KU LEUVEN kulak

Chapter 15: Dynamic Programming

(based on book “Introduction to Algorithms” of Cormen et al.)

Flor De Meulemeester

KU Leuven Kulak

Academiejaar 2024 – 2025

Overzicht

Overzicht

Fibonacci recursief

- We berekenen het n-de Fibonacci getal

Fibonacci recursief

- We berekenen het n-de Fibonacci getal
- Recursief is niet efficiënt maar hoe kunnnen we het beter maken?
- Doen we dubbel werk?

Fibonacci recursief

- We berekenen het n-de Fibonacci getal
- Recursief is niet efficiënt maar hoe kunnnen we het beter maken?
- Doen we dubbel werk?
- Idee: Oplossingen van overlappende subproblemen opslaan

Fibonacci recursief

- We berekenen het n-de Fibonacci getal
- Recursief is niet efficiënt maar hoe kunnnen we het beter maken?
- Doen we dubbel werk?
- Idee: Oplossingen van overlappende subproblemen opslaan
- Gelijkaardig met *divide and conquer*

Fibonacci recursief

- We berekenen het n-de Fibonacci getal
- Recursief is niet efficiënt maar hoe kunnnen we het beter maken?
- Doen we dubbel werk?
- Idee: Oplossingen van overlappende subproblemen opslaan
- Gelijkaardig met *divide and conquer*
- Techniek vooral goed voor optimalisatie problemen (bepaalde waarde maximaliseren/minimaliseren)

Stappen plan voor *dynamic programming*

1. Bepaal de **structuur** van de oplossing
2. Bepaal de **recursieve** relatie van een waarde van een oplossing t.o.v. die van de subproblemen
3. Bereken de **waarde** van de optimale oplossing
4. Construeer het **pad** naar de optimale oplossing

Overzicht

Stap 1: Probleem voorstellen

- Gegeven volgende tabel:

lengte i	1	2	3	4	5	6	7	8	9	10
prijs p_i	1	5	8	9	10	17	17	20	24	30

- Een stuk met lengte i wordt verkocht aan prijs p_i

Stap 1: Probleem voorstellen

- Gegeven volgende tabel:

lengte i	1	2	3	4	5	6	7	8	9	10
prijs p_i	1	5	8	9	10	17	17	20	24	30

- Een stuk met lengte i wordt verkocht aan prijs p_i
- Verdeel staaf van lengte n in $1 \leq k \leq n$ stukken zodat de prijs maximaal is

Stap 1: Probleem voorstellen

- Gegeven volgende tabel:

lengte i	1	2	3	4	5	6	7	8	9	10
prijs p_i	1	5	8	9	10	17	17	20	24	30

- Een stuk met lengte i wordt verkocht aan prijs p_i
- Verdeel staaf van lengte n in $1 \leq k \leq n$ stukken zodat de prijs maximaal is
- De lengte is $n = i_1 + i_2 + ... + i_k$

Stap 1: Probleem voorstellen

- Gegeven volgende tabel:

lengte i	1	2	3	4	5	6	7	8	9	10
prijs p_i	1	5	8	9	10	17	17	20	24	30

- Een stuk met lengte i wordt verkocht aan prijs p_i
- Verdeel staaf van lengte n in $1 \leq k \leq n$ stukken zodat de prijs maximaal is
- De lengte is $n = i_1 + i_2 + ... + i_k$
- Totale prijs is $p = p_{i_1} + p_{i_2} + ... + p_{i_k}$

Stap 1: Probleem voorstellen

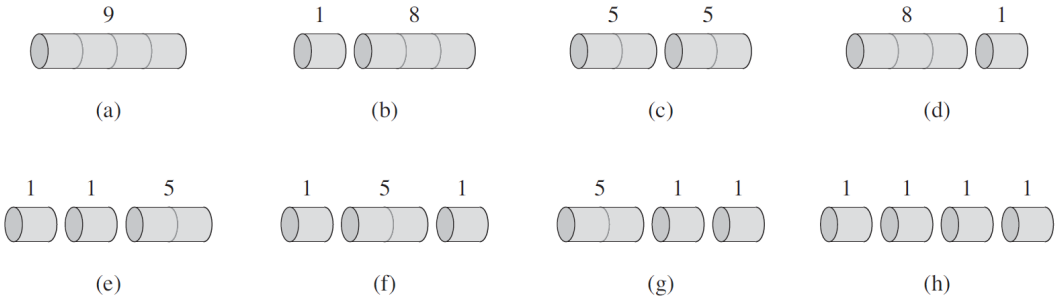
- Gegeven volgende tabel:

lengte i	1	2	3	4	5	6	7	8	9	10
prijs p_i	1	5	8	9	10	17	17	20	24	30

- Een stuk met lengte i wordt verkocht aan prijs p_i
- Verdeel staaf van lengte n in $1 \leq k \leq n$ stukken zodat de prijs maximaal is
- De lengte is $n = i_1 + i_2 + ... + i_k$
- Totale prijs is $p = p_{i_1} + p_{i_2} + ... + p_{i_k}$
- Bijvoorbeeld: Een stuk met lengte $n = 4$ kunnen we verdelen in 2 stukken van lengte 2 en dus $p_2 + p_2 = 5 + 5 = 10$ is de prijs.

Voorbeeld

Alle mogelijke onderverdelingen voor $n = 4$:



Stap 2: Recursieve oplossing

- We kiezen een index i waarop we snijden maximaal en gaan recursief voor op het deel dat over blijft.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Stap 2: Recursieve oplossing

- We kiezen een index i waarop we snijden maximaal en gaan recursief voor op het deel dat over blijft.

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

- Recurrentie relatie: $T(0) = 1$ en $T(n) = 1 + \sum_{j=0}^{n-1} 1 + T(j) = 2^n$

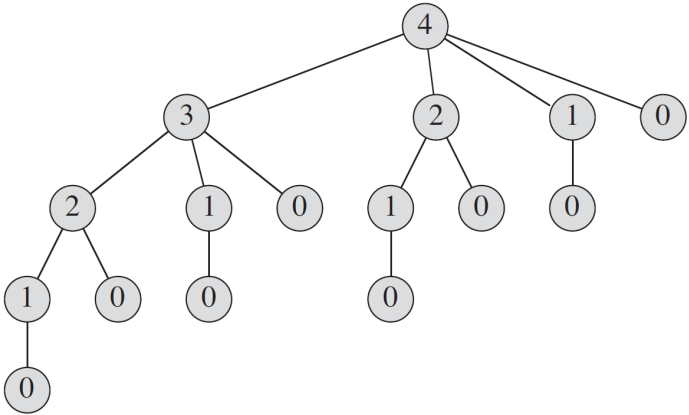
Stap 2: Recursieve oplossing

- We kiezen een index i waarop we snijden maximaal en gaan recursief voor op het deel dat over blijft.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

- Recurrentie relatie: $T(0) = 1$ en $T(n) = 1 + \sum_{j=0}^{n-1} 1 + T(j) = 2^n$
- We hebben dus een algoritme met **exponentiële** uitvoeringstijd, $\mathcal{O}(2^n)$.

Recursion tree



Stap 3: DP toepassen

- DP zal in polynomiale tijd werken als er een polynomiaal aantal verschillende subproblemen bestaan.

Stap 3: DP toepassen

- DP zal in polynomiale tijd werken als er een polynomiaal aantal verschillende subproblemen bestaan.
- **Time-memory trade-off**

Stap 3: DP toepassen

- DP zal in polynomiale tijd werken als er een polynomiaal aantal verschillende subproblemen bestaan.
- **Time-memory trade-off**
- We kunnen kiezen uit 2 implementaties
  1. **Top-down** with memoization
  2. **Bottom-up** method

Top-down implementatie

- We beginnen met ons probleem van groote n, als we subprobleem nodig hebben berekenen we het eenmalig en slaan het op.

Top-down implementatie

- We beginnen met ons probleem van groote n, als we subprobleem nodig hebben berekenen we het eenmalig en slaan het op.
- Idee: we hebben array (waarde ophalen in $\mathcal{O}(1)$) met de resultaten van subproblemen.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

Top-down implementatie

- We beginnen met ons probleem van groote n, als we subprobleem nodig hebben berekenen we het eenmalig en slaan het op.
- Idee: we hebben array (waarde ophalen in $\mathcal{O}(1)$) met de resultaten van subproblemen.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

- We hebben dus een algoritme met **polynomiale** uitvoeringstijd, $\Theta(n^2)$.

Bottom-up implementatie

- We berekenen alle subproblemen klein naar groot

Bottom-up implementatie

- We berekenen alle subproblemen klein naar groot
- Opnieuw met array (waarde ophalen in $\mathcal{O}(1)$) met de resultaten van subproblemen.

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

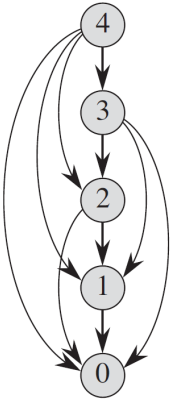
Bottom-up implementatie

- We berekenen alle subproblemen klein naar groot
- Opnieuw met array (waarde ophalen in $\mathcal{O}(1)$) met de resultaten van subproblemen.

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

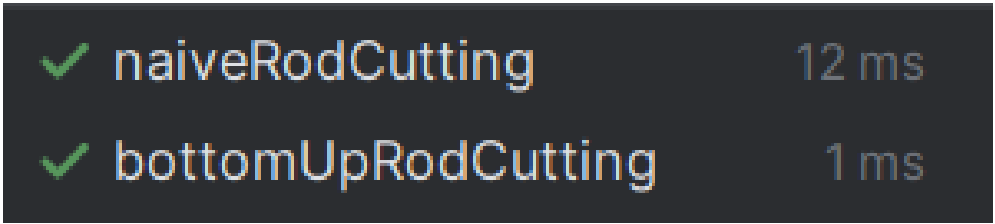
- We hebben dus een algoritme met **polynomiale** uitvoeringstijd, $\Theta(n^2)$.

Subproblem graph



Java implementatie

- Voor $n = 1, ..., 10$



Stap 4: De verdeling voor de optimale waarde

- We passen het algoritme lichtjes aan om de gemaakte stappen bij te houden.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

```
while n > 0
    print s[n]
    n = n - s[n]
```

Overzicht

Stap 1: Probleem voorstellen

- Gegeven volgende tabel:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensie	30×35	35×15	15×5	5×10	10×20	20×25

Stap 1: Probleem voorstellen

- Gegeven volgende tabel:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensie	30×35	35×15	15×5	5×10	10×20	20×25

- Matrices niet commutatief wel associatief, we willen haakjes zo efficiënt mogelijk zetten zodat we minimaal aantal scalair vermenigvuldigingen moeten berekenen.

Stap 1: Probleem voorstellen

- Gegeven volgende tabel:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensie	30×35	35×15	15×5	5×10	10×20	20×25

- Matrices niet commutatief wel associatief, we willen haakjes zo efficiënt mogelijk zetten zodat we minimaal aantal scalair vermenigvuldigingen moeten berekenen.
- Voor een matrix van $p \times q$ en een matrix $q \times r$ is dit pqr aantal bewerkingen.

Voorbeeldje

- $A_1 A_2 A_3 A_4$

Mogelijkheden:

Voorbeeldje

- $A_1 A_2 A_3 A_4$

Mogelijkheden:

- $(A_1(A_2(A_3 A_4)))$
- $(A_1((A_2 A_3)A_4))$
- $((A_1 A_2)(A_3 A_4))$
- $((A_1(A_2 A_3))A_4)$
- $(((A_1 A_2)A_3)A_4)$

Voorbeeldje

- $A_1 A_2 A_3 A_4$

Mogelijkheden:

- $(A_1(A_2(A_3 A_4)))$
- $(A_1((A_2 A_3)A_4))$
- $((A_1 A_2)(A_3 A_4))$
- $((A_1(A_2 A_3))A_4)$
- $(((A_1 A_2)A_3)A_4)$

- $P(n) = \#$ mogelijkheden voor n

$$P(1) = 1$$

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \text{ met } n \geq 2$$

Voorbeeldje

- $A_1 A_2 A_3 A_4$

Mogelijkheden:

- $(A_1(A_2(A_3 A_4)))$
- $(A_1((A_2 A_3)A_4))$
- $((A_1 A_2)(A_3 A_4))$
- $((A_1(A_2 A_3))A_4)$
- $(((A_1 A_2)A_3)A_4)$

- $P(n) = \#$ mogelijkheden voor n

$$P(1) = 1$$

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \text{ met } n \geq 2$$

- $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, \ldots$

Voorbeeldje

- $A_1 A_2 A_3 A_4$

Mogelijkheden:

- $(A_1(A_2(A_3A_4)))$
- $(A_1((A_2A_3)A_4))$
- $((A_1A_2)(A_3A_4))$
- $((A_1(A_2A_3))A_4)$
- $(((A_1A_2)A_3)A_4)$

- $P(n) = \#$ mogelijkheden voor n

$$P(1) = 1$$

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \text{ met } n \geq 2$$

- $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, \ldots$
- De Catalan-getallen groeien met $\Omega(4^n/n^{\frac{3}{2}})$

Stap 2: Recursieve oplossing

- We zullen een subprobleem $A_i A_{i+1} ... A_j$ als volgt opsplitsen in 2 nieuwe subproblemen $A_i A_{i+1} ... A_k$ en $A_{k+1} A_{k+2} ... A_j$.

Stap 2: Recursieve oplossing

- We zullen een subprobleem $A_i A_{i+1} ... A_j$ als volgt opsplitsen in 2 nieuwe subproblemen $A_i A_{i+1} ... A_k$ en $A_{k+1} A_{k+2} ... A_j$.
- We zetten haakjes achter A_k

Stap 2: Recursieve oplossing

- We zullen een subprobleem $A_iA_{i+1}...A_j$ als volgt opsplitsen in 2 nieuwe subproblemen $A_iA_{i+1}...A_k$ en $A_{k+1}A_{k+2}...A_j$.
- We zetten haakjes achter A_k
- De oplossingen van deze subproblemen zullen we bijhouden in een $n \times n$ matrix m. De oplossing voor $A_iA_{i+1}...A_{j-1}A_j$ vinden we dan in $m[i, j]$.

Stap 2: Recursieve oplossing

- We zullen een subprobleem $A_iA_{i+1}...A_j$ als volgt opsplitsen in 2 nieuwe subproblemen $A_iA_{i+1}...A_k$ en $A_{k+1}A_{k+2}...A_j$.
- We zetten haakjes achter A_k
- De oplossingen van deze subproblemen zullen we bijhouden in een $n \times n$ matrix m. De oplossing voor $A_iA_{i+1}...A_{j-1}A_j$ vinden we dan in $m[i, j]$.
- $m[i, i] = 0$

Stap 2: Recursieve oplossing

- We zullen een subprobleem $A_iA_{i+1}...A_j$ als volgt opsplitsen in 2 nieuwe subproblemen $A_iA_{i+1}...A_k$ en $A_{k+1}A_{k+2}...A_j$.
- We zetten haakjes achter A_k
- De oplossingen van deze subproblemen zullen we bijhouden in een $n \times n$ matrix m. De oplossing voor $A_iA_{i+1}...A_{j-1}A_j$ vinden we dan in $m[i, j]$.
- $m[i, i] = 0$
- De naïeve recursieve oplossing zal opnieuw **exponentiële** uitvoeringstijd hebben.

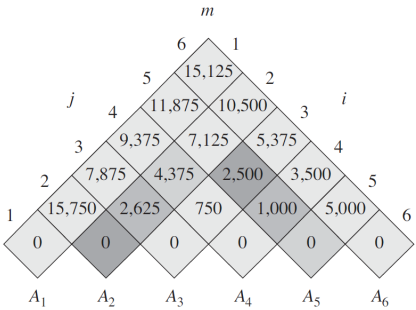
Stap 3: DP toepassen

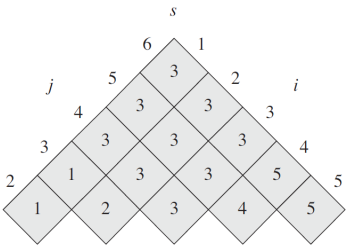
```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

- Complexiteit: $\mathcal{O}(n^3)$

Matrices m and s

- **Bottom-up** voor elke l de optimale oplossing berekenen voor verschillende matrices.





Stap 4: Optimale haakjes

- In s houden we bij op welke k we optimaal gesplitst hebben.
- Zo kunnen we de oplossing reconstrueren.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2        print "A"_i
3  else print "("
4        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6        print ")"
```

Overzicht

Eigenschap 1: Optimale substructuur

- Als de optimale oplossing vervat zit in de optimale oplossing van de subproblemen.

Eigenschap 1: Optimale substructuur

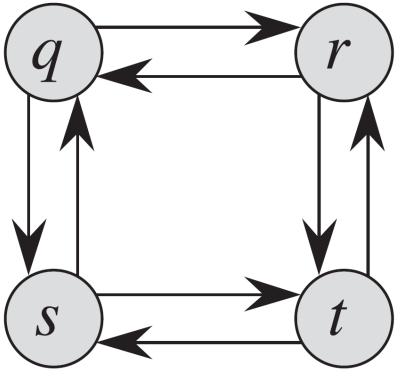
- Als de optimale oplossing vervat zit in de optimale oplossing van de subproblemen.
- Vaak complexiteit van de vorm $\mathcal{O}(n^{a+b})$
  - $a = \#$ subproblemen per probleem

Eigenschap 1: Optimale substructuur

- Als de optimale oplossing vervat zit in de optimale oplossing van de subproblemen.
- Vaak complexiteit van de vorm $\mathcal{O}(n^{a+b})$
  - $a = \#$ subproblemen per probleem
  - $b = \#$ keuzes per subprobleem

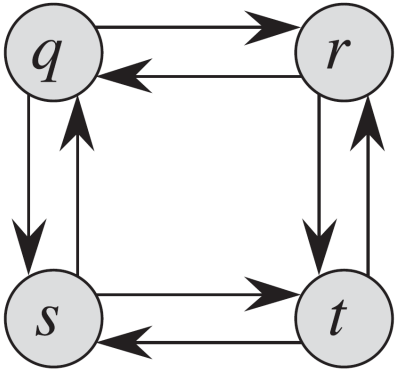
Subproblemen moet onafhankelijk zijn

- $q \to r$ en $r \to t$ zijn subproblemen van $q \to t$



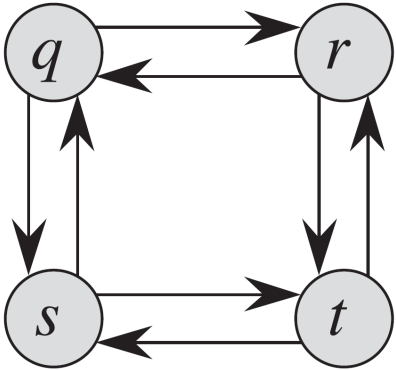
Subproblemen moet onafhankelijk zijn

- $q \to r$ en $r \to t$ zijn subproblemen van $q \to t$
- Ongewogen kortste pad
  - $q \to r$
  - $r \to t$



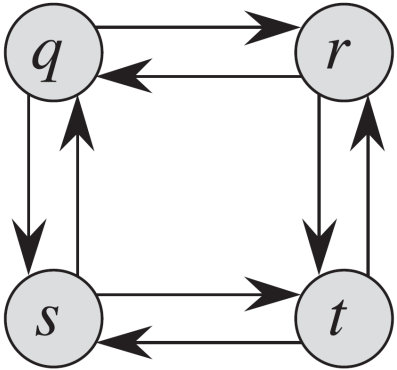
Subproblemen moet onafhankelijk zijn

- $q \to r$ en $r \to t$ zijn subproblemen van $q \to t$
- Ongewogen kortste pad
  - $q \to r$
  - $r \to t$
- Ongewogen langste **simpel** pad



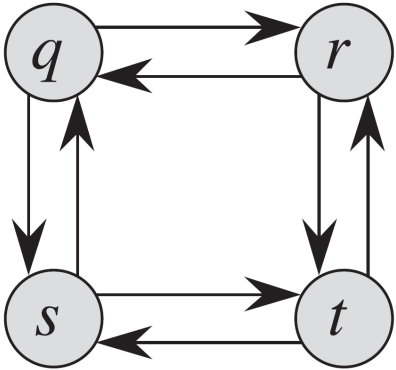
Subproblemen moet onafhankelijk zijn

- $q \to r$ en $r \to t$ zijn subproblemen van $q \to t$
- Ongewogen kortste pad
  - $q \to r$
  - $r \to t$
- Ongewogen langste **simpel** pad
  - $q \to s \to t \to r$



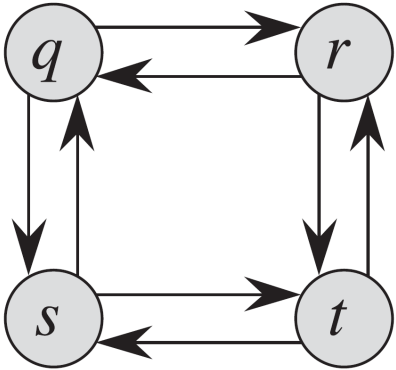
Subproblemen moet onafhankelijk zijn

- $q \to r$ en $r \to t$ zijn subproblemen van $q \to t$
- Ongewogen kortste pad
  - $q \to r$
  - $r \to t$
- Ongewogen langste **simpel** pad
  - $q \to s \to t \to r$
  - $r \to q \to s \to t$



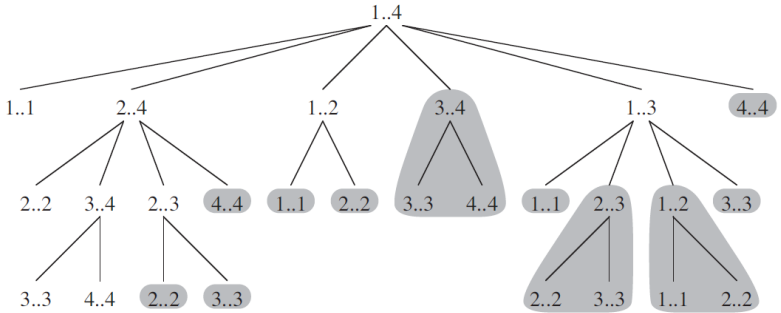
Subproblemen moet onafhankelijk zijn

- $q \to r$ en $r \to t$ zijn subproblemen van $q \to t$
- Ongewogen kortste pad
  - $q \to r$
  - $r \to t$
- Ongewogen langste **simpel** pad
  - $q \to s \to t \to r$
  - $r \to q \to s \to t$
  - Pad is niet meer simpel



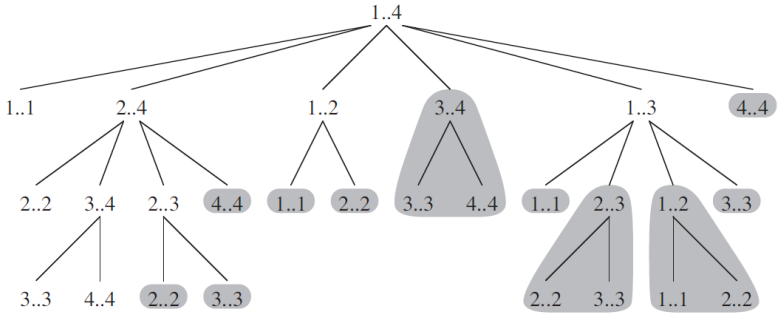
Eigenschap 2: Overlappende subproblemen

- Het matrix probleem heeft ook overlappende problemen



Eigenschap 2: Overlappende subproblemen

- Het matrix probleem heeft ook overlappende problemen
- Polynomiaal #verschillende subproblemen



Vragen

Zijn er nog vragen?