

# Programmietechnik II

## Sortieren: Einfache Algorithmen

# 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 Schlüssel
  - 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)

- 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

# 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

# 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-kleinsten 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 (2)

```
static void selection(ITEM[] a, int L, int R)
// nach Sedgwick
{
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);
}
}
```

# 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 (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;
    }
}
```

# 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  $\text{compExch}(k, k)$  ohne Effekt

## 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);
}
```

# 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 so viele 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
  - $a[0] \leq a[h] \leq a[2 \cdot h] \dots$
  - $a[1] \leq a[h+1] \leq a[2 \cdot h+1] \dots$
  - ...
- Sortierung zunächst für große Werte von  $h$ , danach für immer kleinere Werte
  - $h = 1$ : insertion sort
  - Beispielfolge:  $h_{n+1} = 3 \cdot h_n + 1$ 
    - 1 4 13 40 121 364 1093 3280 9841

## 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 (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  $4^{i+1} + 3 \cdot 2^{i+1}$  (1 8 23 77 281 1073 ...)
    - weniger als  $O(N^{4/3})$  Vergleiche



# 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?

# Ausblick

- Quicksort
- Heapsort
- Mergesort
- Radixsort