ETH zürich

Übungslektion 10 – Dynamische Programmierung I

Informatik II

Willkommen!

Polybox

Passwort: jschul

Personal Website

`https://jschultev.github.io/personal_website/`

Heutiges Programm

Rückblick

Königsweg

Stab schneiden

Zusammenfassung

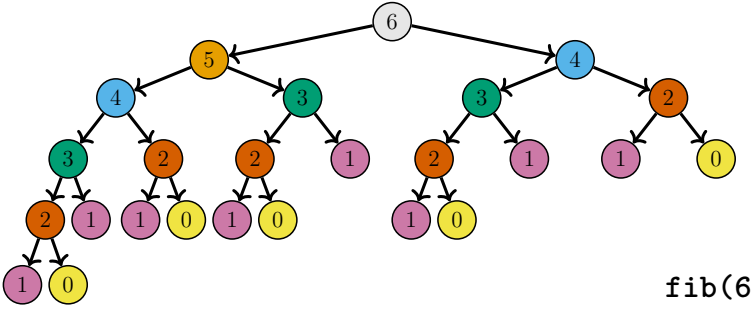
Prüfungsaufgaben

Hausaufgaben

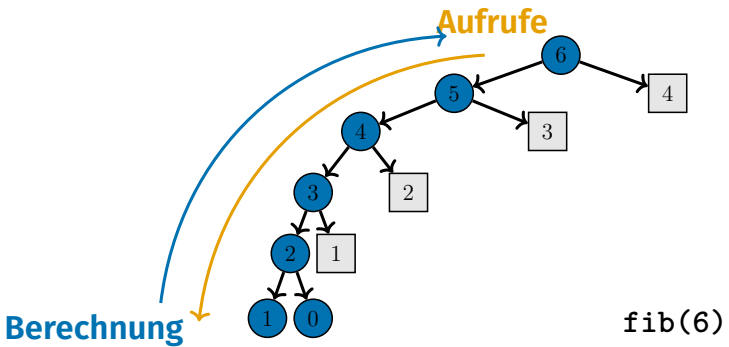
1. Rückblick

Wiederholung der Vorlesung

Probleme, die rekursiv gelöst werden können, wie Fibonacci, können wir mit Memoisierung (top-down) oder Dynamischer Programmierung (bottom-up) optimieren, indem wir redundante Berechnungen vermeiden.

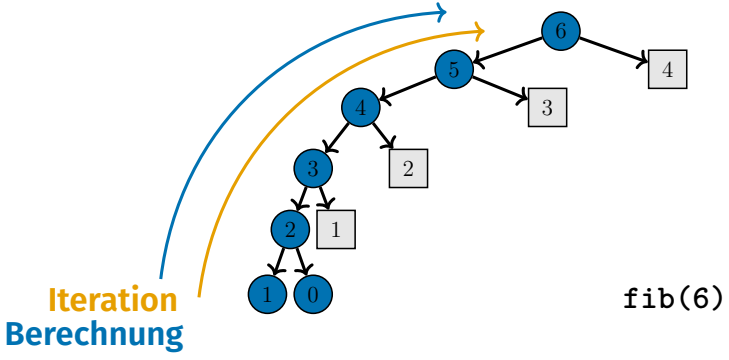


Memoisierung…



Memoisierung funktioniert, indem Lösungen zu Teilproblemen in einer Tabelle gespeichert werden. Diese Tabelle wird mit jedem **rekursiven Aufruf** *weitergegeben*, um redundante Berechnungen zu vermeiden, indem zuvor berechnete Ergebnisse (quadratische Knoten) wiederverwendet statt neu berechnet werden.

…vs. Dynamische Programmierung: Tabellierung



Bottom-up dynamische Programmierung löst Probleme **iterativ** vom kleinsten Teilproblem aus aufwärts. Sie verwendet eine Tabelle, um die Ergebnisse zu speichern und redundante Berechnungen zu vermeiden.

Memoisierung != Dynamische Programmierung

Memoisierung

- **Ansatz?** Löst nur die benötigten Teilprobleme.
- **Rekursion?** Mit Rekursion implementiert, Tabelle wird mit jedem Aufruf weitergegeben.
- **Aufrufrichtung?** Top-down.

Dynamische Programmierung

Memoisierung != Dynamische Programmierung

Memoisierung

- **Ansatz?** Löst nur die benötigten Teilprobleme.
- **Rekursion?** Mit Rekursion implementiert, Tabelle wird mit jedem Aufruf weitergegeben.
- **Aufrufrichtung?** Top-down.

Dynamische Programmierung

- **Ansatz?** Löst alle Teilprobleme im Voraus, auch ungenutzte.
- **Rekursion?** Typischerweise iterativ implementiert.
- **"Aufruf"-Richtung?** Bottom-Up.

Memoisierung != Dynamische Programmierung

Memoisierung

- **Ansatz?** Löst nur die benötigten Teilprobleme.
- **Rekursion?** Mit Rekursion implementiert, Tabelle wird mit jedem Aufruf weitergegeben.
- **Aufrufrichtung?** Top-down.

Dynamische Programmierung

- **Ansatz?** Löst alle Teilprobleme im Voraus, auch ungenutzte.
- **Rekursion?** Typischerweise iterativ implementiert.
- **"Aufruf"-Richtung?** Bottom-Up.

Beide Taktiken lösen dasselbe Problem, aber auf unterschiedliche Weise. Eine Bottom-Up-Lösung ist eleganter und füllt den Call Stack nicht, während Memoisierung schnell implementiert ist, ohne das Problem im Detail zu betrachten.

2. Königsweg

Aufgabenbeschreibung: Königsweg

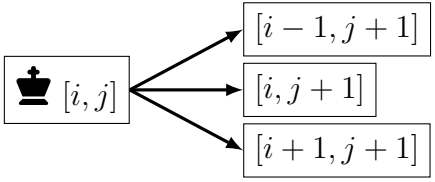
- Gegeben: Ein Schachbrett a mit dem König startend auf Zelle $[0, 0]$, wobei jede Zelle eine nicht-negative Belohnung (0 oder mehr) enthält.

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

Aufgabenbeschreibung: Königsweg

- Gegeben: Ein Schachbrett a mit dem König startend auf Zelle $[0, 0]$, wobei jede Zelle eine nicht-negative Belohnung (0 oder mehr) enthält.
- Auf jeder Zelle hat der König drei Optionen, um sich von Position $[i, j]$ zu bewegen, solange er das Schachbrett nicht verlässt.

♚	1	3	0	1
0	1	0	1	1
0	1	0	0	0

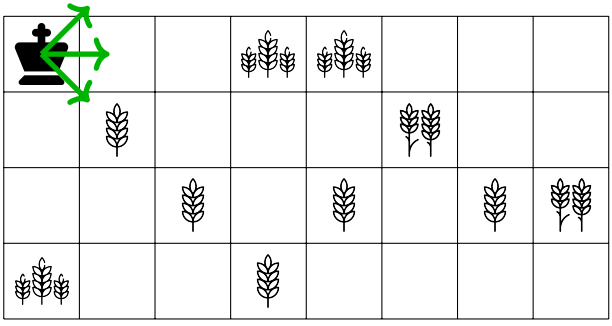


Aufgabenbeschreibung: Königsweg

- **Eingabe**: Ein Schachbrett a als 2D-Array der Grösse $n \times m$, wobei jede Zelle eine nicht-negative ganze Zahl enthält, welche die entsprechende Belohnung angibt.
- **Ziel**: Der König startet auf Zelle $[0, 0]$ und kann sich in drei Richtungen bewegen: nord-östlich, östlich und süd-östlich. Das Ziel ist es, den optimalen Pfad zu finden, um die Gesamtbelohnung zu maximieren, während der König sich zur rechten Seite bewegt.
- **Ausgabe**: Die maximale Belohnung, die der König auf seinem Weg zur rechten Seite des Schachbretts einsammeln kann.

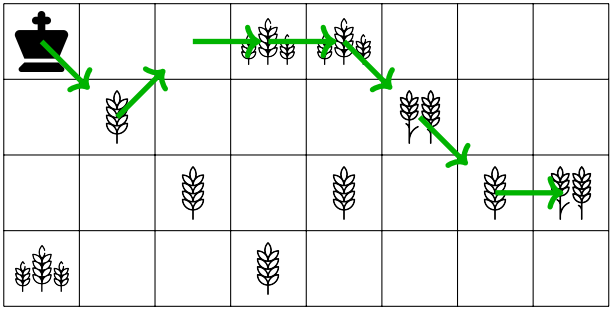
Aufgabenbeschreibung: Königsweg Beispiel

Was ist die maximale Belohnung, die der König einsammeln kann?

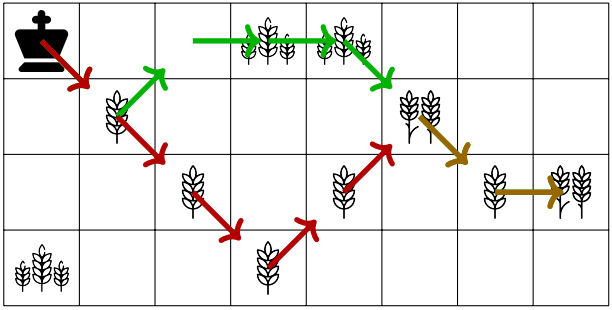


Aufgabenbeschreibung: Königsweg Beispiel

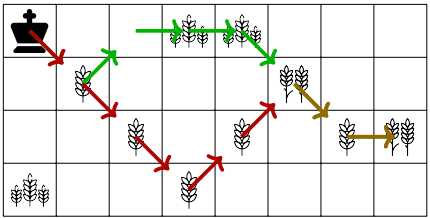
Was ist die maximale Belohnung, die der König einsammeln kann? **12 Weizen**



Greedy Lösung

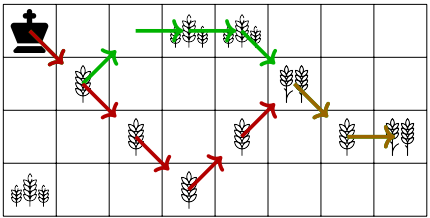


Greedy Lösung



- Nur die sofortige Belohnung zu nehmen, kann zu suboptimalen Entscheidungen führen, welche grössere Belohnungen weiter vorne verpassen.

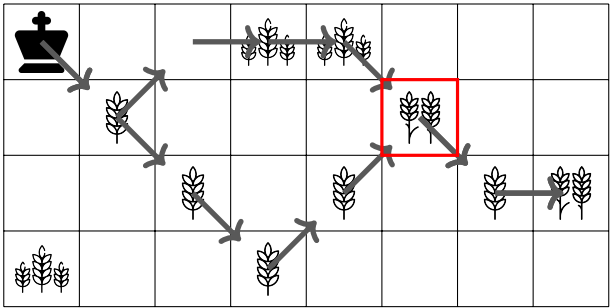
Greedy Lösung



- Nur die sofortige Belohnung zu nehmen, kann zu suboptimalen Entscheidungen führen, welche grössere Belohnungen weiter vorne verpassen.
- Daher müssen wir jeden möglichen Pfad evaluieren, um den global optimalen Pfad zu finden.

Überlappende Teilprobleme

Da mehrere Pfade zur gleichen Zelle führen können, wird das Teilproblem mehrfach gelöst, was zu redundanten Berechnungen führt. Im Beispiel gibt es zwei Pfade, die an der markierten Zelle zusammenlaufen.



Warum ist DP nützlich?

- **Brute Force**: Das Vergleichen aller Pfade hat eine exponentielle Zeitkomplexität von $\mathcal{O}(3^{n \times m})$, was ineffizient ist.

Warum ist DP nützlich?

- **Brute Force**: Das Vergleichen aller Pfade hat eine exponentielle Zeitkomplexität von $\mathcal{O}(3^{n \times m})$, was ineffizient ist.
- **Dynamische Programmierung** zerlegt das Problem in kleinere Teilprobleme und speichert Lösungen zur Wiederverwendung.

Warum ist DP nützlich?

- **Brute Force**: Das Vergleichen aller Pfade hat eine exponentielle Zeitkomplexität von $\mathcal{O}(3^{n \times m})$, was ineffizient ist.
- **Dynamische Programmierung** zerlegt das Problem in kleinere Teilprobleme und speichert Lösungen zur Wiederverwendung.
- **Zeitkomplexität**: $\mathcal{O}(n \times m)$, da wir eine Berechnung von $\mathcal{O}(1)$ für jede Zelle im $n \times m$ Gitter durchführen.

Warum ist DP nützlich?

- **Brute Force**: Das Vergleichen aller Pfade hat eine exponentielle Zeitkomplexität von $\mathcal{O}(3^{n \times m})$, was ineffizient ist.
- **Dynamische Programmierung** zerlegt das Problem in kleinere Teilprobleme und speichert Lösungen zur Wiederverwendung.
- **Zeitkomplexität**: $\mathcal{O}(n \times m)$, da wir eine Berechnung von $\mathcal{O}(1)$ für jede Zelle im $n \times m$ Gitter durchführen.
- **Effizienz**: DP vermeidet redundante Berechnungen, wodurch es signifikant effizienter wird.

Die drei Schritte der DP

1. **Identifiziere die Teilprobleme** und analysiere, wie sie sich überlappen.
   Hier: Mehrere Pfade, die auf derselben Zelle zusammenlaufen.

Die drei Schritte der DP

1. **Identifiziere die Teilprobleme** und analysiere, wie sie sich überlappen.
   Hier: Mehrere Pfade, die auf derselben Zelle zusammenlaufen.
2. **Definiere die rekursive Formel** durch Betrachtung möglicher Übergänge zwischen Teilproblemen und optimaler Entscheidungen.
   Hier: Bewegungen des Königs.

Die drei Schritte der DP

1. **Identifiziere die Teilprobleme** und analysiere, wie sie sich überlappen. *Hier: Mehrere Pfade, die auf derselben Zelle zusammenlaufen.*
2. **Definiere die rekursive Formel** durch Betrachtung möglicher Übergänge zwischen Teilproblemen und optimaler Entscheidungen. *Hier: Bewegungen des Königs.*
3. **Implementiere eine Lösung** mit einer geeigneten Datenstruktur und der korrekten Berechnungsreihenfolge. *Die Struktur ist fast immer eine Art Array oder Matrix.*

Schritt 1: Identifiziere das Teilproblem

- Was ist das Teilproblem?

Für jede Zelle $[i, j]$ auf dem Schachbrett ($i < \mathtt{n}$, $j < \mathtt{m}$) repräsentiert $S[i, j]$ die maximale Belohnung, die der König einsammeln kann, indem er sich von dieser Zelle zur letzten Spalte $j = \mathtt{m} - 1$ bewegt und dabei alle möglichen Züge berücksichtigt.

Schritt 1: Identifiziere das Teilproblem

- Was ist das Teilproblem?

Für jede Zelle $[i, j]$ auf dem Schachbrett ($i < \mathtt{n}$, $j < \mathtt{m}$) repräsentiert $S[i, j]$ die maximale Belohnung, die der König einsammeln kann, indem er sich von dieser Zelle zur letzten Spalte $j = \mathtt{m} - 1$ bewegt und dabei alle möglichen Züge berücksichtigt.

- Welche Optionen hat der König an Position $[i, j]$?
  - Bewege nach Nord-Ost $[i - 1, j + 1]$, falls $i > 0$
  - Bewege nach Ost $[i, j + 1]$
  - Bewege nach Süd-Ost $[i + 1, j + 1]$, falls $i < n - 1$

Randbedingungen

Kann der König sich immer nach Nord-Ost oder Süd-Ost bewegen?

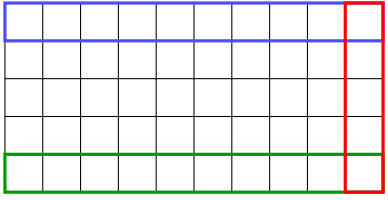
Nein, Randbedingungen müssen berücksichtigt werden!

- Oberste Zeile $(i = 0)$: Der König kann sich nicht nach Nord-Ost bewegen.
- Unterste Zeile $(i = n - 1)$: Der König kann sich nicht nach Süd-Ost bewegen.

Basisfall

Sind das die einzigen Einschränkungen, denen der König auf dem Brett begegnet?

Nein, eine weitere zentrale Einschränkung ist, dass der König anhalten muss, wenn er die äusserste rechte Spalte $(j = m - 1)$ erreicht, da darüber hinaus keine weiteren Züge möglich sind. Dies ist der **Basisfall**: Der König kann sich nicht weiterbewegen, also gilt $S[i, j] = \texttt{a}[\texttt{i}, \texttt{j}]$.



Schritt 1: Identifiziere die Teilprobleme

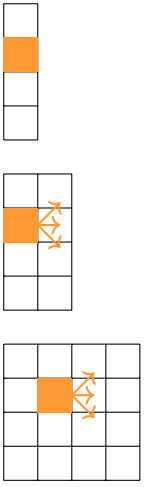
- Eine Spalte $\rightarrow$ Der König erhält die Belohnung der Zelle (Basisfall).

Schritt 1: Identifiziere die Teilprobleme

- Eine Spalte → Der König erhält die Belohnung der Zelle (Basisfall).
- Zwei Spalten → Der König hat drei mögliche Züge: nord-östlich, östlich und süd-östlich. Er wählt den Zug mit der maximalen Belohnung.

Schritt 1: Identifiziere die Teilprobleme

- Eine Spalte → Der König erhält die Belohnung der Zelle (Basisfall).
- Zwei Spalten → Der König hat drei mögliche Züge: nord-östlich, östlich und süd-östlich. Er wählt den Zug mit der maximalen Belohnung.
- Ganzes Schachbrett → Der König hat die gleichen drei Optionen. Aber um zu wissen, welche die beste ist, muss er zuerst die maximale Belohnung für alle drei Teilprobleme evaluieren.



Schritt 2: Definiere die rekursive Formel

$S[i,j]$ repräsentiert die maximale Belohnung, die der König einsammeln kann, wenn er auf Zelle $[i,j]$ startet und optimal zur äussersten rechten Spalte läuft.

$$S[i,j] = \begin{cases} a[i,j] & j = m-1 \\ a[i,j] + \max(S[i-1,j+1],\, S[i,j+1]) & i = n-1,\ j < m-1 \\ a[i,j] + \max(S[i,j+1],\, S[i+1,j+1]) & i = 0,\ j < m-1 \\ a[i,j] + \max\begin{cases} S[i-1,j+1] \\ S[i,j+1] \\ S[i+1,j+1] \end{cases} & 0 < i < n-1,\ j < m-1 \end{cases}$$

Schritt 3: Implementierung

- Welche Datenstruktur ist am besten geeignet, um die Teilprobleme zu speichern?

Schritt 3: Implementierung

- Welche Datenstruktur ist am besten geeignet, um die Teilprobleme zu speichern?
  Eine Matrix `s` der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i <$ `n` und $j <$ `m`.

Schritt 3: Implementierung

- Welche Datenstruktur ist am besten geeignet, um die Teilprobleme zu speichern?
  Eine Matrix `s` der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i < \mathtt{n}$ und $j < \mathtt{m}$.
- Um $S[i, j]$ zu berechnen, welche Teilprobleme müssen zuerst gelöst werden?

Schritt 3: Implementierung

- Welche Datenstruktur ist am besten geeignet, um die Teilprobleme zu speichern?
  Eine Matrix `s` der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i <$ `n` und $j <$ `m`.
- Um $S[i, j]$ zu berechnen, welche Teilprobleme müssen zuerst gelöst werden?
  $S[i, j + 1]$, $S[i - 1, j + 1]$ wenn $i > 0$, und $S[i + 1, j + 1]$ wenn $j < m - 1$, wobei die Gittergrenzen respektiert werden.

Schritt 3: Implementierung

- Welche Datenstruktur ist am besten geeignet, um die Teilprobleme zu speichern?
  Eine Matrix `s` der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i <$ `n` und $j <$ `m`.
- Um $S[i, j]$ zu berechnen, welche Teilprobleme müssen zuerst gelöst werden?
  $S[i, j + 1]$, $S[i - 1, j + 1]$ wenn $i > 0$, und $S[i + 1, j + 1]$ wenn $j < m - 1$, wobei die Gittergrenzen respektiert werden.
- In welcher Reihenfolge soll S anschliessend ausgefüllt werden?

Schritt 3: Implementierung

- Welche Datenstruktur ist am besten geeignet, um die Teilprobleme zu speichern?
  Eine Matrix `s` der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i <$ `n` und $j <$ `m`.
- Um $S[i, j]$ zu berechnen, welche Teilprobleme müssen zuerst gelöst werden?
  $S[i, j + 1]$, $S[i - 1, j + 1]$ wenn $i > 0$, und $S[i + 1, j + 1]$ wenn $j < m - 1$, wobei die Gittergrenzen respektiert werden.
- In welcher Reihenfolge soll S anschliessend ausgefüllt werden?
  Von rechts nach links, Spalte für Spalte. Die Reihenfolge, in der eine Spalte ausgefüllt wird, spielt keine Rolle.

DP-Tabelle

Während des DP-Übergangs müssen Randbedingungen beachtet werden:

- **DP-Tabellengrösse**: Wir verwenden eine DP-Tabelle mit derselben Grösse wie das Schachbrett, um die optimale Lösung für jede Zelle zu speichern.

DP-Tabelle

Während des DP-Übergangs müssen Randbedingungen beachtet werden:

- **DP-Tabellengrösse**: Wir verwenden eine DP-Tabelle mit derselben Grösse wie das Schachbrett, um die optimale Lösung für jede Zelle zu speichern.
- **Durchlauf**: Wir müssen die DP-Tabelle so durchlaufen, dass sichergestellt ist, dass alle Teilprobleme gelöst sind, bevor wir das aktuelle Problem berechnen.

DP-Tabelle

Während des DP-Übergangs müssen Randbedingungen beachtet werden:

- **DP-Tabellengrösse**: Wir verwenden eine DP-Tabelle mit derselben Grösse wie das Schachbrett, um die optimale Lösung für jede Zelle zu speichern.
- **Durchlauf**: Wir müssen die DP-Tabelle so durchlaufen, dass sichergestellt ist, dass alle Teilprobleme gelöst sind, bevor wir das aktuelle Problem berechnen.
- **Randbedingungen**:

DP-Tabelle

Während des DP-Übergangs müssen Randbedingungen beachtet werden:

- **DP-Tabellengrösse**: Wir verwenden eine DP-Tabelle mit derselben Grösse wie das Schachbrett, um die optimale Lösung für jede Zelle zu speichern.
- **Durchlauf**: Wir müssen die DP-Tabelle so durchlaufen, dass sichergestellt ist, dass alle Teilprobleme gelöst sind, bevor wir das aktuelle Problem berechnen.
- **Randbedingungen**:
  - Bei $j = m - 1$ (letzte Spalte) kann der König sich nicht weiter nach rechts bewegen. Dies ist der Basisfall, bei dem keine weitere Bewegung möglich ist und die Lösung unkompliziert ist.

DP-Tabelle

Während des DP-Übergangs müssen Randbedingungen beachtet werden:

- **DP-Tabellengrösse**: Wir verwenden eine DP-Tabelle mit derselben Grösse wie das Schachbrett, um die optimale Lösung für jede Zelle zu speichern.
- **Durchlauf**: Wir müssen die DP-Tabelle so durchlaufen, dass sichergestellt ist, dass alle Teilprobleme gelöst sind, bevor wir das aktuelle Problem berechnen.
- **Randbedingungen**:
  - Bei $j = m - 1$ (letzte Spalte) kann der König sich nicht weiter nach rechts bewegen. Dies ist der Basisfall, bei dem keine weitere Bewegung möglich ist und die Lösung unkompliziert ist.
  - Wenn $i = 0$ (oberste Zeile), ist der Zug nord-ost nicht möglich.
  - Wenn $\texttt{i} = n - 1$ (unterste Zeile), ist der Zug süd-ost nicht möglich.

DP-Tabelle

Während des DP-Übergangs müssen Randbedingungen beachtet werden:

- **DP-Tabellengrösse**: Wir verwenden eine DP-Tabelle mit derselben Grösse wie das Schachbrett, um die optimale Lösung für jede Zelle zu speichern.
- **Durchlauf**: Wir müssen die DP-Tabelle so durchlaufen, dass sichergestellt ist, dass alle Teilprobleme gelöst sind, bevor wir das aktuelle Problem berechnen.
- **Randbedingungen**:
  - Bei $j = m - 1$ (letzte Spalte) kann der König sich nicht weiter nach rechts bewegen. Dies ist der Basisfall, bei dem keine weitere Bewegung möglich ist und die Lösung unkompliziert ist.
  - Wenn $i = 0$ (oberste Zeile), ist der Zug nord-ost nicht möglich.
  - Wenn $\texttt{i} = n - 1$ (unterste Zeile), ist der Zug süd-ost nicht möglich.
  - An Randpositionen sollten nur die zulässigen Züge berücksichtigt werden.

Datenstruktur

Durch die rekursive Formulierung haben wir die Abhängigkeiten für die Berechnung der Lösung des Königswegs an Position $[i, j]$ definiert.

$$S[i,j] = \begin{cases} a[i,j] & j = m-1 \\ a[i,j] + \max(S[i-1, j+1],\ S[i, j+1]) & i = n-1,\ j < m-1 \\ a[i,j] + \max(S[i, j+1],\ S[i+1, j+1]) & i = 0,\ j < m-1 \\ a[i,j] + \max \begin{cases} S[i-1, j+1] \\ S[i, j+1] \\ S[i+1, j+1] \end{cases} & 0 < i < n-1,\ j < m-1 \end{cases}$$

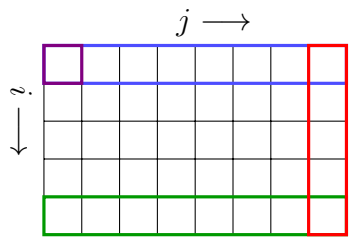
Die rekursive Lösung in der Tabelle

- Sei `S[i,j]` die maximale Belohnung, wenn der König beginnt, sich von dieser Zelle aus nach rechts zu bewegen.

  $\rightarrow$ Nachdem die ganze Tabelle berechnet wurde, könnten wir den König auf ein beliebiges Feld setzen und wüssten, welche maximale Belohnung von dieser Startzelle aus erzielt werden kann.

DP-Tabelle

Die endgültige Lösung ist durch $S[0,0]$ gegeben, was die obere linke Ecke der DP-Tabelle ist.

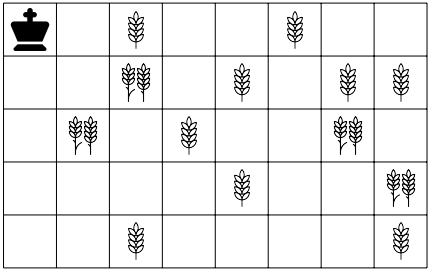


Randbedingungen:

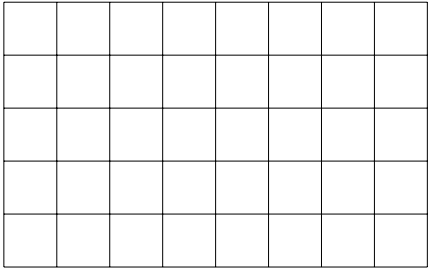
- $j = m - 1$:
  $\mathtt{a[i,j]}$
- $i = 0,\ j < m - 1$:
  $\mathtt{a[i,j]} + \max(\rightarrow, \searrow)$
- $i = n - 1,\ j < m - 1$:
  $\mathtt{a[i,j]} + \max(\rightarrow, \nearrow)$
- sonst:
  $\mathtt{a[i,j]} + \max(\nearrow, \rightarrow, \searrow)$

Beispiel: Wie man eine Tabelle füllt

$a =$



$S =$

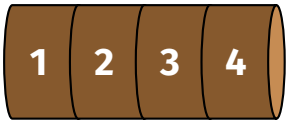


Python-Implementierung: Maximale Belohnung

```
import numpy as np

def bestway(a):
    a = np.array(a) # If a is not yet a numpy array
    n,m = a.shape
    S = np.zeros((n,m))

    for j in range(m-1, -1, -1):
        for i in range(n):
            if j == m-1:
                S[i,j] = a[i,j]
            else:
                northeast = S[i-1,j+1] if i > 0 else -1
                east = S[i,j+1]
                southeast = S[i+1,j+1] if i < n - 1 else -1
                S[i,j] = a[i,j] + max(northeast, east, southeast)

    return S[0,0]
```

3. Stab schneiden

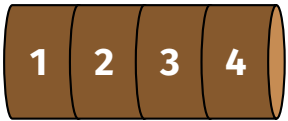
Aufgabenbeschreibung: Stab schneiden



Eingabe:

- Ein Stab der Länge n, den wir zerschneiden. Stellen Sie sich zum Beispiel einen Baumstamm vor.

Aufgabenbeschreibung: Stab schneiden

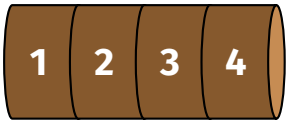


Länge $\boldsymbol{i}$	0	1	2	3	4
Preis $\boldsymbol{p}[\boldsymbol{i}]$	0	1	5	8	9

Eingabe:

- Ein Stab der Länge n, den wir zerschneiden. Stellen Sie sich zum Beispiel einen Baumstamm vor.
- Eine Preisliste p für Stäbe verschiedener Längen. Wir fügen das offensichtliche Feld $i = 0$ hinzu, damit die Länge dem Index entspricht.

Aufgabenbeschreibung: Stab schneiden



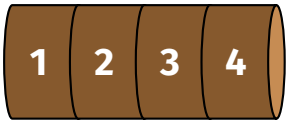
Länge $\boldsymbol{i}$	0	1	2	3	4
Preis $\boldsymbol{p}[\boldsymbol{i}]$	0	1	5	8	9

Eingabe:

- Ein Stab der Länge n, den wir zerschneiden. Stellen Sie sich zum Beispiel einen Baumstamm vor.
- Eine Preisliste p für Stäbe verschiedener Längen. Wir fügen das offensichtliche Feld $i = 0$ hinzu, damit die Länge dem Index entspricht.

Ziel: Zerschneide den Stab so, dass die Stücke zum höchstmöglichen Preis verkauft werden können.

Aufgabenbeschreibung: Stab schneiden



Länge i	0	1	2	3	4
Preis $p[i]$	0	1	5	8	9

Eingabe:

- Ein Stab der Länge n, den wir zerschneiden. Stellen Sie sich zum Beispiel einen Baumstamm vor.
- Eine Preisliste p für Stäbe verschiedener Längen. Wir fügen das offensichtliche Feld $i = 0$ hinzu, damit die Länge dem Index entspricht.

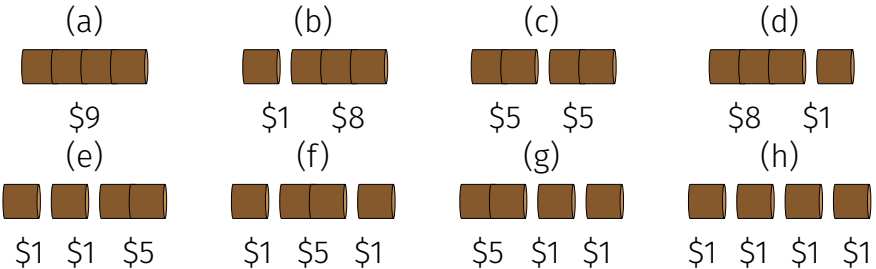
Ziel: Zerschneide den Stab so, dass die Stücke zum höchstmöglichen Preis verkauft werden können.

Ausgabe: Der maximale Wert $S[n]$, der durch das Zerschneiden des Stabes erzielt werden kann.

Beispiel

Für einen Stab der Länge 4 gibt es 8 Möglichkeiten, ihn zu zerschneiden:

Länge i	0	1	2	3	4
Preis $p[i]$	0	1	5	8	9



Wobei (c) mit \$5 + \$5 = \$10 den höchsten Gewinn erzielt.

Aufgabenbeschreibung

- **Problem**: Für einen Stab der Länge n gibt es 2^{n-1} Möglichkeiten.

Aufgabenbeschreibung

- **Problem**: Für einen Stab der Länge n gibt es 2^{n-1} Möglichkeiten.

  $\rightarrow$ Um ein Verständnis zu erhalten, beginnen wir mit den kleinsten Längen.

Die drei Schritte der DP

1. **Identifiziere die Teilprobleme** und analysiere, wie sie sich überlappen.
2. **Definiere die rekursive Formel** durch Betrachtung lokaler Übergänge und optimaler Entscheidungen.
3. **Implementiere eine Lösung** mit einer geeigneten Datenstruktur und der korrekten Berechnungsreihenfolge.

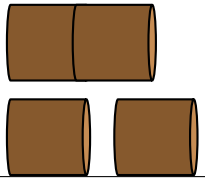
Schritt 1: Identifiziere die Teilprobleme

- Für Stäbe der Länge 0 und 1 haben wir nur jeweils eine Option.

Länge	Optionen	Optimaler Preis
0		0
1		$p[1]$

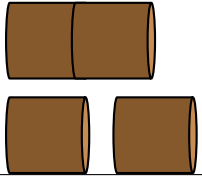
Schritt 1: Identifiziere die Teilprobleme

- Für einen Stab der Länge 2 haben wir bereits mehrere Optionen.

Länge	Optionen	Preisoptionen
0		`p[2]` `p[1] + p[1]`

Schritt 1: Identifiziere die Teilprobleme

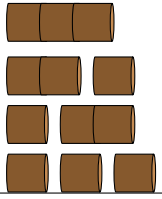
- Für einen Stab der Länge 2 haben wir bereits mehrere Optionen.
- Die bessere der beiden bestimmt den optimalen Preis.

Länge	Optionen	Optimaler Preis
0		$\max\begin{Bmatrix} \texttt{p[2]} \\ \texttt{p[1] + p[1]} \end{Bmatrix}$

Schritt 1: Identifiziere die Teilprobleme

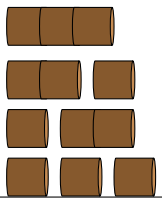
- Die möglichen Schnitte für einen Stab der Länge 3 hinterlassen einige Stücke mit einer nicht-trivialen Länge von 2.

Länge	Optionen	Preisoptionen

0		`p[3]`
		`S[2]` + `p[1]`
		`p[1]` + `S[2]`
		`p[1]` + `p[1]` + `p[1]`

Schritt 1: Identifiziere die Teilprobleme

- Die möglichen Schnitte für einen Stab der Länge 3 hinterlassen einige Stücke mit einer nicht-trivialen Länge von 2.
- Der optimale Preis des Stabes der Länge 3 hängt vom optimalen Preis für den Stab der Länge 2 ab.

Länge	Optionen	Optimaler Preis
0		$\max \begin{cases} \texttt{p[3]} \\ \texttt{S[2]} + \texttt{p[1]} \\ \texttt{p[1]} + \texttt{S[2]} \\ \texttt{p[1]} + \texttt{p[1]} + \texttt{p[1]} \end{cases}$

Schritt 1: Identifiziere die Teilprobleme

- Was sind die Teilprobleme?

Schritt 1: Identifiziere die Teilprobleme

- Was sind die Teilprobleme?
  $S[i]$ bezeichnet den optimalen Preis, den man für einen Stab der Länge i erhalten kann. Er hängt von allen $S[j]$ mit $j < i$ ab.

Schritt 2: Definiere die rekursive Formel

- Der optimale Preis eines Stabes der Länge i besteht aus dem Preis des abgeschnittenen Stücks der Länge k und dem Preis des verbleibenden Stücks der Länge $i - k$. Wie sieht das als Formel aus?

Schritt 2: Definiere die rekursive Formel

- Der optimale Preis eines Stabes der Länge i besteht aus dem Preis des abgeschnittenen Stücks der Länge k und dem Preis des verbleibenden Stücks der Länge $i - k$. Wie sieht das als Formel aus?

$$S[i] = p[k] + S[i - k]$$

Schritt 2: Definiere die rekursive Formel

- Der optimale Preis eines Stabes der Länge i besteht aus dem Preis des abgeschnittenen Stücks der Länge k und dem Preis des verbleibenden Stücks der Länge $i - k$. Wie sieht das als Formel aus?

$$S[i] = p[k] + S[i - k]$$

- Da die optimale Schnittlänge k unbekannt ist, müssen alle möglichen Optionen bewertet werden, um die beste auszuwählen. Wie machen wir das?

Schritt 2: Definiere die rekursive Formel

- Der optimale Preis eines Stabes der Länge i besteht aus dem Preis des abgeschnittenen Stücks der Länge k und dem Preis des verbleibenden Stücks der Länge $i - k$. Wie sieht das als Formel aus?

$$S[i] = p[k] + S[i - k]$$

- Da die optimale Schnittlänge k unbekannt ist, müssen alle möglichen Optionen bewertet werden, um die beste auszuwählen. Wie machen wir das?

$$S[i] = \begin{cases} 0 & \text{if } i = 0 \\ \max\{p[k] + S[i - k] : k \in [1, i]\} & \text{if } i \neq 0 \end{cases}$$

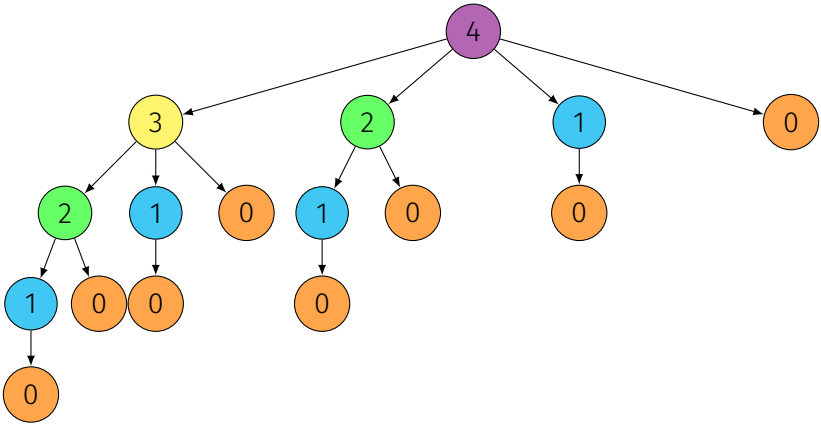
- Beachte, dass diese Formel nur gilt, wenn p einen Preis für jede Länge von 1 bis i angibt.

Rekursiver Algorithmus

```
def max_value(p, n):

    #Base case
    if n == 0:
        return 0

    #Optimal Price
    options = [p[k] + max_value(p, n-k) for k in range(1, n+1)]
    return max(options)
```

Überlappende Teilprobleme

Basierend auf dem Beispiel eines Stabes der Länge 4.



- Die Teilprobleme werden mehrfach berechnet! Dies führt zu einer exponentiellen asymptotischen Laufzeit von $\mathcal{O}(2^n)$.

Dynamische Programmierung

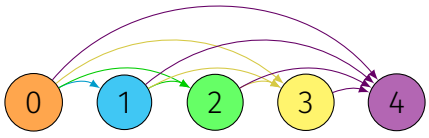
- Ein rekursives Problem mit überlappenden Teilproblemen ist gegeben.
  $\rightarrow$ Stab schneiden kann daher mittels Dynamischer Programmierung optimiert werden.

Schritt 3: Implementierung

- Bestimme eine geeignete Datenstruktur:
  - 1D-Array der Länge $n + 1$
- Identifiziere die Abhängigkeiten:
  - Um $S[i]$ zu berechnen, müssen wir zuerst $S[i-1], \ldots, S[i-k]$ kennen.
  - Letztendlich hängt alles von $S[0]$ ab, dem Basisfall.
- Identifiziere die korrekte Berechnungsreihenfolge:
  - Aus den Abhängigkeiten wissen wir, dass wir bei $i = 0$ beginnen und das Array von links nach rechts ausfüllen müssen: Von kleineren zu grösseren Schnitten.

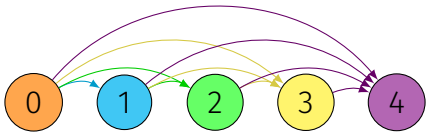
Dynamische Programmierung: Theorie

- $S[3]$, der optimale Preis eines Stabes der Länge 3, hängt von $S[2]$ und $S[1]$ ab, sowie vom Basisfall $S[0]$.
- $S[2]$ hängt von $S[1]$ und $S[0]$ ab.
- $S[1]$ hängt nur von $S[0]$ ab.



Dynamische Programmierung: Theorie

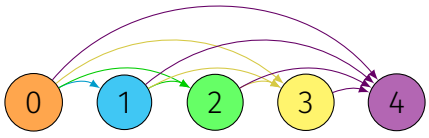
- $S[3]$, der optimale Preis eines Stabes der Länge 3, hängt von $S[2]$ und $S[1]$ ab, sowie vom Basisfall $S[0]$.
- $S[2]$ hängt von $S[1]$ und $S[0]$ ab.
- $S[1]$ hängt nur von $S[0]$ ab.



- Da $S[0] = 0$ bereits bekannt ist, beginnen wir mit $S[1]$ und berechnen die Ergebnisse Schritt für Schritt bis zu $S[n]$.

Dynamische Programmierung: Theorie

- $S[3]$, der optimale Preis eines Stabes der Länge 3, hängt von $S[2]$ und $S[1]$ ab, sowie vom Basisfall $S[0]$.
- $S[2]$ hängt von $S[1]$ und $S[0]$ ab.
- $S[1]$ hängt nur von $S[0]$ ab.



- Da $S[0] = 0$ bereits bekannt ist, beginnen wir mit $S[1]$ und berechnen die Ergebnisse Schritt für Schritt bis zu $S[n]$.
- Es gibt keine Schleifen im Graphen. Wir nennen ihn gerichtet azyklisch. Sie werden dieses Muster in allen DP-Problemen bemerken, die wir präsentieren.

Dynamische Programmierung

Mit diesem Wissen können wir leicht eine "Richtung" finden, um die Probleme zu lösen!

```
import numpy as np
def max_value_dp(p, n):

    # Create data structure
    S = np.zeros(n+1)

    # Implement Base case (redundant through np.zeros)
    S[0] = 0

    #Fill the data structure in the appropriate direction.
    for i in range(1, n+1):
        options = [p[k] + S[i-k] for k in range(1, i+1)]
        S[i] = max(options)

    return S[n]
```

Memoisierung

→ In rot die Änderungen am naiven rekursiven Algorithmus.

```
def max_value(p, n, memo = None):
    # Create data structure if not provided
    if memo is None:
        memo = dict()

    # Compute subproblem only if it hasn't been solved
    if n not in memo:

        if n == 0:
            memo[0] = 0
        else:
            options = [p[k] + max_value(p, n-k, memo)
                for k in range(1, n+1)]
            memo[n] = max(options) # Store the result

    return memo[n]
```

4. Zusammenfassung

Zusammenfassung: Dynamische Programmierung

Die drei Schritte der DP

1. **Identifiziere die Teilprobleme** und analysiere, wie sie sich überlappen.
2. **Definiere die rekursive Formel** durch Betrachtung möglicher Übergänge zwischen Teilproblemen und optimaler Entscheidungen.
3. **Implementiere eine Lösung** mit einer geeigneten Datenstruktur und der korrekten Berechnungsreihenfolge.

Zusammenfassung: Dynamische Programmierung

Allgemeiner Algorithmus für dynamische Programmierung:

1. Initialisiere die Datenstruktur
2. Implementiere den Basisfall
3. Fülle die Datenstruktur aus
4. Gib das Ergebnis zurück

Zusammenfassung: Königsweg

1. Initialisiere Datenstruktur:
   - 2D-Matrix

```
S = np.zeros((n,m))
```

2. Implementiere Basisfall:
   - Äusserste rechte Spalte

```
S[:, m] = a[:, m]
```

3. Fülle die Datenstruktur aus:
   - Von rechts nach links

```
for j in [...]:
  for i in [...]:
    opt1 = [...]
    opt2 = [...]
    opt3 = [...]
    S[i, j] = max(op1, opt2, opt3)
```

4. Gib Ergebnis zurück

```
return S[0,0]
```

Zusammenfassung: Stab schneiden

1. Initialisiere Datenstruktur:
   - 1D-Array

```
S = np.zeros(n+1)
```

2. Implementiere Basisfall:
   - Stab der Länge 0

```
S[0] = 0
```

3. Fülle die Datenstruktur aus:
   - Variable Anzahl an Optionen

```
for i in [...]:
    options = [... for k in cuts]
    S[i] = max(options)
```

4. Gib Ergebnis zurück

```
return S[n]
```

BAUG 2023

Gegeben ist eine Liste A vom Typ `int` der Länge n. Eine Spielfigur befindet sich auf Position 0. Die Spielfigur, auf Position i, kann entweder die Belohnung $A[i]$ beanspruchen und dann zur Position $i+1$ bewegen oder sie kann stattdessen die Belohnung $2A[i]$ beanspruchen und dann zur Position $i+2$ bewegen, solange sie die Liste nicht verlässt. Erreicht die Figur die letzte Position $n-1$, nimmt sie die Belohnung $A[n-1]$ ein und das Spiel endet.

Betrachten Sie die folgende Liste A: [3, 2, 3, 1, 1].
Was ist die maximal erreichbare Belohnung?

Betrachten Sie die folgende unvollständige Rekurrenz

$$S(i)\begin{cases}0 & \text{if } i = n\\ A[n-1] & \text{if } i = n-1\\ ??? & \text{if } 0 \le i < n-1\end{cases}$$

Welche ist die fehlende Aussage?

6. Hausaufgaben

Übung 9: DP I

Übung 9: DP I

- Tribonacci
- Catalan-Zahlen
- Blöcke
- Aufgabenplanung 2.0

Abgabedatum: Montag 04.05.2026, 20:00 MEZ

KEINE HARDCODIERUNG

Fragen?

Feedback

https://jschultev.github.io/personal_website/Feedback