

## 2. Sorting

### Das Sortierproblem

Gegeben: Folge von Objekten

Gesucht: Sortierung gemäß bestimmten Schlüsselwertes

### Schlüsselproblem

Schlüssel müssen nicht eindeutig, aber sortierbar sein

Im Folgenden Annahme, dass es totale Ordnung  $\leq$  auf der Menge  $M$  aller Schlüsselwerte gibt

Betrachtung von Schlüsselwerten ohne Satellitendaten, meist Zahlen

Satellitendaten sind "unnötig", nur Schlüsselwert ist relevant

### Insertion Sort

```
insertionSort(A)
  FOR i=1 TO A.length-1 DO
    // insert A[i] in pre-sorted sequence A[0..i-1]
    key=A[i];
    j=i-1; // search for insertion point backwards
    WHILE j>=0 AND A[j]>key DO
      A[j+1]=A[j]; // move elements to right
      j=j-1;
    A[j+1]=key;
```

- $A$  ist ein Array/Liste/...
- $A[0..i-1]$  ist immer bereits sortiert
- Wert an der Stelle  $A[i]$  wird dann im sortierten Bereich an der richtigen Stelle eingefügt, dabei wird alles immer verschoben

### Laufzeitanalysen: O-Notation

Wie viele Schritte macht ein Algorithmus in Abhängigkeit von der Eingabekomplexität?

- Man nimmt meist Worst-Case für alle Eingaben gleicher Komplexität
- (Worst-Case-)Laufzeit:  $T(n) = \max\{\text{Anzahl Schritte}\}$  für eine Aufgabe
- Komplexität wird meist von einem Faktor dominiert, wie z.B. der Anzahl zu sortierender Zahlen  $n$

### Laufzeitanalyse für einen Algorithmus

- Nehme ein festes  $n$ , z.B. Anzahl zu sortierender Elemente
- Wie oft wird jede Zeile maximal ausgeführt (in Abhängigkeit von  $n$ )?
- Jeder Zeile  $i$  wird Aufwand  $c_i$  zugeordnet, wird dann mit Anzahl der Ausführungen multipliziert
- Elementare Operationen (Zuweisung, Vergleich,...) haben konstanten Aufwand 1
- $T(n)$  ist dann sehr komplex, siehe Insertion Sort-Beispiel

## Asymptotische Vereinfachung

- 1. Vereinfachung: Man nimmt nur dominanten Term  $D(n)$  von  $T(n)$
- 2. Vereinfachung: Nur abhängigen  $A(n)$  Term betrachten, Vorfaktoren entfernen
  - Konstante Vorfaktoren sind von Berechnungsmodell, Leistung abhängig

## $\Theta$ -Notation/Landau-Symbole

Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}_{>0}$  Funktionen,  $\mathbb{N}$  ist die Eingabekomplexität,  $\mathbb{R}_{>0}$  die Laufzeit.

$$\Theta(g) := \{f : \exists c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Schreibweise:  $f \in \Theta(g), f = \Theta(g)$

- $g(n)$  ist eine asymptotisch scharfe Schranke von  $f(n)$
- $\Theta$ -Notation beschränkt eine Funktion asymptotisch von oben und unten
- Beispiel: **Insertion Sort**:  $T(n) \in \Theta(n^2)$  für  $c_1 = \frac{3}{2}, c_2 = 7, n_0 = 2$

## $O$ -Notation

$g$  ist obere Schranke von  $f$

$$O(g) := \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq c g(n)\}$$

Sprechweise:  $f$  wächst höchstens so schnell wie  $g$

Schreibweise:  $f = O(g), f \in O(g)$

$\Theta(g(n)) \subseteq O(g(n)) \rightsquigarrow f(n) \in \Theta(g) \Rightarrow f(n) \in O(g)$

## Rechenregeln

- Konstanten:  $f(n) = a, a \in \mathbb{R}_{>0} \Rightarrow f(n) \in O(1)$
- Skalarmultiplikation:  $f \in O(g), a \in \mathbb{R}_{>0} \Rightarrow a \cdot f \in O(g)$
- Addition:  $f_1 \in O(g_1), f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max\{g_1, g_2\})$ , max ist punktweise
- Multiplikation:  $f_1 \in O(g_1), f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2)$

## $\Omega$ -Notation

$g$  ist untere Schranke von  $f$

$$\Omega(g) := \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, 0 \leq c g(n) \leq f(n)\}$$

Sprechweise:  $f$  wächst mindestens so schnell wie  $g$

Schreibweise:  $f = \Omega(g)$ ,  $f \in \Omega(g)$

$\Theta(g(n)) \subseteq \Omega(g(n)) \rightsquigarrow f(n) \in \Theta(g) \Rightarrow f(n) \in \Omega(g)$

## Zusammenhang $O$ , $\Omega$ , $\Theta$

$f(n) \in \Theta(g(n))$  gdw.  $f(n) \in O(g(n))$  und  $f(n) \in \Omega(g(n))$

## Anwendung $O$ -Notation

$f = O(g)$  ist üblich,  $f \in O(g)$  ist wahre Bedeutung und besser, da  $O(g)$  Menge ist  
 $O(n^4) = O(n^5)$  gilt, nicht jedoch  $O(n^5) = O(n^4)$ !

## Ungleichungen

- $\leq$  nur mit  $O$  verwenden
- $\geq$  nur mit  $\Omega$  verwenden

## Insertion Sort Beispiel

Algorithmus macht maximal  $T(n)$  viele Schritte,  $T(n) \in \Theta(n^2)$

$\rightsquigarrow$  Laufzeit  $\leq T(n) \in O(n^2)$

Für "gute" Eingaben (bereits vorsortiert) macht Algorithmus  $\Theta(n)$  viele Schritte

Es wird aber mit Worst-Case gearbeitet, Insertion Sort hat quadratische Laufzeit

## Komplexitätsklassen

Klasse	Bezeichnung	Beispiel
$\Theta(1)$	Konstant	Einzeloperation
$\Theta(\log n)$	Logarithmisch	Binäre Suche
$\Theta(n)$	Linear	Sequentielle Suche
$\Theta(n \log n)$	Quasilinear	Sortieren eines Arrays
$\Theta(n^2)$	Quadratisch	Matrixaddition
$\Theta(n^3)$	Kubisch	Matrixmultiplikation
$\Theta(n^k)$	Polynomiell	
$\Theta(2^n)$	Exponentiell	Travelling-Salesman
$\Theta(n!)$	Faktoriell	Permutationen

## $o$ -Notation, $\omega$ -Notation

Gelten für alle Konstanten, nicht nur eine

$$o(g) := \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

$$2n \in o(n^2), 2n^2 \notin o(n^2)$$

$$\omega(g) := \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

$$\frac{n^2}{2} \in \omega(n), \frac{n^2}{2} \notin \omega(n^2)$$

## Bubble Sort

```
bubbleSort(A)
  FOR i=A.length-1 DOWNTO 0 DO
    FOR j=0 TO i-1 DO
      IF A[j]>A[j+1] THEN SWAP(A[j],A[j+1]);
      // SWAP: temp=A[j+1]; A[j+1]=A[j]; A[j]=temp;
```

- Quadratische Laufzeit
- Große Werte "steigen nach oben" und sammeln sich am Ende
- $A[i..A.length - 1]$  ist nach jedem Durchlauf der äußeren Schleife korrekt

## Merge Sort

### Idee: Divide & Conquer (& Combine)

Teile Liste in Hälften, sortiere (rekursiv) Hälften, sortiere wieder zusammen  
(Teil-)Sortierung erfolgt im Array selbst, Teillisten werden genutzt

Siehe auch [7. Advanced Designs](#)

### Algorithmus

```
mergeSort(A,l,r) // initial call: l=0,r=A.length-1
  IF l<r THEN // more than one element
    m=floor((l+r)/2); // m (rounded down) middle
    mergeSort(A,l,m); // sort left part
    mergeSort(A,m+1,r); // sort right part
    merge(A,l,m,r); // merge into one

merge(A,l,m,r) // requires l<=m<=r
  //array B with r-l+1 elements as temporary storage
  pl=l; pr=m+1; // position left, right
  FOR i=0 TO r-l DO // merge all elements
    IF pr>r OR (pl<=m AND A[pl]<=A[pr]) THEN
      B[i]=A[pl];
      pl=pl+1;
    ELSE //next element at pr
      B[i]=A[pr];
      pr=pr+1;
  FOR i=0 TO r-l DO A[i+l]=B[i]; //copy back to A
```

- Es wird zwischen Position  $l$  und  $r$  sortiert
- $m$  ist der letzte Index des linken Teils
- Es wird aufgeteilt, bis die Teillisten Länge 1 haben
- Dann werden sie zusammengefügt und dabei sortiert
  - *merge* nimmt immer das kleinste Element aus den beiden Listen und fügt es der Ergebnisliste in  $B$  hinzu
- Laufzeit  $\Theta(n \cdot \log n)$
- $T(n) \geq \Omega(n \cdot \log n)$

## Laufzeitanalyse: Rekursionsgleichungen

### Rekursion manuell iterieren

Beispiel Merge Sort,  $T(n)$  ist max. Anzahl an Schritten für Arrays der Größe  $n$ :

$$T(n) \leq 2T\left(\frac{n}{2}\right) + c + dn \leq \dots \leq 2^{\log_2 n} \cdot c + \log_2 n \cdot cn \in O(n \log n)$$

### Allgemeiner Ansatz: Mastermethode

Allgemeine Form der Rekursionsgleichung:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), T(n) \in \Theta(1)$$

mit  $a \geq 1, b > 1, f(n)$  asymptotisch positive Funktion.

### Interpretation

- Problem wird in  $a$  Teilprobleme der Größe  $\frac{n}{b}$  aufgeteilt
- Lösen jeder der  $a$  Teilprobleme benötigt Zeit  $T\left(\frac{n}{b}\right)$
- $f(n)$  umfasst Kosten für Aufteilen und Zusammenfügen

### Mastertheorem

Seien  $a \geq 1, b > 1$  konstant,  $f(n)$  eine positive Funktion und  $T(n)$  über den nicht-negativen ganzen Zahlen durch folgende Rekursiongleichung definiert:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), T(1) \in \Theta(1)$$

$\frac{n}{b}$  wird hierbei entweder auf- oder abgerundet.

Dann besitzt  $T(n)$  die folgenden asymptotischen Schranken:

1. Gilt  $f(n) \in O(n^{\log_b(a)-\epsilon})$  für ein  $\epsilon > 0$ , dann gilt  $T(n) \in \Theta(n^{\log_b a})$
2. Gilt  $f(n) \in \Theta(n^{\log_b a})$ , dann gilt  $T(n) \in \Theta(n^{\log_b a} \cdot \log_2 n)$
3. Gilt  $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$  für ein  $\epsilon > 0$  und  $af\left(\frac{n}{b}\right) \leq cf(n)$  für ein  $c < 1$  und hinreichend große  $n$ , dann ist  $T(n) \in \Theta(f(n))$

### Interpretation

Entscheidend ist das Verhältnis von  $f(n)$  zu  $n^{\log_b a}$ :

1. Wenn  $f(n)$  polynomiell kleiner als  $n^{\log_b a}$ , dann gilt  $T(n) \in \Theta(n^{\log_b a})$
  2. Wenn  $f(n), n^{\log_b a}$  gleiche Größenordnung, dann  $T(n) \in \Theta(n^{\log_b a} \cdot \log n)$
  3. Wenn  $f(n)$  polynomiell größer als  $n^{\log_b a}$  und  $af\left(\frac{n}{b}\right) \leq cf(n)$ , dann  $T(n) \in \Theta(f(n))$
- Regularität Fall 3:  $af\left(\frac{n}{b}\right) \leq cf(n), c < 1$
  - $f(n)$  dominiert asymptotisch den Ausdruck
  - Fall 3 bedeutet, dass die Wurzel den größten Arbeitsaufwand verrichtet, diese wird hiermit sichergestellt
- Wenn das Mastertheorem nicht anwendbar ist, ist die Baumstruktur zu analysieren

## Quicksort

### Idee

- Divide & Conquer
- Mehr Arbeit in Aufteilen, Zusammenfügen kostenlos
- Wählt 1. Element als Pivot-Element
- Dann Partitionieren der Elemente, sodass  $\leq$  Pivot links,  $\geq$  Pivot rechts
- Rekursiv fortsetzen

### Algorithmus

```
quicksort(A,l,r) // initial call: l=0,r=A.length-1
    IF l<r THEN //more than one element
        p=partition(A,l,r); // p partition index
        quicksort(A,l,p); // sort left part
        quicksort(A,p+1,r); // sort right part

partition(A,l,r) //requires l<r, returns int in l..r-1
    pivot=A[l];
    pl=l-1; pr=r+1; //move from left resp. right
    WHILE pl<pr DO
        REPEAT pl=pl+1 UNTIL A[pl]>=pivot; //move left up
        REPEAT pr=pr-1 UNTIL A[pr]<=pivot; //move right down
        IF pl<pr THEN Swap(A[pl],A[pr]);
        p=pr; //store current value
    return p // A[l..p] left, A[p+1..r] right
```

- Wenn das übergebene Array nur 1 Element hat, wird nichts getan
- Partition:
  - `pl`, `pr` werden vergrößert/verkleinert, bis das Element an der Position nicht passt (größer/kleiner als `pivot`)
  - falls dann `pl` noch kleiner als `pr` ist, sind die beiden Elemente zu vertauschen

- Wiederholung
- Am Ende, nachdem alles vertauscht wurde, wird der letzte Wert von `pr` zurückgegeben, dies ist dann der letzte Index des linken Teilarrays
- Wenn alle Teilarrays Größe 1 haben, ist man fertig

## Laufzeit

### Worst-Case

- Immer nur Arrays der Größe 1 abgespalten
- $\Theta(n^2)$

### Best-Case

- Aufteilung in gleich große Arrays
- $\Theta(n \log n)$

### Average Case

- Nehme erwartete Anzahl von Schritten über eine Verteilung der Komplexität  $n$
- $T(n) = E_{D(n)}[t]$ ,  $t$  ist Anzahl der Schritte für  $x$
- $O(n \log n)$

## Randomisierte Variante

```
partition(A,l,r) //requires l<r, returns int in l..r-1
    j=RANDOM(l,r); Swap(A[l],A[j]); //j uniform in [l..r]
    pivot=A[l];
    ...
```

- Wähle zufälliges Element, vertausche es dann mit 1. Element, sonst alles gleich

### Erwartete Laufzeit (Average-Case)

- Zufällige Wahl des Pivot-Elementes teilt Array im Durchschnitt mittig, unabhängig davon, wie Array aussieht
- Worst-Case:  $T(n) = \max\{\text{\#steps for } x\}$
- Erwartete Laufzeit:  $T(n) = \max\{E_A[\text{\#steps for } x]\}$ 
  - zufällige Wahl des Algorithmus  $A$  für schlechteste Eingabe, Komplexität  $n$
  - $O(n \log n)$

## Vergleich

### Insertion Sort

- $\Theta(n^2)$
- Einfach
- Für kleine  $n \leq 50$  beste Wahl

## Merge Sort

- Beste asymptotische Laufzeit  $\Theta(n \log n)$

## Quicksort

- Worst-Case  $\Theta(n^2)$ , randomisiert erwartet  $\Theta(n \log n)$
- Praxis: Schneller als [Merge Sort](#), da weniger Kopieroperationen
- Implementierungen nutzen [Insertion Sort](#) für kleine  $n$

## Untere Schranke für vergleichsbasiertes Sortieren

- nur deterministische Algorithmen, gilt im Schnitt auch für Randomisierte
- Jeder (korrekte) vergleichsbasierte Sortieralgorithmus muss mindestens  $\Omega(n \log n)$  viele Vergleiche machen.

## Radix-Sort

### Ansatz

- Schlüssel sind  $d$ -stellige Werte in  $D$ -närem Zahlensystem
- "Buckets" erlauben Einfügen, Entnehmen in eingefügter Reihenfolge
  - konstanter Zeitaufwand
  - Umsetzung durch [Queues](#)

## Algorithmus

```
radixSort(A) // keys: d digits in range [0,D-1]
// B[0][],..., B[D-1][] buckets (init: B[k].size=0)
  FOR i=0 TO d-1 DO //0 least, d-1 most sign. digit
    FOR j=0 TO n-1 DO putBucket(A,B,i,j);
    a=0;
    FOR k=0 TO D-1 DO //rewrite to array
      FOR b=0 TO B[k].size-1 DO
        A[a]=B[k][b]; //read out bucket in order
        a=a+1;
      B[k].size=0; //clear bucket again

  return A

putBucket(A,B,i,j) // call-by-reference
```



```
z=A[j].digit[i]; // i-th digit of A[j]
b=B[z].size; // next free spot
B[z][b]=A[j];
B[z].size=B[z].size+1;
```

- $i$ -te Iteration ( $i \in [0..d-1]$ ):
  1. Sortiere Zahlen anhand  $i$ . Ziffer in entsprechenden Bucket
  2. Gehe aufsteigend durch Buckets und führe in nächster Stelle im Array ein
- Mit höchstwertiger Ziffer beginnen funktioniert nicht

## Laufzeit

$O(d \cdot (n + D))$

$D$  oft als konstant angesehen  $\rightsquigarrow O(dn)$

Linear, wenn  $d$  auch als konstant angesehen

Eindeutige Schlüssel für  $n$  Elemente benötigen  $d = \Theta(\log_D n)$  Ziffern  $\rightsquigarrow O(n \log n)$

#TODO Fix style

#TODO Eventuell Schranke weiter ausarbeiten