Algo Zusammenfassung

- Stirling Formel → Approximation der Fakultät

Multiplikation ganzer Zahlen

Karatsuba und Ofman

- Normaler Ansatz: n^2 Multiplikationen

$$z2 = 10a + b \; und \; z2 \; 10c + d$$

$$(10a + b) * (10c + d) = 100ac + 10ac + 10bd + bd + 10(a - b) * (d - c)$$

- Nur drei Multiplikationen → x10 ist billig
- Mehr als zweistellige Zahl → Berechnung ist rekursiv (oder induktiv)

$$M(2^k) = \begin{cases} 1 & \text{falls } k = 0 \text{ ist} \\ 3 \cdot M(2^{k-1}) & \text{falls } k > 0 \text{ ist.} \end{cases}$$

- Um Rekusionsgleichung auszulösen → Teleskopieren
  - Einsetzen, bis Vermutung für explizite Form
- Karatsuba → n^1.58

Induktionsbeweis:

1. Induktionsanfang: Zeige, dass die Aussage für n = 1 gilt
2. Induktionshypothese: Wir nehmen an, die Aussage sei gültig für ein allgemeines n Element N.
3. Inkuktionsschritt: Zeige, dass aus der Gültigkeit der Aussage für n (Induktionshypothese) die Gültigkeit der Aussage für n + 1 folgt.

Star finden

- Alle kennen Star, Star kennt niemand
  - Keinen Star
  - Einen Star
  - Nur einen Star

Naiv:

- Jeden jeden fragen (n * (n – 1) alle möglichen Fragen

Algorithmus 2a:

- Rekursiv finden:
  - Eine Person herausschicken, Star suchen, Person hereinschicken, checken ob diese Person Star ist

F(n) = 2(n – 1) + F(n – 1) = 2(n – 1) + 2(n – 2) + … + 2 = n(n – 1) → keine Verbesserung

Algorithmus 2b (Verbesserung)

- $$F(n) = \begin{cases} 2 & \text{für } n = 2 \\ 1 + F(n-1) + 2 & \text{für } n > 2. \end{cases} \quad (3)$$

  Wie zuvor teleskopieren wir und erhalten

  $$F(n) = 3 + F(n-1) = 3 + 3 + F(n-2) = \cdots = 3(n-2) + 2 = 3n - 4, \quad (4)$$

Kostenmodell

- Rechenoperationen: + - * / zwei natürlicher Zahlen
- Vergleichsoperationen < > = zwei nat. Zahlen
- Zuweisungen <- A, Variable x links, Zahlenwert Ausdruck A rechts

O-Notation, O-Kalkül, Bachmann-Landau-Notation

- G1 Element O(f) und g2 Element O(f) ist auch g1 + g2 Element O(f)

$$\mathcal{O}(f) := \left\{ g : \mathbb{N} \to \mathbb{R}^+ \;\middle|\; \begin{array}{l} \text{es gibt } c > 0, n_0 \in \mathbb{N} \text{ so, dass } g(n) \leq cf(n) \\ \text{für alle } n \geq n_0 \end{array} \right\}. \quad (5)$$

Anschaulich bedeutet dies, dass f (bis auf einen konstanten Faktor) asymptotisch gesehen eine obere Schranke für g bildet (siehe Abbildung 1.6). Diese *$\mathcal{O}$-Notation* (auch *$\mathcal{O}$-Kalkül* oder *Bachmann-Landau-Notation* genannt) erlaubt uns nun, Ausdrücke zu vereinfachen und nach oben abzuschätzen. So ist z.B. $3n - 4 \in \mathcal{O}(n)$. Dies kann man sehen, indem man etwa $c = 3$ und $n_0 = 1$ wählt. Weitere Beispiele sind

- $2n \in \mathcal{O}(n^2)$ (mit $c = 2$ und $n_0 = 1$),
- $n^2 + 100n \in \mathcal{O}(n^2)$ (mit $c = 2$ und $n_0 = 100$),
- $n + \sqrt{n} \in \mathcal{O}(n)$ (mit $c = 2$ und $n_0 = 1$).

$$\Omega(f) := \left\{ g : \mathbb{N} \to \mathbb{R}^+ \;\middle|\; \begin{array}{l} \text{es gibt } c > 0, n_0 \in \mathbb{N} \text{ so, dass } g(n) \geq cf(n) \\ \text{für alle } n \geq n_0 \end{array} \right\}.$$

$$\Theta(f) := \mathcal{O}(f) \cap \Omega(f). \quad (7)$$

Im die Konzepte zu vertiefen, betrachten wir nun einige weitere Beispiele. B

- $n \in \mathcal{O}(n^2)$: Diese Aussage ist korrekt, aber ungenau. Tatsächlich gelten ja auch $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$: Auch diese Aussage ist korrekt, man würde aber üblicherweise $\mathcal{O}(n^2)$ statt $\mathcal{O}(2n^2)$ schreiben, da Konstanten in der $\mathcal{O}$-Notation weggelassen werden können.
- $2n^2 \in \mathcal{O}(n)$: Diese Aussage ist falsch, da $\frac{2n^2}{n} = 2n \xrightarrow[n\to\infty]{} \infty$ und daher durch keine *Konstante* nach oben beschränkt werden kann.

Theorem 1.1. *Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ zwei Funktionen. Dann gilt:*

1) Ist $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, dann ist $f \in \mathcal{O}(g)$, und $\mathcal{O}(f) \subsetneq \mathcal{O}(g)$.

2) Ist $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0$ (wobei C konstant ist), dann ist $f \in \Theta(g)$.

3) Ist $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, dann ist $g \in \mathcal{O}(f)$, und $\mathcal{O}(g) \subsetneq \mathcal{O}(f)$.

Komplexität, Kosten, Laufzeit

Summenformeln

- Durch Induktion beweisen
- Geometrische Formel

Kombinatorik

n! abschätzen

$$2^n \leq n! \leq n^n \, \text{für } n \geq 2$$

$$ln(ab) = ln(a) + ln(b)$$

$$\ln(n!) = n\ln(n) - n + \mathcal{O}(\ln n) \quad (17)$$

$$\Rightarrow n! = e^{\mathcal{O}(\ln n)} \cdot \left(\frac{n}{e}\right)^n. \quad (18)$$

Man beachte, dass $e^{\mathcal{O}(\ln n)} \neq \mathcal{O}(n)$ ist, denn $e^{\mathcal{O}(\ln n)}$ enthält zum Beispiel die Funktion $e^{2\ln n} = (e^{\ln n})^2 = n^2$, die nicht in $\mathcal{O}(n)$ liegt. Eine genauere Abschätzung liefert die *Stirling-Formel*

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \quad (19)$$

- Menge allelr Teilmengen von M, P(M) = {A| A C_ M} heisst Potenzmenge von M
- Hat |M| n Elemente, dann hat die Potenzmenge von M Kardinalität $|P(M)| = S$^2.

- $\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k}$ für $1 \leq k \leq n$. Dies kann ebenfalls durch Termumformung der Definition bewiesen werden:

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-1-k)!}$$
$$= \frac{(n-1)!}{(k-1)!(n-k-1)!} \cdot \left(\frac{1}{n-k} + \frac{1}{k}\right)$$
$$= \frac{(n-1)!}{(k-1)!(n-k-1)!} \cdot \left(\frac{n}{k(n-k)}\right) = \binom{n}{k}.$$

- $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$ für alle $x, y \in \mathbb{R}$. Dies kann durch vollständige Induktion über n bewiesen werden (Übung).

Rekurrenz

$$S(0) = C(0), S(n) = a \cdot S(n-1) + C(n)\, für\, n \geq 1$$

Maximum Subarray Sum

Algorithmus 1 Naiv

- Alle möglichen Intervalle ausprobieren

NAIVER ALGORITHMUS

MSS-NAIV$(a_1, \ldots, a_n)$

1 maxS $\leftarrow 0$
2 Für $i \leftarrow 1, \ldots, n$ (alle Anfänge)
3 Für $j \leftarrow i, \ldots, n$ (alle Enden)
4 $S \leftarrow \sum_{k=i}^{j} a_k$ (berechne Summe)
5 Merke maximales S

- Theta n^3

Algorithmus 2 (Voreberechnung Präfixsummen

- Doppeltes Berechnen der Teilsummen
- Berechne Summe von Position 1 bis und mit Position i

- Summe von ak genügt es S i-1 von Sj zu subtrahieren

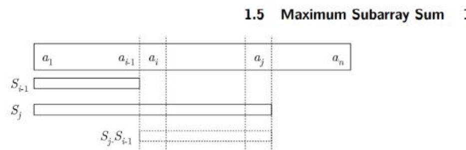


Abb. 1.10 Zur Berechnung von $\sum_{k=i}^{j} a_k$ genügt es, S_{i-1} von S_j zu subtrahieren.

- Daher kann Summe ak von O(1) Zeit berechnet werden

Einleitung

```
MSS-Präfixsummen(a_1, ..., a_n)
1 S_0 ← 0
2 Für i ← 1, ..., n
3     S_i ← S_{i-1} + a_i
4 maxS ← 0
5 Für i ← 1, ..., n
6     Für j ← i, ..., n
7         S ← S_j − S_{i-1}
8         Merke maximales S
```

- Theta(n^2)

Algorithmus 3 (Divide and Conquer)

- Halbiere Array, drei Fälle
  - Lösung liegt vollständig links
  - Lösung liegt vollständig rechts
  - Lösung läuft über Mitte
- Um Wert bester Lösung zu ermitteln, die über Mitte läuft, reicht es also die grösste Suffixsumme in der ersten und die grösste in der zwetien Hälfte zu berechnen, und Werte addieren

```
MSS-Divide-and-Conquer(a_1, ..., a_n)
1 Wenn n = 1 ist, dann gib max{a_1, 0} zurück.
2 Wenn n > 1 ist:
3     Teile die Eingabe in A_1 = ⟨a_1, ..., a_{n/2}⟩ und A_2 = ⟨a_{n/2+1}, ..., a_n⟩ auf.
4     Berechne rekursiv den Wert W_1 einer besten Lösung für das Array A_1.
5     Berechne rekursiv den Wert W_2 einer besten Lösung für das Array A_2.
6     Berechne grösste Suffixsumme S in A_1.
7     Berechne grösste Präfixsumme P in A_2.
8     Setze W_3 ← S + P.
9     Gib max{W_1, W_2, W_3} zurück.
```

- Theta(nlogn)

Algorithmus 4 (Induktiv von links nach rechts)

Induktiver Algorithmus

```
MSS-Induktiv(a_1, ..., a_n)
1 randmax ← 0
2 maxS ← 0
3 Für i ← 1, ..., n:
4     randmax ← randmax + a_i
5     Wenn randmax > maxS:
6         maxS ← randmax
7     Wenn randmax < 0:
8         randmax ← 0
9 Gib maxS zurück.
```

- DP Artig
- Theta(n)
- Geht nicht besser als Theta(n) → Algorithmus muss alle Elemente ansehen

Sortieren

Binary Search

1. B = A[m] Schlüssel an Pos. M gefunden, return m
2. B < A[m], rekursive Suche links (Pos 1,…,m-1)
3. B > A[m], recursive Suche rechts (Pos m+1,…, n)

(Array muss sortiert sein!)

Interpolationssuche

- Ahnung, wo Schlüssel sein könnte → Verringerung / Vergrösserung des ½ Faktors

Gut: O(loglogn) schlecht Omega(n)

$$\rho = \frac{b - A[\text{left}]}{A[\text{right}] - A[\text{left}]} \in [0, 1],$$

Exponentielle Suche

```
EXPONENTIAL-SEARCH(A = (A[1], ..., A[n]), b)

1 r ← 1                                          ▷ Initiale rechte Grenze
2 while r ≤ n and b > A[r]                       ▷ Finde rechte Grenze
3     r ← 2 · r                                  ▷ Verdopple rechte Grenze
4 return BINARY-SEARCH((A_1, ..., A[min(r, n)]), b) ▷ Weiter mit binärer Suche
```

Untere Schranke

- Binary Search geht nicht besser als O(logn)
  - Suche nach Element b ist erfolgreich → b kann an n Positionen stehen
  - Für n mögliche Ergebnisse der Suche, Entscheidungsbaum muss min. einen Knoten enthalten (Knotenzahl min. n)
  - Vergleiche im schlechtesten Fall: Höhe Baum (Weg von Wurzel zu Blatt)
  $$2^0 + 2^1 + \cdots + 2^{h-1} = 2^h - 1 < 2^h$$
  $$n \leq Anzahl\ Knoten\ im\ Entscheidungsbaum < 2^h -> h > log_2(n)$$
  Omega(logn)

Suchen in unsortierten Arrays

Immer O(n)

- Beispiel mit Gruppen, siehe Skript S. 33.

Elementare Sortierverfahren

Bubblesort

- n-1 Vergleiche
- if A[i] > A[i+1], swap keys at pos i and i+1
- n-1 Wiederholungen sollten reichen
-

```
BUBBLESORT(A = (A[1], ..., A[n]))

1 for j ← 1 to n − 1 do
2     for i ← 1 to n − 1 do
3         if A[i] > A[i + 1] then Vertausche A[i] und A[i + 1].
```

-
- Theta(n^2)
- In place sort

j=1, i=1	3	7	5	1	4
i=2	3	5	7	1	4
i=3	3	5	1	7	4
i=4	3	5	1	4	7
j=2, i=1	3	5	1	4	7
i=2	3	1	5	4	7
i=3	3	1	4	5	7
j=3, i=1	1	3	4	5	7

Selectionsort

- Suche j + 1 grössten Schlüssel (bis j ist sortiert)

i=0	3	7	5	1	4
i=1	3	4	5	1	7
i=2	3	4	1	5	7
i=3	3	1	4	5	7
i=4	1	3	4	5	7

SelectionSort(A)

```
1 for k ← n downto 2 do                         ▷ k Schlüssel sind noch nicht sortiert
2     Berechne den Index i des maximalen Schlüssels in A[1],...,A[k].
3     Vertausche A[i] und A[k].                 ▷ A[k] enthält (n − k + 1)-grössten Schl.
```

- Theta(n^2)
- Schlüsselvertauschungen n-1

Insertionsort

- Sortiert, aber nicht an korrekter Stelle
- Binary Search nach A[k+1] auf A[1]…A[k]
- Beim Einsetzen müssen Werte zwischengepseichert werden und verschoben werden

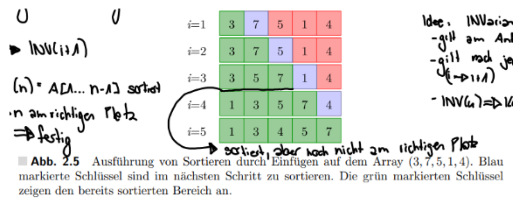


Abb. 2.5 Ausführung von Sortieren durch Einfügen auf dem Array (3, 7, 5, 1, 4). Blau markierte Schlüssel sind im nächsten Schritt zu sortieren. Die grün markierten Schlüssel zeigen den bereits sortierten Bereich an.

InsertionSort($A = (A[1], \ldots, A[n])$)

```
1 for k ← 1 to n − 1 do                          ▷ k Schlüssel sind bereits sortiert
2     Benutze binäre Suche auf A[1],...,A[k], um die Position i
      zu finden, an die A[k + 1] eingefügt werden muss.
3     b ← A[k + 1]                               ▷ Speichere A[k + 1] in b zwischen
4     for j ← k downto i do                      ▷ Verschiebe A[i],...,A[k]
5         A[j + 1] ← A[j]                        ▷ auf A[i + 1],...,A[k + 1]
6     A[i] ← b                                   ▷ Speichere A[k + 1] an A[i]
```

- O(nlogn) Vergleiche
- Schlechtester Fall Theta(n^2)

Heapsort

- Maximum ist zuoberst
- Alle unteren stellen einen erneuten Heaptree dar
- Heap-Eigenschaft: für Knoten mit Schlüssel x, Schlüssel der entsprechenden Nachfolger höchstens so gross wie x selbst

2.4 Heapsort 35

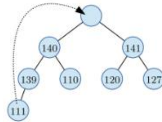


Abb. 2.7 Extraktion des Maximums (Schlüssel 193) aus dem in Abbildung 2.6 dargestellten Heap. Im ersten Schritt wird der in $A[8]$ gespeicherte Schlüssel nach $A[1]$ kopiert.

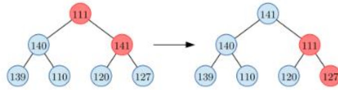


Abb. 2.8 Wiederherstellung der Heap-Eigenschaft für die in Abbildung 2.7 dargestellte Struktur. Zunächst werden die Schlüssel 111 und 141 vertauscht, danach 111 und 127.

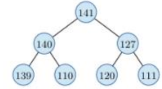


Abb. 2.9 Der rekonstruierte Heap $A = (141, 140, 127, 139, 110, 120, 111)$.

WIEDERHERSTELLUNG DES HEAPS

```
RESTORE-HEAP-CONDITION(A, i, m)

1 while 2·i ≤ m do                        ▷ A[i] hat linken Nachfolger
2     j ← 2·i                              ▷ A[j] ist linker Nachfolger
3     if j+1 ≤ m then                      ▷ A[i] hat rechten Nachfolger
4         if A[j] < A[j+1] then j ← j+1    ▷ A[j] ist grösserer Nachfolger
5     if A[i] ≥ A[j] then STOP             ▷ Heap-Bedingung erfüllt
6     Vertausche A[i] und A[j]             ▷ Reparatur
7     i ← j                                ▷ Weiter mit Nachfolger
```

- O(logn) → Restore Heap Condition
- O(nlogn) unsortiertes Array zu Max-Heap

Mergesort

- Array wird halbiert
- Halbierte werden wieder halbiert
- Wenn fertig werden sie verschmolzen und das gesamte ist sortiert

Verschmelzen

- Zwei Zeigern von vorn nach hinten durchlaufen (F = F1 + F2)
- Bis F1 oder F2 vollständig durchlaufen ist → dann das andere einfach angehängt

```
MERGE(A, left, middle, right)

1 B ← new Array[right-left+1]              ▷ Hilfsarray zum Verschmelzen
2 i ← left; j ← middle+1; k ← 1            ▷ Zeiger zum Durchlaufen
3 ▷ Wiederhole, solange beide Teilfolgen noch nicht erschöpft sind
  while (i ≤ middle) and (j ≤ right) do
4     if A[i] ≤ A[j] then B[k] ← A[i]; i ← i+1
5     else B[k] ← A[j]; j ← j+1
6     k ← k+1
7 ▷ Falls die erste Teilfolge noch nicht erschöpft ist, hänge sie hinten an
  while i ≤ middle do B[k] ← A[i]; i ← i+1; k ← k+1
8 ▷ Falls die zweite Teilfolge noch nicht erschöpft ist, hänge sie hinten an
  while j ≤ right do B[k] ← A[j]; j ← j+1; k ← k+1
9 ▷ Kopiere Inhalte von B nach A zurück
  for k ← left to right do A[k] ← B[k − left + 1]
```

Beweis für Sortierung Skript S.42

REKURSIVER MERGESORT

```
MERGESORT(A, left, right)

1 if left < right then
2     middle ← ⌊(left + right)/2⌋          ▷ Mittlere Position
3     MERGESORT(A, left, middle)            ▷ Sortiere vordere Hälfte von A
4     MERGESORT(A, middle + 1, right)       ▷ Sortiere hintere Hälfte von A
5     MERGE(A, left, middle, right)         ▷ Verschmilz Teilfolgen
```

Theta(nlogn) viele Schlüsselvergleiche- bewegungen

```
NaturalMergesort(A)

 1 repeat
 2    right ← 0                          ▷ Elemente bis A[right] sind bearbeitet
 3    while right < n do                 ▷ Finde und verschmilz die nächsten Runs
 4       left ← right+1                  ▷ Linker Rand des ersten Runs
 5       middle ← left                   ▷ A[left], . . . , A[middle] ist bereits sortiert
 6       while (middle < n) and A[middle + 1] ≥ A[middle] do
 7          middle ← middle+1            ▷ Ersten Run um ein Element vergrössern
 8       if middle < n then              ▷ Es gibt einen zweiten Run
 9          right ← middle+1             ▷ Rechter Rand des zweiten Runs
10          while (right < n) and A[right + 1] ≥ A[right] do
11             right ← right+1           ▷ Zweiten Run um ein Element vergrössern
12          Merge(A, left, middle, right)
13       else right ← n                  ▷ Es gibt keinen zweiten Run
14 until left = 1
```

```
StraightMergesort(A)

 1 length ← 1                            ▷ Länge bereits sortierter Teilfolge
 2 while length < n do                   ▷ Verschmilz Folgen d. Länge leng
 3    right ← 0                          ▷ A[1], . . . , A[right] ist abgearbeite
 4    while right+length < n do          ▷ Es gibt noch mind. zwei Teilfolge
 5       left ← right+1                  ▷ Linker Rand der ersten Teilfolge
 6       middle ← left+length−1          ▷ Rechter Rand der ersten Teilfol
 7       right ← min(middle+length, n)   ▷ Rechter Rand der zweite Teilfol
 8       Merge(A, left, middle, right)   ▷ Verschmilz beide Teilfolgen
 9    length ← length·2                  ▷ Verdoppelte Länge der Teilfolge
```

Quicksort

5	9	2	1	17	11	8
5	1	2	9	17	11	8
5	1	2	8	17	11	9

- Wählt irgendein Schlüssel p → Pivot
  - Prüfen ob Schlüssel grösser oder kleiner als p sind
  - Ordnet sie entsprechend um
- Durchlaufe Array von links, bis Schlüssel A[i] gefunden, welcher grösser als p ist
- Durchlaufe Array von rechts, bis wir einen Schlüssel A[j] finden, welcher kleiner als p ist
- Ist i < j, dann sind sowohl A[i] als auch A [j] in den falschen Seiten
- Schluss: A[i] wird mit p vertauscht

```
Aufteilungs-   Partition(A, l, r)
schritt
                1 i = l
                2 j = r − 1
                3 p = A[r]
                4 repeat
                5    while i < r and A[i] < p do i = i + 1
                6    while j > l and A[j] > p do j = j − 1
                7    if i < j then Vertausche A[i] und A[j]
                8 until i ≥ j
                9 Tausche A[i] und A[r]
               10 return i
```

- Pivotelement dient als Stopper

2	7	1	5	9	8	11	17	24

- Schlechtester Fall: Theta(n^2)
- Rekursionstiefe von n
- Schlechtesten Fall Omega(n) Speicher
  - Führen Rekursion auf kürzerem Teil durch
  - Rekursionstiefe O(logn)

Untere Schranke vergleichsbasierten Sortierverfahren

- Baum benötigt n! viele Knoten
- Binär Baum mit n! Blättern, mindestens Höhe log(n!)
- Was in Omega(n log n) liegt

DP

- Fibonacci → sehr teuer. Kann besser sein: mit Memoization

```
Fibonacci-Memoization(n)

1 if n-te Fibonacci-Zahl bereits berechnet then
2    f ← memo[n]                          ▷ Benutze gespeicherten Wert
3 else                                    ▷ Berechne n-te Fibonacci-Zahl
4    if n ≤ 2 then f ← 1
5    else f ← Fibonacci-Memoization(n − 1)+
              Fibonacci-Memoization(n − 2)
6    memo[n] ← f                          ▷ Speichere berechneten Wert
7 return f
```

DP Programmierung: Vorgehen

- DP-Tabelle
- Optimale Lösungen der entsprechenden Teilprobleme
- Informationen über Lösung selbst könnten dort gespeichert werden
- Anfang: alle Einträge füllen, die Elementar- bz. Randfällen entsprechen
- Zugreifen auf vorherige Einträge

1. Definition der DP-Tabelle: Welche Dimensionen hat die Tabelle? Was ist die Bedeutung jedes Eintrags?
   a. Tabelle T Grösse 1 x n, wobei der i-te Eintrag die i-te berechnete Fibonacci Zahl enthält
2. Berechnung eines Eintrags: Wie berechnet sich ein Eintrag aus den Werten von anderen Einträgen? Welche Einträge hängen nicht von anderen Einträgen ab?
   a. Für i = 1 und i = 2 setzen wir T[i] <- 1. Für allgemeines i >= 3 setzen wir T[i] <- T[i-1]+T[i-2]
3. Berechnungsreihenfolge: In welcher Reihenfolge kann man die Einträge berechnen, so dass die jeweils benötigten anderen Einträge bereits vorher berechnet wurden
   a. Die Einträge T[i] werden mit aufsteigendem i berechnet. Wir berechnen also zuerst T[1] dann T[2] dann T[3] usw.
4. Auslesen der Lösung: Wie lässt sich die Lösung am Endke aus der Tabelle auslesen?
   a. Die zu berechnenden n-te Fibonacci-Zahl ist am Ende im Eintrag T[n] gespeichert

Pseudeopolynomiell: Laufzeit ist polynomiell, wenn Wert aller Eingaben vorkommenden Zahlen polynomiell in der Eingabelänge beschränkt ist

Datenstrukturen

Stack

Push(x, S):	legt Objekt x auf Stapel S	
Pop(S) :	Entfernt oberstes Objekt von S, returned es	→ S keine Objekte, return null
Top(S):	Return oberstes Objekt von S, entferne nicht.	→ S keine Objekte, return null
IsEmpty(S):	True, wenn Stack S leer ist (keine Objekte) false, wenn Objekte	
EmptyStack:	Liefert leeren Stackl	

Queue

Dequeue(Q):	entfernt hinterstes Objekt und returned es	→ Q keine Objekte, return null
Front(Q):	Return letztes Objekt queue, entferne nicht	→ Q keine Objekte, return null
IsEmpty(Q):	True wenn Q leer ist, false, wenn Objekte	
EmptyQueue:	liefert leere Queue	
Enqueue(x, Q):	Objekt x zu hinterst Q	

PriorityQueue

- Enqueue(x, Q) wird mit insert(x, p, Q) ersetzt → p ist priority
- Dequeue(Q) wird mit extract-Max(Q) ersetzt → return Objekt mit höchster Priorität

Multistack

- Multipop(k, S):entfernt k zuletzt hinzugefügten Objekte aus S und liefert sie /nach Zeitpunkt der Hinzufügung absteigend sortiert) zurück. Enthält S weniger als k Elemente, werden entsprechend weniger Elemente entfernt und zurückgeliefert. Insbesondere wird null zurückgeliefert, wenn S vor Aufruf der Operation keine Elemente enthielt
- Laufzeit O(k) anstatt O(1)

Natürliche Suchbäume

Binärer Baum:

- Blatt → Baum ist leer
- Besteht aus innerer Knoten v mit zwei Bäumen Tl(v) und Tr(v) als linken, rechten Nachfolger.
- Ein Knoten ohne Vorgänger → Wurzel (Root)
- In Knoten v gespeichert:
  - Schlüssel v (Key)
  - Zeiger v.left auf linken Nachfolgerknoten
  - Zeiger v.right auf rechten Nachfolgerknoten
- Linker Teilbaum: Schlüssel sind kleiner
- Rechter Teilbaum: Schlüssel sind grösser

```
SEARCH(k)

1 v ← ROOT
2 while v ist kein Blatt do
3     if k = v.KEY then return true            ▷ Element gefunden
4     else if k < v.KEY then v ← v.LEFT        ▷ Suche links weiter
6     else v ← v.RIGHT                         ▷ Suche rechts weiter
7 return false                                 ▷ k nicht gefunden
```

- Laufzeig O(h) (Höhe des Baums)

Einfügen

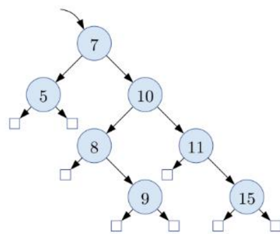
- Suche nach k
- Wenn k gefunden → nicht nochmals eingefügt
- Ansonsten inneren Knoten mit Schlüssel k und zwei Blättern ersetzen
- O(h)

Entfernen

1. Beide Nachfolger von v sind Blätter:
   a. Knoten v direkt gelöscht werden. Nachfolger von u wird durch Blatt ersetzt
2. Genau ein Nachfolger von v ist Blatt
   a. W innerer Knoten → Nachfolger von v. Entsprechende Nachfolger von u wird durch w ersetztzt und v gelöscht
3. Kein Nachfolger von v ist Blatt
   a. Symmetrischer Nachfolger von v. v wird durch w ersetzt und dann gelöscht
   b. Einmal rechts, immer links oder einmal links, immer rechts → tauschen
   c. Beim Löschen von w nur einer der beiden erstgenannten Fällen tritt auf

Durchlaufordnungen für Bäume

- Hauptreiehnfolge: Rekursiv Tl(v) verarbeitet dann Tr(v) 7, 5, 10, 8, 9, 11, 15
- Nebenreihenfolge: Rückwärts Rekursion 5, 9, 8, 15, 11, 10, 7
- Sym. Reihenfolge: Symmetrische Reihenfolge
- 5, 7, 8, 9, 10, 11, 15



Operationen

- Min(T) return minimum Schlüssel. O(h)
- Extract-min(T) return and remove key mit minimalem Wert O(h)
- List(T) Sortierte List emit T gepseicherten Schlüssel (sym. Folge) (On)
- Join(T1, T2) Extract-Min(T2) erhaltne Schlüssel k und resultierenden Baum → neu

AVL-Bäume

$$bal(v). = h\big(T_r(v)\big) - h\big(T_l(v)\big)$$

AVL-Bedingung besagt, alle Knoten v des Baums $bal(v) \in \{-1,0,1\}$

Einfügen

- Zunächst wie natürlichen Suchbaum
- Testen ob AVL-Bedingung gilt
- Pro Knoten Höhe des repräsentierten Teilbaums speichern
  - Oder aktuelle Balance speichern (-1, linkerTeilbaum höher, 0, beide Teilbäume gleich hoch, +1, rechter Teilbaum höher)

Upin Methode, aufgerufen wenn:

1. Neu eingefügte Schlüssel befindet sich im Teilbaum mit Wurzel u und durch Einfügung ist Höhe des Teilbaums mit Wurzel u um 1 gewachsen
2. U hat eine von 0 verschiedene Balance
3. U hat Vorgänger w

94 Datenstrukturen für Wörterbücher

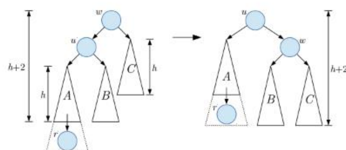


Abb. 4.21 Wurde der neue Schlüssel in den linken Teilbaum von u eingefügt, dann genügt eine einfache Rechtsrotation um den Knoten w zur Wiederherstellung der AVL-Bedingung bei w. Wie zuvor sind die Blätter von r nicht eingezeichnet.

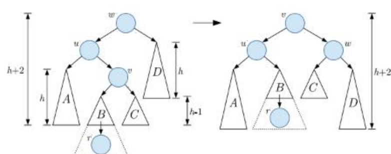


Abb. 4.22 Wurde der neue Schlüssel in den rechten Teilbaum von u eingefügt, dann wird eine Doppelrotation zur Rebalancierung benutzt. Dazu wird zunächst eine einfache Linksrotation um u und direkt danach eine einfache Rechtsrotation um w durchgeführt, sodass sich der rechts dargestellte Baum ergibt. Der Fall, in dem der neue Schlüssel in C (statt in B) eingefügt wurde, wird identisch gehandhabt.

Zeit: O(log n)

Graphenalgorithmen

- Graphen, welcher jede Kante genau einmal benutzt wird → Eulerscher Zyklus
- Gerade Anzahl Kanten (jeder Knoten hat geraden Grad)

Definition Gerichteter Graph

- Kanten gehen in bestimmte Richtung
- U nach v
- Kante ist ein Paar von Knoten
- Schleifen, wenn v, v

Definition Ungerichteter Graph

- Kanten haben keine Richtung, man kann in beide gehen
- Schleife, wenn v

Allgemein

- Graph G = (V, E) vollständig: E jede Kante zwischen zwei verschiedenen Knoten v und w enthält
- Graph G = (V, E) bipartit: Graph aufteilen, sodass gleich viele Knoten in U und in W sind
- Graph G = (V, E) gewichtet: Kantengewicht
- Adjazent: Wenn Knoten v Kante E enthält
- Schleife werden im gerichteten Graphen nur einmal gezählt
- Pfad: Weg, der keinen Knoten mehrfach benutzt
- Zusammenhängend: Ungerichteter Graph in dem zwischen jedem Paar zweier Knoten v und w ein Weg existiert
- Zyklus: Start- Endknoten überein
- Kreis: Zyklus, welcher Knoten nicht mehr als einmal benutzt (ausser Start/Endknoten)=
- Kreis Länge > 2
- Baum: ungerichteter Graph, zusammenhängend, kreisfrei
  - Hat n Knoten, n-1 Kanten

Darstellungsformen

- Adjazenzmatrix
  - Grösse nxn
  - Einträge {0, 1}
  - 1 wenn E Kante (vi, vj) enthält
  - Platz Theta(|V|^2)
- Adjazenzlistendarstellung
  - Besteht aus Array (A[1],…, A[n])
  - Eintrag A[i] einfach verkettete Liste aller Knoten
  - Platz Theta(|V|+|E|)
- Theorem:
  - G Graph und t Element Natürliche Zahlen
  - Element an Position (i, j= in Matrix A^t_G gibt Anzahl der Wege der Länge t von vi nach vj an)

Dreiecke überprüfen

- Diagonale muss addiert / 6 = 1 sein → dann Dreieck → Matrix vorher dreimal potenzieren

Beziehung zu Relationen

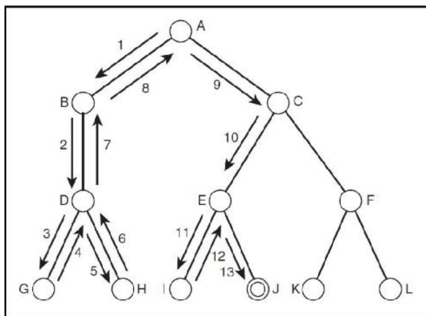
- Reflexiv, wenn E jede Kante mit v Element V enthält, G also Schleife um jeden Knoten hat.
- Symmetrisch, wenn ungerichtet
- Transitiv, wenn jedes Paar zweier Kanten u, v und v, w auch u, w ist

- Äquivalenzrelation: wenn alle erfüllt
  - Kollektion vollständiger, ungerichteter Graphen, jeder Knote eine Schleife hat
- Reflexive, transitive Hülle beschreibt Erreichbarkeitsrelation

```
ReflexiveTransitiveHull(A)
1 for k ← 1,...,n do
2     a_kk ← 1                                    ▷ Reflexive Hülle
3     for i ← 1,...,n do
4         for j ← 1,...,n do
5             if a_ik = 1 and a_kj = 1 then a_ij ← 1   ▷ Berechne Weg über v_k
```

Theta(n^3)

DFS (Tiefensuche)

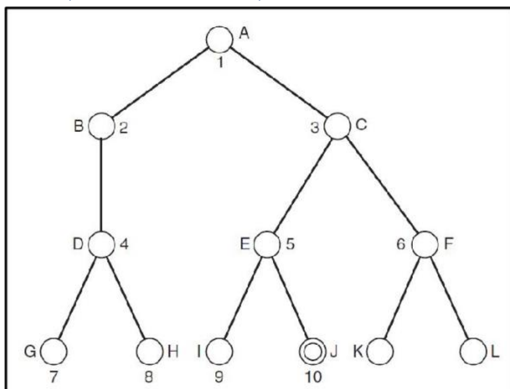


Nachfolger eines Knotens genau dann in alphabetischer Reihenfolge abgearbeitet werden, wenn sie in der Adjazenzliste in alphabetischer Reihenfolge vorkommen → nicht unbedingt der Fall

Laufzeit: Theta(|V| + |E|)

```
DFS-Visit-Iterative(G, v)
1 S ← ∅                                  ▷ Initialisiere leeren Stapel S
2 Push(v, S)                             ▷ Leg v oben auf S
3 while S ≠ ∅ do
4     w ← Pop(S)                         ▷ Aktueller Knoten
5     if w noch nicht besucht then
6         Markiere w als besucht
7         for each (w, x) ∈ E in reverse order do
8             if x noch nicht besucht then
9                 Push(x, S)             ▷ Leg x oben auf S
```

BFS (Breitensuche)



- Alle Nachfolger eines Knotens abarbeiten, dann alle Nachfolger dieser Nachfolger

BFS-Visit-Iterative(G, v)

1	$Q \leftarrow \emptyset$	▷ *Initialisiere leere Schlange Q*
2	Markiere v als aktiv	
3	Enqueue(v, Q)	▷ *Füge v zur Schlange Q hinzu*
4	**while** $Q \neq \emptyset$ **do**	
5	$\quad w \leftarrow$ Dequeue(Q)	▷ *Aktueller Knoten*
6	$\quad$ Markiere w als besucht	
7	$\quad$ **for each** $(w, x) \in E$ **do**	
8	$\quad\quad$ **if** x nicht aktiv und x noch nicht besucht **then**	
9	$\quad\quad\quad$ Markiere x als aktiv	
10	$\quad\quad\quad$ Enqueue(x, Q)	▷ *Füge x zur Schlange Q hinzu*

Laufzeit Theta (|V| + |E|)

Zusammenhangskomponenten

- Ungerichteter Graph
- Berechnung Zusammenhangskomponenten
  - Äquivalenzklassen reflexiven und transitiven Hülle von G
- Anzahl Äquivalenzklassen exakt Anzahl der Neustarts im DFS- BFS-Rahmenprogramm entspricht
- Alle Knoten, welche vom Startknoten aus erreichbar sind → Zusammenhangskomponente¨

Topologische Sortierung

- Gerichteter Graph
- Topologische Sortierung, wenn kreisfrei
- Suche Knoten v0 mit Eingangsgrad 0 und gib v0 aus
  - Dann entferne v0 und v0 ausgehenden Kanten aus G
  - Verfahren rekursiv auf verbleibenden Graphen
  - Bis keine Knoten mit Eingangsgrad 0 mehr besitzt / keine Knoten
- Keine Knoten: topologische Sortierung
- Keine Knoten mit Eingangsgrad 0: Kreis
- Speichern Knoten mit Eingangsgrad 0 in Array, auf Stapel tun

Algorithmus Topological-Sort(V, E)

1	$S \leftarrow$ EmptyStack	▷ *Stapel S initialisieren*
2	**for each** $v \in V$ **do** $A[v] \leftarrow 0$	
3	**for each** $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$	▷ *Berechne Eingangsgrade*
4	**for each** $v \in V$ **do**	▷ *Lege Knoten mit Eingangs-*
5	$\quad$ **if** $A[v] = 0$ **then** Push(v, S)	▷ *grad 0 auf den Stapel S*
6	$i \leftarrow 1$	
7	**while** S not empty **do**	
8	$\quad v \leftarrow$ Pop(S)	▷ *Knoten mit Eingangsgrad 0*
9	$\quad$ ord$[v] \leftarrow i$; $i \leftarrow i + 1$	▷ *Weise korrekte Position zu*
10	$\quad$ **for each** $(v, w) \in E$ **do**	▷ *Verringere Eingangsgrad*
11	$\quad\quad A[w] \leftarrow A[w] - 1$	▷ *der Nachfolger*
12	$\quad\quad$ **if** $A[w] = 0$ **then** Push(w, S)	
13	**if** $i = \lvert V \rvert + 1$ **then return** ord **else** "Graph enthält einen Kreis"	

Laufzeit Theta(|V| + |E|)

Kürzeste Wege

- Gerichteter, gewichteter Graph
- Kürzester Weg, Bezeichnung auf Kantengewichte und nicht auf Anzahl der Kanten

Uniformen Kantengewicht

- Modifizierte BFS, wenn alle Gewichte gleich sind

Nicht negativen Kantengewicht

- Dijkstra
  - Obere Schranke merken

- Laufzeit O(|V|^2)
- Verbesserung mit Prioritäts Schlange
- Priorität aktuellen Wert d[v] nehmen

Extract-Min(Q): liefert irgendeinen Knoten v zurück, dessen Prioriät unter allen in Q verwalteten Knoten minimal ist, entferne ihn. Operation nur, wenn Q nicht leer

Insert(v, Pi, Q): Fügt Knoten v mit Priorität Pi ein. Operation nur aufgerufen, wenn Q den Knoten v noch nicht enthält

Decrease-Key(v, Pi, Q): setzt Priorität des Knotens v auf Pi. Nur aufgerufen, wenn Q den Knoten v enthält und die Priorität von v vor Aufruf der Operation grösser, gleich Pi ist.

```
DIJKSTRA(G = (V, E), s)
 1 for each v ∈ V\{s} do                    ▷ Initialisiere für alle Knoten die
 2     d[v] ← ∞; p[v] ← null                ▷ Distanz zu s sowie Vorgänger
 3 d[s] ← 0; p[s] ← null                    ▷ Initialisierung des Startknotens
 4 Q ← ∅                                    ▷ Leere Prioritätswarteschlange Q
 5 INSERT(s, 0, Q)                          ▷ Füge s zu Q hinzu
 6 while Q ≠ ∅ do
 7     u ← EXTRACT-MIN(Q)                   ▷ Aktueller Knoten
 8     for each (u, v) ∈ E do               ▷ Inspiziere Nachfolger
 9       if p[v] = null then                ▷ v wurde noch nicht entdeckt
10         d[v] ← d[u] + w((u, v))          ▷ Berechne obere Schranke
11         p[v] ← u                         ▷ Speichere u als Vorgänger von v
12         ENQUEUE(v, d[v], Q)              ▷ Füge v zu Q hinzu
13       else if d[u] + w((u, v)) < d[v] then ▷ Kürzerer Weg zu v entdeckt
14         d[v] ← d[u] + w((u, v))          ▷ Aktualisiere obere Schranke
15         p[v] ← u                         ▷ Speichere u als Vorgänger von v
16         DECREASE-KEY(v, d[v], Q)         ▷ Setze Priorität von v herab
```

Falls mit Heaps implementiert dann O((|V| + |E|)log|V|) → noch besser mit Fibonacci-Heap O((|V| log|V|+ |E|)

Allgemeine Kantengewichte

$$T[v, i] \leftarrow \min\Big(T[v][i-1], \min_{(u,v)\in E}\big(T[u][i-1] + w((u,v))\big)\Big),$$

- Laufzeit O(|V|^3)

```
BELLMAN-FORD(G = (V, E), s)
 1 for each v ∈ V\{s} do                  ▷ Initialisiere für alle Knoten die
 2     d[v] ← ∞; p[v] ← null              ▷ Distanz zu s sowie Vorgänger
 3 d[s] ← 0; p[s] ← null                  ▷ Initialisierung des Startknotens
 4 for i ← 1, 2, ..., |V| − 1 do          ▷ Wiederhole |V| − 1 Mal
 5     for each (u, v) ∈ E do             ▷ Iteriere über alle Kanten (u, v)
 6       if d[v] > d[u] + w((u, v)) then  ▷ Relaxiere Kante (u, v)
 7         d[v] ← d[u] + w((u, v))        ▷ Berechne obere Schranke
 8         p[v] ← u                       ▷ Speichere u als Vorgänger von v
 9 for each (u, v) ∈ E do                 ▷ Prüfe, ob eine weitere Kante
10     if d[u] + w((u, v)) < d[v] then    ▷ relaxiert werden kann
11       Melde Kreis mit negativem Gewicht
```

Bellman und Ford: O(|V||E|)

Alle Paaren von Knoten Floyd-Warhsall

Graph: (Knoten 1, 2, 3, 4)

Init: $d^0_{uu} = 0$
$d^0_{uv} = 1$ (u,v) Kante
$d^0_{uv} = \infty$ sonst

d^i_{uv} = kürzester Weg $u \rightsquigarrow v$ mit Zwischenknoten $\leq i$

$d^i_{uv} = \min(d^{i-1}_{uv}, d^{i-1}_{ui} + d^{i-1}_{iv})$

i=0: (Zeilen u, Spalten v)

	1	2	3	4
1	0	-	1	1
2	-	0	-	1
3	1	1	0	-
4	-	-	1	0

i=1: ($\min(d_{12}, d_{11} + d_{12})$; $d_{31} + d_{14}$)

	1	2	3	4
1	0	-	1	1
2	-	0	-	1
3	1	1	0	2
4	-	-	1	0

i=2:

	1	2	3	4
1	0	-	1	1
2	-	0	-	1
3	1	1	0	2
4	-	-	1	0

i=3: ($d_{13} + d_{32}$; $d_{43} + d_{32}$)

	1	2	3	4
1	0	2	1	1
2	-	0	-	1
3	1	1	0	2
4	2	2	1	0

i=4: ($d_{24} + d_{41}$)

	1	2	3	4
1	0	2	1	1
2	3	0	2	1
3	1	1	0	2
4	2	2	1	0

Beachte: Hier hatten wir noch keine Kantengewichte

```
FW(G)   // G = (V, E, c)
  // Initialisiere Tableau
  for u ∈ V: d0_uu = 0
  for (u,v) ∈ E: d0_uv = c(u,v); for (u,v) ∉ E: d0_uv = ∞
  // DP
  for i = 1..n
    for u = 1..n
      for v = 1..n
        d^i_uv = min(d^{i-1}_uv, d^{i-1}_ui + d^{i-1}_iv)
  return d
```

wenn inplace lässt man den Index weg

Funktioniert wenn keine negativen Zyklen
Test: v ist in negativem Zyklus $\Leftrightarrow d^n_{vv} < 0$

Beispiel: wenn $i_1 < i_2 < i_3$ hat man $d^{i_3}(v,v) < 0$

(v →(-1) i_1 →(2) i_2 →(-3) i_3 →(-7) v)

Kürzeste Wege merken: $O(n^2)$ Extraplatz (ähnlich Ford's Algorithmus)

Laufzeit: $O(n^3)$

Variante: Transitive Closure für Relationen

$G = (V, E)$ beschreibt Relation ρ auf V: $u \rho v \Leftrightarrow (u,v) \in E$

hier: d^i_{uv} = "v ist erreichbar von u mit Knoten $\leq i$"

$d^i_{uv} = d^{i-1}_{uv} \vee (d^{i-1}_{ui} \wedge d^{i-1}_{iv})$ Übung!

Iteriere, schau ob mit Zwischenstopp mehr, kürzerer Weg entsteht

Alle Paaren von Knoten Johnson

1. Füge Graphen neue Knoten N E U sowie Kanten (N E U; v) zu allen Knoten v mit Gewicht 0 ein
2. Benutze Algo von Bellman Ford ausgehend von N E U zur Berechnung aller Höhen h(v). (Wenn negativer Kreis, abbrechen)
3. Berechne neue Kantengewichte $w'(u,v) = w(u,v) + h(u) - h(v)$
4. Für jeden Knoten u starte Dijkstras Algorithmus von u Element V\{N E U} aus, um Wege zu allen Knoten v Element V des Graphen zu berechnen. Gefundene Distanzen um h(u) – h(v) reduziert

Laufzeit O(|V|^2 log|V| + |V||E|)

MST

- Zusammenhängender Teilgraph mit minimalem Gewicht

Boruvka

- Iteriere über jeden Knoten und wähle immer leichtestes Gewicht. Wenn noch nicht MST aber über alle Knoten iteriert, wähle leichtestes Gewicht zwischen den Zusammenhangskomponenten
- O(mlogn)

Prim

- Wähle immer leichtestes Gewicht welches an Graphen anschliesst
  - Falls Kreis bildet, nimm Kante nicht
- O((V + E)logn)

Kruskal

- Sortiere Kanten nach Gewicht

- Wähle immer leichtestes Gewicht (unabhängig ob angeschlossen oder nicht)
- (E + V log V