

# Algorithmen und Datenstrukturen 2

## Lerneinheit 4: Dynamisches Programmieren

Prof. Dr. Christoph Karg

Studiengang Informatik  
Hochschule Aalen



Sommersemester 2016



10.5.2016

## Einleitung

Diese Lerneinheit widmet sich einer weiteren Programmietechnik zur Bearbeitung von Optimierungsproblemen, dem **Dynamischen Programmieren**.

Es werden folgende Fragen behandelt:

- Was ist die Idee hinter dynamischem Programmieren?
- Welche Voraussetzungen sind für dynamisches Programmieren notwendig?
- Was gibt es bei der Implementierung zu beachten?

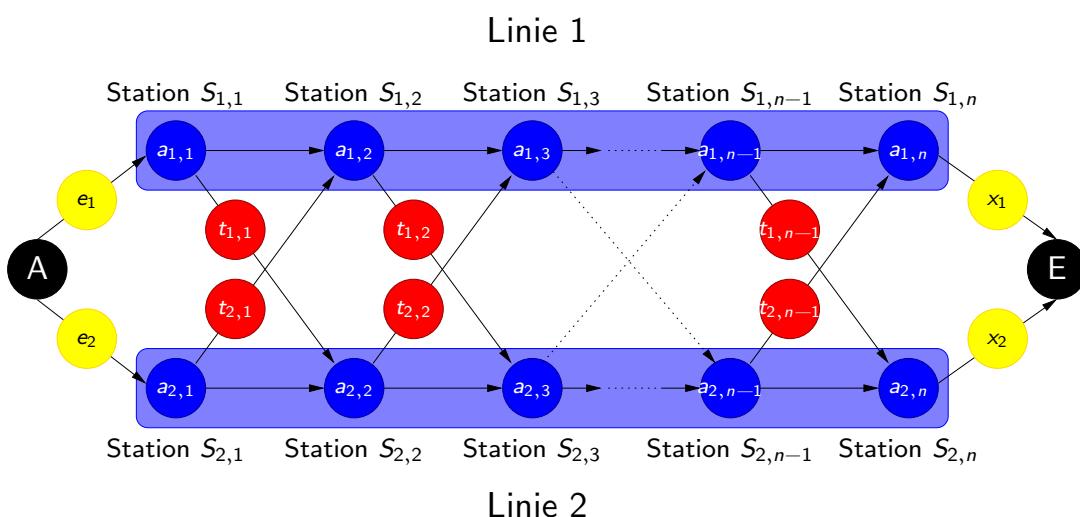
# Dynamisches Programmieren

Der Begriff „Programmierung“ steht für die schrittweise Berechnung einer Tabelle

Schritte zur Entwicklung eines Algorithmus:

1. Charakterisierung der Struktur einer optimalen Lösung
2. Berechnung des Werts einer optimalen Lösung
  - ▷ Top Down  $\rightsquigarrow$  rekursiver Algorithmus
  - ▷ Bottom Up  $\rightsquigarrow$  iterativer Algorithmus
3. Erweiterung des Algorithmus zur Konstruktion einer optimalen Lösung

# Fertigungsplanung



Aufgabe: Finde schnellste Produktionsroute

## Fertigungsplanung (Forts.)

- Zur Herstellung eines Produkts sind  $n$  Arbeitsschritte notwendig
- Die Arbeitsschritte müssen sequentiell durchlaufen werden
- Zwei Fertigungslinien mit jeweils  $n$  Stationen stehen bereit:

Linie 1:  $S_{1,1}, S_{1,2}, \dots, S_{1,n}$

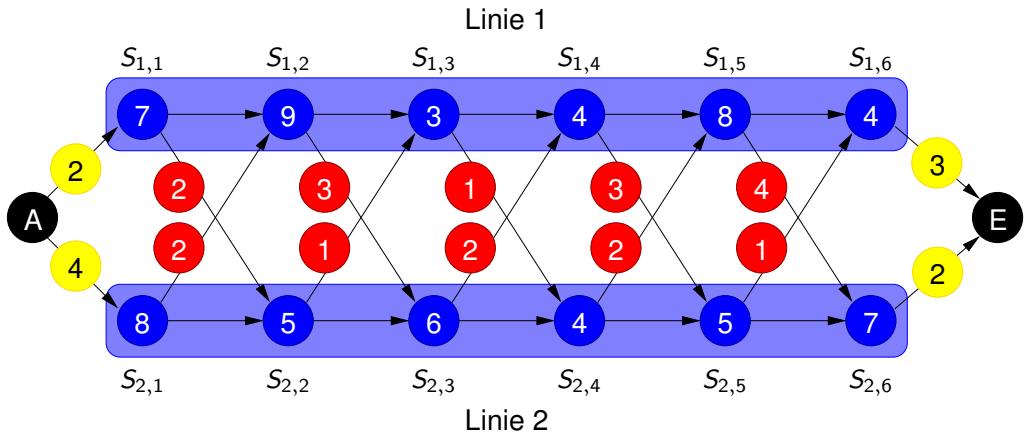
Linie 2:  $S_{2,1}, S_{2,2}, \dots, S_{2,n}$

- In der Regel durchläuft ein Produkt während der Herstellung eine der beiden Linien
- Für Expressbestellungen kann während der Produktion die Linie gewechselt werden

## Fertigungsplanung (Forts.)

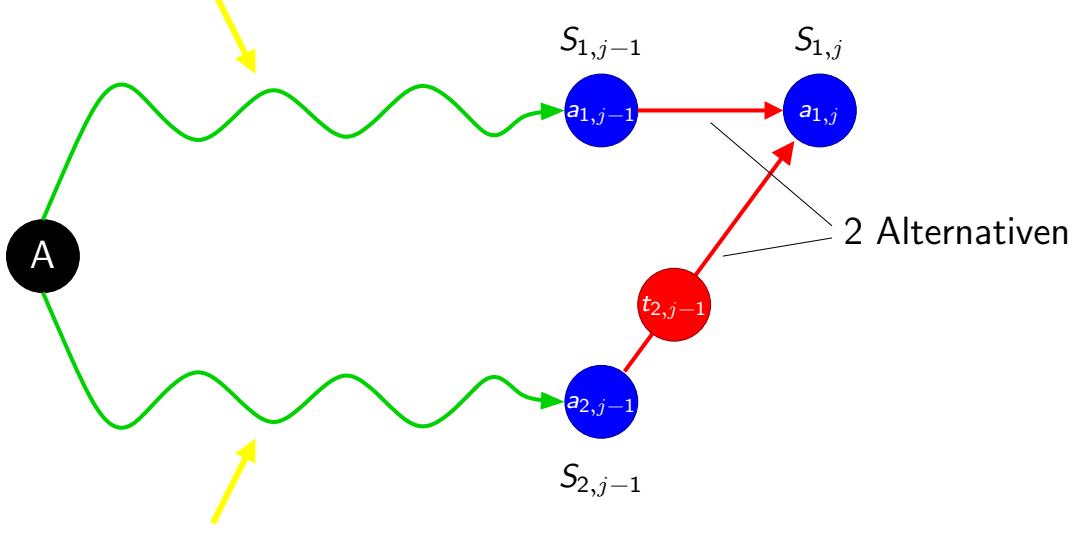
- Fertigungszeiten für Linie  $i = 1, 2$ :
  - ▷ Bereitstellungszeit  $e_i$ : Rohling betritt die Linie
  - ▷ Bearbeitungszeit  $a_{i,j}$  in Station  $j$ ,  $j = 1, \dots, n$ : Dauer des  $j$ -ten Fertigungsschritts in Linie  $i$
  - ▷ Beendigungszeit  $x_i$ : fertiges Produkt verlässt die Linie
- Die Dauer des Transports von Station  $j$  zu Station  $j + 1$  innerhalb derselben Linie ist vernachlässigbar
- Wechsel der Produktionslinien
  - ▷  $t_{1,j}$ : Dauer des Wechsels von  $S_{1,j}$  nach  $S_{2,j+1}$
  - ▷  $t_{2,j}$ : Dauer des Wechsels von  $S_{2,j}$  nach  $S_{1,j+1}$
- Aufgabe: Optimierung der Produktionszeit für Expressbestellungen

## Fertigungsplanung (Beispiel)



## Idee zur Berechnung des Optimums

Optimum zu Station  $S_{1,j-1}$



Optimum zu Station  $S_{2,j-1}$

## Rekursiver Ansatz

- **Ziel:** Rekursive Definition einer optimalen Lösung  $f^*$
- $f_i[j]$  bezeichnet die Zeit um auf schnellsten Weg vom Produktionsbeginn zum Ende von Station  $S_{i,j}$  zu gelangen
- Die Produktion beginnt bei Station  $S_{1,1}$  oder  $S_{2,1}$ :

$$\begin{aligned}f_1[1] &= e_1 + a_{1,1} \\f_2[1] &= e_2 + a_{2,1}\end{aligned}$$

- Ein Produkt muss alle  $n$  Stationen von Linie 1 oder 2 durchlaufen:

$$f^* = \min\{f_1[n] + x_1, f_2[n] + x_2\}$$

## Rekursiver Ansatz (Forts.)

Zwei Alternativen für den schnellsten Weg zu Station  $S_{1,j}$  wobei  $j = 2, 3, \dots, n$ :

- Direkter Weg von  $S_{1,j-1}$ :

$$v_1 = f_1[j-1] + a_{1,j}$$

- Transfer von Station  $S_{2,j-1}$  der Linie 2 auf Linie 1:

$$v_2 = f_2[j-1] + t_{2,j-1} + a_{1,j}$$

Der schnellste Weg verursacht minimale Kosten:

$$f_1[j] = \min\{v_1, v_2\}$$

Analoge Berechnung für  $f_2[j]$

## Rekursiver Ansatz (Forts.)

Rekursionsgleichung zur Berechnung von  $f_1[j]$ :

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & j = 1 \\ \min\{f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}\} & j \geq 2 \end{cases}$$

Rekursionsgleichung zur Berechnung von  $f_2[j]$ :

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & j = 1 \\ \min\{f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}\} & j \geq 2 \end{cases}$$

## Korrektheit des Verfahrens

**Satz:** Für alle  $i \in \{1, 2\}$  und alle  $j \in \{1, \dots, n\}$  ist  $f_i[j]$  die kürzeste Produktionszeit bis zu Station  $S_{i,j}$ .

*Beweis.* Induktion über  $n$ .

$\textcolor{blue}{j=1}$ : Da die Werte  $f_1[1] = e_1 + a_{1,1}$  und  $f_2[1] = e_2 + a_{2,1}$  die einzigen möglichen sind, sind sie optimal.

$\textcolor{blue}{j-1 \rightsquigarrow j}$ : Sei  $i = 1$  und  $w$  die minimale Produktionszeit bis zu Station  $S_{i,j}$ . Betrachte den durch die Rekursion berechneten Wert  $f_1[j]$ .

$\textcolor{blue}{w \leq f_1[j]}$ : ✓

## Korrektheit des Verfahrens (Forts.)

$w \geq f_1[j]$ : Angenommen, die optimale Produktionsroute zu  $S_{1,j}$  führt über  $S_{1,j-1}$ .

Da laut Annahme  $f_1[j-1]$  und  $f_2[j-1]$  optimal sind, gilt:

$$w \geq f_1[j-1] + a_{1,j} \geq f_1[j]$$

Führt der optimale Weg über  $S_{2,j-1}$ , dann gilt:

$$w \geq f_2[j-1] + t_{2,j-1} + a_{1,j} \geq f_1[j]$$

Insgesamt:  $w = f_1[j]$ . Der rekursiv berechnete Wert ist also optimal.

Analog:  $i = 2$

## Erweiterung: Berechnung des besten Weges

- $\ell_i[j]$  bezeichnet die Nummer der Produktionslinie deren Station  $S_{\ell_i[j],j-1}$  auf dem schnellsten Weg zu  $S_{i,j}$  liegt
- $\ell^*$  bezeichnet die Linie, deren  $n$ -te Station auf dem schnellsten Produktionsweg liegt
- Die Werte  $\ell_i[j]$  für  $i = 1, 2$  und  $j = 2, 3, \dots, n$  sind ein Nebenprodukt der Berechnung obiger Rekursionsgleichungen
- Beachte:  $\ell_i[1]$  ist nicht definiert

## Rekursiver Algorithmus

$\text{RECF}(i, j, e, a, t)$

**Output:**  $f_i[j]$

```
1 if  $i = 1$  then
2   if  $j = 1$  then
3     return  $e_1 + a_{1,1}$ 
4   else
5      $v_1 := \text{RECF}(1, j - 1, e, a, t) + a_{1,j}$ 
6      $v_2 := \text{RECF}(2, j - 1, e, a, t) + t_{2,j-1} + a_{1,j}$ 
```

## Rekursiver Algorithmus

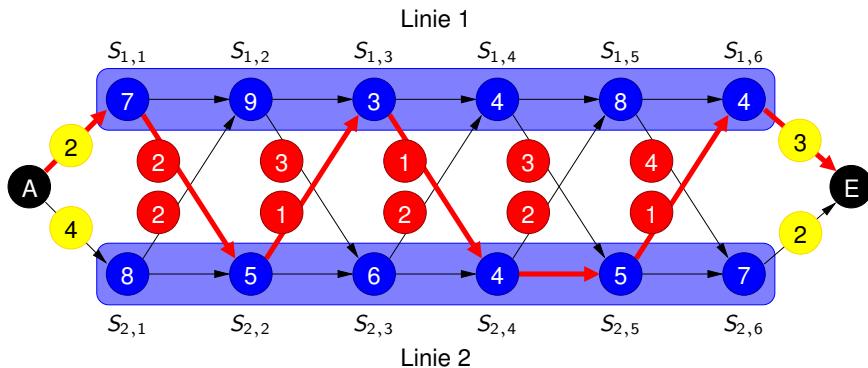
```
7   if  $v_1 \leq v_2$  then
8      $\ell_1[j] := 1$ 
9     return  $v_1$ 
10   else
11      $\ell_1[j] := 2$ 
12     return  $v_2$ 
13 else
14   Analog für  $i = 2$ 
```

**Ergebnis:**

$$f^* = \min\{\text{RECF}(1, n, e, a, t) + x_1, \text{RECF}(2, n, e, a, t) + x_2\}$$

Die optimale Route wird intern im Array  $\ell$  gespeichert.

## Beispiel



$j$	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$j$	2	3	4	5	6
$\ell_1[j]$	1	2	1	1	2
$\ell_2[j]$	1	2	1	2	2

**Ergebnis:**  $f^* = 38$ ,  $\ell^* = 1$

## Laufzeit des rekursiven Algorithmus

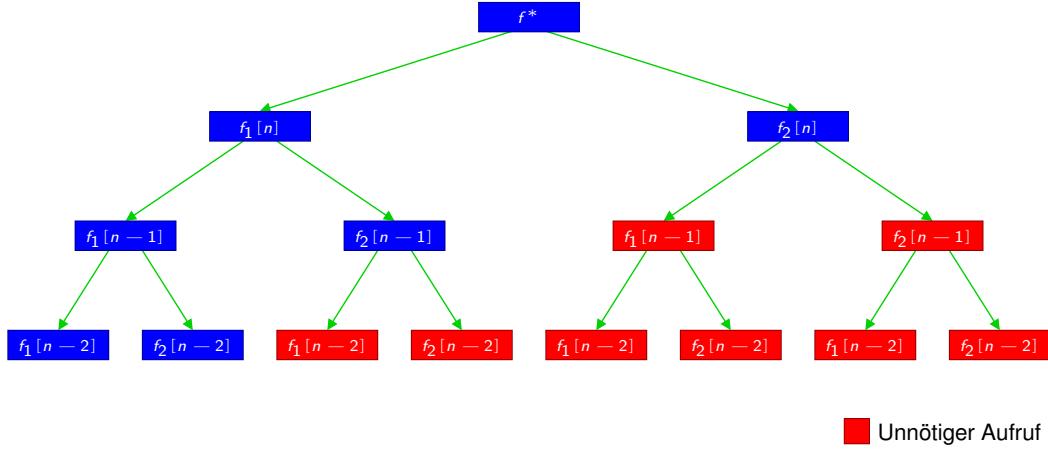
**Laufzeit:** Für die Anzahl  $T(n)$  der rekursiven Aufrufe von  $\text{RECF}(i, n, e, a, t)$  gilt:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T(n-1) & n \geq 2 \end{cases}$$

**Somit:** Laufzeit von  $\text{RECF}(i, n, e, a, t)$  ist  $\Theta(2^n)$

**Fazit:** rekursiver Algorithmus unbrauchbar

## Grund für die Ineffizienz



**Problem:** Die meisten rekursiven Aufrufe sind unnötig

**Lösung:** Speichere bereits berechnete Werte in einer Tabelle ( $\rightsquigarrow$  Memoization)

## Modifizierter Algorithmus

MEMORECF( $i, j, e, a, t$ )

**Output:**  $f_i[j]$

```
1 if  $f_i[j] < \infty$  then
2   return  $f_i[j]$ 
3 else
4   if  $i = 1$  then
5     if  $j = 1$  then
6       return  $e_1 + a_{1,1}$ 
7     else
8        $v_1 := \text{MEMORECF}(1, j - 1, e, a, t) + a_{1,j}$ 
9        $v_2 := \text{MEMORECF}(2, j - 1, e, a, t) + t_{2,j-1} + a_{1,j}$ 
```

## Rekursiver Algorithmus

```
10      if  $v_1 \leq v_2$  then
11           $f_1[j] := v_1$ 
12           $\ell_1[i] := 1$ 
13          return  $v_1$ 
14      else
15           $f_1[j] := v_2$ 
16           $\ell_1[i] := 2$ 
17          return  $v_2$ 
18      else
19          Analog für  $i = 2$ 
```

**Laufzeit:**  $O(n)$ , da die Anzahl der rekursiven Aufrufe gleich  $2n$  ist

## Optimale Teilstruktur

- **Optimale Teilstruktur Eigenschaft:**  
Jede optimale Lösung eines Problems enthält optimale Lösungen für die im Problem enthaltenen Teilprobleme
- Das Problem Fertigungsplanung besitzt diese Eigenschaft, denn der schnellste Weg zu  $S_{1,j}$  ist entweder
  - ▷ der schnellste Weg zu  $S_{1,j-1}$  und direkt weiter zu  $S_{1,j}$ , oder
  - ▷ der schnellste Weg zu  $S_{2,j-1}$  und ein Wechsel von Linie 2 zu Linie 1, also zu  $S_{1,j}$Analog für  $S_{2,j}$
- Die Optimale Teilstruktur Eigenschaft ermöglicht eine effiziente Berechnung einer optimalen Lösung.

## Iterativer Algorithmus

```
ITERATIVEF( $n, e, a, t, x$ )
1  $f_1[1] := e_1 + a_{1,1}$ 
2  $f_2[1] := e_2 + a_{2,1}$ 
3 for  $j := 2$  to  $n$  do
4   if  $f_1[j - 1] + a_{1,j} \leq f_2[j - 1] + t_{2,j-1} + a_{1,j}$  then
5      $f_1[j] := f_1[j - 1] + a_{1,j}$ 
6      $\ell_1[j] := 1$ 
7   else
8      $f_1[j] := f_2[j - 1] + t_{2,j-1} + a_{1,j}$ 
9      $\ell_1[j] := 2$ 
```

## Iterativer Algorithmus (Forts.)

```
10  if  $f_2[j - 1] + a_{2,j} \leq f_1[j - 1] + t_{1,j-1} + a_{2,j}$  then
11     $f_2[j] := f_2[j - 1] + a_{2,j}$ 
12     $\ell_2[j] := 2$ 
13  else
14     $f_2[j] := f_1[j - 1] + t_{1,j-1} + a_{2,j}$ 
15     $\ell_1[j] := 1$ 
16  if  $f_1[n] + x_1 \leq f_2[n] + x_2$  then
17     $f^* := f_1[n] + x_1$ 
18     $\ell^* := 1$ 
19  else
20     $f^* := f_2[n] + x_2$ 
21     $\ell^* := 2$ 
```

## Bemerkungen zu ITERATIVEF

- ITERATIVEF() berechnet die Tabellen  $f$  und  $\ell$  in Bottom Up Manier
- In der for-Schleife in Zeilen 3–15 werden die Tabellen schrittweise unter Verwendung der Rekursionsgleichungen berechnet
- Die if-Anweisung in Zeilen 16–21 dient zur Berechnung der optimalen Werts
- Die Laufzeit von ITERATIVEF() ist  $\Theta(n)$ , also linear in der Anzahl der Stationen einer Produktionslinie

## Ausgabe des schnellsten Pfads

Nach Aufruf von ITERATIVEF() enthält die  $\ell$ -Tabelle alle Informationen um den schnellen Produktionspfad auszugeben.

```
PRINTSTATIONS( $\ell$ ,  $n$ )
1 for  $j := 2$  to  $n$  do
2    $i := \ell_i[j]$ 
3   print "Linie "  $i$  ", Station "  $j - 1$ 
4    $i := \ell^*$ 
5   print "Linie "  $i$  ", Station "  $n$ 
```

# Matrizenmultiplikation

- Eine  $n \times m$  Matrix  $A$  ist eine Tabelle mit  $m$  Zeilen und  $n$  Spalten:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m-1} & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m-1} & a_{2,m} \\ \vdots & & & & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m-1} & a_{n,m} \end{pmatrix}$$

- Zwei Matrizen  $A$  und  $B$  kann man multiplizieren, falls die Anzahl der Spalten von  $A$  gleich der Anzahl der Zeilen von  $B$  ist

# Matrizenmultiplikation (Forts.)

- Ist  $A$  eine  $n \times m$  Matrix und  $B$  eine  $m \times \ell$  Matrix, dann ist das Produkt  $C = A \cdot B$  eine  $n \times \ell$ , wobei

$$c_{i,j} = \sum_{k=1}^m a_{i,k} \cdot b_{k,j}$$

für  $i = 1, 2, \dots, n$  und  $j = 1, 2, \dots, \ell$

- Falls man  $A$  und  $B$  multiplizieren kann, dann nennt man  $A$  und  $B$  (multiplikations-) kompatibel

## Algorithmus MATRIXMULTIPLY( $A, B$ )

MATRIXMULTIPLY( $A, B$ )

**Input:** Matrizen  $A$  und  $B$

**Output:** Matrix  $C = A \cdot B$ , falls Produkt berechenbar

```
1 if columns(A)  $\neq$  rows(B) then
2   print "Dimensionen nicht kompatibel"
3 else
4   for  $i := 1$  to rows(A) do
5     for  $j := 1$  to columns(B) do
6        $C[i, j] := 0$ 
7       for  $k := 1$  to columns(A) do
8          $C[i, j] := C[i, j] + A[i, k] \cdot B[k, j]$ 
9   return  $C$ 
```

**Laufzeit:**  $O(\text{rows}(A) \cdot \text{columns}(B) \cdot \text{columns}(A))$

## Assoziativitat der Matrixmultiplikation

- Gegeben: Matrizen  $A_1, A_2, \dots, A_n$ , wobei  $A_i$  und  $A_{i+1}$  fur alle  $i = 1, 2, \dots, n-1$  kompatibel sind
- Zur Berechnung von  $C = A_1 \cdot A_2 \cdot \dots \cdot A_n$  mussen  $n-1$  Matrixmultiplikationen durchgefuhrt werden
- Da die Matrixmultiplikation assoziativ ist, ist die Reihenfolge der Multiplikationen (aus mathematischer Sicht) irrelevant
- Die Reihenfolge beeinflusst jedoch den Aufwand zur Berechnung der Multiplikationen

# Kettenmultiplikation von Matrizen

Matrizen-Kettenmultiplikation steht für das folgende Optimierungsproblem:

- **Gegeben:** Eine Folge (Kette) von Matrizen  $\langle A_1, A_2, \dots, A_n \rangle$ , wobei  $A_i$  die Dimension  $p_{i-1} \times p_i$  hat
- **Aufgabe:** Berechne das Produkt  $A_1 \cdot A_2 \cdot \dots \cdot A_n$  mit minimalem Aufwand

**Beispiel:** Betrachte die Matrizen  $A_1$ ,  $A_2$  und  $A_3$  mit Dimension  $10 \times 100$ ,  $100 \times 5$  und  $5 \times 50$

- $(A_1 \cdot (A_2 \cdot A_3)) \rightsquigarrow 75000$  Multiplikationen
- $((A_1 \cdot A_2) \cdot A_3) \rightsquigarrow 7500$  Multiplikationen

# Anzahl verschiedener Klammerungen

- **Frage:** Auf wieviele verschiedene Arten kann  $n$  Matrizen klammern?
- Die Anzahl  $P(n)$  der Klammerungen ist

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n \geq 2 \end{cases}$$

- Es gilt:  $P(n) = \Omega(2^n)$
- **Fazit:** Alle Klammerungen zu analysieren, um die Matrix Kettenmultiplikation optimal zu berechnen, ist nicht effizient.

## Lösung mittels dynamischem Programmieren

Die optimale Klammerung kann mittels dynamischem Programmieren berechnet werden.

Vier Schritte:

1. Analyse der Struktur einer optimalen Klammerung
2. Rekursive Berechnung der Anzahl Skalarmultiplikationen der optimalen Klammerung
3. Berechnung der Anzahl Skalarmultiplikationen durch schrittweise Berechnung einer Tabelle
4. Berechnung der optimalen Klammerung

## Schritt 1: Strukturanalyse

Betrachte eine optimale Klammerung für  $A_i A_{i+1} \dots A_j$  wobei  $1 \leq i < j \leq n$ :

- Die Klammerung teilt das Produkt  $A_i A_{i+1} \dots A_j$  in zwei Hälften  $A_i A_{i+1} \dots A_k$  und  $A_{k+1} \dots A_j$  wobei  $i \leq k < j$
- Die Klammerung von  $A_i A_{i+1} \dots A_k$  ist optimal. Andernfalls könnte man die optimale Klammerung für  $A_i A_{i+1} \dots A_j$  verbessern
- Die Klammerung von  $A_{k+1} \dots A_j$  ist ebenfalls optimal

**Fazit:** Matrix Kettenmultiplikation besitzt die Optimale Teilstruktur Eigenschaft

## Schritt 2: Rekursive Berechnung des Optimums

- Definiere  $A_{i..j} = A_i \cdot A_{i+1} \cdot \dots \cdot A_j$  wobei  $i \leq j$
- Sei  $m[i, j]$  die minimale Anzahl Skalarmultiplikationen die zur Berechnung von  $A_{i..j}$  notwendig sind
- Es gilt:  $A_{i..j} = A_{i..k} \cdot A_{k+1..j}$  für  $k = i, i + 1, \dots, j - 1$
- Der Aufwand zur Berechnung von  $A_{i..k} \cdot A_{k+1..j}$  ist

$$m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

- Rekursive Berechnung von  $m[i, j]$ :

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

## Schritt 3: Berechnung der Lösung

- In  $s[i, j]$  wird die Stelle  $k$  gespeichert, an der  $A_{i..j}$  zerlegt wird
- Die direkte Implementierung der Rekursionsgleichung ist ineffizient
- Zur Berechnung werden die Werte  $m[i, j]$  für  $1 \leq i \leq j \leq n$  benötigt
- Die Anzahl der  $m$ 's ist  $\binom{n}{2} + n = \Theta(n^2)$
- Die  $m$ 's werden in einer Dreiecksmatrix gespeichert

## Algorithmus MATRIXCHAINORDER( $p$ )

MATRIXCHAINORDER( $p$ )

**Input:** Dimensionsfolge  $p = \langle p_0, p_1, \dots, p_n \rangle$

**Output:** Tabelle  $m$  und  $s$

```
1   $n := \text{length}(p) - 1$ 
2  for  $i := 1$  to  $n$  do
3     $m[i, i] := 0$ 
4  for  $\ell := 2$  to  $n$  do
5    for  $i := 2$  to  $n - \ell + 1$  do
6       $j := i + \ell - 1$ 
7       $m[i, j] := \infty$ 
```

## Algorithmus MATRIXCHAINORDER( $p$ )

```
8    for  $k := i$  to  $j - 1$  do
9       $q := m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$ 
10     if  $q < m[i, j]$  then
11        $m[i, j] := q$ 
12        $s[i, j] := k$ 
13   return  $m$  und  $s$ 
```

**Laufzeit:**  $\Theta(n^3)$

## Schritt 4: Berechnung der optimalen Klammerung

Die korrekte Klammerung wird anhand der Tabelle  $s$  berechnet:

```
PRINTOPTIMALPARENS( $s, i, j$ )
1 if  $i = j$  then
2   print "A["  $i$  "]"
3 else
4   print "("
5   PRINTOPTIMALPARENS( $s, i, s[i, j]$ )
6   PRINTOPTIMALPARENS( $s, s[i, j] + 1, j$ )
7   print ")"
```

## Beispiel für Matrix Kettenmultiplikation

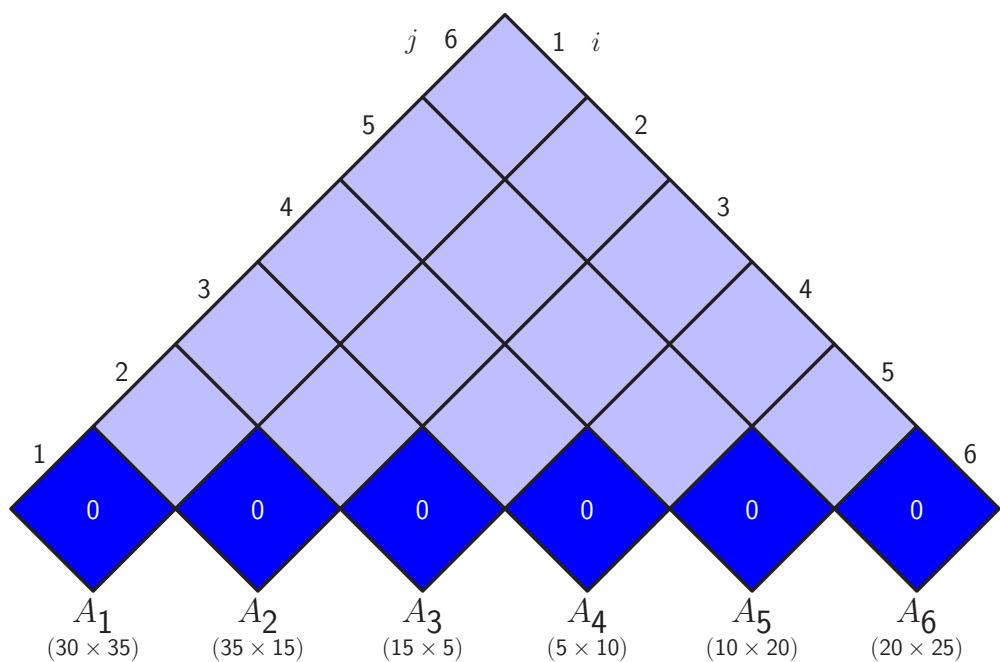
Gegeben sind die Matrizen  $A_1, A_2, \dots, A_6$  mit folgenden Dimensionen:

Matrix	Dimension
$A_1$	$30 \times 35$
$A_2$	$35 \times 15$
$A_3$	$15 \times 5$
$A_4$	$5 \times 10$
$A_5$	$10 \times 20$
$A_6$	$20 \times 25$

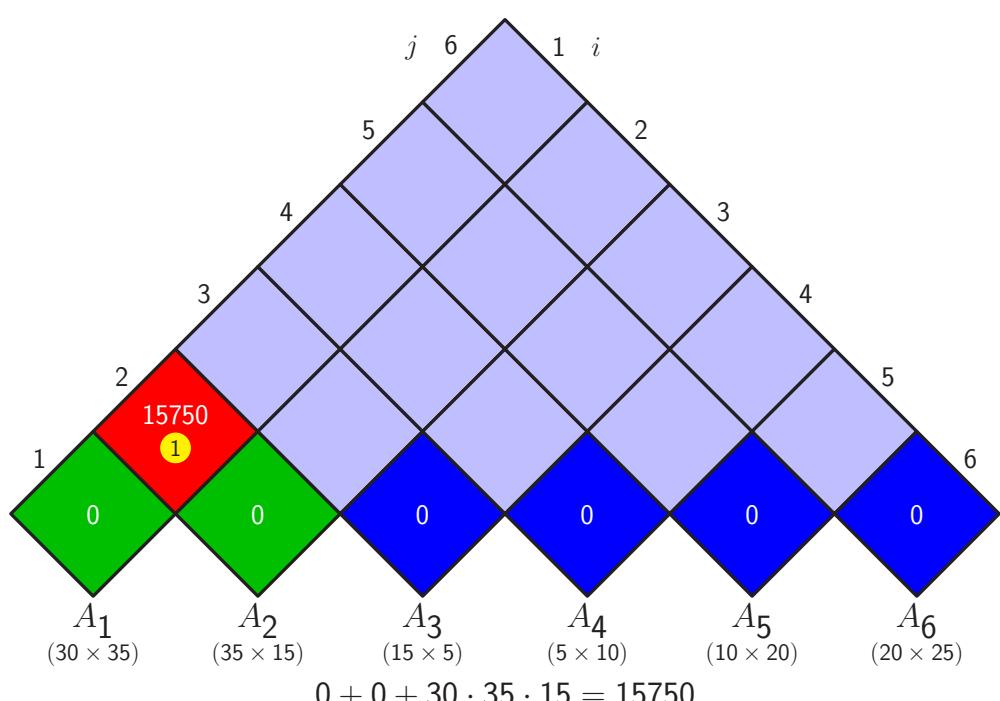
Eingabe für MATRIXCHAINORDER( $p$ ):

$$p = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$$

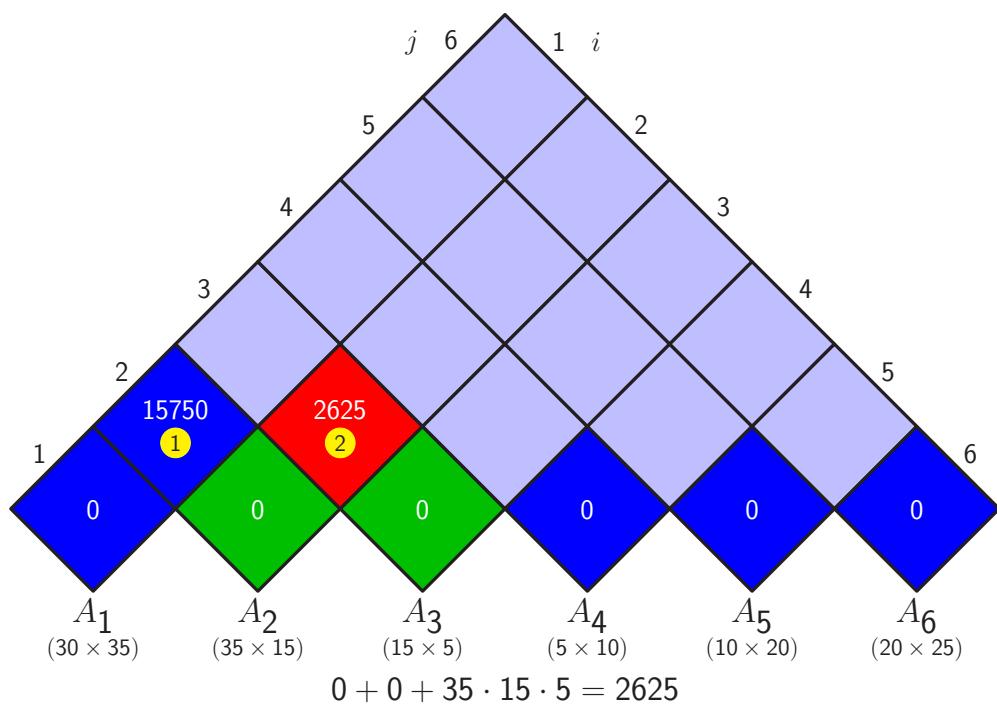
## Beispiel (Forts.)



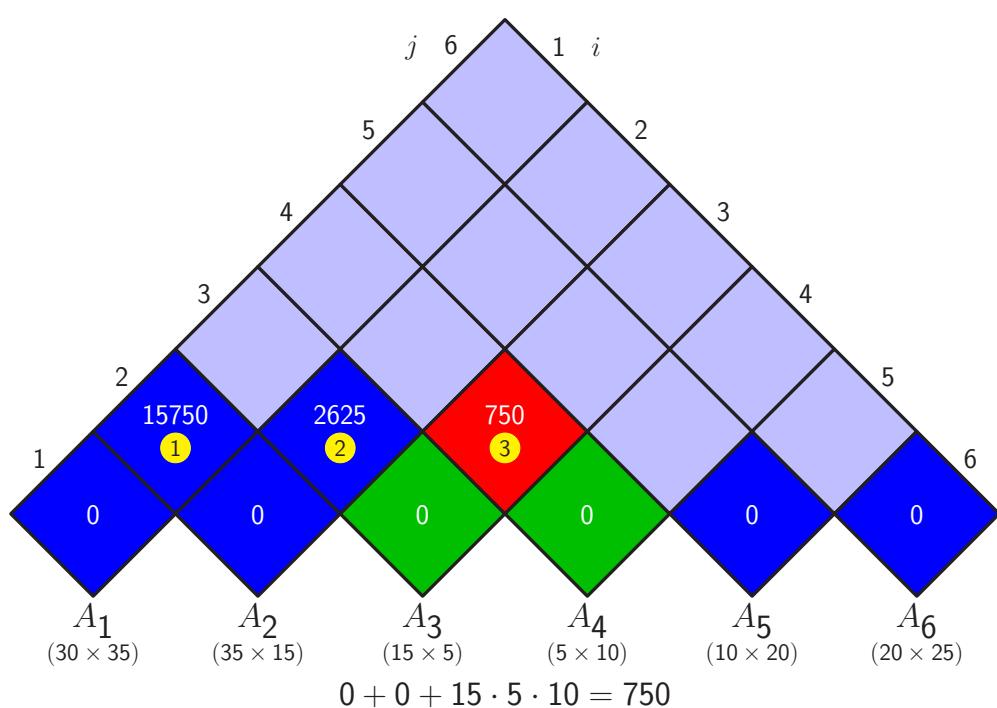
## Beispiel (Forts.)



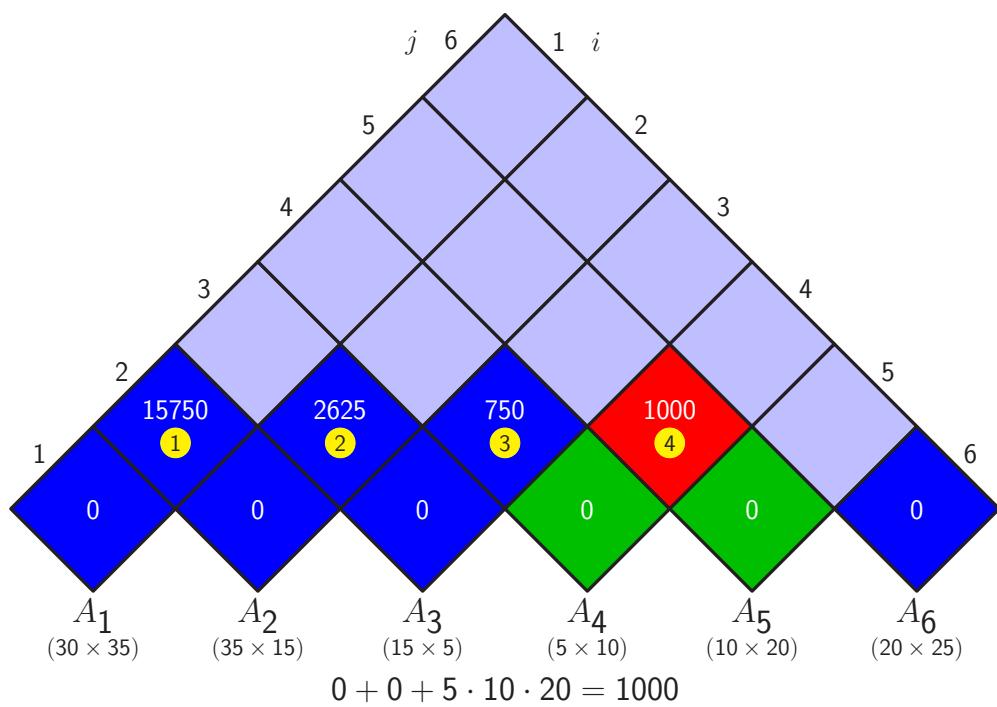
## Beispiel (Forts.)



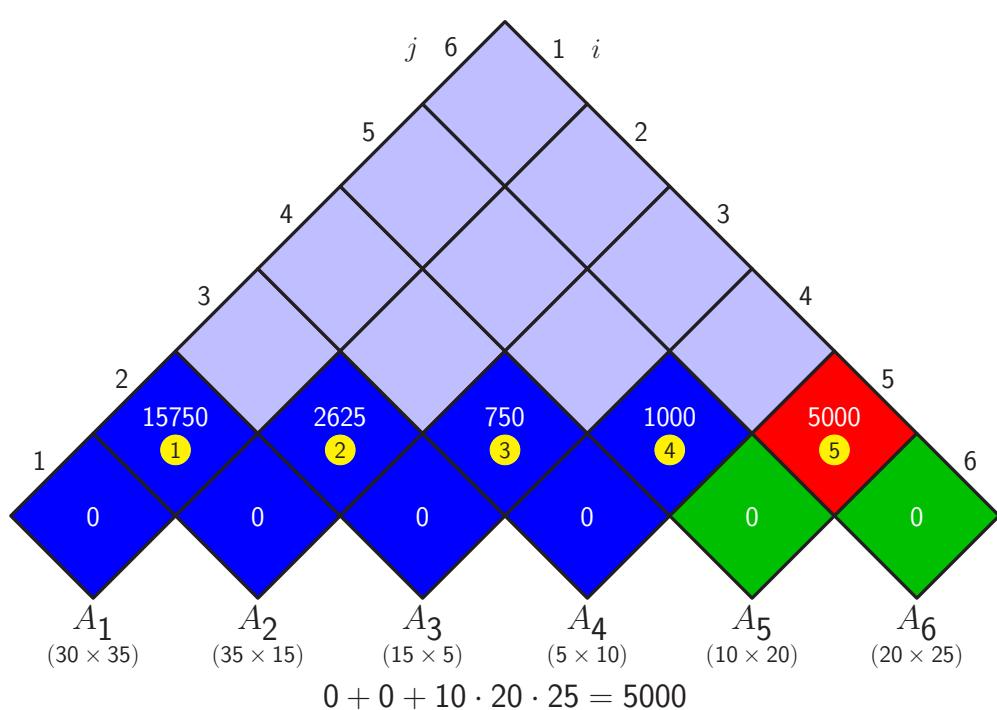
## Beispiel (Forts.)



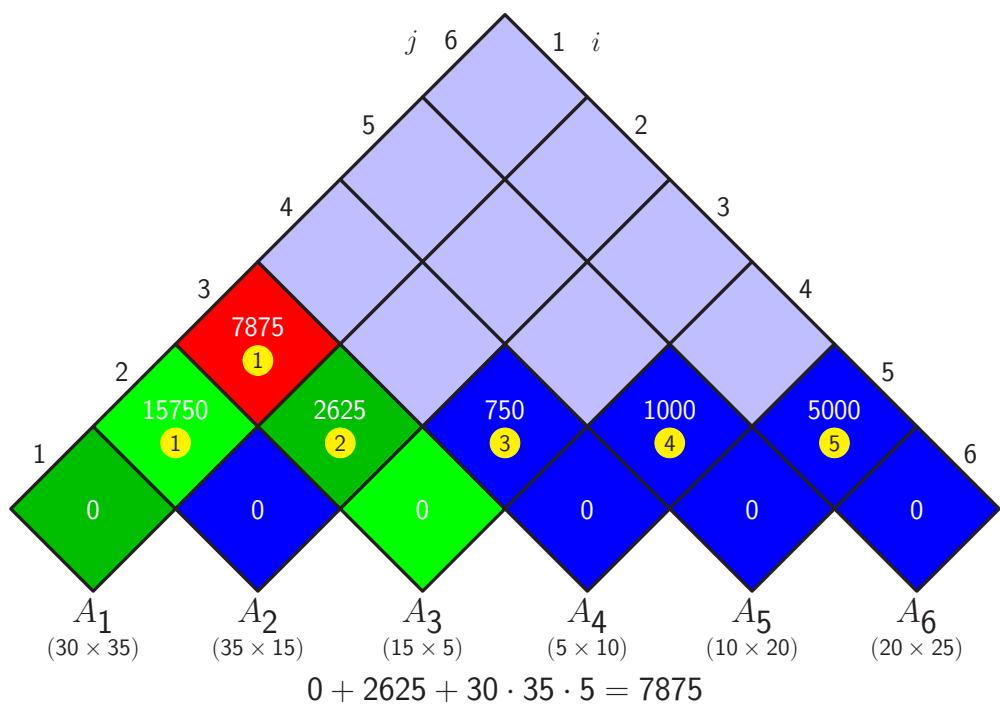
## Beispiel (Forts.)



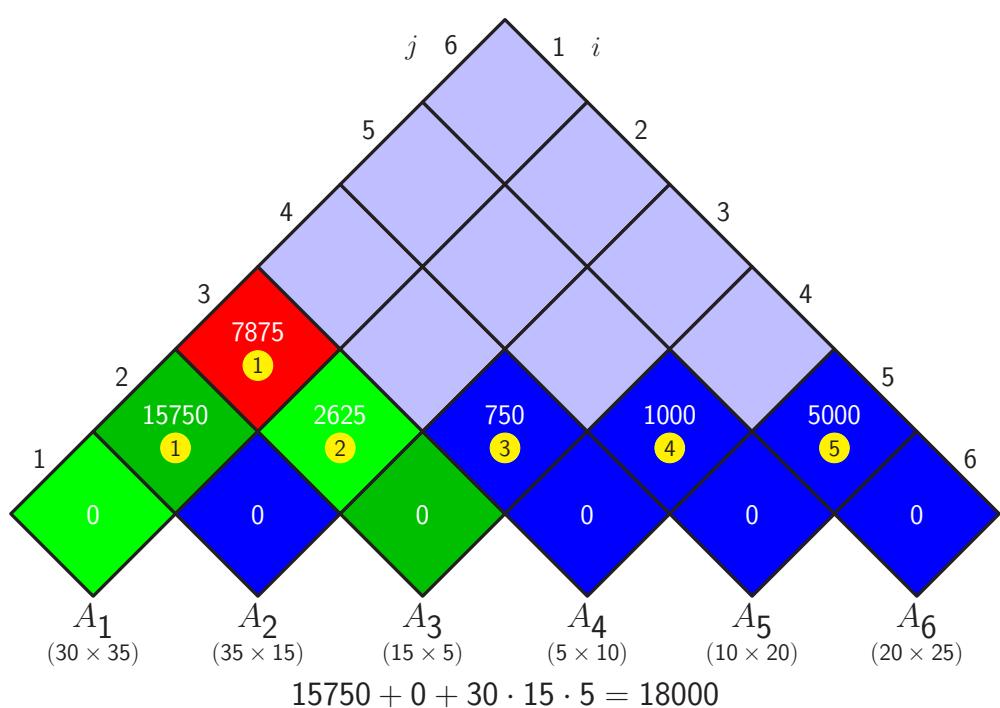
## Beispiel (Forts.)



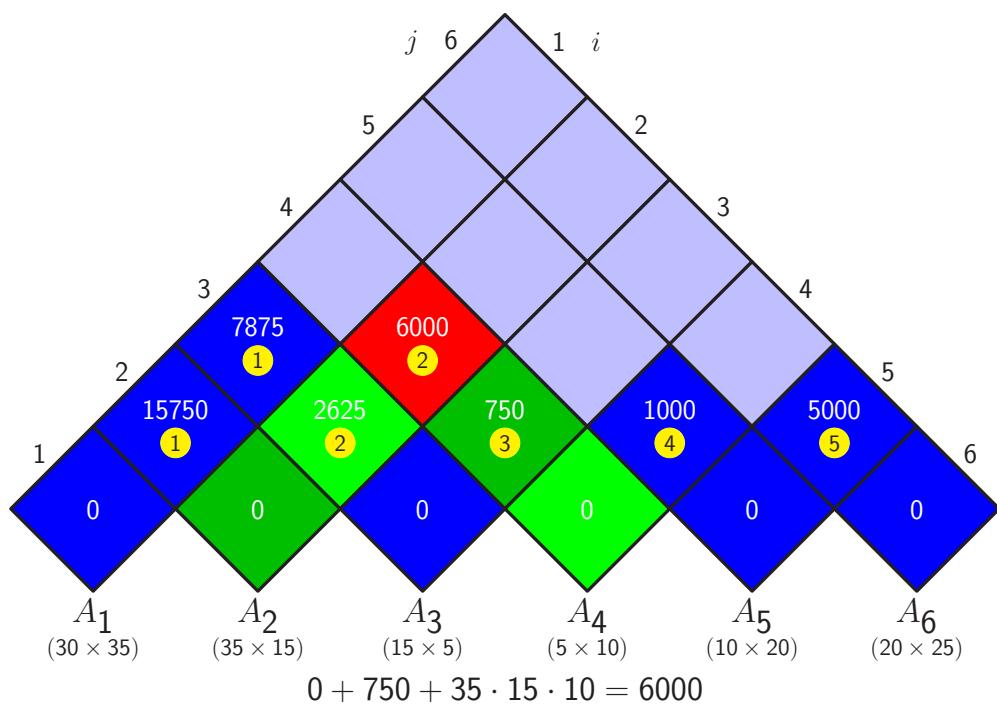
## Beispiel (Forts.)



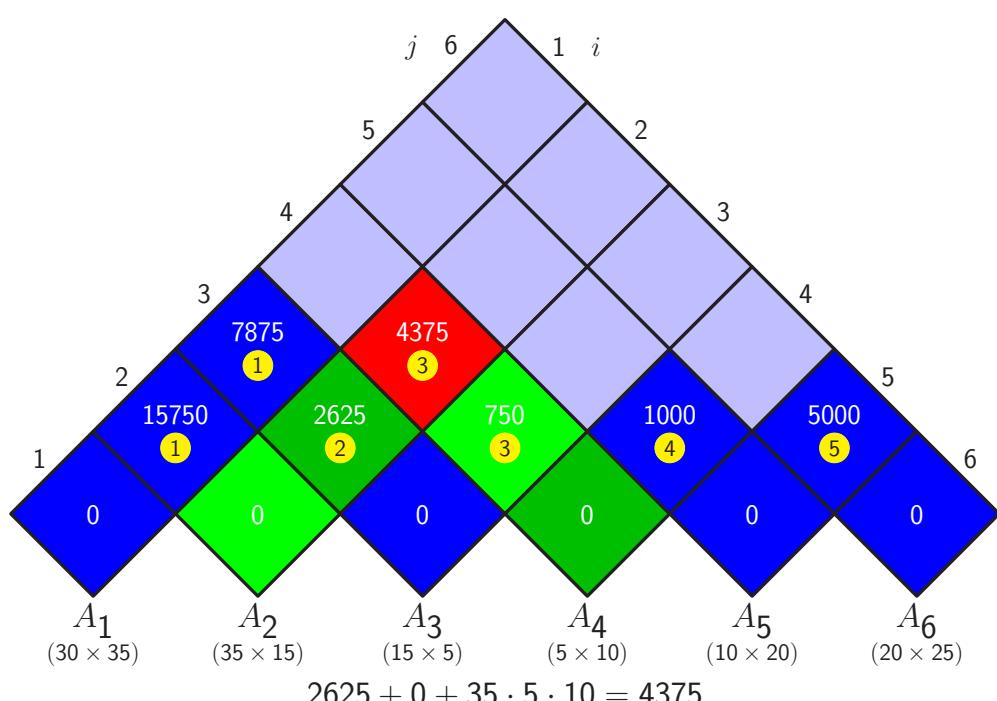
## Beispiel (Forts.)



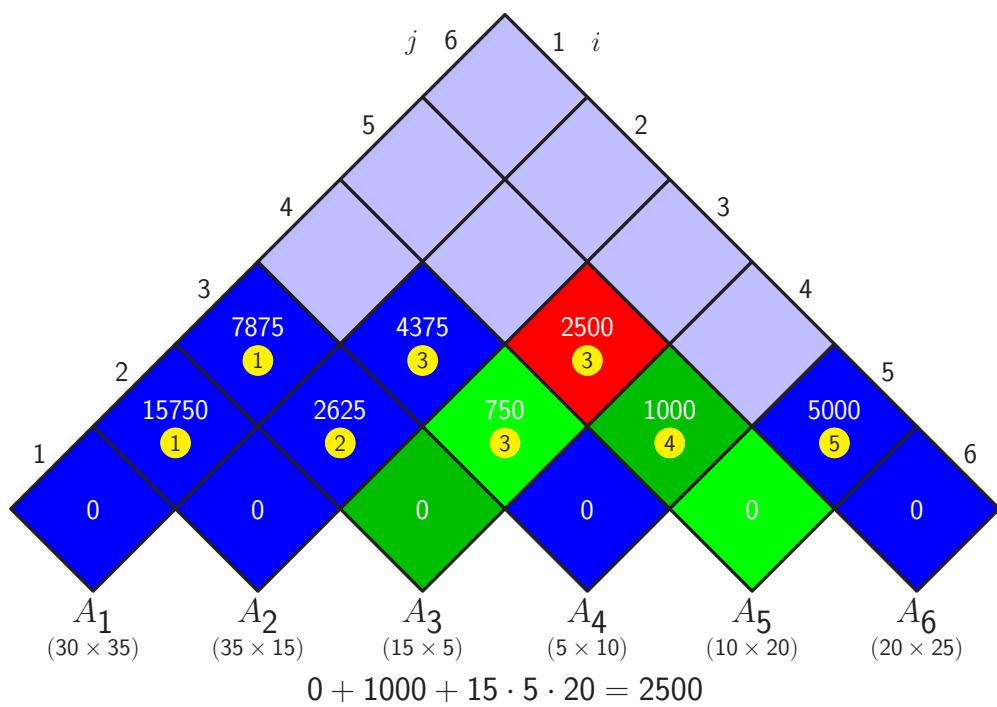
## Beispiel (Forts.)



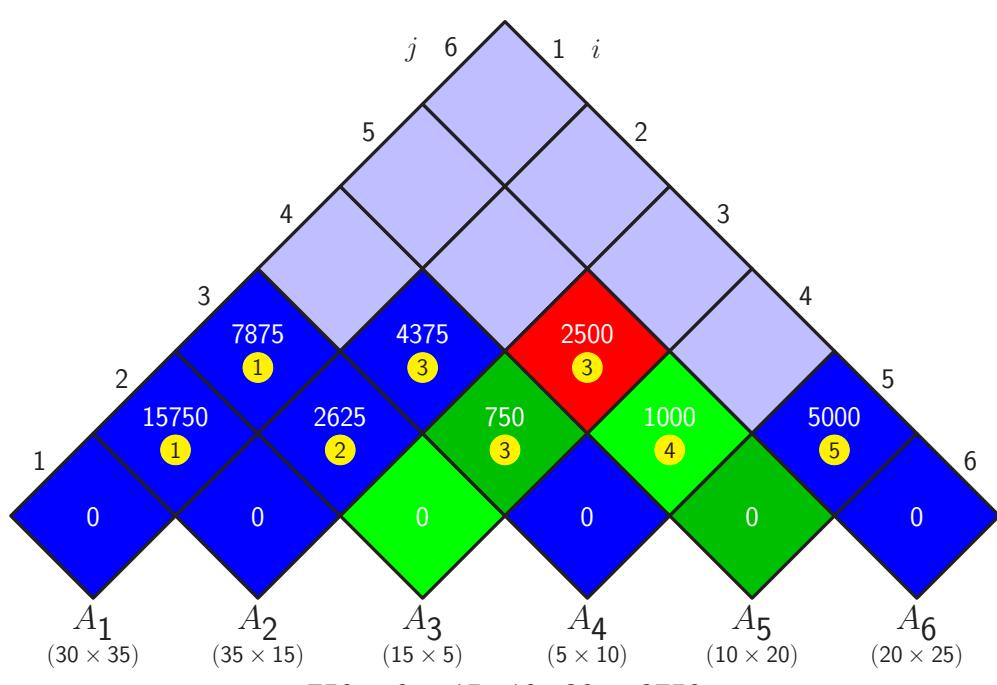
## Beispiel (Forts.)



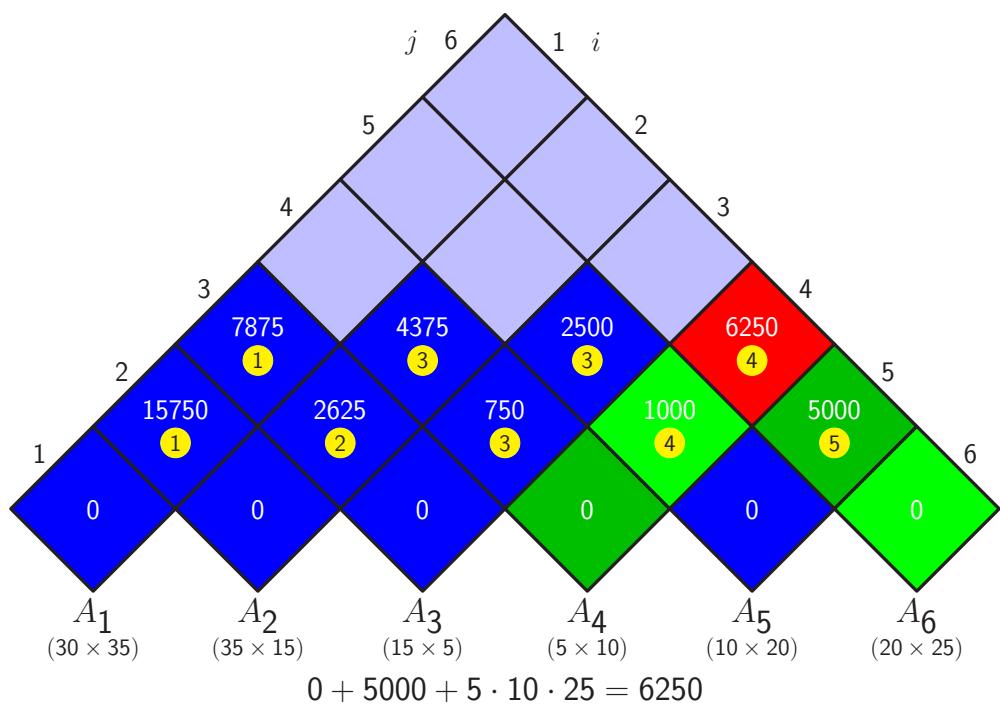
## Beispiel (Forts.)



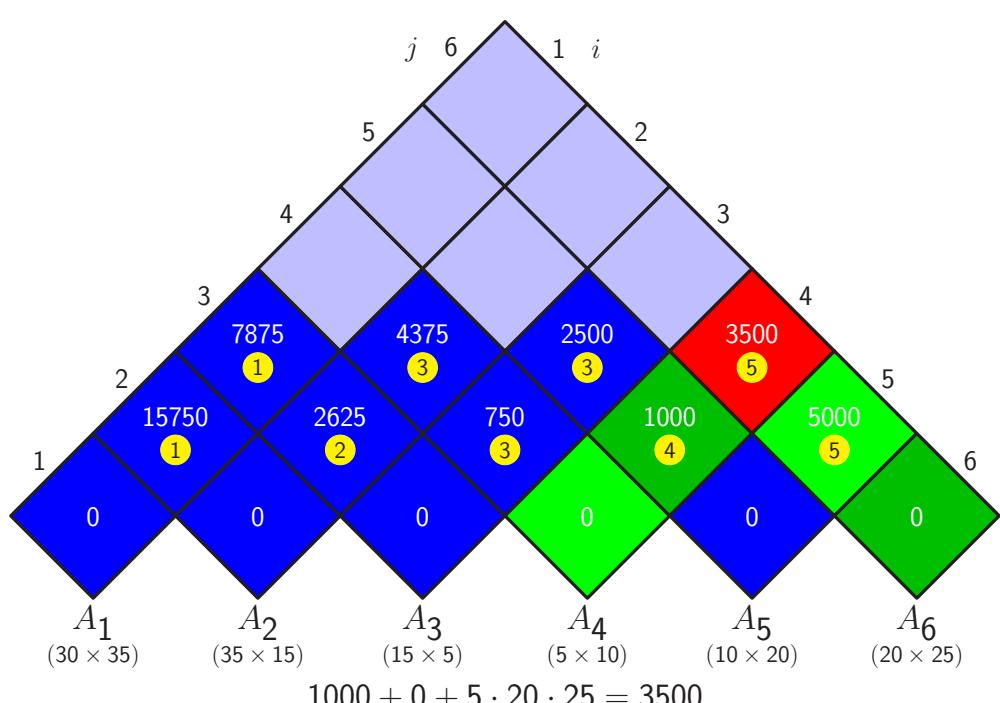
## Beispiel (Forts.)



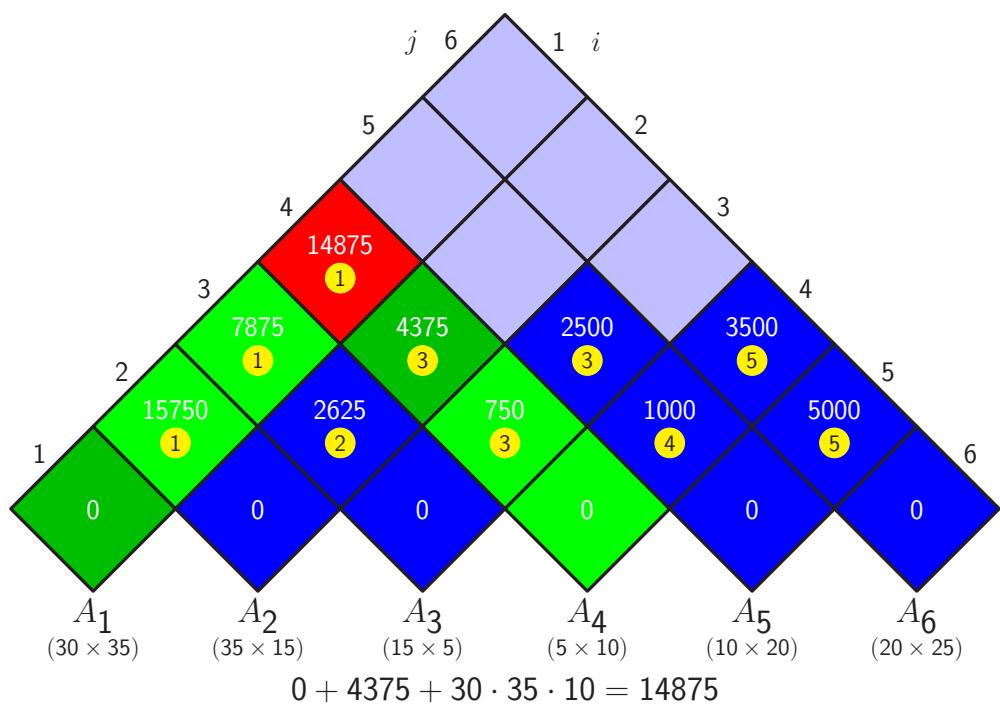
## Beispiel (Forts.)



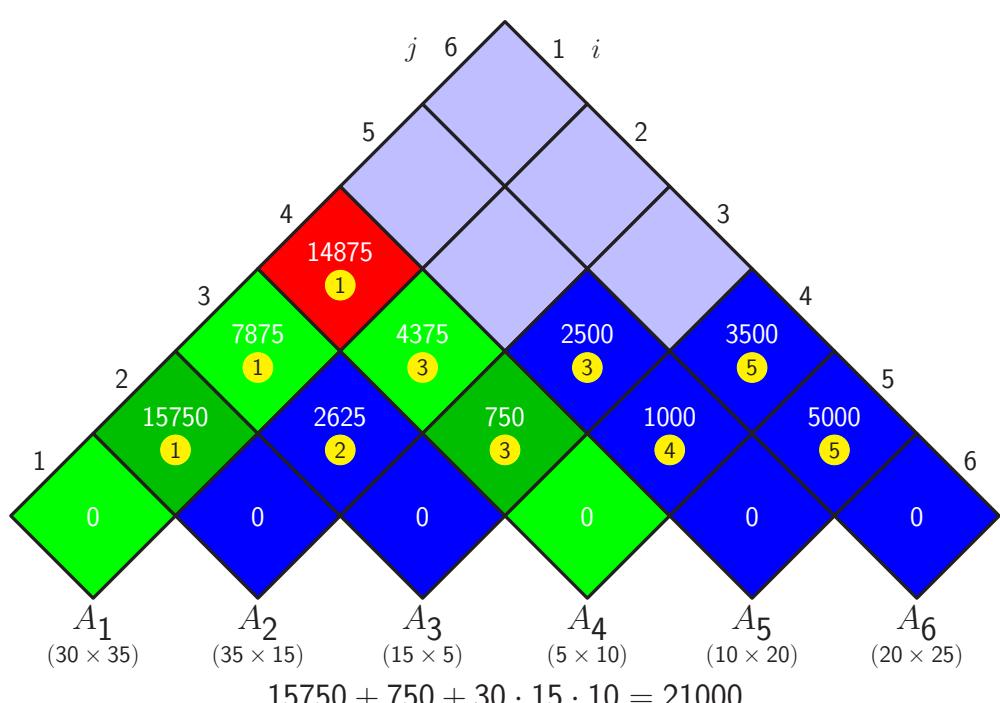
## Beispiel (Forts.)



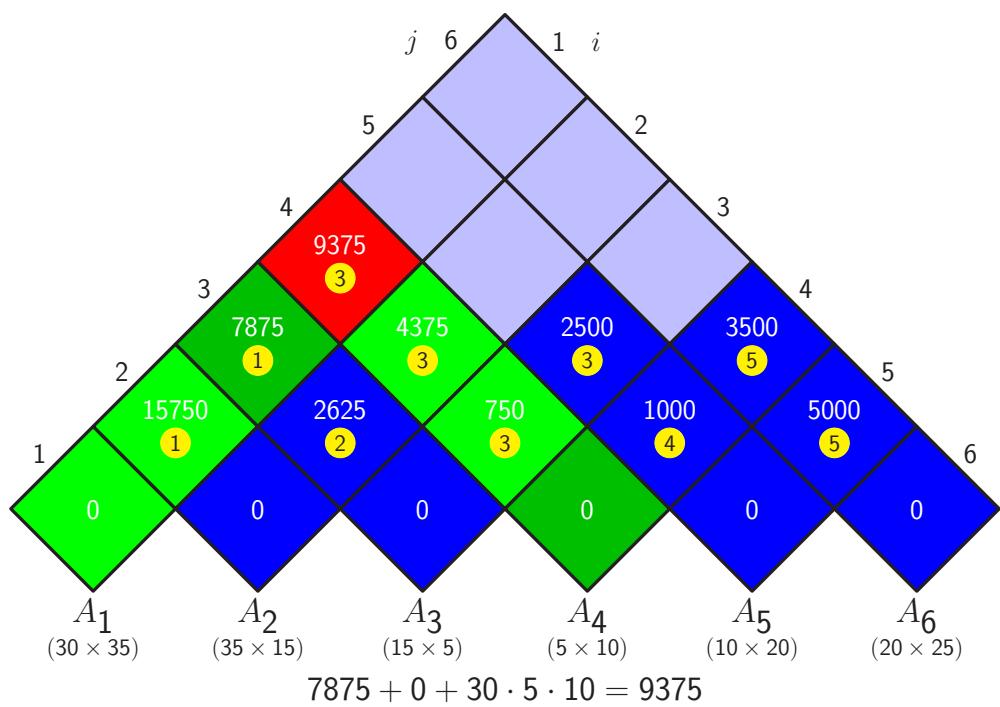
## Beispiel (Forts.)



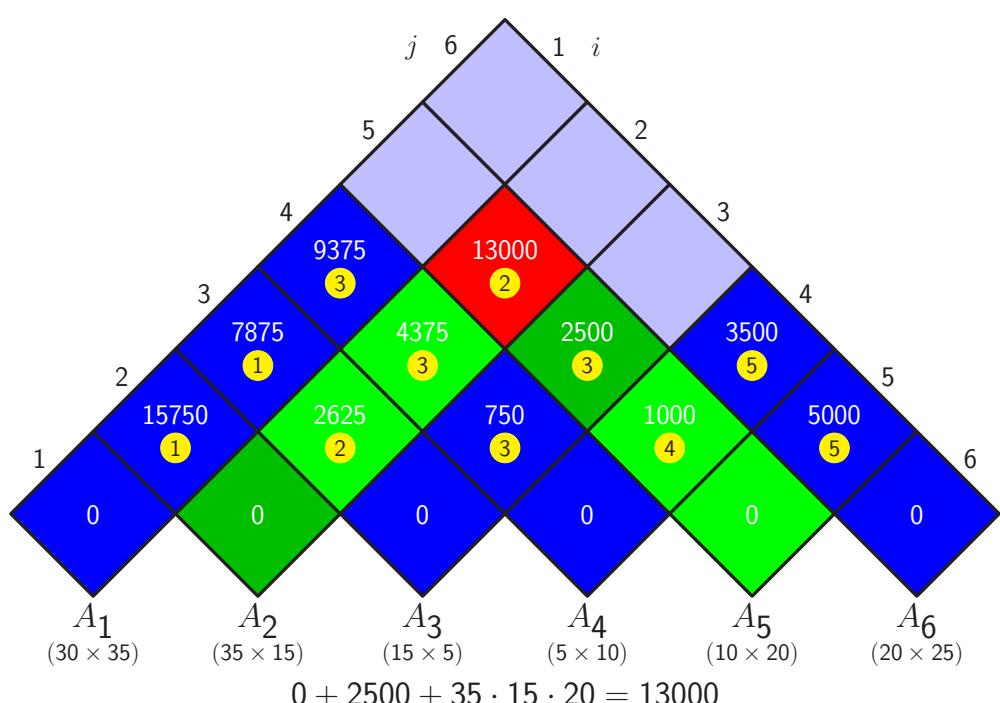
## Beispiel (Forts.)



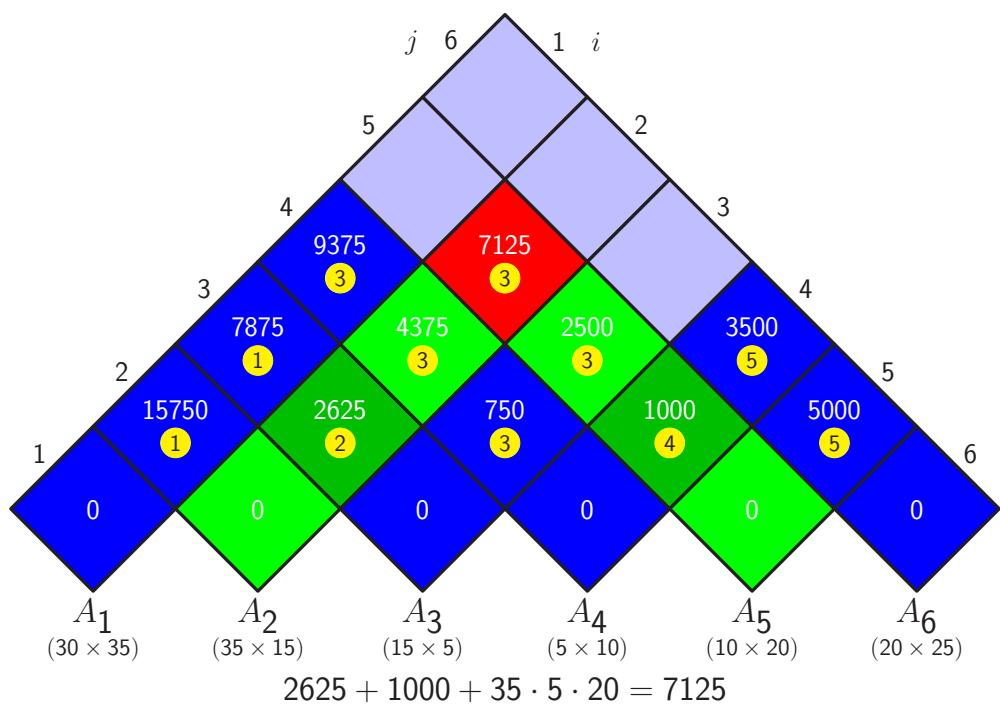
## Beispiel (Forts.)



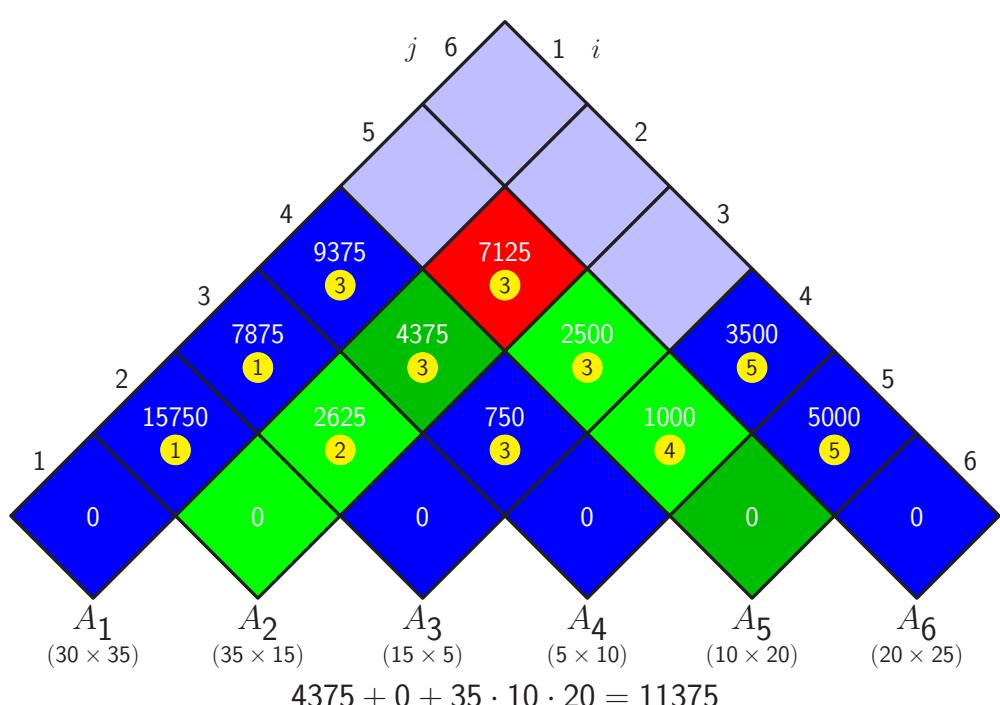
## Beispiel (Forts.)



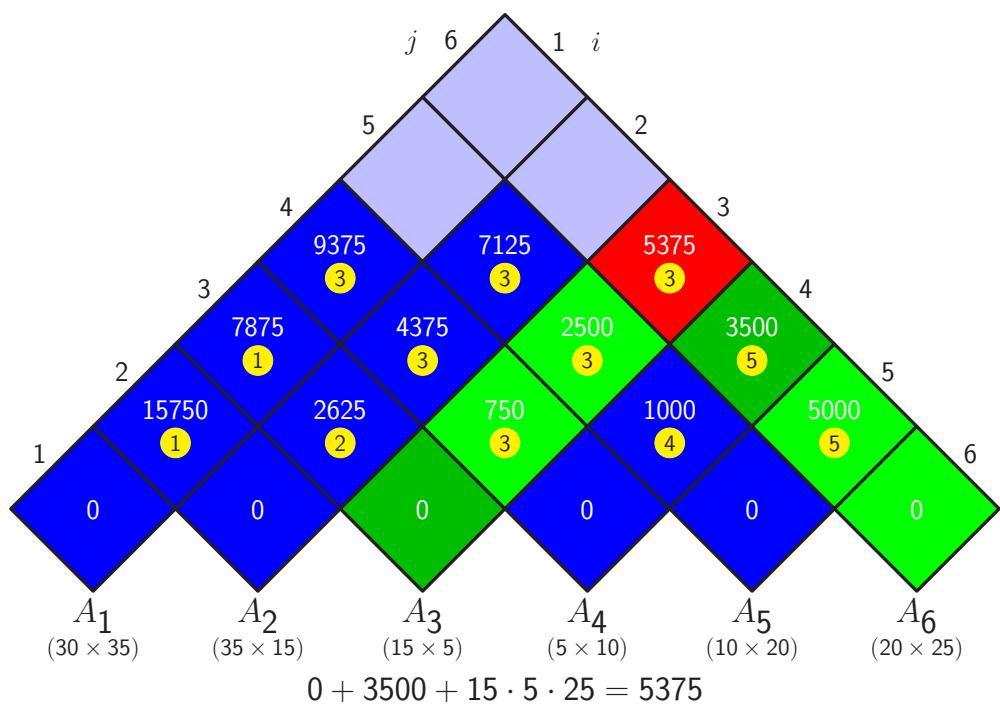
## Beispiel (Forts.)



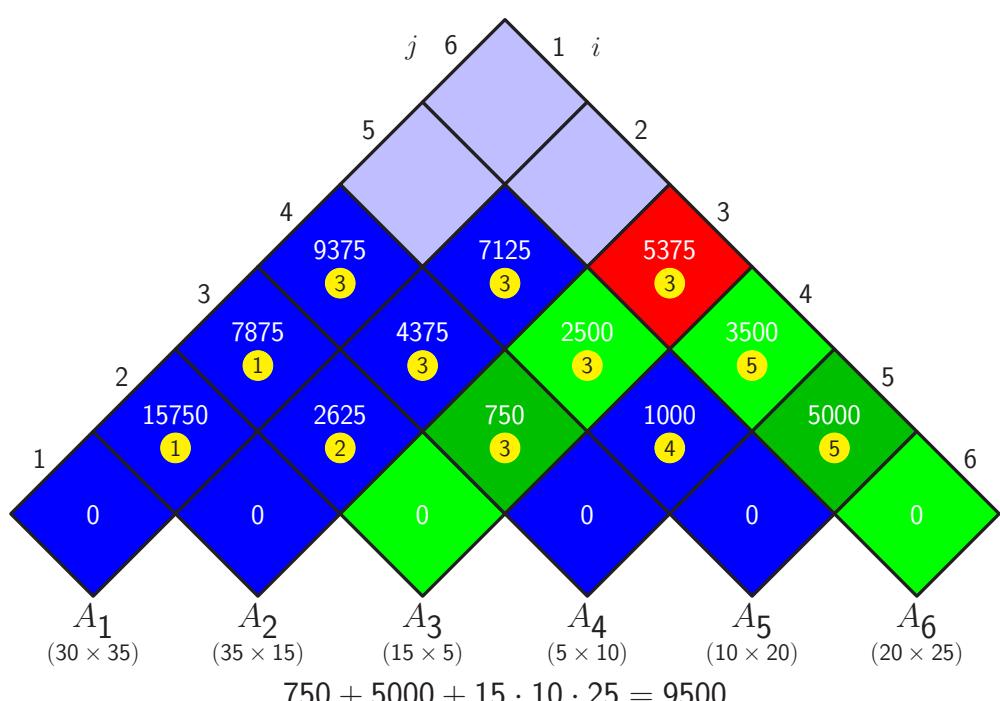
## Beispiel (Forts.)



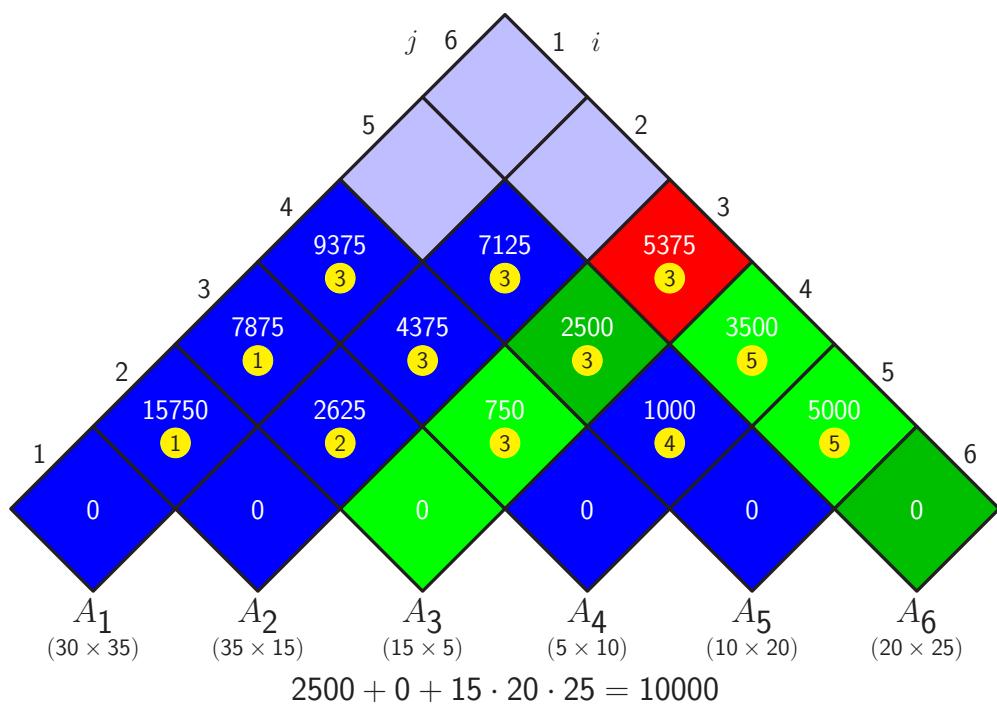
## Beispiel (Forts.)



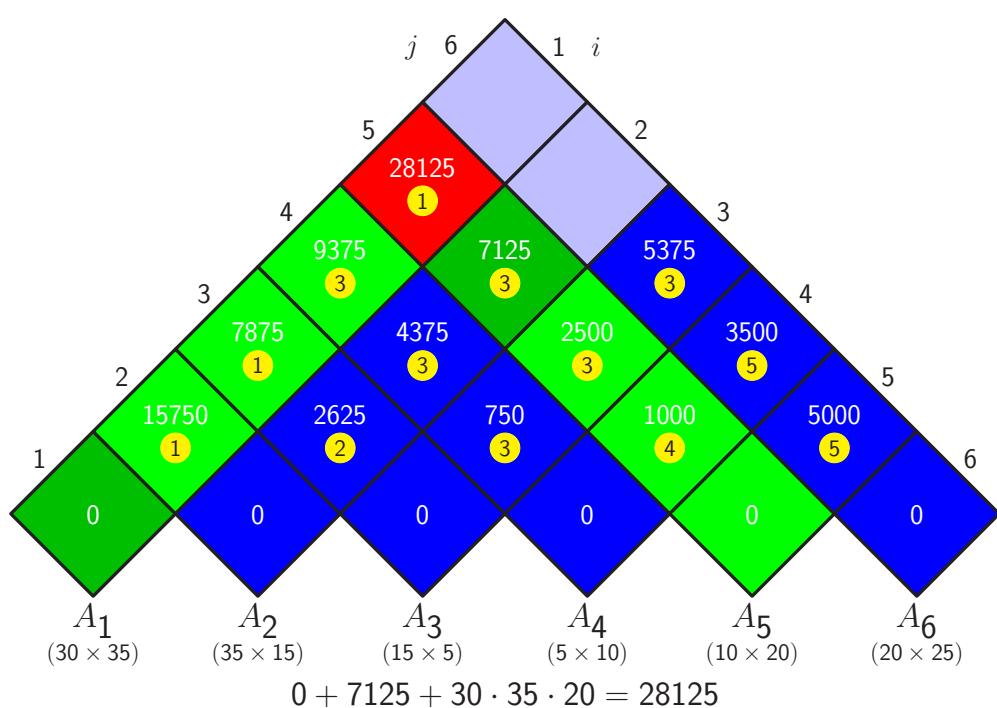
## Beispiel (Forts.)



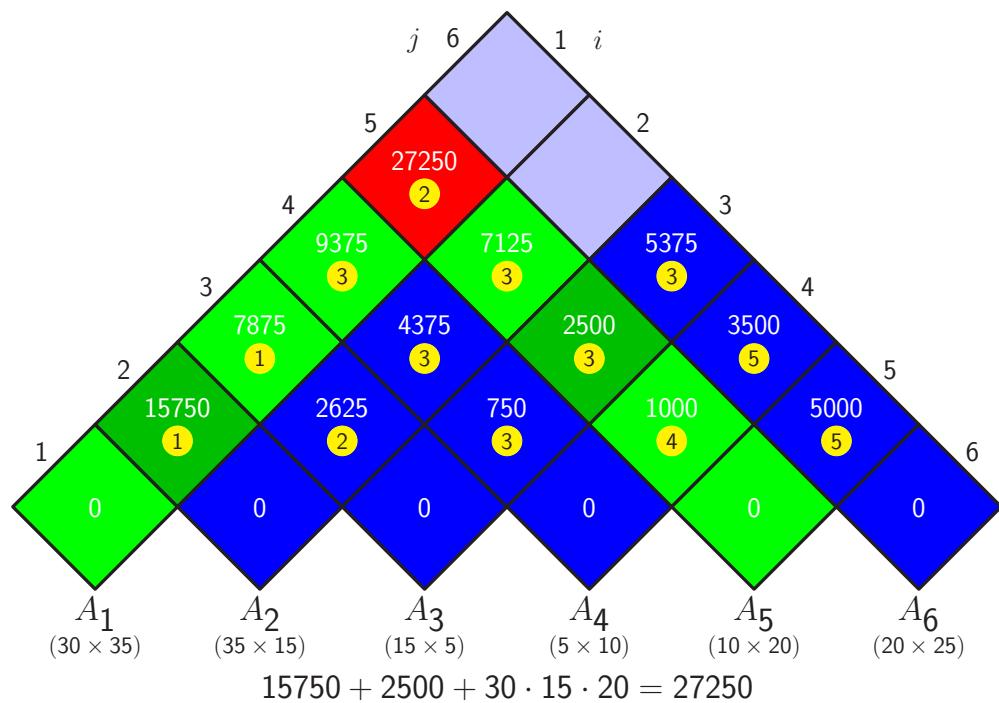
## Beispiel (Forts.)



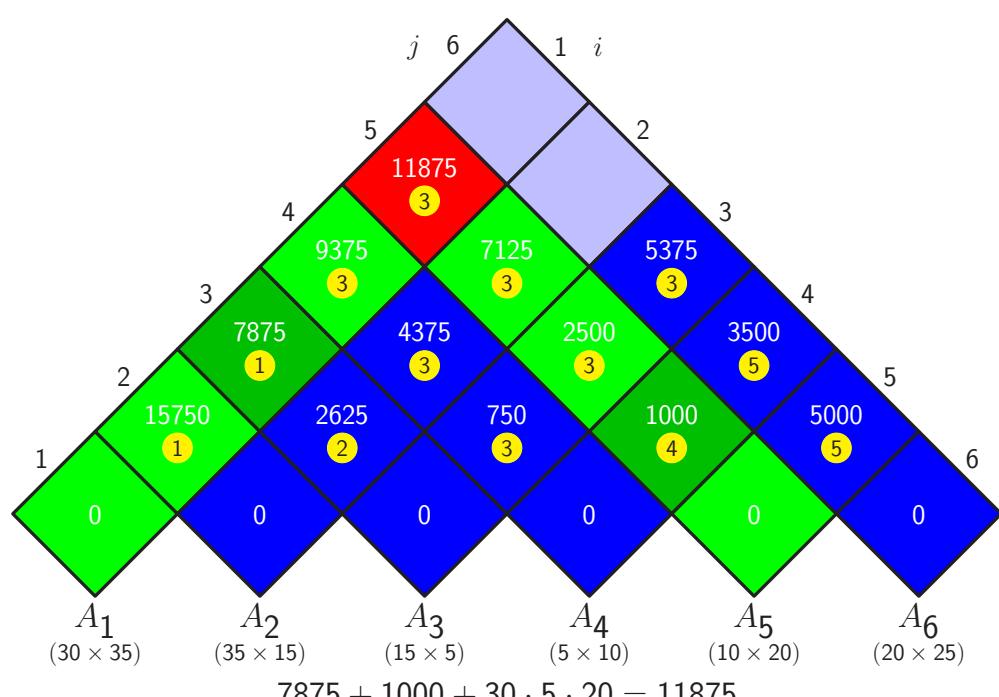
## Beispiel (Forts.)



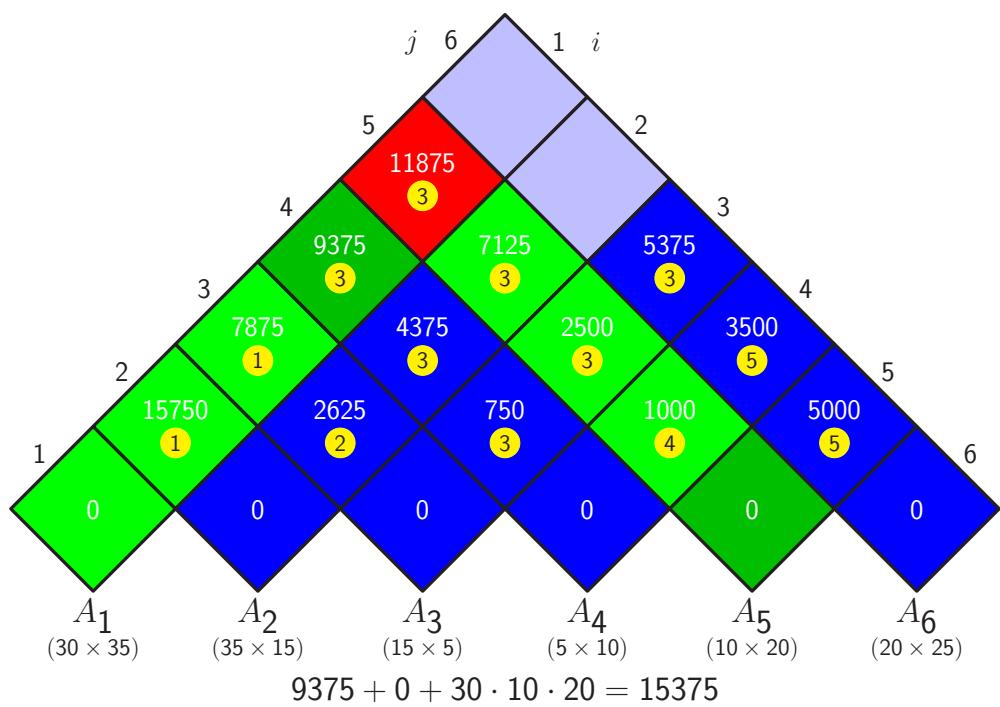
## Beispiel (Forts.)



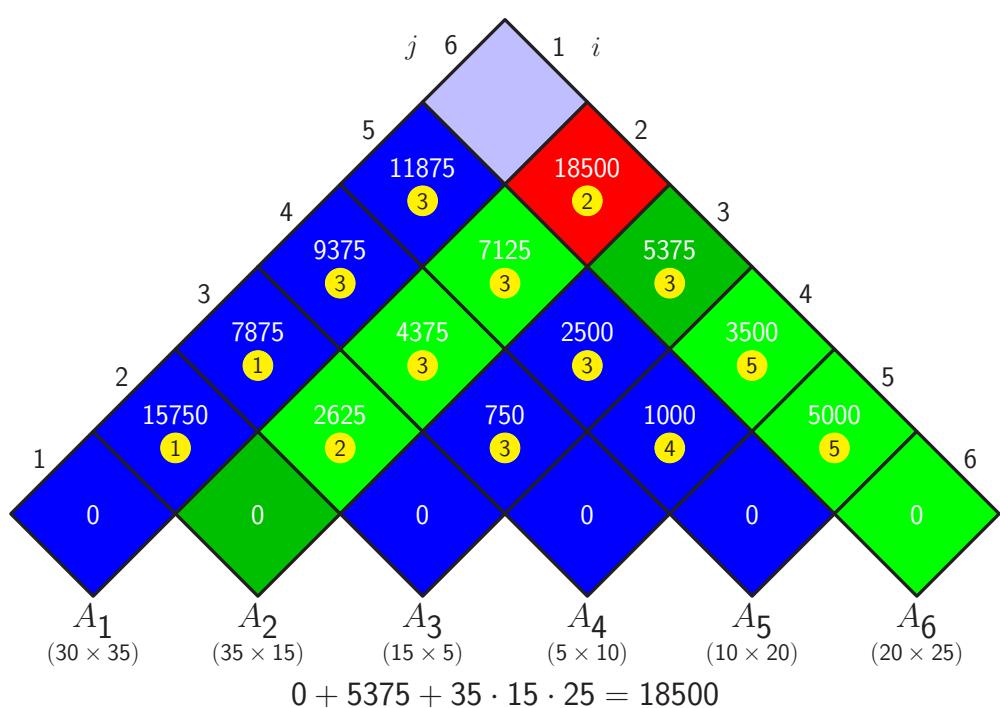
## Beispiel (Forts.)



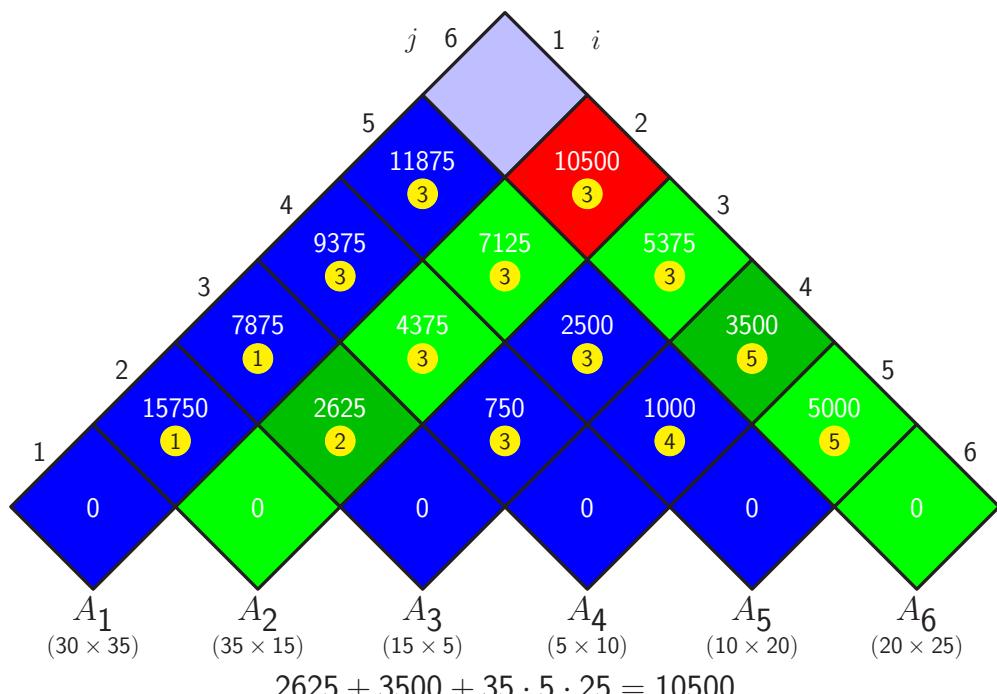
## Beispiel (Forts.)



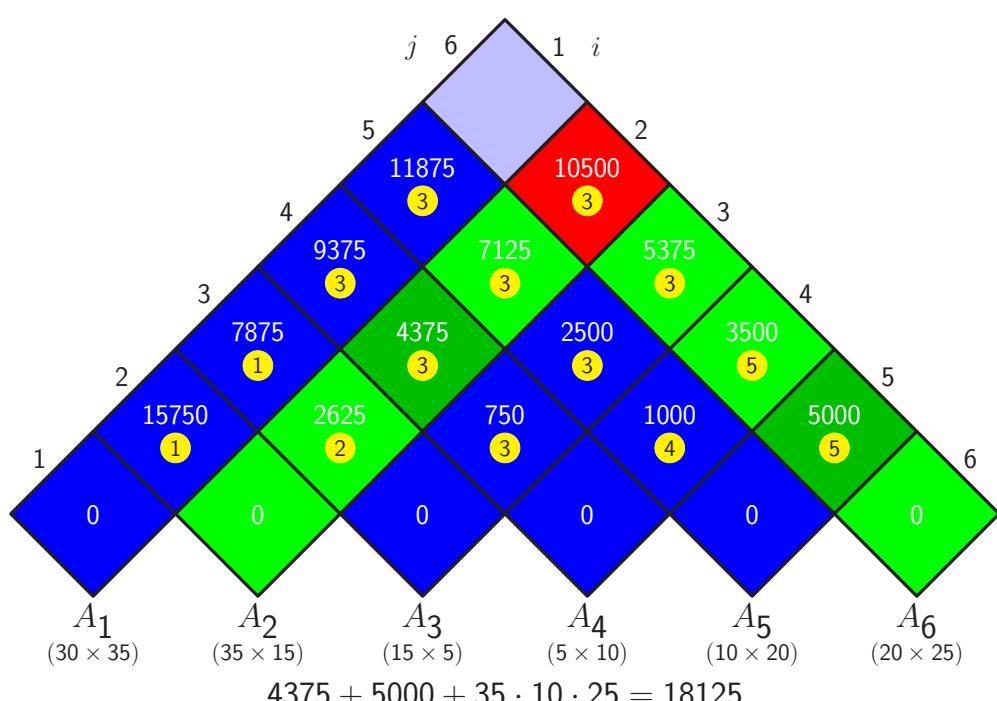
## Beispiel (Forts.)



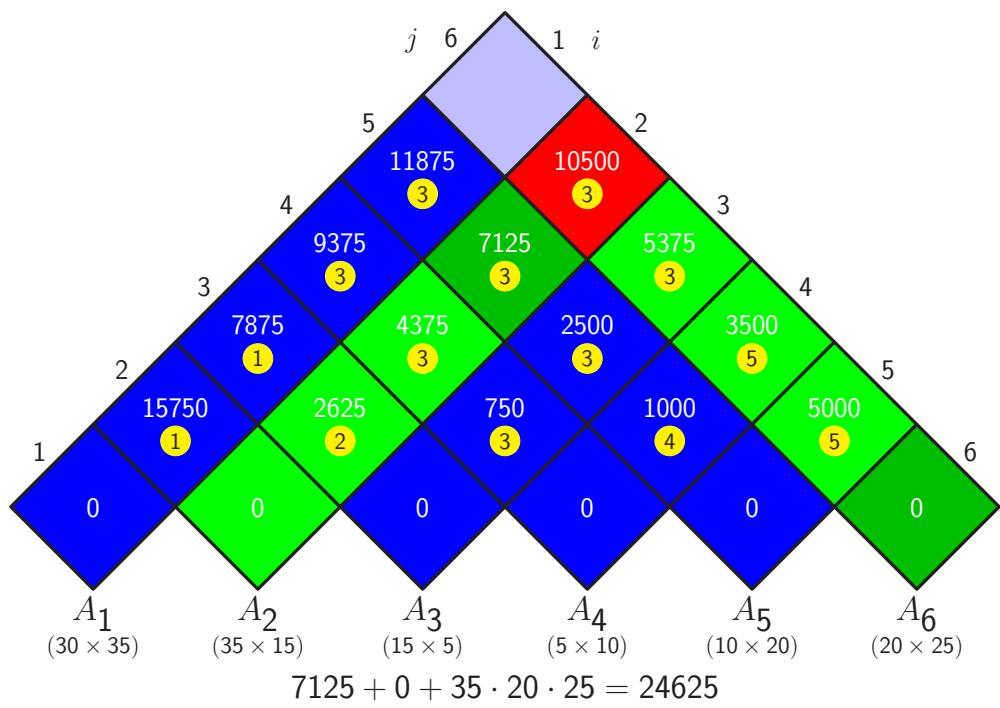
## Beispiel (Forts.)



## Beispiel (Forts.)



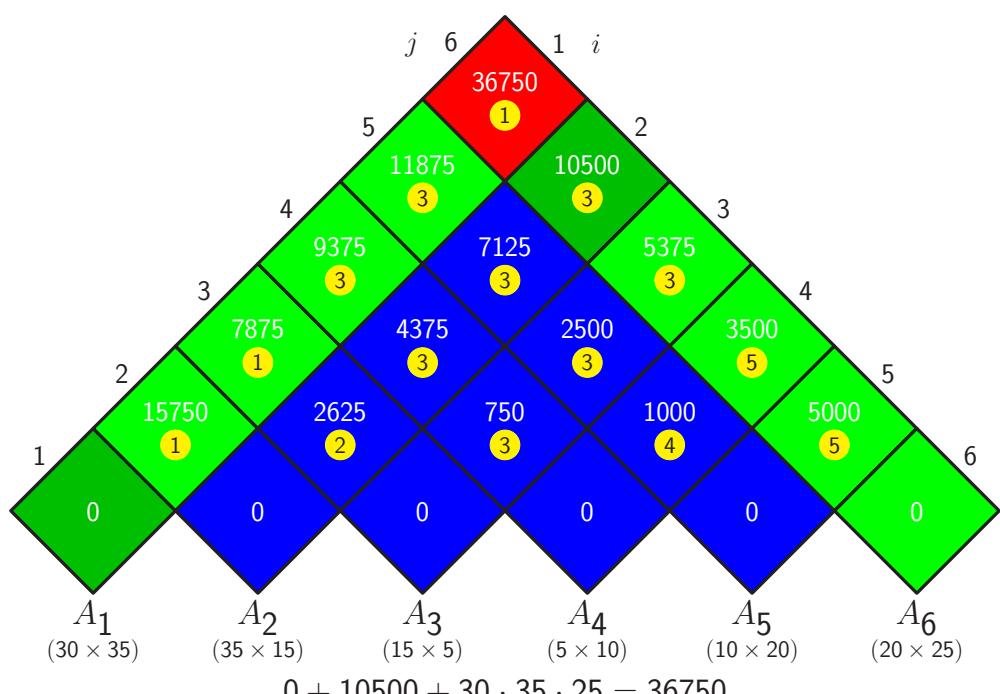
## Beispiel (Forts.)



Prof. Dr. C. Karg: Algorithmen und Datenstrukturen 2

71/80

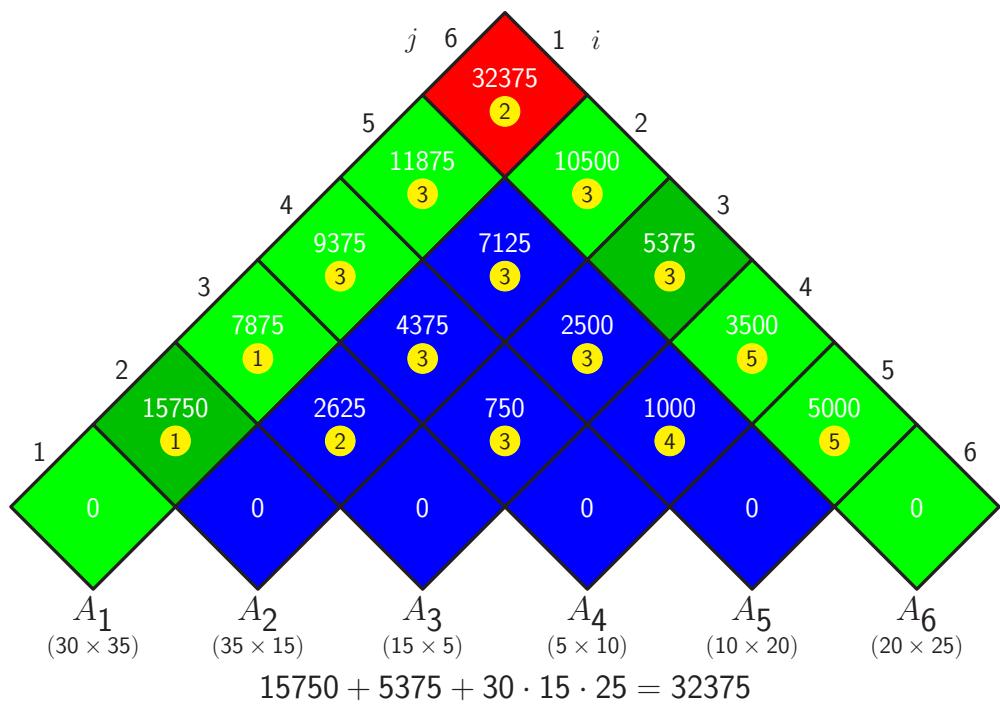
## Beispiel (Forts.)



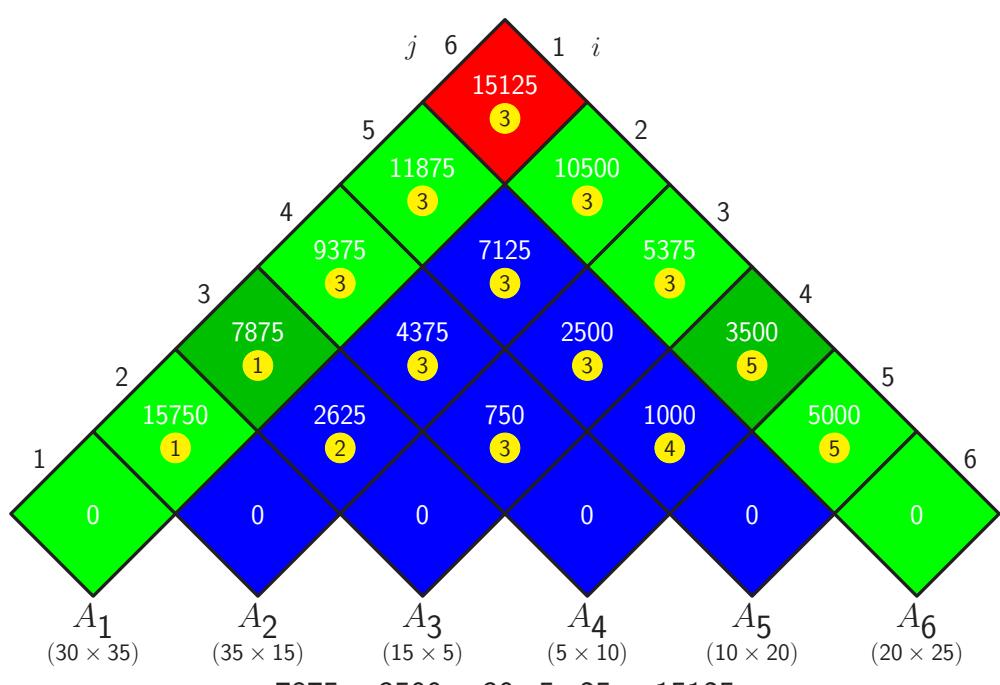
Prof. Dr. C. Karg: Algorithmen und Datenstrukturen 2

72 / 80

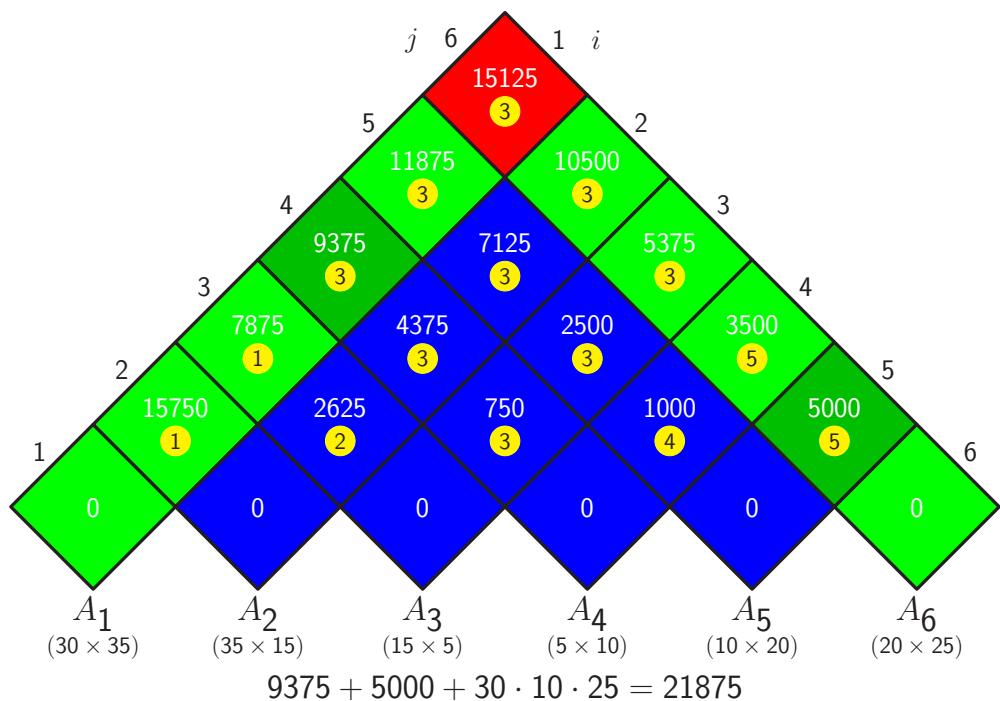
## Beispiel (Forts.)



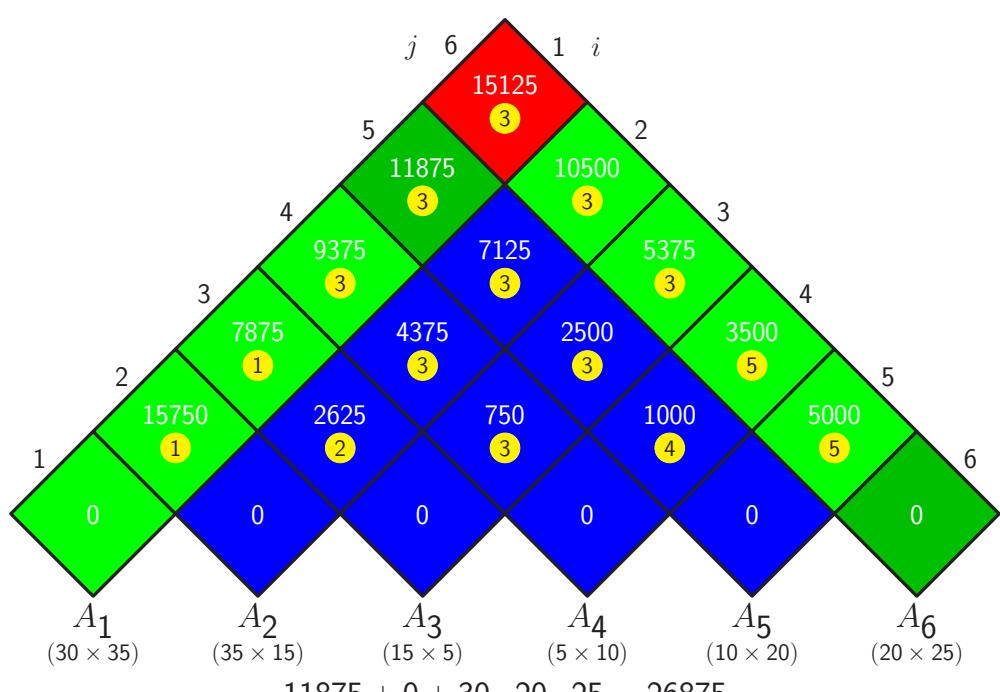
## Beispiel (Forts.)



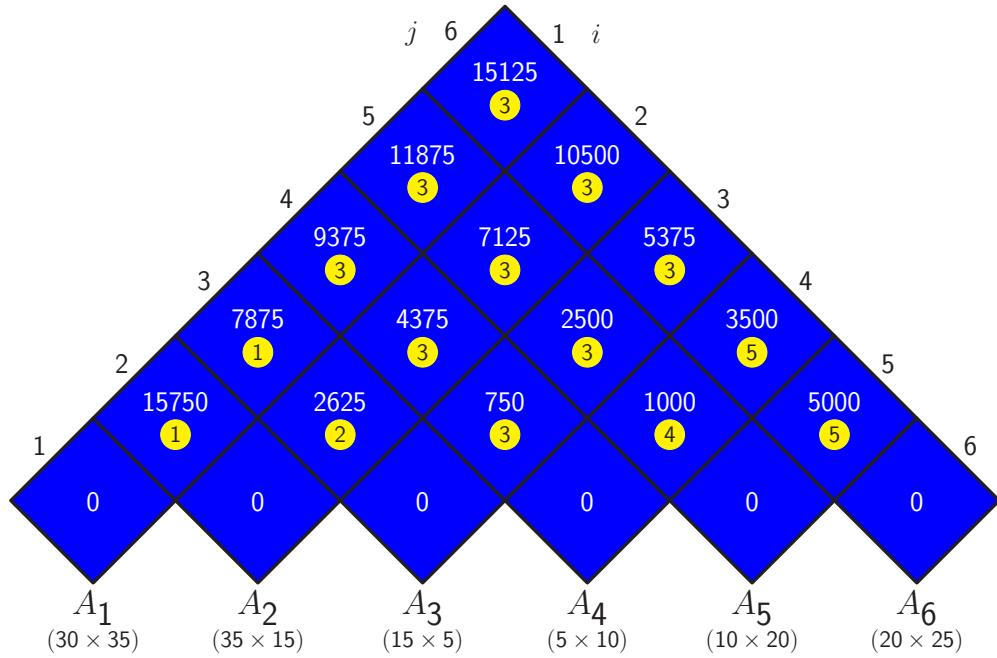
## Beispiel (Forts.)



## Beispiel (Forts.)



## Beispiel (Forts.)



## Beispiel (Forts.)

Berechnung der optimalen Klammerung:

$$\begin{aligned}
 A[1..6] &= (A[1..3])(A[4..6]) \quad \text{da } s[1,6]=3 \\
 &= ((A[1])(A[2..3]))(A[4..6]) \quad \text{da } s[1,3]=1 \\
 &= ((A[1])((A[2])(A[3])))((A[4..6])) \quad \text{da } s[2,3]=2 \\
 &= ((A[1])((A[2])(A[3])))(((A[4..5])(A[6]))) \quad \text{da } s[4,6]=5 \\
 &= ((A[1])((A[2])(A[3])))(((A[4])(A[5]))(A[6])) \quad \text{da } s[4,5]=4
 \end{aligned}$$

Bemerkung:  $A[i..j]$  steht für  $A_i \cdot \dots \cdot A_j$

**Optimale Klammerung:**  $((A_1(A_2A_3))((A_4A_5)A_6))$

## Anwendbarkeit von dynamischem Programmieren

Ein Optimierungsproblem ist mittels dynamischem Programmieren effizient lösbar, wenn es **zwei Eigenschaften** besitzt:

- **Optimale Teilstruktur:** Die optimale Lösung eines Problems enthält optimale Lösungen für die im Problem enthaltenen Teilprobleme
- **Überlappende Teilprobleme:** Die Menge der Teilprobleme zur rekursiven Berechnung der optimalen Lösung ist klein, d.h., polynomial in der Größe des Problems

## Zusammenfassung

- Dynamisches Programmieren ist eine Technik zur Bearbeitung von Optimierungsproblemen
- Die Technik besteht darin, eine Tabelle mit Teillösungen zu berechnen und auf Basis dieser eine optimale Lösung zu ermitteln
- Dynamisches Programmieren eignet sich für Optimierungsprobleme mit der “Optimale Teilstruktur” Eigenschaft
- Dynamisches Programmieren wird auch als Heuristik für schwer zu handhabende Optimierungsprobleme eingesetzt