Inhaltsverzeichnis

1 Sortierverfahren **1**
1.1 Bubblesort . . . 1
1.2 Bucketsort . . . 2
1.3 Selectionsort . . . 2
1.4 Quicksort . . . 3
1.5 Mergesort . . . 3
1.6 Insertionsort . . . 4
1.7 Heapsort . . . 4

1 Sortierverfahren

1.1 Bubblesort

Eigenschaften

- in-place
- worst-case: $\Theta(n^2)$ (absteigend sortierter Input)
- best-case: $\Theta(n)$ (aufsteigend sortierter Input)
- sequentieller Zugriff auf Input

Code

```
void bubbleSort(int[] A) {
  int n = A.length;
  do {
    boolean vertauscht = false;
    for (int i=1; i<n; i++)
      if (A[i] > A[i + 1]) {
        swap(A[i], A[i + 1]);
        vertauscht = true;
      }
    n--;
  } while (vertauscht && n >= 1);
}
```

1.2 Bucketsort

Eigenschaften

- stabil
- Komplexität im best-/worst-case: $\mathcal{O}(n+k)$ mit k = Anzahl Buckets; ist diese konstant, dann $\Theta(n)$
- out-of-place

Code

```
void bucketSort(int[] a, int k) {
  // Histogramm erstellen
  int[] buckets = new int[k];
  for (int i=0; i<n; i++)
    buckets[a[i]]++;
  // sortieren
  int x = 0;
  for (int i=0; i<b; i++)
    while (buckets[i] > 0) {
      a[x++] = i;
      buckets[i]--;
    }
}
```

1.3 Selectionsort

Eigenschaften

- in-place
- lässt sich stabil implementieren
- sequentieller Zugriff auf Input (selten große Seeks)
- Komplexität in $\mathcal{O}(n^2)$, da $n-1$-mal Bestimmung des Minimums und Vertauschung

$$\Rightarrow \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - n \in \mathcal{O}(n^2)$$

Code

```
void selectionSort(int[] a) {
  for (int i=0; i<a.length; i++)
    swap(a, minPos(a, i), i);
}

int minPos(int[] a, int from) {
  for (int i=from; i<a.length; i++)
    if (a[i] < a[from])
      from = i;
  return from;
}
```

1.4 Quicksort

Eigenschaften

- in-place
- kein sequentieller Zugriff, große Seeks (deshalb nicht für Listen geeignet)
- Komplexität:
  - worst-case: Laufzeit in $\mathcal{O}(n^2)$, Rekursionsstackbedarf in $\mathcal{O}(n)$
    - Input auf- oder absteigend vorsortiert
    - Als Pivot wird stets das erste oder letzte Element verwandt
  - average-/best-case: Laufzeit in $\mathcal{O}(n \cdot \log n)$, Rekursionsstackbedarf in $\mathcal{O}(\log n)$
    - Input unsortiert
    - Wahl des Pivot so, dass die jeweils entstehenden Teillisten (ungefähr) gleich lang sind

1.5 Mergesort

Eigenschaften

- stabil
- worst-/average-/best-case: $\mathcal{O}(n \cdot \log n)$
- Speicherbedarf in $\mathcal{O}(n)$
- sequentieller Zugriff auf Input

Code

```
int[] mergeSort(int[] a) {
  if (a.length <= 1)
    return a;
  int m = a.length / 2;
  int[] a1 = new int[m];
  int[] a2 = new int[a.length - m];
  return merge(
      mergesort(a1),
      mergesort(a2));
}

int[] merge(int[] a1, int[] a2) {
  int[] a = new int[a1.length + a2.length];
  int l = 0, r = 0, i = 0;
  while (l < a1.length && r < a2.length) {
    if (a1[l] <= a2[r]) {
      a[i++] = a1[l++];
    } else {
      a[i++] = a2[r++];
    }
  }
  while (l < a1.length)
    a[i++] = a1[l++];
  while (r < a2.length)
    a[i++] = a2[r++];
  return a;
}
```

1.6 Insertionsort

Eigenschaften

- $\mathcal{O}(1)$ zusätzlicher Speicher, Laufzeitkomplexität:
  - best-case: $\mathcal{O}(n)$ (Eingabe sortiert)
  - worst-case: $\mathcal{O}(n^2)$ (Eingabe absteigend sortiert)
- stabil
- braucht nicht alle Eingabedaten zu Beginn

Code

```
void insertionSort(int[] a) {
  for (int i=2; i<n; i++) {
    int t = a[i], j;
    for (j=i; j>1; j--) {
      if (a[j - 1] >= t) break;
      a[j] = a[j - 1];
    }
    a[j] = t;
}
```

1.7 Heapsort

Eigenschaften

- nicht stabil
- in-place
- Komplexität: $\mathcal{O}(n \cdot \log n)$
  - worst-case: Input absteigend sortiert
  - best-case: Input enthält viele gleiche Elemente
- kein sequentieller Zugriff, große Seeks (deshalb nicht für Listen geeignet)

```
void heapSort(int[] a) {
  generateMaxHeap(a);                          // Aufbauphase
  for (int i=a.length-1; i>0; i--) {           // Selektionsphase
    swap(a, i, 0);
    versenke(a, 0, i);
  }
}

void generateMaxHeap(int[] a) {
  for (int i=(a.length/2)-1; i>=0; i--)
    versenke(a, i, a.length);
}

void versenke(int[] a, int i,int n) {
  while (i <= (n / 2) - 1) {
    int kindIndex = ((i+1) * 2) - 1;           // Index des linken Kinds
    if (kindIndex + 1 < n)                     // bestimme ob ein rechtes Kind existiert
      if (a[kindIndex] < a[kindIndex + 1])     // falls ja, gehe zu diesem
        kindIndex++;
    if (a[i] < a[kindIndex]) {
      swap(a, i, kindIndex);                   // Element versenken
      i = kindIndex;                           // wiederhole den Vorgang mit der neuen Position
    } else break;
  }
}
```