Einführung in die Objektorientierte Programmierung
Vorlesung 17: Dynamische Programmierung

Sebastian Küpper

Redundanz Rekursiver Lösungen

- Rekursion kann elegante Bescheibungen zur Problemlösung ergeben
- Anwendbar (vor allem), wenn Probleme sich auf kleinere Instanzen zurückführen lassen
- Problem: Hierbei können Redundanzen entstehen.

Redundanz bei der Fibonacci-Folge

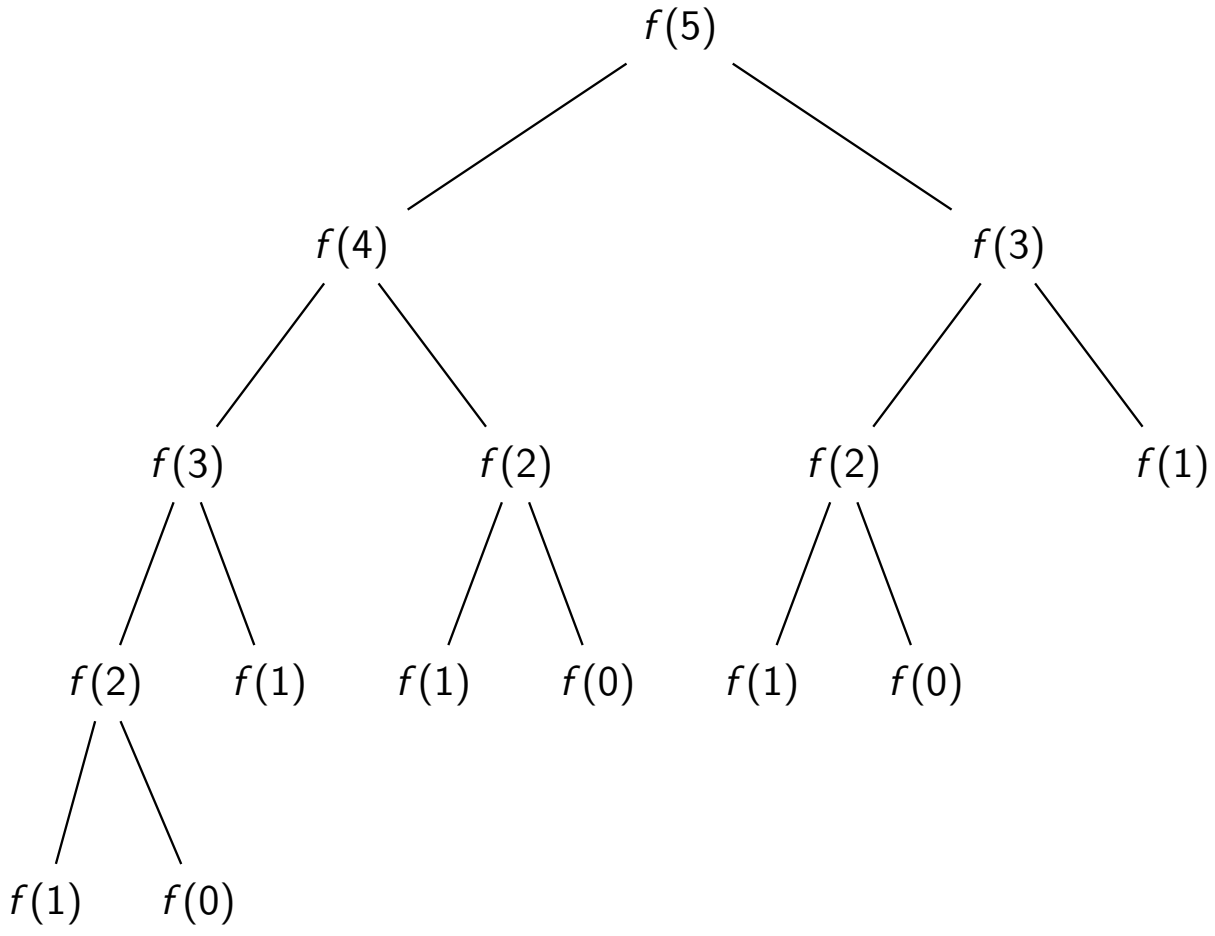
Wir betrachten die folgende rekursive Implementierung der Fibonacci-Folge[1]:

```
long f(int n) {
   if (n == 0 || n == 1) {
      return n;
   }
   return f(n - 1) + f(n - 2);
}
```

[1] zur besseren Übersicht wird auf eine Behandlung fehlerhafter Eingaben verzichtet.

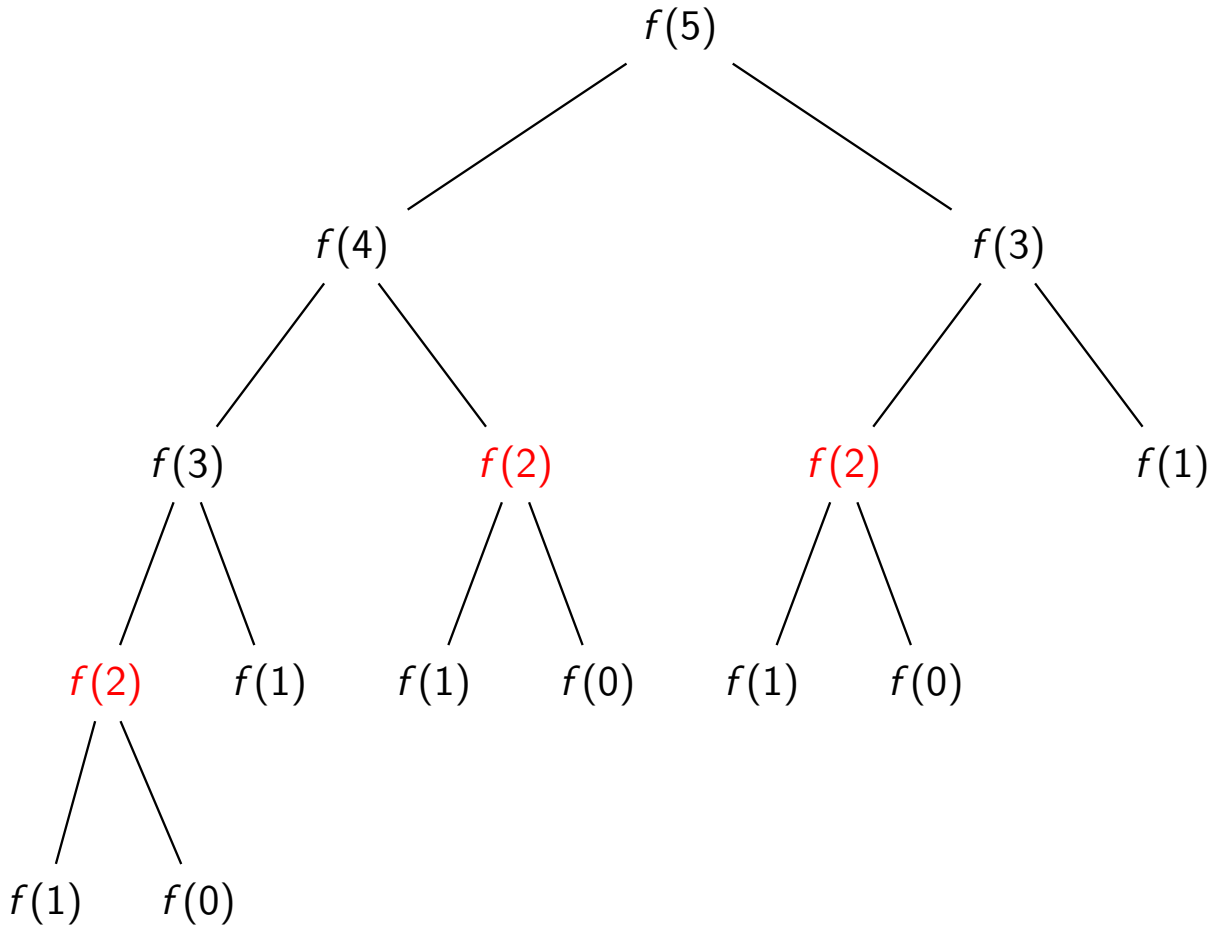
Aufrufbaum bei der Fibonacci-Folge

Es ergibt sich z. B. der folgende Aufrufbaum für die fünfte Fibonacci-Zahl, $f(5)$:



Wiederholte Berechnungen bei der Fibonacci-Folge

Es ergibt sich z. B. der folgende Aufrufbaum für die fünfte Fibonacci-Zahl, $f(5)$:



Exponentielles Wachstum des Aufrufbaums

Es sei $a(n)$ die Zahl der rekursiven Aufrufe von f für den Startwert n, dann gilt:

$$a(n) = a(n-1) + a(n-2) \geq a(n-2) + a(n-2) = 2 \cdot a(n-2),$$

Exponentielles Wachstum des Aufrufbaums

Es sei $a(n)$ die Zahl der rekursiven Aufrufe von f für den Startwert n, dann gilt:

$$a(n) = a(n-1) + a(n-2) \geq a(n-2) + a(n-2) = 2 \cdot a(n-2),$$

also

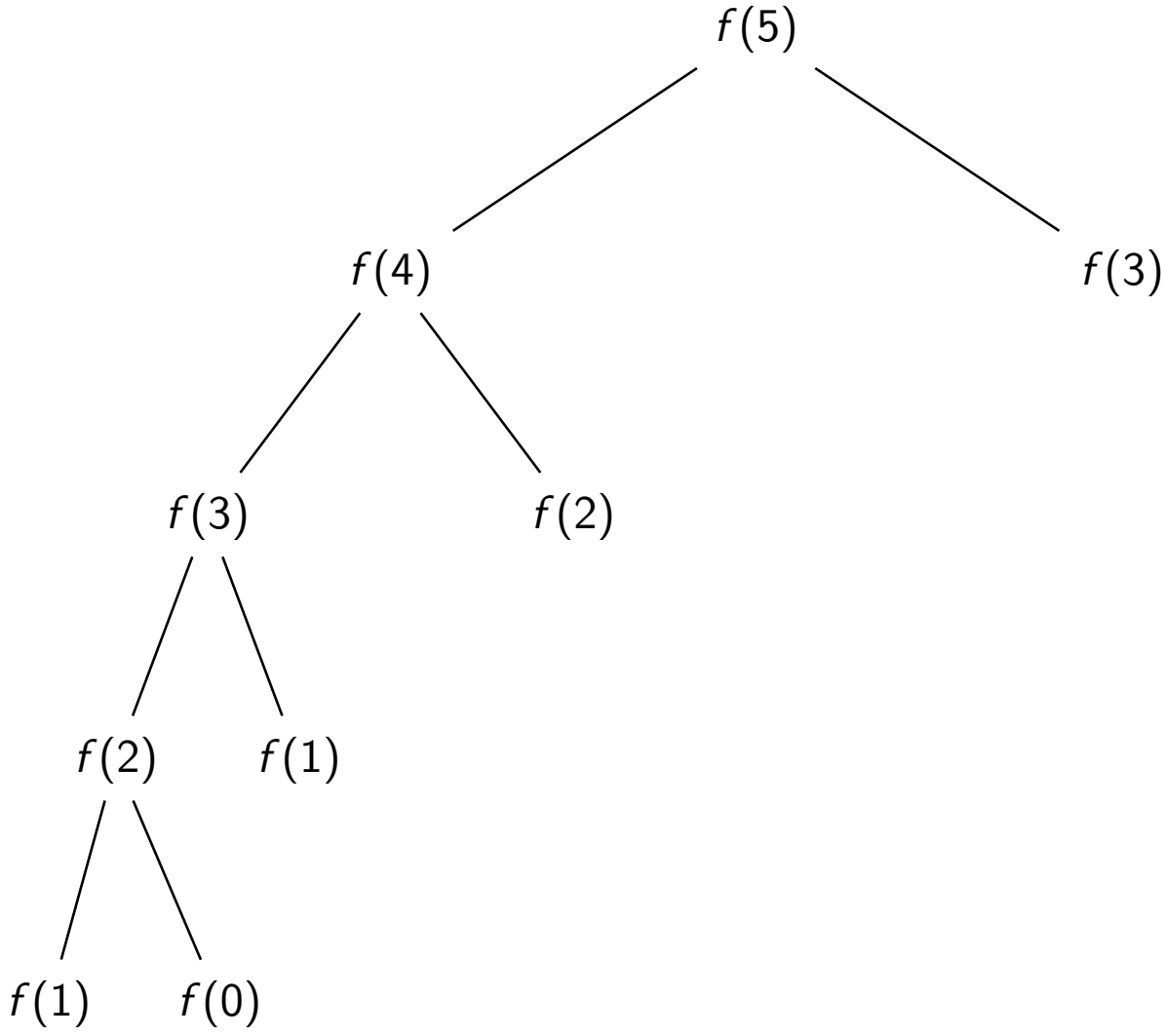
$$a(n) \approx 2^{\frac{n}{2}} \in \mathcal{O}(2^n).$$

Verbesserung durch Zwischenspeichern der Teilergebnisse

Die Grundidee der dynamischen Programmierung ist, einmal gelöste Teilprobleme zu speichern und wiederzuverwenden:

```
public long fibDynProg(int n) {
   long[] fiboFeld = new long[n+1];
   fibRekursivDyn(n, fibofeld);
   return fiboFeld[n];
}
private long fibRekursivDyn(int n, long[] fiboFeld) {
   if (n == 0 || n == 1) {
      fiboFeld[n]=1;
      return 1;
   }
   if(fiboFeld[n] != 0)
      return fiboFeld[n];
   return fiboFeld[n] = fibRekursivDyn(n-1,fiboFeld) +
       fibRekursivDyn(n-2,fiboFeld);
}
```

Aufrufbaum Fibonacci-Folge mit dynamischer Programmierung



Laufzeitgewinn und Trade-Off

- Die Lösung mit dynamischer Programmierung berechnet jede Fibonacci-Zahl nur noch ein Mal und hat eine Laufzeit von $\mathcal{O}(n)$
- Zur Erinnerung: Die rekursive Lösung hatte eine Laufzeit von $\mathcal{O}(2^n)$.

Laufzeitgewinn und Trade-Off

- Die Lösung mit dynamischer Programmierung berechnet jede Fibonacci-Zahl nur noch ein Mal und hat eine Laufzeit von $\mathcal{O}(n)$
- Zur Erinnerung: Die rekursive Lösung hatte eine Laufzeit von $\mathcal{O}(2^n)$.
- Dieser Laufzeitgewinn wird aber durch zusätzlichen Speicherbedarf erkauft
- Das ist eine generelle Strategie zur Optimierung: Speicherbedarf und Laufzeit können in vielen Fällen gegeneinander ausgetauscht werden.

Weitere Optimierung: Umformen in iterative Lösung

```
public long fibDynProgIterativ(int n) {
   long[] berechneteFiboWerte = new long[n+1];
   if(n < 2)
      return 1;
   berechneteFiboWerte[0] = berechneteFiboWerte[1] = 1;
   for(int i=2; i<=n; ++i)
      berechneteFiboWerte[i] =
         berechneteFiboWerte[i-1] +
            berechneteFiboWerte[i-2];
   return berechneteFiboWerte[n];
}
```

Optimierungsprobleme

- Wir konzentrieren uns auf Optimierungsprobleme:
- Probleme, die mehrere verschiedene Lösungen mit einem zugehörigen Wert (i.A: eine reelle Zahl) haben
- Ziel: Finden einer Lösung optimalen (d.h. minimalen oder maximalen) Wertes

Optimierungsprobleme

- Wir konzentrieren uns auf Optimierungsprobleme:
- Probleme, die mehrere verschiedene Lösungen mit einem zugehörigen Wert (i.A: eine reelle Zahl) haben
- Ziel: Finden einer Lösung optimalen (d.h. minimalen oder maximalen) Wertes
- Beispiele:
  - Kürzester Weg zwischen zwei Punkten auf einer Karte
  - Länge der längsten gemeinsamen Teilsequenz (z.B. in DNA-Strängen)

Die vier Schritte der dynamischen Programmierung

1. Charakterisierung der Struktur einer optimalen Lösung.
2. Rekursive Definition des Wertes einer optimalen Lösung.
3. Berechnung der optimalen Lösung, wahlweise mit einem iterativen (bottom-up) Verfahren, oder mit einem rekursiven Verfahren mit Speicherung der Zwischenergebnisse (Memoisation).
4. Konstruktion der Optimallösung aus den Zwischenergebnissen (sofern erforderlich).

Running Example: Das Stabzerlegungsproblem

Definition (Stabzerlegungsproblem)

Gegeben:

- $n \in \mathbb{N}$ (Stablänge)
- $p\colon \{1, 2, ..., n\} \to \mathbb{N}$ (Preisliste).

Gesucht:

- $z\colon \{1, 2, ..., n\} \to \mathbb{N}_0$ (Zerlegung), mit $\sum_{i=1}^{n} i \cdot z(i) = n$.

Wert (Erlös) einer Lösung (Zerlegung): $e(z) = \sum_{i=1}^{n} z(i) \cdot p(i)$

Wir nennen $e(z)$ den Erlös von z.

Ziel: Finde Zerlegung z, die den Erlös $e(z)$ maximiert.

Charakterisierung der Struktur einer optimalen Lösung

- Suche rekursive Formel, die die optimale Lösung in Abhängigkeit von optimalen Lösungen von Teilproblemen angibt.

Charakterisierung der Struktur einer optimalen Lösung

- Suche rekursive Formel, die die optimale Lösung in Abhängigkeit von optimalen Lösungen von Teilproblemen angibt.
- Beim Stabzerlegungsproblem:

  $$o(n) = \max\{p(n), o(1)+o(n-1), o(2)+o(n-2), ..., o(n-1)+o(1)\}$$

  wobei $o(0) = 0$
- Teilproblem heißt hierbei: vergleiche jede mögliche Schnittstelle des Stabes mit der Entscheidung, den Stab nicht weiter zu teilen.

Rekursive Definition des Wertes einer optimalen Lösung

- Ausgehend von der Rekursionsformel wird eine erste Implementation erstellt, die noch nicht optimiert ist.

Rekursive Definition des Wertes einer optimalen Lösung

- Ausgehend von der Rekursionsformel wird eine erste Implementation erstellt, die noch nicht optimiert ist.
- Beim Stabzerlegungsproblem:

```
public int eisenSchneidenRek(int n, int[] p) {
   if (n == 0)
      return 0;
   int optimum = p[n-1];
   for(int i = 1; i<n; ++i){
      int aktuellerErloes = eisenSchneidenRek(i, p)
         + eisenSchneidenRek(n-i,p);
      if(aktuellerErloes > optimum)
         optimum = aktuellerErloes;
   }
   return optimum;
}
```

Berechnung der optimalen Lösung

- Modifiziere die vorherige rekursive Lösung so, dass einmal gelöste Teilprobleme zwischengespeichert werden.

Berechnung der optimalen Lösung

- Modifiziere die vorherige rekursive Lösung so, dass einmal gelöste Teilprobleme zwischengespeichert werden.
- Beim Stabzerlegungsproblem:

```
public int eisenSchneidenDyn(int n, int[] p, int[] r) {
   if (r[n-1] > 0) return r[n-1];
   if (n == 0) return 0;
   int optimum = p[n-1];
   for(int i = 1; i < n; ++i){
      int aktuellerErloes = eisenSchneidenDyn(i, p, r)
         + eisenSchneidenDyn(n-i, p, r);
      if(aktuellerErloes > optimum)
         optimum = aktuellerErloes;
   }
   return r[n-1] = optimum;
}
```

Umformen in eine iterative Lösung

- Wenn erkennbar ist, welche Teilprobleme relevant sind, spart Umformen in eine iterative Lösung Overhead.

Umformen in eine iterative Lösung

- Wenn erkennbar ist, welche Teilprobleme relevant sind, spart Umformen in eine iterative Lösung Overhead.
- Beim Stabzerlegungsproblem:

```
public int eisenSchneidenIte(int n, int[] p) {
   int[] r = new int[n];
   for(int j = 1; j <= n; ++j){
      int optimum = p[j-1];
      for(int i = 1; i < j; ++i){
         int aktuellerErloes = r[i-1] + r[j-i-1];
         if(aktuellerErloes > optimum)
            optimum = aktuellerErloes;
      }
      r[j-1]=optimum;
   }
return r[n-1];
}
```

Konstruktion der Optimallösung

- Bislang haben wir nur den Wert der optimalen Lösung ermittelt.
- Die Lösung wird aber implizit konstruiert.

Konstruktion der Optimallösung

- Bislang haben wir nur den Wert der optimalen Lösung ermittelt.
- Die Lösung wird aber implizit konstruiert.
- Hierzu bei jeder Aktualisierung des Optimalwertes getroffene Entscheidung merken.
- Lösung kann durch kleine Modifikation der Implementation rekonstruiert werden.

Konstruktion der Optimallösung

Beim Stabzerlegungsproblem (Lösung in Feld s):

```
public int eisenSchneidenIte(int n, int[] p, int[] s) {
   int[] r = new int[n];
   for(int j = 1; j <= n; ++j){
      int optimum = p[j-1];
      for(int i = 1; i < j; ++i){
         int aktuellerErloes = r[i-1] + r[j-i-1];
         if(aktuellerErloes > optimum){
            optimum = aktuellerErloes;
            s[j]=i;
         }
      }
      r[j-1]=optimum;
   }
   return r[n-1];
}
```

Bedingungen für Dynamische Programmierung

- Dynamische Programmierung ist flexibel einsetzbar, aber kein Allheilmittel.
- Zwei Bedingungen müssen erfüllt sein, damit dynamische Programmierung einsetzbar ist:

Bedingungen für Dynamische Programmierung

- Dynamische Programmierung ist flexibel einsetzbar, aber kein Allheilmittel.
- Zwei Bedingungen müssen erfüllt sein, damit dynamische Programmierung einsetzbar ist:
  - Die Optimale Teilstruktur-Eigenschaft
  - Überlappende Teilprobleme

Optimale Teilstruktur

- Beschreibt Teilbarkeit in Teilprobleme
- Genauer: Wenn eine Lösung eines Problems optimal ist, dann ist auch die Lösung aller enthaltener Teilprobleme optimal
- Eigenschaft ist notwendig, um Korrektheit zu garantieren; ohne optimal Teilstruktur scheitert bereits die rekursive Lösung.

Beweis der Optimalen Teilstruktur

Optimale Teilstruktur-Eigenschaft zeigt man per Widerspruch mit Cut & Replace-Strategie:

1. Nimm an, Eigenschaft ist nicht erfüllt
2. Dann gibt es Probleminstanz P deren Optimallösung ein Teilproblem nicht ideal löst

Beweis der Optimalen Teilstruktur

Optimale Teilstruktur-Eigenschaft zeigt man per Widerspruch mit Cut & Replace-Strategie:

1. Nimm an, Eigenschaft ist nicht erfüllt
2. Dann gibt es Probleminstanz P deren Optimallösung ein Teilproblem nicht ideal löst
3. Ermittle optimale Lösung des Teilproblems
4. Ersetze Lösung des Teilproblems in Optimallösung von P

Beweis der Optimalen Teilstruktur

Optimale Teilstruktur-Eigenschaft zeigt man per Widerspruch mit Cut & Replace-Strategie:

1. Nimm an, Eigenschaft ist nicht erfüllt
2. Dann gibt es Probleminstanz P deren Optimallösung ein Teilproblem nicht ideal löst
3. Ermittle optimale Lösung des Teilproblems
4. Ersetze Lösung des Teilproblems in Optimallösung von P
5. Zeige, dass so korrekte Lösung von P entsteht
6. Zeige, dass Wert der Lösung besser als angenommene Optimallösung ist

Optimale Teilstruktur bei Stabteilung

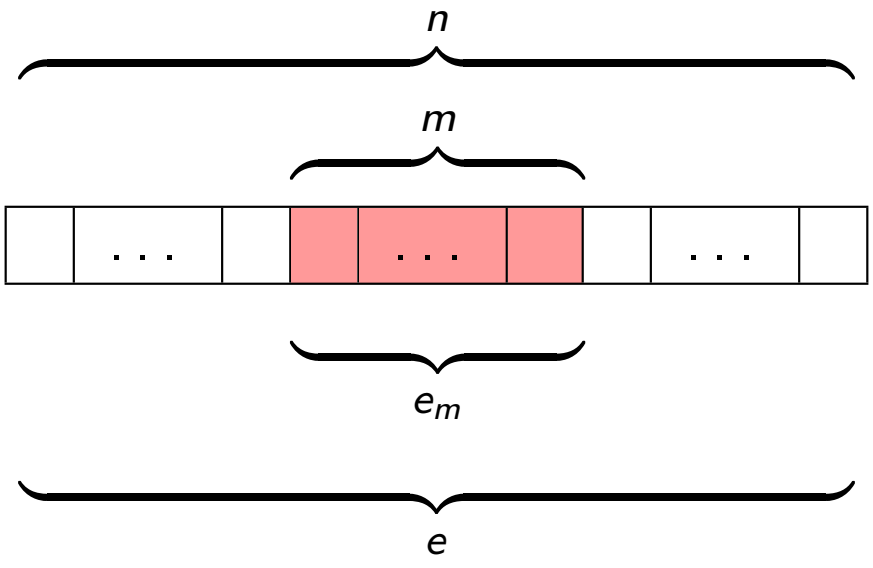
1. Nimm an, Eigenschaft ist nicht erfüllt

Angenommen, Stabteilungsproblem hat keine optimale Teilstruktur

Optimale Teilstruktur bei Stabteilung

2. Dann gibt es Probleminstanz P deren Optimallösung ein Teilproblem nicht ideal löst

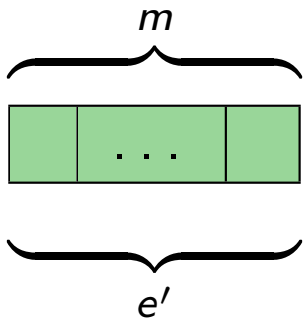
Es gibt Stablänge n, Preisliste p und Zerlegung z mit maximalem Erlös e, so dass es eine Teilstange der Länge m gibt, deren Erlös e_m nicht maximal ist:



Optimale Teilstruktur bei Stabteilung

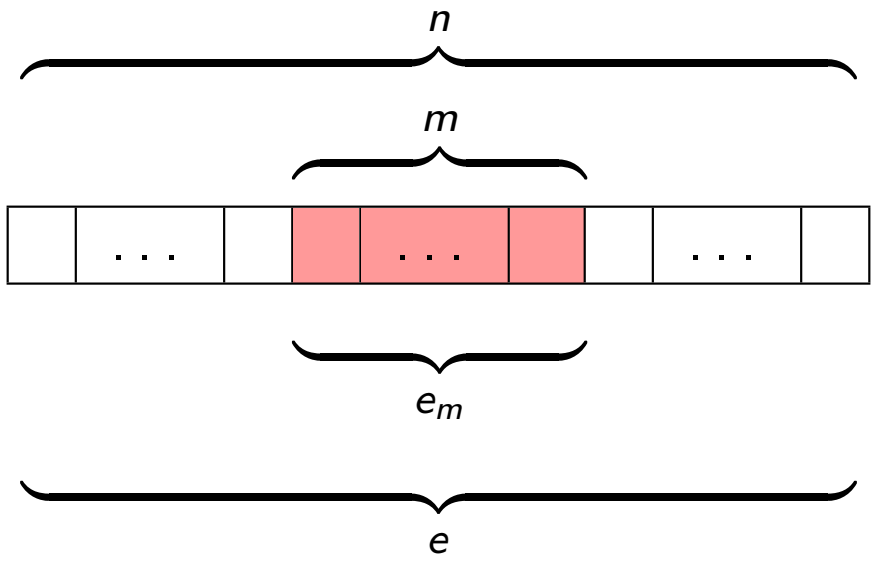
3. Ermittle optimale Lösung des Teilproblems

Es gibt also eine (andere) Zerteilung eines Stabes der Länge m mit Erlös $e' > e_m$



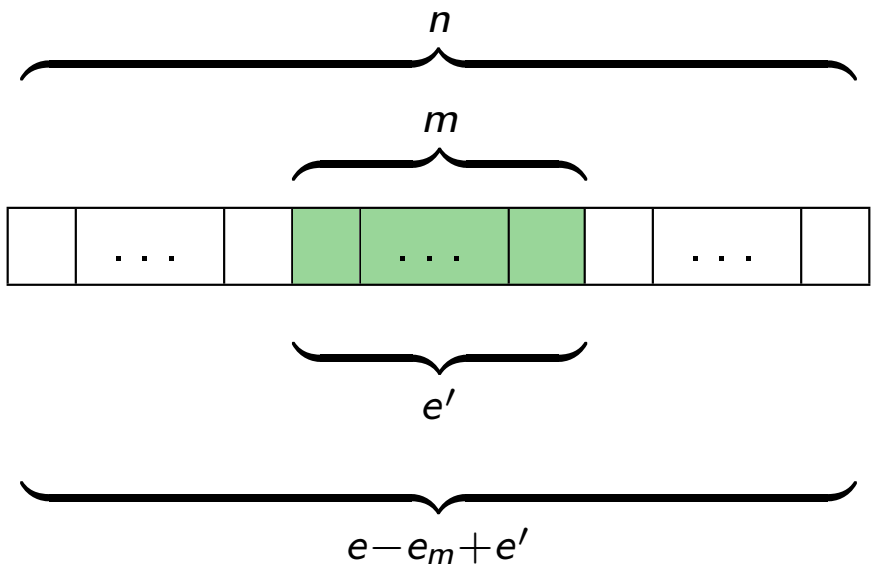
Optimale Teilstruktur bei Stabteilung

4. Ersetze Lösung des Teilproblems in Optimallösung von P



Optimale Teilstruktur bei Stabteilung

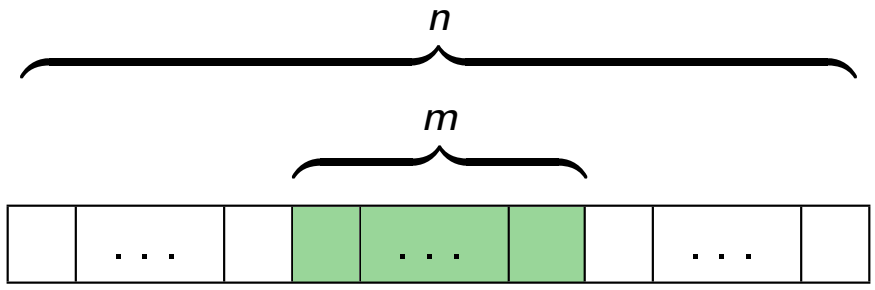
4. Ersetze Lösung des Teilproblems in Optimallösung von P



Optimale Teilstruktur bei Stabteilung

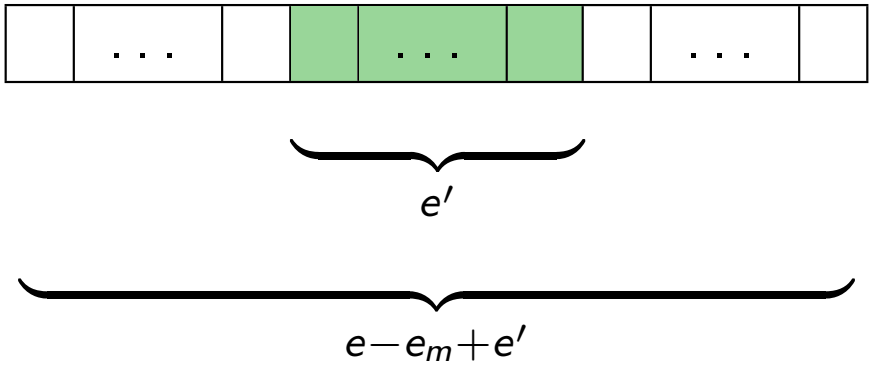
5. Zeige, dass so korrekte Lösung von P entsteht

Wir haben Stabteile der Gesamtlänge m entfernt und gegen eine andere Menge von Stabteilen ersetzt, die Gesamtlänge aller Stabteile bleibt unverändert n.



Optimale Teilstruktur bei Stabteilung

6. Zeige, dass Wert der Lösung besser als angenommene Optimallösung ist



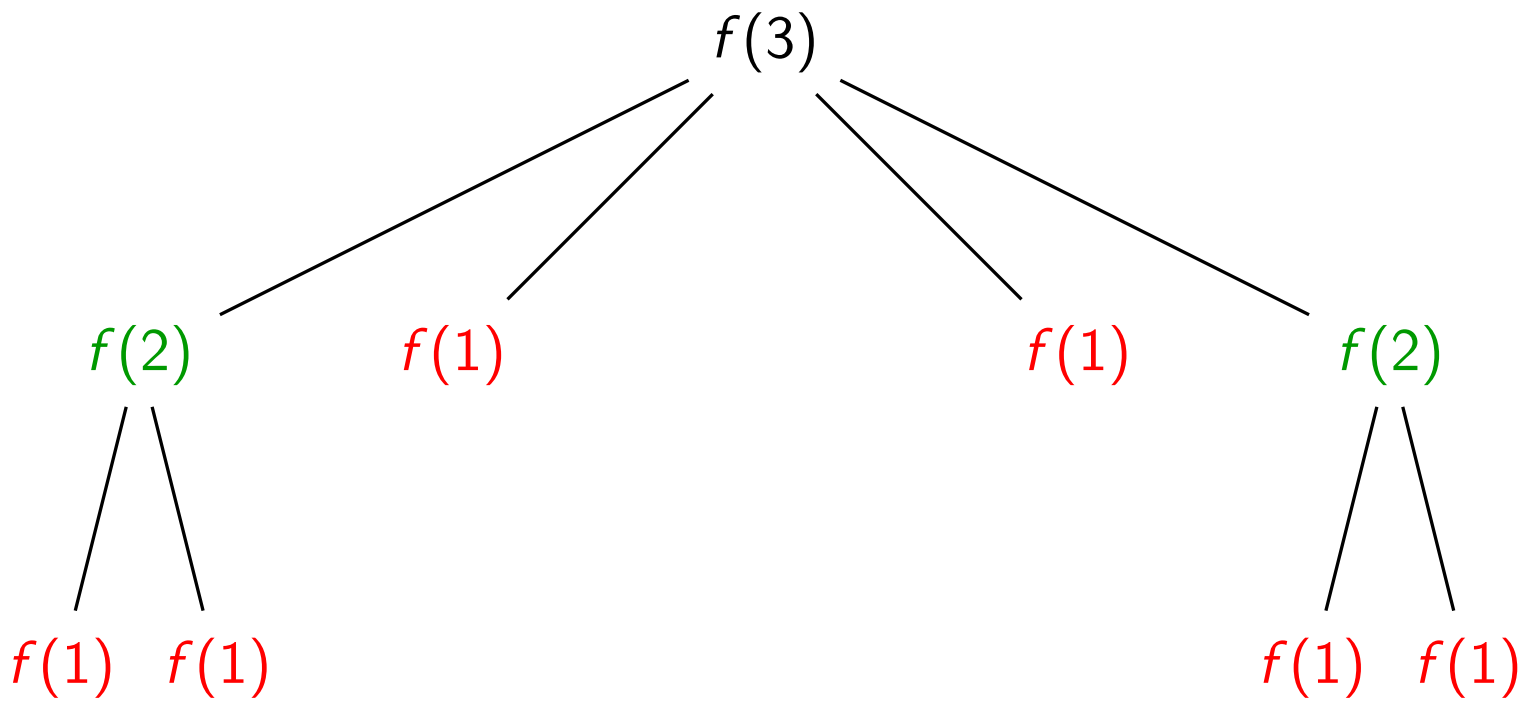
Da $e' > e_m$ gilt $e - e_m + e' > e$.

Überlappende Teilprobleme

- Besagt, dass bei Unterteilung in Teilprobleme mehrere gleiche Instanzen entstehen
- Genauer: Bei der rekursiven Definition wird mehrfach auf das gleiche Teilproblem Bezug genommen
- Eigenschaft ist notwendig, damit dynamische Programmierung einen Vorteil bietet

Überlappende Teilprobleme bei Stabteilung

Beispiel für einen Stab der Länge 3:



Fragen zur Vorlesungseinheit

1. Was sind die vier Schritte um einen Algorithmus mittels dynamischer Programmierung zu entwerfen?
2. Welche Eigenschaften muss ein Problem erfüllen um durch dynamische Programmierung gelöst zu werden?
3. Wie funktioniert der Beweis durch Cut & Replace?