Quiz

A polynomial-time algorithm for Subset Sum would imply that $P = NP$.

(True)

False

P = Menge der Entscheidungsprobleme, die in polynomieller Zeit O(poly(n)) lösbar sind, wobei n die Länge des Inputs ist.

NP = Menge der Entscheidungsprobleme, für die man eine gegebene Lösung in polynomieller Zeit verifizieren kann

Bsp Input: A = [2, 3, 7, 1], b = 8

Lösung: 7 + 1 = 8 → in polynomieller Zeit verifizierbar

The knapsack approximation algorithm from the lecture may exceed the capacity W by up to εW .

True

(False)

Nur Profite p_i werden gerundet, Gewichte w_i bleiben exakt

For every array A, any correct algorithm must read all entries of A to solve the Jump Game.

Bsp. n = 100, A = [100, 2, 3, ...]

muss nicht gelesen werden

True

(False)

The lecture's edit-distance algorithm for arrays $A[1, \ldots, n]$ and $B[1, \ldots, m]$ runs in $O(mn)$ time. How is this running time classified? Select one answer.

(a.) polynomial

b. pseudopolynomial, but not polynomial

c. neither pseudopolynomial nor polynomial

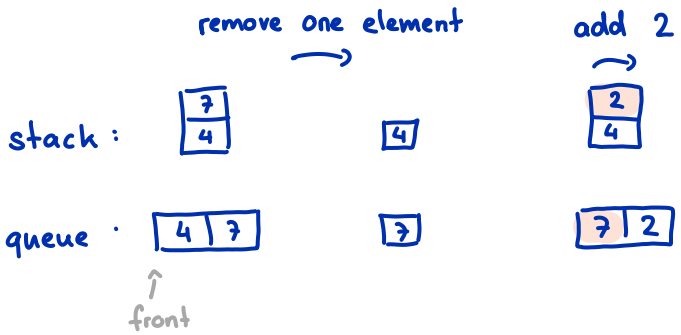
An initially empty stack and an initially empty queue each receive $4, 7$, in that order. Each removes one element, then receives the element 2. What do the next pop and dequeue return, respectively? Select one answer.

a. Stack: 4, queue: 7.

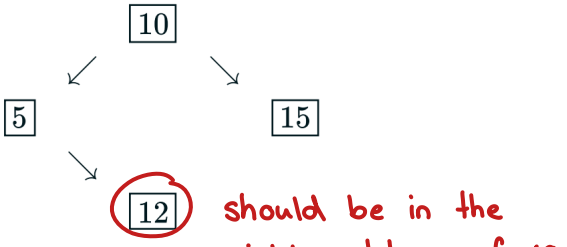
b. Stack: 4, queue: 2.

c. Stack: 2, queue: 7.

d. Stack: 2, queue: 2.



The following tree is a valid binary search tree.



should be in the right subtree of 10

True

False

In the coin-change DP from Homework 6, $f[N]$ is the minimum number of coins needed to pay N. With coin values $1, 4, 6$, which earlier entries does the recurrence use to compute $f[8]$? Select one answer.

a. $f[5], f[6], f[7]$

b. $f[1], f[4], f[6]$ last coin is either 1, 4 or 6.

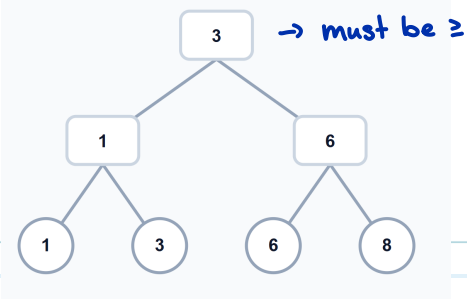
c. $f[2], f[4], f[7]$ what remains is either $8-1=7$, $8-4=4$, or $8-6=2$

A 2-3 tree has a root with three leaf children containing keys $1, 3, 8$. After inserting 6, which is a possible separator in the new root? Select one answer.

a. 7

b. 2

c. 4



→ must be ≥ 3 and ≤ 5

For knapsack, let OPT be an optimal feasible subset for profits p_i, and let $\widetilde{OPT}$ be an optimal feasible subset for the lecture's modified profits $\widetilde{p}_i$. Which inequality is guaranteed? Select one answer.

a. $\sum_{i \in \widetilde{OPT}} \widetilde{p}_i \geq \sum_{i \in OPT} \widetilde{p}_i$

b. $\sum_{i \in \widetilde{OPT}} \widetilde{p}_i \leq \sum_{i \in OPT} p_i$

c. $\sum_{i \in \widetilde{OPT}} p_i \leq \sum_{i \in OPT} p_i$

$\widetilde{OPT}$ ist per Definition die optimale Lösung für die $\widetilde{p}_i$'s
Deshalb ist es ≥ jede andere mögliche Lösung für die $\widetilde{p}_i$'s

OPT ist per Definition die optimale Lösung für die p_i's
Deshalb ist es ≥ jede andere mögliche Lösung für die p_i's

A max-heap is stored as $A[1, \ldots, 7] = (9, 6, 8, 1, 4, 7, 5)$. What is $A[3]$ after one ExtractMax operation?

Answer: 7



Let $L(i, j)$ be the length of a longest common subsequence of $A[1, \ldots, i]$ and $B[1, \ldots, j]$. For $i, j \geq 1$, decide whether each statement is always true.

Wahr	Falsch	
	✗	If $A[i] \neq B[j]$, then $L(i, j) = L(i-1, j-1)$.
✗		$L(i, j) \geq L(i-1, j)$.
✗		$L(i, j) \leq \min\{i, j\}$.

Counterexample: A = [1, 2], B = [2, 3]

if the input is longer, the LCS can not become shorter

All elements of an LCS must be contained in A and B
So if A only has i elements, the LCS cannot be longer than i
(same for B)

(schwierig)

Let $n \geq 4$ be a power of two. How many calls to f does $g(n)$ make? Select one answer.

```
function g(n):
    if n = 1 : return 1
    x ← g(n/2)
    for i = 1, ..., x : f()
    return 2x
```

try n=4: $G(4) = G(2) + g(2) \cdot 1$

$= (G(1) + g(1) \cdot 1) + g(2) \cdot 1$

$= 0 + 1 + 2$

$= 3$

a. $\log_2 n$

b. $n \log_2 n$

} falsch für n=4

c. $n - 1$

Alice chooses a secret integer $x \in \{1, \ldots, 8\}$. Bob may only ask "Is $x > y$?", choosing an integer y for each question. Decide whether each statement is true or false.

Wahr	Falsch		
X		If he chooses all questions before receiving any answers, Bob needs at least seven questions to guarantee determining the parity.	
	X	Bob can always determine the parity of x using at most two questions. (if even or odd)	
X		Whenever Alice's answers determine the parity with certainty, they also determine x uniquely.	

Bob hat immer ein Suchintervall der Form [a,. , b]

Discussion of Exercise Sheet 6

Exercise 6.3 *Introduction to dynamic programming* **(2 bonus points)**.

Consider the recurrence

$$A_1 = 1$$
$$A_2 = 2$$
$$A_3 = 3$$
$$A_4 = 4$$
$$A_n = A_{n-1} + A_{n-3} + 2A_{n-4} \text{ for } n \geq 5.$$

(a) Provide a recursive function (using pseudo code) that computes A_n for $n \in \mathbb{N}$. You do not have to argue correctness.

Algorithm 1 $A(n)$

```
if n ≤ 4 then
    return n
else
    return A(n − 1) + A(n − 3) + 2A(n − 4)
```

Bemerkung: A(n-4) + A(n-4) ist nicht dasselbe wie 2·A(n-4)

(b) Lower bound the run time of your recursion from (a) by $\Omega(C^n)$ for some constant $C > 1$.

Assume n is divisible by 4

Number of calls: $T(n) \geq T(n-1) + T(n-3) + T(n-4)$ for $n \geq 5$

$\geq T(n-4) + T(n-4) + T(n-4)$

$= 3T(n-4)$

$= 3 \cdot 3\ T(n-8)$

$= 3\ 3\ 3 \cdot T(n-12)$

$= 3^{\frac{n}{4}-1}\ T(4)$

$= 3^{\frac{n}{4}}\ \frac{1}{3}\ T(4)$

$= (3^{\frac{1}{4}})^n\ \frac{1}{3}\ T(4)$

$\geq \Omega(C^n)$ for $C = 3^{\frac{1}{4}} \approx 1.32$

$\frac{1}{3} T(4)$ ist ein konstanter Faktor

(c) Improve the run time of your algorithm using memoization. Provide pseudo code of the improved algorithm and analyze its run time. You do not need to prove correctness of the algorithm.

Algorithm 2 Compute A_n using memoization

```
memory ← 1-dimensional array of length n, filled with -1
function A_MEM(n)
    if memory[n] ≠ −1 then                          ▷ If A_n is already computed.
        return memory[n]                            } memoization
    if n ≤ 4 then
        (memory[n] ← n) unnecessary                 } base case
        return n
    else
        A_n ← A_Mem(n − 1) + A_Mem(n − 3) + 2A_Mem(n − 4)   } recursion
        memory[n] ← A_n
        return A_n
```

When calling A_Mem(n), each A_k for $1 \leq k \leq n$ is computed at most once and then stored in memory. Thus the run time of A_Mem(n) is $\Theta(n)$.

(d) Compute A_n using bottom-up dynamic programming and state the run time of your algorithm. In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Dimension: DP[1 .. n]

Teilproblem: DP[i] := A_i

Base Case: DP[1] = 1, DP[2] = 2, DP[3] = 3, DP[4] = 4

Rekursion: DP[i] = DP[i-1] + DP[i-3] + 2 DP[i-4] für $5 \leq i \leq n$

Korrektheit folgt direkt aus der Definition von A_k

Berechnungsreihenfolge: for i = 5.. n: berechne DP[i]

Lösung: DP[n]

Laufzeit: $\Theta(n)$

Neue Gruppen:

Group	Names	Emails
Group 1	Nico Reali Besart Memisi	nreali@student.ethz.ch bmemisi@student.ethz.ch
Group 2	Elaine Banaag Aquino Paul Iurian	ebanaag@student.ethz.ch piurian@student.ethz.ch
Group 3	Lauren Thai Gréta Frei	lathai@student.ethz.ch grfrei@student.ethz.ch
Group 4	Marc Leonard Overbeck Nicolas Fierz	moverbeck@student.ethz.ch nfierz@student.ethz.ch
Group 5	Noel Hobi Max Schaffner	hobino@student.ethz.ch schaffnerma@student.ethz.ch
Group 6	Nelio Gautschi Sih-Cheng Chen	ngautsch@student.ethz.ch chensih@student.ethz.ch
Group 7	Noa Kündig Charlyne Dippon	nkuendig@student.ethz.ch cdippon@student.ethz.ch
Group 8	Lucien Gees Silvan Wohlgensinger Zhentian Qi	lugees@student.ethz.ch swohlgensing@student.ethz.ch zhenqi@student.ethz.ch
Group 9	Lucien Isler Matin Kamil Saih	islerlu@student.ethz.ch kamilm@student.ethz.ch
Group 10	Raghav Tandon Floria Schwedt	tandonr@student.ethz.ch fschwedt@student.ethz.ch
Group 11	Colin Jenny Simon Spycher	cojenny@student.ethz.ch sispycher@student.ethz.ch
Group 12	Benjamin Stupf Durvish Battini	bstupf@student.ethz.ch dbattini@student.ethz.ch

Subset Sum (≠ Subarray Sum)

Gegeben: Array $A[1,..,n]$ und target b

Gesucht: Eine Teilmenge von Indices $I \subseteq \{1,..,n\}$ sodass $\sum_{i \in I} A[i] = b$

Dimension: $DP[0 \dots n][0 \dots b]$

Teilproblem:
$$DP[i,s] := \begin{cases} 1 & \text{falls es eine Teilmenge von Indices } I \subseteq \{1,..,i\} \text{ gibt, sodass } \sum_{j \in I} A[j] = s \\ 0 & \text{sonst} \end{cases}$$

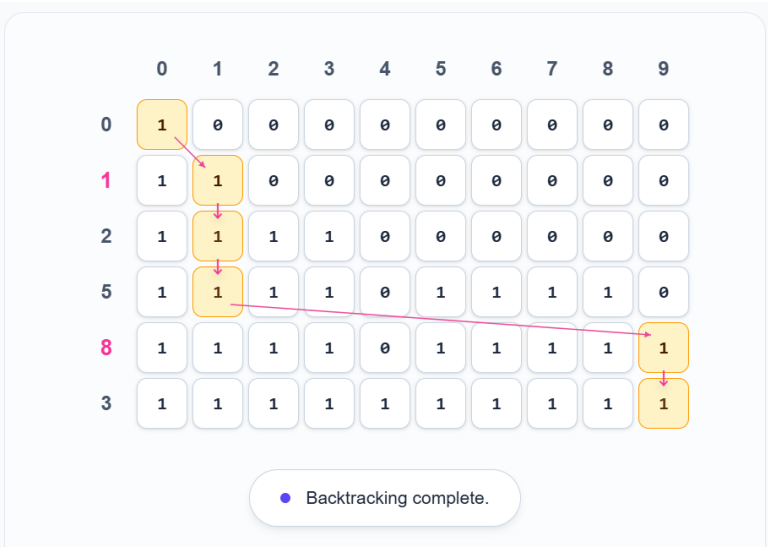
Base Case:
$DP[0,0] = 1$ → keine Elemente summieren zu 0 ✓

$DP[0,s] = 0$ für $1 \leq s \leq n$ → keine Elemente können nicht zu $s \geq 1$ summieren

Rekursion:
$$DP[i,s] = \underbrace{DP[i-1,s]}_{\text{wir brauchen } A[i] \text{ nicht und können ohne } A[i] \text{ zu } s \text{ summieren}} \vee \overbrace{\underbrace{DP[i-1,\ s-A[i]]}_{\text{wir brauchen } A[i]. \text{ Dann muss es eine Teilmenge der ersten } i-1 \text{ Elemente geben, die zu } s-A[i] \text{ summieren}}}^{:= 0 \text{ falls } s-A[i] < 0} \quad \text{für } 1 \leq i \leq n, 0 \leq s \leq b$$

Berechnungsreihenfolge:
```
for i = 1.. n
    for s = 0... b:
        berechne DP[i,s]
```

Lösung: $DP[n,b]$ sagt uns, ob eine Lösung existiert Falls ja, können wir die gesuchte Teilmenge der Elemente durch backtracking finden



Bottom-Up | Top-Down

SUBSET SUM $\mathcal{O}(n \cdot target)$

```
SUBSET-SUM(A, target)
  initialize DP[0...n][0...target]
  DP[0...n][0] = 1
  DP[0][1...target] = 0
  for i ← 1,...,n do
    for s ← 1,...,target do
      DP[i][s] ← DP[i-1][s]
      if A[i] ≤ s
        DP[i][s] ← DP[i][s] or DP[i-1][s - A[i]]
  return DP[n][target]
```

Laufzeit: $\Theta(n \cdot b)$ Bsp. $b = 4$, $b = n^2$, $b = n^{37} \log n$

Pseudopolynomiell: Ob der Algorithmus in polynomieller oder exponentieller Zeit läuft, hängt von b ab Bsp $b = 2^n$, $b = 3^n$

Anmerkung zur Notation: $DP[0 \dots n][0 \dots b] = DP[0 \dots n, 0. b] = (n+1) \times (b+1)$ Array

$DP[i,s] = DP[i][s] = DP(i,s) = T[i,s] = memo[i,s]$

$$DP[i,s] := \begin{cases} 1 & \text{falls } A \\ 0 & \text{sonst} \end{cases} = \begin{cases} \text{true} & \text{falls } A \\ \text{false} & \text{sonst} \end{cases} = \begin{cases} 1 & \text{falls } A \\ 0 & \text{falls } \neg A \end{cases}$$

Begründungssatz der Laufzeit: (für Aufgaben und Prüfung) Es gibt $(n+1)(b+1)$ DP-Einträge, wobei jeder einzelne Eintrag in $\Theta(1)$ berechnet werden kann, was zu einer Gesamtlaufzeit von $\Theta(n\,b)$ führt (Die Lösung kann in $O(1)$ ausgelesen werden)

Rucksack Problem

Gegeben: Gewichtslimit W, n items mit Gewicht $w_i \in \mathbb{N}$ und Profit $p_i \in \mathbb{N}$

Gesucht: Eine Teilmenge von Indices $I \subseteq \{1, .., n\}$ sodass das Gewichtslimit $\sum_{i \in I} w_i \leq W$ eingehalten wird und der Gesamtprofit $\sum_{i \in I} p_i$ dabei maximal ist.

Dimension: $P[0 .. n][0 .. W]$

Teilproblem: $P[i, w]$:= maximaler Profit, wenn wir nur die ersten i items betrachten und dabei das Gewichtslimit w einhalten

Base Case: $P[0, w] = 0$ für $0 \leq w \leq W$ → 0 items ergeben bei egal welchem Gewichtslimit genau 0 Profit

Rekursion:

$$P[i, w] = \max \{ \underbrace{P[i-1, w]}_{\text{:= 0 falls } w - w_i < 0}, \underbrace{P[i-1, w - w_i] + p_i}_{} \} \quad \text{für } 1 \leq i \leq n, \; 0 \leq w \leq W$$

Entweder wir nehmen item i nicht:
Profit und Gewicht bleibt gleich wie bei $i-1$ items

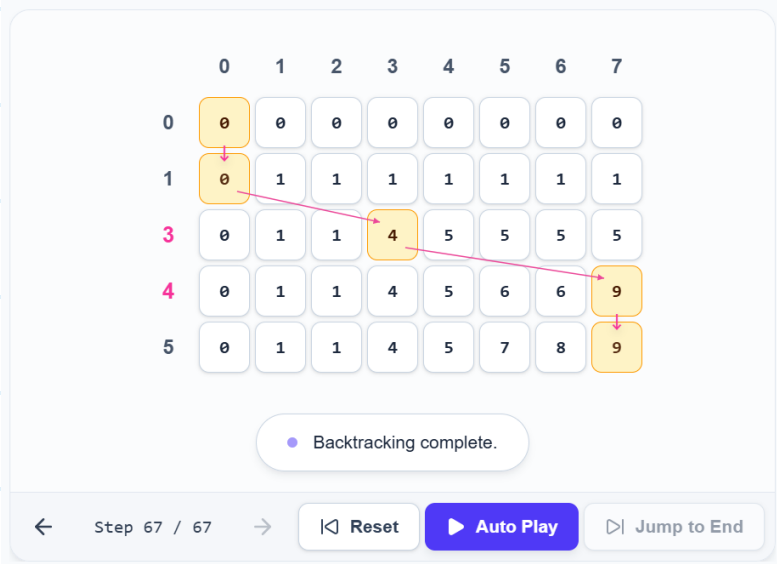
Oder wir nehmen item i:
Profit erhöht sich um p_i
Die ersten $i-1$ items müssen dafür aber das Gewichtslimit $w - w_i$ einhalten

Berechnungsreihenfolge:

```
for i = 1 .. n
    for w = 0 .. W:
        berechne P[i, w]
```

Lösung: $P[n, W]$ ist der maximale Profit. Wir können die gesuchte Teilmenge der Elemente durch backtracking finden

Laufzeit: $\Theta(n W)$, pseudopolynomiell



KNAPSACK $\mathcal{O}(n \cdot W)$

```
1  KNAPSACK(W, wt, val, n)
2    initialize DP[0...n][0...W]
3    for i ← 0,...,n do
4      for w ← 0,...,W do
5        if i == 0 or w == 0
6          DP[i][w] ← 0
7        else if wt[i] ≤ w
8          DP[i][w] ← max(val[i] + DP[i-1][w-wt[i]],
8                         DP[i-1][w])
10       else
11         DP[i][w] ← DP[i-1][w]
12   return DP[n][W]
```

Alternative Lösung:

Dimension: $G[0..n][0..P]$ wobei $P = \sum_{i=1}^{n} p_i$

Teilproblem: $G[i, w]$:= minimales Gewicht, wenn wir nur die ersten i items betrachten um mindestens Profit p zu erreichen ($:= \infty$ falls nicht möglich)

Base Case: $G[0,0] = 0$

$G[0, p] = \infty$ für $1 \leq p \leq P$ → es ist unmöglich mit 0 items, Profit $p \geq 1$ zu erreichen

Rekursion:

$$G[i, p] = \min \{ \underbrace{G[i-1, p]}_{\text{wir nehmen item } i \text{ nicht}}, \underbrace{G[i-1, p-p_i] + w_i}_{\text{wir nehmen item } i} \} \quad \text{für } 1 \leq i \leq n, \; 0 \leq p \leq P$$

$:= 0$ falls $p - p_i < 0$

Berechnungsreihenfolge:

```
for i = 1..n
    for p = 0..P:
        berechne G[i, p]
```

Lösung: Grösstes p sodass $G[n, p] \leq W$ Wir können die gesuchte Teilmenge der Elemente durch backtracking finden

Laufzeit: $\Theta(n\,P)$, pseudopolynomiell

Längste aufsteigende Teilfolge (LAT / LAS)

Gesucht: Länge einer längsten aufsteigenden Teilfolge von $A[1..n]$

Dimension: $M[1..n][1..n]$

Teilproblem: $M[i, \ell]$:= kleinstmögliche Endung einer aufsteigenden Teilfolge von $A[1..i]$ der Länge ℓ. (:= ∞ falls es keine solche Teilfolge gibt)

Base Case: $M[1,1] = A[1]$ → aufsteigende Teilfolge der Länge 1 aus $A[1]$ ist $A[1]$

$M[1,\ell] = \infty$ für $2 \leq \ell \leq n$ → es ist unmöglich mit nur einem Element eine Teilfolge der Länge $\ell \geq 2$ zu haben

Rekursion:

$$M[i,\ell] = \begin{cases} \min\{M[i-1,\ell], A[i]\} & \text{falls } \ell = 1 \text{ oder } M[i-1,\ell-1] < A[i] \\ M[i-1,\ell] & \text{sonst} \end{cases}$$

wir benutzen $A[i]$ nicht — wir hängen $A[i]$ an

→ wir können $A[i]$ anhängen

für $2 \leq i \leq n, 1 \leq \ell \leq n$

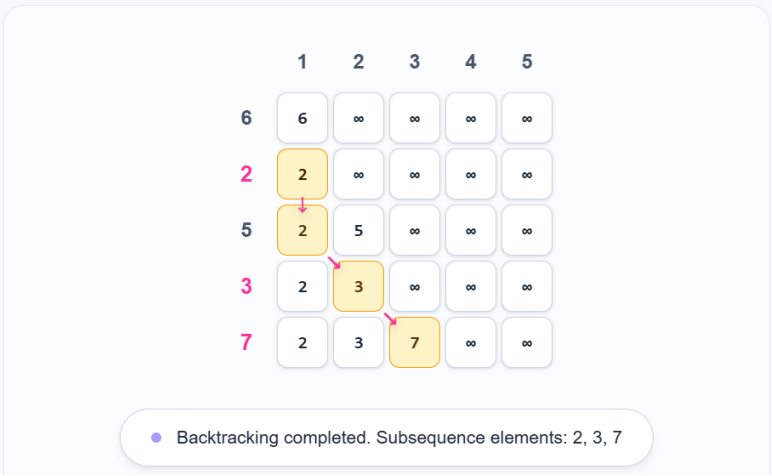
Berechnungsreihenfolge:

```
for i = 2 .. n
    for ℓ = 1 .. n:
        berechne M[i, ℓ]
```

Lösung: Grösstes ℓ sodass $M[n,\ell] < \infty$

Laufzeit: $\Theta(n^2)$

→ Lecture



LAS $\mathcal{O}(n^2)$

```
1  LAS(A)
2    DP[1][1] ← A[1]; DP[1][2...n] = ∞
3    for i ← 2,...,n do
4      for l ← 1,...,n do
5        exclude ← DP[i-1][l]
6        include ← ∞
7        if l = 1 or DP[i-1][l-1] < A[i]
8          include ← A[i]
9        DP[i][l] ← min(exclude, include)
10   return max l where DP[n][l] < ∞
```

Verbesserung in $O(n \log n)$ mit Binary Search:

Idee: es verändert sich nur ein DP-Eintrag pro Reihe

Bsp. $A = [3, 5, 4, 8, 9, 7, 2, 6]$

~~3~~, ~~∞~~, ~~∞~~, ~~∞~~, ∞, ∞, ∞, ∞

2 ~~5~~ ~~8~~ 9

4 ~~7~~

6

$B = [4, 6, 2, 7, 5, 1, 3, 9]$

4 ∞ ∞ ∞ ∞ ∞ ∞ ∞

Wie Teilproblem finden?

1) Kategorie bestimmen:

- Optimierungsproblem (Bsp MSS, Jump Game, Edit distance, LAT, Rucksack)
  - → Was ist der maximale / minimale / kleinste / grösste / kürzeste / längste ...
- Entscheidungsproblem (Bsp Subset Sum)
  - → Gibt es eine Teilmenge/Teilarray/Teilfolge .. sodass ...
- Berechnungsproblem (Bsp Fibonacci)
  - → Was ist die Anzahl / das Ergebnis, wenn ...

2.) zusätzliche Bedingungen hinzufügen, die das Problem einfacher machen:

- Für Arrays, Strings, Mengen, und alles was man indexieren kann:
  - → ... wenn wir nur die ersten i Elemente betrachten
  - → ... wenn wir nur die ersten i Elemente betrachten, und Element i zwingend gewählt werden muss
  - → ... wenn wir bei 1 beginnen
  - → ... wenn wir bei 1 beginnen, und Element i zwingend gewählt werden muss
  - → ... wenn wir bei 1 beginnen und bei j enden
- Für Geld, Zeit, Tickets, und alles was man ausgeben kann:
  - → ... und wir mit t Geld/Zeit/Tickets starten
- Für Grenzen und Limits, die man einhalten muss:
  - → ... und wir dabei das Zeit-/Gewichts/-Limit L einhalten

Aufgaben:

You are walking up a stair. You can can take either 1 step or 2 steps at a time. How many different ways can you climb a stair of height n.

Dimension: DP[1 ... n]

Teilproblem: DP[i] := number of different ways to climb a stair of height i.

Base Case: DP[1] = 1 und DP[2] = 2

Rekursion: $DP[i] = \underbrace{DP[i-2]}_{\text{last is a double step}} + \underbrace{DP[i-1]}_{\text{last is a single step}}$ für $3 \leq i \leq n$

Berechnungsreihenfolge: for i = 3 .. n berechne DP[i]

Lösung: DP[n]

Laufzeit: $\Theta(n)$

You want to choose a new pin for your computer. You are allowed to use up to n digits, where each digit is between 0 and 9. How many distinct PINs can you create?

Dimension: DP[1 ... n]

Teilproblem: DP[i] := number of different PINs with exactly i digits

Base Case: DP[1] = 10

Rekursion: $DP[i] = \underbrace{DP[i-1]}_{\text{Number of PINs of length } i-1} \cdot 10$ für $2 \leq i \leq n$

Berechnungsreihenfolge: for i = 2 .. n berechne DP[i]

Lösung: $\sum_{i=1}^{n} DP[i]$

Laufzeit: $\Theta(n)$

You want to build a vertical tower of exact height n units. You have an unlimited supply of three types of bricks:

- **Size 1 bricks** (height of 1 unit)
- **Size 2 bricks** (height of 2 units)
- **Size 4 bricks** (height of 4 units)

Two towers are considered **distinct** if the sequence of brick sizes stacked from bottom to top differs. For example, placing a 1-brick then a 2-brick is different from placing a 2-brick then a 1-brick.

Find the total number of distinct ways to build a tower of height n.

Dimension: DP[1..n]

Teilproblem: DP[i] := number of distinct ways to build a tower of height i

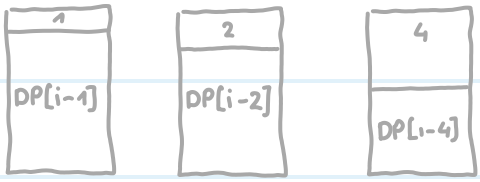
Base Case: DP[1] = 1

DP[2] = 2

DP[3] = 3

DP[4] = 6

Rekursion: DP[i] = DP[i-1] + DP[i-2] + DP[i-4] für $5 \leq i \leq n$



Berechnungsreihenfolge: for i = 5 ... n berechne DP[i]

Lösung: DP[n]

Laufzeit: $\Theta(n)$

~ 20 min Zeit

/ 8 P von 100 Punkten

Theory Task T3.

An array of non-negative integers $A = [a_1, \ldots, a_n]$ is called *summy* if and only if, for all $i \in \{2, \ldots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \ldots, i-1\}$ such that $a_i = \sum_{j \in I} a_j$. In other terms, every integer in the array except the first one must be the sum of (distinct) integers that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime (where $\max A$ means the maximum value of entries in A). Address the following aspects in your solution:

↳ Hinweis für die Dimension

1) *Definition of the DP table*: What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry ?
2) *Computation of an entry*: How can an entry be computed from the values of other entries ? Specify the base cases, i.e., the entries that do not depend on others.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and $\max A$, and justify your answer.

↳ immer wenn ihr das lest, müsst ihr mindestens einen Satz schreiben

Dimension: $DP[0 \ldots n][0 \ldots \max A]$

Teilproblem: $DP[i, k] := \begin{cases} \text{true} & \text{falls es eine Teilmenge } I \subseteq \{1, \ldots, i\} \text{ gibt, sodass } \sum_{j \in I} a_j = k \\ \text{false} & \text{sonst} \end{cases}$

Base Case: $DP[0,0] = \text{true}$ → leere Summe ergibt 0, ist möglich

$DP[0,k] = \text{false}$ für $1 \le k \le \max A$ → nicht möglich

Rekursion: $DP[i,k] = DP[i-1, k] \vee DP[i-1, k-a_i]$ für $1 \le i \le n$, $0 \le k \le \max A$

$DP[i-1, k-a_i]$:= false falls $k - a_i < 0$

wir brauchen a_i nicht und können ohne a_i zu k summieren

wir brauchen a_i. Dann muss es eine Teilmenge der ersten $i-1$ Elemente geben, die zu $k-a_i$ summieren

Berechnungsreihenfolge:

```
for i = 1 .. n
    for k = 0 ... maxA
        berechne DP[i,k]
```

Lösung:

$$= \begin{cases} \text{true} & \text{falls } DP[i-1, a_i] = \text{true} \text{ für alle } 2 \leq i \leq n \\ \text{false} & \text{sonst} \end{cases}$$

Laufzeit: $\Theta(n \cdot maxA)$ → pseudopolynomiell

→ Ex 7.1

You are a history student, "Alex," standing at the entrance of the "Grand Boulevard of Museums" (museum 0). You have a research paper due tomorrow, and your goal is to acquire the maximum amount of "knowledge" possible.

You have two limited resources:

- A starting budget of M dollars.
- A total time limit of T hours.

The boulevard has N museums, numbered 0 to $N-1$. For each museum i, you are given three pieces of information:

- `ticket_cost[i]` : The cost in dollars to buy a ticket.
- `visit_time[i]` : The time in hours it takes to see the exhibit.
- `knowledge[i]` : The knowledge points (KP) you gain from the exhibit.

The Rules of the Tour

When you are at the entrance of museum i, you have two choices:

1. **Visit the Museum:**
   - You **can only** do this if you have enough money (`>= ticket_cost[i]`) AND enough time (`>= visit_time[i]`).
   - If you visit, you pay the `ticket_cost[i]` , spend the `visit_time[i]` , and gain the `knowledge[i]` .
   - After your visit, you immediately move to the front of the next museum ($i+1$).
2. **Skip the Museum:**
   - You **can only** do this if you have at least **1 hour** of time remaining.
   - Skipping costs you no money, but it costs **1 hour** of time to walk to the next museum.
   - You gain **0** knowledge.
   - You move to the front of the next museum ($i+1$).

Your tour ends when you have considered the last museum (i.e., you arrive at index N) or when you run out of time and cannot even afford to skip (i.e., you have 0 time left).

What is the absolute maximum amount of knowledge you can gain?

Example

- $N = 3$ museums
- $M = 10$ (dollars)
- $T = 9$ (hours)

Museum (i)	ticket_cost[i]	visit_time[i]	knowledge[i]
skip → 0	5	2	7
visit → 1	4	3	3
visit → 2	6	5	8

Dimension: $DP[0 \dots n][0 \dots T][0 \dots M]$

Teilproblem: $DP[i, t, m]$:= maximum knowledge Alex can gain, starting at museum i with t hours left and m dollars

Base Case: $DP[N, t, m] = 0 \quad \text{for } 0 \le t \le T,\ 0 \le m \le M$

→ Museums are at indexes $0 \dots N-1$. At N, Alex is done with his trip. Thus he can gain no more knowledge

Rekursion:

$$DP[i, t, m] = \max \begin{cases} DP[i+1, t-1, m] & \text{:= 0 falls } t-1 < 0 \quad \text{→ überspringen} \\ DP[i+1, t - visit_time[i],\ m - ticket_cost[i]] + knowledge[i] & \text{:= 0 falls } t - visit_time[i] < 0 \text{ oder } m - ticket_cost[i] < 0 \quad \text{→ besuchen} \end{cases}$$

Berechnungsreihenfolge:

```
for i = N-1 ... 0:
    for t = 0 .. T:
        for m = 0 ... M:
            berechne DP[i,t,m]
```

Lösung: $DP[0, T, M]$ → wir starten bei Museum 0 mit T Stunden und M Dollars

Laufzeit: $\Theta(N\,T\,M)$

Bottom-up:

```java
public static int solve(int N, int M, int T, int[] ticketCost, int[] visitTime, int[] knowledge) {
    DP = new int[N + 1][T + 1][M + 1];
    // Base Case: automatisch gemacht weil alle Einträge auf 0 initialisiert werden

    for (int i = N - 1; i >= 0; i--) {
        for (int t = 0; t <= T; t++) {
            for (int m = 0; m <= M; m++) {
                // Rekursion:
                int knowledgeFromSkip = 0;
                if (t >= 1) knowledgeFromSkip = DP[i + 1][t - 1][m];
                int knowledgeFromVisit = 0;
                if (t >= visitTime[i] && m >= ticketCost[i]) {
                    knowledgeFromVisit = DP[i + 1][t - visitTime[i]][m - ticketCost[i]] + knowledge[i];
                }
                DP[i][t][m] = Math.max(knowledgeFromSkip, knowledgeFromVisit);
            }
        }
    }
    // Lösung:
    return DP[0][T][M];
}
```

Top-down:

```java
public static int solve2(int N, int M, int T, int[] ticketCost, int[] visitTime, int[] knowledge) {
    DP = new int[N + 1][T + 1][M + 1];
    for (int i = 0; i<DP.length; i++) {
        for (int j = 0; j<DP[0].length; j++) {
            for (int k = 0; k<DP[0][0].length; k++) DP[i][j][k] = -1;
        }
    }
    // Lösung:
    return compute(0, T, M);
}
public static int compute(int i, int t, int m) {
    // Base Case:
    if (i >= N) return 0;

    // Memoization:
    if (DP[i][t][m] != -1) return DP[i][t][m];

    // Rekursion:
    int knowledgeFromSkip = 0;
    if (t >= 1) knowledgeFromSkip = compute(i + 1, t - 1, m);

    int knowledgeFromVisit = 0;
    if (t >= visitTime[i] && m >= ticketCost[i]) { // check if we have enough time and money
        knowledgeFromVisit = knowledge[i] + compute(i + 1, t - visitTime[i], m - ticketCost[i]);
    }
    // Store result in DP table:
    DP[i][t][m] = Math.max(knowledgeFromSkip, knowledgeFromVisit);
    return DP[i][t][m];
}
```

Given a binary string $S \in \{0,1\}^n$ of length n, let $g(S)$ be the **positional sum** of the string. We define the positional sum as the sum of all 1-based indices i such that the i-th character of S is a '1'.

For example, if $S = \text{"}01011\text{"}$:

- The '1's are at indices 2, 4, and 5.
- $g(S) = 2 + 4 + 5 = 11$.
- For $S = \text{"}1000\text{"}$, $g(S) = 1$.
- For $S = \text{"}0000\text{"}$, $g(S) = 0$.

Given n and **K**, the goal is to count the number of binary strings S of length n with $g(S) =$ **K**

Describe a DP algorithm that, given positive integers n and **K**, reports the required number. Your solution should have a time complexity of at most $O(n\mathbf{K})$.

Dimension: DP[0...n][0.. K]

Teilproblem: DP[i, k] := number of binary Strings of length i with positional sum = k

Base Case: DP[0,0] = 1 → the empty string

DP[0,k] = 0 for $1 \leq k \leq K$

Rekursion: DP[i, k] = DP[i-1, k] + DP[i-1, k-i] for $1 \leq i \leq n, 0 \leq k \leq K$

DP[i-1, k-i] := 0 falls k-i < 0

DP[i-1, k]: wir hängen eine 0 an

DP[i-1, k-i]: wir hängen eine 1 an

Berechnungsreihenfolge:
for i = 1 .. n
  for k = 0 .. K:
    berechne DP[i,k]

Lösung: DP[n,K]

Laufzeit: $\Theta(nK)$