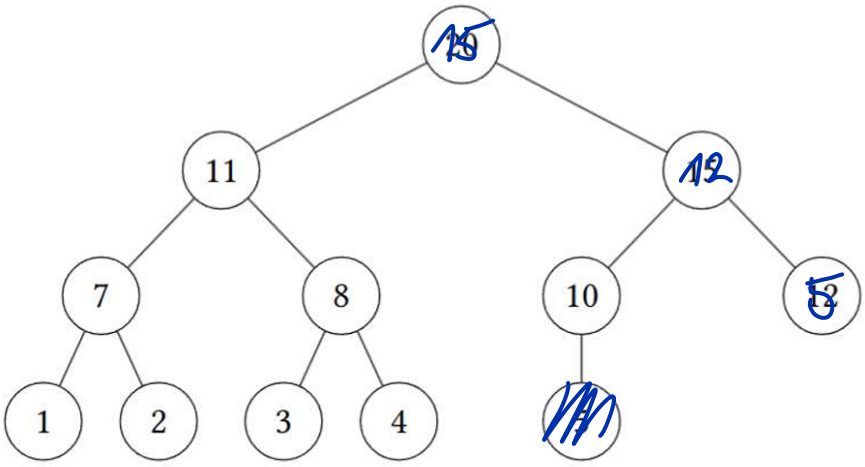
Quiz

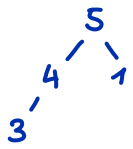
Consider the following heap.



Suppose we extract the number 20. What number will occupy the position currently held by 15, after the insertion?

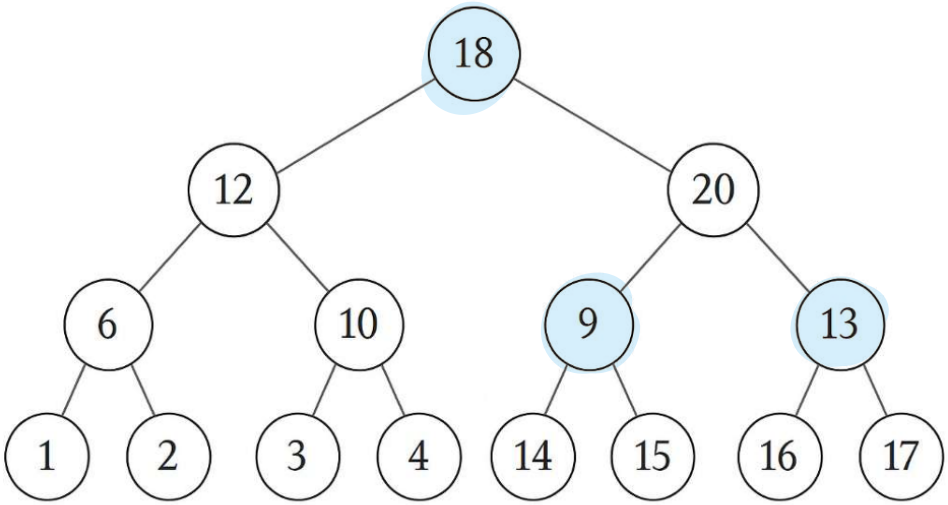
Answer: 12

In a max-heap, every key at depth i is larger than every key at depth $i+1$.



- True
- False

Consider the following tree.



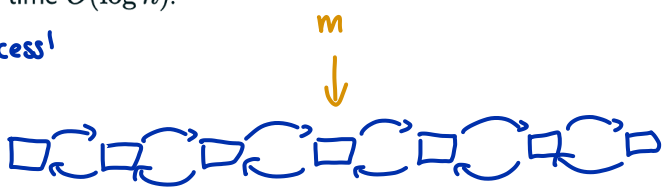
How many vertices violate the max-heap property?

Answer: 3

Binary Search in a doubly linked list takes time $O(\log n)$.

kein random access!

- True
- ✗ False



Consider the operation get(i), which returns the i-th item in a list. A 2-3-tree is a more efficient data structure for this operation than an array.

O(log n)

O(1)

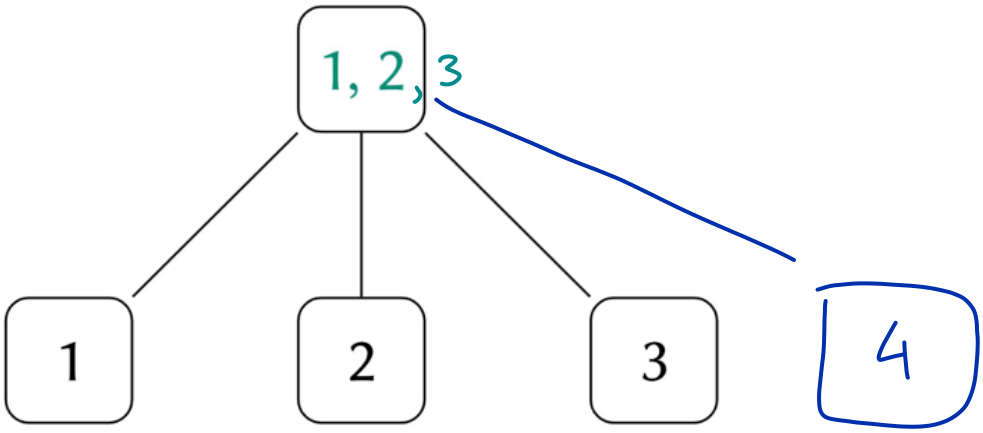
- True
- ✗ False

A 2-3 tree with height h has $\Theta(2^h)$ leaves.

- True
- ✗ False

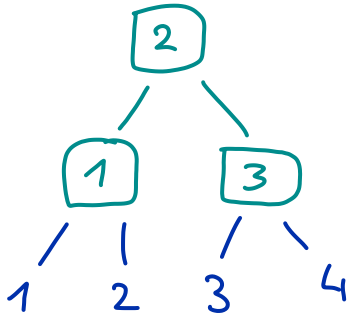
if all nodes have 3 children, it has $\Theta(3^h)$ leaves

Consider the following 2-3 tree:



What is the element in the root of the tree after we insert the number 4?

- a. 1,2
- b. 1, 2, 3
- c. 1, 3
- ✗ d. 2
- e. 1
- f. 4
- g. 3, 4



Consider the pseudocode below:

```
1: function F(n)
2:     if n ≤ 2 then
3:         return 1
4:     else
5:         return F(n-1) + F(n-2)
6:     end if
7: end function
```

What is the asymptotic runtime of this function?

- a. $\Theta(\log n)$
- b. $\Theta(n)$
- c. $\Theta(n^2)$
- ✗ d. $\Theta(C^n)$, for some constant $C > 0$. from lecture

There exists an algorithm that can compute the edit distance between two strings $A[1 \ldots n]$ and $B[1 \ldots m]$ in time $O(mn)$ using extra space $O(n)$.

- ✗ True from lecture
- False

For two strings $A[1 \ldots n]$ and $B[1 \ldots n]$, let L be a longest common subsequence. Then an optimal sequence of edit operations to transform A into B (i.e., of minimal length) will not change any element of L.

Counterexample A = 01
B = 10

- True
- ✗ False

ADT Liste

	Array	einf. verlinkte Liste	dopp. verlinkte Liste
`insert`(k, L)	$O(1)$	$O(1)$	$O(1)$
`get`(i, L)	$O(1)$	$O(\ell)$	$O(\ell)$
`insertAfter`(k, k', L)	$O(\ell)$	$O(1)$	$O(1)$
`delete`(k, L)	$O(\ell)$	$O(\ell)$	$O(1)$

Array: 

einf verlinkte Liste: 

doppelt verlinkte Liste: 

→mehr in EProg

ADT Wörterbuch

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.

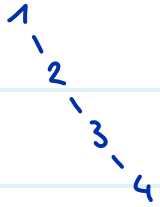
3) `delete`(x, W): Finde und lösche den Schlüssel x aus W, falls er darin vorkommt.

	`search`	`insert`	`delete`
unsortiertes Array	$O(n)$	$O(1)$	$O(n)$
sortiertes Array	$O(\log n)$	$O(n)$	$O(n)$
Doppelt verk. Liste	$O(n)$	$O(1)$	$O(n)$

Binärer Suchbaum

- Suchbaum Bedingung: alle Schlüssel im linken Teilbaum < Schlüssel des Knotens < alle Schlüssel im rechten Teilbaum

- search $\leq O(h)$
- insert $\leq O(h)$
- delete $\leq O(h)$
- Optimal falls $h \leq O(\log n)$
- Schlecht falls $h = \Theta(n)$



Lösung: 2-3-Bäume (Achtung unterschiedliche Namen in anderen Quellen)

2-3-Baum

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")
- Alle Blätter befinden sich auf derselben Ebene (Tiefe). perfekt balanciert, $h \leq O(\log n)$
- 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 $> s_2$.

"bei Gleichheit nach links"

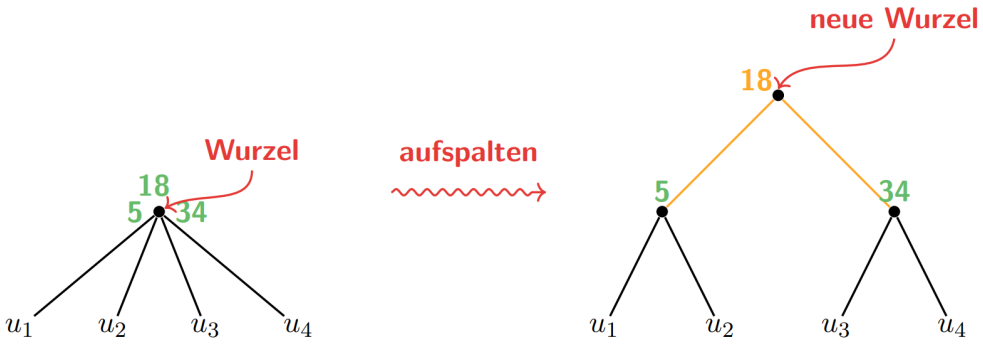
- search $\leq O(\log n)$
- insert(x): 1) suche richtigen Platz für x

  2) erstelle neues Blatt mit Schlüssel x und einen passenden Separator beim Elternknoten hinzu

  3) Falls Elternknoten nun 3 Separatoren hat, teile ihn in zwei Knoten

auf mit dem kleinsten und grössten Separator, und gib den mittleren Separator eine Ebene nach oben.

Wiederhole rekursiv falls 2-3-Bedingung verletzt



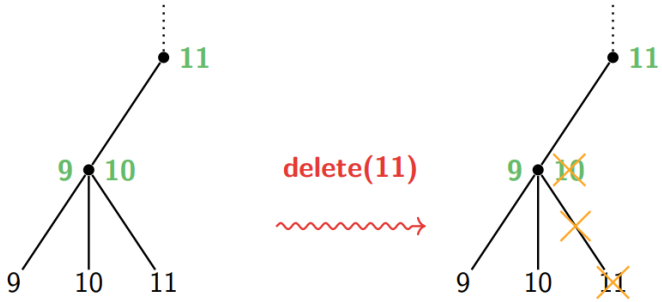
- delete(x): 1) suche x

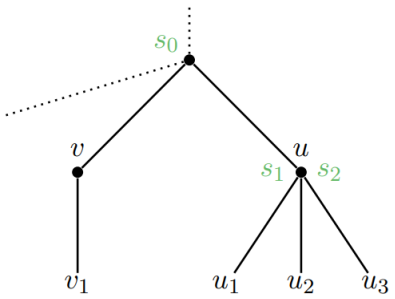
  2) lösche x und den entsprechenden Separator

  3) Falls 2-3-Bedingung verletzt:

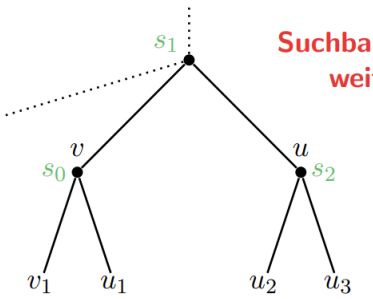
     i) Falls ein Geschwisterknoten 3 Kinder hat, dann adoptieren wir eines und aktualisieren die Separatoren entsprechend

     ii) Sonst verschmelze den Knoten mit einem Geschwisterknoten (der 2 Kinder hat). Falls dadurch ein Level weiter oben ein Problem entsteht, dann rebalanciere rekursiv.

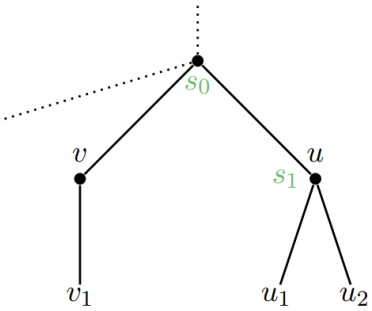




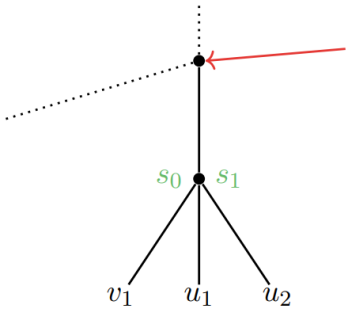
Adoption



Suchbaumbedingung weiter erfüllt



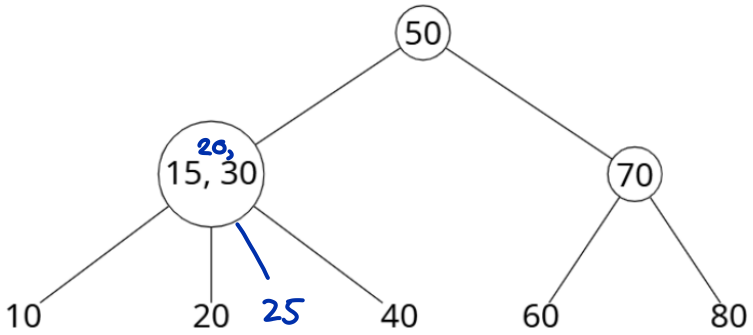
Verschmelzen

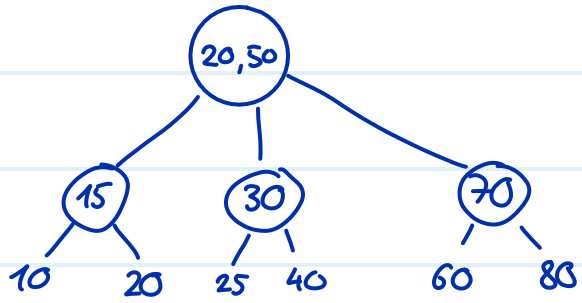


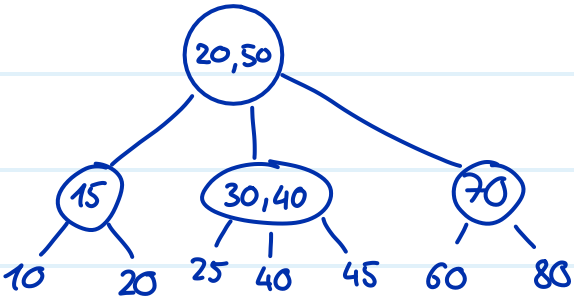
verliert ein Kind

cool: funktioniert gleichzeitig als Max/Min Heap

Aufgabe: insert(25), insert(45)







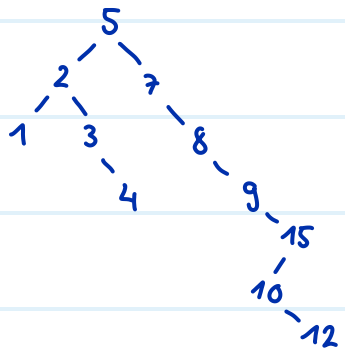
Algorithm 1

```
function g(n)
    i ← 1
    while i < n do
        f()
        i ← i + 2
    g(n/2)
    g(n/2)
    g(n/2)
```

Base Case bei Rekursion nicht vergessen!

- BubbleSort

Aufgabe: Füge die Schlüssel 5, 2, 7, 8, 1, 3, 4, 9, 15, 10, 12 in einen leeren binären Suchbaum.



Dynamische Programmierung

Herangehensweise:

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

Zum Programmieren:

Zwei Varianten: Rekursiv (Top-Down, Memoization) vs. Iterativ (Bottom up)

MSS

[2, 3, -1, 4, 0, 1]

Dimension: $R[1 \dots n]$

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

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

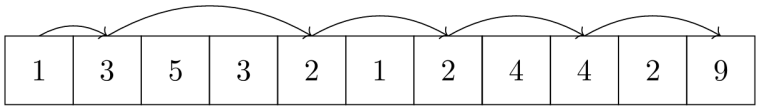
Rekursion: $R[i] = \max\{R[i-1] + A[i], A[i]\}$ für $2 \leq i \leq n$

Berechnungsreihenfolge: For $i = 1 \dots n$ · compute $R[i]$

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

Laufzeit: $O(n)$

Jump Game



Dimension: $M[0..n]$

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

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

Rekursion: $M[k] = \max\{ i + A[i] \mid M[k-2] < i \leq M[k-1] \}$ für $2 \leq k \leq n$

Berechnungsreihenfolge: For $k = 1 \ldots n$ · compute $M[k]$

Lösung: $\min\{k \mid M[k] \geq n\}$

Laufzeit: $O(n)$

Längste Gemeinsame Teilfolge

Dimension: L ist ein Array der Dimension $(n+1) \times (m+1)$

Teilproblem: $L[i,j]$:= längste gemeinsame Teilfolge von $(a_1, a_2, \ldots, a_i)$ und $(b_1, b_2 \ldots b_j)$

Base Case: $L[0,j] = 0$ für $0 \leq j \leq m$

$L[i,0] = 0$ für $0 \leq i \leq n$

Rekursion:

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

für $1 \leq i \leq n$, $1 \leq j \leq m$

entweder a_i oder b_j ist nicht das letzte Element der LGT

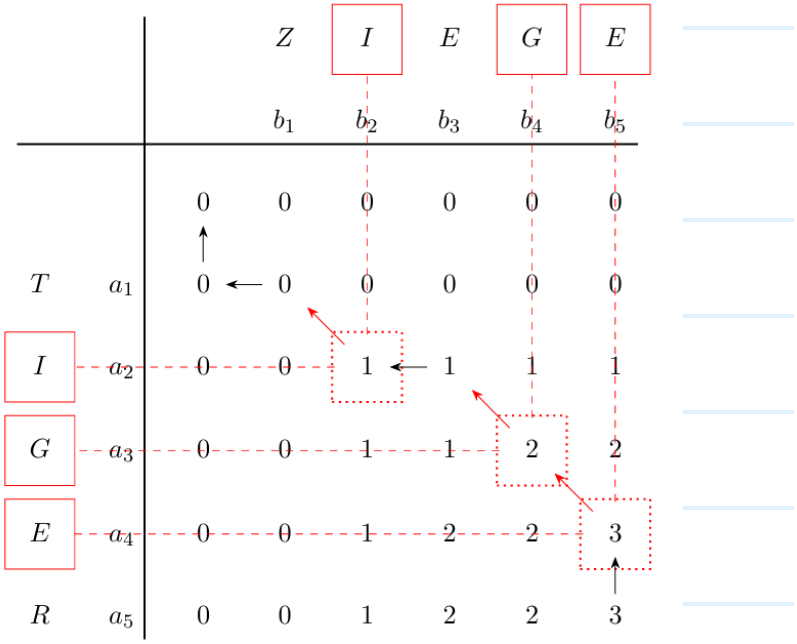
Berechnungsreihenfolge:

For $i = 0 .. n$:
  For $j = 0 .. m$:
    compute $L[i,j]$

Lösung: $L[n,m]$

Laufzeit: $O(n \cdot m)$

			Z	I	E	G	E
			b_1	b_2	b_3	b_4	b_5
		0	0	0	0	0	0
T	a_1	0	0	0	0	0	0
I	a_2	0	0	1	1	1	1
G	a_3	0	0	1	1	2	2
E	a_4	0	0	1	2	2	3
R	a_5	0	0	1	2	2	3



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 ist ein Array der Dimension $(n+1) \times (m+1)$

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

Base Case: $ED[0,j] = j$ für $0 \leq j \leq m$

$ED[i,0] = i$ für $0 \leq i \leq n$

wir können das letzte Zeichen ignorieren

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 machen eine Operation

$ED[i-1,j]$: löschen; $ED[i,j-1]$: hinzufügen; $ED[i-1,j-1]$: ersetzen

$$(a_1, \ldots, \boxed{a_i}) \rightarrow (\overset{?}{\boxed{b_1, \ldots, b_j}})$$

Auch falls es überschrieben wird, verfolgen wir das Zeichen trotzdem weiter. Da sich am Ende des Editiervorgangs das Wort $B[1..j]$ ergibt, welches Länge j hat, muss einer der folgenden drei Fälle auftreten:

1. a_i wird irgendwann gelöscht.

   $$(a_1, \ldots, \boxed{a_i}) \rightarrow (b_1, \ldots, b_j)$$ löschen

2. a_i wird nicht gelöscht und steht am Ende an einer Stelle aus $\{1, \ldots, j-1\}$.

   $$(a_1, \ldots, \boxed{a_i}) \rightarrow (\boxed{b_1, \ldots, b_{j-1}}, b_j)$$ hinzufügen

3. a_i wird nicht gelöscht und steht am Ende an Stelle j.

   $$(a_1, \ldots, \boxed{a_i}) \rightarrow (b_1, \ldots, \boxed{b_j})$$ ersetzen falls $a_i \neq b_j$

   so lassen falls $a_i = b_j$

Berechnungsreihenfolge:
For i = 0 .. n :
  For j = 0 .. m :
    compute ED[i,j]

Lösung: ED[n,m]

Laufzeit: $O(n \cdot m)$

			Z	I	E	G	E
			b_1	b_2	b_3	b_4	b_5
		0	1	2	3	4	5
T	a_1	1	1	2	3	4	5
I	a_2	2	2	1	2	3	4
G	a_3	3	3	2	2	2	3
E	a_4	4	4	3	2	3	2
R	a_5	5	5	4	3	3	3

Aufgabenstellung: Du bist ein professioneller Einbrecher und planst, Häuser in einer Straße auszurauben. Jedes Haus i hat einen bestimmten Geldbetrag $nums[i]$. Es gibt eine Sicherheitsregel: Du kannst **nicht zwei direkt benachbarte Häuser** in derselben Nacht ausrauben, sonst geht der Alarm los.

Was ist der maximale Geldbetrag, den du stehlen kannst, ohne den Alarm auszulösen?

Beispiel: $nums = [2, 7, 9, 3, 1]$ nums[1 .. n]

- Option 1: Haus 0 + Haus 2 + Haus 4 $\rightarrow 2 + 9 + 1 = 12$
- Option 2: Haus 1 + Haus 3 $\rightarrow 7 + 3 = 10$
- ... $\rightarrow$ Der maximale Betrag ist 12.

Dimension: DP[0 ... n]

Teilproblem: DP[i] = maximaler Betrag wenn wir nur die ersten i Häuser betrachten

Base Case: DP[0] = 0, DP[1] = nums[1]

Rekursion: DP[i] = max { DP[i-2] + nums[i] , DP[i-1] } für $2 \leq i \leq n$

einbrechen — weiter gehen

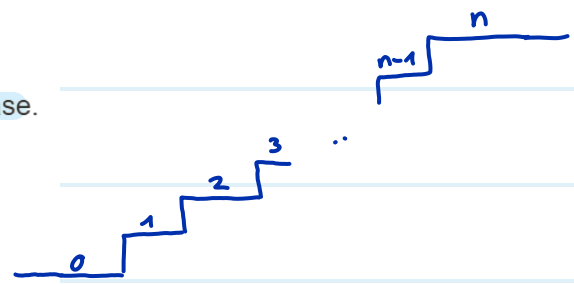
Berechnungsreihenfolge: For i = 0 ... n : compute DP[i]

Lösung: DP[n]

Laufzeit: O(n)

Description

cost[0 ... n-1]



You are given an integer array cost where cost[i] is the cost of ith step on a staircase. 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.

Examples

Example 1:

Input: cost = [10,15,20]
Output: 15

Explanation: You will start at index 1.

- Pay 15 and climb two steps to reach the top.
  The total cost is 15.

Example 2:

Input: cost = [1,100,1,1,1,100,1,1,100,1]
Output: 6

Explanation: You will start at index 0.

- Pay 1 and climb two steps to reach index 2.
- Pay 1 and climb two steps to reach index 4.
- Pay 1 and climb two steps to reach index 6.
- Pay 1 and climb one step to reach index 7.
- Pay 1 and climb two steps to reach index 9.
- Pay 1 and climb one step to reach the top.
  The total cost is 6.

Constraints:

```
2 <= cost.length <= 1000
0 <= cost[i] <= 999
```

Dimension: DP[0...n]

Teilproblem: Minimum cost to reach step i

Base Case: DP[0] = 0, DP[1] = 0

Rekursion: DP[i] = min { DP[i-1] + cost[i-1], DP[i-2] + cost[i-2] } für $2 \leq i \leq n$

Berechnungsreihenfolge: For i = 0...n: compute DP[i]

Lösung: DP[n]

Laufzeit: O(n)

Iterativ:

```java
public static int minCostClimbingStairs(int[] cost) {
  // TODO:
  int n = cost.length;
  int[] DP = new int[n+1];

  // Base Cases:
  DP[0] = 0;
  DP[1] = 0;

  // Rekursion:
  for (int i = 2; i<=n; i++) {
    DP[i] = Math.min(DP[i-2]+cost[i-2], DP[i-1]+cost[i-1]);
  }

  // Lösung:
  return DP[n];
}
```

Rekursiv:

```java
public static int minCostClimbingStairs(int[] cost) {
  // DP initialisieren:
  int n = cost.length;
  DP = new int[n+1];
  for (int i = 0; i<DP.length; i++) DP[i] = -1;

  // Lösung:
  return compute(n);
}

public static int compute(int i) {
  // Base case:
  if (i == 0 || i == 1) return 0;

  // Memoization: (check if already computed)
  if (DP[i] != -1) return DP[i];

  // Rekursion:
  DP[i] = Math.min(compute(i-2)+cost[i-2], compute(i-1)+cost[i-1]);
  return DP[i];
}
```