

ADT Wörterbuch

Gewünschte Operationen:

- 1) `search( $x, W$ )`: Entscheide, ob der Schlüssel x in W vorkommt, und gib in diesem Fall die Position im Speicher zurück.
- 2) `insert( $x, W$ )`: Füge den Schlüssel x in W ein, falls er noch nicht darin vorkommt.
 $\Rightarrow$ wir müssen immer zuerst suchen, ob x schon enthalten ist
- 3) `delete( $x, W$ )`: Finde und lösche den Schlüssel x aus W , falls er darin vorkommt.

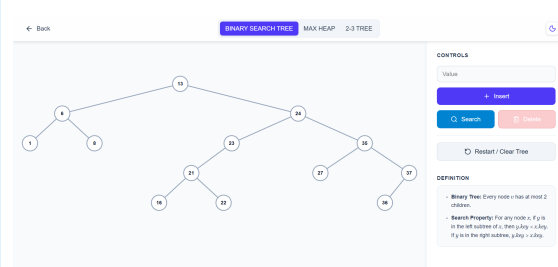
Mögliche Implementierungen.

	search	insert	delete	
unsortiertes Array	$O(n)$	$O(n)$	$O(n)$	mit BinarySearch
sortiertes Array	$O(\log n)^*$	$O(n)^*$	$O(n)$	beim Einfügen am Anfang müssen $O(n)$ Elemente nach hinten gerückt werden
Doppelt verk. Liste	$O(n)$	$O(n)$	$O(n)$	
Binärer Suchbaum	$O(h)$	$O(h)$	$O(h)$	} heute
2-3-Baum	$O(\log n)$	$O(\log n)$	$O(\log n)$	

Binärer Suchbaum:

- Jeder Knoten hat maximal 2 Kinder
- Suchbaum-Bedingung:

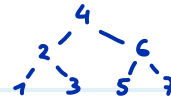
alle Schlüssel im linken Teilbaum < Schlüssel des Knotens < alle Schlüssel im rechten Teilbaum



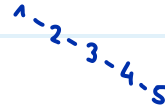
- Alle Operationen in $O(h)$, wobei h die Höhe des Baumes ist

(die Operation Löschen ist nicht prüfungsrelevant)

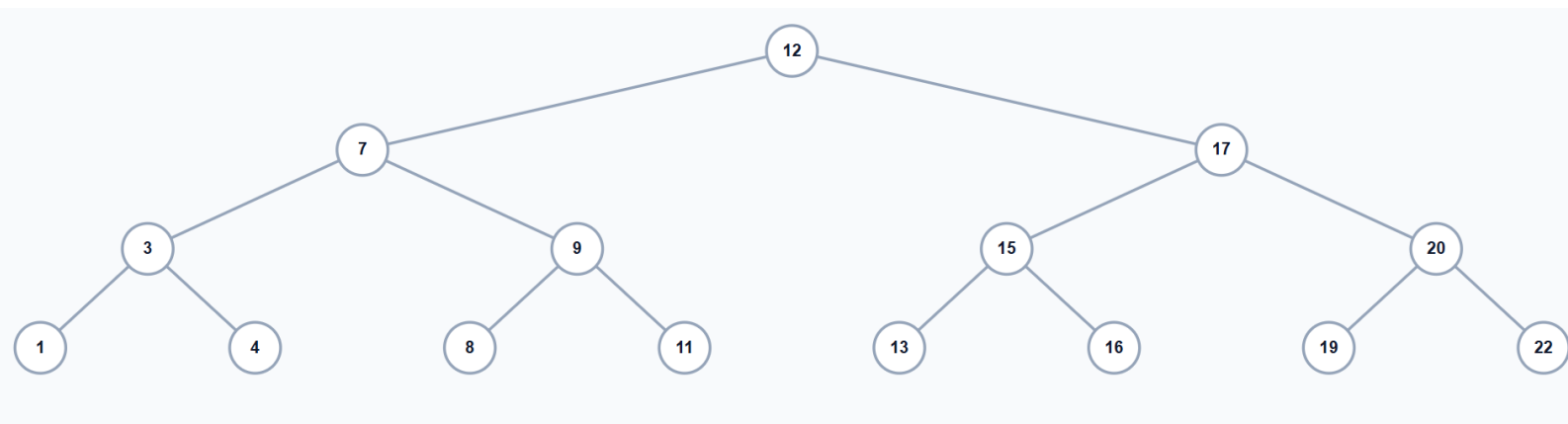
- Gut falls $h = \Theta(\log n)$ ist (balanciert)



- Schlecht falls $h = \Theta(n)$ ist (unbalanciert)



Verbindung zwischen binary search und binary search tree:



als Liste: $[1, 3, 4, 7, 8, 9, 11, 12, 13, 15, 16, 17, 19, 20, 22]$

search(8):

Verbesserung: 2-3-Bäume

(Achtung: unterschiedliche Namen in anderen Quellen)

Ein 2-3-Baum für $n \geq 2$ Schlüssel erfüllt die folgenden Bedingungen:

- Jeder innere Knoten hat entweder 2 oder 3 Kinder. ("2-3-Bedingung")
Knoten die keine Blätter sind
- Alle Blätter befinden sich auf derselben Ebene (Tiefe). $\Rightarrow$ perfekt balanciert, $h = \Theta(\log n)$
- Die Schlüssel sind in den Blättern gespeichert. Jedes Blatt enthält genau einen Schlüssel.
- Ein Knoten mit 2 Kindern enthält einen Separator s_1 . Alle Schlüssel im linken Teilbaum sind $\leq s_1$, und alle im rechten sind $> s_1$ ("Separatorbedingung bei zwei Kindern").
- Ein Knoten mit 3 Kindern enthält zwei Separatoren s_1 und s_2 (mit $s_1 < s_2$). Für die Schlüssel in den drei Teilbäumen gilt ("Separatorbedingung bei drei Kindern"):
 - linker Teilbaum: alle Schlüssel $\leq s_1$,
 - mittlerer Teilbaum: alle Schlüssel $> s_1$ und $\leq s_2$,
 - rechter Teilbaum: alle Schlüssel $> s_2$.

Merkhilfe "Bei Gleichheit nach links"

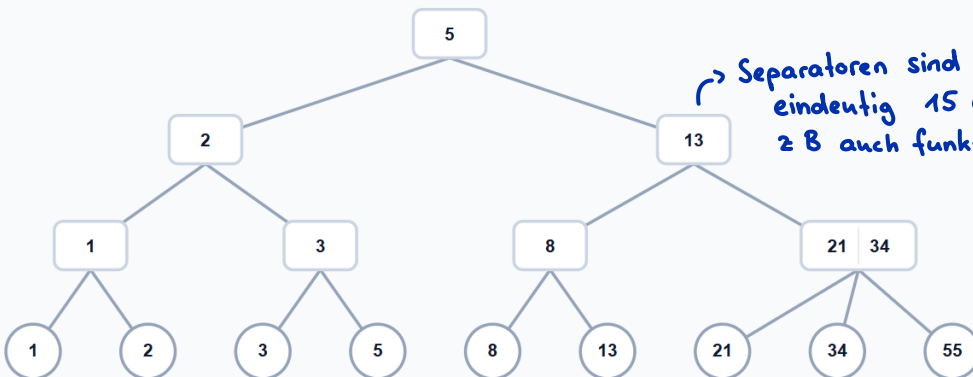
← Back

BINARY SEARCH TREE

MAX HEAP

2-3 TREE

↺



CONTROLS

Value

+ Insert

Q Search

🗑 Delete

🔄 Restart / Clear Tree

DEFINITION

- **2-3 Condition:** Every internal node has either 2 or 3 children.
- **Balance:** All leaves are at the same depth.
- **Leaves:** Keys are stored in leaves. Each leaf contains exactly 1 key.
- **2-Node (1 Separator s_1):**
 - Left subtree keys $\leq s_1$
 - Right subtree keys $> s_1$
- **3-Node (2 Separators $s_1 < s_2$):**
 - Left subtree keys $\leq s_1$
 - Middle subtree keys $> s_1$ and $\leq s_2$
 - Right subtree keys $> s_2$

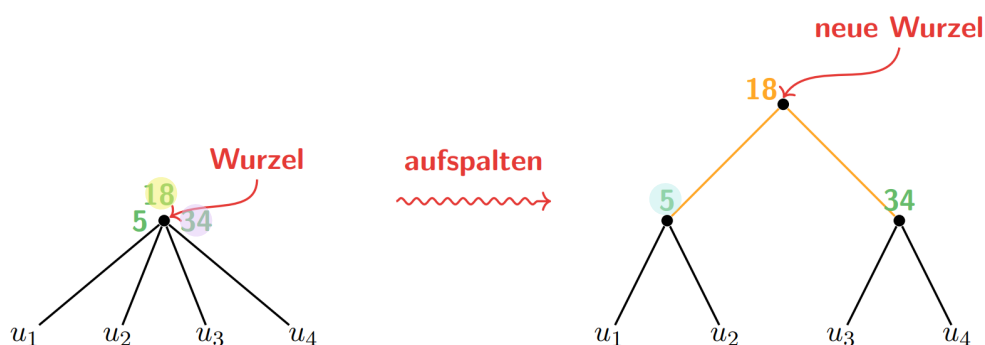
search(x): Bei der Wurzel beginnen und den Separatoren entlang folgen

insert(x): 1) Suche richtigen Platz für x

2) Erstelle ein neues Blatt mit Schlüssel x, und füge beim Elternknoten einen passenden Separator hinzu.

3) Falls der Elternknoten nun 4 Kinder hat, teile ihn in zwei Knoten auf die den **kleinsten** und **grössten** Separator übernehmen, und schiebe den **mittleren** Separator eine Ebene nach oben

Falls die 2-3-Bedingung dadurch ein Level weiter oben verletzt wurde, repariere rekursiv.



delete(x): 1) Suche x

2) Lösche x und den dazugehörenden Separator

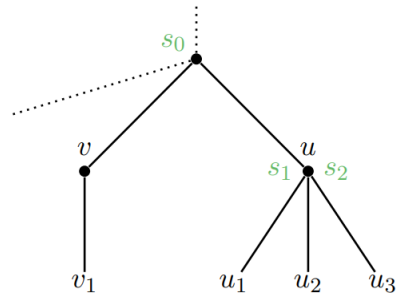
3.) Falls 2-3-Bedingung verletzt: (= nur noch 1 Kind)

i) Falls ein Geschwisterknoten 3 Kinder hat, dann adoptieren wir eines und aktualisieren die dazugehörenden Separatoren

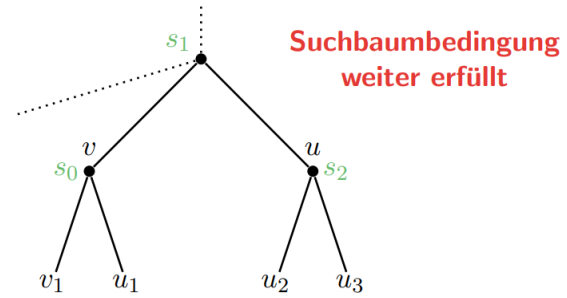
ii) Sonst verschmelze den Knoten mit einem Geschwisterknoten (der 2 Kinder hat)

Falls die 2-3-Bedingung dadurch ein Level weiter oben verletzt wurde, repariere rekursiv.

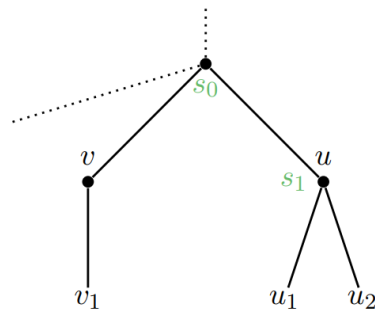
Bsp. Fall (i)



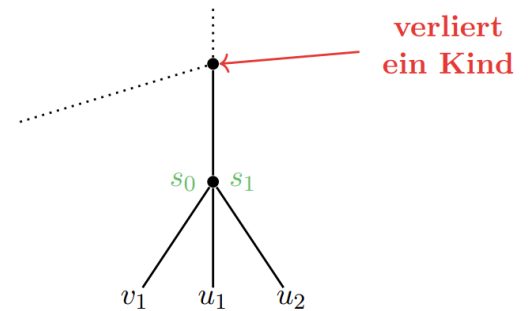
Adoption
~~~~~→



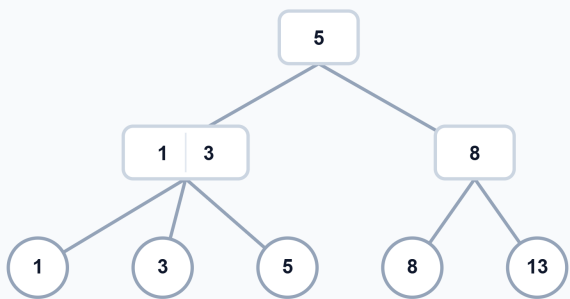
## Bsp. Fall (ii)



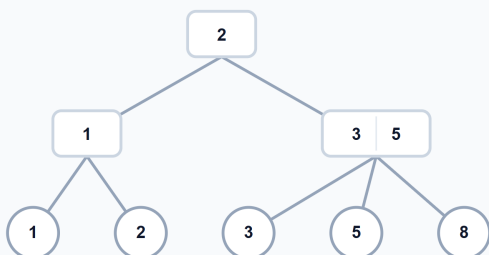
Verschmelzen  
~~~~~→



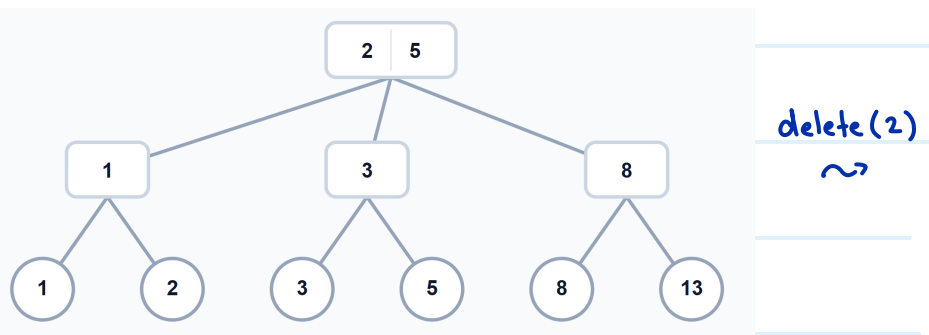
Aufgaben:



insert(21)
~~~~~



insert(13)  
~~~~~



Füge die Schlüssel 3, 1, 2, 5, 8, 21, 13, 34 in dieser Reihenfolge in einen leeren binären Suchbaum:

Challenge: Füge die Schlüssel 3, 1, 2, 5, 8, 21, 13, 34 in dieser Reihenfolge in einen leeren 2-3-Baum:

Dynamische Programmierung

Herangehensweise

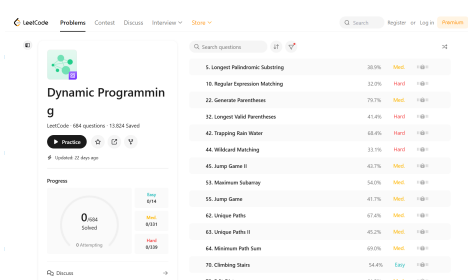
- 1.) Dimension der DP-Tabelle definieren
- 2.) Teilproblem definieren, d.h. die Bedeutung eines DP-Entries festlegen
- 3.) Base Cases und Rekursion bestimmen
- 4.) Berechnungsreihenfolge
- 5.) Lösung aus der DP-Tabelle auslesen
- 6.) Laufzeit analysieren

Zwei Varianten zum Programmieren:

- Bottom-up (= iterativ) ist am Anfang einfacher
- Top-down (= rekursiv = mit memoization) besonders geeignet wenn man sich nicht um die Berechnungsreihenfolge kümmern möchte

Wie lernen?

- möglichst viele Aufgaben lösen
- mit einfachen Aufgaben beginnen und sie Bottom-up und Top-down programmieren
- Zusatzaufgaben auf leetcode.com sind sehr hilfreich



Fibonacci

Dimension: $F[1 \dots n]$

alternative: $F[0 \dots n]$

Teilproblem: $F[i] := i\text{-te Fibonacci Zahl}$

Base Case: $F[1] = 1$ und $F[2] = 1$

alternative: $F[0] = 0$ und $F[1] = 1$

2 Base Cases weil sich die Rekursion auf 2 vorherige Entries bezieht

Rekursion: $F[i] = F[i-1] + F[i-2]$ für $3 \leq i \leq n$

Berechnungsreihenfolge: for $i = 3 \dots n$: berechne $F[i]$

Lösung: $F[n]$

Laufzeit: $O(n)$

← Back

FIBONACCI

MAXIMUM SUBARRAY SUM

JUMP GAME

LCS

EDIT DISTANCE

SUBSET SUM

KNAPSACK

LAS

🔄

Bottom-Up

Top-Down

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|----|---|---|
| F | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | | |

• $F[7] = F[6] (8) + F[5] (5) = 13$

<> FIBONACCI

$O(n)$

```

1  FIBONACCI(n)
2  F[0] ← 0
3  F[1] ← 1
4  for i ← 2, ..., n do
5    F[i] ← F[i-1] + F[i-2]
6  return F[n]
```

← Step 14 / 19 →

⏮ Reset

▶ Auto Play

⏭ Jump to End

📖 Algorithm Explanation

MSS

Dimension: $R[1..n]$

Teilproblem: $R[i] :=$ maximale Summe eines Teilarrays von A , das bei Index i endet

Base Case: $R[1] = A[1]$

Rekursion: $R[i] = \max \left\{ \underbrace{A[i]}_{\text{neu starten}}, \underbrace{R[i-1] + A[i]}_{\text{weiter anhängen}} \right\} \quad \text{für } 2 \leq i \leq n$

Berechnungsreihenfolge: for $i = 2 \dots n$: berechne $R[i]$

Lösung: $\max \{ R[1], R[2], \dots, R[n], 0 \}$

Laufzeit: $O(n)$

[← Back](#)
[FIBONACCI](#)
[MAXIMUM SUBARRAY SUM](#)
[JUMP GAME](#)
[LCS](#)
[EDIT DISTANCE](#)
[SUBSET SUM](#)
[KNAPSACK](#)
[LAS](#)

1 2 3 4 5 6 7 8 9

A -2 1 -3 4 -1 2 1 -5 4

1 2 3 4 5 6 7 8 9

DP -2 1 -2 4 3 5 6 1 5

• Return $\max(\text{max_sum}, 0) = 6$.

← Step 27 / 27 → [Reset](#) [Auto Play](#) [Jump to End](#)

INPUT ARRAY

Comma separated numbers (max 9)

-2, 1, -3, 4, -1, 2, 1, -5, 4

[Bottom-Up](#)
[Top-Down](#)

<> MAXIMUM SUBARRAY SUM

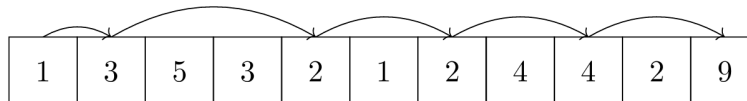
$O(n)$

```

1  MSS(A)
2  DP[1] ← A[1]
3  max_sum ← DP[1]
4  for i ← 2, ..., n do
5    DP[i] ← max(A[i], DP[i-1] + A[i])
6    max_sum ← max(max_sum, DP[i])
7  return max(max_sum, 0)

```

Jump Game



Dimension: $M[0..n]$

Teilproblem: $M[k] :=$ maximaler Index, der mit k Sprüngen erreichbar ist

Base Case: $M[0] = 1$ und $M[1] = 1 + A[1]$

2 Base Cases weil sich die Rekursion auf 2 vorherige Entries bezieht

Rekursion: $M[k] = \max \left\{ i + A[i] \mid \underbrace{M[k-2] < i \leq M[k-1]}_{\text{mögliche Indices, die wir mit } k-1 \text{ Sprüngen erreichen können (ohne Sprünge zu verschwenden)}} \right\}$ für $2 \leq k \leq n$

$i + A[i]$ $\underbrace{\hspace{1cm}}$ k -ter Sprung

Berechnungsreihenfolge: for $k=2 \dots n$: berechne $M[k]$

Lösung: minimales k , sodass $M[k] \geq n$

Laufzeit: $O(n)$

← Back

FIBONACCI

MAXIMUM SUBARRAY SUM

JUMP GAME

LCS

EDIT DISTANCE

SUBSET SUM

KNAPSACK

LAS

↺



● Target reached! Minimal jumps = 4.

← Step 16 / 16 →

⏮ Reset

▶ Auto Play

⏭ Jump to End

INPUT ARRAY

Comma separated numbers (max 9, non-negative)

2, 1, 1, 4, 2, 5, 1, 3, 7

<> JUMP GAME

$O(n)$

```

1  JUMP-GAME(A)
2  DP[0] ← 1, DP[1] ← A[1] + 1
3  k ← 1
4  while DP[k] < n do
5    k ← k + 1
6    DP[k] ← max { i + A[i] | DP[k-2] < i ≤ DP[k-1] }
7  return k

```

Längste Gemeinsame Teilfolge (LGT / LCS)

Dimension: $L[0..n][0..m]$

Teilproblem: $L[i,j] :=$ Länge einer LGT von $(a_1, a_2, \dots, a_i)$ und $(b_1, b_2, \dots, b_j)$

Base Case: $L[0,j] = 0$ für $0 \leq j \leq m$ (A ist leer)

$L[i,0] = 0$ für $0 \leq i \leq n$ (B ist leer)

Rekursion:
$$L[i,j] = \begin{cases} L[i-1,j-1] + 1 & \text{falls } a_i = b_j \\ \max\{L[i-1,j], L[i,j-1]\} & \text{falls } a_i \neq b_j \end{cases} \quad \text{für } 1 \leq i, j \leq n$$

entweder a_i oder b_j kommt nicht vor in der LGT

Berechnungsreihenfolge: for $i = 1 \dots n$
for $j = 1 \dots m$
berechne $L[i,j]$

Lösung: $L[n,m]$

Laufzeit: $O(n \cdot m)$

(das Skript "versteckt" die Base Cases in der Rekursion)

| | ε | P | F | E | R | D |
|---|---|---|---|---|---|---|
| ε | 0 | 0 | 0 | 0 | 0 | 0 |
| Z | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 1 | 1 | 1 |
| B | 0 | 0 | 0 | 1 | 1 | 1 |
| R | 0 | 0 | 0 | 1 | 2 | 2 |
| A | 0 | 0 | 0 | 1 | 2 | 2 |

• Backtracking from (0,2) to (5,5). Highlighted cells are LCS matches.

← Step 53 / 53 → [Reset](#) [Auto Play](#) [Jump to End](#)

CONFIGURATION

STRING A (MAX 7)

ZEBRA

STRING B (MAX 7)

PFERD

[Bottom-Up](#)

[Top-Down](#)

<> **LCS**

$O(n \cdot m)$

```

1  LCS(A, B)
2  n ← len(A), m ← len(B)
3  initialize DP[0...n][0...m] with 0
4  for i ← 1, ..., n do
5    for j ← 1, ..., m do
6      if A[i] == B[j] then DP[i][j] ← 1 + DP[i-1][j-1]
7      else DP[i][j] ← max(DP[i-1][j], DP[i][j-1])
8  return DP[n][m]
```

Editierdistanz

- **Einfügen**: Ein Zeichen an einer beliebigen Position in den String einfügen.
- **Löschen**: Ein Zeichen an einer beliebigen Position aus dem String löschen.
- **Ersetzen**: Ein Zeichen an einer beliebigen Position durch ein anderes Zeichen ersetzen.

Dimension: $ED[0..n][0..m]$

Teilproblem: $ED[i,j] :=$ Editierdistanz zwischen $(a_1, a_2, \dots, a_i)$ und $(b_1, b_2, \dots, b_j)$

Base Case: $ED[0,j] = j$ für $0 \leq j \leq m$ (A ist leer)

$ED[i,0] = i$ für $0 \leq i \leq n$ (B ist leer)

Rekursion: $ED[i,j] = \begin{cases} ED[i-1,j-1] & \text{falls } a_i = b_j \\ 1 + \min \{ ED[i-1,j], ED[i,j-1], ED[i-1,j-1] \} & \text{falls } a_i \neq b_j \end{cases}$

wir können das letzte Zeichen einfach ignorieren, weil es bei beiden gleich ist

wir machen 1 Operation

a_i aus A löschen ($\Leftrightarrow a_i$ zu B hinzufügen)

b_j zu A hinzufügen ($\Leftrightarrow b_j$ aus B löschen)

a_i durch b_j ersetzen ($\Leftrightarrow b_j$ durch a_i ersetzen)

Berechnungsreihenfolge: for $i = 1 \dots n$

Lösung: $ED[n,m]$ for $j = 1 \dots m$ berechne $ED[i,j]$

Laufzeit: $O(n \cdot m)$

| | ϵ | A | D | L | E | R |
|------------|------------|---|---|---|---|---|
| ϵ | 0 | 1 | 2 | 3 | 4 | 5 |
| E | 1 | 1 | 2 | 3 | 3 | 4 |
| U | 2 | 2 | 2 | 3 | 4 | 4 |
| L | 3 | 3 | 3 | 2 | 3 | 4 |
| E | 4 | 4 | 4 | 3 | 2 | 3 |

• Backtracking complete. Path highlighted.

EDIT DISTANCE

$O(n \cdot m)$

```

1  EDIT-DISTANCE(A, B)
2  n ← len(A), m ← len(B)
3  initialize DP[0...n][0...m], fill base cases
4  for i ← 1, ..., n do
5    for j ← 1, ..., m do
6      if A[i] == B[j]
7        DP[i][j] ← DP[i-1][j-1]
8      else
9        DP[i][j] ← 1 + min(
10         DP[i-1][j], DP[i][j-1], DP[i-1][j-1]
11      )
12  return DP[n][m]
```

Programmier-Muster

Bottom-up:

- 1) DP-Array erstellen (im Pseudo code weggelassen)
- 2) BC eintragen
- 3) for-Schleife(n) anfangen
- 4) Rekursionsgleichung, direkt im DP table eintragen
- 5) ausserhalb der Schleife(n), die Lösung auslesen und returnen

```

1  FIBONACCI(n)
2  F[0] ← 0
3  F[1] ← 1
4  for i ← 2,...,n do
5    F[i] ← F[i-1] + F[i-2]
6  return F[n]
```

```

1  MSS(A)
2  DP[1] ← A[1]
3  max_sum ← DP[1]
4  for i ← 2,...,n do
5    DP[i] ← max(A[i], DP[i-1] + A[i])
6  max_sum ← max(max_sum, DP[i]) → macht das auslesen
7  return max(max_sum, 0)           der Lösung einfacher
```

```

1  LCS(A, B)
2  n ← len(A), m ← len(B)
3  initialize DP[0...n][0...m] with 0
4  for i ← 1,...,n do
5    for j ← 1,...,m do
6      if A[i] == B[j] then DP[i][j] ← 1 + DP[i-1][j-1]
7      else DP[i][j] ← max(DP[i-1][j], DP[i][j-1])
8  return DP[n][m]
```

↳ this solves the base case implicitly

Top-down:

- 1) DP-Array erstellen und alles mit -1 füllen
- 2) compute(..) aufrufen und Lösung auslesen
- 3) compute Funktion beginnen
- 4) Falls es schon berechnet wurde, direkt returnen (= memoization)
Reihenfolge egal
- 5) Base cases mit if's abfangen
- 6) Sonst Rekursion mit compute(..) Aufrufen
- 7) Den eben berechneten Wert returnen

```

1  FIBONACCI(n)
2  initialize F[0...n] with -1
3  return compute(n)
4  compute(i)
5  if F[i] ≠ -1: return F[i]
6  if i ≤ 1: F[i] ← i
7  else F[i] ← compute(i-1) + compute(i-2)
8  return F[i]
```

```

1  MSS(A)
2  initialize DP[1...n] with -1
3  compute(n)
4  return max(max(DP), 0)
5  compute(i)
6  if i == 1 return A[1]
7  if DP[i] ≠ -1 return DP[i]
8  DP[i] ← max(A[i], compute(i-1) + A[i])
9  return DP[i]
```

```

1  LCS(A, B)
2  initialize DP[0...n][0...m] with -1
3  return compute(n, m)
4  compute(i, j)
5  if i == 0 or j == 0 return 0
6  if DP[i][j] ≠ -1 return DP[i][j]
7  if A[i] == B[j]
8    DP[i][j] ← 1 + compute(i-1, j-1)
9  else DP[i][j] ← max(compute(i-1, j), compute(i, j-1))
10 return DP[i][j]
```

Aufgaben:

Ein Einbrecher plant, einige Häuser in einer Strasse mit n Häusern auszurauben.

Haus i hat einen bestimmten Geldbetrag $\text{money}[i]$, der gestohlen werden kann, wobei $1 \leq i \leq n$.

Es gibt eine Sicherheitsregel: Er darf nicht zwei direkt benachbarte Häuser ausrauben, sonst geht der Alarm los.

Was ist der maximale Geldbetrag, den er stehlen kann, ohne den Alarm auszulösen?

Beispiel: $\text{money} = [2, 7, 9, 3, 1]$

- Option A: Haus 1 + Haus 3 + Haus 5 $\rightarrow$ Geldbetrag: $2 + 9 + 1 = 12$
- Option B: Haus 1 + Haus 4 $\rightarrow$ Geldbetrag: $2 + 3 = 5$
- ...
- $\rightarrow$ Der maximale Geldbetrag ist 12.

Dimension:

Teilproblem:

Base Case:

Rekursion:

Berechnungsreihenfolge:

Lösung:

Laufzeit:

You are given an integer array `cost` where `cost[i]` is the cost of i -th step on a staircase, where $0 \leq i \leq n - 1$.

Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index 0, or the step with index 1.

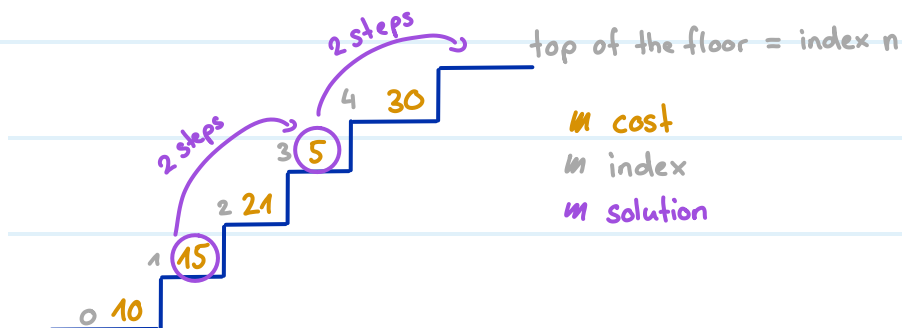
Return the minimum cost to reach the top of the floor.

Example: `cost = [10, 15, 21, 5, 30]`

Output: 20

Explanation: You will start at index 1.

- Pay 15 and climb two steps to reach index 3.
- Pay 5 and climb two steps to reach the top.
- The total cost is $15 + 5 = 20$.



Dimension:

Teilproblem:

Base Case:

Rekursion:

Berechnungsreihenfolge:

Lösung:

Laufzeit:

The next year has n days. You need to travel to ETH with SBB on some of these days. The boolean array $\text{travel}[1\dots n]$ contains a 1 if we need to travel on day i , and a 0 otherwise.

There are three types of tickets to buy: They are valid for 1, 7, or 30 days and their cost is given by C_1 , C_7 and C_{30} .

Find the minimum cost if you want to have a valid ticket on every day you need to travel.

Example: $\text{travel} = [1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1]$ and $n = 11$ and $C_1 = 1$, $C_7 = 5$, $C_{30} = 10$

We buy a 1-day ticket on day 1, a 7-day ticket on day 3, and a 1-day ticket on day 11.

Thus the minimum total cost is $C_1 + C_7 + C_1 = 1 + 5 + 1 = 7$

Dimension:

Teilproblem:

Base Case:

Rekursion:

Berechnungsreihenfolge:

Lösung:

Laufzeit: