# Algorithmen

Sortieren

# Problemstellung

- Gegeben ist ein Array von Daten
- Gesucht ist die Umordnung des Arrays anhand eines Sortierkriteriums
- Vereinfacht hier statt Schlüssel und Vergleichsfunktion einfache numerische Ordnung der Werte
- Üblicherweise Aufwand für Vergleiche und Umordnung der Elemente deutlich verschieden

# Selection Sort

- Auswahl des kleinsten/größten Elements
- Vertauschen des ersten/letzten und des kleinsten/größten Elements
- Wiederholung mit dem Restfeld
- O(n) Vertauschungen
- O(n²) Vergleiche

=> beliebt für große Datensätze

# Selection Sort

```
selectionSort :: Ord a => [a] -> [a]
selectionSort [] = []
selectionSort xs = m : selectionSort (r1 ++ r2)
    where
        m = foldl1 min xs
        (r1, (_:r2)) = break (== m) xs
```

# Selection Sort Code

```
void selectionsort(int* data, int len) {
   int i,m,k;
   for(i=0; i<len; i++) { /* von vorn nach hinten */
      for(m=i, k=i+1; k<len; k++)
         if(data[k]<data[m]) /* neues Minimum gefunden? */
            m = k;
      swap(data, i, m); /* Minimum nach vorn tauschen */
   }
}
```

# Insertion Sort

- Einsortieren von Elementen in schon sortierte Teile
- Beginnend links/rechts die Elemente nach links/rechts einsortieren
- $O(n^2)$ Vergleiche
- $O(n^2)$ Vertauschungen
- $O(n)$ bei fast sortierten Mengen

=> Ideal zum Nach- oder Fertigsortieren

# Insertion Sort

```
insertionSort :: Ord a => [a] -> [a]
insertionSort = foldr insert []

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
    | x <= y      = x : y : ys
    | otherwise = y : insert x ys
```

# Insertion Sort Code

```
void insertionsort(int* data, int len) {
   int i,k;
   for(i=len; i-->0; ) { /* von hinten nach vorn */
      for(k=i; k<len-1 && data[k] > data[k+1]; k++)
         swap(data, k, k+1); /* nach hinten schieben */
   }
}
```

# Bubble Sort

- Durchlaufen der Menge und vertauschen zweier benachbarter Elemente, falls nötig
- Abbruch wenn keine Vertauschungen mehr ausgeführt wurden
- $O(n^2)$ Vertauschungen
- $O(n^2)$ Vergleiche
- Höchst ineffizientes Verfahren

# Bubble Sort

```
bubbleSort :: Ord a => [a] -> [a]
bubbleSort [] = []
bubbleSort xs = y : bubbleSort ys
    where
        (y:ys) = bubble xs      -- das kleinste Element ist gefunden
        bubble (x:xs@(_:_))   -- noch zwei Elemente da
                    | x <= y      = x : y : ys
                    | otherwise = y : x : ys
              where (y:ys) = bubble xs
        bubble xs = xs
```

# Bubble Sort Code

```
void bubblesort(int* data, int len) {
   int i,k;
   for(i=0; i<len; i++)
      for(k=len-1; k-->i; ) /* Blasen von hinten nach vorn */
         if(data[k] > data[k+1]) /* blubbern lassen */
            swap(data, k, k+1);
}
```

# Shell Sort

- Insertion Sort über größere Entfernungen
- Unterschiedliche Entfernungsraster anwenden
- Ergebnis ist eine fast sortierte Datei
- Abschließend Insertion Sort mit „1-er Raster“
- Verfahren empfindlich für Rasterabfolge
- Laufzeit gut, aber theoretisch nicht untermauert

=> gutes Allzwecksortierverfahren

# Shell Sort

```
shellSort :: Ord a => [a] -> [a]
shellSort xs
    = foldr hSort xs
    . (1:) . takeWhile ((length xs `div` 3) >=)
    $ iterate (\x -> 3*x+1) 4

hSort :: Ord a => Int -> [a] -> [a]
hSort n = concat . transpose . map insertionSort .
   transpose . groupEvery n
```

# Shell Sort

```
groupEvery :: Int -> [a] -> [[a]]
groupEvery n xs = case splitAt n xs of
    (y,[])  -> [y]
    (y,ys) ->  y : groupEvery n ys
```

# Shell Sort Code

```
void shellsort(int* data, int len) {
   int s,o,i,k;
   for(s=1; s<len; s=3*s+1) {} /* maximale Schrittweite */
   while((s/=3) > 0)
      for(o=0; o<s; o++) /* Alle Streifen */
         for(i=len-o; (i-=s)>=0; ) /* insertionSort mit Schritt */
            for(k=i; k<len-o-s && data[k] > data[k+s]; k+=s)
               swap(data, k, k+s);
}
```

# Distribution Counting

- Beschränkte Wertemenge lassen sich zählen
- Zählung ergibt direkt den Zielplatz des Wertes
- Erfordert umkopieren, jedoch keine Vergleiche
- O(n) Zeitbedarf für alle Fälle
- O(n+m) Platzbedarf – m ist der Wertebereich

=> Ideal zur Erstsortierung unsortierter Bestände

# Distribution Counting Code

```
distributionCounting :: Ix a => [a] -> [a]
distributionCounting = concatMap (\(x,c) -> replicate c x)
                                 . histogramm

histogramm :: Ix a => [a] -> [(a,Int)]
histogramm xs = assocs $ sumup xs [(x,1) | x <- xs]
 where
   sumup :: Ix a => [a] -> [(a,Int)] -> UArray a Int
   sumup xs = accumArray (+) 0 (foldl1 min xs, foldl1 max xs)
```

# Distribution Counting Code

```
void distributioncounting(int* data, int len){
  int i, min, max, *temp, *count;
  for(min=max=data[i=0]; i<len; i++) {
    min = min < data[i] ? min : data[i];
    max = max > data[i] ? max : data[i];
  }
  temp  = malloc(len * sizeof(int));
  count = malloc((max-min+1) * sizeof
    (int));

  for(i=min; i<=max; i++)
    count[i-min] = 0;
  for(i=0; i<len; i++)
    count[(temp[i] = data[i])-min]++;
  for(i=min; i<max; i++)
    count[i+1-min] += count[i-min];
  for(i=len; i-->0;)
    data[--count[temp[i]-min]] = temp[i];

  free(count);
  free(temp);
}
```

# Quicksort

- Zerlegung des Feldes in zwei Teile
  - Alles im ersten Teil ist kleiner als alles im Zweiten
- Rekursive Anwendung auf die beiden Teile
- Ein „Teile und Herrsche“ Verfahren
- O(n*log n) für unsortierte Daten
- $O(n^2)$ bei sortierten Eingaben

=> Äußerst schnelles Erstsortierverfahren

# Quicksort Code

```
quickSort [] = []
quickSort (x:xs) = quickSort ys ++ [x] ++ quickSort zs
    where (ys,zs) = partition (<x) xs
```

# Quicksort Code

```
void quicksort(int* data, int len) {
   int l,r;
   if(len<2)   return;
   for(l=0, r=len-1; l<r; ) {
      while(l<r && data[l]<=data[r]) r--;       if(l<r) swap(data,l,r);
      while(l<r && data[l]<=data[r]) l++;       if(l<r) swap(data,l,r);
   }
   quicksort(data, l);
   quicksort(data+l+1, len-l-1);
}
```

IKS

# Mergesort

- Zerlegen in gleichgroße Teile erzwingen
- Zusammenführen sortierter Teile einfach
- O(n*log n) in allen Fällen
- O(n) Platzbedarf für Umkopieren

=> Sehr schnelles Allzwecksortierverfahren

# Mergesort Code

```
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = merge (mergeSort ys) (mergeSort zs)
  where   (ys,zs) = splitAt (length xs `div` 2) xs

merge [] ys = ys
merge xs [] = xs
merge xs@(x:xs') ys@(y:ys') | x <= y      = x : merge xs' ys
                            | otherwise = y : merge xs ys'
```

# Mergesort Code

```
void mergesort(int* data, int len) {
  int len2, i, l, r, *temp;

  if(len < 2) return;

  len2 = len/2;
  mergesort(data, len2);
  mergesort(data+len2, len-len2);

  temp = malloc(len * sizeof(int));
  for(i=0; i<len2; i++)
    temp[i] = data[i];
  for(; i<len; i++)
    temp[i] = data[(len-1) – (i-len2)];
  /* Maximum steht in der Mitte */
  for(i=0, l=0, r=len-1; i<len; i++)
    data[i] = temp[
      temp[l] < temp[r] ? l++ : r--];
  free(temp);
}
```

# Radix Exchange Sort

- Sortierung der Schlüssel nach fester Stelligkeit
- Höchstwertige Stellen zuerst (wie Quicksort)
- Die entstehenden Bereiche anhand der nächsten Stelle rekursiv sortieren
- Sortierung nach Bits ergibt Binärbaum
- O(n*log n) durchschnittlich
- O(n*bits) im schlimmsten Fall

=> Ähnlich Quicksort, verschiedene Schlüssel

# Radix Exchange Sort Code

```
void radixexchangesort1(int* data, int len, int radix, int mask) {
   int l,r,testbit = 1<<radix;
   if(len<2) return;
   /* Partitionierung anhand des vorgegeben Bits durchführen */
   for(l=0, r=len-1; l<r; ) {
      while(l<r && (mask ^  data[r]) & testbit)     r--;
      while(l<r && (mask ^ ~data[l]) & testbit)     l++;
      if(l<r) swap(data,l,r);
   }
```

# Radix Exchange Sort Code

```
  if(radix>0) { /* Rekursiv weiter sortieren */
    if((mask ^ data[l]) & testbit) { /* Trennstelle zuornden */
      radixexchangesort1(data, l, radix-1, 0);
      radixexchangesort1(data+l, len-l, radix-1, 0);
    } else {
      radixexchangesort1(data, l+1, radix-1, 0);
      radixexchangesort1(data+l+1, len-l-1, radix-1, 0);
    }
  }
}
```

IKS

# Radix Exchange Sort Code

```
void radixexchangesort(int* data, int len) {
   int radix = 8*sizeof(int)-1; /* höchstwertiges Bit */

   /* Höchstwertiges Bit negativer Zahlen == 1
      Deswegen im ersten Aufruf Bits invertieren */
   radixexchangesort1(data, len, radix, ~0);
}
```

# Straight Radix Sort

- Sortierung der Schlüssel nach fester Stelligkeit
- Niederwertigstewertige Stellen zuerst
- Stabilität des Sortierens notwendig
- Distribution Counting pro Stelle (fixer Speicher)
- bits/m Durchläufe bei $O(2^m)$ Speicher
- O(n) bei geschickter Wahl der Basis

=> Praktisch linear für kleine Schlüssel

# Straight Radix Sort Code

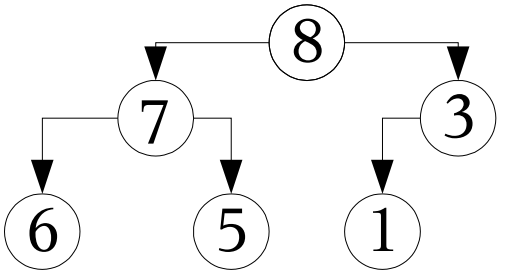
```
void straightradixsort1(int* data, int len, int radix, int mask) {
   int i,count[2] = {0}, *temp = malloc(sizeof(int)*len);
   /* Klassisches Distribution Counting */
   for(i=0; i<len; i++)
      count[((mask ^ (temp[i] = data[i]))>>radix) & 1]++;
   count[1] += count[0];
   for(i=len; i-->0;)
     data[--count[((mask ^ temp[i])>>radix) & 1]] = temp[i];
   free(temp);
}
```

```
void straightradixsort(int* data, int len) {
   int i;
   /* von niedrigen Bits zu höherwertigen */
   for(i=0; i<sizeof(int)*8-1; i++)
       straightradixsort1(data, len, i, 0);
   /* höchstwertiges Bit invertiert behandeln */
   straightradixsort1(data, len, i, ~0);
}
```

# Prioritätswarteschlangen

- Teilsortierte Datenstruktur, die schnell:
  - Erstellen der Struktur aus einem Feld
  - Einfügen eines neues Elements
  - Entfernen des wichtigsten Elements
  - Ersetzen des wichtigsten Elements mit einem Neuen
  - Ändern der Priorität eines beliebigen Elements
  - Löschen eines beliebigen Elements
  - Zusammenfügen zweier Heaps

# Heap




[_,8,7,3,6,5,1,_,_,_,...]
*0 1 2 3 4 5 6 7 8 9 ...*

- Levelorder Zuordnung zwischen Baum und Feld
- i/2 -> Elternknoten
- 2*i und 2*i+1 -> Kindknoten
- Heapbedingung: Elt größer als Kindknoten
- Teilsortiert, deswegen Freiraum für Schnelligkeit

# Heap Code

```
struct Heap {
    int* feld;
    int n; /* Füllstand */
}
upheap(struct Heap* h, int i) {
    while(i>1 &&
      h->feld[i] < h->feld[i/2]) {
        swap(h->feld, i, i/2);
        i /= 2;
    }
}
```

```
insert(struct Heap* h, int neu) {
    h->n++;
    h->feld[h->n] = neu;
    upheap(h, n);
}
```

# Heap Code

```
downheap(struct Heap* h, int i) {
    int j;   /* größter Kindknoten */
    while(j= 2*i; j <= h->n; i=j, j*=2;) {
        if(j < h->n && h->feld[j] < h->feld[j+1])
            j++; /* größter Knoten ist rechts,  nicht links */
        if(h->feld[i] >= h->feld[j])
            break;  /* Abbruch, da nur kleinere Kindsknoten */
        swap(h->feld, i, j);
        i = j;
    }
}
```

# Heap Code

```
int remove(struct Heap* h) {
    int result = h->feld[1];
    swap(h->feld, 1, h->n--);
    downheap(h, 1);
    return result;
}
int replace(struct Heap* h, int v) {
    h->feld[0] = v;
    downheap(h, 0);
    return h->feld[0] ;
}
```

```
heapsort(int* data, int len) {
    struct Heap h = {data-1, len};
    int i;
    for(i = len/2; i > 0; i--) {
        downheap(h, i);
    }
    while(h.n > 0) {
        remove(h);
    }
}
```

# Heap Sort

- Anwendung einer Prioritätswarteschlange
- Trickreiche Inplace Anordnung
- Erweiterbar für externe Sortierverfahren
- Abbruch nach einigen Werten billig möglich
- O(n) für den Aufbau des Heaps
- O(n*log n) für die Sortierung

=> Schnelles (Teil)Sortierverfahren