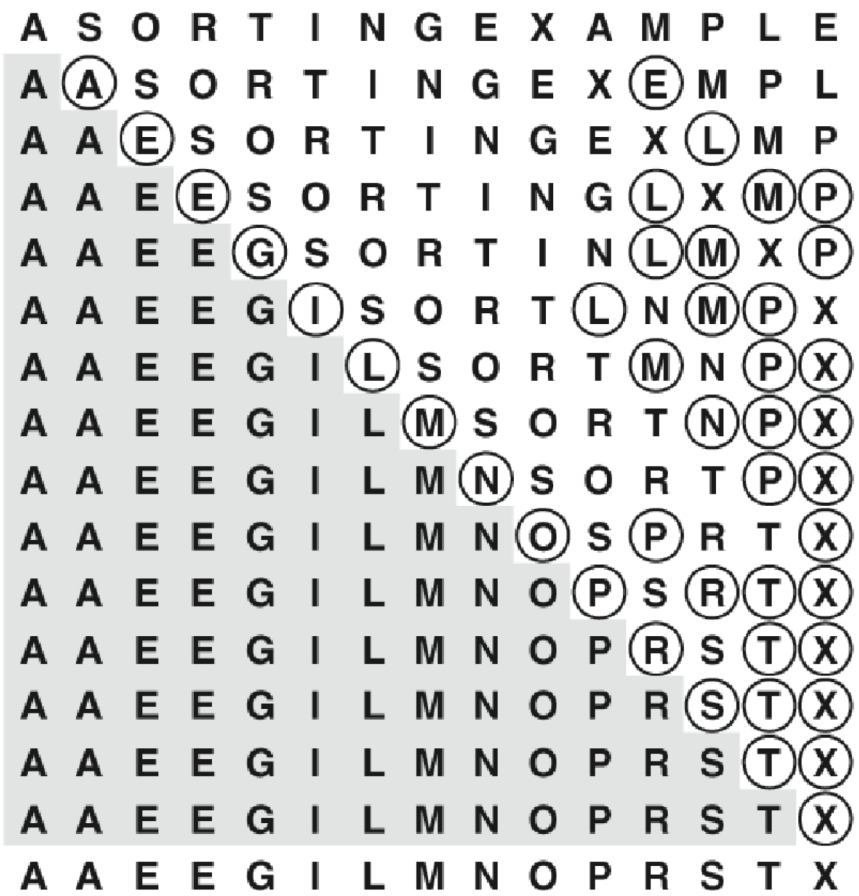
```
static void insertion(ITEM[] a, int L, int R)
{
    int i;
    for (i = R; i > L; i--)
        compExch(a, i-1, i);
    for (i = L+2; i <= R; i++) {
        int j = i; ITEM v = a[i];
        while (less(v, a[j-1])) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = v;
    }
}
```

InsertionSort.java

Bubble Sort

- Sortiere kleine Schlüssel durch paarweises Vertauschen nach vorn
  - “leichtere” Elemente steigen “wie Blasen” nach oben auf
- Runde 1: kleinstes Element auf Position 0
- Runde k: k-kleinstes Element auf Position k-1
- adaptiv? Algorithmus kann abbrechen, wenn keine Vertauschung stattgefunden hat
- stabil, da compExch(k, k) ohne Effekt

Bubble Sort – ein Beispiel



In Bubblesort wandern kleine Schlüssel nach links. Der von rechts nach links voranschreitende Sortiervorgang tauscht jeden Schlüssel mit seinem linken Nachbarn aus, bis ein kleinerer Schlüssel gefunden wird. Beim ersten Durchgang wird das E *mit dem* L*, dem* P *und dem* M *ausgetauscht, bevor die Tauschoperationen beim rechten* A *stoppen; dann bewegt sich dieses* A *an den Anfang der Datei und stoppt beim anderen* A*, das sich bereits an seiner endgültigen Position befindet. Der i-kleinste Schlüssel erreicht seine endgültige Position nach dem i-ten Durchgang, genau wie beim Sortieren durch Auswählen, wobei aber auch andere Schlüssel näher an ihre endgültige Position rücken.*

Bubble Sort (2)

```
static void bubble(ITEM[] a, int L, int R)
{
    for(int i = L; i < R; i++)
       for(int j = R; j > i; j--)
              compExch(a, j – 1, j);
}
```

BubbleSort.java

Komplexität der einfachen Algorithmen

- Selection sort:
  - etwa $N^2/2$ Vergleiche, N Vertauschungen
- Insertion sort:
  - etwa $N^2/4$ Vergleiche, $N^2/4$ Halb-Vertauschungen (Zuweisungen) im Durchschnitt
  - doppelt soviele im schlechtesten Fall
- Bubble sort:
  - etwa $N^2/2$ Vergleiche, $N^2/2$ Vertauschungen
    (sowohl im Durchschnitt wie auch im schlechtesten Fall)

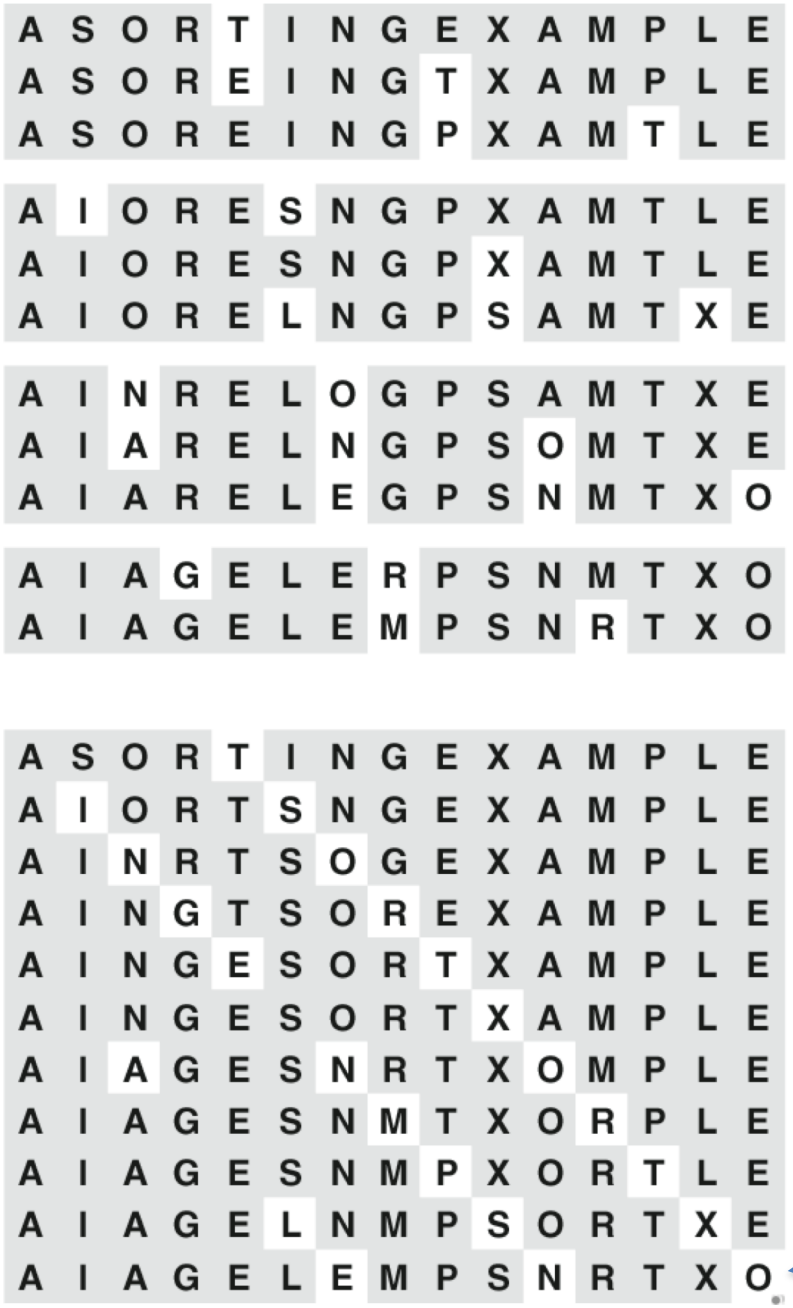
Komplexität (2)

- Vorsortierte Eingaben?
- “Inversion”: Paar von Elementen, die in falscher Reihenfolge sind
- Betrachten Laufzeit in Abhängigkeit von Zahl der Elemente, unter der Annahme, dass Zahl der Inversionen konstant ist
  - “fast-sortierte Daten”
  - Insertion sort und Bubble sort benötigen lineare Zahl von Vergleichen und Vertauschungen
- andere mögliche Annahmen
  - Daten haben eine konstante Zahl von Elementen mit mehr als konstanter Zahl von Inversionen
    - Insertion sort immer noch linear, Bubble sort und Selection Sort nicht mehr

Shellsort

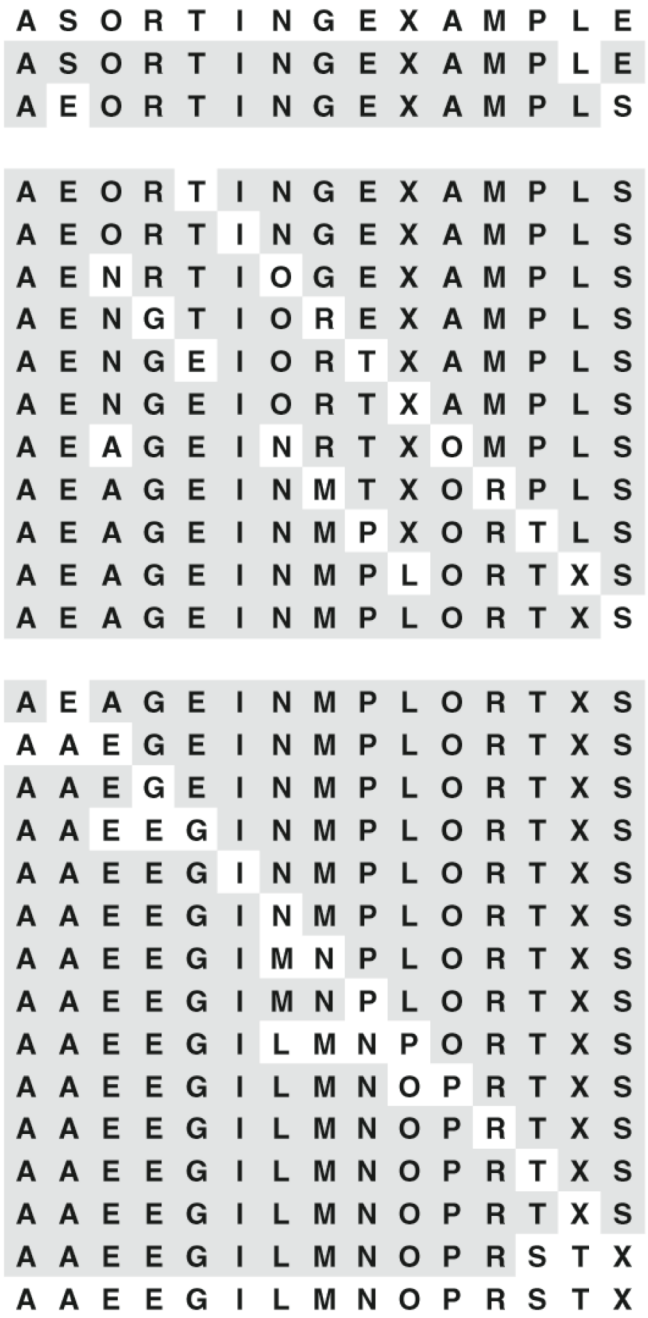
- Erweiterung von Insertion sort:
  - Einfügen eines Elements ineffektiv, weil viele Elemente bewegt werden müssen
  - Idee: Zerlegen der Eingabe in h Teile, separates Sortieren der Teile
- h-Sortierung: Eingabe ist überlappend in h sortierten Teilen
  - a[0] <= a[h] <= a[2*h] ...
  - a[1] <= a[h+1] <= a[2*h+1] ...
  - ...
- Sortierung zunächst für große Werte von h, danach für immer kleinere Werte
  - h = 1: insertion sort
  - Beispielfolge: $h_{n+1} = 3*h_n+1$
    - 1 4 13 40 121 364 1093 3280 9841

Sortieren mit Sprungweite 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 Z… Abschnitt, dann die… jedem Abschnitt us… zur unteren Darste…

Sortieren durch Einfügen der letzten Zeile liefert sortierte Ausgabe



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.

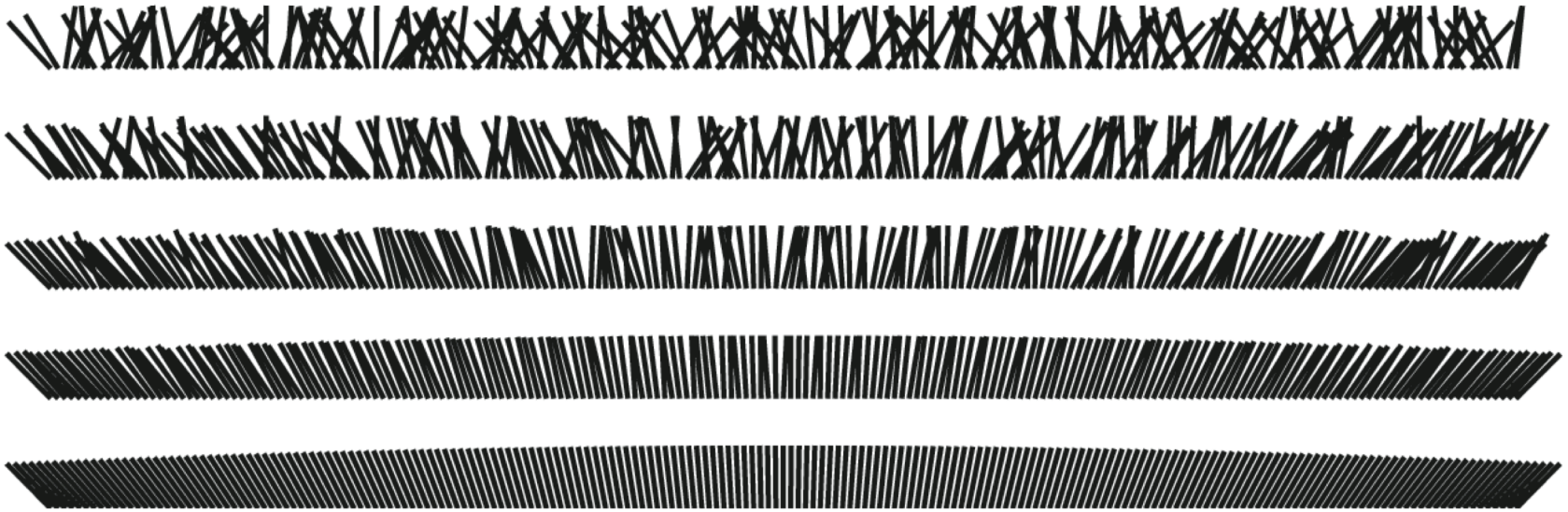
Shellsort (2)

```
static void shell(ITEM[] a, int L, int R)
{   int h;
    for (h = 1; h <= (R-L)/9; h = 3*h+1) /* leer */;
    for (; h > 0; h /= 3)
        // ersetze jede “1” in Insertion sort durch “h”
        for (int i = L+h; i <= R; i++) {
            int j = i; ITEM v = a[i];
            while (j >= L+h && less(v, a[j-h]))
            { a[j] = a[j-h]; j -=h; }
            a[j] = v;
    }
}
```

ShellSort.java

Shellsort veranschaulicht

- Sortieren zufällige Permutation mit Shellsort



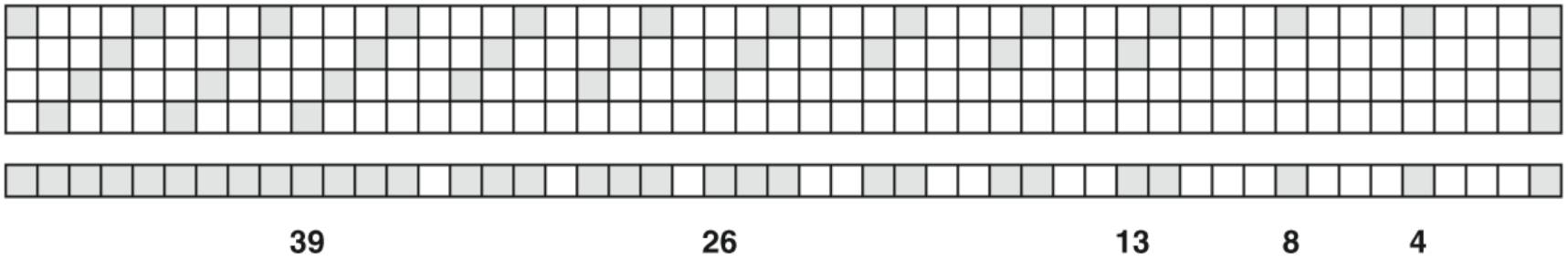
Die Wirkung jedes Durchgangs in Shellsort besteht darin, die Datei als Ganzes näher an die sortierte Reihenfolge zu bringen. Die Datei wird 40-sortiert, dann 13-sortiert, danach 4-sortiert und schließlich 1-sortiert. Jeder Durchgang bringt die Datei näher an die sortierte Reihenfolge.

Shellsort (3)

- Welche ist die beste Folge für h?
  - Originale Folge (von Donald L. Shell, 1959): 1 2 4 8 16 32 64 ... ist ineffizient, weil Elemente an geraden und ungeraden Positionen bis zur letzten Runde nie verglichen werden
  - 1 4 13 40 ... wurde 1969 von Knuth vorgeschlagen
  - “optimale” Folge ist nicht bekannt
    - Exakte Komplexität hängt von der Folge ab und ist oft nicht bekannt
- 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 sind
  - Shellsort benötigt weniger als $O(N^{3/2})$ Vergleiche für die h-Folge 1 4 13 40 ...
  - Folge $4^{i+1} + 3*2^i+1$ (1 8 23 77 281 1073 ...)
    - weniger als $O(N^{4/3})$ Vergleiche

Komplexitätsbetrachtungen

Shellsort benötigt weniger als N(h-1)(k-1)/g Vergleiche, um eine Datei, die h- und k-sortiert ist, mit g-Sortieren zu sortieren, vorausgesetzt, dass h und k teilerfremd sind



Eine 4- und 13-sortierte Datei

Die untere Zeile zeigt ein Array, indem die grauen Elemente kleiner oder gleich dem Element am rechten Rand sein müssen, wenn das Array sowohl 4- als auch 13-sortiert ist. Die vier oberen Zeilen geben das ursprüngliche Muster an. Wenn sich das rechte Element an der Arrayposition i befindet, bedeutet 4-Sortieren, dass Elemente an Arraypositionen i – 4, i – 8, i – 12, … kleiner oder gleich sind (oben); 13-Sortieren bedeutet, dass das Element an i – 13 und aufgrund des 4-Sortierens die Elemente an i – 17, i – 21, i – 25, … kleiner oder gleich sind (zweite Zeile von oben); außerdem sind das Element an i – 26 und aufgrund des 4-Sortierens die Elemente an i – 30, i – 34, i – 38, … kleiner oder gleich (dritte Zeile von oben) usw. Die verbleibenden weißen Quadrate sind diejenigen, die größer sein können als das linke Element; es gibt höchstens 18 solche Elemente (und das am weitesten entfernte Element an i – 36). Somit sind höchstens 18N Vergleiche für ein Sortieren durch Einfügen einer 13-sortierten und 4-sortierten Datei der Größe N notwendig.

Sortieren verketteter Listen

- Problem: Kein wahlfreier Zugriff
  - “bessere” Algorithmen (Quicksort, Heapsort) nicht anwendbar
- Liste selbst soll umgeordnet werden
  - Aufbau einer Ausgabeliste
- Selection Sort:
  - Entfernen des Minimums aus der Eingabeliste, Anhängen an Ausgabeliste
- Insertion Sort:
  - Entfernen des ersten Elements der Eingabeliste
  - Durchsuchen der Ausgabeliste nach nächstgrößerem Element
- Bubble Sort:
  - Rückwärts iterieren: doppelt verkettete Liste?

Zusammenfassung

- Sortieren
  - Elementdarstellung, Primitive Operationen
  - Internes vs. Externes Sortieren
- Bewertung
  - Zahl Vergleiche, Austauschoperationen, Speicherplatzbedarf
- InsertionSort, SelectionSort, BubbleSort
  - Komplexität der einfachen Algorithmen
  - Sortieren von Listen
- ShellSort
- Ausblick
  - Quicksort, Heapsort, Mergesort, Radixsort