Dynamische Programmierung 7. November 2016

Handout 7: DP Theorie

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Dynamic Programming

Those who cannot remember the past are doomed to repeat it
- George Santayana, The Life of Reason

Referenz

Folgendes Handout fasst einige Notizen zu Dynamischer Programmierung zusammen. Sie sind teilweise direkt übernommen von folgenden Quellen

- Algorithm Design, Kleinberg, Tardos, Kapitel 6
  Eine der besten einführenden Texte, das Kapitel ist online frei verfügbar
  http://www.aw-bc.com/info/kleinberg/chapter.html
  (http://goo.gl/oAYE1)
- Introduction to Algorithms, Cormen et al., Kapitel 15
  Sehr ausführliche Darstellung
- Algorithms, Dasgupta et al., Kapitel 6
  http://www.cs.berkeley.edu/~vazirani/algorithms/
  (http://goo.gl/oA9QJ)
- Prof. Jeff Erickson Lecture Notes
  http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/
  (http://goo.gl/Kd52A)
- The Source of All Lies
  http://en.wikipedia.org/wiki/Dynamic_programming
  (http://goo.gl/dxwf)

Dynamic Programming

Idee: "Rekursion ohne Wiederholung"

Dynamic Programming (*Dynamisches Programmieren*) ist eine Technik für die effiziente Implementierung gewisser rekursiv formulierter (Optimierungs)Probleme. Dies geschieht durch Speichern von Teilresultaten.

Wie *Divide & Conquer* - Algorithmen findet ein solcher Algorithmus die Lösung für ein Problem durch Kombination von Lösungen von Teilproblemen. *Im Gegensatz allerdings zu den Divide & Conquer Algorithmen, ist Dynamic Programming anwendbar, wenn die Teilprobleme "überlappend" sind. Dies führt dazu, dass ein naiver rekursiver*

Algorithmus Teilprobleme wiederholt löst und deshalb einen exponentiellen Zeitaufwand aufweist.

Dynamic Programming löst jedes Teilproblem nur einmal und speichert die Lösung ab. D.h. man exploriert implizit alle möglichen Lösungen, in dem man das Problem in eine Reihe von *Teilproblemen* zerlegt und dann mit den Lösungen für diese, die Lösung für grössere Teilprobleme berechnet.

Wir werden die Charakteristiken von Dynamic Programming an Hand eines konkreten Beispiels, des *Rucksack Problems*, erörtern.

Das 0/1 Rucksackproblem

Problem

Gegeben sind n Gegenstände mit jeweils ganzzahligem Gewicht $w_1, w_2, \ldots, w_n$ und einem Wert $v_1, \ldots, v_n$ sowie einen Rucksack mit einem maximalen Tragevermögen von W.

Ziel: Finde eine Auswahl $S \subset \{1, \ldots, n\}$ der Gegenstände, so dass der Gesamtwert $\sum_{i \in S} v_i$ maximal ist und das Gesamtgewicht vom Rucksack getragen werden kann, also $\sum_{i \in S} w_i \leqslant W$ ist.

1. Struktur einer optimalen Lösung (aka Aufteilung in Teilprobleme)

Folgendes gilt ziemlich offensichtlich für eine optimale Lösung: Entweder ist der n.te Gegenstand (der letzte) in der Auswahl oder nicht. Schauen wir beide Möglichkeiten an:

- Ist der $n$te Gegenstand nicht in der Auswahl, dann muss dieselbe Auswahl auch optimal sein für die Situation, wenn wir nur die Gegenstände 1 bis $n-1$ haben.
- Ist der $n$te Gegenstand in der optimalen Auswahl, dann ist der Wert v_n gewonnen, aber der Rucksack hat nur noch ein Tragevermögen von $W - w_n$. Wir können diesen Restplatz mit den verbliebenen Gegenständen füllen.
  Beobachtung: eine optimale Auswahl für das gesamte Problem ist auch eine optimale Auswahl für die Gegenstände 1 bis $n-1$ wenn ein Tragegewicht von $W - w_n$ übrig ist! Dies kann durch einen Widerspruchsbeweis (Cut & Paste - Argument) gezeigt werden:

  Ist dies nicht der Fall, so ersetze (*Cut*) die Lösung für die Gegenstände 1 bis $n-1$ durch eine optimale (bessere) (*Paste*) und nehme zusätzlich den letzten ($n$ten) Gegenstand. Die so erreichte Gesamtlösung ist nun besser als die vorherige Gesamtlösung. Dies ist ein Widerspruch zur Annahme der Optimalität.

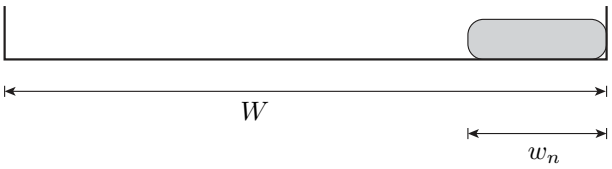


Abb.: Nach dem der n.te Gegenstand genommen wurde ist ein Gewicht von w_n gebraucht und es ist noch eine Tragevermögen von $W - w_n$ vorhanden

All dies legt nahe, dass um eine optimale Lösung für das gesamte Problem zu erhalten, wir die optimalen Lösungen von Teilproblemen (mit weniger Gegenständen, Tragegewicht) betrachten müssen. Das heisst, das Problem hat eine "Optimale (Sub)struktur": Eine (optimale) Lösung für das Gesamtproblem kann aus (optimalen) Lösungen für Teilprobleme berechnet werden.

Dies ist ein Kriterium, dass ebenfalls auch für Divide & Conquer Ansätze gelten muss.

2. Rekursive Formulierung

Obige Zusammenhänge können wir nun direkt rekursiv formulieren.

Sei $K(i, w)$ der Gesamtwert einer optimalen Auswahl aus den Gegenständen 1,. . . ,i für einen Rucksack mit einem Tragevermögen von w. Die gesuchte Gesamtlösung ist also gegeben durch $K(n, W)$ (alle Gegenstände, das gesamte Tragevermögen).

- *Verankerung*: Für $i = 0$: $K(0, W)$ hat die trivially Lösung 0
- Mit einer analogen Argumentation wie oben für das Gesamtproblem ist klar, dass wenn $w < w_i$ ist, dann ist $K(i, w) = K(i-1, w)$ (der $i$te Gegenstand ist zu schwer). Ansonsten

$$K(i,w) = \max \left\{ \underbrace{K(i-1,w)}_{\text{nehme } i\text{ten Gegenstand nicht}} \quad , \quad \underbrace{v_i + K(i-1, W-w_i)}_{\text{nehme } i\text{ten Gegenstand}} \right\}$$

Mit diesen Überlegungen erhalten wir direkt einen rekursiven Algorithmus der $K(n, W)$ berechnet:

```
COMPUTE-K(i, w)
  if i == 0
      return 0
  else if w < w_i
          return COMPUTE-K(i − 1, w) // zu schwerer Gegenstand
      else return max (COMPUTE-K(i − 1, w), v_i + COMPUTE-K(i − 1, w − w_i))
```

Versuchen wir den oben formulierten rekursiven Algorithmus zu analysieren: leider zeigt sich, dass wenn wir den Algorithmus wie so hingeschrieben implementieren, er im Worst-Case einen exponentiellen Zeitaufwand benötigt (Think about it! Versuche eine Klasse von Beispielen anzugeben, die den Worst-Case provoziert).

Nun sehe aber folgendes: $K(i, w)$ hat zwei Parameter. Es gibt also nur maximal $n*W$ verschiedene (nichttriviale) Aufrufe/Teilprobleme!. Was bedeutet dies? Da der Algorithmus eine exponentielle Laufzeit hat, heisst dies (Schubfachprinzip), dass er notwendigerweise (viele) Teilprobleme wiederholt löst!

Betrachte folgendes Beispiel mit Gewichten $3, 4, 7$:

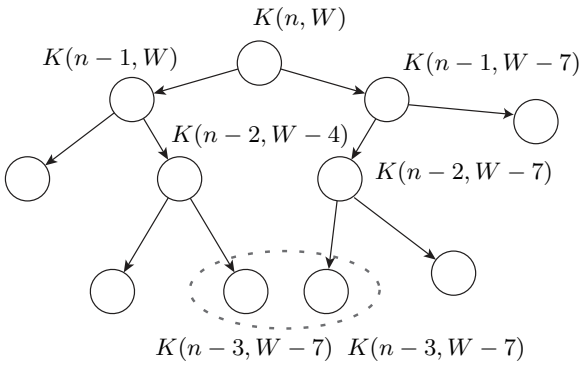


Abb.: Wiederholte Berechnung von Teilproblemen

Auf grösseren Beispielen wird die Mehrfachberechnung deutlicher. Denke hierbei auch z.B. an die naive rekursive Implementierung der ersten Programmieraufgabe und wieso diese schlecht (exponentiell) war.

Memoization

Konfrontiert mit obiger Beobachtung, kann die Idee sein, dass man Resultate "cached": Wir speichern den Wert von $K(i, w)$ in einer global zugreifbaren Datenstruktur das erste Mal nachdem wir ihn berechnet haben ab, und verwenden danach einfach den vorberechneten Wert in allen zukünftigen rekursiven Aufrufen (dies kann natürlich nur funktionieren, wenn die zu betrachtende Funktion keine "side-effects" hat, was aber hier der Fall ist). Dieses Verfahren nennt sich *Memoization* (ja, kein 'r' ;-)).

"Memoization", entstanden 1968 durch Donald Michie aus seinem Artikel *Memo functions and machine learning*, ist aus dem lateinischen Wort Memoradum abgeleitet, was so viel wie "das zu Erinnernde" bedeutet.

Im Allgemeinen kann diese Datenstruktur ein Array, Hash-Table, Suchbaum etc. sein. In diesem Beispiel reicht aber ein 2 dimensionales Array $M[0 \ldots n][0 \ldots W]$. $M[i][w]$ sei zu Beginn "undefiniert" (NIL), wird aber den Wert von $K(i, w)$ enthalten, sobald dieser berechnet wurde. Folgende Prozedur implementiert dieses Verfahren.

```
M-Compute-K(i, w)
    if i == 0
        return 0
    else if M[i][w] ≠ Nil
            return M[i][w]
        else if w < w_i
                M[i][w] = M-Compute-K(i − 1, w)
            else M[i][w] = max(M-Compute-K(i − 1, w), v_i + M-Compute-K(i − 1, w − w_i))
            return M[i][w]
```

Behauptung. *Die Laufzeit von* M-Compute-K(n, W) *ist* $\mathcal{O}(nW)$.

Proof. Die Zeit in einem einzelnen Aufruf von M-Compute-K ist $\mathcal{O}(1)$, ausgenommen die Zeit die für die rekursiven Aufrufe benötigt wird. Die Gesamtlaufzeit ist deshalb beschränkt durch eine Konstante mal die totale Anzahl der Aufrufe von M-Compute-K. Da die Implementation keinen expliziten Aufschluss über die totale Anzahl der rekursiven Aufrufe gibt, suchen wir nachen einem guten "Mass" für den "Progress". Ein solches ist die Anzahl Einträge in M die nicht "undefiniert" sind. Zu Beginn ist diese Zahl 0; aber jedes Mal wenn die Prozedur die rekursiven Aufrufe tätigt wird am Ende ein neuer Eintrag ausgefüllt, und erhöht deshalb diese Zahl um 1. Da M aber höchstens nW Einträge hat, folgt daraus, dass die Anzahl Aufrufe an M-Compute-K höchstens nW ist und deshalb ist die Gesamtlaufzeit von M-Compute-K(n, W) in $\mathcal{O}(nW)$.

□

3. Bottom-Up Berechnung, DP

In vielen Fällen können wir die rekursiven Aufrufe analysieren und die Teilresultate a priori direkt in der Datenstruktur abspeichern. Der Punkt ist, dass wir direkt die Einträge in M mit einem iterativen Algorithmus berechnen können, anstatt mit memoisierter Rekursion. Diese Überführung des Caches in ein oft (multi-dimensionales) Array führt vielfach zu einem einfacheren, direkteren und eleganteren Algorithmus. Manchmal kann damit auch die Zeit und Speicherkomplexität gegenüber Memoization verbessert werden.

Es ist dieser *Antizipierungs*-Schritt, die systematische iterative Vorberechnung der Teilprobleme, (*Iteration über Teilprobleme*) die Dynamic Programming auszeichnet. Dafür ist es notwendig, dass es eine Durchlaufordnung der Teilprobleme von den "kleinsten" zu den "grössten" gibt, so dass für jedes Teilproblem alle benötigen Teilresultate bereits berechnet worden sind.

In dem Rucksack-Problem Beispiel können wir zum Beispiel sehen, dass jeder zu berechnende Eintrag $M[i][w]$ nur von maximal zwei Werten $M[i-1][w]$ und $M[i-1][w-w_i]$ abhängig, wobei die Parameter der benötigen Resultate kleiner sind. Wir können deshalb diese Werte iterativ nach aufsteigenden Parameterwerten berechnen.

Die folgende Prozedur implementiert dieses Verfahren:

I-COMPUTE-K

// Array $M[0 \ldots n, 0 \ldots W]$
initialize $M[0][w] = 0$ for each $w = 0, \ldots, W$ // Verankerung
for $i = 1$ **to** n
    for $w = 0$ **to** W
        if $w < w_i$
            $M[i][w] = M[i-1][w]$ // zu schwerer Gegenstand
        else $M[i][w] = \max(M[i-1][w], v_i + M[i-1][w-w_i])$
return $M[n][W]$

Laufzeit: $\mathcal{O}(nW)$

Folgende Tabelle zeigt die sukzessive Berechnung der Teilprobleme. In dem wir eine geeignete *Evaluation Order* (Durchlaufordnung) wählen (hier: links-rechts, oben-unten) können wir garantieren, dass für jedes Teilproblem die zu seiner Lösung benötigten Lösungen von (kleineren) Teilprobleme bereits berechnet wurden.

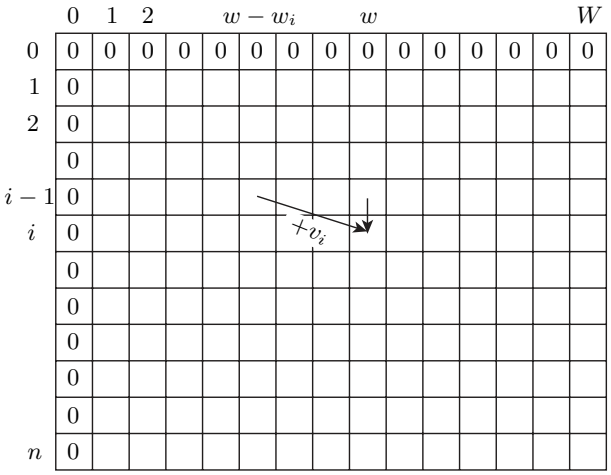


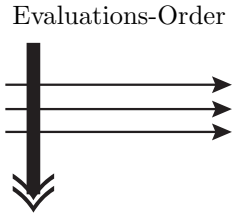
Abb.: 2-dimensionale Tabelle der K Werte. Die linke Spalte und oberste Zeile ist immer 0.
Eintrag $K(i, w)$ wird berechnet mit den beiden Einträgen $K(i-1, w)$ und $K(i-1, w-w_i)$
wie durch die Pfeile angezeigt.

Beispiel: Es sei $W = 10$ und folgende Gegenstände sind gegeben:

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

Folgend die ausgefüllte Tabelle

K \ W	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90



Bemerkungen:

- Die Lösung ist gegeben durch das letzte Feld der Tabelle, $M[4][10] = 90$
- Das bisher beschriebene Verfahren gibt nicht an, welche Teilmenge die optimale Lösung ergibt. (Es ist $\{2, 4\}$ in diesem Beispiel)

Beobachtung/Anmerkung: Vollziehe folgende Überlegung nach: jedem Dynamic Programming Algorithmus ist implizit ein DAG (Directed Acyclic Graph) zugrunde liegend:

Wir stellen uns die Teilprobleme als Knoten in einem gerichteten Graph vor, und jede Kante repräsentiert einen "Precedence Constraint" auf die Durchlaufordnung in welcher die Teilprobleme abgearbeitet werden. Eine Kante (i, j) bedeutet, dass die Lösung des Teilproblems i für die Berechnung der Lösung des Teilproblems j notwendig ist (vergleiche mit den Pfeilen in der Abbildung der Tabelle für das Rucksackproblem).

Dann ist eine Durchlaufordnung gültig wenn sie einer *topologischen Sortierung* dieses DAG entspricht!

Im allgemeinen gibt es mehrere gültige Durchlaufordnungen. Im Beispiel des Rucksack-Problems könnten wir auch zuerst aufsteigend nach w und dann aufsteigend nach i durchgehen. Welche der möglichen validen Durchlaufordnungen gewählt wird ist manchmal beliebig und kann nicht immer mit der asymptotischen Betrachtungsweise beantwortet werden. Beispielsweise könnten wir ein Verfahren priorisieren, dass die beste "Lokalität" der Zugriffe auf die Matrix aufweist, so dass ein gutes Cache Verhalten gewährleistet ist.

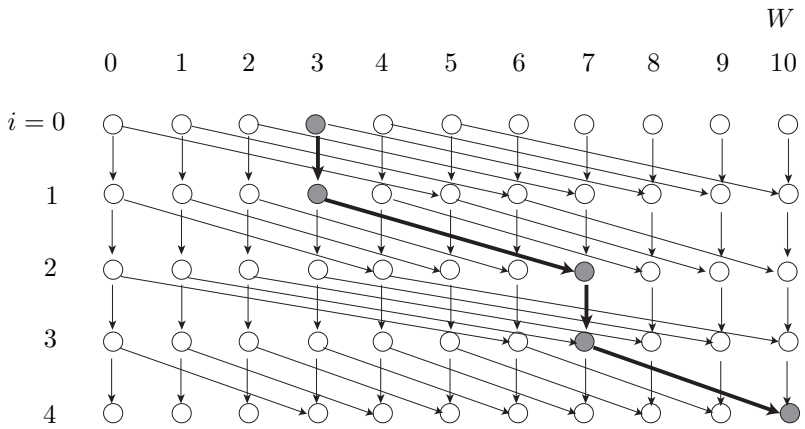


Abb.: Der zugrundeliegende DAG der Teilprobleme für das oben gewählte Beispiel

Bemerke, dass wenn wir die Kanten gewichten mit 0, wenn wir den $i$ten Gegenstand nicht nehmen (Kanten die vertikal nach unten gehen) und mit v_i wenn wir den $i$ten Gegenstand nehmen, wir dann den *längsten Weg im DAG* suchen (der beim letzten Knoten endet).

Dynamische Programmierung funktioniert oft gut auf Objekten die linear geordnet sind und deren (relative) "Position" nicht geändert werden kann: Beispielsweise Characters in einem String, Matrizen in einem Kettenprodukt, Punkte entlang dem Kantenzug eines Polygons, die links-nach-rechts Ordnung der Blätter eines Suchbaumes. (Versuche Dynamic Programming wenn man mit Objekten konfrontiert ist die in einer (links-rechts) Art sortiert sind)

Die beiden Ansätze *Dynamic Programming* und *Memoization* ähneln sich auf eine Weise stark, da sie beide daraus entstanden sind, die zugrunde liegende Rekursionsgleichung zu analysieren und zu sehen, dass Teilprobleme wiederholt gelöst werden. Im Gegensatz zu Dynamic Programming bewahrt aber Memoization die grundlegende Struktur/Vorgehensweise des rekursiven Verfahrens.

Beide Ansätze führen vielfach auf die gleiche Laufzeit, aber die (verborgenen) konstanten Faktoren in der Big-O Notation können bei Memoization erheblich grösser sein auf Grund des Overheads der Rekursion.

In Spezialfällen allerdings kann Memoization vorteilhaft sein: Dynamic Programming löst typischerweise jedes mögliche Teilproblem das eventuell gebraucht werden könnte, während Memoization nur diejenigen löst die wirklich gebraucht werden. Nehme beispielsweise an, dass W und alle Gewichte w_i ein Vielfaches von 100 sind. Dann ist ein Teilproblem $K(i, w)$ unnötig wenn w nicht durch 100 teilbar ist. Das memoisierte Verfahren wird diese überflüssigen Einträge nie betrachten.

Stop and Think: Unter welchen Umständen kann es schwierig/nicht möglich sein, einen iterativen Algorithmus zu erhalten? Wann ist nur Memoizing möglich?

4. Lösung berechnen

Der bisherig beschriebene Algorithmus berechnet nicht eine explizite Auswahl die zum optimalen Wert führt. Um die tatsächliche Lösung zu erhalten, gibt es folgende Möglichkeiten

- **Rückverfolgung** im ausgefüllten Tableau
  Beginnend vom Lösungsfeld, nachvollziehen wir, anhand der Werte in der Tabelle, welche Wahl wir während des Lösungsvorgangs getroffen haben ($i$ter Gegenstand nehmen?). Wir wandern also den DAG rückwärts bis zum Feld $[0, 0]$.

  Vergleiche auch das anschliessende *Triangle*-Beispiel.

  Aufgabe: Zeige, dass es in diesem Beispiel in $\mathcal{O}(n)$ Zeit möglich ist, die tatsächliche Lösung mit Hilfe der ausgefüllten Tabelle zu rekonstruieren.

- **Explizite Speicherung der getroffenen Wahl** (benötigt zusätzlichen Speicher)
  Wir verwenden ein zusätzliches Boolean Array $take[i][w]$ das TRUE ist wenn wir den $i$ten Gegenstand in $K(i, w)$ verwenden und FALSE sonst. Wir setzen die Werte während des Ausfüllen der Tabelle (abhängig, welche Wahl zum jeweiligen maximalen Eintrag führt).

  Wie bestimmen wir anschliessend die optimale Auswahl S? Wenn $take[n][W] ==$ TRUE, dann ist $n \in S$. Wir können nun die selbe Argumentation wiederholen für $take[n-1][W-w_n]$. Ist $take[n][W] ==$ FALSE, dann ist $n \notin S$ und wir fahren mit $take[n-1][W]$ fort.

  Folgende Prozedur implementiert dieses Verfahren und bestimmt die optimale Auswahl S in $\mathcal{O}(n)$

```
// Berechne S
S = ∅
w = W
for i = n downto 1
    if take[i][w] == TRUE
        S = S ∪ {i}
        w = w − w_i
```

Zusammenfassend gilt: Das Rucksack Problem kann in $\mathcal{O}(nW)$ Zeit gelöst werden.

Stop and Think: Es ist bekannt, dass das Rucksack Problem *NP-hard*, also ein schwieriges Problem ist. Dies bedeutet insbesondere, dass bisher kein polynomieller Algorithmus bekannt ist. Betrachte die erreichte Laufzeit von $\mathcal{O}(nW)$ genauer unter diesem Aspekt.

Prinzipien/Charakteristiken von Dynamic Programming

(Iteration über Teilprobleme, Antizipation)

Charakterisik (Wann ist DP anwendbar?)

- *Optimale (Sub)struktur*: Die (optimale) Lösung kann aus (optimalen) Lösungen für Teilprobleme berechnet werden. (Principle of Optimality)
- *Überlappende Teilprobleme*: $\rightsquigarrow$ dieselben Teilprobleme treten wiederholt auf. (ansonsten müssen die Lösungen nicht gespeichert werden)
  Dies ist ein wesentlicher Unterschied zu Divide & Conquer: hier sind Teilprobleme (meist) disjunkt.
- Die Parameter/Teilprobleme können antizipiert werden ($\rightsquigarrow$ Bottom-up Verfahren, Vorberechnung)
- Es gibt nur wenige (polynomiell) viele Teilprobleme die betrachtet werden müssen (ansonsten erhalten wir keinen polynomiellen Algorithmus).

Schema einer DP Lösung

1. **Struktur**: Charakterisiere die Struktur einer optimalen Lösung. Zeige das sie zerlegt werden kann in *Teilprobleme* (resp. optimale Lösungen zu Teilproblemen). Beweise die Optimalität durch Widerspruchsbeweis (*Cut & Paste* - Argument). (Optimalitätsprinzip von Bellman)
2. **Rekursion**: Definiere die optimale Lösung rekursiv mittels optimalen Lösungen für Teilprobleme (top-down).

   (Die beiden ersten Schritte sind oft die schwierigsten)

   Hier hilft es oft darüber zu denken, ob das Problem in eine Reihe von "Decisions"/ Ent-scheidungen unterteilt werden kann und ob diese dann auf kleinere Teilprobleme der gleichen Art führen. Betrachte z.B. letzte Entscheidung (a la "ist Element n in der Auswahl?")

   Überprüfe ob Teilresultate wiederholt gelöst werden.
3. **Bottom-Up**: Berechne die optimale Lösung bottom-up und speichere die Lösungen zu Teilproblemen ab. Nach vorne schauen. (*Antizipierungs*-Schritt)

   Gebe einen Algorithmus an der mit den "Base-Cases"/Verankerung der Rekursion beginnt (Initialisierung) und sich hocharbeitet zur Gesamtlösung in einer korrekten Durchlaufordnung.

   Vorgehen:

   i) **Identifiziere Teilprobleme**: Auf welche verschiedene Arten kann der rekursive Algorithmus sich selbst aufrufen? Was ist der Wertebereich der Parameter?

   ii) **Analysiere die Zeit- und Speicherkomplexität**: Die Anzahl der möglichen Teilprobleme bestimmt die Speicherkomplexität des memoisierten Algorithmus. Die Gesamtlaufzeit einer memoisierten Version ist beschränkt durch (eine Konstante mal) die totale Anzahl der rekursiven Aufrufe.

   iii) **Wähle eine Datenstruktur um Teilresultate zu memoisieren**: Für viele Probleme können die Teilprobleme durch ein paar Integer-Werte identifiert werden, so dass ein (multidimensionales) Array reicht. Für andere Probleme mag eine etwas aufwendigere Datenstruktur notwendig sein.

iv) **Identifiziere Abhängigkeiten zwischen Teilproblemen**: Bis auf die trivialen Fälle hängt jedes rekursive Teilproblem von anderen Teilproblemen ab – welche?

v) **Finde eine gültige Durchlaufordnung**: Beginne mit den trivialen Fällen (Initialisierung).

4. **Lösung berechnen**: Berechne eine Lösungsinstanz zu deinem optimalen Wert.

Anders ausgedrückt ist Folgendes bei einer DP-Lösung (mindestens) erwartet

1. Definiere eine DP Tabelle (Bedeutung eines Feldes)
2. Zeige wie jedes Feld mit Hilfe anderer Felder berechnet werden kann (Rekursionsgleichung)
3. Gib eine Reihenfolge an, in der die Felder berechnet werden, so dass alle benötigten Felder bereits berechnet sind
4. Gib an, wie die Lösung aus der Tabelle abgelesen werden kann

Anmerkungen

Zusammenfassend gesagt ist Dynamic Programming Rekursion ohne Wiederholung. Dynamic Programming Algorithmen speichern Lösungen zu Teilproblemen ab um eine Mehrfachberechnung zu verhindern, dies oftmals (aber nicht immer) in einer Art Tabelle/Array.

Dynamic programming is *not* about filling in tables! It's about smart recursion.

Einige häufige Teilprobleme

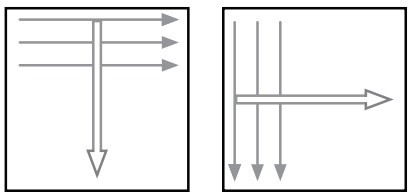
Die richtige rekursive Formulierung und Art der Teilprobleme zu finden braucht etwas Übung. Es gibt aber ein paar Formen die häufig in Dynamic Programming aufzutreten scheinen.

i) Die Eingabe ist $x_1, \ldots, x_n$ und ein Teilproblem ist $x_1, \ldots, x_i$.

$$\boxed{x_1\ x_2\ x_3\ x_4\ x_5\ x_6}\ x_7\ x_8\ x_9\ x_{10}$$

Die Anzahl der Teilprobleme ist deshalb linear.

Gültige Durchlaufordnungen (in 2D) sind oft

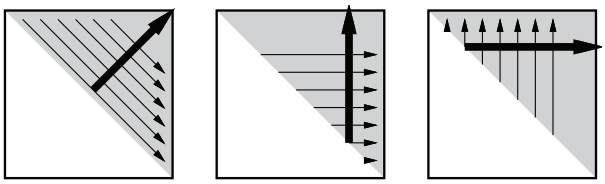


ii) Die Eingabe ist $x_1, \ldots, x_n$ und ein Teilproblem ist $x_i, x_{i+1}, \ldots, x_j$.

$$x_1\ x_2\ \boxed{x_3\ x_4\ x_5\ x_6}\ x_7\ x_8\ x_9\ x_{10}$$

Die Anzahl der Teilprobleme ist $\mathcal{O}(n^2)$

Gültige Durchlaufordnungen (in 2D) sind oft

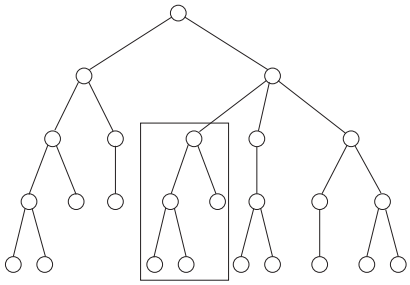


iii) Die Eingabe ist $x_1, \ldots x_n$ und $y_1, \ldots, y_m$. Ein Teilproblem ist $x_1, \ldots, x_i$ und $y_1, \ldots, y_j$.

$$\boxed{x_1\ x_2\ x_3\ x_4\ x_5\ x_6}\ x_7\ x_8\ x_9\ x_{10}$$
$$\boxed{y_1\ y_2\ y_3\ y_4\ y_5}\ y_6\ y_7\ y_8$$

Die Anzahl der Teilprobleme ist $\mathcal{O}(nm)$.

iv) Die Eingabe ist ein "rooted" tree. Ein Teilproblem ist ein Teilbaum.



Hat der totale Baum n Knoten, gibt es n Teilprobleme.

Wieso heisst das ganze "Dynamic Programming"?

Der Begriff wurde von dem amerikanischen Mathematiker Richard Bellman eingeführt, der diese Methode auf dem Gebiet der Regelungstheorie anwendete.

Richard Bellman über die Namenswahl:

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word "research". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term "research" in his presence. You can imagine how he felt, then, about the term "mathematical". The RAND Corporation was employed by the Air Force,

and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?

Vergleiche auch hierzu:

The word dynamic was chosen by Bellman to capture the time-varying aspect of the problems, and also because it sounded impressive. The word programming referred to the use of the method to find an optimal program, in the sense of a military schedule for training or logistics (planning).

Design rekursiver Algorithmen

Folgend sind ein paar häufig auftretende rekursive Definitionen, die Grundlage für eine rekursive Gleichung sein können

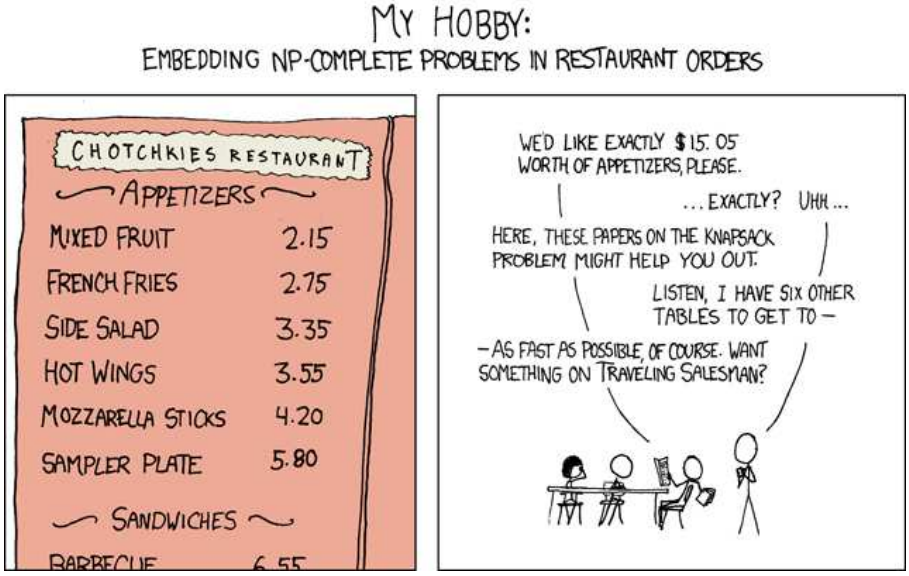
- Eine natürliche Zahl ist entweder 0 oder der Nachfolger einer natürlichen Zahl
- Eine Sequenz/Liste ist entweder leer (empty) oder ein Element gefolgt von einer Sequenz/Liste
- Eine Sequenz der Länge n ist entweder leer, ein Singelton, oder eine Sequenz der Länge $\lfloor n/2 \rfloor$ gefolgt von einer Sequenz der Länge $\lceil n/2 \rceil$
- Eine Menge ist entweder leer oder die Vereinigung einer Menge und eines Singeltons (einelementige Menge)
- Eine nichtleere Menge ist entweder ein Singelton oder die Vereinigung zweier nichtleerer Mengen
- Ein binärer Baum ist entweder leer oder ein Knoten mit Zeigern auf zwei binäre Bäume (Kinder)
- Ein trianguliertes Polygon ist entweder leer oder bestehend aus einem Dreieck "glued" an ein trianguliertes Polygon
- Ein trianguliertes Polygon ist entweder leer oder bestehend aus einem Dreieck zwischen zwei triangulierten Polygonen
- Ein String balancierter Klammern ist entweder leer oder die Konkatenation zweier Strings balancierter Klammern oder ein String balancierter Klammern innerhalb eines Klammernpaares

Zusätzliche Links (vgl. Website)

The Canadian Airline Problem: viele zusätzliche Beispiele, ausgeführt und mit C++ - Code
http://ctp.di.fct.unl.pt/~pg/docs/canadian.htm
(http://goo.gl/HdJy3)

Für ein paar Tricks (z.B. Bit-Masks für DP über Mengen) und Beispiele siehe

- http://www.ugrad.cs.ubc.ca/~cs490/sec202/notes/dp/DP%202.pdf (http://goo.gl/ebGnD)
- http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/06-sparsedynprog.pdf (http://goo.gl/DcBGF)



http://xkcd.com/287/

Brain Teaser: Pirates!

Fünf Piraten haben eine Kiste von 100 Goldstücken ergattert. Da sie eine Horde demokratischer Piraten sind (lange lebe der Piratenkodex!) einigten sie sich auf folgendes Vorgehen um die Beute zu teilen: Der älteste Pirat schlägt eine Verteilung der Goldstücke vor. Alle Piraten (auch der älteste selbst) können dann abstimmen. Wenn ein absolutes Mehr (3 Piraten in diesem Fall) den Vorschlag annimmt wird das Gold entsprechend verteilt. Wenn nicht, wird der Pirat den Haien zum Frass vorgeworfen und der nächstälteste Pirat ist an der Reihe. Das ganze wird wiederholt solange bis ein Vorschlag angenommen wird.

Nehme an dass alle Piraten absolut rational handeln: Sie wollen unbedingt am Leben bleiben und als zweitens so viel Gold wie möglich für sich beanspruchen können. Und, blutrünstiges Pack wie sie sind, wollen sie weniger Piraten auf dem Schiff wenn sie zwischen zwei für sie gleichen Vorgschlägen auswählen können.

Wie wird die Beute schlussendlich geteilt?

Beispiel

Gegeben sei ein Dreieck mit (nichtnegativen) Zahlen. Z.B.

8
4 9
9 2 1
3 8 5 5
5 6 3 7 6

Berechne den Pfad von der Spitze bis zum Boden eines Dreiecks, wobei man in jedem Schritt eine Zeile tiefer und dabei ein Feld nach links oder rechts gehen kann, so dass die totale Summe der besuchten Felder maximal ist.

Z.B. hat das Dreieck

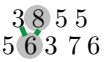
8
4 9
9 2 1
3 8 5 5
5 6 3 7 6

eine optimalen Pfadlänge von 35 mit eingezeichnetem Pfad.

1. Struktur

Denken wir darüber nach, was eine optimale Lösung charakterisiert. In jedem Schritt gehen wir eine Zeile tiefer und dabei eins nach links oder rechts. Betrachten wir nun den letzten Schritt in einer optimalen Lösung.

Das Feld mit der 6 kann erreicht werdem, in dem man von der 8 oder der 3 her im vorletzten Schritt gekommen ist.



Der optimale Pfad kommt von der 8 her. Was wissen wir über den Pfad von der Spitze des Dreicks bis zu der 8? Dieser Pfad muss einer sein mit der höchsten Summe aller möglichen Pfade von der Spitze des Dreiecks bis zu der 8. Wieso? Cut & Paste-Argument: Wäre dem nicht so, so gäbe es einen Pfad von der Spitze bis zu der 8 der eine höhere Summe hat. Dann könnten wir aber mit diesem Pfad einen insgesamt besseren Pfad bis zur 6 finden, indem man einfach diesen besseren Pfad von der Spitze des Dreicks bis zur 8 geht, und von der 8 dann im letzten Schritt zur 6.

Dies ist aber ein Widerspruch, da wir von einer optimalen Lösung ausgegangen sind.

d.h. das Problem hat eine *optimale Substruktur*, eines der Indizien für Dynamisches Programmieren.

2. Rekursion

Versuchen wir das Problem rekursiv zu formulieren.

Sei $opt(i, j)$ die maximale Summe eines Pfades von der Spitze des Dreiecks bis zum Feld (i, j), dies ist das j-te Feld in der i-ten Zeile. Z.B. ist die 9 in der zweiten Zeile das Feld $(2, 2)$

Die Lösung wäre dann also das Maximum der Felder der letzten Zeile, also $\max_{i=1,\ldots,N} opt(i, N)$

Wie kann $opt(i, j)$ berechnet werden? Dazu überlegen wir uns, wie kann ich auf das Feld (i, j) kommen? Entweder von dem Feld rechts $(i-1, j)$ oder links $(i-1, j-1)$ in der Zeile darüber. Wir wählen den besseren Weg, also:

$opt(i, j) = T[i][j] + \max\{opt(i-1, j), opt(i-1, j-1)\}$, wobei $T[i][j]$ die Zahl angibt, die im Feld (i, j) steht.

Um den optimalen Pfad von der Spitze des Dreiecks bis zu einem Feld (i, j) zu berechnen, bestimme wir also die optimalen Pfade bis zu den zwei Feldern direkt darüber, und wählen dann den besseren davon.

die Verankerung: $opt(1, 1) = T[1][1]$

Man könnte dies nun so rekursiv implementieren. Dann würden aber die selben Teilprobleme mehrmals (und zwar sehr oft) gelöst werden (Wieso? *think about it!*), deshalb speichern wir die Lösung von bisher gelösten Teilproblemen ab um einen effizieten Algorithmus zu erhalten.

3. Bottom-Up

Für die Berechnung verwende ein Array $opt[][]$ und berechne iterativ die Lösung. Man muss noch den Sonderfall betrachten, wenn man am Rand des Dreiecks ist, dann kann man dieses Feld nur über das direkt darüber erreichen.

TriangleSolver(T)
// Berechne den Wert des Pfades mit maximaler Summe im Zahlen-Dreieck T

```
1  opt[1][1] = T[i][j]
2  for i = 2 to height[T]
3        for j = 1 to i
4
5              if j = 1
6                    opt[i][j] = T[i][j] + opt[i − 1][j]
7              elseif j = i
8                    opt[i][j] = T[i][j] + opt[i − 1][j − 1]
9              else opt[i][j] = T[i][j] + max(opt[i − 1, j], opt[i − 1, j − 1])
```

Die Laufzeit ist $\mathcal{O}(n^2)$, wobei n die Anzahl der Zeilen des Dreiecks ist.

4. Lösung berechnen

An diesem Punkt haben wir die Länge des optimalen Pfades, wir wollen aber unter Umständen auch einen solchen Pfad berechnen können. Dazu finden wir zuerst das Ende des optimalen Pfades (wie oben hergeleitet ist es einfach das Maximum der Felder der letzten Zeile) und schauen dann von welchem Feld her wir diesen Wert erreicht haben. Das ist einfach: es ist genau das grössere der beiden Felder der Argumente von max aus Zeile 9.

TriangleSolverPath(T)

// Berechnung eines optimalen Pfades, im Zahlen-Dreieck T

```
1   path[height[T]] = arg max_{i=1,...,height[T]} {opt[height[T]][i]}
2   for i = height[T] to 2
3
4       if path[i] = 1
5           path[i − 1] = 1
6       elseif path[i] = i
7           path[i − 1] = i − 1
8       else
9           if opt[i][path[i]] = T[i][path[i]] + opt[i − 1, path[i] − 1]
10              path[i − 1] = path[i] − 1 // Wir kamen von links
11          else path[i − 1] = path[i] // Wir kamen von rechts
```

Die Laufzeit zum rekonstruieren des Pfades ist $O(n)$ (*again, think about it!*), wobei n die Anzahl der Zeilen des Dreiecks ist.

Alternativ kann auch während des Lösens in einem separaten Array die getroffene Wahl explizit abgespeichert werden, so dass keine Rückverfolgung im ausgefüllten Tableau notwendig ist.

Im Folgenden sind ein paar zusätzliche Beispiele von Dynamischer Programmierung, nicht vollständig ausformuliert.

Crazy Eight

Gegeben ist eine Hand Pokerkarten: $c[0], \ldots, c[n-1]$. Beispielsweise 7♡, 6♡, 7♢, 3♢, 8♣, J♠

Finde den längsten links-nach-rechts "Trick" (subsequence): $c[i_1], c[i_2], \ldots, c[i_k]$ ($i_1 < i_2 < \ldots < i_k$), wobei jeweils zwei aufeinanderfolgende Karten $c[i_j], c[i_{j+1}]$ *matchen* müssen: entweder

- haben gleiche Farbe
- oder haben gleichen Zahlenwert
- oder eine der Karten hat den Wert 8

Dies ist fast das Longest-Increasing-Subsequence Problem.

Rekursive Formulierung:

- $trick[i]$ = Länge des besten Tricks endend mit der Karte $c[i]$
- $trick[0] = 1$
- $trick[i] = 1 + \max\{trick[j] \text{ für } j \in range(0, i-1) \text{ falls } match(c[i], c[j])\}$
- $best = \max_i trick[i]$

DP: $trick[i]$ hängt nur von $trick[< i]$ ab.

Laufzeit: #Teilprobleme · Zeit/Teilproblem = $\mathcal{O}(n^2)$

Folding

siehe http://www.spoj.pl/problems/FOLD/ (http://goo.gl/Hqf4J)

Gegeben eine "Berg-Tal" Sequenz die die Falzlinien angeben, falte das Blatt zusammen mit möglichst wenig Faltungen.

1. Teilproblem: Teilstring des Blattes (i, j)
2. Iteriere über mögliche Faltposition für ersten Falt
3. Check mögliche Faltposition: *links muss Gegenteil von rechter Seite sein*

   Nach dem Falt ist längere Seite "unverändert", Rekursion auf längerem Teil
   + 1 für ersten Falt
   Nehme Minimum
4. Verankerung: keine Falten $\rightsquigarrow 0$

Take The Coins

Gegeben ist eine Reihe mit einer geraden Anzahl n von Münzen

mit Werten $v[1], \ldots, v[n]$. Wir spielen folgendes Spiel gegen Charles Deville Wells: die 2 Spieler wählen abwechselnd die *erste* oder *letzte* Münze der verbliebenen Reihe. Wird eine Münze gewählt, wird sie entfernt und der Spieler kriegt den entsprechenden Wert.

Bestimme den maximalen Wert den wir auf sicher erziehlen können wenn wir den *ersten Zug* machen und unter der Annahme, dass Charles optimal spielt. d.h. maximiere den Gesamtwert der Münzen die wir kriegen.

Aufgabe: Finde ein Gegenbeispiel bei dem ein *Greedy*-Ansatz, bei dem in jedem Zug (kurzsichtigerweise) jeweils die Münze mit dem grösseren Wert genommen wird, fehlschlägt.

Lösungsskizze: Es sei $V(i, j)$ = maximaler Wert der (garantiert) erziehlt werden kann, wenn wir an der Reihe sind und nur die Reihe $v[i] \ldots v[j]$ übrig ist.

Die Werte $V(i,i)$ und $V(i,i+1)$ können einfach berechnet werden:

$$V(i,i) = v[i], V(i,i+1) = \max\{v[i], v[i+1]\}$$

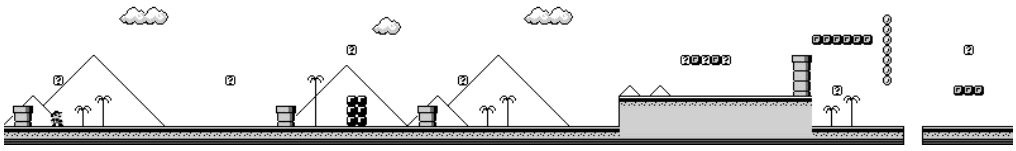
Berechne danach $V(i,i+2)$, $V(i,i+3)\ldots$ auf folgende Art: wenn wir an der Reihe sind, können wir entweder die erste oder letzte nehmen. Nach unserem Zug wird Charles, da er optimal spielt, diejenige Münze nehmen die unseren Restgewinn minimieren wird. Danach sind zwei Münzen weniger im Spiel und wir sind wieder an der Reihe und wir wissen den maximalen Wert für diese Restreihe, da wir die Werte in aufsteigender Länge berechnen.

Deshalb (think about it):

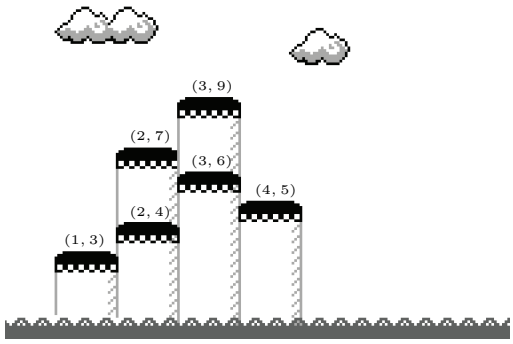
$$V(i,j) = \max\left\{\underbrace{\min\{V(i+1,j-1), V(i+2,j)\}}_{\text{nehme erste Münze } (v[i])}, \underbrace{\min\{V(i,j-2), V(i+1,j-1)\}}_{\text{nehme letzte Münze } (v[j])}\right\}$$

Laufzeit: wir haben n^2 Teilprobleme und für jedes einen konstanten Aufwand, daher $\mathcal{O}(n^2)$.

Mario



Betrachte die folgende vereinfachte Variante eines Jump and Run Spieles.



Gegeben sind (vgl. obige Abbildung):

- P Plattformen, bei $(x_i, y_i) \in \mathbb{N}^2$ (wir betrachten also nur Integer Positionen)
- Jede Plattform hat die Länge 1

- Ziel: Startend von Plattform 1, Mario will die letzte Plattform P erreichen
- "Pure pwnage"
  - Nur Bewegungen nach rechts sind möglich
  - Erreiche Ziel mit *minimaler Anzahl Moves*

Gültige Moves von (x, y) aus:

- walk: $(x + 1, y)$
- jump: $(x + 1, y + 1)$ oder $(x + 1, y + 2)$
- super-jump: $(x + 1, y + 3)$ oder $(x + 1, y + 4)$
- fall: $(x + d, y - d - d')$ solange $d + d' < 5$

Problem: Minimale Anzahl Moves um von Plattform 1 zu Plattform P zu gelangen.

Optimal Sub-Structure: Betrachte eine optimale Lösung und betrachte den letzten Schritt: Mario ist auf Plattform Q bevor er die letzte Plattform P erreicht. Dann muss die optimale Lösung mit der minimalen Anzahl Moves von Plattform 1 zu Plattform Q gelangen (übliches Cut-&-Paste Argument).

Es sei

- $d[p]$ = minimale Anzahl von Moves um von Plattform 1 bis zu Plattform p zu gelangen.
- $prev[p]$ = Vorgänger Plattform für p
- $move[p]$ = Move von $prev[p]$ nach p (walk, jump, super-jump, fall)

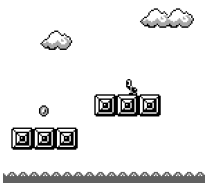
Bottom-up Lösung: Sortiere die Plattformen nach x-Koordinate, dann (da man nur nach rechts sich bewegen kann) hängt $dp[p]$ nur von Teillösungen $dp[p']$ ab, mit $p' < p$.

Laufzeit:

- Sortieren $\mathcal{O}(P \log P)$
- P Teilprobleme, für jedes Teilproblem: betrachte alle möglichen Vorgänger ($\mathcal{O}(P)$)

Die totale Laufzeit ist deshalb $\mathcal{O}(P^2)$.

Aufgabe: Mögliche Erweiterung: Es gibt nun auch Felder mit Feinden, auf denen man ein Leben verliert. Auf anderen Feldern kriegt man wieder ein Leben. Finde kürzesten Weg, so dass man nie 0 Leben hat. (Mario hat zu Beginn 1 Leben).



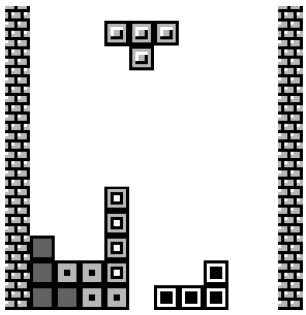
Aufgabe: Überlege, wie man das Problem auch als Graphen-Problem modellieren kann. Nun kann man auch Rückwärtsbewegungen erlauben (Wieso war es für die DP-Lösung wichtig, dass man sich nur in eine Richtung bewegen kann?). Welcher Algorithmus braucht man nun? Laufzeit?

Tetris Pwnage

Betrachte folgende Tetris Variante:

- Jedes Teil
  1. Bestimme (zu Beginn) Rotation und horizontale Position
  2. Lasse das Teil fallen

  man kann also nicht ein Teil noch rotieren oder bewegen während des Fallens.
- Vollständige Zeilen verschwinden nicht
- Ziel: Überlebe so lange wie möglich, d.h. versuche so viele Teile wie möglich unterzubringen

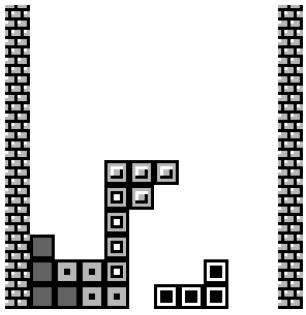


Set Up

- Spielfeld der Breite N und Höhe H
- K Teile, jedes mit eigener Form
- versuche so viele Teile wie möglich unterzubringen
- Für jedes Teil: gebe Rotation und horizontale Position von wo es fallen gelassen wird

Lösung: Ziehe folgende Beobachtung nach: Ein Spielzustand (Konfiguration) kann beschrieben werden durch die Anzahl Teile auf dem Feld plus die "Skyline": Teile können nicht durch andere Teile gehen, deshalb spielt es keine Rolle was "unter" der "Skyline" ist.

Für folgendes Beispiel ist die Konfiguration gegeben durch: 5 Spielteile, Skyline: $(3, 2, 2, 6, 6, 6, 1, 2, 0, 0)$



Idee: Konfigurationen von P Teilen hängen nur von Konfigurationen von $P-1$ Teilen ab. Es sei $d[p][skyline] = 1$ wenn mit p Teilen die gegebene Skyline erreicht werden kann.