# Programmiertechnik II

## Sortieren: Quicksort und Mergesort

# Charles Antony Richard Hoare

- Geboren 11. 1. 1934 in Colombo (Sri Lanka)
- Studium in Oxford (Philosophie, Latein, Griechisch) und Moskau (Übersetzung natürlicher Sprache)
- 1960 erste kommerzielle Implementierung von Algol-60
- Professor in Belfast seit 1968, in Oxford seit 1977
- 2000 geadelt “for services to computer science”
- Erfindungen:
  - Quicksort
  - Hoare logic
  - CSP (Communicating Sequential Processes)
- Zitate
  - “Premature optimization is the root of all evil”
  - “There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”

# Quicksort

- C.A.R. Hoare 1960
- Meist-implementierter und meist-studierter Algorithmus
- In-place
  - Aber: Speicherbedarf für Rekursion
- Komplexität im Mittel O(N log N)
- Komplexität im schlechtesten Fall $O(N^2)$

# Quicksort: Grundidee

- Teile-und-herrsche-Algorithmus
- Partitionierung:
  - Zerlege Menge in zwei Teile: Kleiner als Pivot-Element, größer als Pivot-Element
  - Auswahl des Pivotelements: Willkürlich, z.B. letztes Element
  - Pivot-Element wird an seine endgültige Position geschrieben
- Sortierung erfolgt rekursiv:
  - Partitionierung
  - Rekursiv sortieren der kleineren Elemente
  - Rekursiv sortieren der größeren Elemente

# Quicksort: Rekursion

```
static void quicksort(ITEM[] a, int L, int R)
{
   if (R <= L)return;
   int i = partition(a, L, R);
   quicksort(a, L, i-1);
   quicksort(a, i+1, R);
}
```

# Quicksort: Partitionierung

1. Wähle a[R] als Pivot-Element v
2. Suche von links (in i) Element, das größer als v
3. Suche von rechts (in j) Element, das kleiner als v
4. Vertausche a[i] und a[j]
5. wiederhole 2-4, bis i >= j
6. Vertausche Pivot-Element in Grenze zwischen kleinen und großen Elementen

# Quicksort: Partitionierung (2)

```
static int partition(ITEM a[], int L, int R)
{
  int i = L - 1, j = R; ITEM v = a[R];
  for(;;) {
    while (less(a[++i], v));
     while (less(v, a[--j])) if (j == L) break;
     if (i >= j) break;
    exch(a, i, j);
  }
  exch(a, i, R);
  return i;
}
```

# Analyse von Quicksort

- Laufzeit hängt von Wahl des Pivot-Elements ab
- Bester Fall: Pivot-Element teilt Menge in zwei gleiche Teile
  - Partitionierung braucht R-L Vergleiche
  - Rekursionstiefe lg N
  - Gesamtlaufzeit etwa N lg N Vergleiche
- Im Mittel: 2N lg N
- Schlechtester Fall: Menge ist bereits sortiert
  - Pivot-Element ist stets größtes Element
  - Rekursionstiefe ist N
  - Gesamtlaufzeit etwa $N^2/2$ Vergleiche

# Optimierung: Kleine Mengen

- Rekursion und Partitionierung meist langsam für kleine Elementsammlungen
- Ausweg: insertion sort
  - if (R-L <= M) insertion(a, L, R);
  - Wahl von M empirisch, etwa 5 bis 20
- Alternativ: Abbruch, wenn R-L<=M
  - Sortieren des von Quicksort vorsortierten Ergebnisses mit Insertionsort am Ende

# Optimierung: Median-of-Three-Partitionierung

- Problem: Quicksort ist ineffizient wenn Pivot größtes oder kleinstes Element
- Heuristik: Median-of-Three
  - Betrachte a[L], a[R], a[(L+R)/2], wähle mittleres dieser Elemente als Pivot-Element

```
exch(a, ((L+R)/2, R-1);
compExch(a, L, R-1);
compExch(a, L, R);
compExch(a, R-1, R);
int i = partition(a, L, R-1);
```

# Mergesort

- Sortieren durch Mischen
- Mischen (merge): zwei sortierte Folgen werden in eine Folge integriert
- Algorithmus vergleicht kleinste Elemente beider Folgen, wählt kleineres der beiden für Ergebnisfolge
  - ausgewähltes Element wird aus Eingabe entfernt
- evtl. ist eine der beiden Folgen zuerst erschöpft
- zusätzlicher Speicherbedarf: Elemente werden beim Mischen in Zielcontainer kopiert
  - Speicherbedarf O(n)
  - Mischen ohne zusätzlichen Speicher: möglich, kompliziert und komplex

# Mischen

```
static void mergeAB(ITEM[]c, int cL,
                    ITEM[]a, int aL, int aR,
                    ITEM[]b, int bL, int bR)
{
   int i = aL, j = bL;
   for (int k = cL; k <= cL+aR-aL+bR-bL+1; k++) {
    if (i > aR) { c[k] = b[j++]; continue; }
    if (j > bR) { c[k] = a[i++]; continue; }
    c[k] = less(a[j], b[i]) ? a[j++] : b[i++];
   }
},
```

Sedgewicks Version kopiert ein Element zu wenig

Formulierung bei Sedgewick ist nicht stabil

# Mischen im selben Speicher

- Abstrakte Operation
  - static void merge(ITEM[] a, int L, int M, int R);
  - beide Eingabefolgen stehen hintereinander in a (von L..M, M+1..R)
  - Ergebnis steht in a (von L..R)
- Idee: Kopieren aller Elemente in temporären Speicher
- Mischen aus temporären Speicher in Originalspeicher
- stabiles Mischen: gleiche Elemente behalten ihre relative Position (auch wenn sie aus verschiedenen Teilfolgen stammen)
- Zahl der Kopieroperationen: 2 * (R-L)
- Zahl der Vergleiche
  - min (M-L, R-M) im besten Fall
  - R-L im schlechtesten Fall

# Top-Down Mergesort

- Teile-und-herrsche-Algorithmus
  - Teile Eingabe in zwei gleiche Teile
  - Sortiere jeden Teil
  - Mische die Ergebnisse
- Abbruchkriterium: Zahl der Elemente in Eingabe <= 1

# Top-Down Mergesort

```
static void mergesort(ITEM[] a, int L, int R)
{
   if (R <= L) return;
   int M = L+(R-L)/2;
   mergesort(a, L, M);
   mergesort(a, M+1, R);
   merge(a, L, M, R);
}
```

# Analyse von Mergesort

- Benötigt ungefähr N lg N Vergleiche im schlechtesten Fall
- Benötigt zusätztlichen Speicher proportional zu N
- Stabil, falls merge-Operation stabil

# Verbesserung: Mischen ohne Kopieren

- Idee: Temporärer Speicher und Eingabespeicher vertauschen pro Rekursionsschritt ihre Rollen
- Problem: bei naiver Implementierung steht das Ergebnis am Ende im falschen Speicher
  - Lösung: Eingabe wird in beide Speicher kopiert
  - Garantie des richtigen Ausgabespeichers durch Konstruktion des Algorithmus

# Verbesserung: Mischen ohne Kopieren (2)

```
static void mergesortABr(ITEM[] a, ITEM[]b, int L, int R)
{  if (R <= L) return;
   int M = L+(R-L)/2;
   mergesortABr(b, a, L, M);
   mergesortABr(b, a, M+1, R);
   mergeAB(a, L, b, L, M, b, M+1, R);
}
static void mergesort(ITEM[]a, int L, int R)
{  ITEM[] aux = new ITEM[a.length];
   for (int i=L; i <= R; i++) aux[i] = a[i];
   mergesortABr(a, aux, L, R);
}
```

# Bottom-up Mergesort

- nicht-rekursiv
- Folge wird erst in Zweiergruppen sortiert, diese dann in Vierergruppen gemischt usw.
- Algorithmus basiert weiter auf merge()
- Partitionierung der Eingabe ist anders als bei Top-Down-Algorithmus, falls Feldlänge keine Zweierpotenz

# Bottom-up Mergesort

```
static int min(int A, int B)
{
   return (A<B) ? A : B;
}

static void mergesort(ITEM[] a, int L, int R)
{
   if (R <= L) return;
   for (int M = 1; M <= R-L; M = M+M)
    for (int i = L; i <= R-M; i += M+M)
         merge(a, i, i+M-1, min(i+M+M-1, R));
}
```