Dynamische Programmierung - wettkampforientiert

Armin Krupp

11. Mai 2010

Inhaltsverzeichnis

Basics Dynamische Programmierung

- Richard Bellmann.
- *Programmierung* → Tabelle.

Basics Dynamische Programmierung

- Richard Bellmann.
- *Programmierung* $\rightarrow$ Tabelle.
- Optimale Teillösungen.

Basics Dynamische Programmierung

- Richard Bellmann.
- *Programmierung* $\rightarrow$ Tabelle.
- Optimale Teillösungen.
- Zwischenspeicherung.

Basics Dynamische Programmierung

- Richard Bellmann.
- *Programmierung* $\rightarrow$ Tabelle.
- Optimale Teillösungen.
- Zwischenspeicherung.
- Laufzeitvorteil.

Basics Dynamische Programmierung

- Richard Bellmann.
- *Programmierung* $\rightarrow$ Tabelle.
- Optimale Teillösungen.
- Zwischenspeicherung.
- Laufzeitvorteil.

Ein klassisches DP-Problem: Rucksack

Rucksackproblem

- Rucksack R mit Kapazität C_R.
- n Gegegenstände.

Ein klassisches DP-Problem: Rucksack

Rucksackproblem

- Rucksack R mit Kapazität C_R.
- n Gegegenstände.
- i-ter Gegenstand hat Größe c_i und Wert v_i.

Ein klassisches DP-Problem: Rucksack

Rucksackproblem

- Rucksack R mit Kapazität C_R.
- n Gegegenstände.
- i-ter Gegenstand hat Größe c_i und Wert v_i.

Problem

Welche Gegenstände sollen eingepackt werden, um den Wert zu maximieren?

Ein klassisches DP-Problem: Rucksack

Rucksackproblem

- Rucksack R mit Kapazität C_R.
- n Gegegenstände.
- i-ter Gegenstand hat Größe c_i und Wert v_i.

Problem

Welche Gegenstände sollen eingepackt werden, um den Wert zu maximieren?

Ein paar Notationen

- Feste Ordnung.
- $val(i, k)$

Ein paar Notationen

- Feste Ordnung.
- $val(i, k)$

Die Struktur der optimalen Lösung

Schritt 1: Optimale Substruktur

Zeige, dass die optimale Lösung aus optimalen Teillösungen besteht!

Die Struktur der optimalen Lösung

Schritt 1: Optimale Substruktur

Zeige, dass die optimale Lösung aus optimalen Teillösungen besteht!

- Angenommen, wir haben eine optimale Rucksackbepackung.

Die Struktur der optimalen Lösung

Schritt 1: Optimale Substruktur

Zeige, dass die optimale Lösung aus optimalen Teillösungen besteht!

- Angenommen, wir haben eine optimale Rucksackbepackung.
- Dann muss die Rucksackbepackung für den Rucksack **ohne** das oberste Element und mit entsprechender reduzierter Kapazität auch optimal sein.

Die Struktur der optimalen Lösung

Schritt 1: Optimale Substruktur

Zeige, dass die optimale Lösung aus optimalen Teillösungen besteht!

- Angenommen, wir haben eine optimale Rucksackbepackung.
- Dann muss die Rucksackbepackung für den Rucksack **ohne** das oberste Element und mit entsprechender reduzierter Kapazität auch optimal sein.
- Wäre dies nicht so, dann gäbe es eine Bepackung für die reduzierte Kapazität mit einem höheren Wert.

Die Struktur der optimalen Lösung

Schritt 1: Optimale Substruktur

Zeige, dass die optimale Lösung aus optimalen Teillösungen besteht!

- Angenommen, wir haben eine optimale Rucksackbepackung.
- Dann muss die Rucksackbepackung für den Rucksack **ohne** das oberste Element und mit entsprechender reduzierter Kapazität auch optimal sein.
- Wäre dies nicht so, dann gäbe es eine Bepackung für die reduzierte Kapazität mit einem höheren Wert.
- Nach Hinzufügen des obersten Elementes hätten wir dann eine bessere Rucksackbepackung

Die Struktur der optimalen Lösung

Schritt 1: Optimale Substruktur

Zeige, dass die optimale Lösung aus optimalen Teillösungen besteht!

- Angenommen, wir haben eine optimale Rucksackbepackung.
- Dann muss die Rucksackbepackung für den Rucksack **ohne** das oberste Element und mit entsprechender reduzierter Kapazität auch optimal sein.
- Wäre dies nicht so, dann gäbe es eine Bepackung für die reduzierte Kapazität mit einem höheren Wert.
- Nach Hinzufügen des obersten Elementes hätten wir dann eine bessere Rucksackbepackung
- Widerspruch zur optimalen Rucksackbepackung!

Die Struktur der optimalen Lösung

Schritt 1: Optimale Substruktur

Zeige, dass die optimale Lösung aus optimalen Teillösungen besteht!

- Angenommen, wir haben eine optimale Rucksackbepackung.
- Dann muss die Rucksackbepackung für den Rucksack **ohne** das oberste Element und mit entsprechender reduzierter Kapazität auch optimal sein.
- Wäre dies nicht so, dann gäbe es eine Bepackung für die reduzierte Kapazität mit einem höheren Wert.
- Nach Hinzufügen des obersten Elementes hätten wir dann eine bessere Rucksackbepackung
- Widerspruch zur optimalen Rucksackbepackung!

Rekursive Definition des Wertes der optimalen Lösung

Rekursive Formulierung

Finde eine rekursive Formulierung des Optimierungsproblems!

- Wir wollen $val(i, k), i \in [n], k \in [C_R]$ berechnen.

Rekursive Definition des Wertes der optimalen Lösung

Rekursive Formulierung

Finde eine rekursive Formulierung des Optimierungsproblems!

- Wir wollen $val(i, k), i \in [n], k \in [C_R]$ berechnen.
- Wir haben nun zwei Möglichkeiten:

Rekursive Definition des Wertes der optimalen Lösung

Rekursive Formulierung

Finde eine rekursive Formulierung des Optimierungsproblems!

- Wir wollen $val(i, k), i \in [n], k \in [C_R]$ berechnen.
- Wir haben nun zwei Möglichkeiten:
- Wir packen das i-te Element in den Rucksack ein.

Rekursive Definition des Wertes der optimalen Lösung

Rekursive Formulierung

Finde eine rekursive Formulierung des Optimierungsproblems!

- Wir wollen $val(i, k), i \in [n], k \in [C_R]$ berechnen.
- Wir haben nun zwei Möglichkeiten:
- Wir packen das i-te Element in den Rucksack ein.
- Dann beträgt der Wert des Rucksacks $val(i-1, k-c_i) + v_i$

Rekursive Definition des Wertes der optimalen Lösung

Rekursive Formulierung

Finde eine rekursive Formulierung des Optimierungsproblems!

- Wir wollen $val(i, k), i \in [n], k \in [C_R]$ berechnen.
- Wir haben nun zwei Möglichkeiten:
- Wir packen das i-te Element in den Rucksack ein.
- Dann beträgt der Wert des Rucksacks $val(i-1, k-c_i) + v_i$
- Oder wir packen es nicht ein.

Rekursive Definition des Wertes der optimalen Lösung

Rekursive Formulierung

Finde eine rekursive Formulierung des Optimierungsproblems!

- Wir wollen $val(i, k), i \in [n], k \in [C_R]$ berechnen.
- Wir haben nun zwei Möglichkeiten:
- Wir packen das i-te Element in den Rucksack ein.
- Dann beträgt der Wert des Rucksacks $val(i-1, k-c_i) + v_i$
- Oder wir packen es nicht ein.
- Dann beträgt der Wert des Rucksacks $val(i-1, k)$.

Rekursive Definition des Wertes der optimalen Lösung

Rekursive Formulierung

Finde eine rekursive Formulierung des Optimierungsproblems!

- Wir wollen $val(i, k), i \in [n], k \in [C_R]$ berechnen.
- Wir haben nun zwei Möglichkeiten:
- Wir packen das i-te Element in den Rucksack ein.
- Dann beträgt der Wert des Rucksacks $val(i-1, k-c_i) + v_i$
- Oder wir packen es nicht ein.
- Dann beträgt der Wert des Rucksacks $val(i-1, k)$.
- Optimierungsproblem $\longrightarrow$ wähle $max(val(i-1, k-c_i) + v_i, val(i-1, k))$

Rekursive Definition des Wertes der optimalen Lösung

Rekursive Formulierung

Finde eine rekursive Formulierung des Optimierungsproblems!

- Wir wollen $val(i, k), i \in [n], k \in [C_R]$ berechnen.
- Wir haben nun zwei Möglichkeiten:
- Wir packen das i-te Element in den Rucksack ein.
- Dann beträgt der Wert des Rucksacks $val(i-1, k-c_i) + v_i$
- Oder wir packen es nicht ein.
- Dann beträgt der Wert des Rucksacks $val(i-1, k)$.
- Optimierungsproblem $\longrightarrow$ wähle
  $max(val(i-1, k-c_i) + v_i, val(i-1, k))$

Berechnung der optimalen Lösung

C++ Code:

Analyse

Laufzeit

Pro Rekusionsschritt werden zwei neue Funtionsaufrufe getätigt!
Laufzeit: $O(2^n)$

Der DP-Ansatz

- $val(i-1,k)$ wird bei $val(i,k), val(i+1,k), \ldots val(n,k)$ (indirekt) aufgerufen!
- Wenn wir den Wert von $val(i-1,k)$ kennen, können wir uns zeitaufwendige Rechnungen ersparen.

Der DP-Ansatz

- $val(i-1,k)$ wird bei $val(i,k), val(i+1,k), \ldots val(n,k)$ (indirekt) aufgerufen!
- Wenn wir den Wert von $val(i-1,k)$ kennen, können wir uns zeitaufwendige Rechnungen ersparen.
- Top-Down $\rightarrow$ *Memoization*

Der DP-Ansatz

- $val(i-1,k)$ wird bei $val(i,k), val(i+1,k), \ldots val(n,k)$ (indirekt) aufgerufen!
- Wenn wir den Wert von $val(i-1,k)$ kennen, können wir uns zeitaufwendige Rechnungen ersparen.
- Top-Down $\rightarrow$ *Memoization*
- Bottom-Up$\rightarrow$ *Tabelle*

Der DP-Ansatz

- $val(i-1,k)$ wird bei $val(i,k), val(i+1,k), \ldots val(n,k)$ (indirekt) aufgerufen!
- Wenn wir den Wert von $val(i-1,k)$ kennen, können wir uns zeitaufwendige Rechnungen ersparen.
- Top-Down $\rightarrow$ *Memoization*
- Bottom-Up$\rightarrow$ *Tabelle*

Der DP-Ansatz

- $val(i-1,k)$ wird bei $val(i,k), val(i+1,k), \ldots val(n,k)$ (indirekt) aufgerufen!
- Wenn wir den Wert von $val(i-1,k)$ kennen, können wir uns zeitaufwendige Rechnungen ersparen.
- Top-Down $\rightarrow$ *Memoization*
- Bottom-Up$\rightarrow$ *Tabelle*

Lösung des Problems - mit DP

Berechnung der Lösung

Löse das Problem **effizient**, indem die Zwischenlösungen in einer entsprechenden Datenstruktur abgespeichert werden.

- Wir haben zwei "Laufvariablen", i und k.

Lösung des Problems - mit DP

Berechnung der Lösung

Löse das Problem **effizient**, indem die Zwischenlösungen in einer entsprechenden Datenstruktur abgespeichert werden.

- Wir haben zwei "Laufvariablen", i und k.
- Erstelle eine $n \times C_R$ Tabelle, der (i, j)-Eintrag enthält $val(i, j)$.

Lösung des Problems - mit DP

Berechnung der Lösung

Löse das Problem **effizient**, indem die Zwischenlösungen in einer entsprechenden Datenstruktur abgespeichert werden.

- Wir haben zwei "Laufvariablen", i und k.
- Erstelle eine $n \times C_R$ Tabelle, der (i, j)-Eintrag enthält $val(i, j)$.
- Oder: Erstelle eine *map*, in der die $val(i, j)$ memoisiert werden.

Lösung des Problems - mit DP

Berechnung der Lösung

Löse das Problem **effizient**, indem die Zwischenlösungen in einer entsprechenden Datenstruktur abgespeichert werden.

- Wir haben zwei "Laufvariablen", i und k.
- Erstelle eine $n \times C_R$ Tabelle, der (i, j)-Eintrag enthält $val(i, j)$.
- Oder: Erstelle eine *map*, in der die $val(i, j)$ memoisiert werden.
- Die optimale Lösung findet sich dann in (n, C_R).

Lösung des Problems - mit DP

Berechnung der Lösung

Löse das Problem **effizient**, indem die Zwischenlösungen in einer entsprechenden Datenstruktur abgespeichert werden.

- Wir haben zwei "Laufvariablen", i und k.
- Erstelle eine $n \times C_R$ Tabelle, der (i, j)-Eintrag enthält $val(i, j)$.
- Oder: Erstelle eine *map*, in der die $val(i, j)$ memoisiert werden.
- Die optimale Lösung findet sich dann in (n, C_R).

Implementierung in C++

Optional - die Lösung

Konstruktion der Lösung

Wenn nicht nur der Wert der Lösung zählt, muss die Lösung mitgeschrieben werden.

- Rucksackbepackung ist von Interesse.

Optional - die Lösung

Konstruktion der Lösung

Wenn nicht nur der Wert der Lösung zählt, muss die Lösung mitgeschrieben werden.

- Rucksackbepackung ist von Interesse.
- Ablesen aus der Tabelle:

Optional - die Lösung

Konstruktion der Lösung

Wenn nicht nur der Wert der Lösung zählt, muss die Lösung mitgeschrieben werden.

- Rucksackbepackung ist von Interesse.
- Ablesen aus der Tabelle:
- $val(i, k) = val(i-1, k) \rightarrow i - te$ Element nicht enthalten.

Optional - die Lösung

Konstruktion der Lösung

Wenn nicht nur der Wert der Lösung zählt, muss die Lösung mitgeschrieben werden.

- Rucksackbepackung ist von Interesse.
- Ablesen aus der Tabelle:
- $val(i, k) = val(i - 1, k) \rightarrow i - te$ Element nicht enthalten.
- Separate Tabelle:

Optional - die Lösung

Konstruktion der Lösung

Wenn nicht nur der Wert der Lösung zählt, muss die Lösung mitgeschrieben werden.

- Rucksackbepackung ist von Interesse.
- Ablesen aus der Tabelle:
- $val(i, k) = val(i - 1, k) \rightarrow i - te$ Element nicht enthalten.
- Separate Tabelle:
- $tab[i][k] = k_{vorher}$

Optional - die Lösung

Konstruktion der Lösung

Wenn nicht nur der Wert der Lösung zählt, muss die Lösung mitgeschrieben werden.

- Rucksackbepackung ist von Interesse.
- Ablesen aus der Tabelle:
- $val(i, k) = val(i - 1, k) \rightarrow i - te$ Element nicht enthalten.
- Separate Tabelle:
- $tab[i][k] = k_{vorher}$

Zusammenfassung der Schritte

1. Nachweis optimale Substruktur.
2. Rekursive Formulierung.

Zusammenfassung der Schritte

1. Nachweis optimale Substruktur.
2. Rekursive Formulierung.
3. Top-Down oder Bottom up.

Zusammenfassung der Schritte

1. Nachweis optimale Substruktur.
2. Rekursive Formulierung.
3. Top-Down oder Bottom up.
4. Konstruktion der Lösung

Zusammenfassung der Schritte

1. Nachweis optimale Substruktur.
2. Rekursive Formulierung.
3. Top-Down oder Bottom up.
4. Konstruktion der Lösung

Checkliste: Wann DP angewenden?

- Optimierungsproblem oder Aufzählproblem.
- Probleminstanzen sind ganzzahlig.

Checkliste: Wann DP angewenden?

- Optimierungsproblem oder Aufzählproblem.
- Probleminstanzen sind ganzzahlig.
- Laufzeit der naiven Lösung überschlagen!

Checkliste: Wann DP angewenden?

- Optimierungsproblem oder Aufzählproblem.
- Probleminstanzen sind ganzzahlig.
- Laufzeit der naiven Lösung überschlagen!
- Eingabegröße:

Checkliste: Wann DP angewenden?

- Optimierungsproblem oder Aufzählproblem.
- Probleminstanzen sind ganzzahlig.
- Laufzeit der naiven Lösung überschlagen!
- Eingabegröße:
  1. $O(n) \longrightarrow$ Wertebereich!

Checkliste: Wann DP angewenden?

- Optimierungsproblem oder Aufzählproblem.
- Probleminstanzen sind ganzzahlig.
- Laufzeit der naiven Lösung überschlagen!
- Eingabegröße:
  1. $O(n) \rightarrow$ Wertebereich!
  2. $O(n^2) \rightarrow \approx 1000$

Checkliste: Wann DP angewenden?

- Optimierungsproblem oder Aufzählproblem.
- Probleminstanzen sind ganzzahlig.
- Laufzeit der naiven Lösung überschlagen!
- Eingabegröße:
  1. $O(n) \rightarrow$ Wertebereich!
  2. $O(n^2) \rightarrow \approx 1000$
  3. $O(n^3) \rightarrow \approx 100$

Checkliste: Wann DP angewenden?

- Optimierungsproblem oder Aufzählproblem.
- Probleminstanzen sind ganzzahlig.
- Laufzeit der naiven Lösung überschlagen!
- Eingabegröße:
  1. $O(n) \rightarrow$ Wertebereich!
  2. $O(n^2) \rightarrow \approx 1000$
  3. $O(n^3) \rightarrow \approx 100$

Überblick

- In einigen UVA/ACM Aufgaben sind klassische DP-Probleme “versteckt”.
- Problem erkannt, Problem gebannt.

Überblick

- In einigen UVA/ACM Aufgaben sind klassische DP-Probleme “versteckt”.
- Problem erkannt, Problem gebannt.

Klassische DP-Probleme

1. Rucksack
2. Levenshtein/Edit - Distance

Klassische DP-Probleme

1. Rucksack
2. Levenshtein/Edit - Distance
3. Longest Common Subsequence

Klassische DP-Probleme

1. Rucksack
2. Levenshtein/Edit - Distance
3. Longest Common Subsequence
4. Cocke - Younger - Kasami (CYK)

Klassische DP-Probleme

1. Rucksack
2. Levenshtein/Edit - Distance
3. Longest Common Subsequence
4. Cocke - Younger - Kasami (CYK)

Edit/Levenshtein - Distance

- Gegeben seien zwei Zeichenketten $P = p_1 \dots p_n$ und $T = t_1 \dots t_m$.
- Wir können folgende Operationen auf diesen Ketten durchführen:

Edit/Levenshtein - Distance

- Gegeben seien zwei Zeichenketten $P = p_1 \dots p_n$ und $T = t_1 \dots t_m$.
- Wir können folgende Operationen auf diesen Ketten durchführen:
  1. Ersetzen: *meer* $\longrightarrow$ *mehr*
  2. Einfügen: *dof* $\longrightarrow$ *doof*
  3. Löschen: *interressant* $\longrightarrow$ *interessant*

Edit/Levenshtein - Distance

- Gegeben seien zwei Zeichenketten $P = p_1 \dots p_n$ und $T = t_1 \dots t_m$.
- Wir können folgende Operationen auf diesen Ketten durchführen:
  1. Ersetzen: *meer* $\longrightarrow$ *mehr*
  2. Einfügen: *dof* $\longrightarrow$ *doof*
  3. Löschen: *interressant* $\longrightarrow$ *interessant*
- Alle Operationen sind gleich teuer.

Edit/Levenshtein - Distance

- Gegeben seien zwei Zeichenketten $P = p_1 \dots p_n$ und $T = t_1 \dots t_m$.
- Wir können folgende Operationen auf diesen Ketten durchführen:
  1. Ersetzen: *meer* $\longrightarrow$ *mehr*
  2. Einfügen: *dof* $\longrightarrow$ *doof*
  3. Löschen: *interressant* $\longrightarrow$ *interessant*
- Alle Operationen sind gleich teuer.
- Die Levenshtein-Distance bezeichnet die minimale Anzahl an Operationen, um P in T umzuwandeln.

Edit/Levenshtein - Distance

- Gegeben seien zwei Zeichenketten $P = p_1 \dots p_n$ und $T = t_1 \dots t_m$.
- Wir können folgende Operationen auf diesen Ketten durchführen:
  1. Ersetzen: *meer* $\longrightarrow$ *mehr*
  2. Einfügen: *dof* $\longrightarrow$ *doof*
  3. Löschen: *interressant* $\longrightarrow$ *interessant*
- Alle Operationen sind gleich teuer.
- Die Levenshtein-Distance bezeichnet die minimale Anzahl an Operationen, um P in T umzuwandeln.
- Aufgabe: Berechne die Levenshtein-Distance!

Edit/Levenshtein - Distance

- Gegeben seien zwei Zeichenketten $P = p_1 \dots p_n$ und $T = t_1 \dots t_m$.
- Wir können folgende Operationen auf diesen Ketten durchführen:
  1. Ersetzen: *meer* $\longrightarrow$ *mehr*
  2. Einfügen: *dof* $\longrightarrow$ *doof*
  3. Löschen: *interressant* $\longrightarrow$ *interessant*
- Alle Operationen sind gleich teuer.
- Die Levenshtein-Distance bezeichnet die minimale Anzahl an Operationen, um P in T umzuwandeln.
- Aufgabe: Berechne die Levenshtein-Distance!

Berechnung der Levenshtein - Distance

- Klar: Optimierungsproblem
- Bezeichne mit $LD(i, j)$ die minimale Anzahl an Operationen, um $p_1 \dots p_i$ in $t_1 \dots t_j$ umzuwandeln.

Berechnung der Levenshtein - Distance

- Klar: Optimierungsproblem
- Bezeichne mit $LD(i, j)$ die minimale Anzahl an Operationen, um $p_1 \dots p_i$ in $t_1 \dots t_j$ umzuwandeln.
- **Optimale Substruktur und rekursive Formulierung**
- Angenommen wir wollen $LD(i, j)$ berechnen:

Berechnung der Levenshtein - Distance

- Klar: Optimierungsproblem
- Bezeichne mit $LD(i,j)$ die minimale Anzahl an Operationen, um $p_1 \dots p_i$ in $t_1 \dots t_j$ umzuwandeln.
- **Optimale Substruktur und rekursive Formulierung**
- Angenommen wir wollen $LD(i,j)$ berechnen:
  1. Übereinstimmung oder Ersetzen: $LD(i,j) = LD(i-1,j-1)$ oder $LD(i,j) = LD(i-1,j-1)+1$

Berechnung der Levenshtein - Distance

- Klar: Optimierungsproblem
- Bezeichne mit $LD(i,j)$ die minimale Anzahl an Operationen, um $p_1 \dots p_i$ in $t_1 \dots t_j$ umzuwandeln.
- **Optimale Substruktur und rekursive Formulierung**
- Angenommen wir wollen $LD(i,j)$ berechnen:
  1. Übereinstimmung oder Ersetzen: $LD(i,j) = LD(i-1,j-1)$ oder $LD(i,j) = LD(i-1,j-1)+1$
  2. Einfügen: $LD(i,j) = LD(i-1,j)+1$

Berechnung der Levenshtein - Distance

- Klar: Optimierungsproblem
- Bezeichne mit $LD(i,j)$ die minimale Anzahl an Operationen, um $p_1 \dots p_i$ in $t_1 \dots t_j$ umzuwandeln.
- **Optimale Substruktur und rekursive Formulierung**
- Angenommen wir wollen $LD(i,j)$ berechnen:
  1. Übereinstimmung oder Ersetzen: $LD(i,j) = LD(i-1,j-1)$ oder $LD(i,j) = LD(i-1,j-1)+1$
  2. Einfügen: $LD(i,j) = LD(i-1,j)+1$
  3. Löschen: $LD(i,j) = LD(i,j-1)+1$

Berechnung der Levenshtein - Distance

- Klar: Optimierungsproblem
- Bezeichne mit $LD(i,j)$ die minimale Anzahl an Operationen, um $p_1 \dots p_i$ in $t_1 \dots t_j$ umzuwandeln.
- **Optimale Substruktur und rekursive Formulierung**
- Angenommen wir wollen $LD(i,j)$ berechnen:
  1. Übereinstimmung oder Ersetzen: $LD(i,j) = LD(i-1,j-1)$ oder $LD(i,j) = LD(i-1,j-1)+1$
  2. Einfügen: $LD(i,j) = LD(i-1,j)+1$
  3. Löschen: $LD(i,j) = LD(i,j-1)+1$
- Mit diesem Wissen lässt sich nun eine DP Formulierung finden.

Berechnung der Levenshtein - Distance

- Klar: Optimierungsproblem
- Bezeichne mit $LD(i,j)$ die minimale Anzahl an Operationen, um $p_1 \dots p_i$ in $t_1 \dots t_j$ umzuwandeln.
- **Optimale Substruktur und rekursive Formulierung**
- Angenommen wir wollen $LD(i,j)$ berechnen:
  1. Übereinstimmung oder Ersetzen: $LD(i,j) = LD(i-1,j-1)$ oder $LD(i,j) = LD(i-1,j-1)+1$
  2. Einfügen: $LD(i,j) = LD(i-1,j)+1$
  3. Löschen: $LD(i,j) = LD(i,j-1)+1$
- Mit diesem Wissen lässt sich nun eine DP Formulierung finden.

Beispiel

		A	C	M
	0	1	2	3
I	1			
C	2			
P	3			
C	4			

Beispiel

		A	C	M
	0	1	2	3
I	1	1	2	3
C	2	2		

Beispiel

		A	C	M
	0	1	2	3
I	1	1	2	3
C	2	2	1	

Beispiel

		A	C	M
	0	1	2	3
I	1	1	2	3
C	2	2	1	2

Beispiel

		A	C	M
	0	1	2	3
I	1	1	2	3
C	2	2	1	2
P	3	3	2	2
C	4	4	3	3

Beispiel

		A	C	M
	0	1	2	3
I	1	1	2	3
C	2	2	1	2
P	3	3	2	2
C	4	4	3	3

Longest-Common-Subsequence

LCS - Problemstellung

Gegeben seien zwei Strings P und T, finde die längste gemeinsame Subsequenz. (Bsp. **Zitrone** und **Zisterne**)

- Dieses Problem ist der Levenshtein-Distance sehr ähnlich:

Longest-Common-Subsequence

LCS - Problemstellung

Gegeben seien zwei Strings P und T, finde die längste gemeinsame Subsequenz. (Bsp. **Zitrone** und **Zisterne**)

- Dieses Problem ist der Levenshtein-Distance sehr ähnlich:
- Wenn wir das Ersetzen nicht zulassen (sehr teuer machen), erhalten wir die minimale Anzahl an Einfüge- und Löschoperationen.

Longest-Common-Subsequence

LCS - Problemstellung

Gegeben seien zwei Strings P und T, finde die längste gemeinsame Subsequenz. (Bsp. **Zitrone** und **Zisterne**)

- Dieses Problem ist der Levenshtein-Distance sehr ähnlich:
- Wenn wir das Ersetzen nicht zulassen (sehr teuer machen), erhalten wir die minimale Anzahl an Einfüge- und Löschoperationen.
- Die längste gemeinsame Sequenz bleibt dabei erhalten!

Longest-Common-Subsequence

LCS - Problemstellung

Gegeben seien zwei Strings P und T, finde die längste gemeinsame Subsequenz. (Bsp. **Zitrone** und **Zisterne**)

- Dieses Problem ist der Levenshtein-Distance sehr ähnlich:
- Wenn wir das Ersetzen nicht zulassen (sehr teuer machen), erhalten wir die minimale Anzahl an Einfüge- und Löschoperationen.
- Die längste gemeinsame Sequenz bleibt dabei erhalten!

Cocke-Younger-Kasami Algorithmus

CYK - Algorithmus

Der Algorithmus von Casami-Younger-Kocke löst das Problem, ob ein Wort w zu einer vorgegebenen kontextfreien Sprache L gehört. Dabei muss die Gramatik G der Sprache $L(G)$ in Chomsky-Normalform vorliegen.

1. Es sei ein Wort $w = w_1 \dots w_n$ gegeben. Wenn w in $L(G)$ liegt, dann gibt es eine Folge von Produktionen, sodass $S \rightarrow AB \rightarrow \dots \rightarrow w$ gilt.

Cocke-Younger-Kasami Algorithmus

CYK - Algorithmus

Der Algorithmus von Casami-Younger-Kocke löst das Problem, ob ein Wort w zu einer vorgegebenen kontextfreien Sprache L gehört. Dabei muss die Gramatik G der Sprache $L(G)$ in Chomsky-Normalform vorliegen.

1. Es sei ein Wort $w = w_1 \dots w_n$ gegeben. Wenn w in $L(G)$ liegt, dann gibt es eine Folge von Produktionen, sodass $S \rightarrow AB \rightarrow \dots \rightarrow w$ gilt.
2. Damit existiert insbesondere ein Index $i \in [1, n-1]$, sodass $w_1 \dots w_i$ ein Produkt von A und $w_{i+1} \dots w_n$ ein Produkt von B ist.

Cocke-Younger-Kasami Algorithmus

CYK - Algorithmus

Der Algorithmus von Casami-Younger-Kocke löst das Problem, ob ein Wort w zu einer vorgegebenen kontextfreien Sprache L gehört. Dabei muss die Gramatik G der Sprache $L(G)$ in Chomsky-Normalform vorliegen.

1. Es sei ein Wort $w = w_1 \dots w_n$ gegeben. Wenn w in $L(G)$ liegt, dann gibt es eine Folge von Produktionen, sodass $S \rightarrow AB \rightarrow \dots \rightarrow w$ gilt.
2. Damit existiert insbesondere ein Index $i \in [1, n-1]$, sodass $w_1 \dots w_i$ ein Produkt von A und $w_{i+1} \dots w_n$ ein Produkt von B ist.

Fortsetzung CYK

1. Daraus lässt sich ein rekursiver Ansatz ableiten.
2. Wir führen eine boolsche Funktion $M(i, j, X)$ ein, die genau dann wahr ist, wenn sich $w_i \dots w_j$ aus X ableiten lässt.

Fortsetzung CYK

1. Daraus lässt sich ein rekursiver Ansatz ableiten.
2. Wir führen eine boolsche Funktion $M(i, j, X)$ ein, die genau dann wahr ist, wenn sich $w_i \dots w_j$ aus X ableiten lässt.
3. Um $M(i, j, X)$ zu berechnen, werten wir alle Produktionen von X aus $M(i, j, X) = \bigvee_{(X \to YZ \in G)} (\bigvee_{i=k}^{j} M(i, k, Y) \wedge M(k, j, Z))$

Fortsetzung CYK

1. Daraus lässt sich ein rekursiver Ansatz ableiten.
2. Wir führen eine boolsche Funktion $M(i, j, X)$ ein, die genau dann wahr ist, wenn sich $w_i \dots w_j$ aus X ableiten lässt.
3. Um $M(i, j, X)$ zu berechnen, werten wir alle Produktionen von X aus $M(i, j, X) = \bigvee_{(X \to YZ \in G)} (\bigvee_{i=k}^{j} M(i, k, Y) \wedge M(k, j, Z))$
4. Wenn $M(1, n, S)$ wahr ist, liegt w in $L(G)$.

Fortsetzung CYK

1. Daraus lässt sich ein rekursiver Ansatz ableiten.
2. Wir führen eine boolsche Funktion $M(i, j, X)$ ein, die genau dann wahr ist, wenn sich $w_i \dots w_j$ aus X ableiten lässt.
3. Um $M(i, j, X)$ zu berechnen, werten wir alle Produktionen von X aus $M(i, j, X) = \bigvee_{(X \to YZ \in G)} (\bigvee_{i=k}^{j} M(i, k, Y) \wedge M(k, j, Z))$
4. Wenn $M(1, n, S)$ wahr ist, liegt w in $L(G)$.

Beispiel: Self-Replicating Worms

Ein Stück Quantonium fiel auf einige Erdwürmer, die dadurch in mehrere Teile zerlegt wurden. Die einzelnen Körperteile der Würmer fingen kurz darauf an, sich zu reproduzieren, wobei sich manche Körperteile nicht mehr verändern, andere jedoch. Eine Reproduktion läuft immer nach folgendem Schema ab:
$K \rightarrow AB$.
Alle Würmer, die über eine Kette von Reproduktionen aus dem Körperteil “bowel” entstanden sind, sind böse, die anderen nicht. General W.R. Monger möchte nun wissen, ob ein gegebener Wurm böse ist oder nicht.
Aufgabe: Finde aus den gegebenen Reproduktionsmechanismen heraus, ob ein Wurm gut oder böse ist.

Fragen

Noch Fragen?

Zum Warmwerden: Welcome to Code Jam (Google Code Jam 2009)

Eingabe

Ein String mit maximal 500 Zeichen (Kleinbuchstaben und Leerzeichen)

Ausgabe

Wie oft ist der String **welcome to code jam** (mit Leerzeichen) als Subsequence vor? Dabei sollen nur die letzten 4 Ziffern der Zahl ausgegeben werden.

Beispieleingabe

Input	Output
elcomew elcome to code jam	Case 1: 0001
wweellccoommee to code qps jam	Case 2: 0256
welcome to codejam	Case 3: 0000

Aufgabe 2: Is bigger smarter (UVA 10131)

Problem

Einige Leute behaupten, je schwerer ein Elefant ist, desto klüger sei er. Um diese Theorie zu widerlegen, sollen aus einer gegebenen Liste von Elefanten (mit Gewicht und IQ) möglichst viele Elefanten gefunden werden, sodass alle Gewichte und IQs der Elefanten verschieden und der IQ eines schwereren Elefanten geringer als der eines leichteren ist.

Input

Eine Liste mit dem Gewicht und IQ des jeweiligen Elefanten.

Aufgabe 2: Is bigger smarter (UVA 10131)

Problem

Einige Leute behaupten, je schwerer ein Elefant ist, desto klüger sei er. Um diese Theorie zu widerlegen, sollen aus einer gegebenen Liste von Elefanten (mit Gewicht und IQ) möglichst viele Elefanten gefunden werden, sodass alle Gewichte und IQs der Elefanten verschieden und der IQ eines schwereren Elefanten geringer als der eines leichteren ist.

Input

Eine Liste mit dem Gewicht und IQ des jeweiligen Elefanten.

Output

Die größte Anzahl der Elefanten, die diesem Kriterium entsprechen sowie deren Indizes in der Anfangsreihenfolge.

Aufgabe 2: Is bigger smarter (UVA 10131)

Problem

Einige Leute behaupten, je schwerer ein Elefant ist, desto klüger sei er. Um diese Theorie zu widerlegen, sollen aus einer gegebenen Liste von Elefanten (mit Gewicht und IQ) möglichst viele Elefanten gefunden werden, sodass alle Gewichte und IQs der Elefanten verschieden und der IQ eines schwereren Elefanten geringer als der eines leichteren ist.

Input

Eine Liste mit dem Gewicht und IQ des jeweiligen Elefanten.

Output

Die größte Anzahl der Elefanten, die diesem Kriterium entsprechen sowie deren Indizes in der Anfangsreihenfolge.

Aufgabe 3: Ferryloading (UVA 10261)

Problem

Um über eine Fluss zu gelangen, warten n Autos in einer Warteschlange vor einer Fähre. Der Kapitän der zweispurigen Fähre teilt die Autos auf die linke oder rechte Fährseite ein. Dabei ist die Reihenfolge der Autos fest vorgegeben und die Gesamtlänge der Autoschlangen auf beiden Seiten der Fähre darf die Länge der Fähre nicht überschreiten. Aufgrund seiner Erfahrung kann der Kapitän die Längen der einzelnen Autos abschätzen.
Mit dieser Information soll nun ein Programm geschrieben werden, welches für eine gegebene Schlange von Autos die optimale Zuteilung bestimmt.

Fortsetzung zu Ferryloading

Input

Die Anzahl der Autos sowie die einzelnen Längen.

Output

Die Anzahl der Autos, die auf die Fähre passen, dazu für jedes Auto entweder ein “Left” oder “Right”, je nach dem, auf welche Seite der Fähre das Auto geschickt wird.

Fortsetzung zu Ferryloading

Input

Die Anzahl der Autos sowie die einzelnen Längen.

Output

Die Anzahl der Autos, die auf die Fähre passen, dazu für jedes Auto entweder ein “Left” oder “Right”, je nach dem, auf welche Seite der Fähre das Auto geschickt wird.

Literatur

- T. Cormen et. al.
  Introduction to Algorithms
  MIT Press 2007
- Steven S. Skiena
  The Algorithm Design Manual
  Springer 2008