AuD - Zusammenfassung

Moritz Gerhardt

Contents

Sektion 1 Was ist ein Algorithmus?

Ein Algorithmus beschreibt eine Handlungsvorschrift zur Umwandlung von Eingaben in eine Ausgabe.
Dabei sollte ein Algorithmus im allgemeinen folgende Vorraussetzungen erfüllen:

1. Bestimmt:
   - Determiniert: Bei gleicher Eingabe liefert der Algortihmus gleiche Ausgabe.
     $\Longrightarrow$ Ausgabe nur von Eingabe abhängig, keine äußeren Faktoren.
   - Determinismus: Bei gleicher Eingabe läuft der Algorithmus immer gleich durch die Eingabe.
     $\Longrightarrow$ Gleiche Schritte, Gleiche Zwischenstände.
2. Berechenbar:
   - Finit: Der Algorithmus ist als endlich definiert. (Theoretisch)
   - Terminierbar: Der Algorithmus stoppt in endlicher Zeit. (Praktisch)
   - Effektiv: Der Algorithmus ist auf Maschine ausführbar.
3. Andwendbar:
   - Allgemein: Der Algorithmus ist für alle Eingaben einer Klasse anwendbar, nicht nur für speziellen Fall.
   - Korrekt: Wenn der Algorithmus ohne Fehler terminiert, ist die Ausgabe korrekt.

Sektion 2 Sortieren

2.1 Sortierproblem

Sortieralgorithmen sind die wohl am häufigsten verwendeten Algorithmen. Hierbei wird als Eingabe eine Folge von Objekten gegeben, die nach einer bestimmten Eigenschaft sortiert werden. Der Algorithmus soll die Eingabe in der richtigen Reihenfolge (nach einer bestimmten Eigenschaft) zur Ausgabe umwandeln. Es wird hierbei meist von einer total geordneten Menge ausgegangen. (Alle Elemente sind miteinander vergleichbar).
Eine Totale Ordnung wie folgt definiert:

Eine Relation $\leq$ auf M ist eine totale Ordnung, wenn:

- Reflexiv: $\forall x \in M : x \leq x$
  (x steht in Relation zu x)
- Transitiv: $\forall x, y, z \in M : x \leq y \wedge y \leq z \implies x \leq z$
  (Wenn x in Relation zu y steht und y in Relation zu z steht, so folgt, dass x in Relation zu z steht)
- Antisymmetrisch: $\forall x, y \in M : x \leq y \wedge y \leq x \implies x = y$
  (Wenn x in Relation zu y steht und y in Relation zu x steht, so folgt, dass x = y)
- Totalität: $\forall x, y \in M : x \leq y \vee y \leq x$
  (Alle Elemente müssen in einer Relation zueinander stehen)

2.2 Insertion Sort

```
1  Function insertion_sort(A):
2      for i = 1 to A.length - 1 do
3          key = A[i]                                  ▷ Element zum Sortieren
4          j = i - 1                                   ▷ Einfügepunkt wird von hinten gesucht
5          while j >= 0 and A[j] > key do
6              A[j + 1] = A[j]                         ▷ Elemente nach Rechts verschieben
7              j = j - 1
8          end
9          A[j + 1] = key                              ▷ Element wird in den Einfügepunkt geschoben
10     end
11 end
```

Prinzip: Die Eingabe wird von links nach rechts durchlaufen. Dafür wird für jedes Element

2.3 Merge Sort

```
1  Function mergeSort(A, left, right):
2  |  if left < right then
3  |  |        ▷ Wenn Bereich ist nicht leer mid = floor((left + right) / 2)        ▷ Nach unten gerundet
4  |  |  mergeSort(A, left, mid)                                  ▷ Sortiert von left zu mid
5  |  |  mergeSort(A,mid + 1, right)                              ▷ Sortiert von mid + 1 zu right
6  |  |  mergeA, left, mid, right                                                  ▷ Fügt Hälften zusammen
7  |  end
8  end
```

```
1  Function merge(A, left, mid, right):
2  |  B = new Array[right - left + 1]                                  ▷ Temp array
3  |  p = left
4  |  q = mid + 1
5  |           ▷ Array A wird mithilfe von zwei Pointern durchgegangen, von links und von mid + 1
6  |  for i = 0 to right - left do
7  |  |                                  ▷ Läuft für jedes Element im Zielbereich
8  |  |  if q > right or (p =< mid and A[p] =< A[q]) then
9  |  |  |  ▷ Wenn q > right, dann ist der rechte Teil durchlaufen. Wenn p =< mid ist, so ist der linke Teil
   |  |  |    noch nicht durchlaufen. Wenn das linke Element =< dem rechten ist, dann wird das Element
   |  |  |    dem temp array B hinzugefügt.
10 |  |  |  B[i] = A[p]
11 |  |  |  p = p + 1
12 |  |  end
13 |  |  else
14 |  |  |                                  ▷ Wenn oben nicht zutrifft:
15 |  |  |  B[i] = A[q]
16 |  |  |  q = q + 1
17 |  |  end
18 |  end
19 |  for i = 0 to right - left do
20 |  |                              ▷ Läuft wieder für jedes Element im Zielbereich
21 |  |  ▷ Kopiert den jetzt sortierten temp array B zurück in den eigentlichen Array an der richtigen Stelle
22 |  |  A[i + left] = B[i]   ▷ i + left, damit nicht am Anfang sondern an dem richtigen Teilbereich eingefügt
   |  |    wird.
23 |  end
24 end
```

2.4 Quicksort

```
1  Function quicksort(A, left right):
2  |  if left < right then
3  |  |  q = partition(A, left, right)
4  |  |  quicksort(A, left, q)
5  |  |  quicksort(A, q + 1, right)
6  |  end
7  end
```

```
1  Function partition(A, left, right):
2  |  pivot = A[left]
3  |  p = left - 1
4  |  q = right + 1
5  |  while p < q do
6  |  |  while A[p] < pivot do
7  |  |  |  p = p + 1
8  |  |  end
9  |  |  while A[q] > pivot do
10 |  |  |  q = q - 1
11 |  |  end
12 |  |  if p < q then
13 |  |  |  temp = A[q]
14 |  |  |  A[q] = A[p]
15 |  |  |  A[p] = temp
16 |  |  end
17 |  end
18 |  return q
19 end
```

2.5 Radix Sort

```
keys = digits d in range [0, D-1]                          ▷ possible digits
B = new Array[0]                                           ▷ Buckets, initially empty
Function radixSort(A):
    for i = 0 to d - 1 do
                                                           ▷ From least to most significant for j = 0 to n - 1 do
        putBucket(A, B, i, j)
    end
    a = 0
    for k = 0 to D - 1 do
        for b = 0 to B[k].size - 1 do
            A[a] = B[k][b]                                 ▷ Read bucket in order
            a = a + 1
        end
        B[k].size = 0                                      ▷ Clear Bucket
    end
end
```

```
Function putBucket(A, B, i, j):
    z = A[j].digit[i]                                      ▷ i-th digit of A[j]
    b = B[z].size                                          ▷ Size corresponds to next free spot
    B[z][b] = A[j]
    B[z].size = B[z].size + 1
end
```