# Repetitorium 23.03.2021

Montag, 22. März 2021 16:45

## Rekursive Laufzeiten

### Rekusionsgleichung aufstellen

**Algorithmus 7:** int Fakulaet(int $x$)

```
1 if x == 0 then
2     return 1
3 return x · Fakultaet(x − 1)
```

Rekursionsgleichung:

1. Laufzeit des Basisfalls feststellen. -> O(1)
2. Wie oft wird die Funktion aufgerufen? -> 1x
3. Wie verändert sich das Argument der Funktion? -> n - 1

=> T(n) = 1 * T(n-1) + 1

T(n) in O(n) ? Wie lösen wir die Gleichung?

Beispiel:

```
f( int k)
if( k = 0){
        Return 1
}
int z = 0;
For(i = 0; i < k; i++){
        z += f(k)
}
Return z
```

T(n) = Summe von i = 0 bis n über T(i) + 1

### Substitutionsmethode

1. Laufzeit raten
2. Beweis durch vollständige Induktion (oder andere Beweismethoden)

Bsp.
T(n) = 3 T(n/4) + n^2

Rate: T(n) in O(n^2).

Induktionsanfang: T(1) <= c * 1^2 für n = 1 und ein c > 0.
Wahr, denn wir können c = |T(1)| wählen. Besser: Wähle c = Max(|T(1)|, 2)
Induktionsannahme: Für ein festes, aber beliebiges i in N gilt T(i) <= c i^2 für alle j < i
Induktionsschritt:
T(i) = 3 T(i/4) + i^2 <= 3 c (i/4)^2 + i^2 = c * 3/16 * i^2 + i^2 = (1 + 3c/16) i^2 <= c*i^2

Dies wäre wahr, wenn 1 + 3c/16 <= c
1 <= 13/16 c
c >= 16/13
Nachtrag

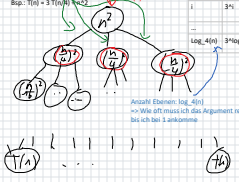
=> Aussage ist wahr, Induktion abgeschlossen
Für alle i in N gilt T(i) <= c * i^2. Damit ist T(n) in O(n^2).

Ratetipps: T(n) = T(n/2) + f(n) => T(n) in O(log n * f(n))
T(n) = T(n-k) + f(n) => T(n) in O(k * f(n))

### Rekursionbaummethode

1. Rekursive Aufrufe als Baum darstellen
2. Beiträge von Knoten + Ebenen berechnen
3. Basisfall T(1) beachten
4. Über alle Ebenenbeiträge summieren

Bsp.: T(n) = 3 T(n/4) + n^2




Anzahl Ebenen: log_4(n)
=> Wie oft muss ich das Argument reduzieren, bis ich bei 1 ankomme

T(1) <- k

| Ebene | Anzahl Knoten | Beitrag Knoten | Beitrag Ebene |
|---|---|---|---|
| 0 | 1 | n^2 | n^2 |
| 1 | 3 | (n/4)^2 | 3 * (n/4)^2 |
| 2 | 9 | (n/16)^2 | 9 * (n/16)^2 |
| ... | | | ... |
| i | 3^i | (n/4^i)^2 | 3^i * (n/4^i)^2 = (3/16)^i * n^2 |
| ... | | | ... |
| Log_4(n) | 3^log_4(n) | k (Laufzeit des Basisfalls ist konstant) | k * 3^log_4(n) |

| Ebene | Anzahl Knoten | Beitrag Knoten | Beitrag Ebene |
|---|---|---|---|
| 0 | 1 | n^2 | n^2 |
| 1 | 3 | (n/4)^2 | 3 * (n/4)^2 |
| 2 | 9 | (n/16)^2 | 9 * (n/16)^2 |
| ... | | | ... |
| i | 3^i | (n/4^i)^2 | 3^i * (n/4^i)^2 = (3/16)^i) * n^2 |
| ... | | | ... |
| Log_4(n) | 3^log_4(n) | k (Laufzeit des Basisfalls ist konstant) | k * 3^log_4(n) |

$$T(n) = k * 3^{\log_4(n)} + \sum_{i=0}^{\log_4(n)-1} (3/16)^i) * n^2$$

= k * 3^log_4(n) + n^2 * $\sum_{i=0}^{\log_4(n)-1} (3/16)^i$

Funktioniert für Summe über q^i mit |q| < 1

<= k * 3^log_4(n) + n^2 * 1/(1 - 3/16)

= k * 3^log_4(n) + 16/13 n^2

= 16/13 n^2 + k * 3^(log_3(n) / log_3(4))
= 16/13 n^2 + k * n^(1/log_3(4))
<= 16/13 n^2 + k * n^2 in Theta(n^2)

=> T(n) in Theta(n^2)

### Meistermethode

Einschränkung: Gilt nur für T(n) = a T(n/b) + f(n)
z.B. nicht bei T(n) = 3 T(n/2) + 2 T(n/3) + n^2

1. Log_b(a) ausrechnen
2. Bestimmen, ob f in O(...), Theta(...), Omega(...). epsilon explizit angeben.
3. Evtl. Regularitätsbedingung prüfen
4. Fall explizit hinschreiben!!!!!!!

Bsp.: T(n) = 3 T(n/4) + n^2

1. Log_b(a) = log_4(3) ~ 0.79
2. f(n) = n^2 in Omega(n^(0.79 + e)) z.B. für e = 1
3. Regularitätsbedingung:
   3 f(n/4) < c f(n)
   3/16 n^2 < c n^2
   Korrekt für z.B. c = 8/16 => t.r.fällt
4. 3. Fall ist erfüllt, T(n) in Theta(f) = Theta(n^2)

Dann gilt

$$T \in \begin{cases} \Theta(n^{\log_b a}) & \text{falls } f \in O(n^{(\log_b a)-\varepsilon}) \text{ für ein } \varepsilon > 0. \\ \Theta(n^{\log_b a} \log n) & \text{falls } f \in \Theta(n^{\log_b a}). \\ \Theta(f) & \text{falls } f \in \Omega(n^{(\log_b a)+\varepsilon}) \text{ für ein } \varepsilon > 0 \\ & \textit{und} \text{ die Regularitätsbedingung gilt.} \end{cases}$$

**Definition:** Die *Regularitätsbedingung* ist erfüllt, falls
$af(n/b) \leq cf(n)$
für ein $c < 1$ und für alle großen $n$.

## Sortiermethoden

### InsertionSort

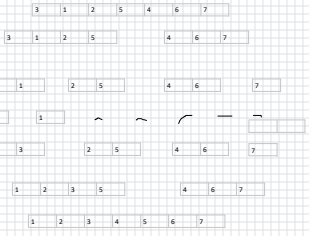
```
InsertionSort(int[] A)
  for j = 2 to A.length do
      key = A[j]
      i = j − 1
      while i > 0 and A[i] >
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

| | | | | | |
|---|---|---|---|---|---|
| 3 | 1 | 2 | 5 | 4 | 6 |
| 1 | 3 | 2 | 5 | 4 | 6 |
| 1 | 2 | 3 | 5 | 4 | 6 |
| 1 | 2 | 3 | 5 | 4 | 6 |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 2 | 3 | 4 | 5 | 6 |

## MergeSort

```
MergeSort(int[] A, int ℓ = 1, int r = A.length)
  if ℓ < r then
    m = ⌊(ℓ + r)/2⌋            } teile
    MergeSort(A, ℓ, m)         } herrsche
    MergeSort(A, m + 1, r)     } herrsche
    Merge(A, ℓ, m, r)          } kombiniere
```

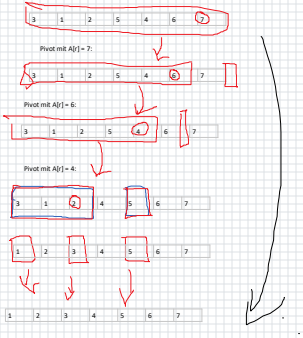
```
Merge(int[] A, int ℓ, int m, int r)
  n1 = m − ℓ + 1;  n2 = r − m
  L = new int[1..n1 + 1]; R = new int[1..n2 + 1]
  L[1..n1] = A[ℓ..m]
  R[1..n2] = A[m + 1..r]
  L[n1 + 1] = R[n2 + 1] = ∞
  i = j = 1
  for k = ℓ to r do
    if L[i] ≤ R[j] then
      A[k] = L[i]
      i = i + 1
    else
      A[k] = R[j]
      j = j + 1
```



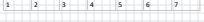
## QuickSort

```
QuickSort(A, ℓ = 1, r = A.length)
  if ℓ < r then
    m = Partition(A, ℓ, r)
    QuickSort(A, ℓ, m − 1)
    QuickSort(A, m + 1, r)
```

```
int Partition(A, ℓ, r)
  pivot = A[r]
  i = ℓ
  for j = ℓ to r − 1 do
    if A[j] ≤ pivot then
      Swap(A, i, j)
      i = i + 1
  Swap(A, i, r)
  return i
```




WorstCase:



## Vergleich

| Algo | Laufzeit | Best | Worst | in-situ | Stabil |
|---|---|---|---|---|---|
| InsertionSort | Theta(n^2) | Theta(n) | Theta(n^2) | + | + |
| MergeSort | Theta(n log n) | Theta(n log n) | Theta(n log n) | - | + |
| QuickSort | Theta(n log n) | Theta(n log n) | Theta(n^2) | + | - |
| HeapSort | Theta(n log n) | Theta(n log n) | Theta(n log n) | + | - |

## CountingSort

```
CountingSort(int[] A, int[] B, int k)
  sei C[0..k] = ⟨0, 0, . . . , 0⟩ ein neues Feld
  for j = 1 to A.length do C[A[j]] = C[A[j]] + 1   // (1a)
  // C[i] enthält jetzt die Anz. der Elem. gleich i in A
  for i = 1 to k do C[i] = C[i] + C[i − 1]         // (1b)
  // C[i] enthält jetzt die Anz. der Elem. ≤ i in A
  for j = A.length downto 1 do
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1                          // (2)
```

Eingabefeld
Ausgabefeld
beschränkt Universum $\{0, \ldots, k\}$

n Zahlen in einer k-elementigen Menge aus Z (z.B. [0..k-1], [-20, ..., k-21], ... (evtl. Indexverschiebung)

Laufzeit: O(n + k)

Bsp.: n = 1.000.000 und k = 10 => deutlich effizienter als Merge/QuickSort
Bsp.: n = 10 und k = 2000 => zwar immer noch linear, aber praktisch inefffizienter (Speicherplatz, Laufzeit) als Merge/QuickSort

## RadixSort

Funktioniert für s-stellige und b-adische Zahlen (b-adisch = in Basis b)

=> Sortiere nach einander und stabil nach den einzelnen Stellen
=> b begrenzt das Universum jeder Stelle, d.h. durch CountingSort kann eine lineare Laufzeit erreich werden

## BucketSort

Sortieralgorithmus mit Laufzeit n log n => Sort
Anzahl an Zahlen, ist die Laufzeit in O(1).

n Zahlen. Teile diese in n Teilintervalle, dann
im Durchschnitt 1 Zahl. Die Teilintervalle könn
sortieren. Dann hänge die sortierten Teilinter

Einschränkungen: Die Zahlen müssen (annähe

```
BucketSort(Feld A von Z
  n = A.length
  lege Feld B[0..n − 1] v
  for j = 1 to n do
    füge A[j] in Liste B
  for i = 0 to n − 1 do
    sortiere Liste B[i]
  hänge B[0], . . . , B[n −
  kopiere das Ergebnis n
```

## Datenstrukturen

### Array

Operatoren: length, []

Feste Länge -> Zusammenhängender Speicherabschnitt

List[3] existiert nicht!

Speicher:

### Stapel

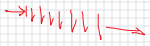
LIFO-Prinzip: Last-in-First-Out

Operationen:
- Empty()
- Push(key k)
- Pop()
- Top()/Peek()

Alle in O(1)

### Schlange

FIFO-Prinzip: First-in-First-Out

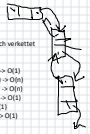


Operationen
- Empty()
- Enqueue(key k)
- Dequeue
- First()/Last()

Alle in O(1)

### Liste

Doppelt bzw. einfach verkettet



Operationen:
- Insert(key k) -> O(1)
- Search(key k) -> O(n)
- Delete(key k) -> O(n)
- Delete(ptr x) -> O(1)
- Empty() -> O(1)
- Evtl. Head() -> O(1)

### Heap

Prioritätsschlange: Schlange, aber sortiert nach Priorität

Operationen
- Insert() -> O(1) -> mit Heap: O(log n)
- FindMax -> O(1) -> mit Heap: O(1)
- ExtractMax -> O(n) -> O(log n)
- IncreaseKey -> O(n) -> O(log n)

Heap: Eine Datenstruktur, bei der die Anordnung der Elemente gewissen Bedingungen genügt:

left.key < parent.key, right.key < parent.key

Left(i) = 2i, right(i) = 2i + 1, parent(i) = floor(i/2)

**BuildMaxHeap**(int $A[]$)
- $A.heap\text{-}size = A.length$
- **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1 **do**
  - MaxHeapify($A, i$)

**MaxHeapify**(int $A[]$, index $i$)
- $\ell = \text{left}(i)$; $r = \text{right}(i)$
- **if** $\ell \leq A.heap\text{-}size$ **and** $A[\ell] > A[i]$ **then**
  - $largest = \ell$
- **else** $largest = i$
- **if** $r \leq A.heap\text{-}size$ **and** $A[r] > A[largest]$
  - $largest = r$
- **if** $largest \neq i$ **then**
  - swap($A, i, largest$)
  - MaxHeapify($A, largest$)


### HeapSort

HeapSort(int[] A
- BuildMaxHeap
- **for** $i = A.leng$
  - $A[i] = Ext$

=> Laufzeit ist n * T(Extract

ahlen in $[0, 1)$)

on Listen an

$[\lfloor n \cdot A[j] \rfloor]$ ein

$= [\frac{i}{n}, \frac{i+1}{n}) \cap A$

1] aneinander

ach $A[1..n]$

iert man eine feste

ist in jeder Teilintervalle
en wir in konstanter Zeit
alle aneinander

rnd) gleichverteilt sein

Hashing

| 31 | 11 | 21 | 35 | 42 | 62 | 77 |
|---|---|---|---|---|---|---|

Verkettung

Linear

Quadratisch

Doppelt

```
A)
o(A)
th downto 2 do
ractMax()
```

Max()) = n log n

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | | | | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | | | | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | | | | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | | | | |