



$$P(x, i) = \begin{cases} P(x, i-1), \\ \max\{P(x, i-1), \dots\} \end{cases}$$

$x \backslash i$	0	1	2	3	3	5
0	1	0	0	0	0	0
1	1	0	0	0	0	0
2	1	0	0	0	+13	
3						

2 Dynamic Programming

Algorithmen und Datenstrukturen 2
Sommer 2024

Prof. Dr. Sándor Fekete

Dynamic Programming für Anfänger

- Wieviel ist das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1 \quad =13$$

Dynamic Programming für Anfänger

- Wieviel ist das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 13$$

- Und das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 13 + 1 = 14$$

Dynamic Programming für Anfänger

- Wieviel ist das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 \quad = 13$$

- Und das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 \quad = 13 + 1 = 14$$

- Dynamic Programming:

Dynamic Programming für Anfänger

- Wieviel ist das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 13$$

- Und das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 13 + 1 = 14$$

- Dynamic Programming:
 - Nicht von vorn lösen

Dynamic Programming für Anfänger

- Wieviel ist das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 13$$

- Und das?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 13 + 1 = 14$$

- Dynamic Programming:
 - Nicht von vorn lösen
 - Zwischenergebnisse merken

Dynamic Programming: Grundstruktur

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die Vorgeschichte beim Weiterrechnen von der Zukunft trennen und deshalb den Übergang durch Enumerieren lösen.

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die Vorgeschichte beim Weiterrechnen von der Zukunft trennen und deshalb den Übergang durch Enumerieren lösen.



$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6										

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die Vorgeschichte beim Weiterrechnen von der Zukunft trennen und deshalb den Übergang durch Enumerieren lösen.



$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6										

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die Vorgeschichte beim Weiterrechnen von der Zukunft trennen und deshalb den Übergang durch Enumerieren lösen.



$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die Vorgeschichte beim Weiterrechnen von der Zukunft trennen und deshalb den Übergang durch Enumerieren lösen.



$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die Vorgeschichte beim Weiterrechnen von der Zukunft trennen und deshalb den Übergang durch Enumerieren lösen.



$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim Weiterrechnen von der Zukunft trennen und deshalb den Übergang durch Enumerieren lösen.



$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der Zukunft trennen und deshalb den Übergang durch Enumerieren lösen.



$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.



$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

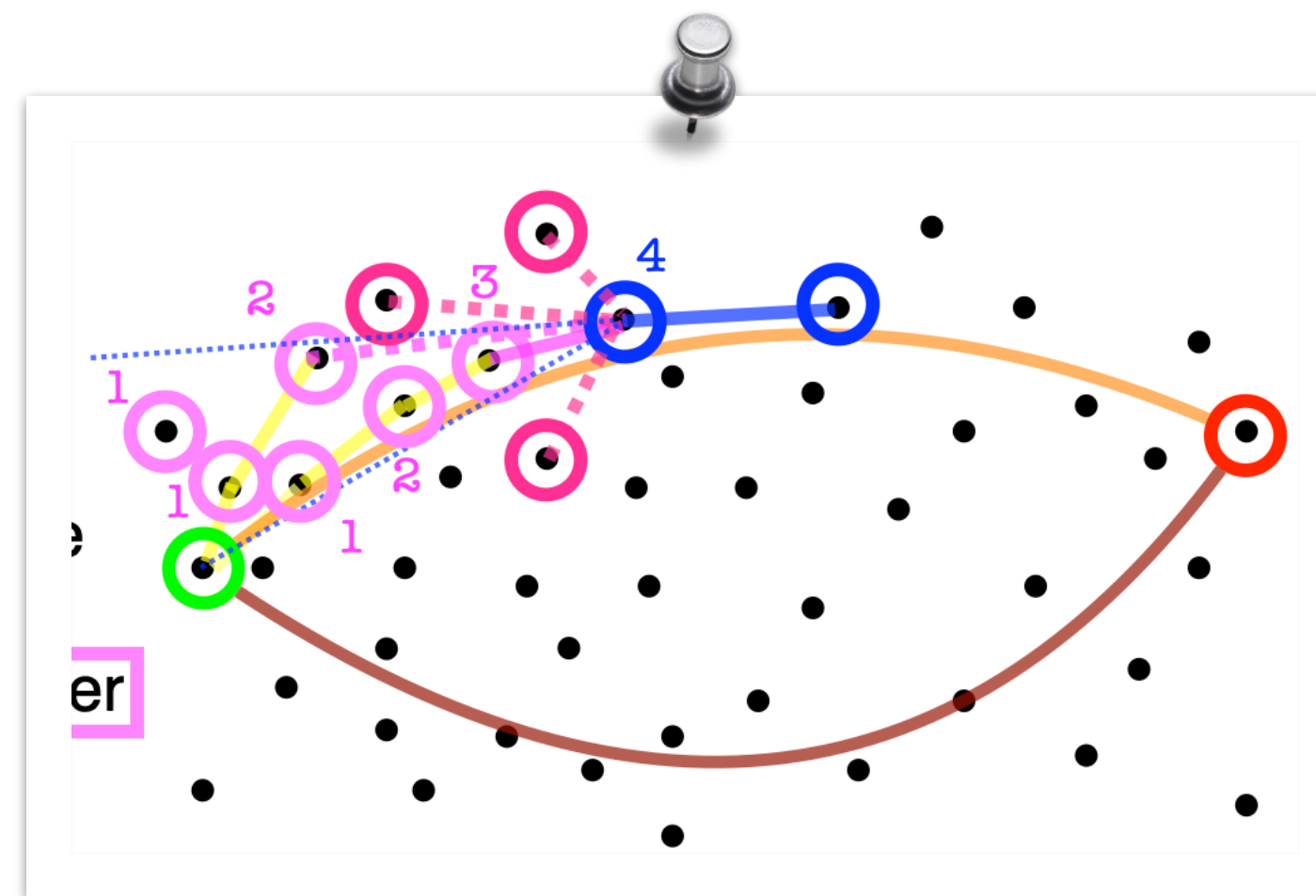
Objekt i

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i

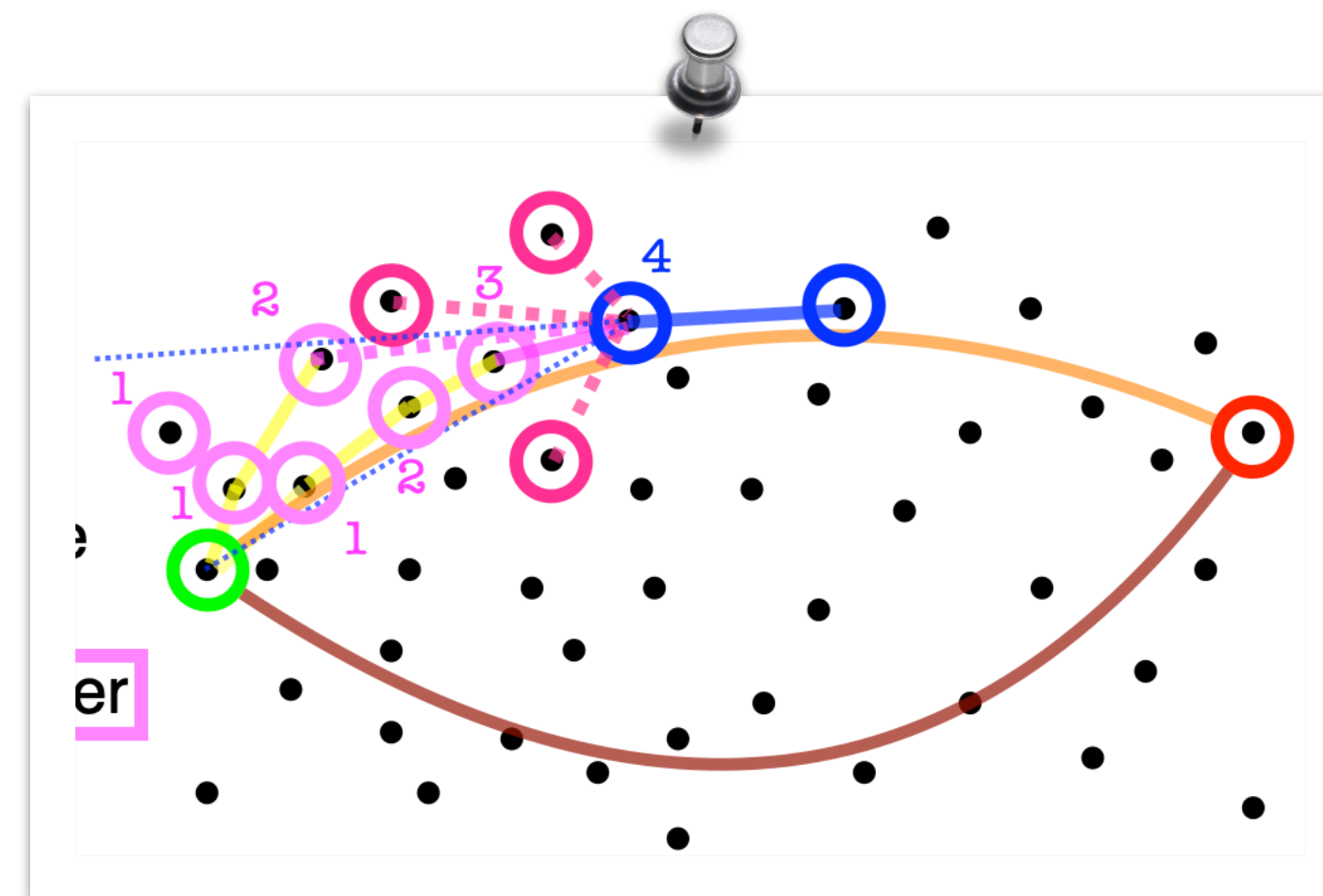


Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

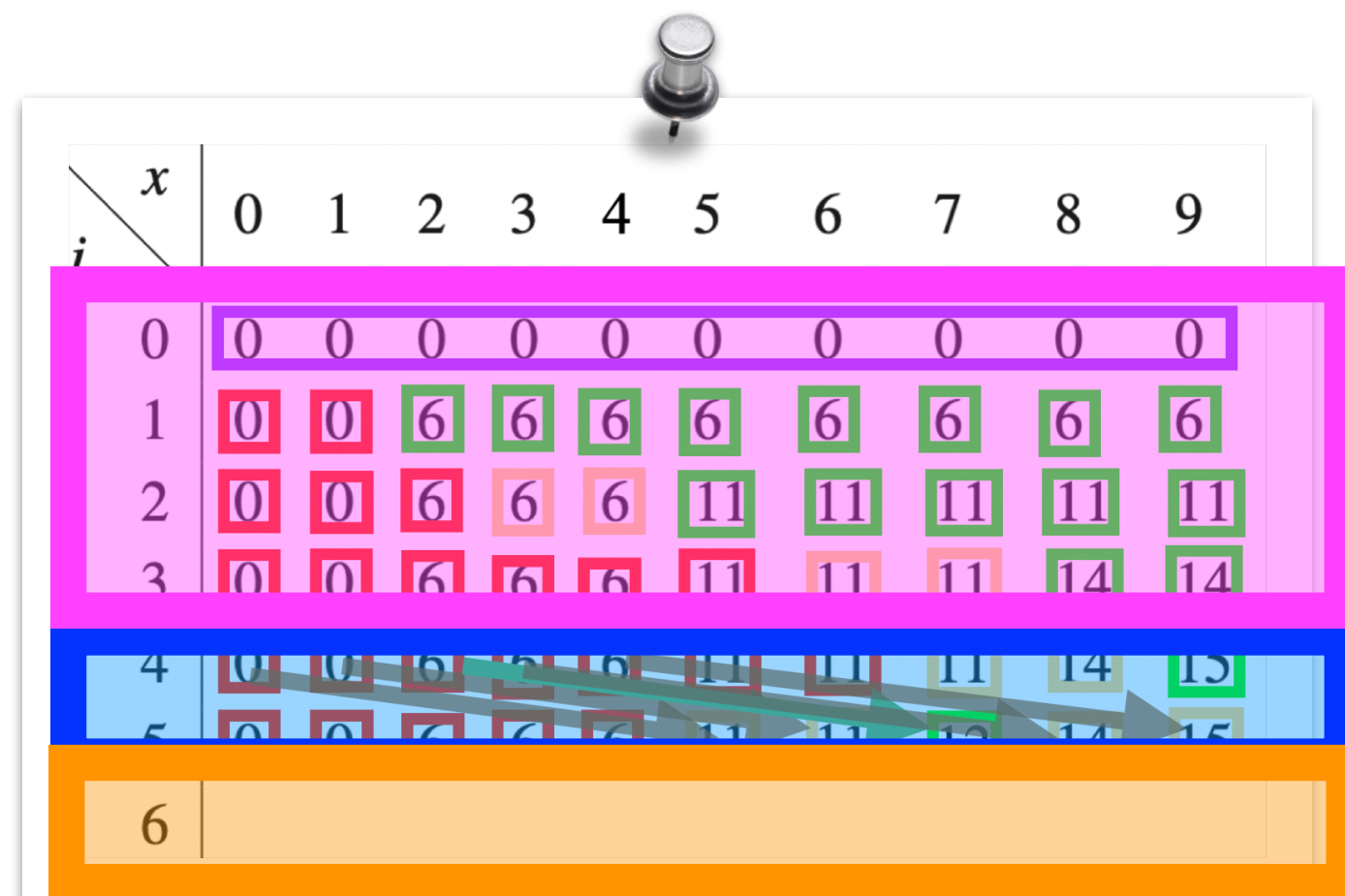
Objekt i



Kante i

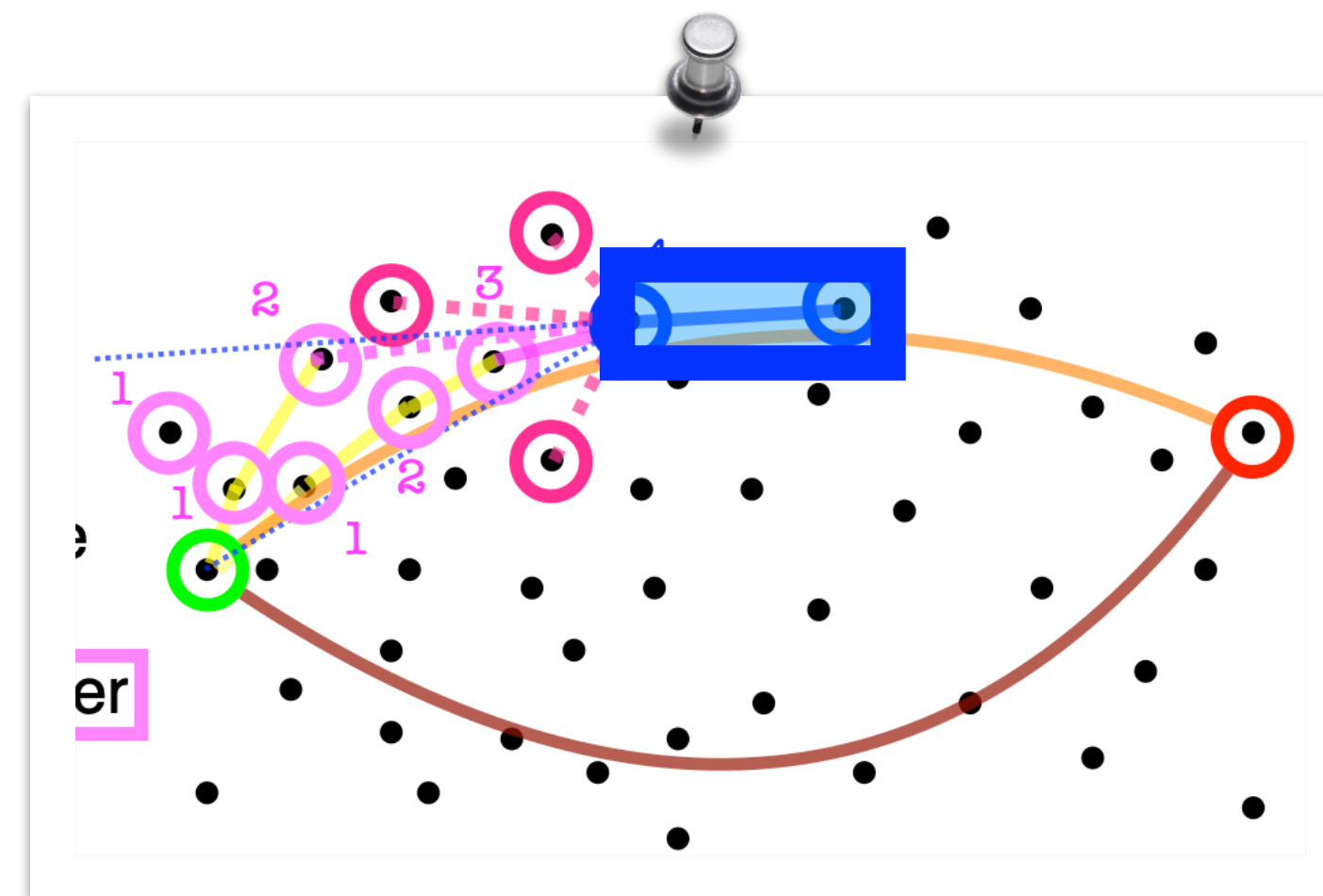
Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.



$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i



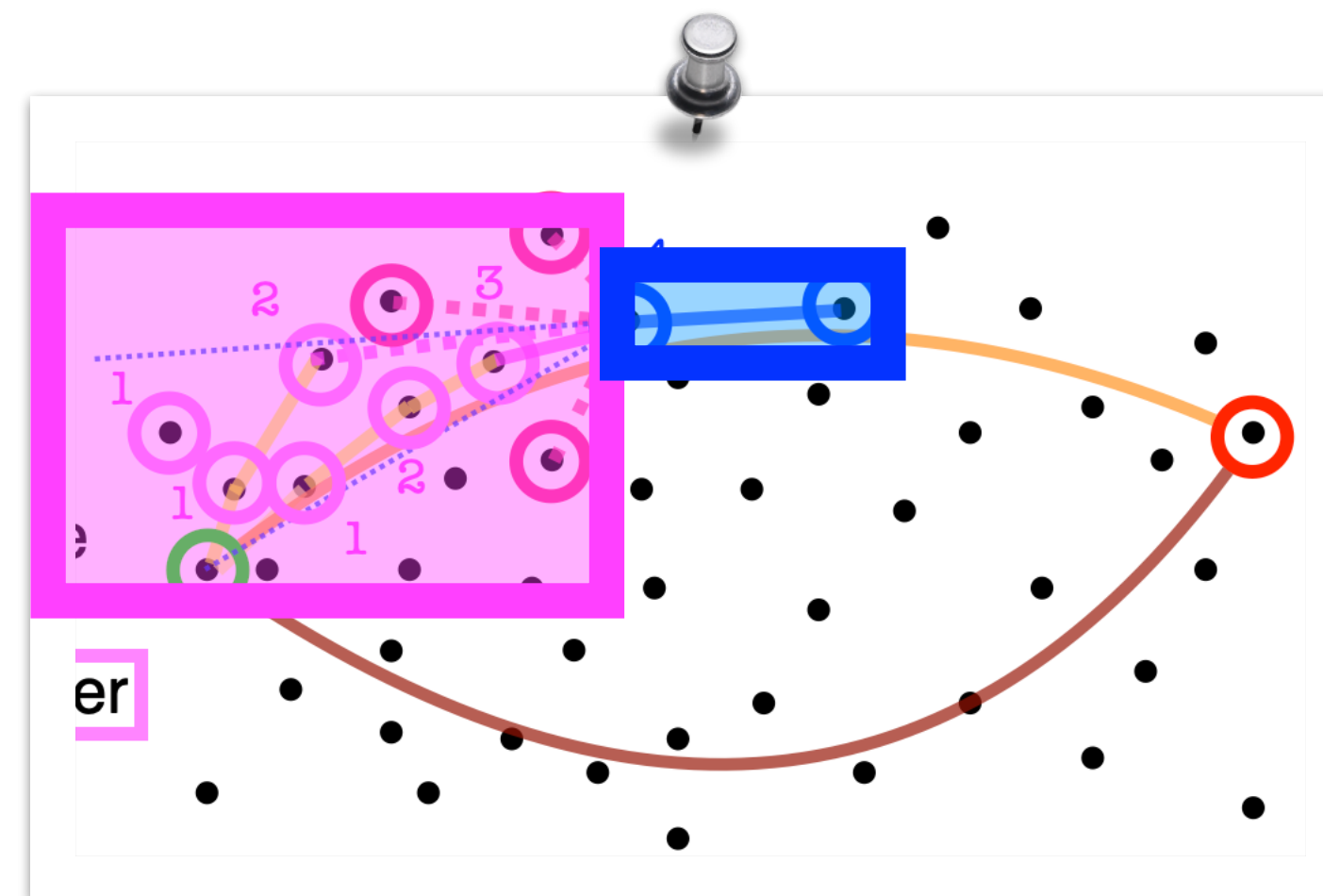
Kante i

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i



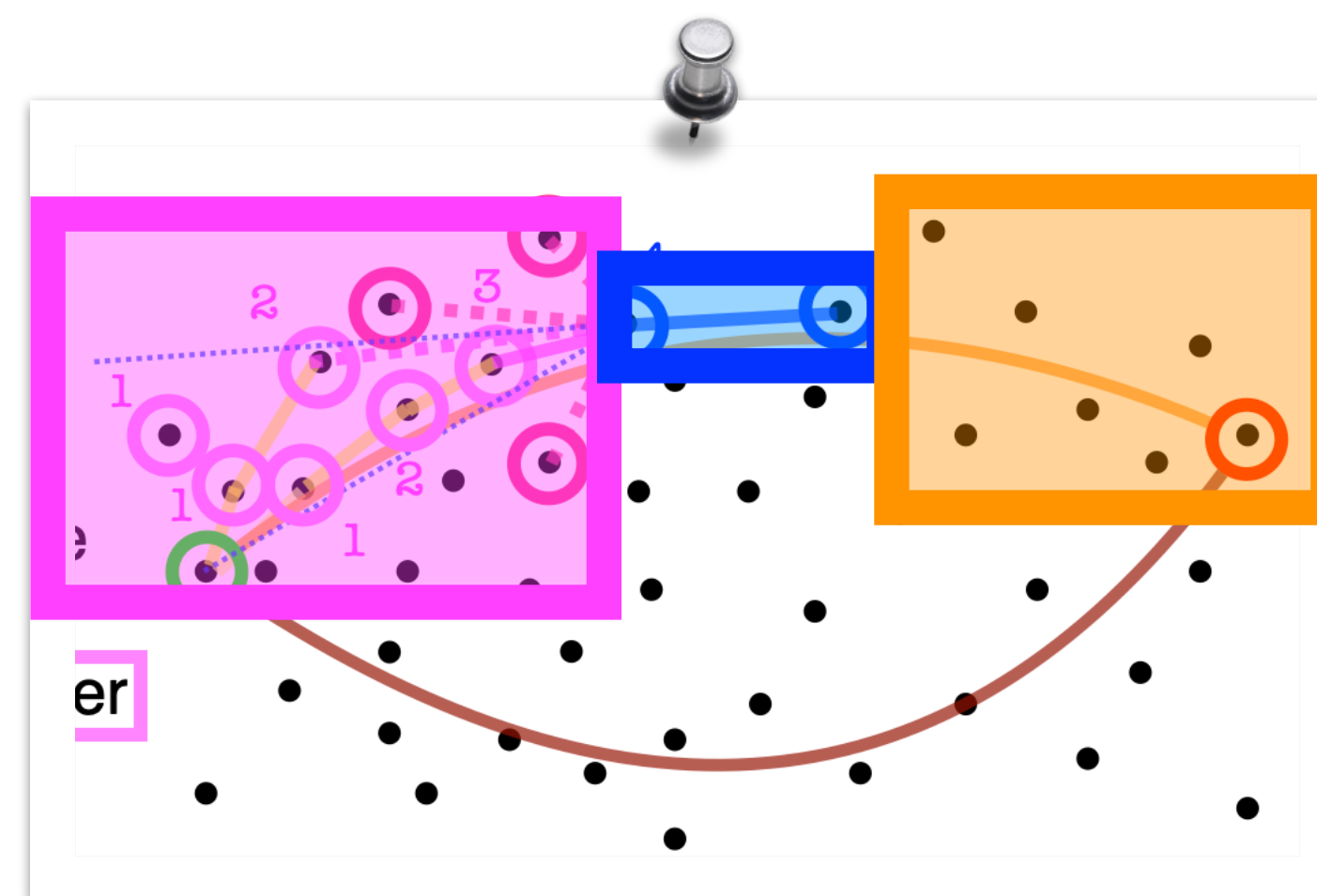
Kante i

Dynamic Programming: Grundstruktur

- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i



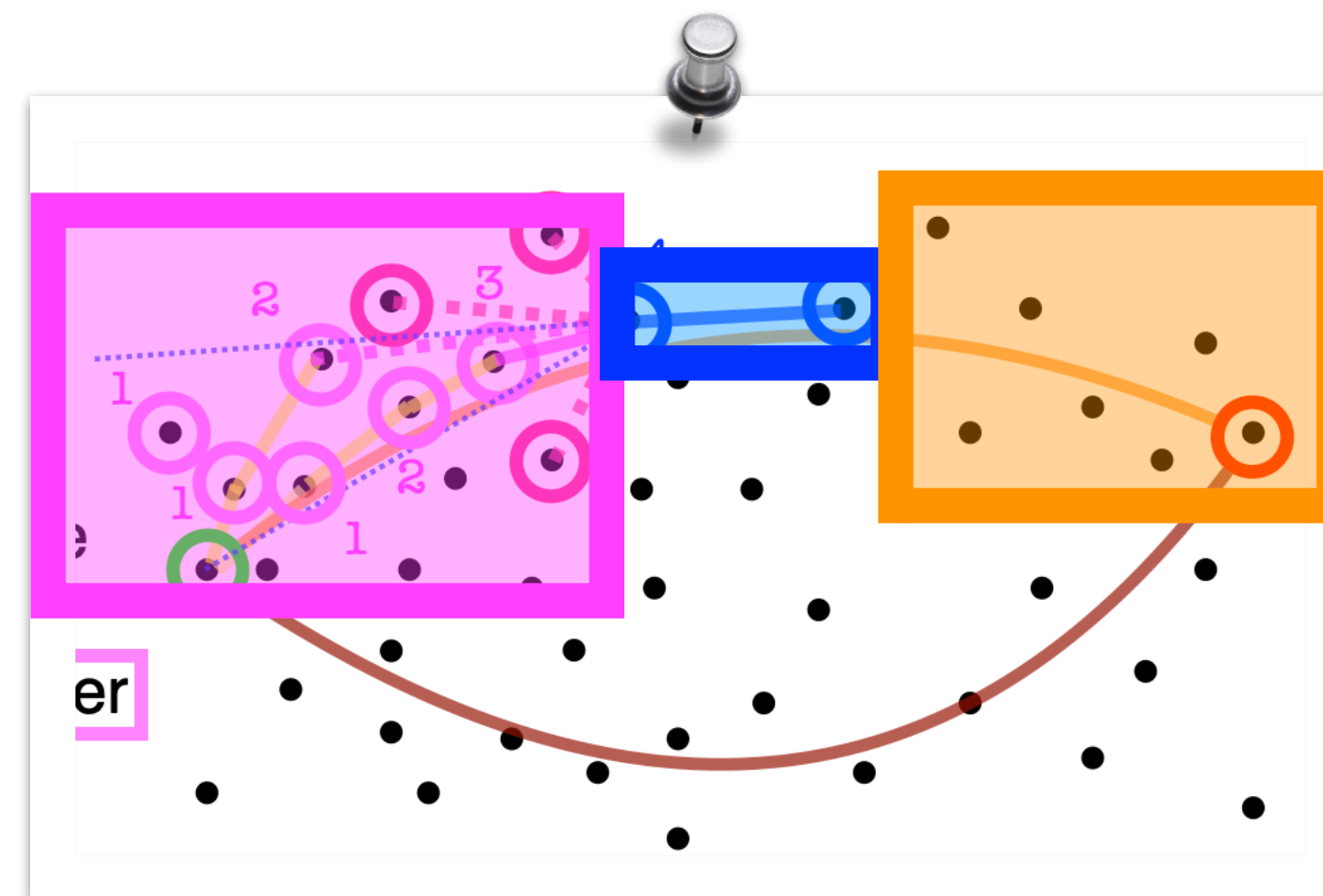
Kante i

Dynamic Programming: Grundstruktur

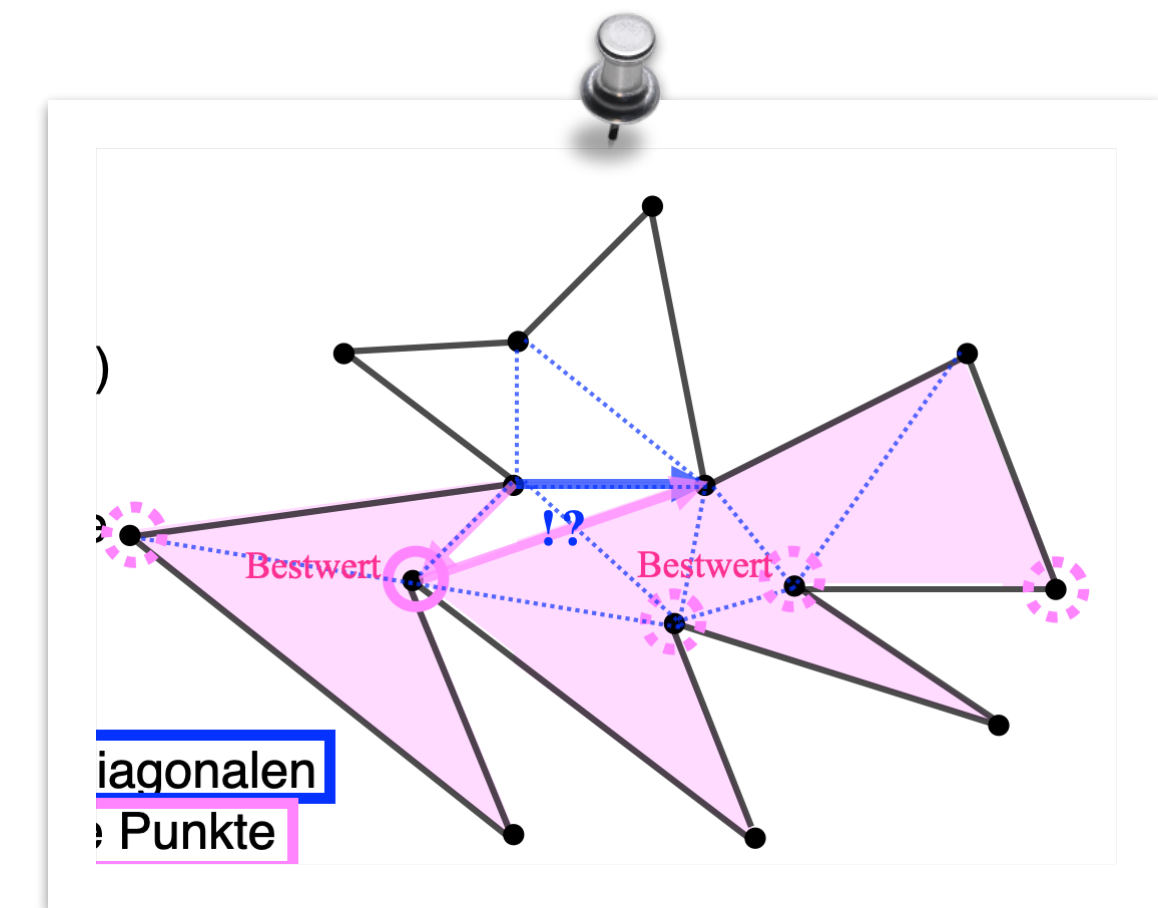
- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i



Kante i

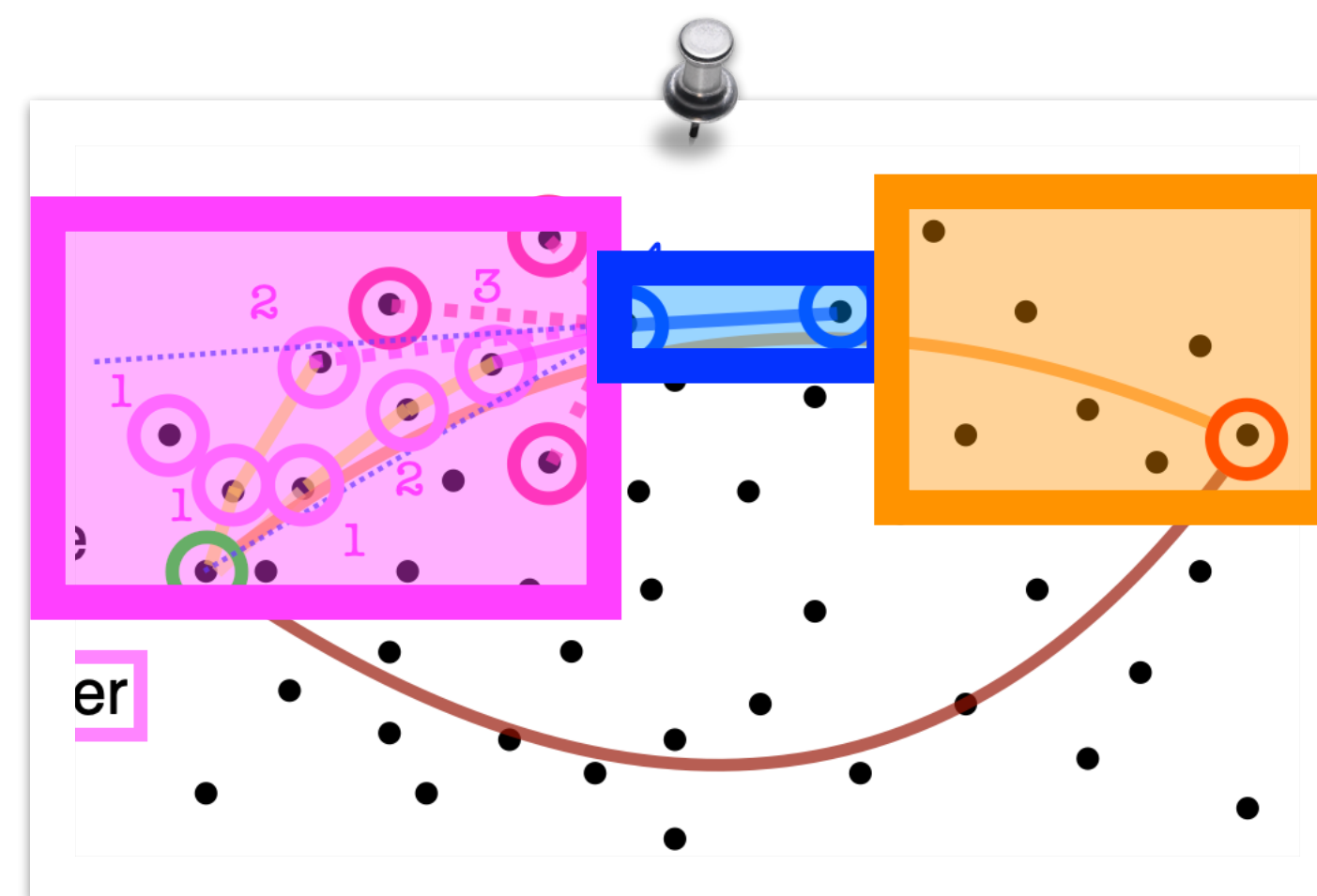


Dynamic Programming: Grundstruktur

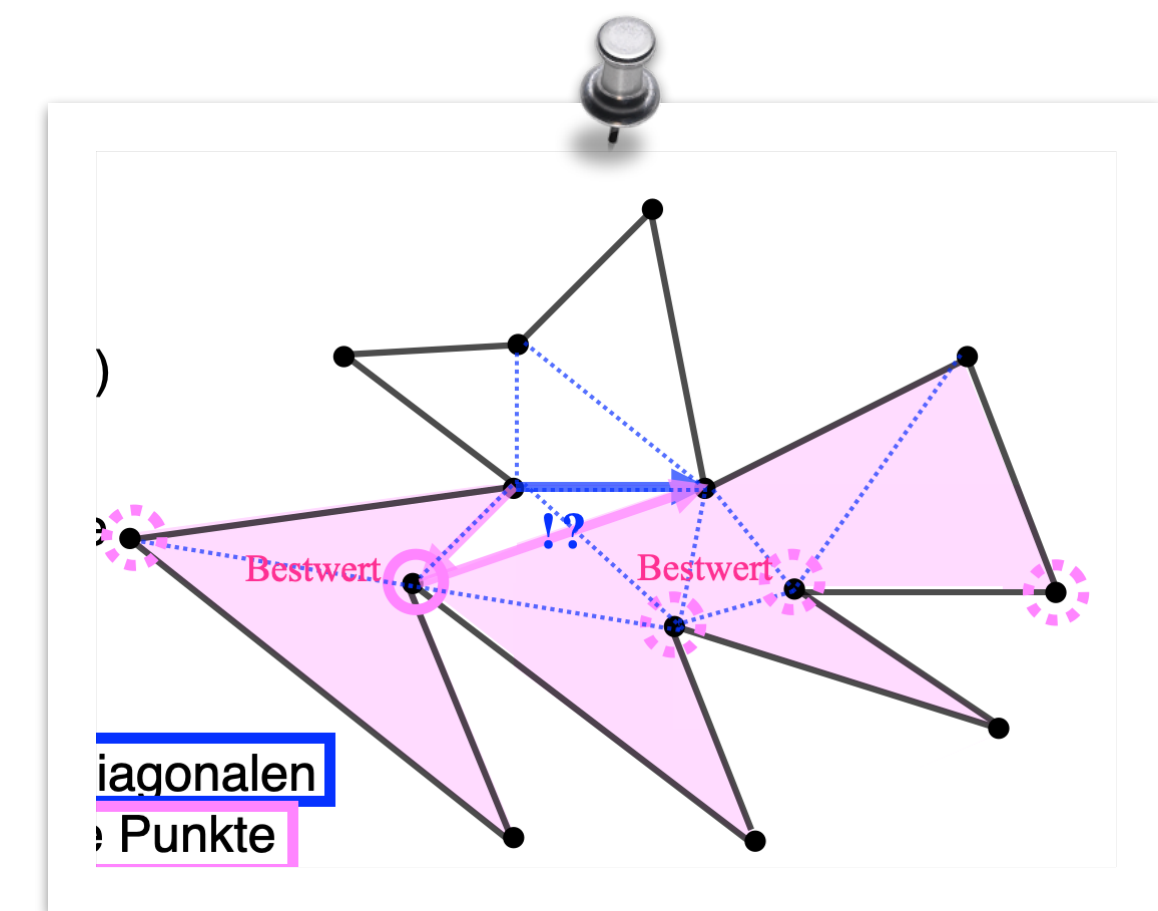
- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i



Kante i



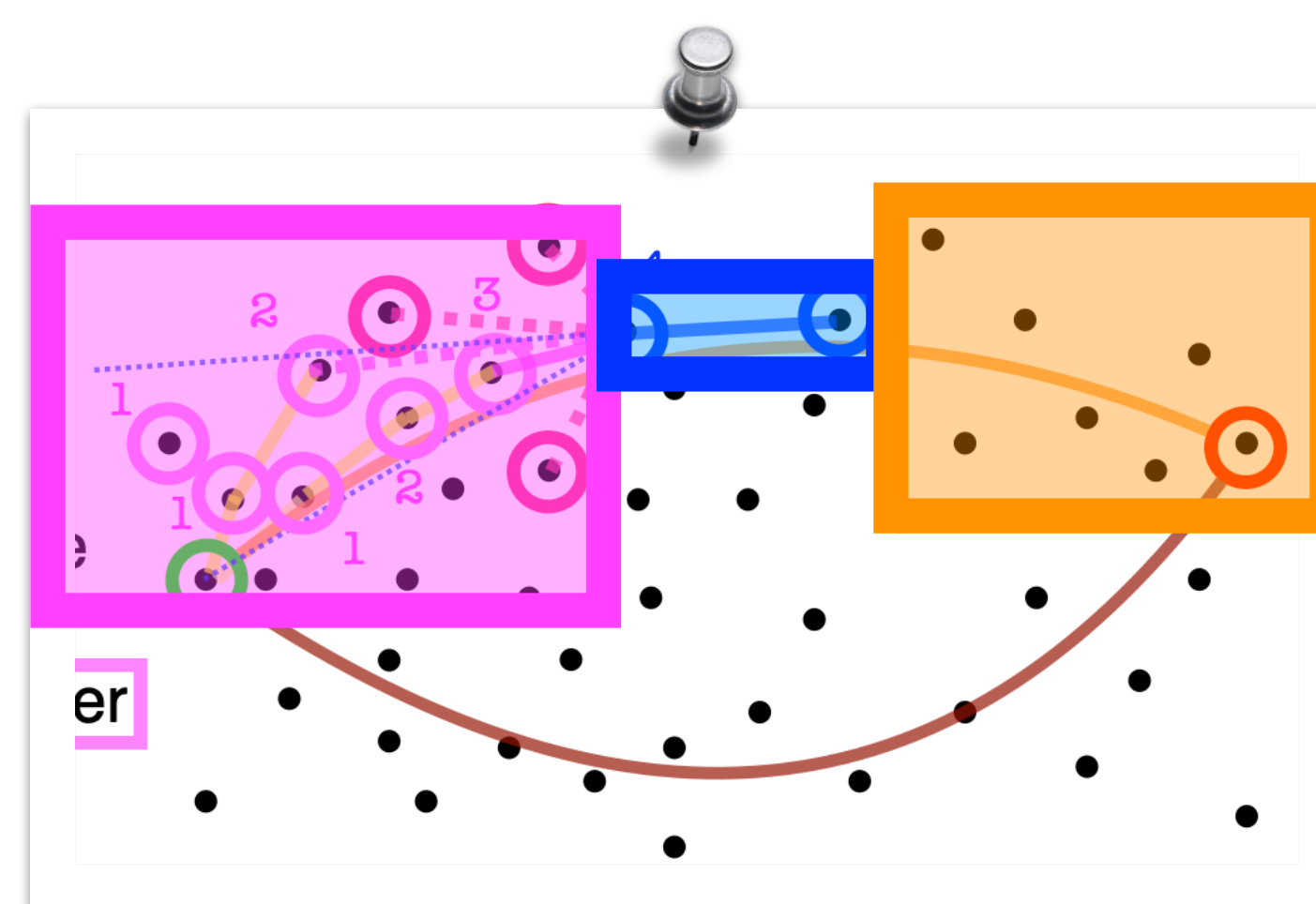
Kante i

Dynamic Programming: Grundstruktur

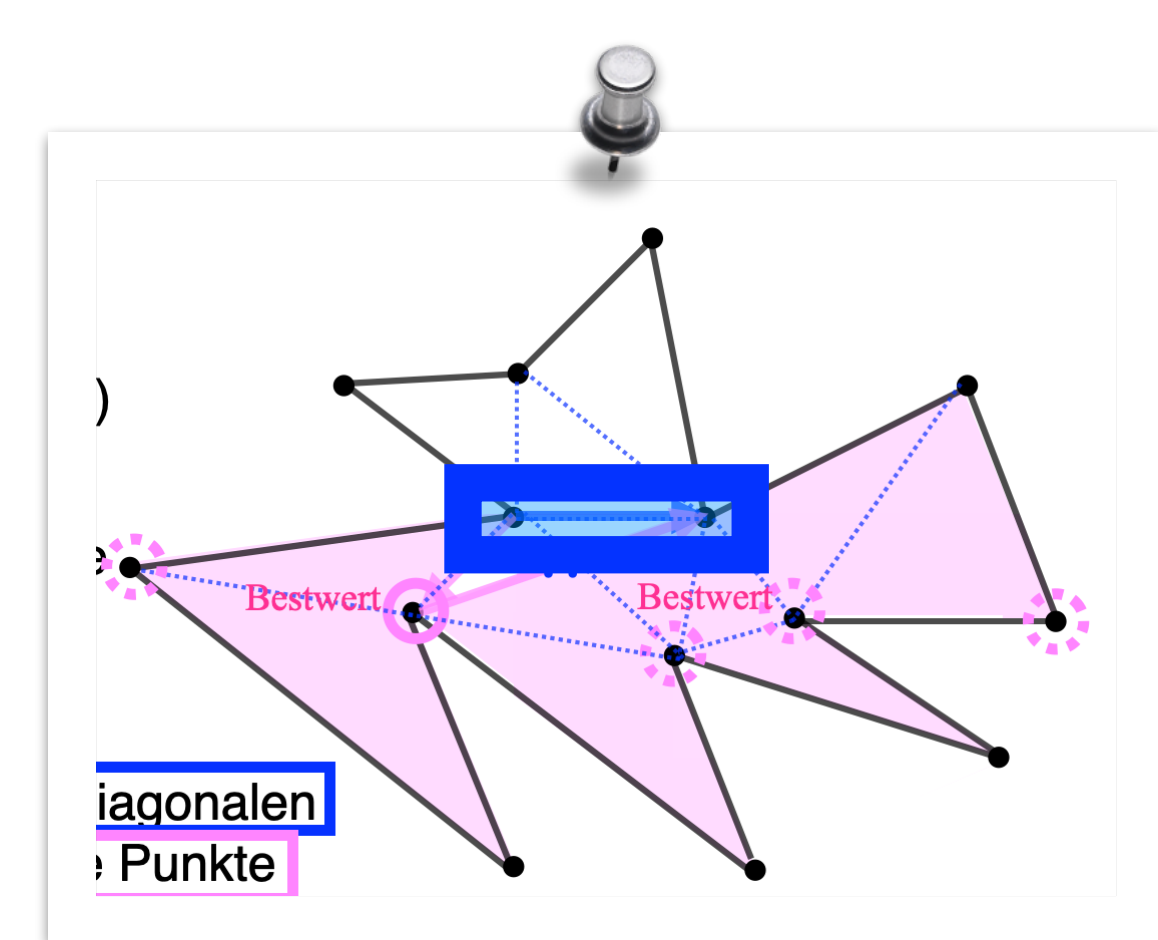
- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i



Kante i



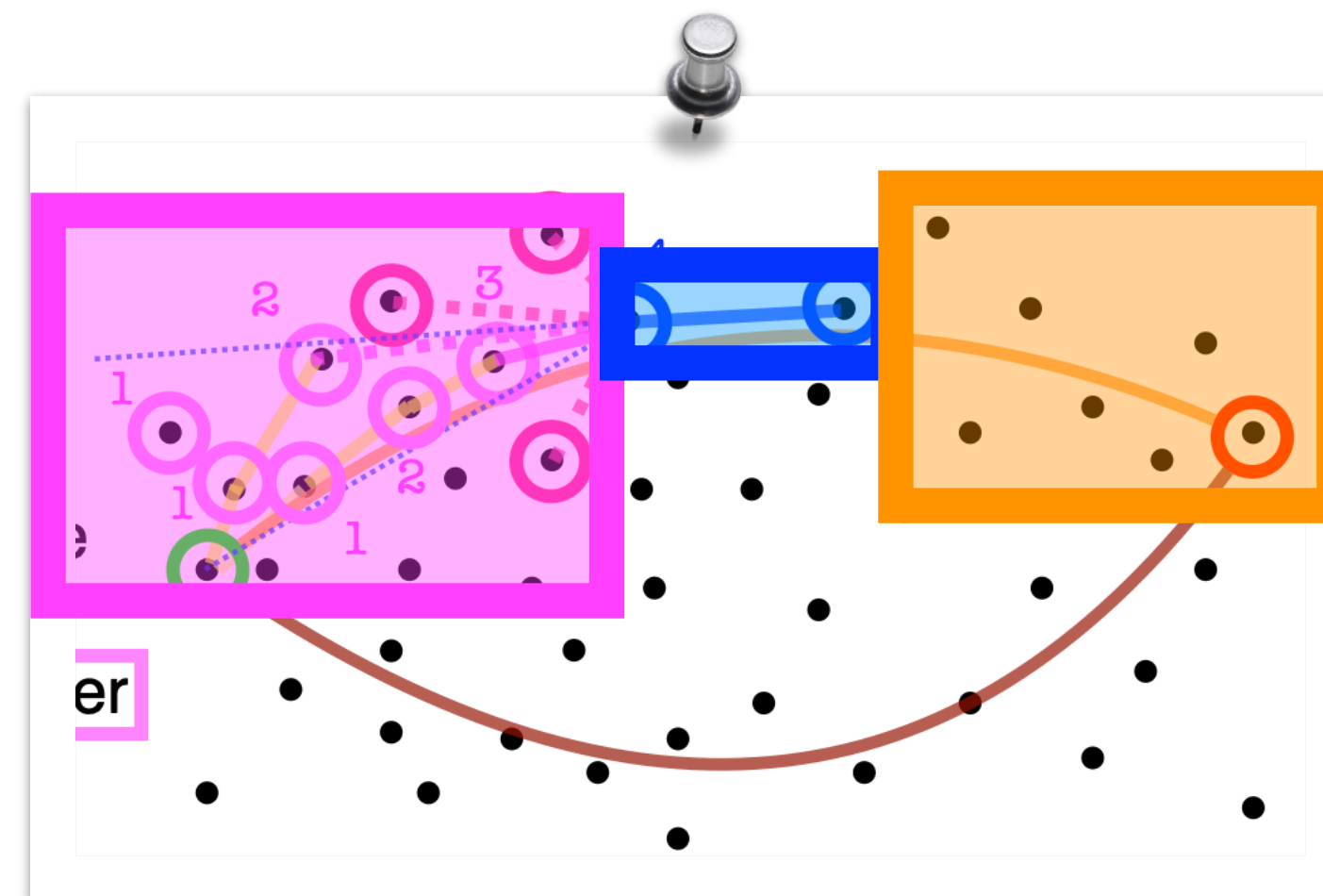
Kante i

Dynamic Programming: Grundstruktur

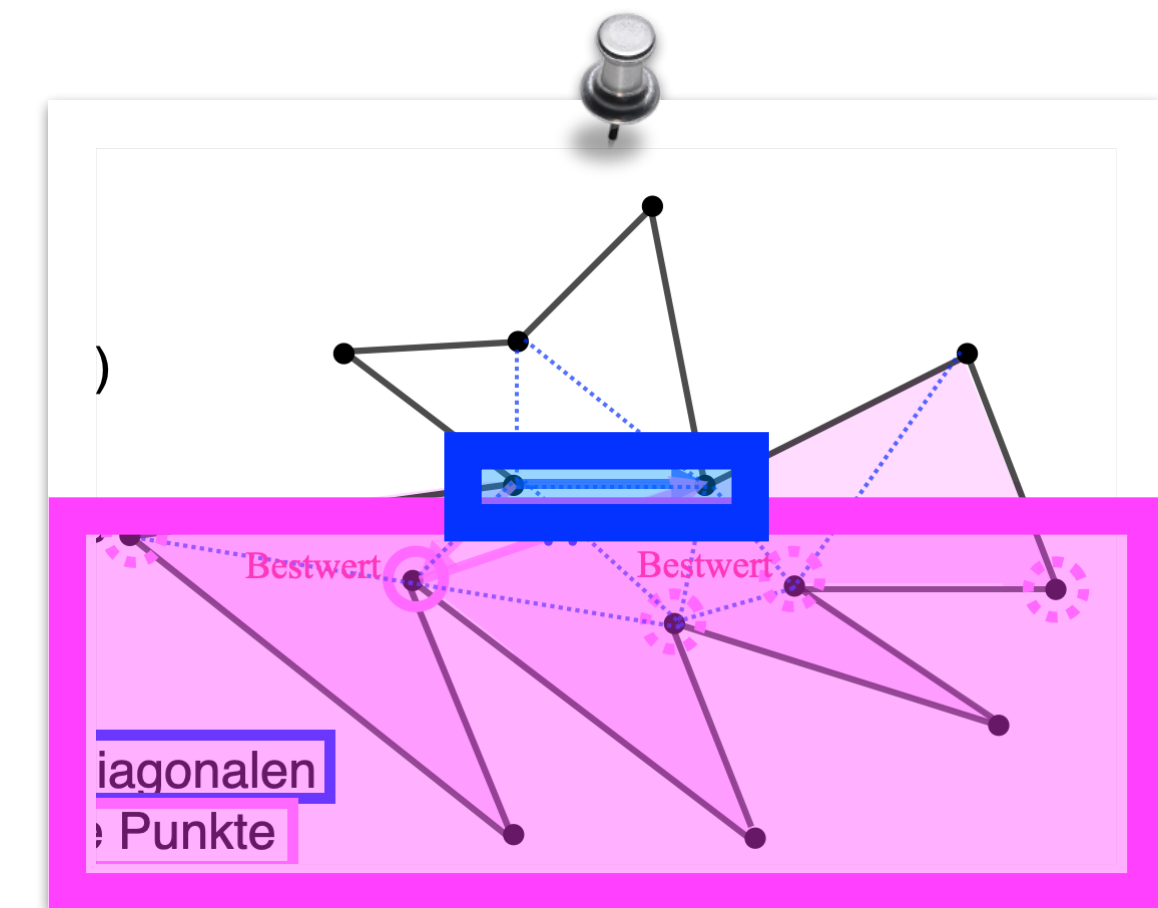
- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	11	14	15
6										

Objekt i



Kante i



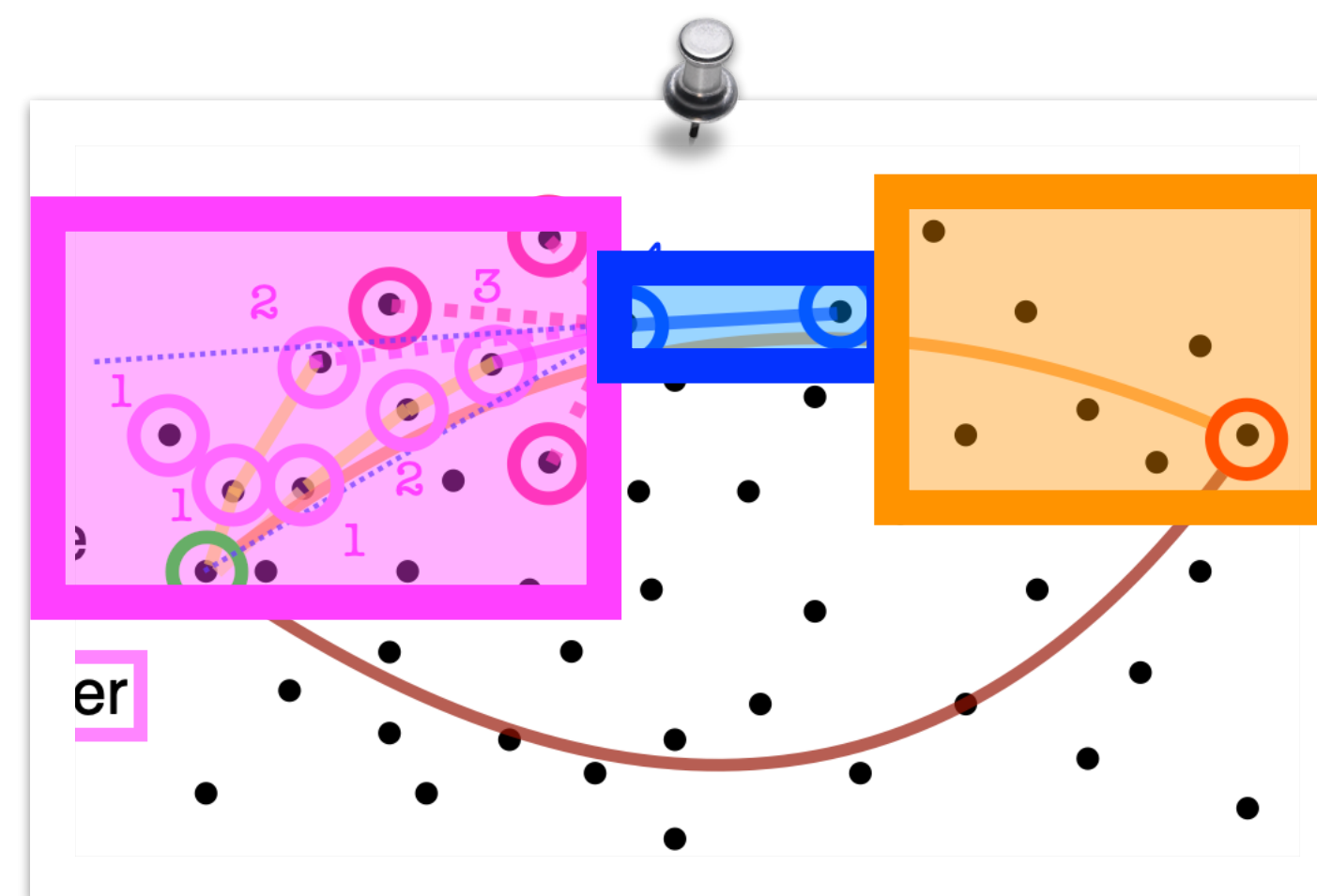
Kante i

Dynamic Programming: Grundstruktur

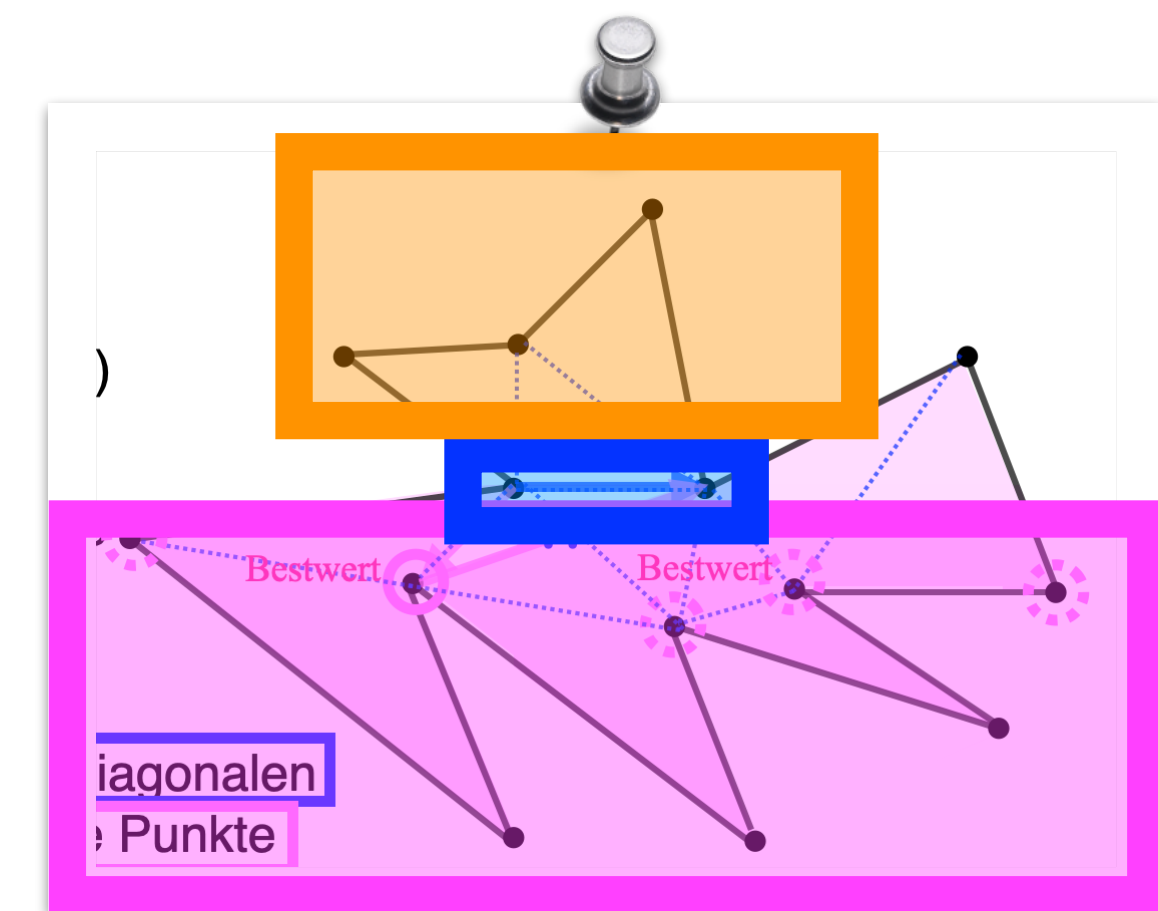
- Nicht von vorn lösen
- Zwischenergebnisse merken
- Man kann die **Vorgeschichte** beim **Weiterrechnen** von der **Zukunft** trennen und deshalb den Übergang durch Enumerieren lösen.

$i \backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	10	16	11	11	11	14	15			
5	10	16	11	11	11	14	15			
6	6									

Objekt i



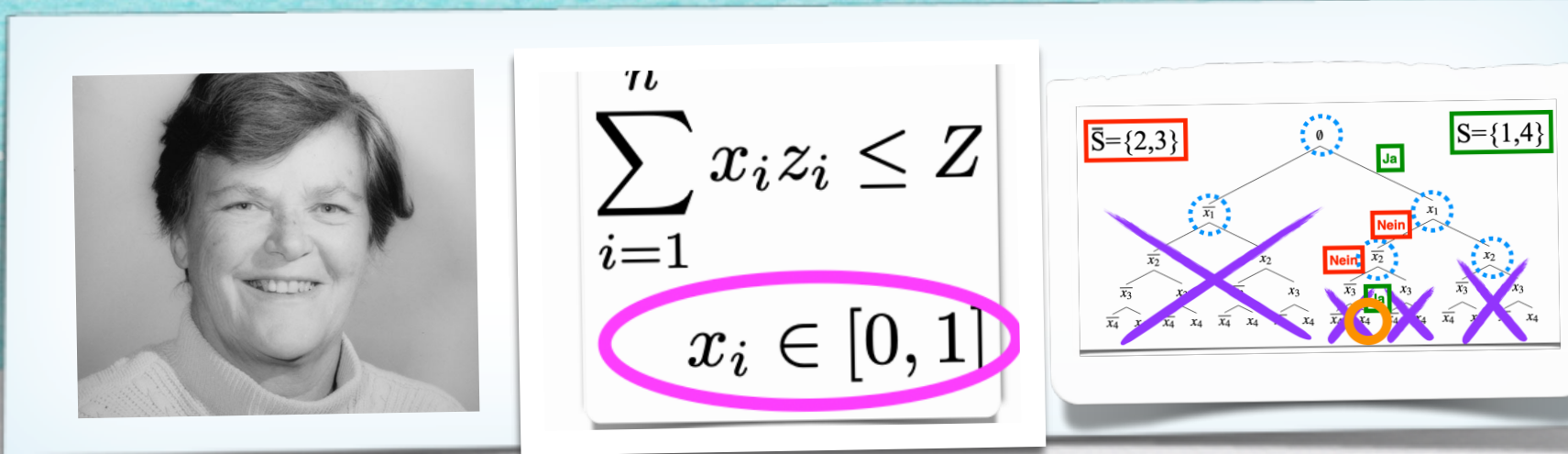
Kante i



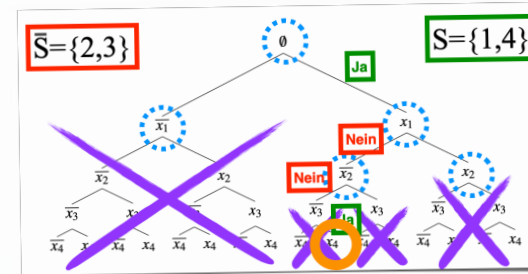
Kante i

Quiz!





$$\sum_{i=1}^n x_i z_i \leq Z$$



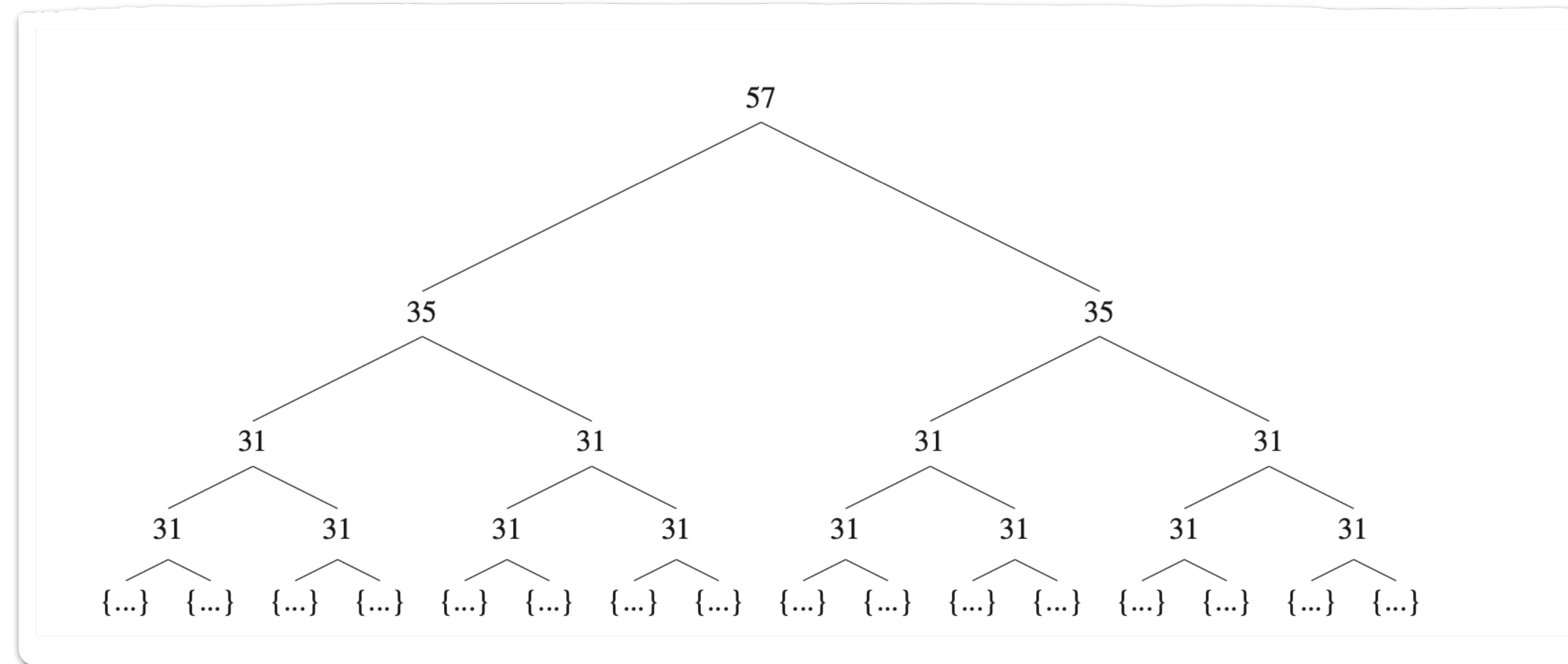
Algorithmen und Datenstrukturen 2

Sommer 2024

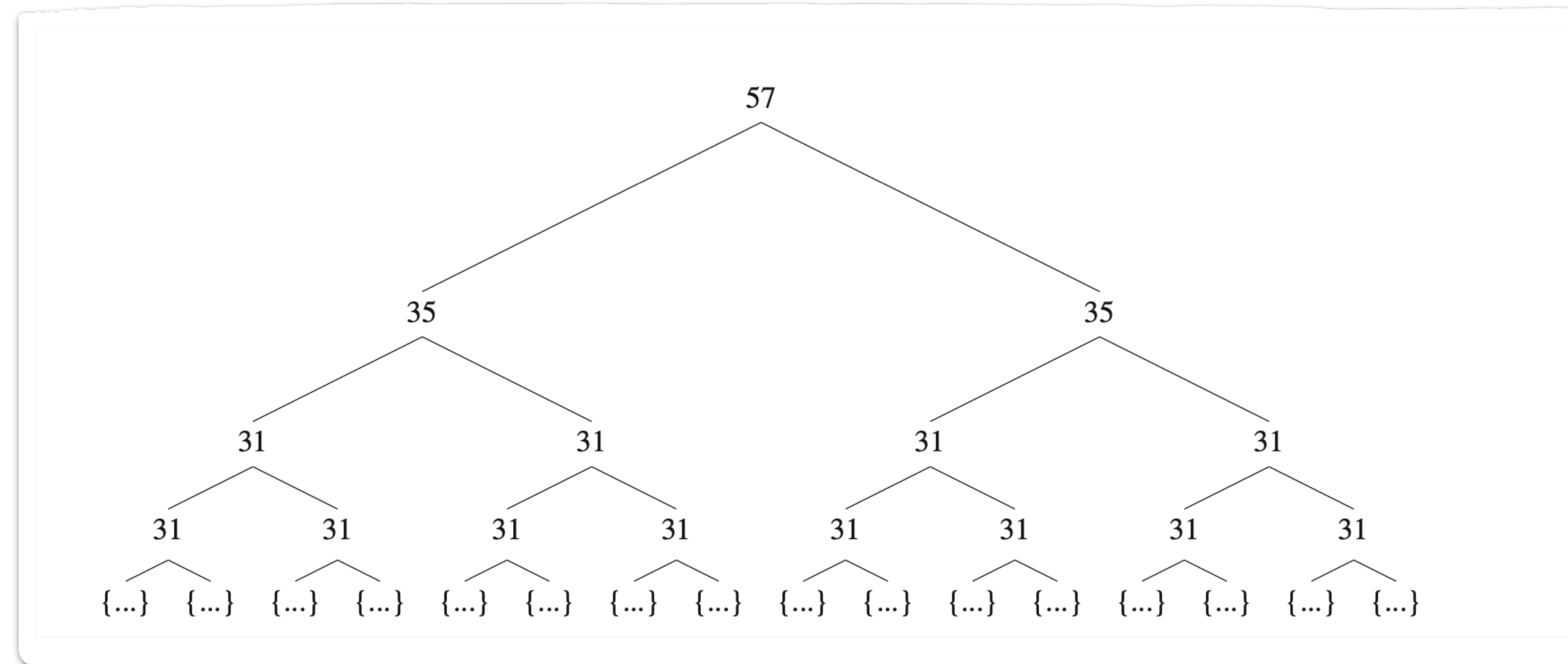
Prof. Dr. Sándor Fekete

3.1 Motivation

Enumerationsprinzip

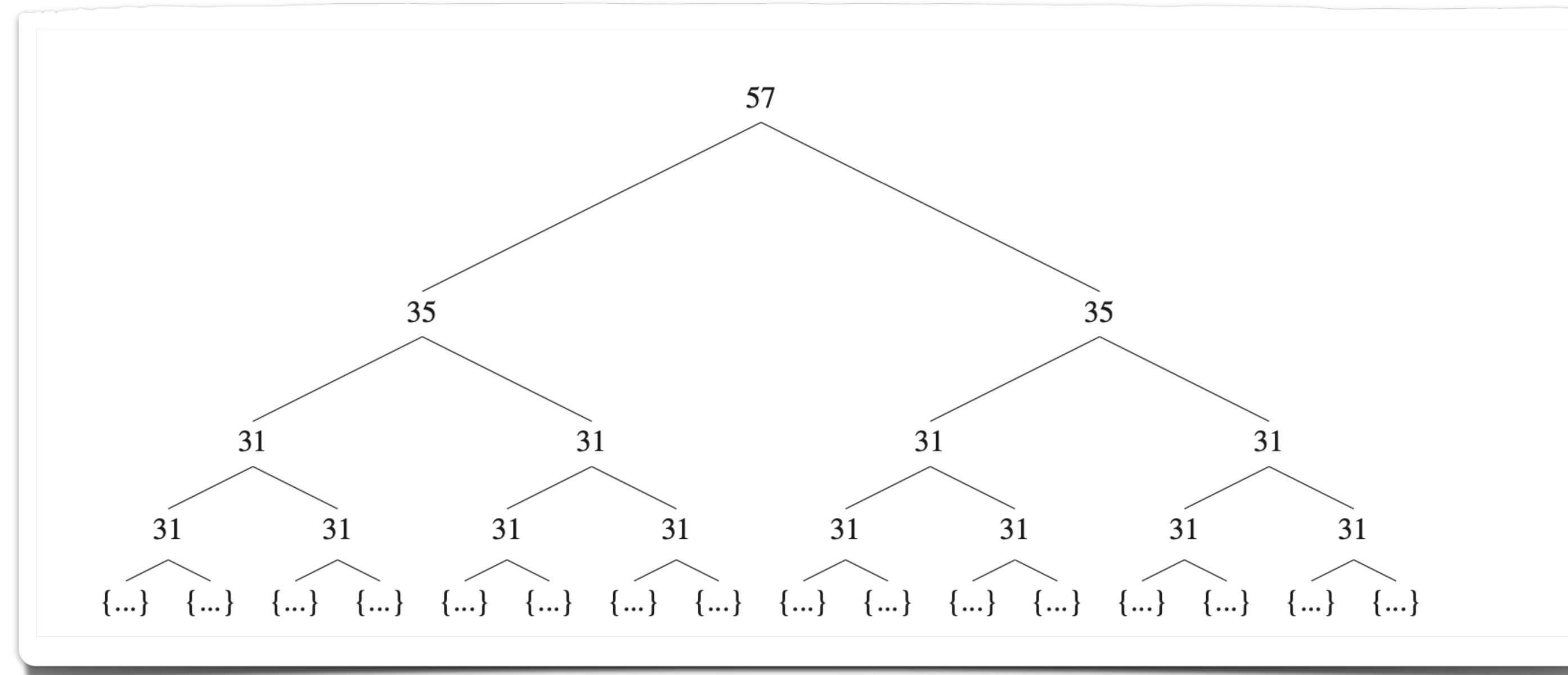


Enumerationsprinzip



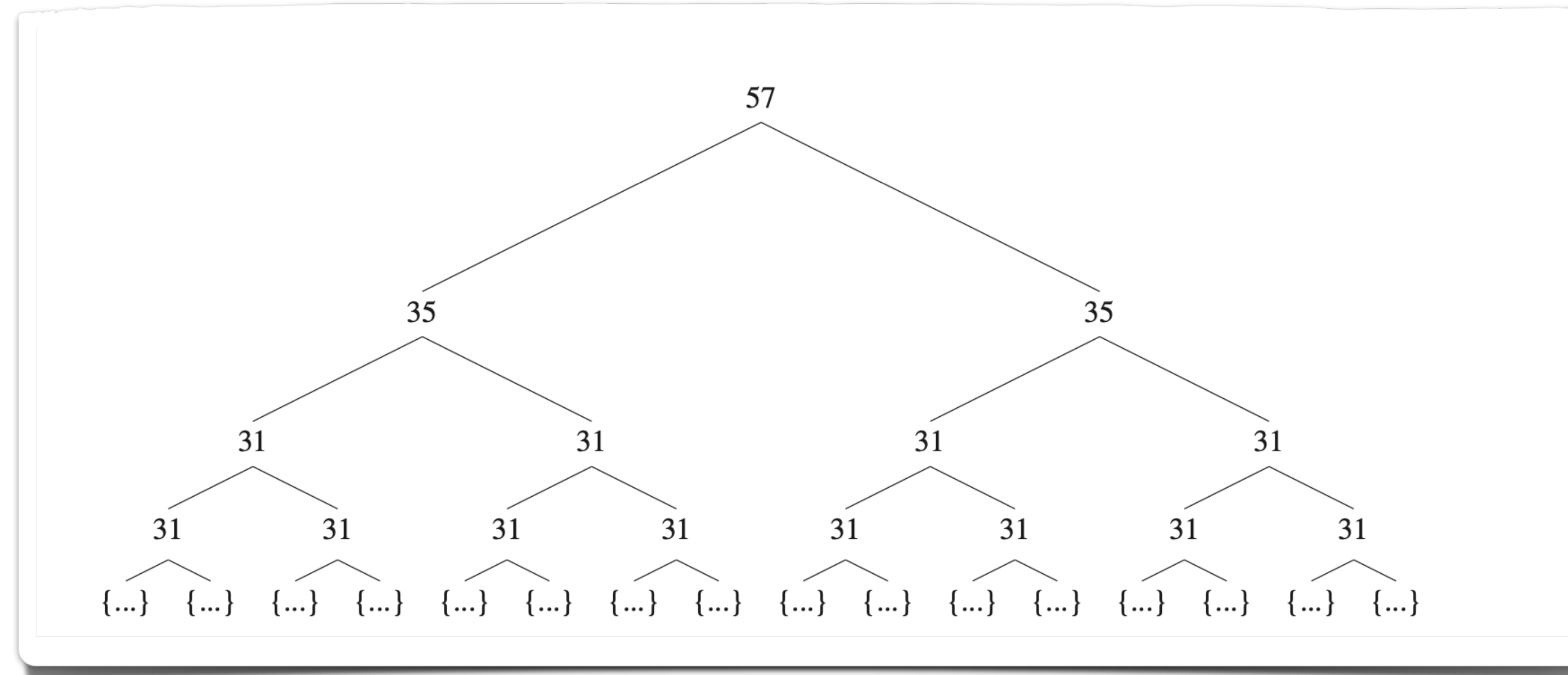
- Exponentiell viele Fälle!

Enumerationsprinzip



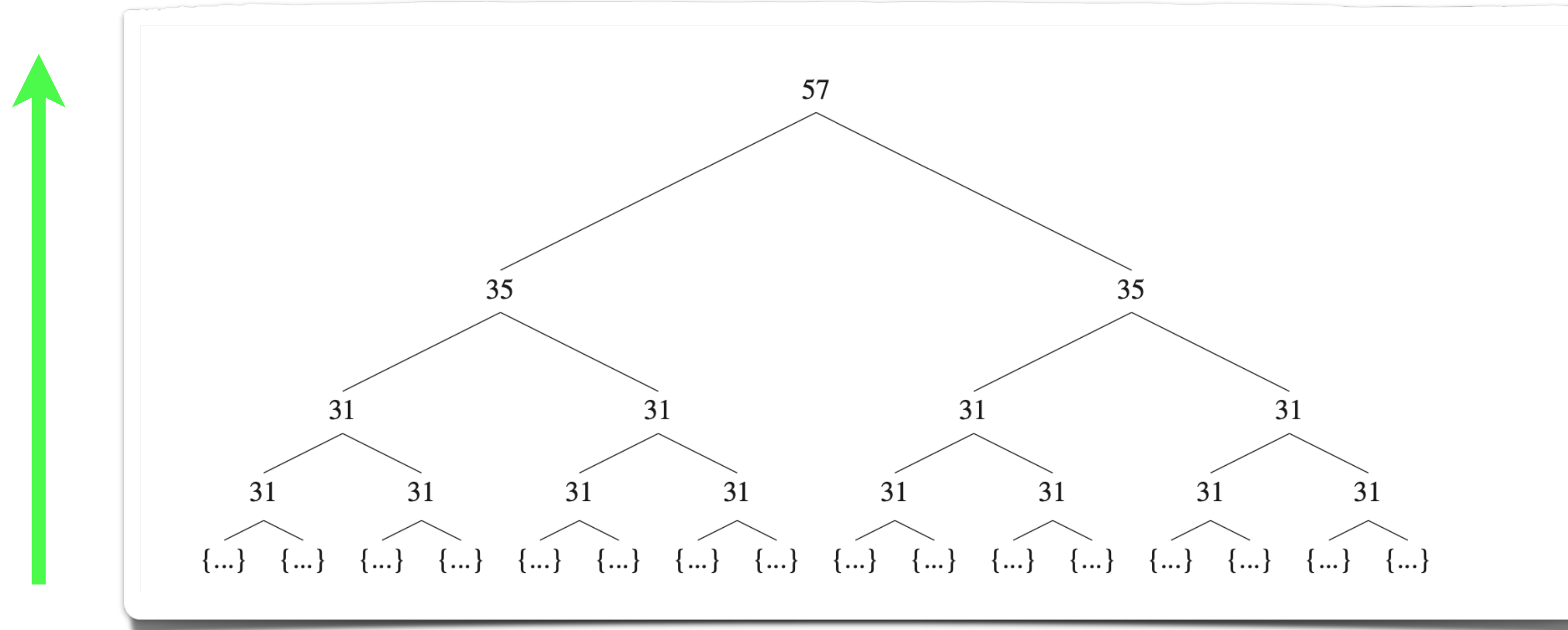
- Exponentiell viele Fälle!
- Wie geht das systematisch?

Enumerationsprinzip



- Exponentiell viele Fälle!
- Wie geht das systematisch?
- Wo kann man Arbeit sparen?

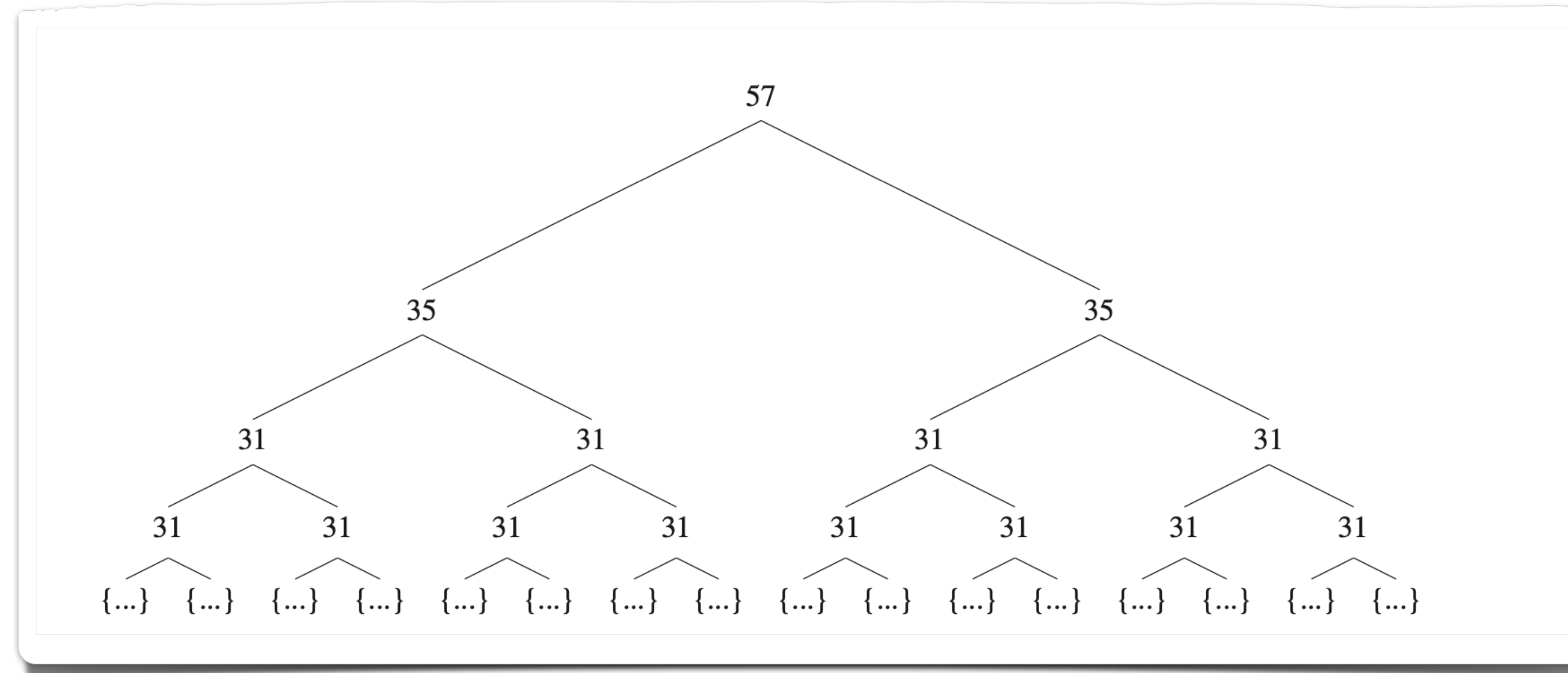
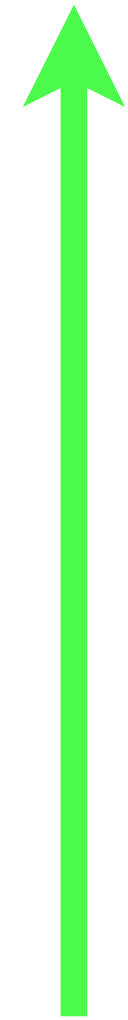
Enumerationsprinzip



- Exponentiell viele Fälle!
- Wie geht das systematisch?
- Wo kann man Arbeit sparen?

Enumerationsprinzip

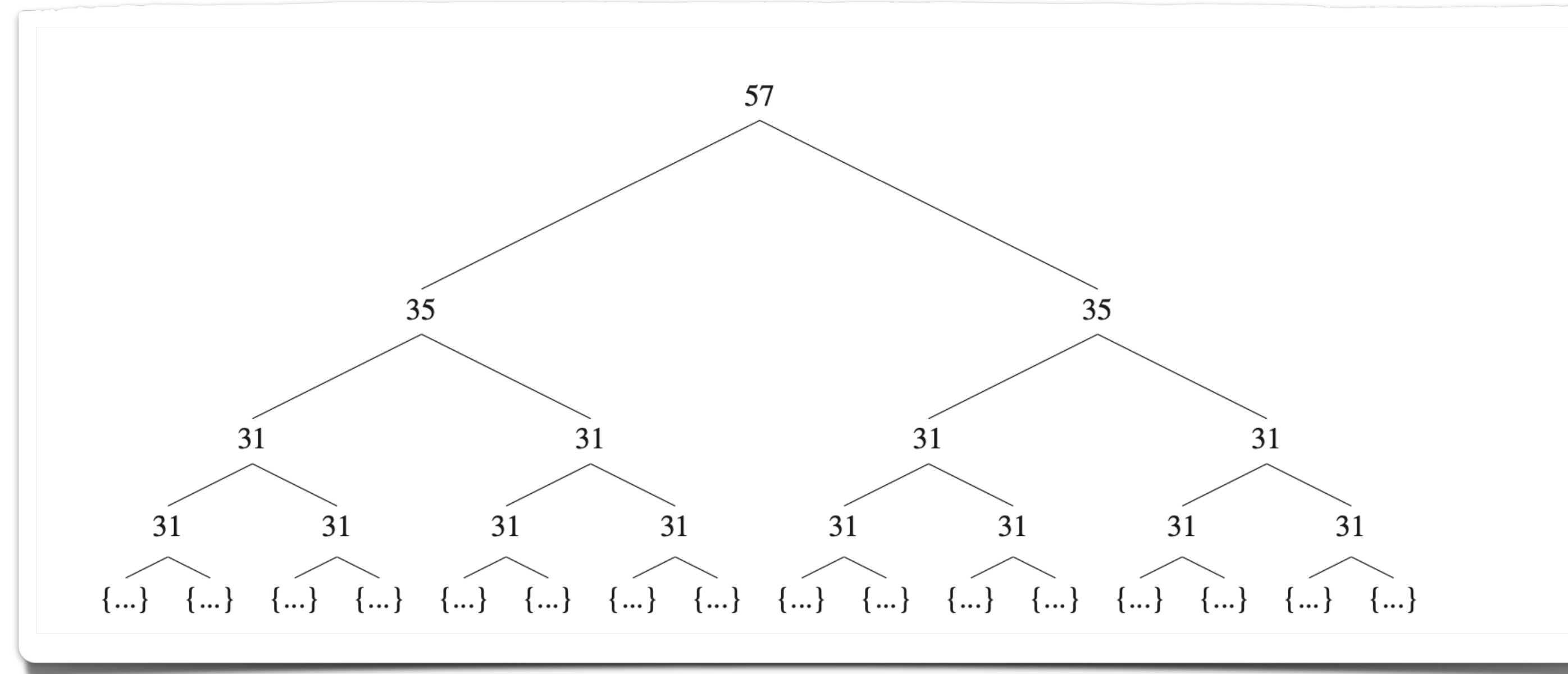
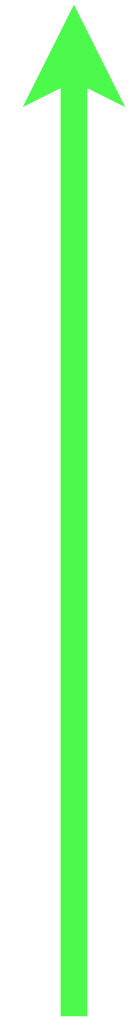
Dynamic
Programming



- Exponentiell viele Fälle!
- Wie geht das systematisch?
- Wo kann man Arbeit sparen?

Enumerationsprinzip

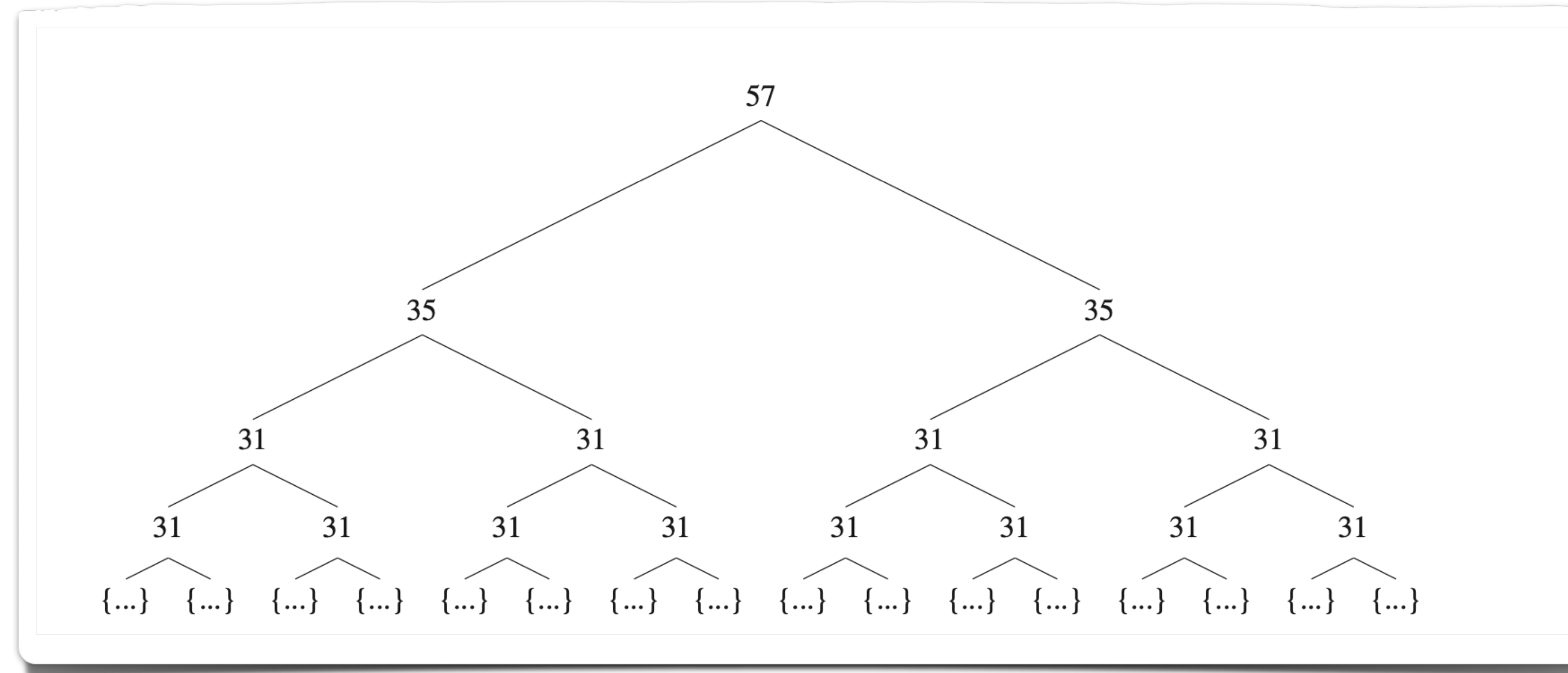
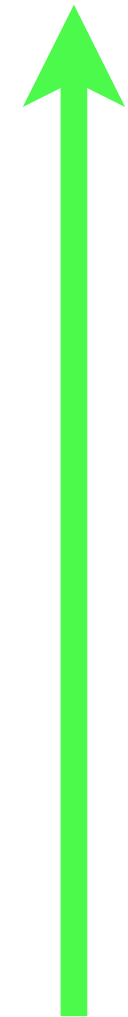
Dynamic
Programming



- Exponentiell viele Fälle!
- Wie geht das systematisch?
- Wo kann man Arbeit sparen?

Enumerationsprinzip

Dynamic
Programming



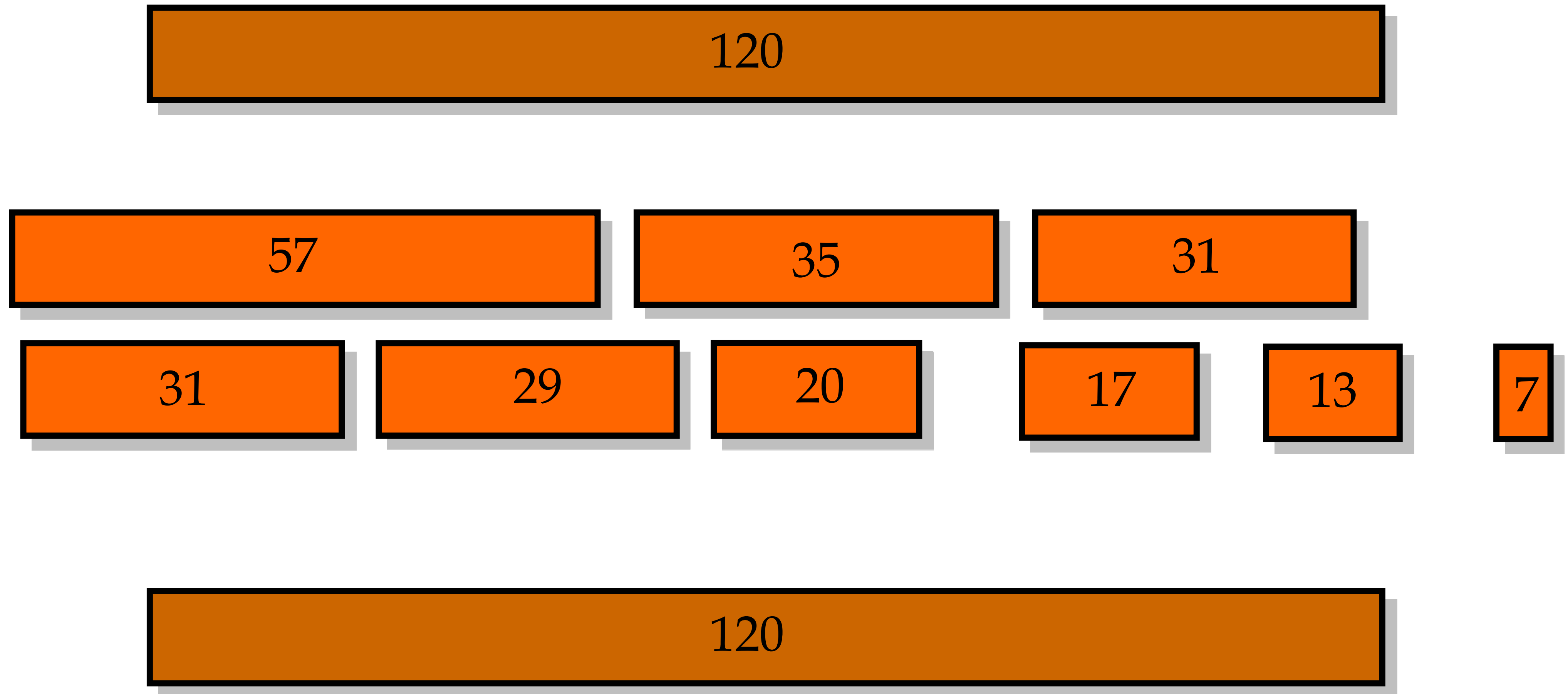
Branch and
Bound



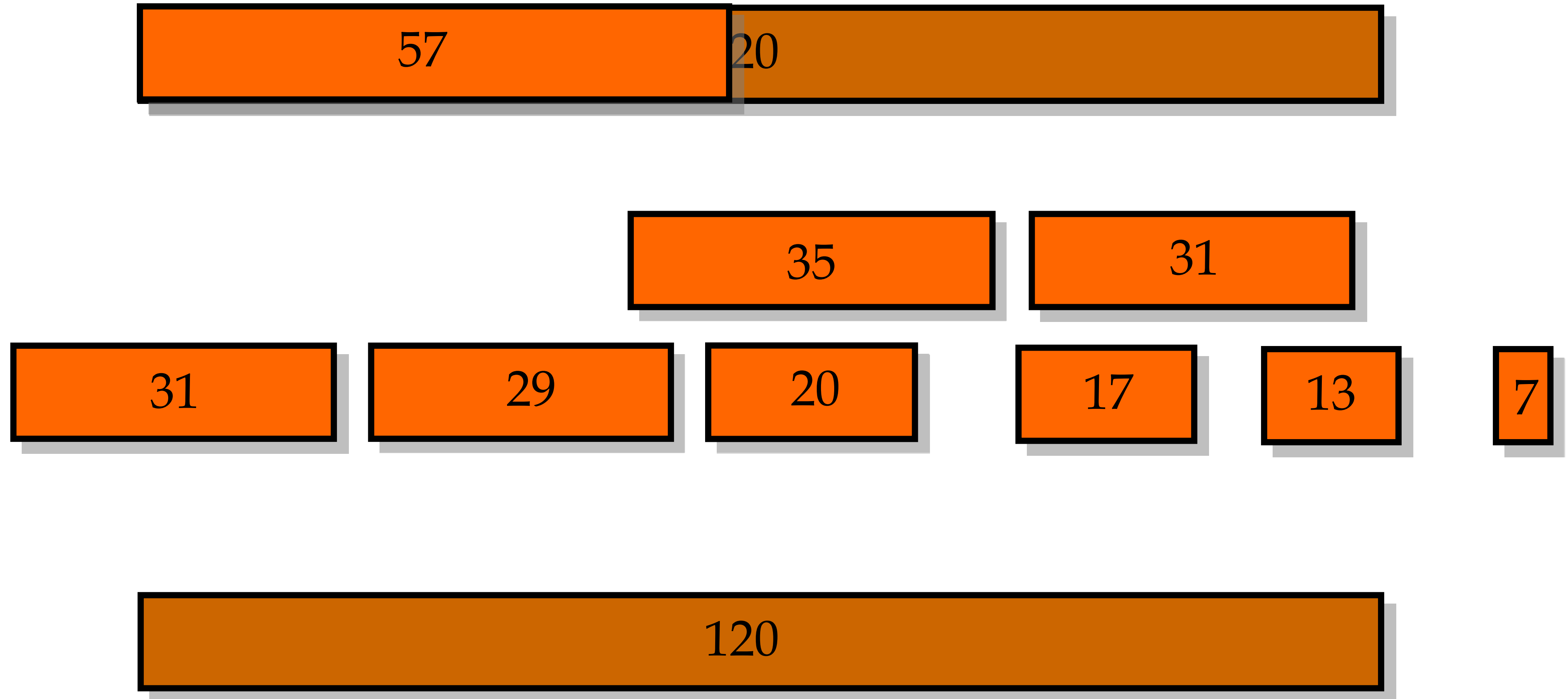
- Exponentiell viele Fälle!
- Wie geht das systematisch?
- Wo kann man Arbeit sparen?

Zur Erinnerung: Subset Sum

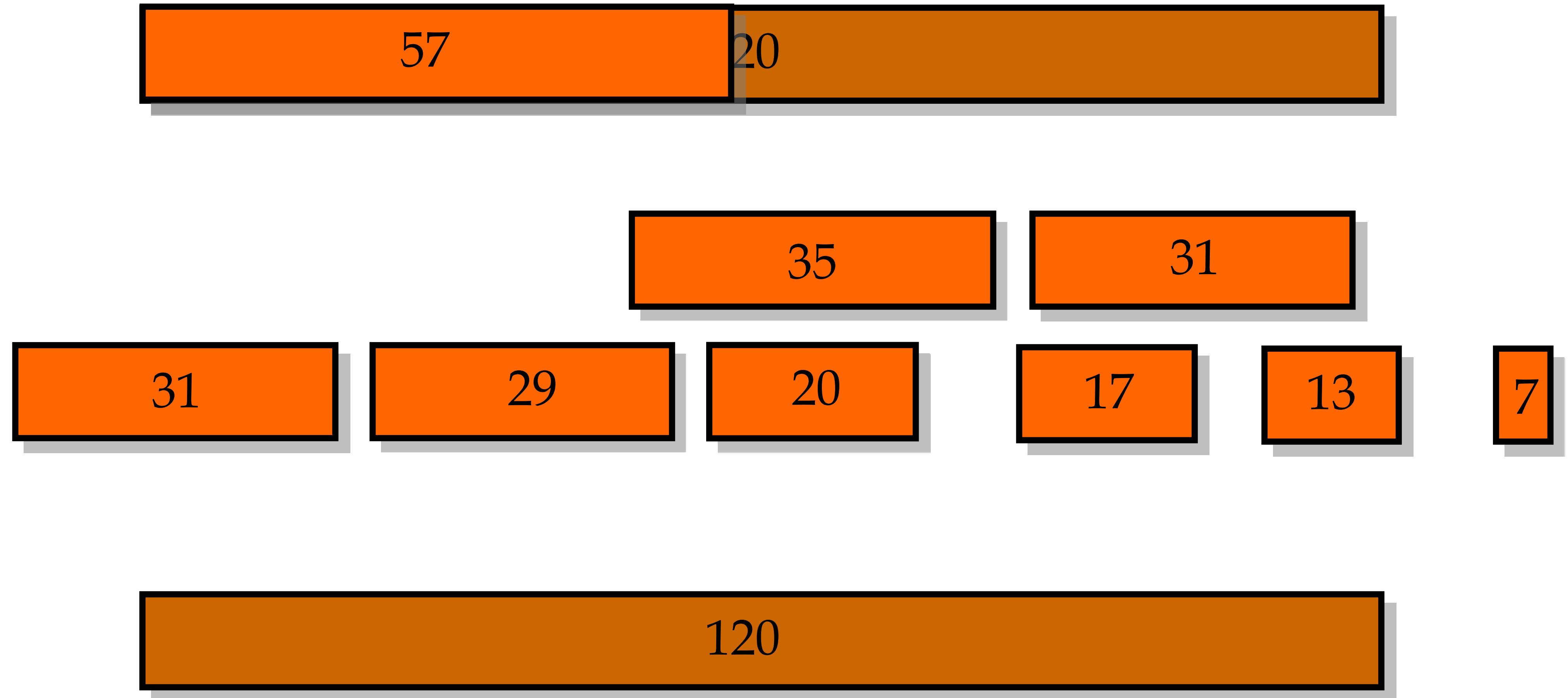
Zur Erinnerung: Subset Sum



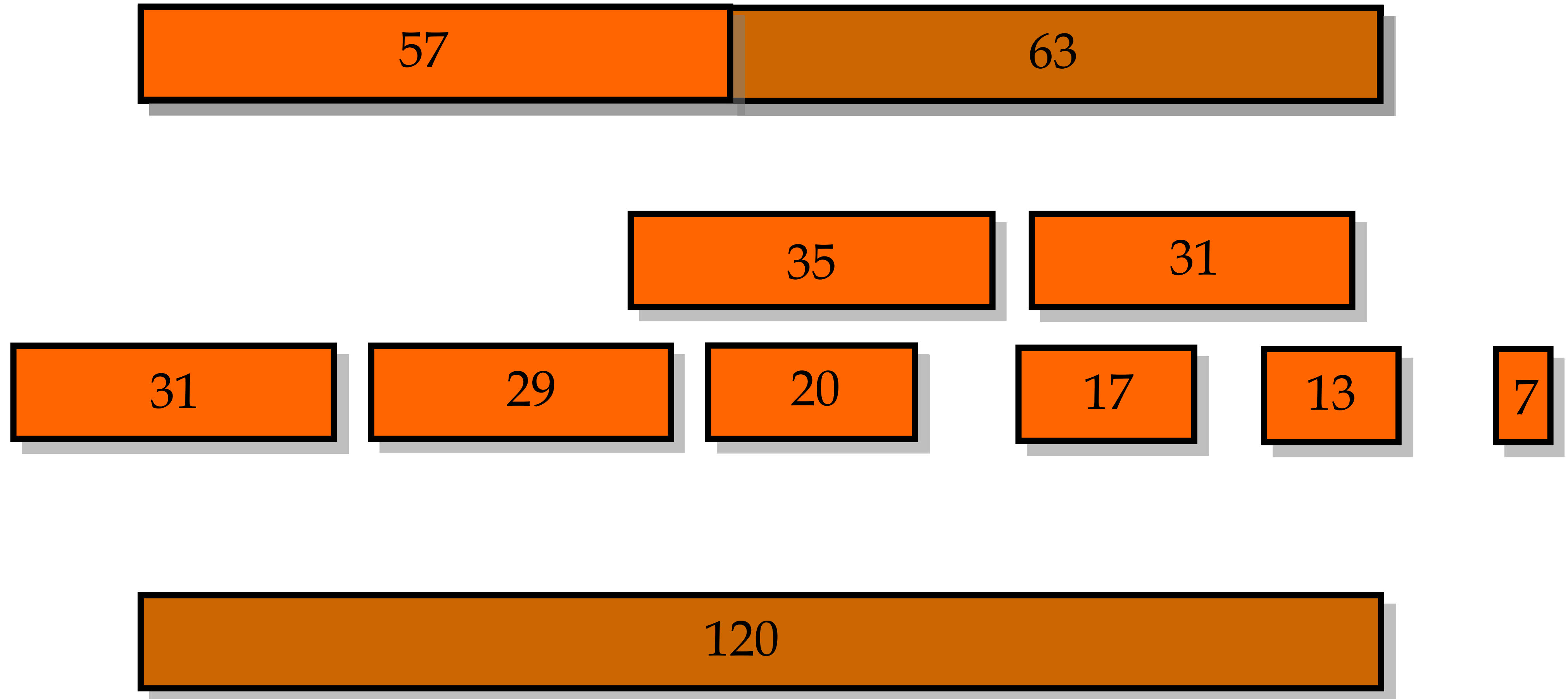
Zur Erinnerung: Subset Sum



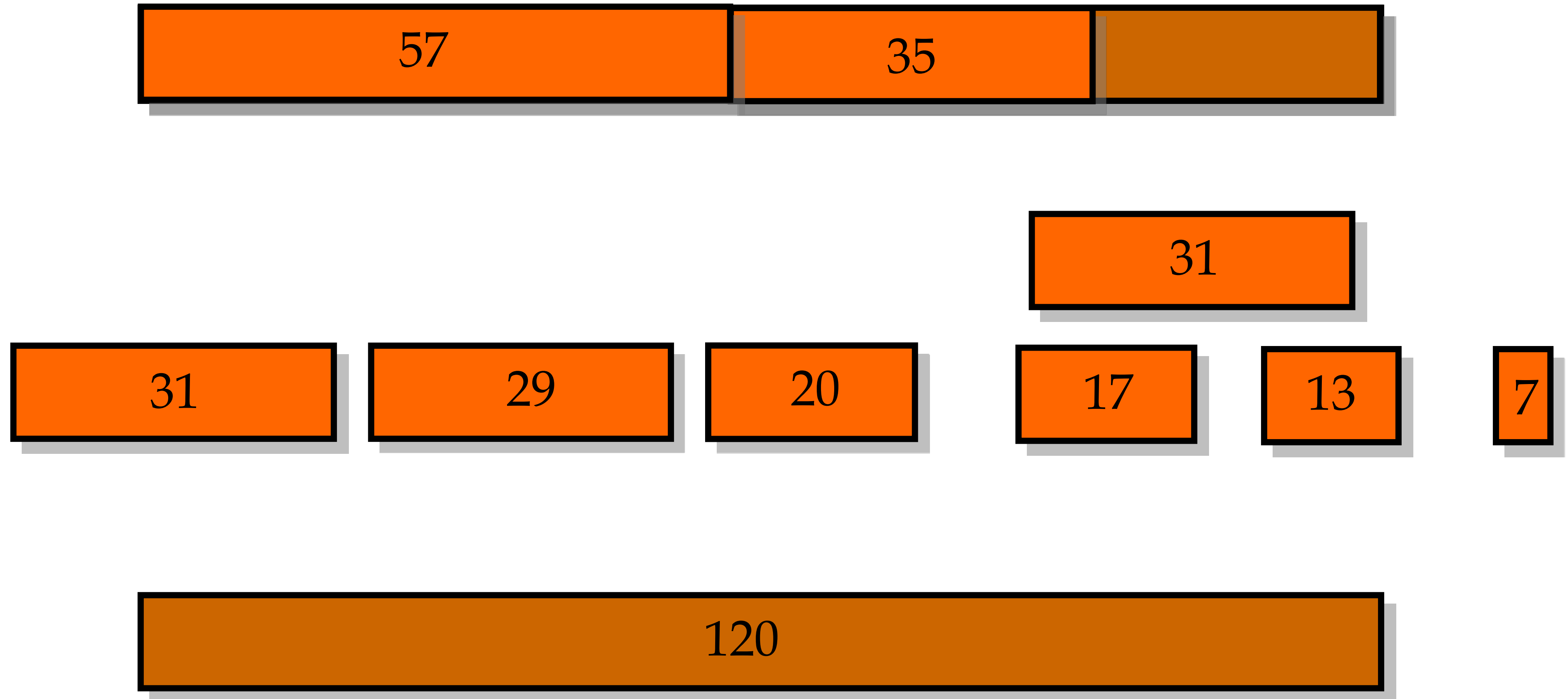
Zur Erinnerung: Subset Sum



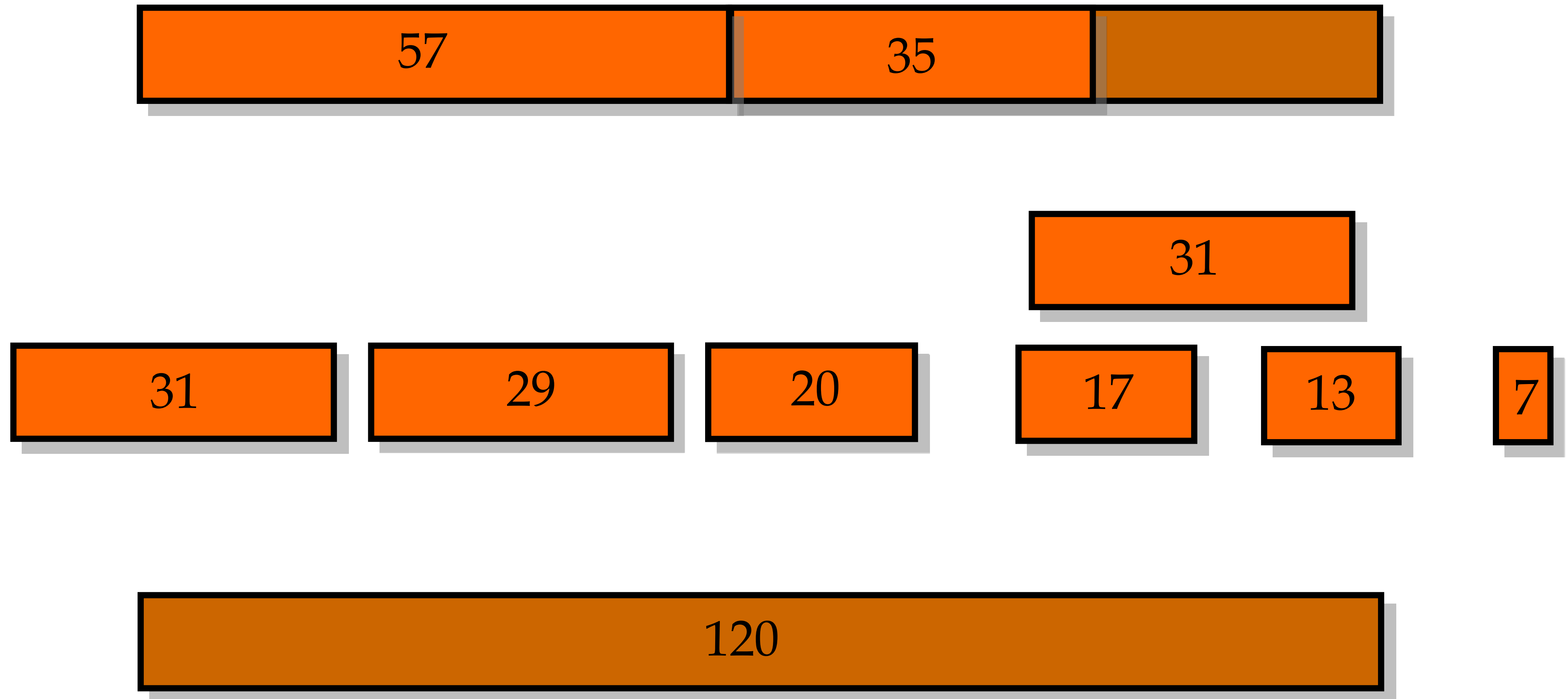
Zur Erinnerung: Subset Sum



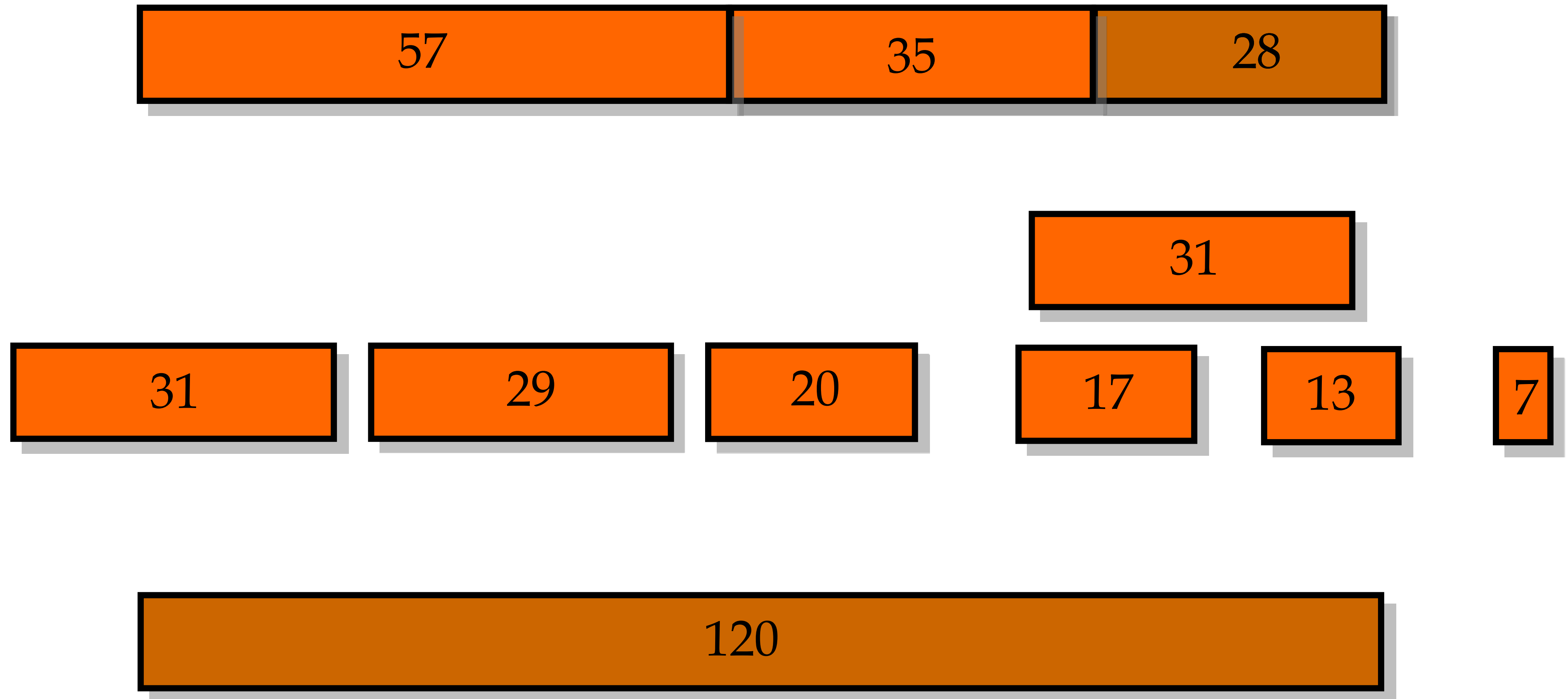
Zur Erinnerung: Subset Sum



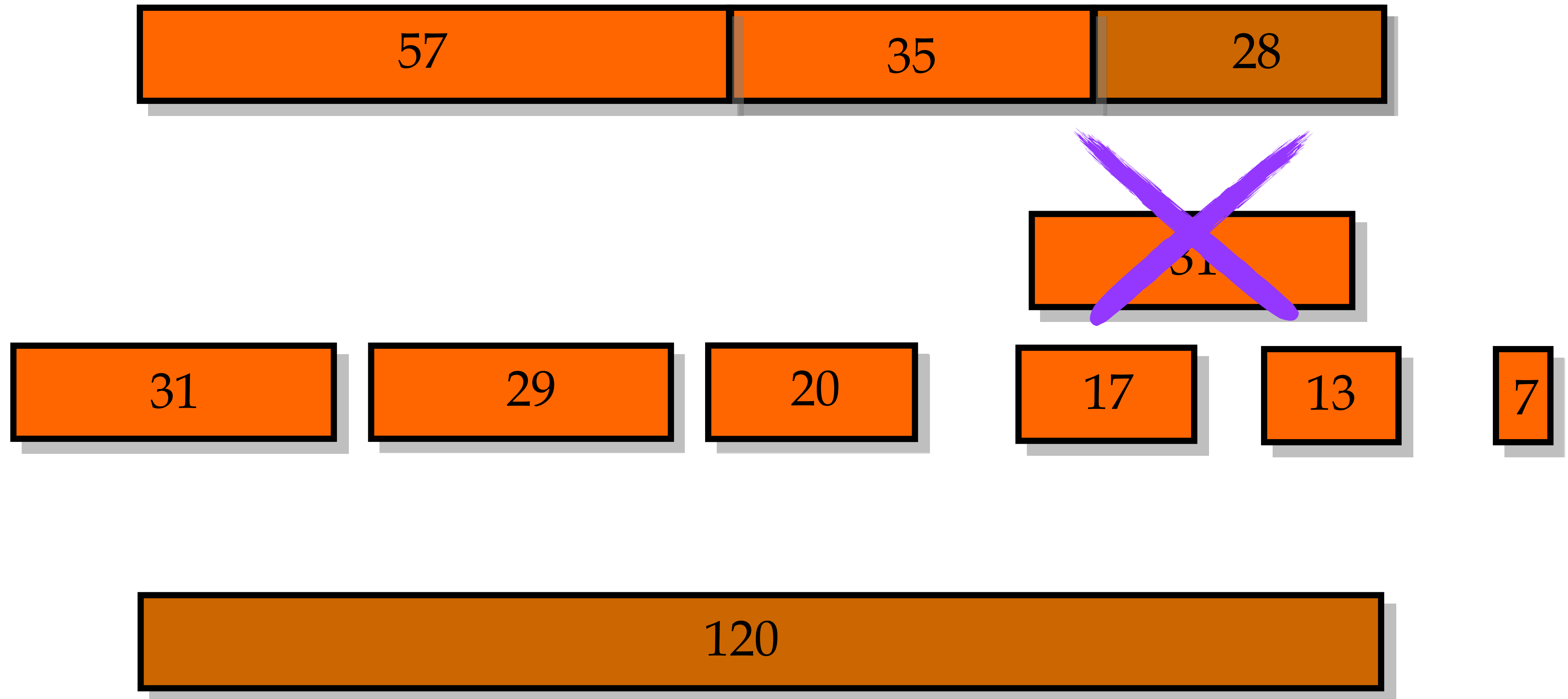
Zur Erinnerung: Subset Sum



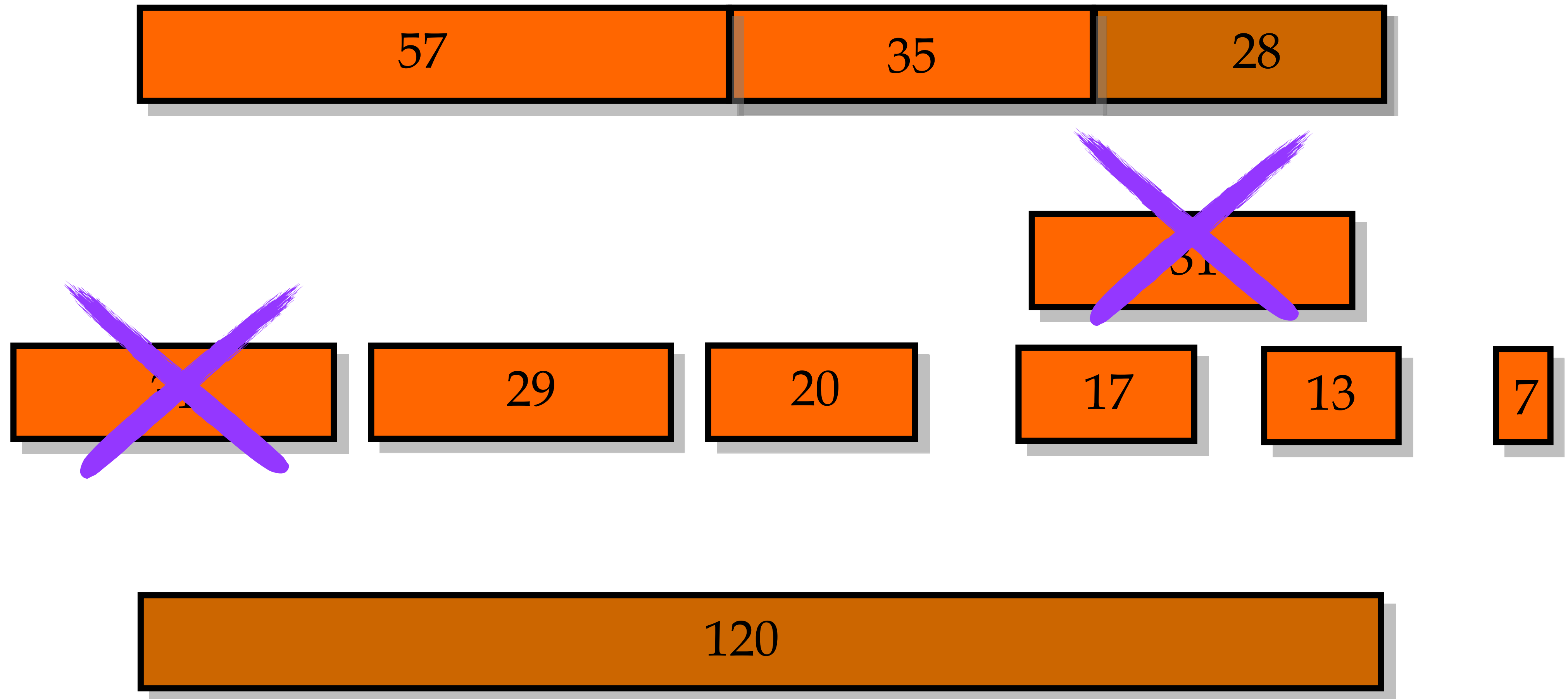
Zur Erinnerung: Subset Sum



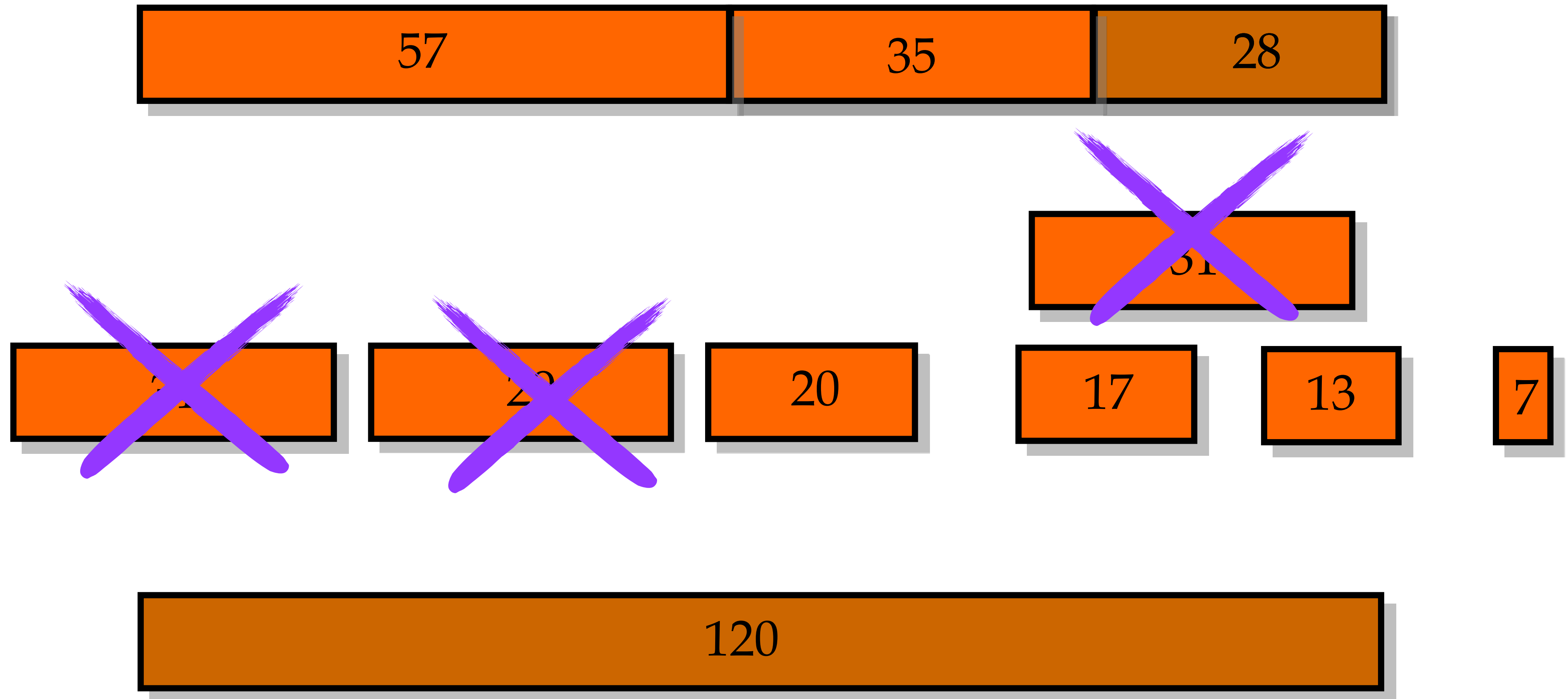
Zur Erinnerung: Subset Sum



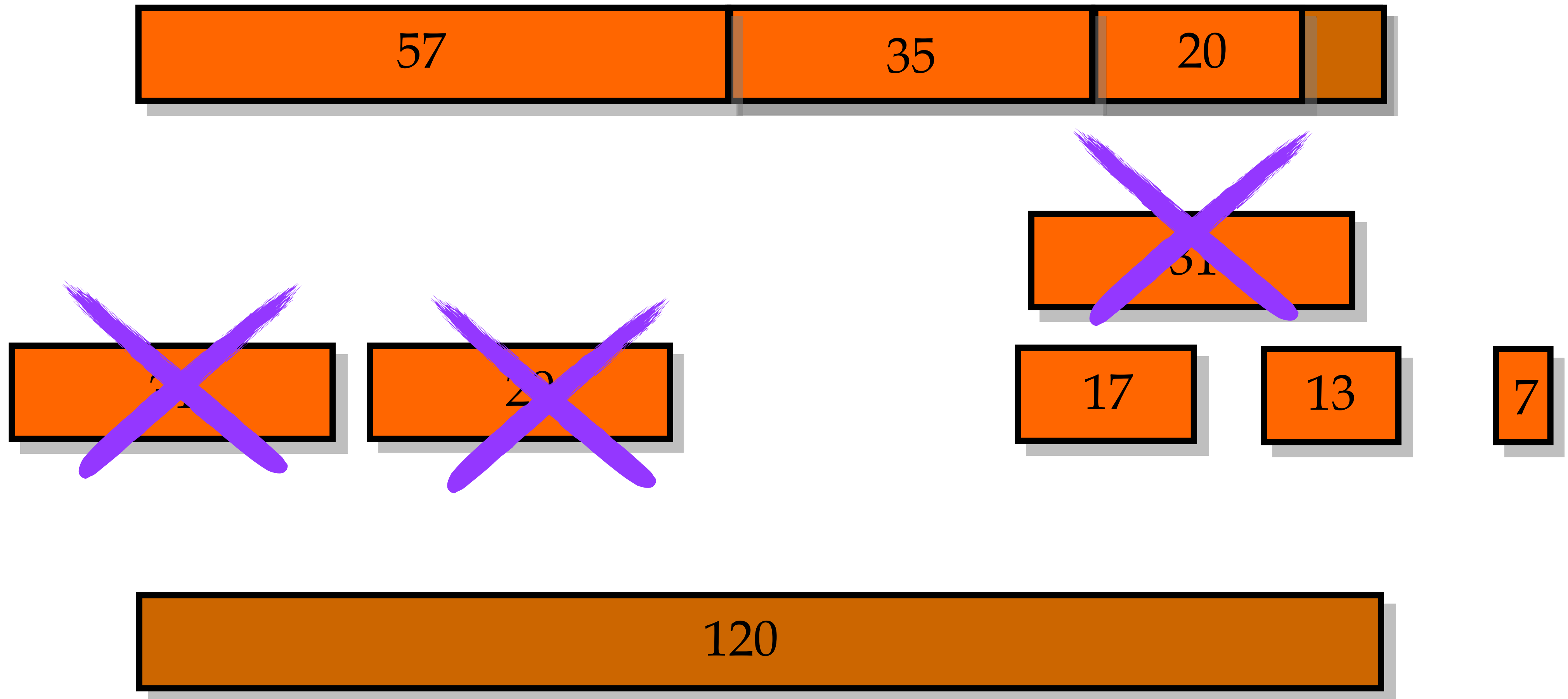
Zur Erinnerung: Subset Sum



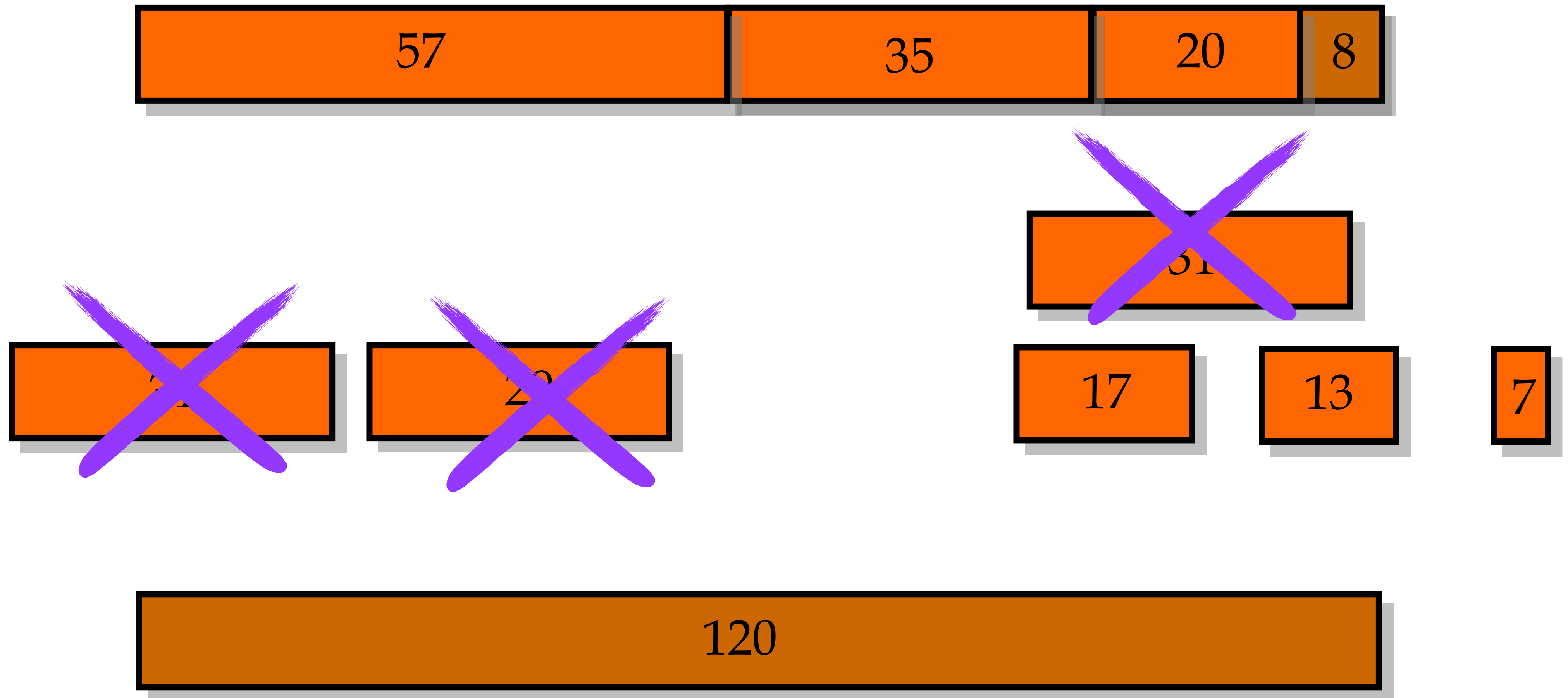
Zur Erinnerung: Subset Sum



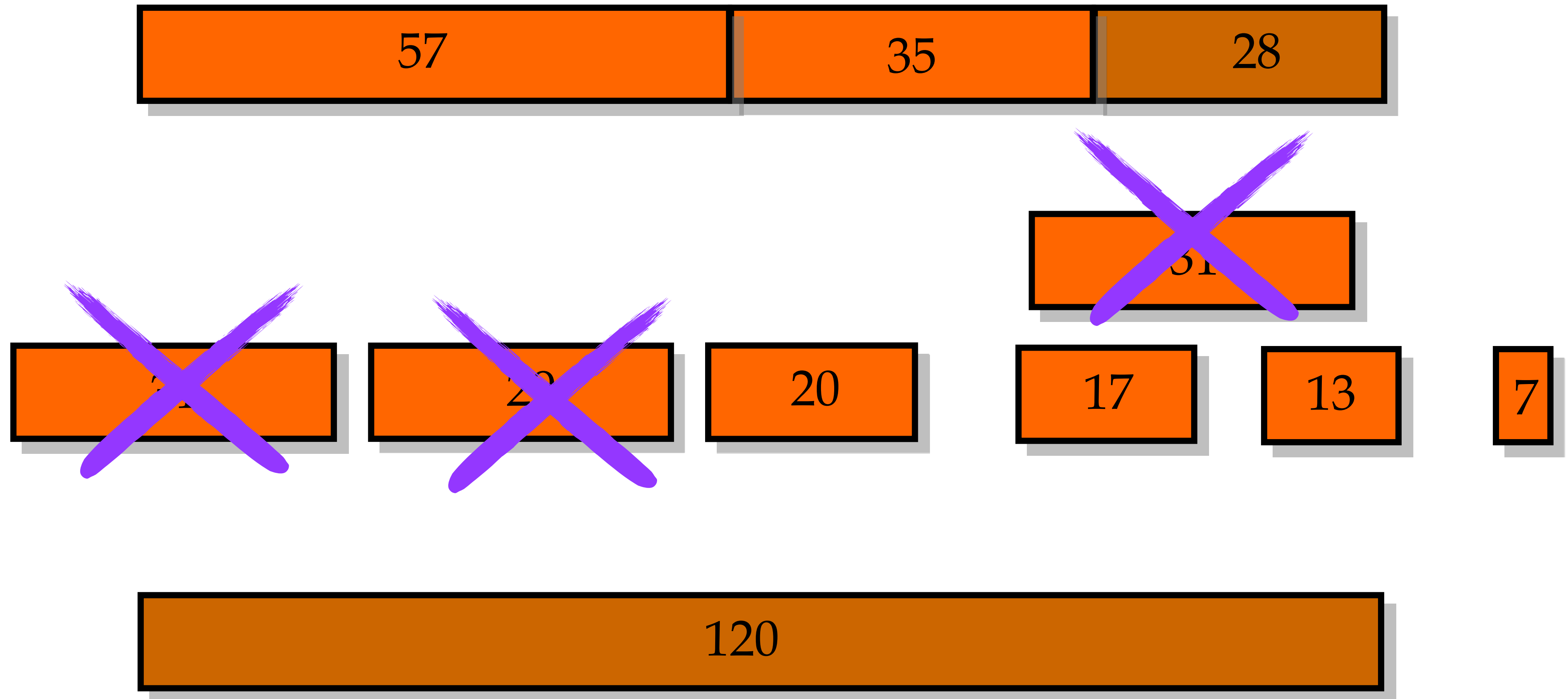
Zur Erinnerung: Subset Sum



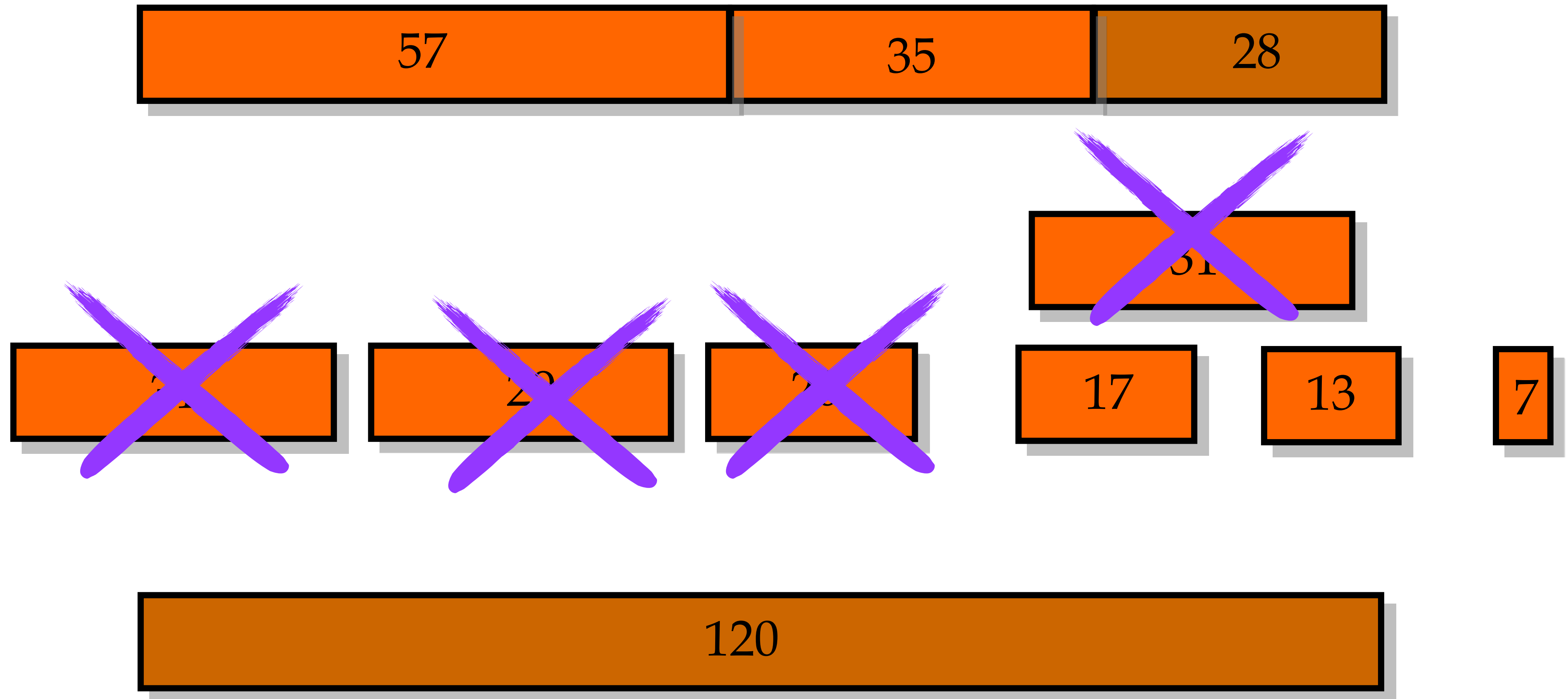
Zur Erinnerung: Subset Sum



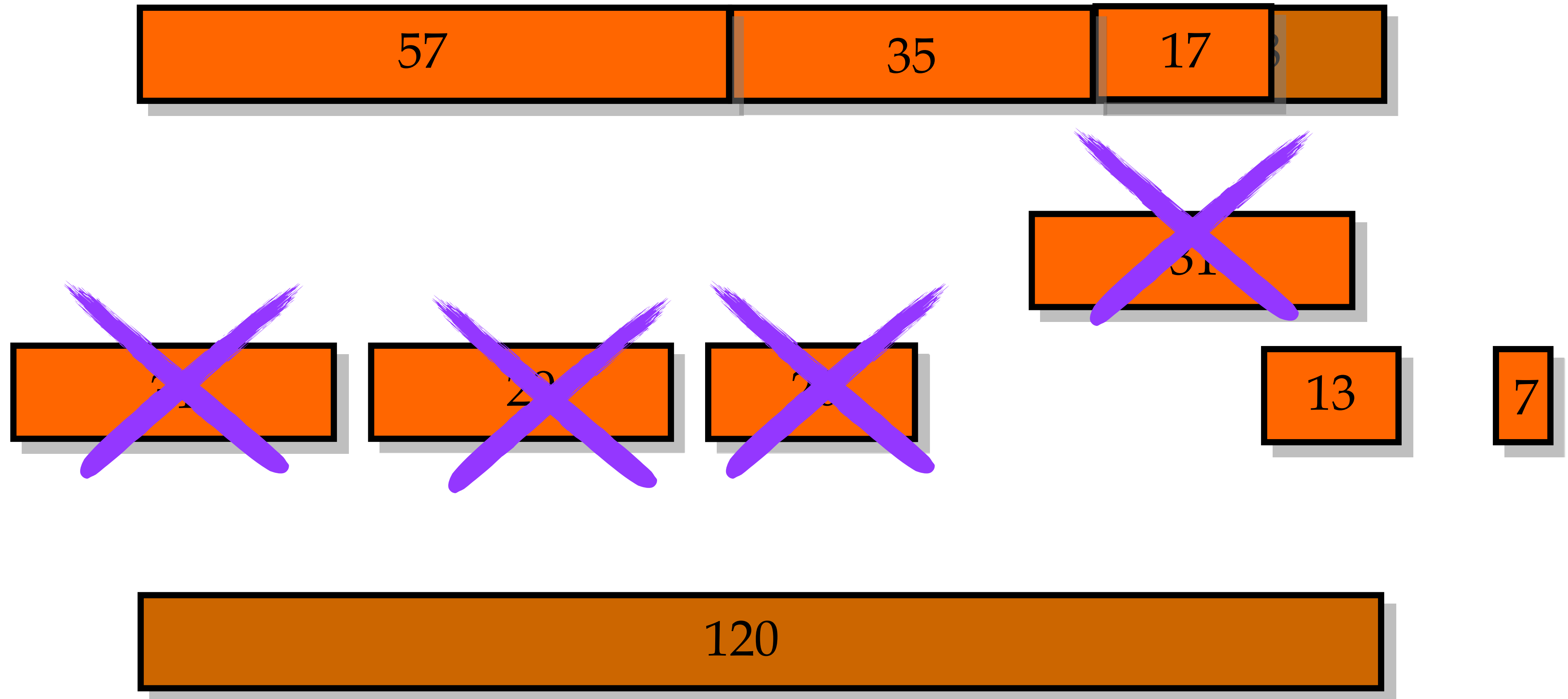
Zur Erinnerung: Subset Sum



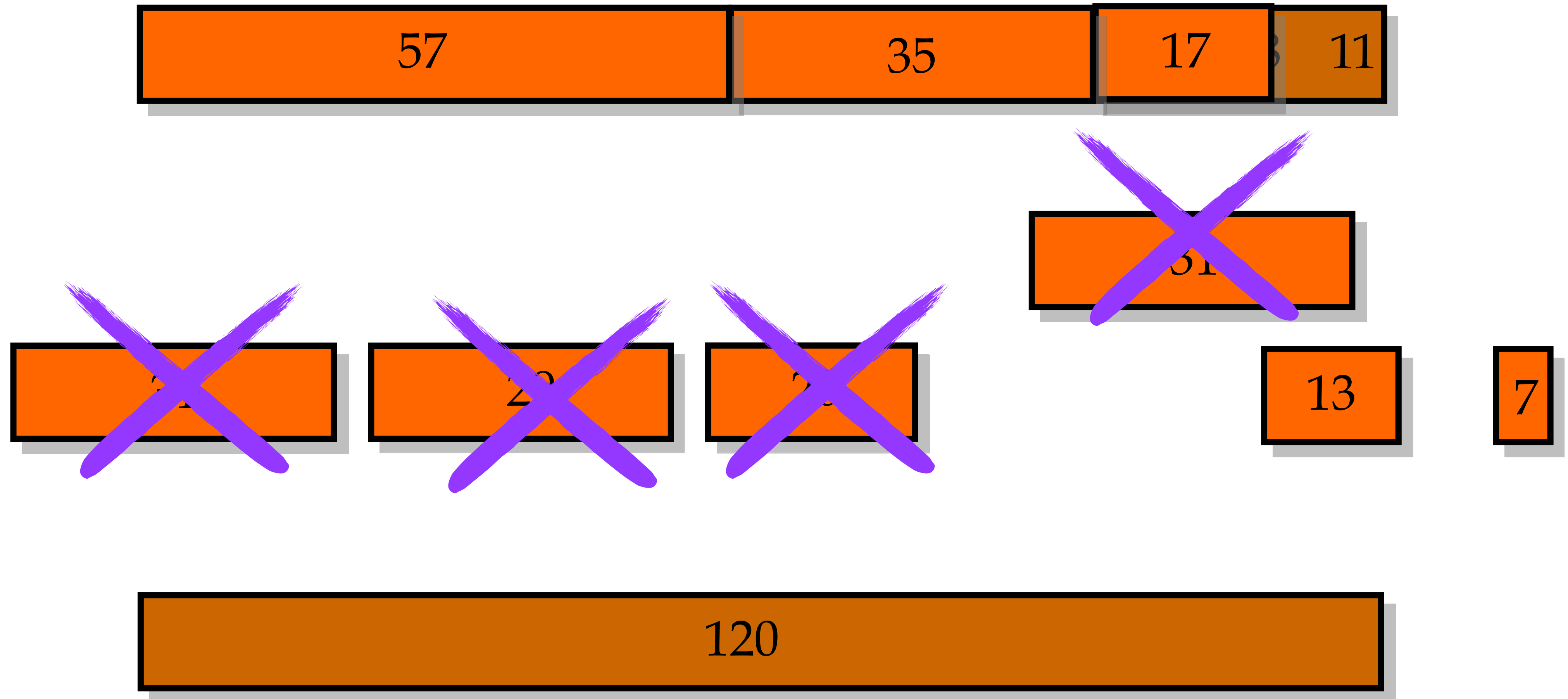
Zur Erinnerung: Subset Sum



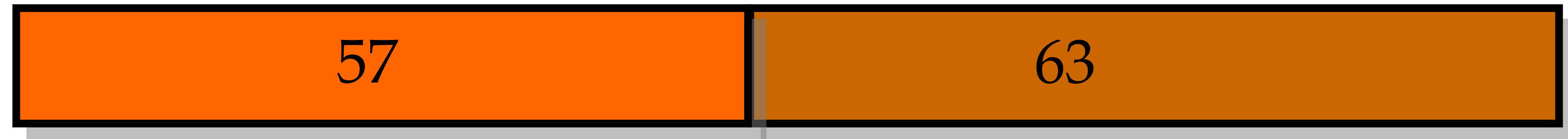
Zur Erinnerung: Subset Sum



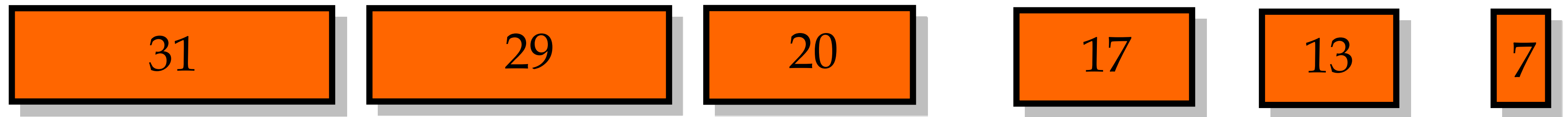
Zur Erinnerung: Subset Sum



Zur Erinnerung: Subset Sum



!

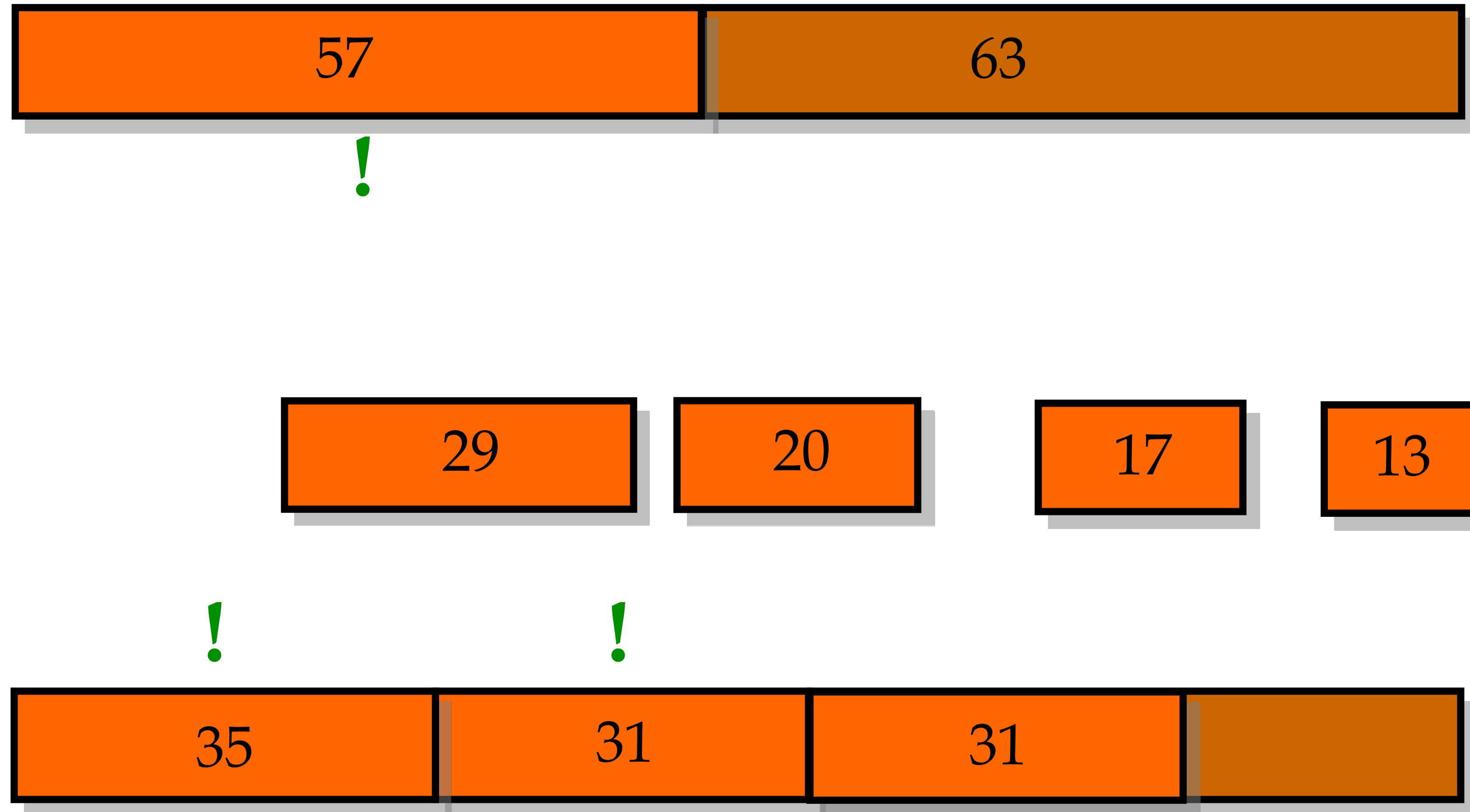


!

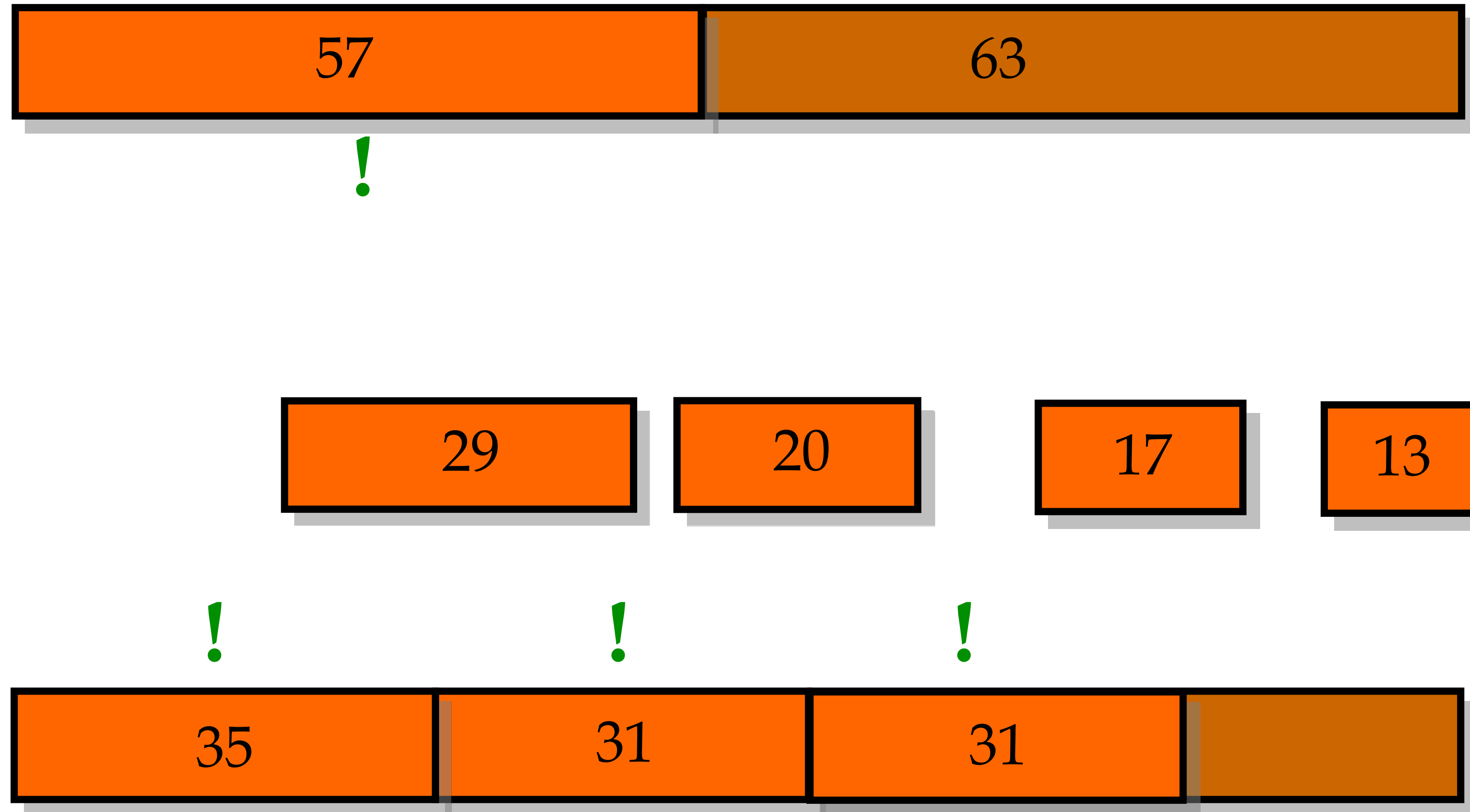
!



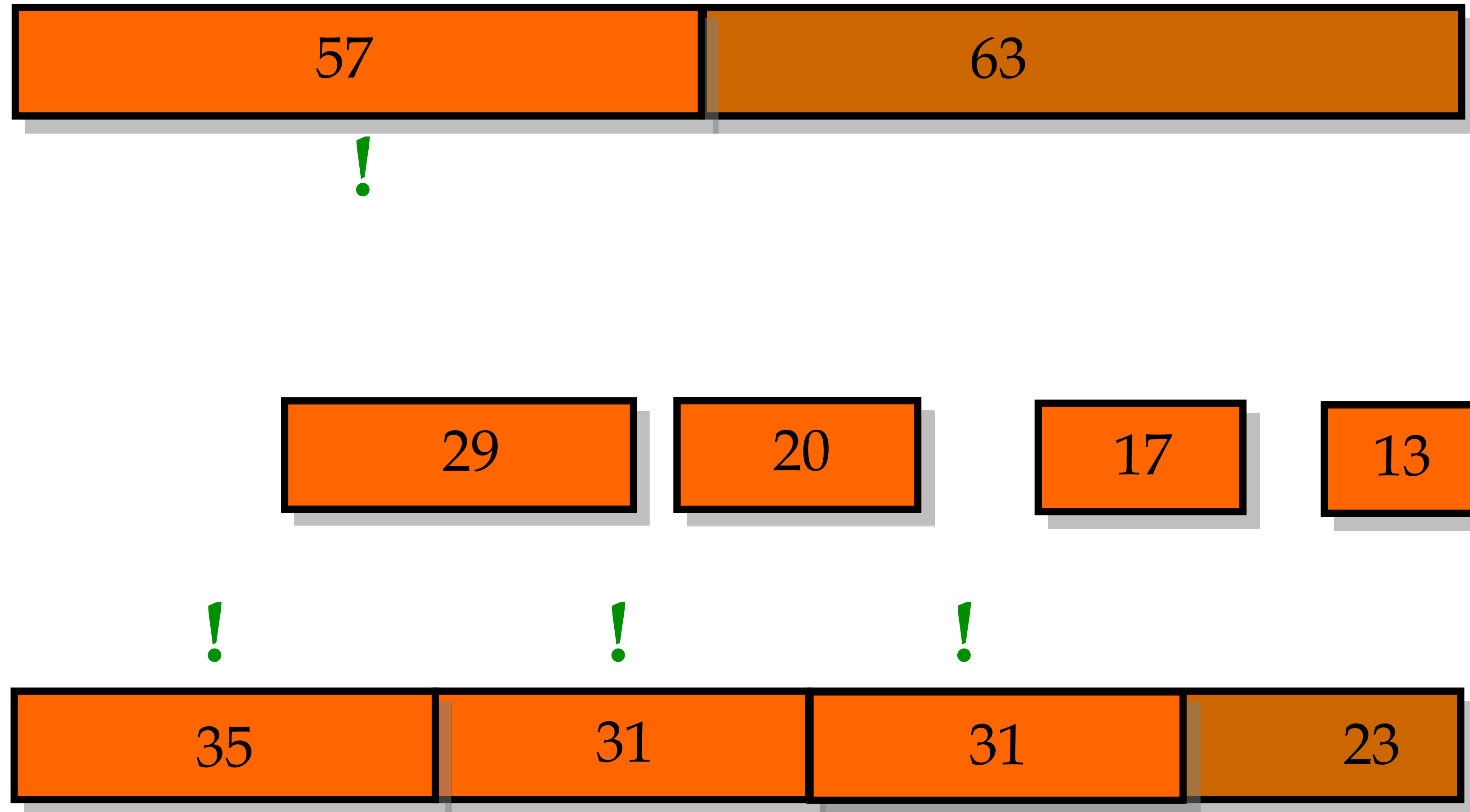
Zur Erinnerung: Subset Sum



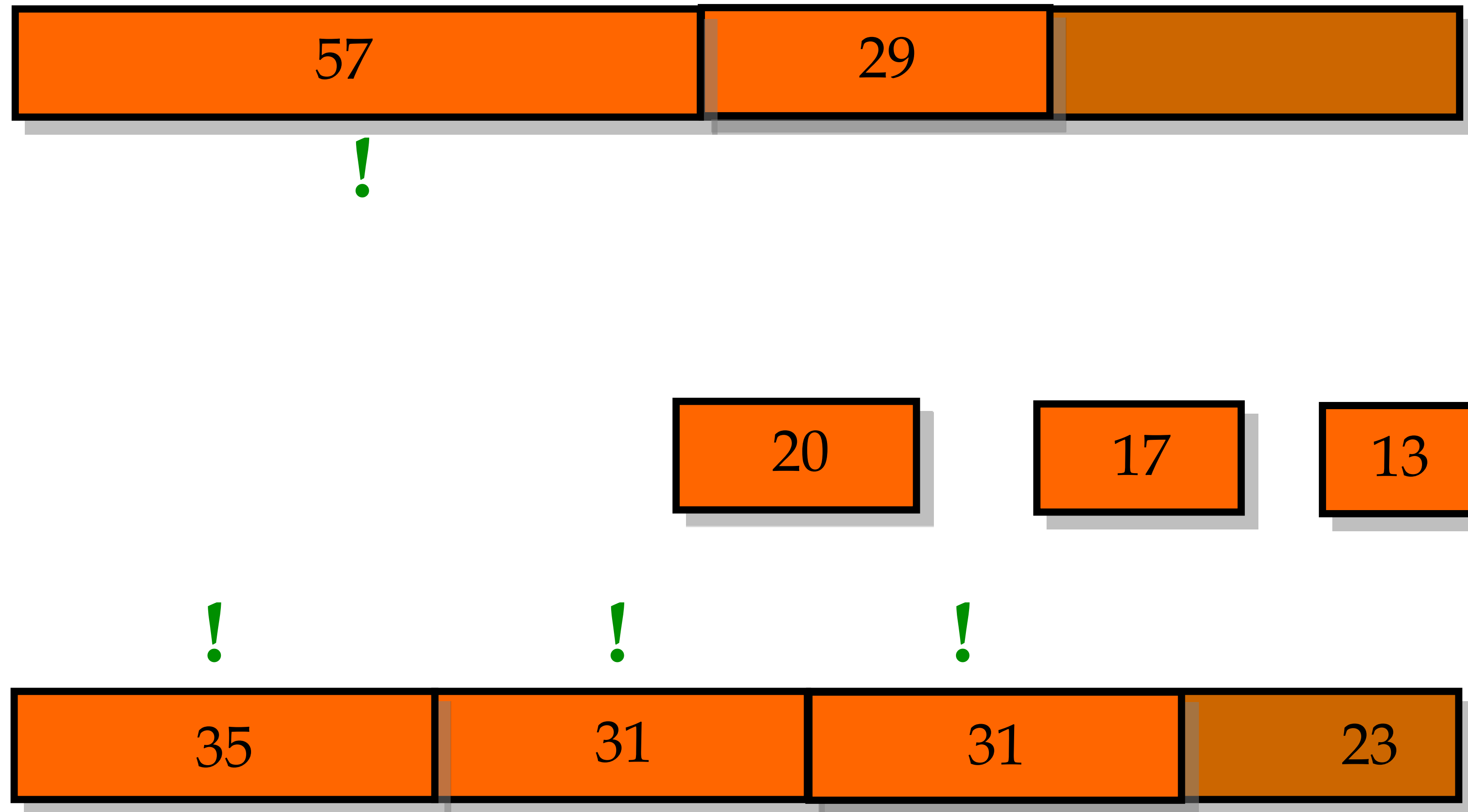
Zur Erinnerung: Subset Sum



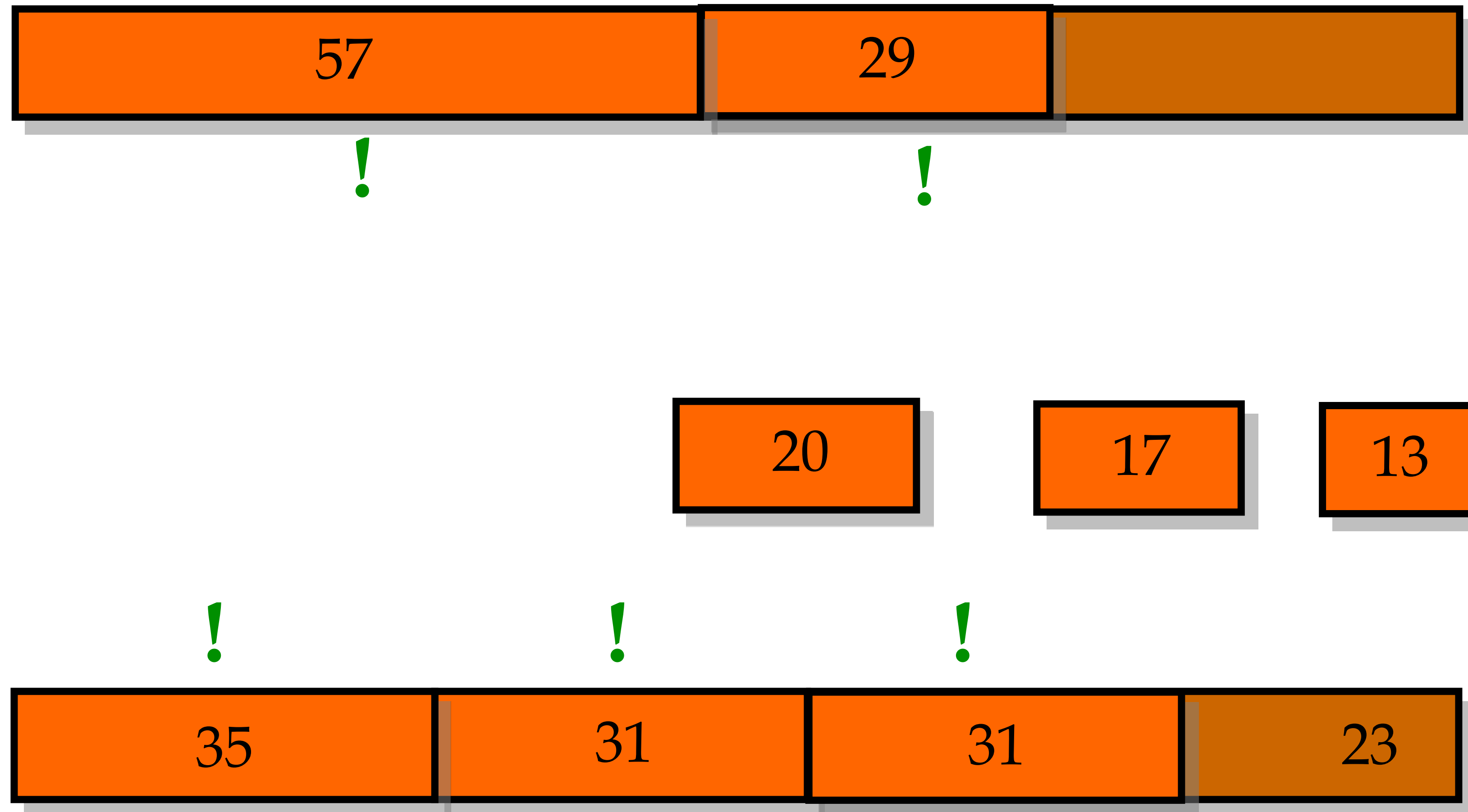
Zur Erinnerung: Subset Sum



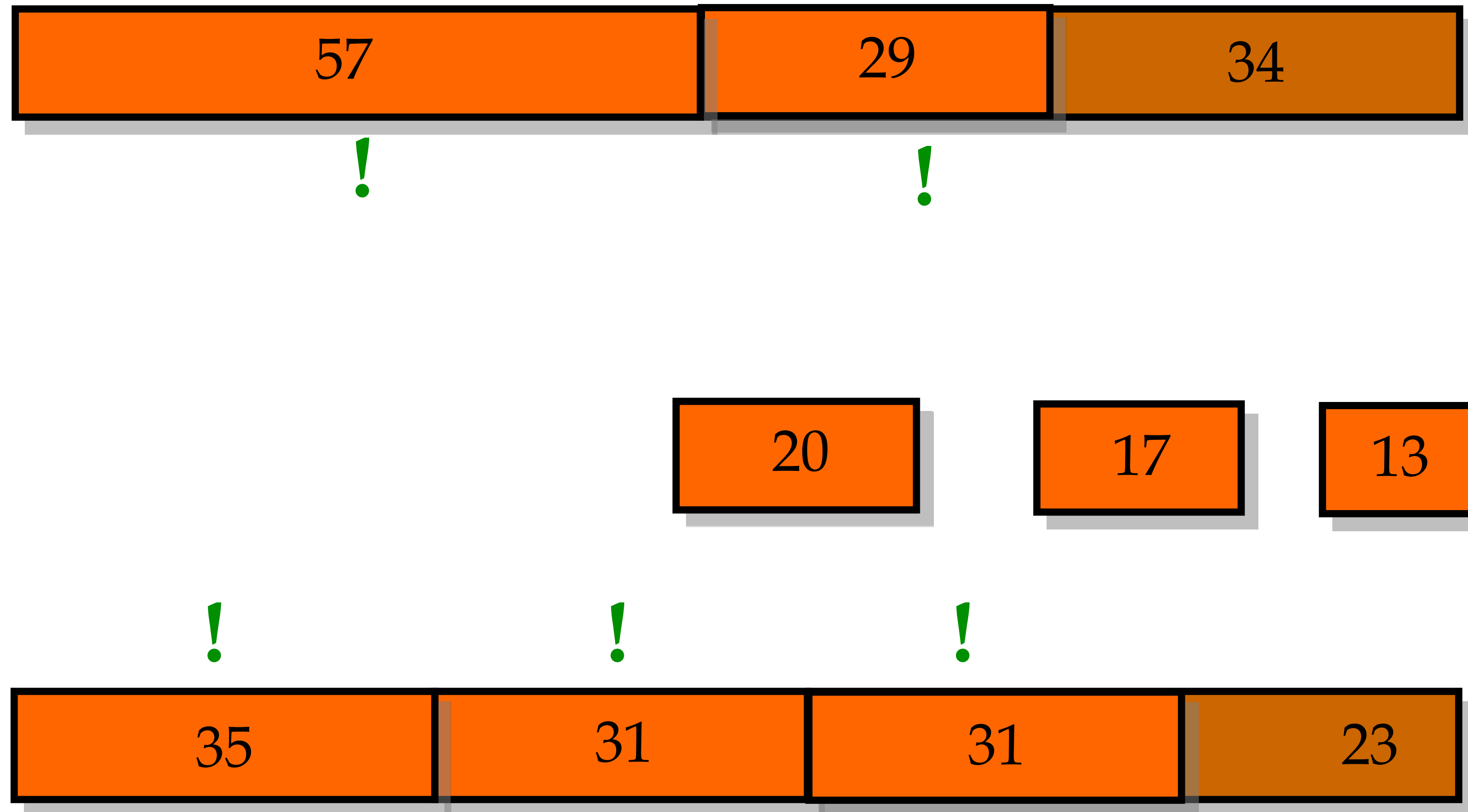
Zur Erinnerung: Subset Sum



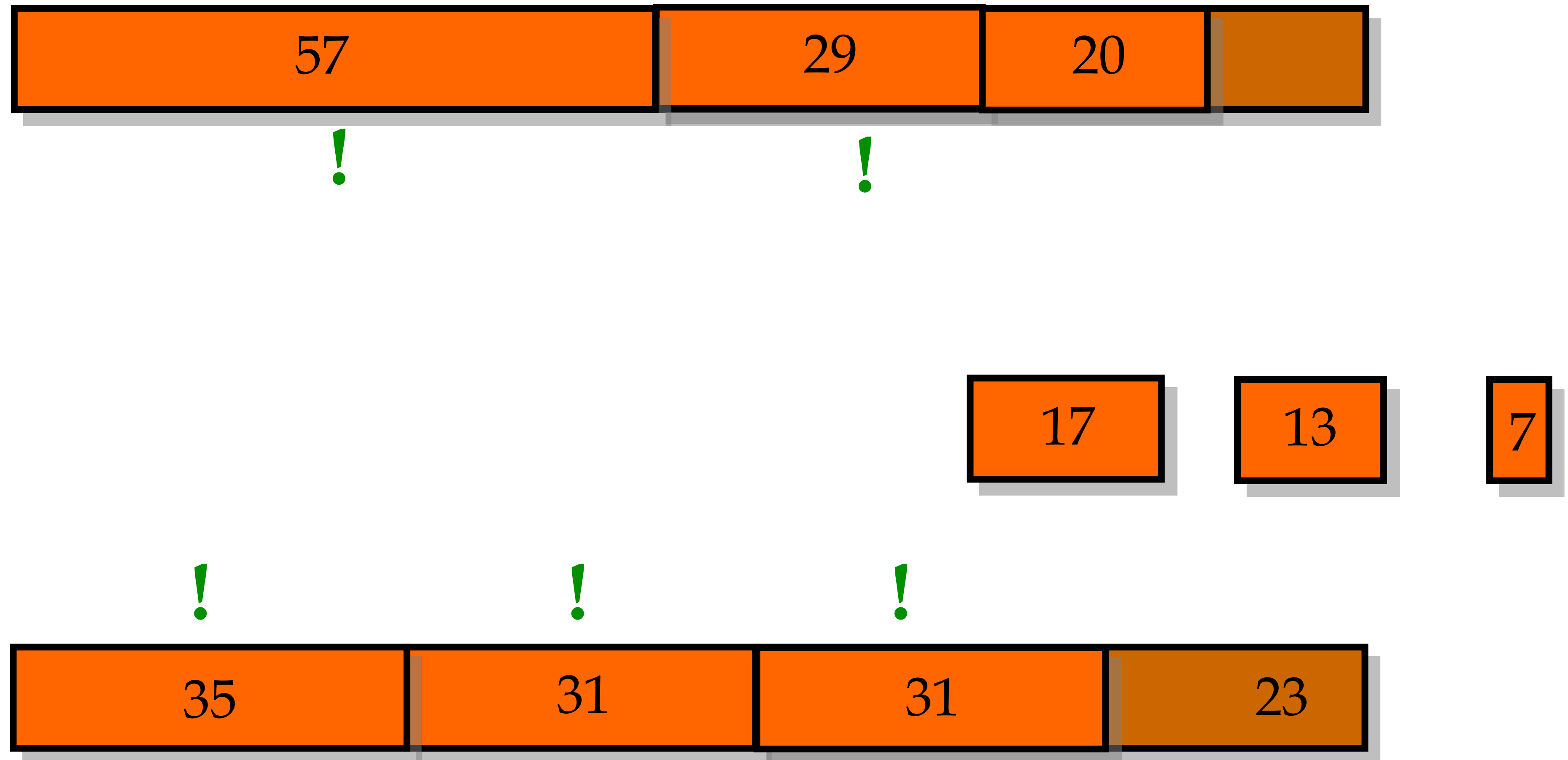
Zur Erinnerung: Subset Sum



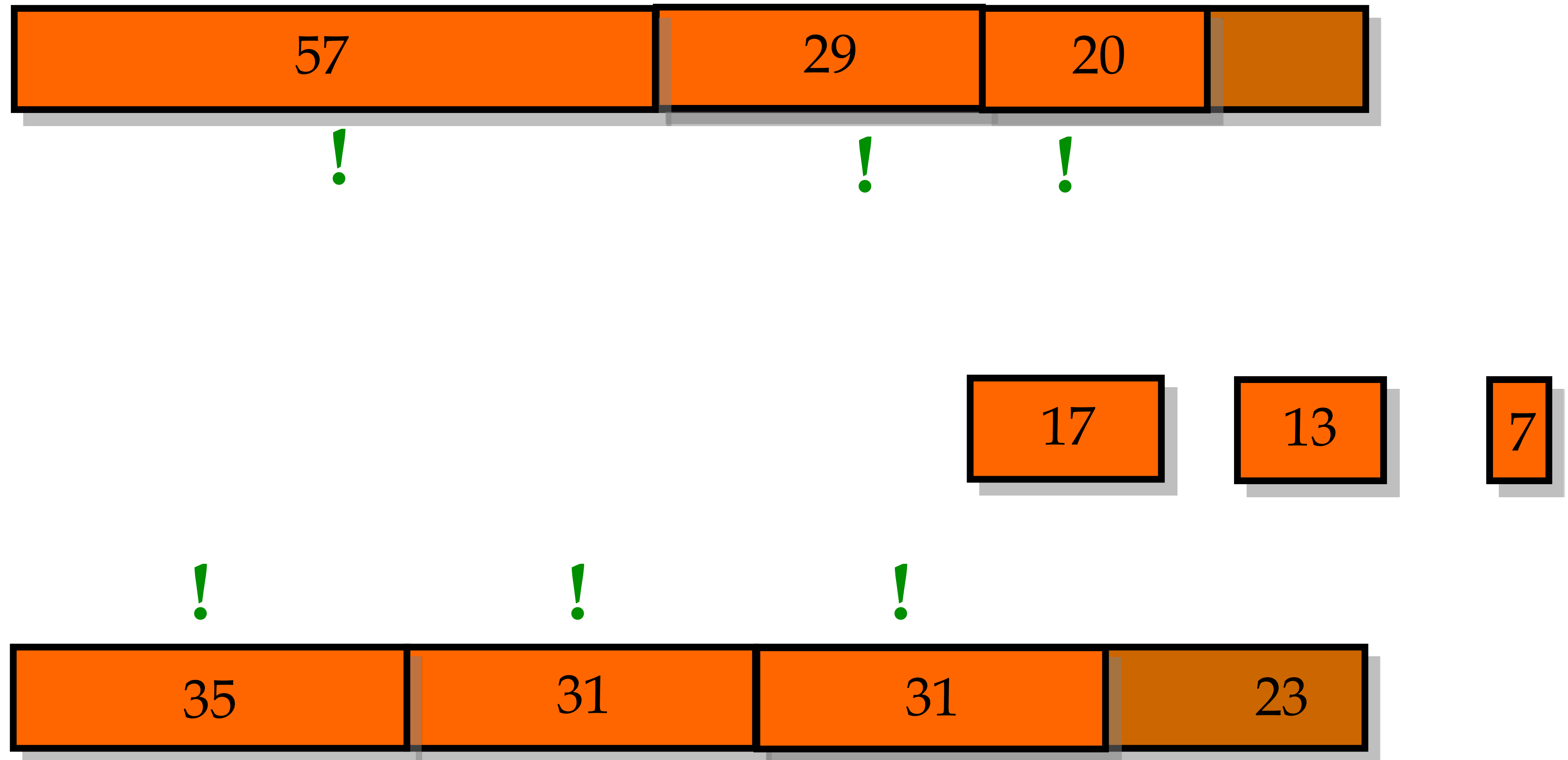
Zur Erinnerung: Subset Sum



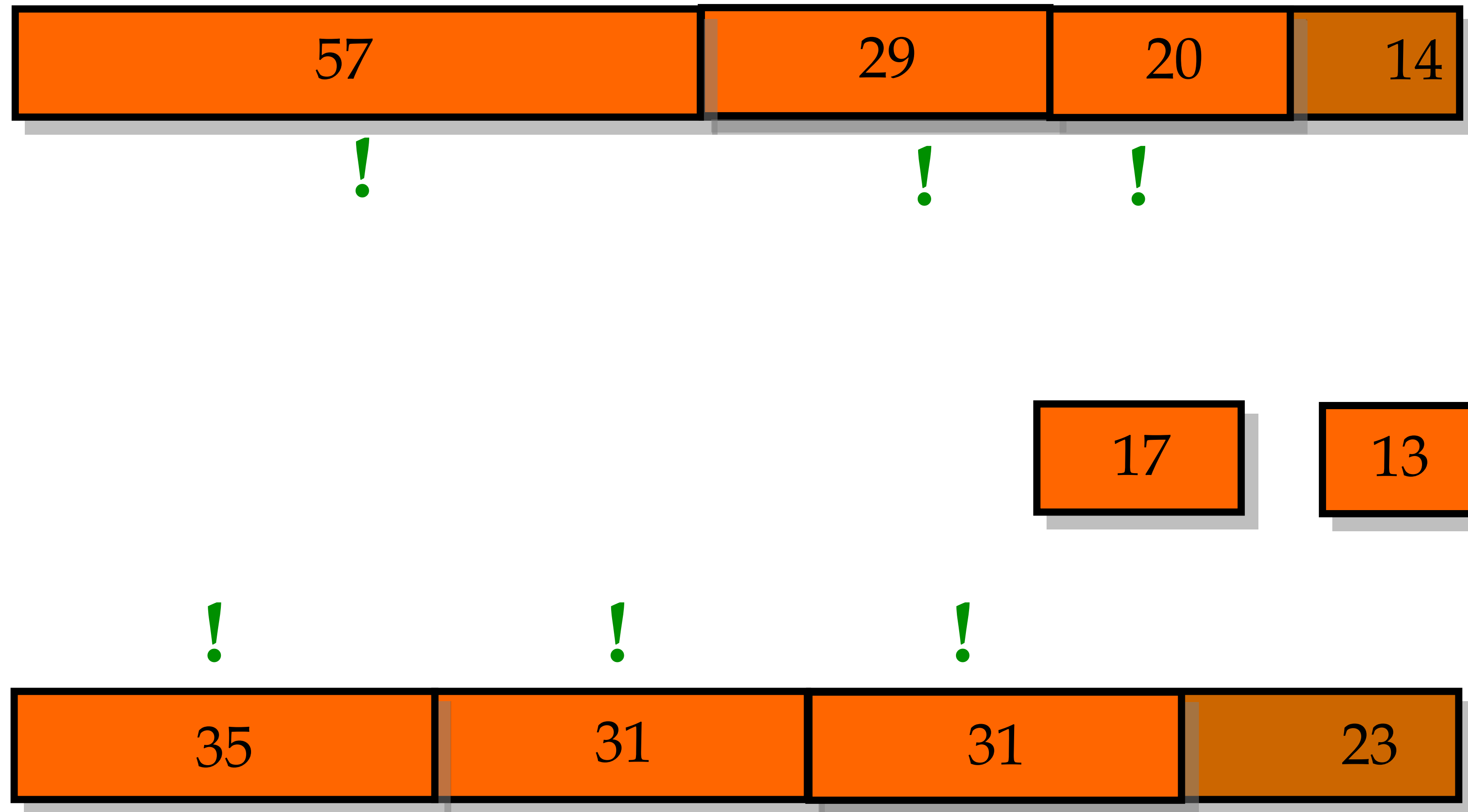
Zur Erinnerung: Subset Sum



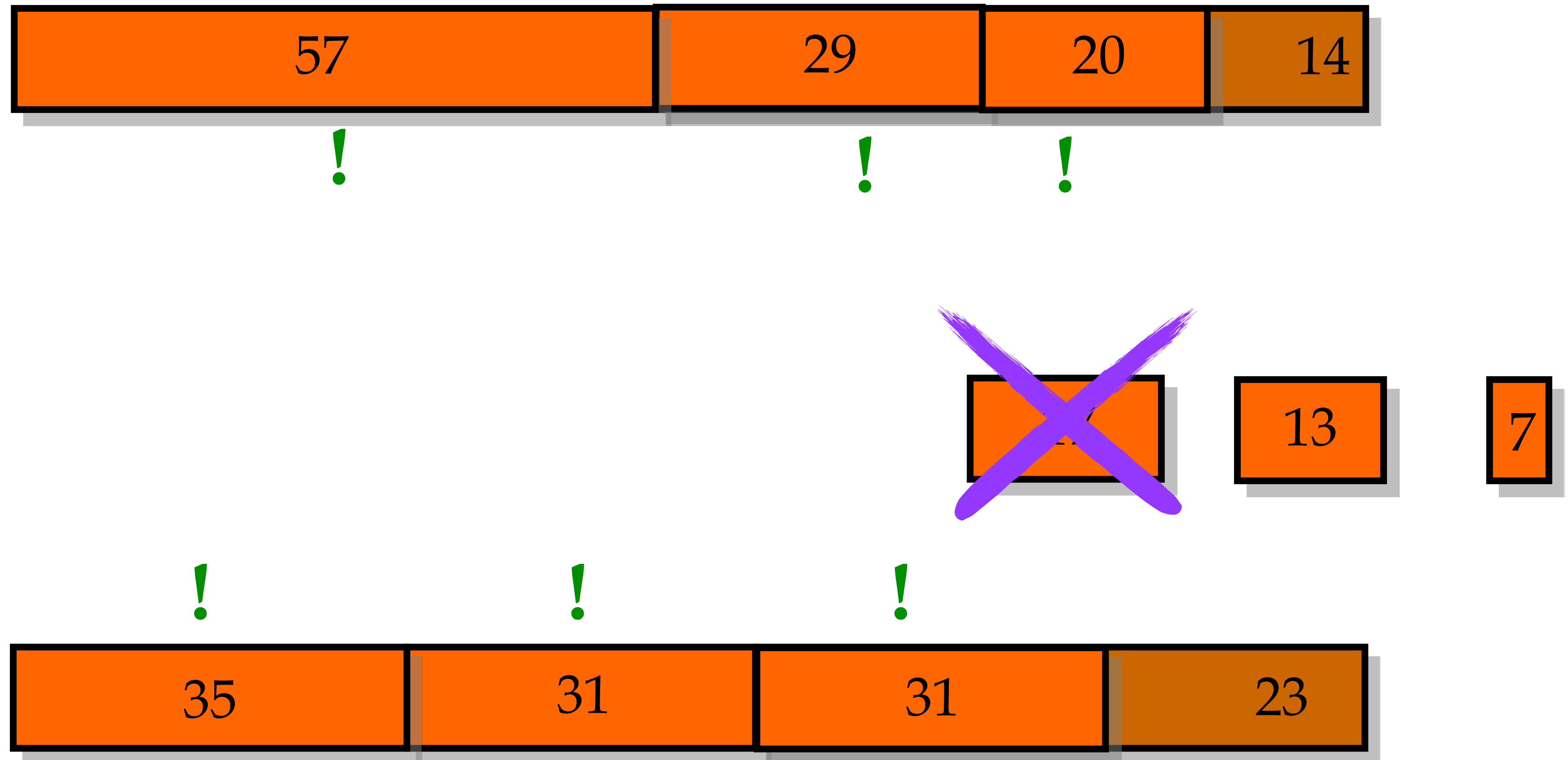
Zur Erinnerung: Subset Sum



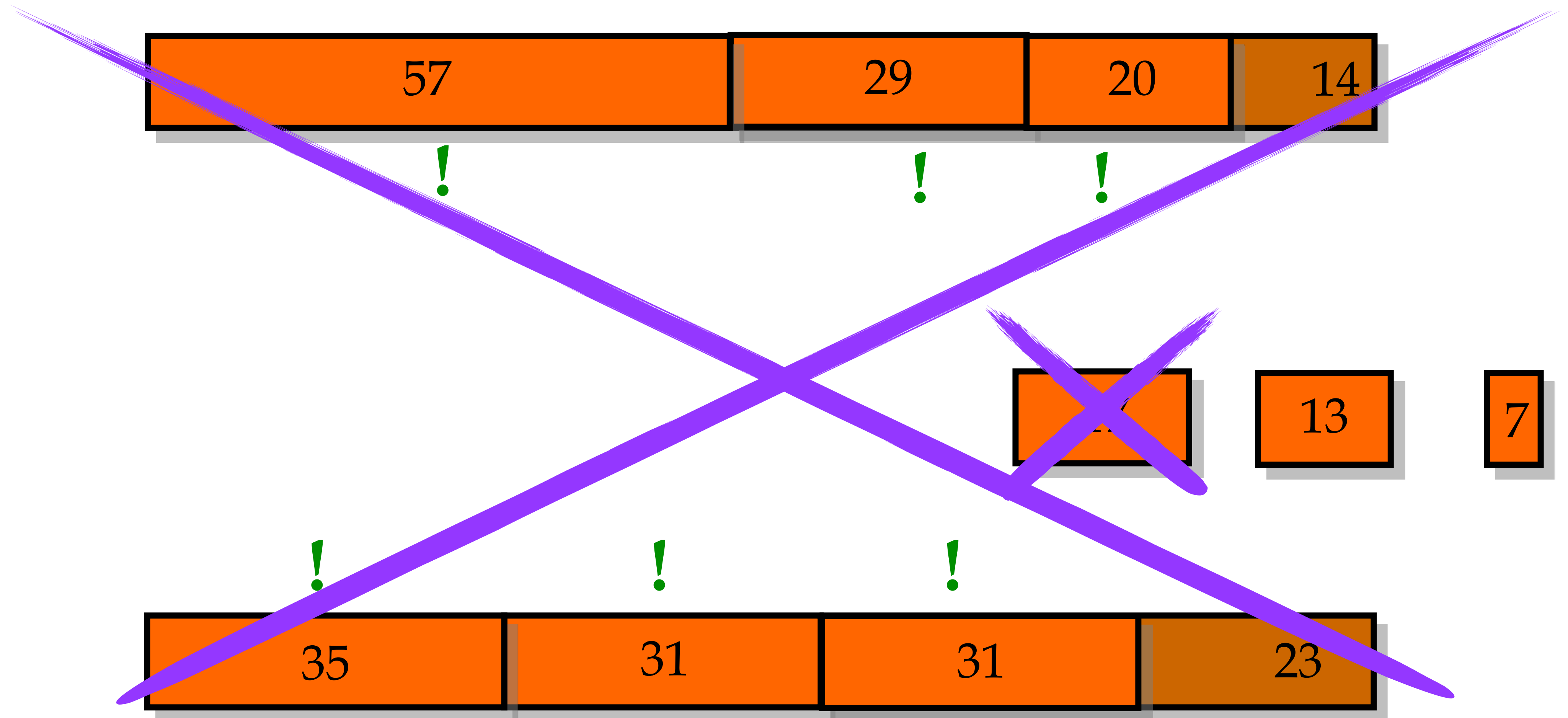
Zur Erinnerung: Subset Sum



Zur Erinnerung: Subset Sum



Zur Erinnerung: Subset Sum



Dynamic Programming

Satz 2.6 *Algorithmus 2.5* berechnet besten Lösungswert für das Rucksackproblem in einer Laufzeit $O(nZ)$.

Algorithmus 2.5 (Dynamic Programming für Knapsack)

Eingabe: $z_1, \dots, z_n, Z, p_1, \dots, p_n$
Ausgabe: Funktion $P : \{1, \dots, Z\} \times \{1, \dots, n\} \rightarrow \mathbb{R}; (x, i) \mapsto P(x, i)$
mit $P(x, i) = \max \sum_{j=1}^i p_j y_j$ mit $\sum_{j=1}^i z_j y_j \leq x$, für $y_j \in \{0, 1\}$

1: **for** ($x = 0$) **to** Z **do**

2: $P(x, 0) := 0$

3: **for** ($i = 1$) **to** n **do**

4: **for** ($x = 0$) **to** $(z_i - 1)$ **do**

5: $P(x, i) := P(x, i - 1)$

6: **for** ($x = z_i$) **to** Z **do**

7: **if** ($(P(x - z_i, i - 1) + p_i) > P(x, i - 1)$) **then**

8: $P(x, i) := P(x - z_i, i - 1) + p_i$

9: **else**

0: $P(x, i) := P(x, i - 1)$

$O(nZ)$ {

$x \backslash i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Induktion!

1. Objekt i ist zu teuer für Kostenschranke x , falls $z_i > x$

2. Objekt i ist nicht zu teuer, bringt aber keine Verbesserung

3. Objekt i ist nicht zu teuer und bringt eine Verbesserung



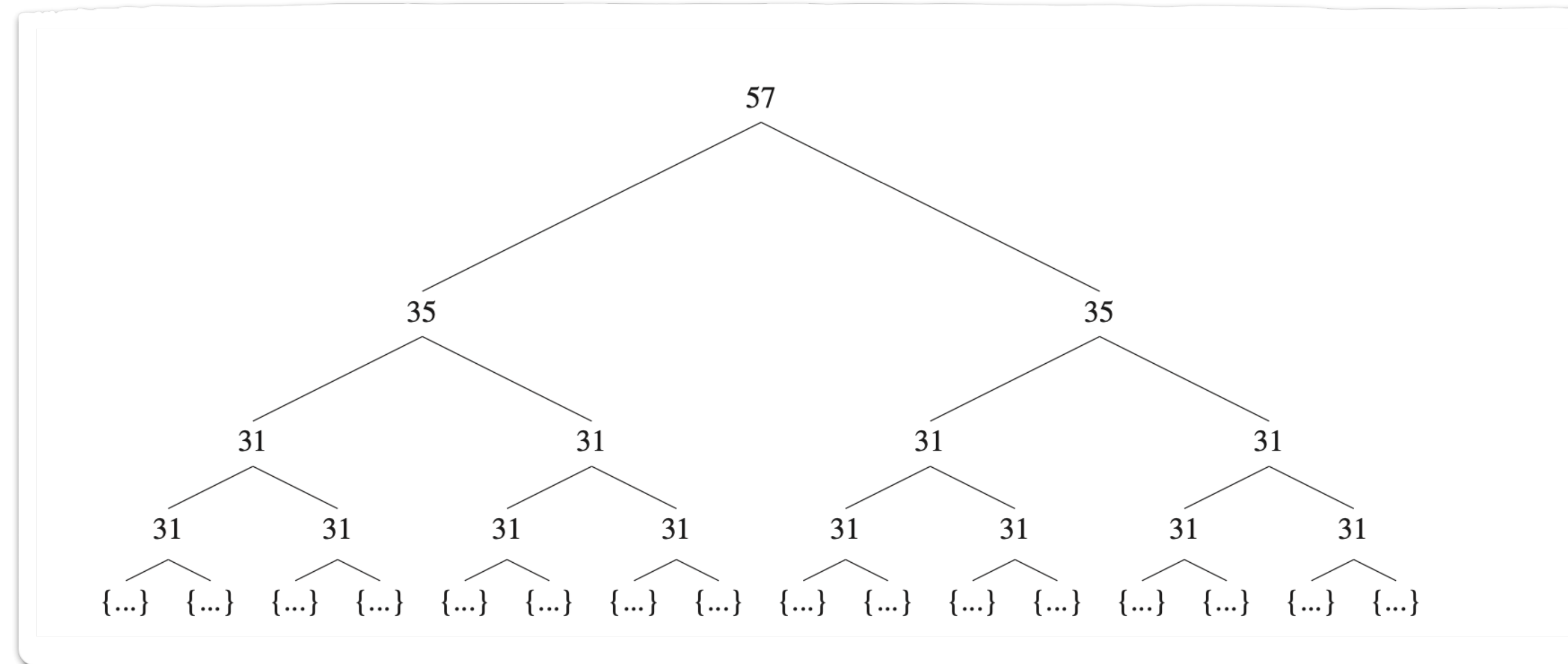
Technische
Universität
Braunschweig

Sándor Fekete | Dynamic Programming | AuD2 2024

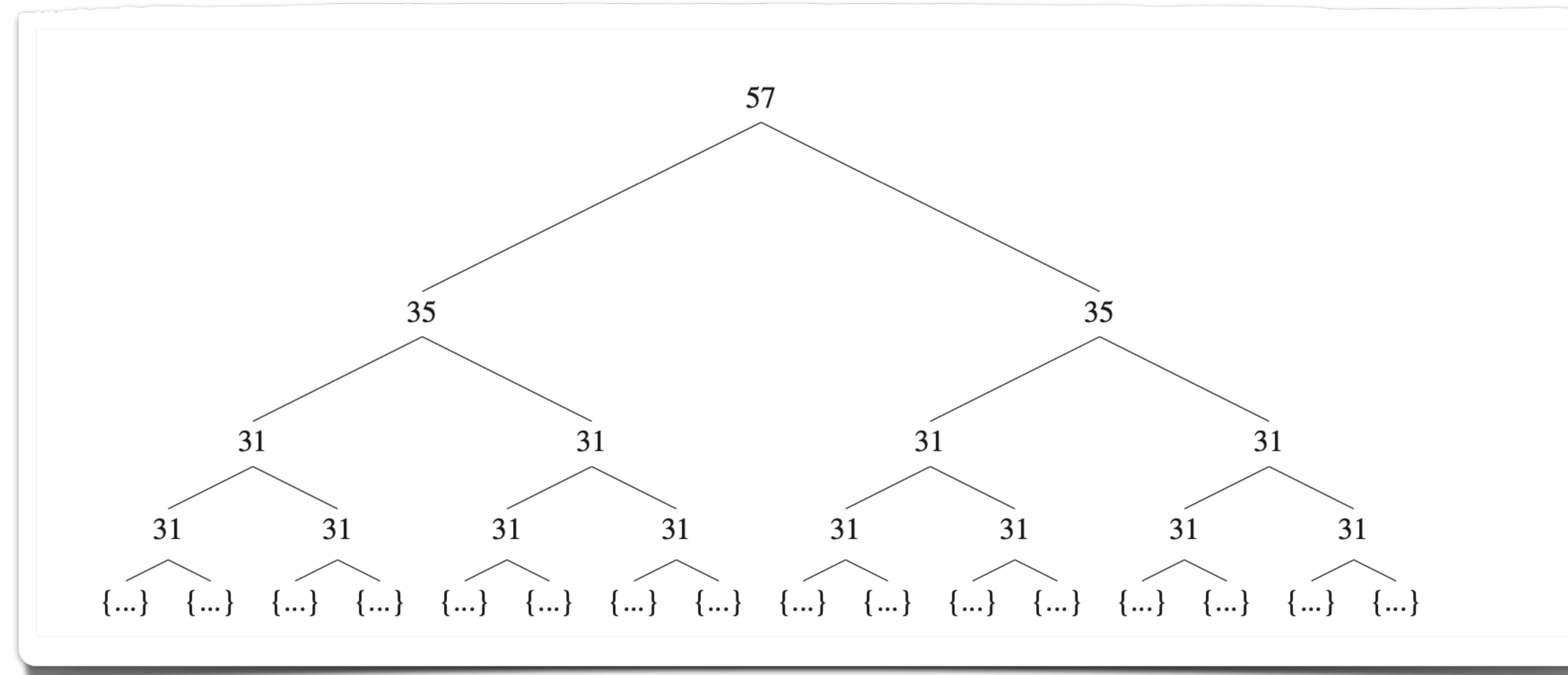
13

Branch-and-Bound

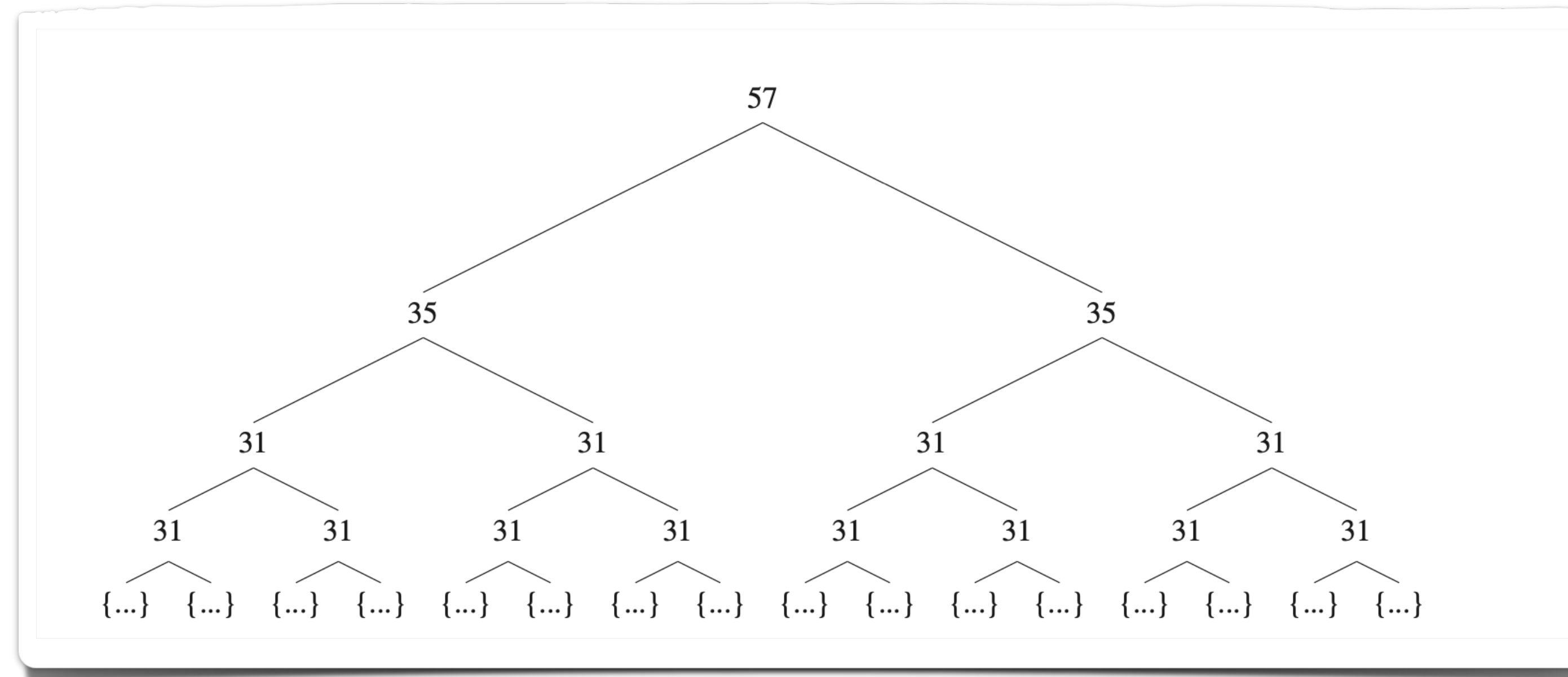
Branch-and-Bound



Branch-and-Bound

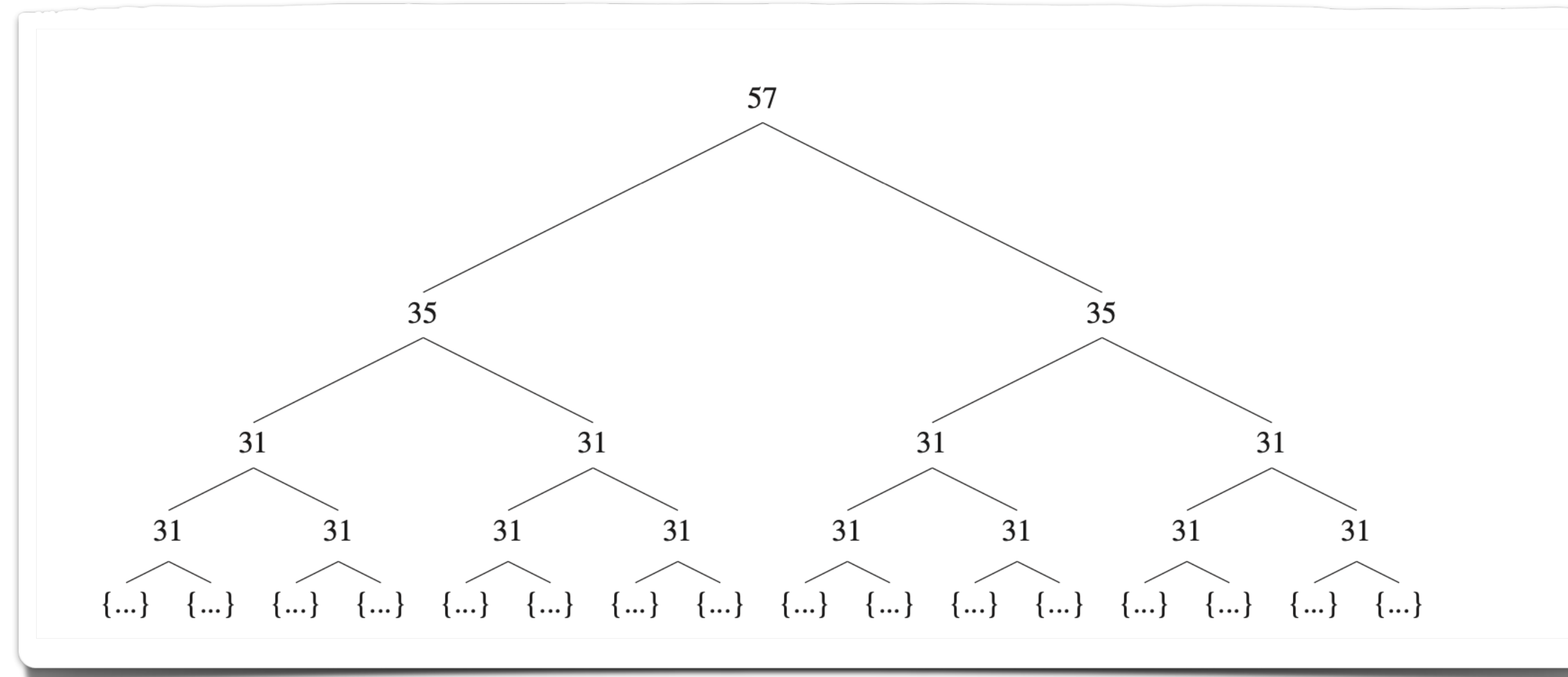


Branch-and-Bound



Branch and
Bound

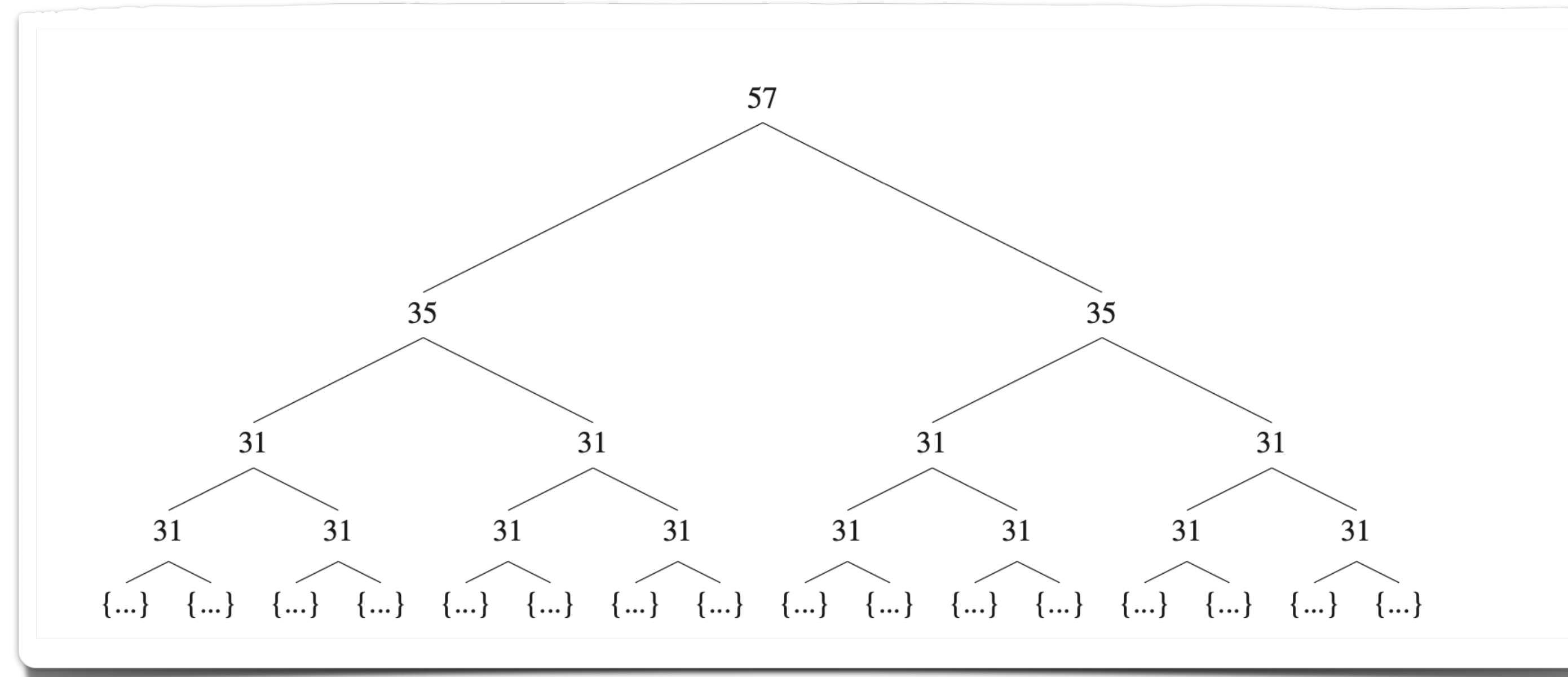
Branch-and-Bound



Branch and
Bound

- Beschneide Enumerationsbaum

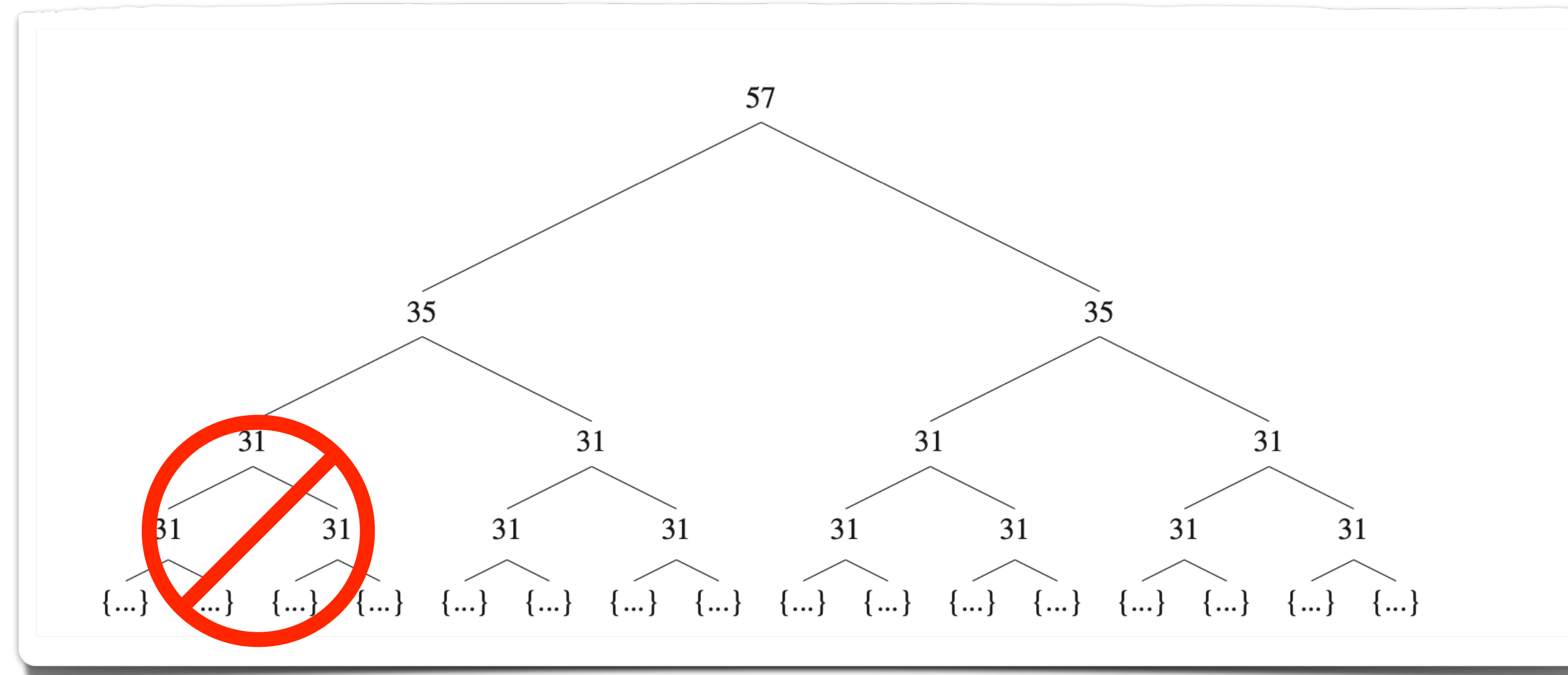
Branch-and-Bound



Branch and
Bound

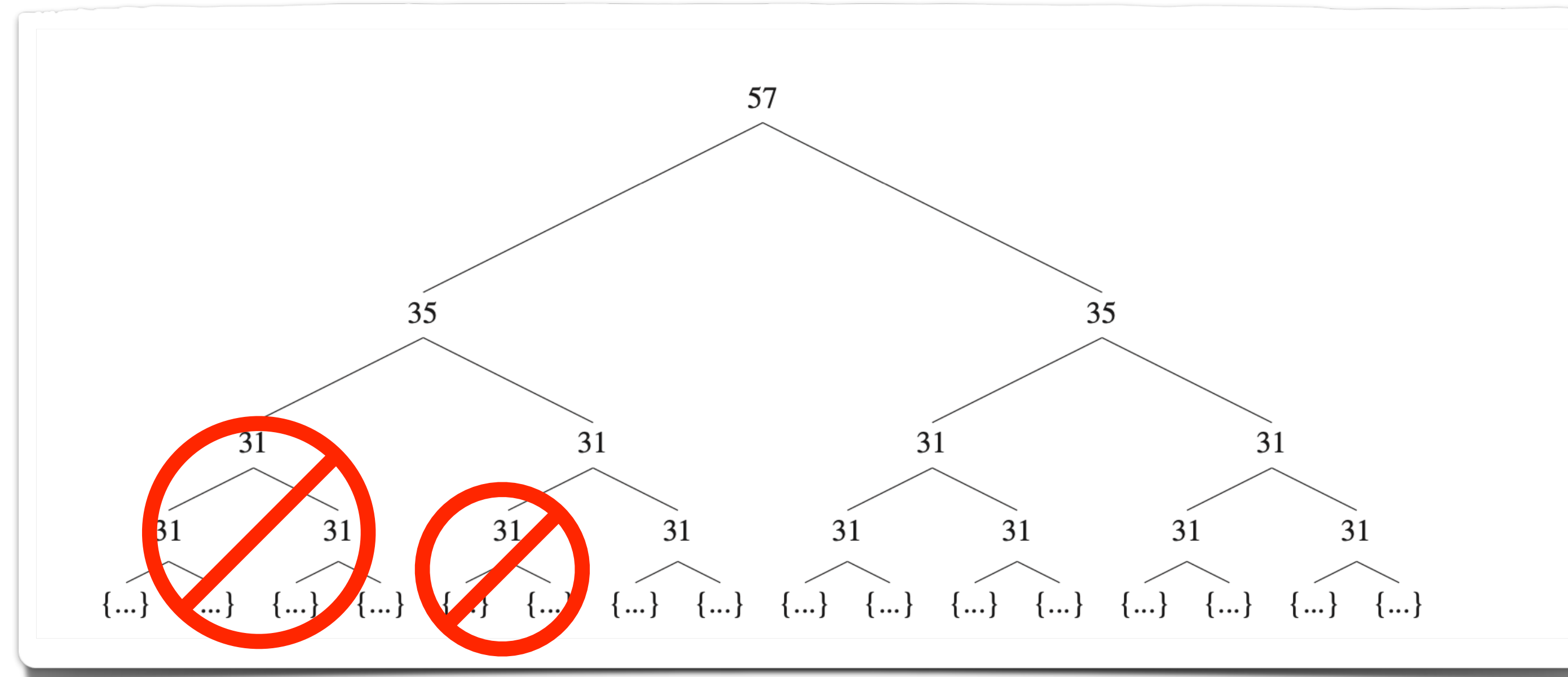
- Beschneide Enumerationsbaum
- Verfolge Mindest- und Maximalwerte

Branch-and-Bound



- Beschneide Enumerationsbaum
- Verfolge Mindest- und Maximalwerte

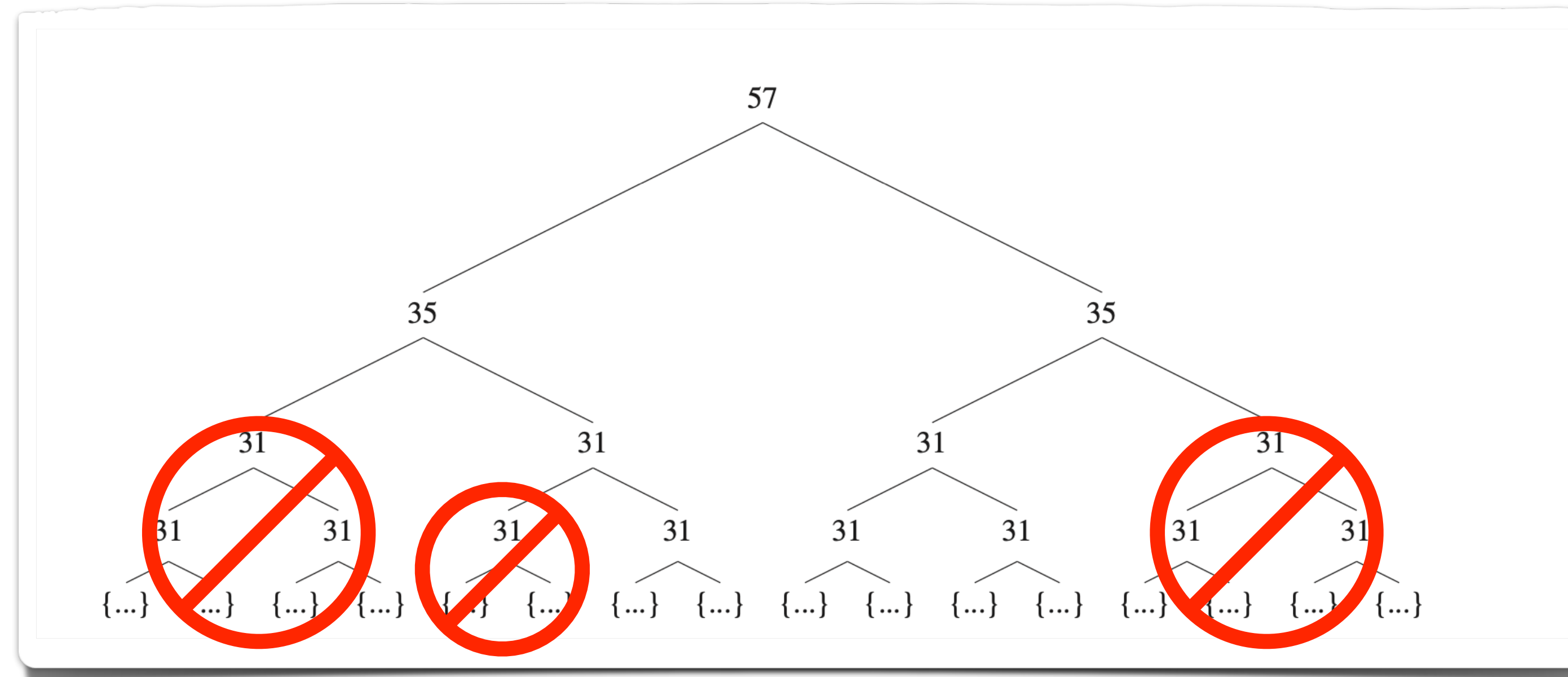
Branch-and-Bound



Branch and
Bound

- Beschneide Enumerationsbaum
- Verfolge Mindest- und Maximalwerte

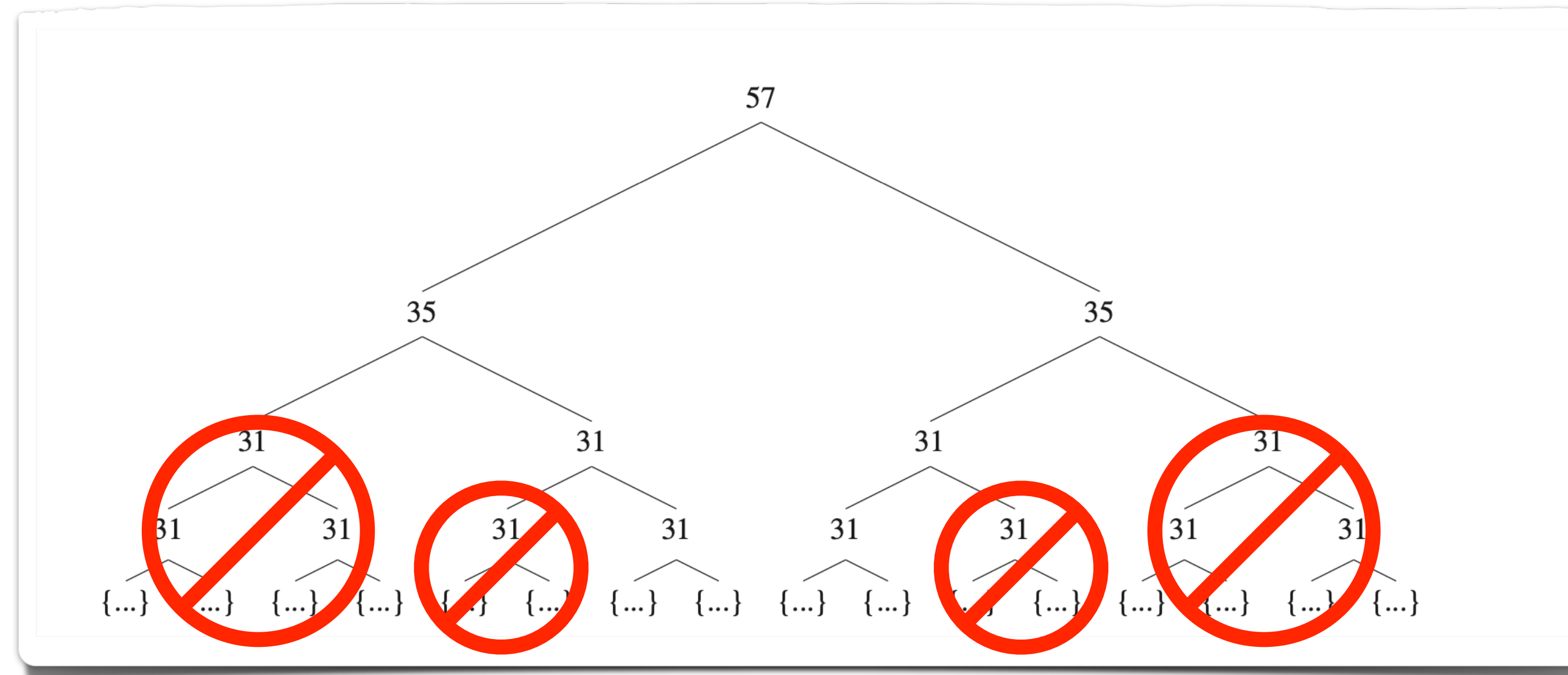
Branch-and-Bound



Branch and
Bound

- Beschneide Enumerationsbaum
- Verfolge Mindest- und Maximalwerte

Branch-and-Bound



- Beschneide Enumerationsbaum
- Verfolge Mindest- und Maximalwerte

Historie

Historie

ECONOMETRICA

VOLUME 28

July, 1960

NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

By A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3-7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand² from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

¹ Or more generally, any value within a bounded interval.

² We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

Historie

1960

ECONOMETRICA

VOLUME 28

July, 1960

NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

By A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3-7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand² from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

¹ Or more generally, any value within a bounded interval.

² We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

Historie

1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule $3 - 7/10$ flights between two cities, or to undertake only $1/4$ of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

Historie

1960

ECONOMETRICA

VOLUME 28

July, 1960

NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

By A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3-7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand² from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

¹ Or more generally, any value within a bounded interval.

² We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

Historie



1960

ECONOMETRICA

VOLUME 28

July, 1960

NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

By A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3-7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand² from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

¹ Or more generally, any value within a bounded interval.

² We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

Historie



Ailsa H. Land

1960

ECONOMETRICA

VOLUME 28

July, 1960

NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

By A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3-7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand² from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

¹ Or more generally, any value within a bounded interval.

² We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

Historie



Ailsa H. Land



1960

ECONOMETRICA

VOLUME 28

July, 1960

NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

By A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3-7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand² from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

¹ Or more generally, any value within a bounded interval.

² We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

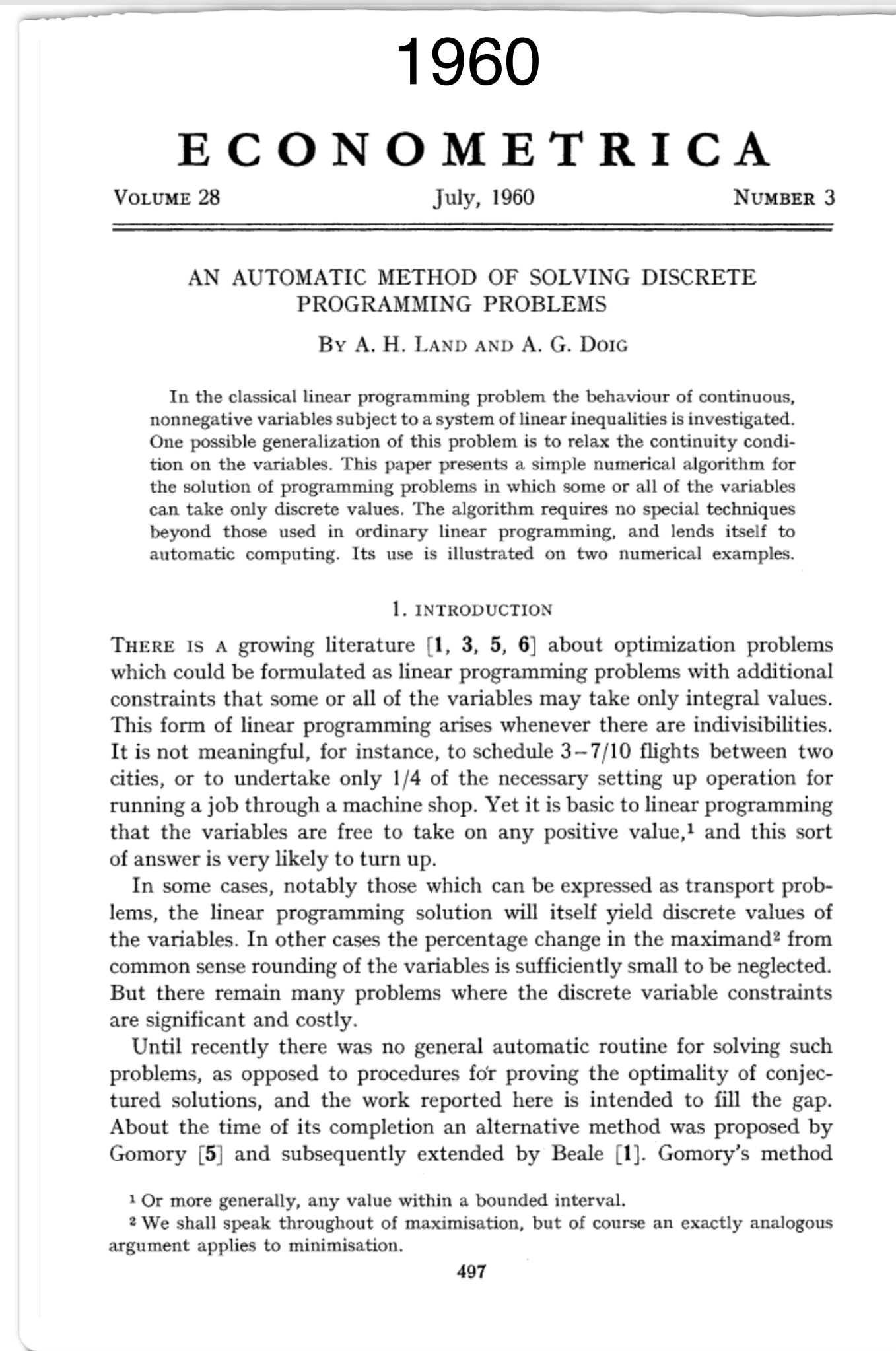
Historie



Ailsa H. Land



Alison Doig



Historie



Ailsa H. Land



Alison Doig

1960 ECONOMETRICA

VOLUME 28

July, 1960

NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

By A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3-7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand² from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

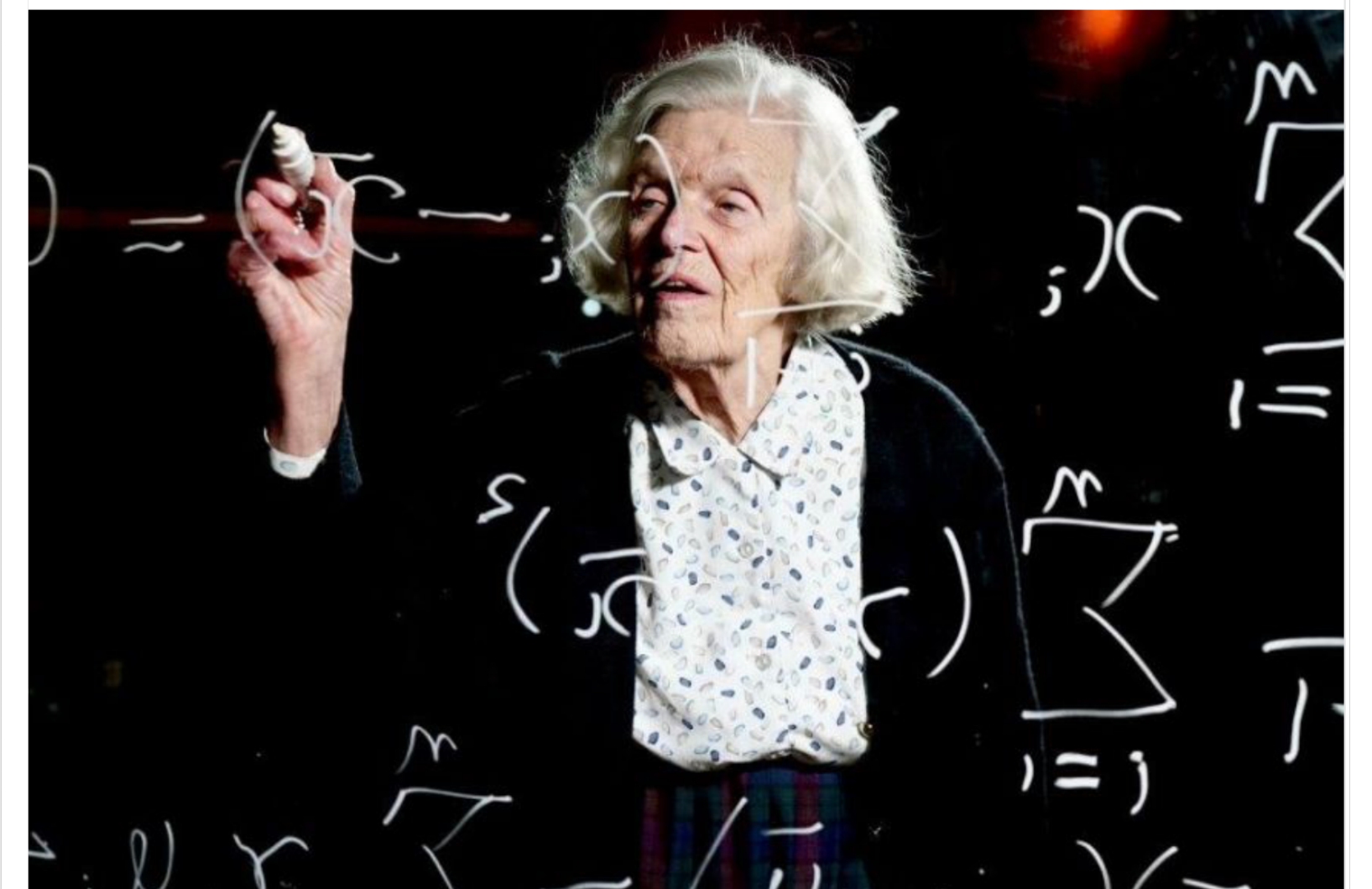
¹ Or more generally, any value within a bounded interval.

² We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

'I've always loved numbers': Meet the ground-breaking grandmother of Australian mathematics

7.30 / By [Lauren Day](#)

Posted 8 Oct 2018, updated 10 Oct 2018



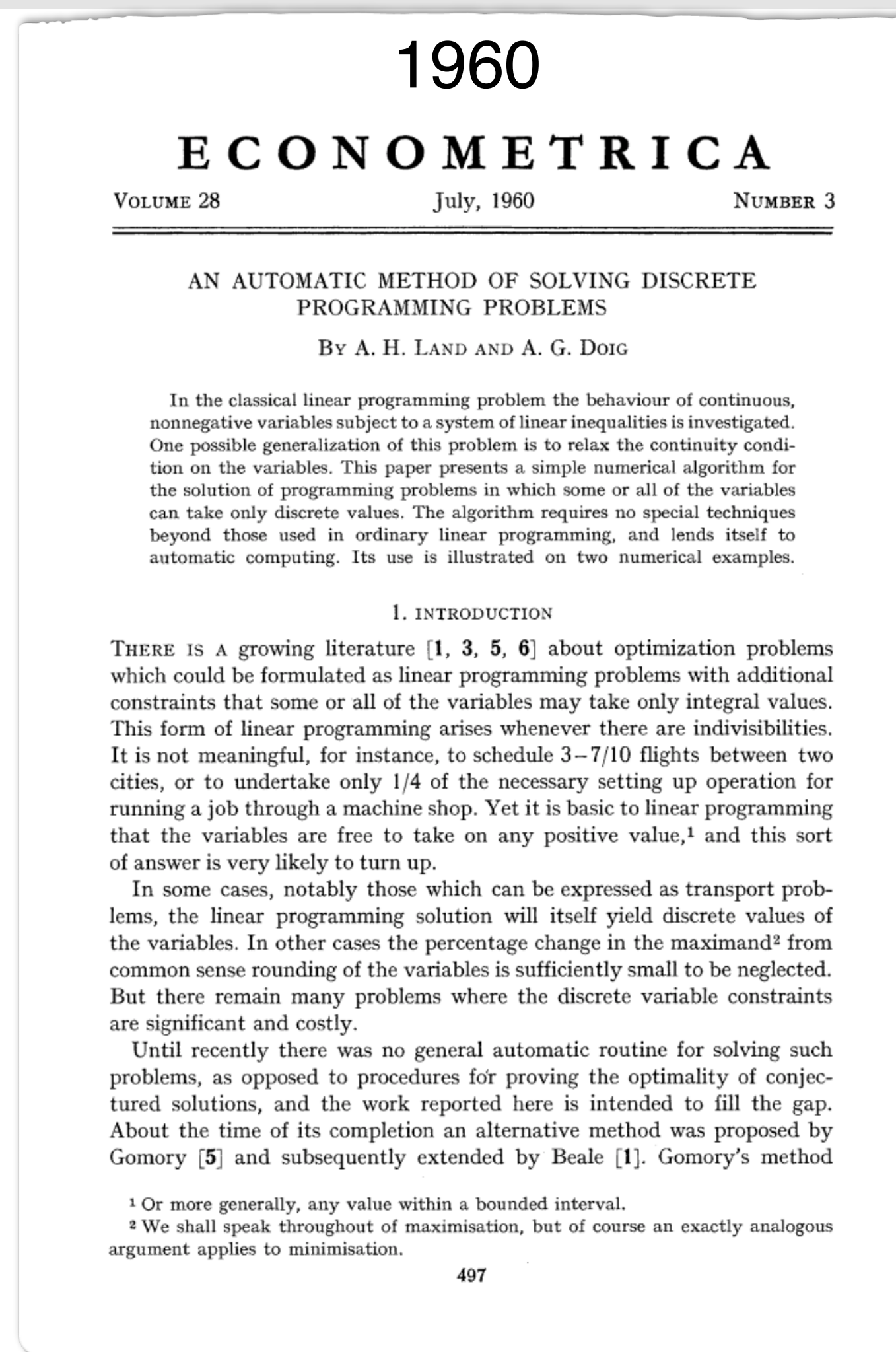
Historie



Ailsa H. Land



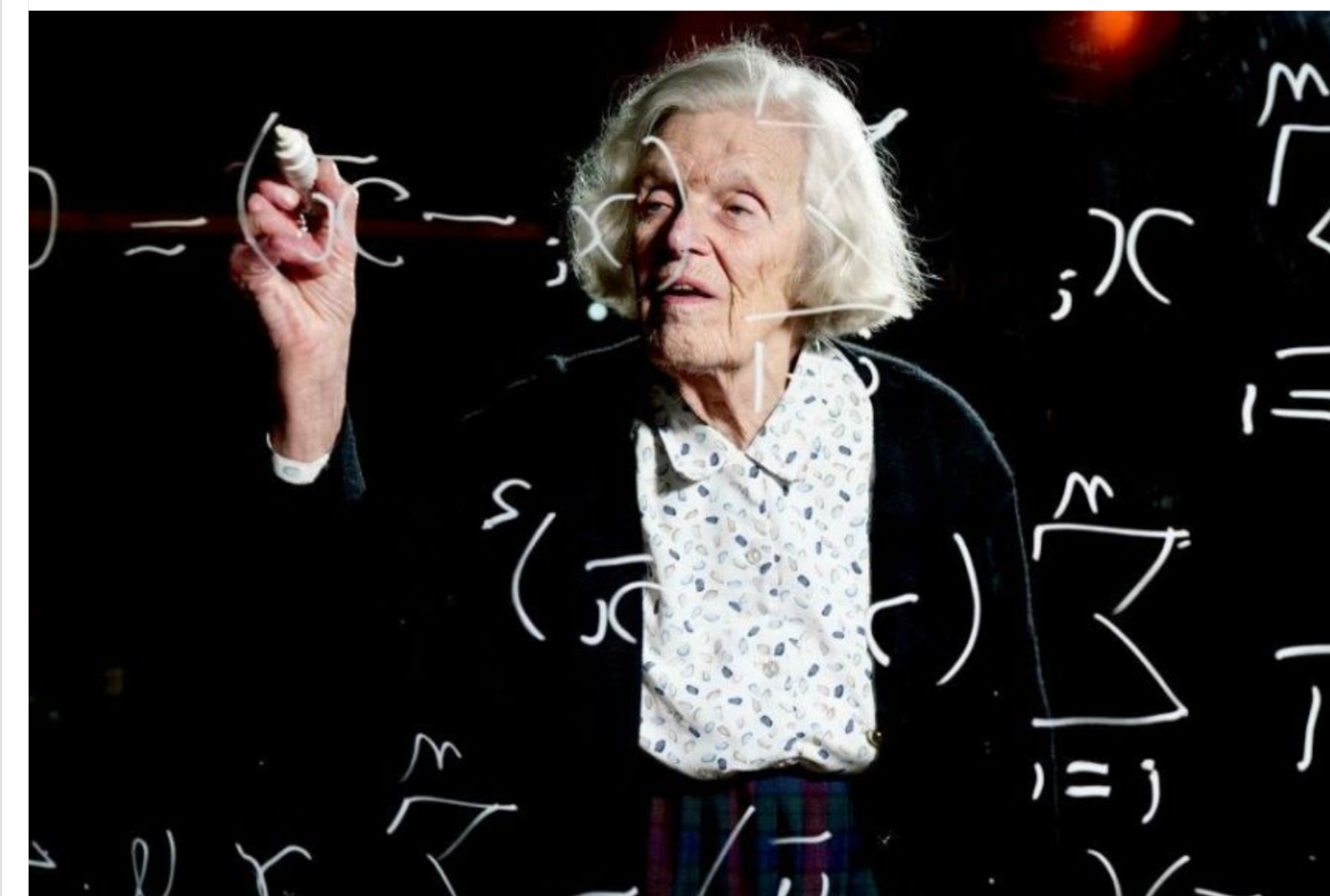
Alison Doig



'I've always loved numbers': Meet the ground-breaking grandmother of Australian mathematics

7:30 / By [Lauren Day](#)

Posted 8 Oct 2018, updated 10 Oct 2018



Alison Grant Harcourt [AO](#)

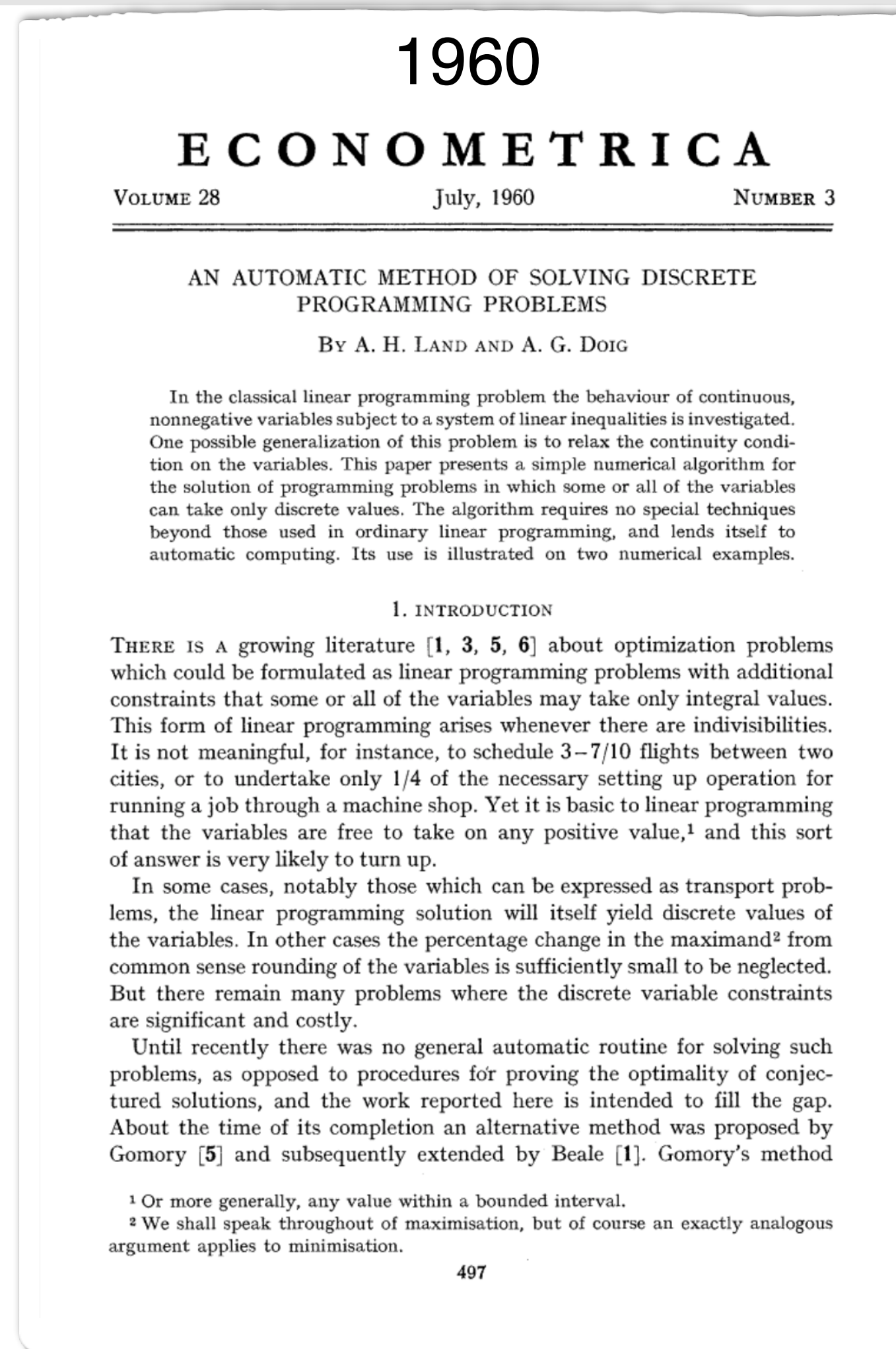
Historie



Ailsa H. Land



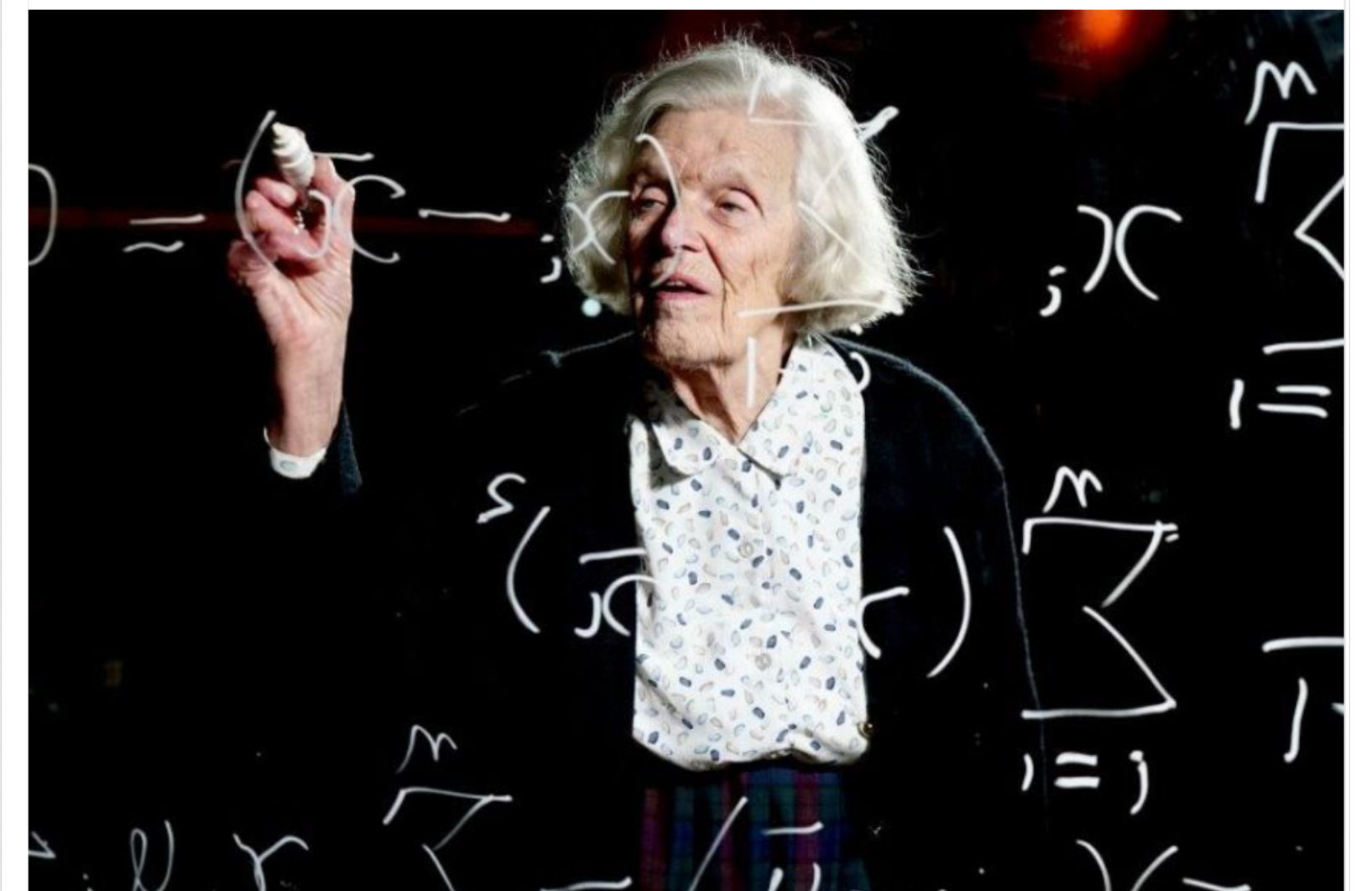
Alison Doig



'I've always loved numbers': Meet the ground-breaking grandmother of Australian mathematics

7:30 / By [Lauren Day](#)

Posted 8 Oct 2018, updated 10 Oct 2018



Alison Grant Harcourt AO

In June 2019, Harcourt was made an [Officer of the Order of Australia](#)

3.2 Methodik

Ideen

Ideen

Grundideen:

Ideen

Grundideen:

1. Enumeriere die möglichen Teilmengen in einem Enumerationsbaum

Ideen

Grundideen:

1. Enumeriere die möglichen Teilmengen in einem Enumerationsbaum
2. Behalte den Zielfunktionswert im Auge:

Ideen

Grundideen:

1. Enumeriere die möglichen Teilmengen in einem Enumerationsbaum
2. Behalte den Zielfunktionswert im Auge:
 - Untere Schranke: Erreichter Zielfunktionswert im ganzen Baum

Ideen

Grundideen:

1. Enumeriere die möglichen Teilmengen in einem Enumerationsbaum
2. Behalte den Zielfunktionswert im Auge:
 - Untere Schranke: Erreichter Zielfunktionswert im ganzen Baum
 - Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Baum

Ideen

Grundideen:

1. Enumeriere die möglichen Teilmengen in einem Enumerationsbaum
2. Behalte den Zielfunktionswert im Auge:
 - Untere Schranke: Erreichter Zielfunktionswert im ganzen Baum
 - Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Baum
3. Beide Schranken sollten möglichst einfach und schnell zu berechnen sein

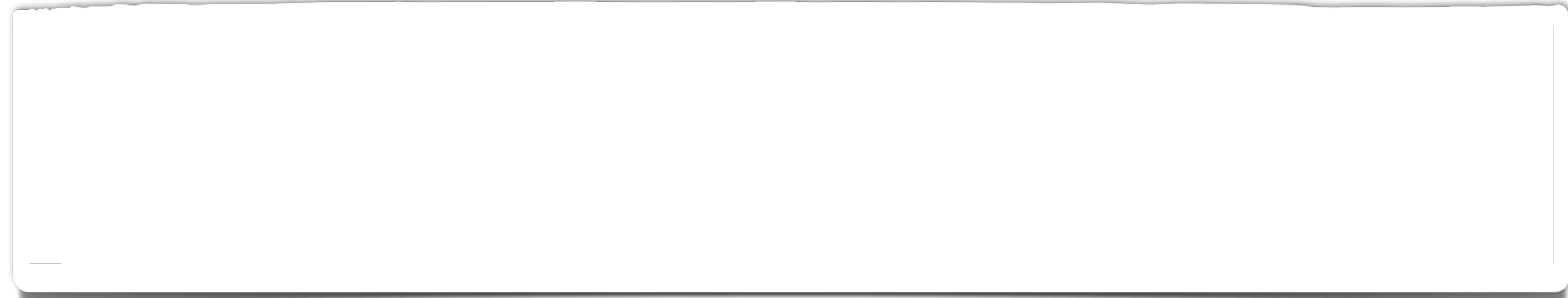
Ideen

Grundideen:

1. Enumeriere die möglichen Teilmengen in einem Enumerationsbaum
2. Behalte den Zielfunktionswert im Auge:
 - Untere Schranke: Erreichter Zielfunktionswert im ganzen Baum
 - Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Baum
3. Beide Schranken sollten möglichst einfach und schnell zu berechnen sein
4. Wenn der erreichbare Wert in einem Teilbaum kleiner bleiben muss als der im ganzen Baum bereits erreichte, können wir den aktuellen Teilbaum abschneiden

1. Enumerationsbaum

1. Enumerationsbaum

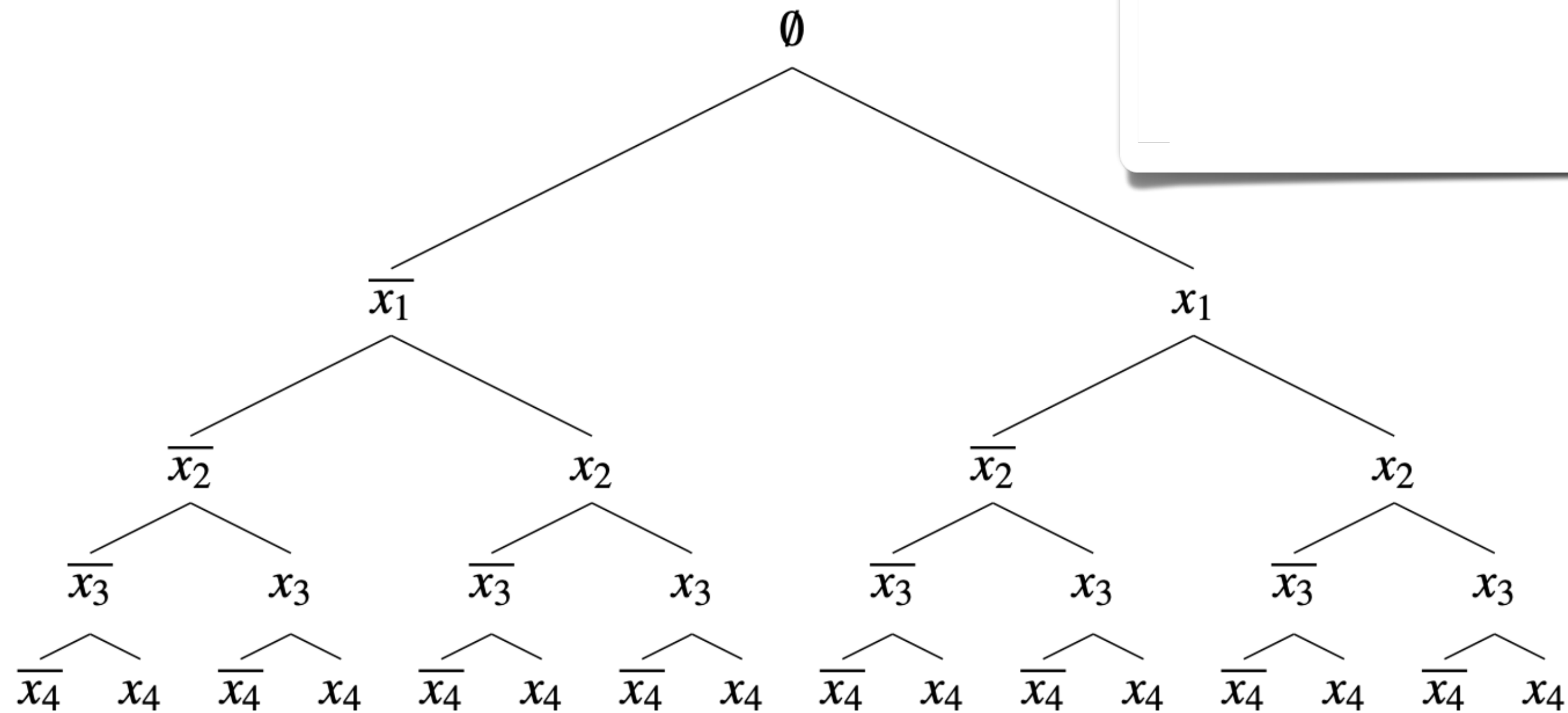


1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$

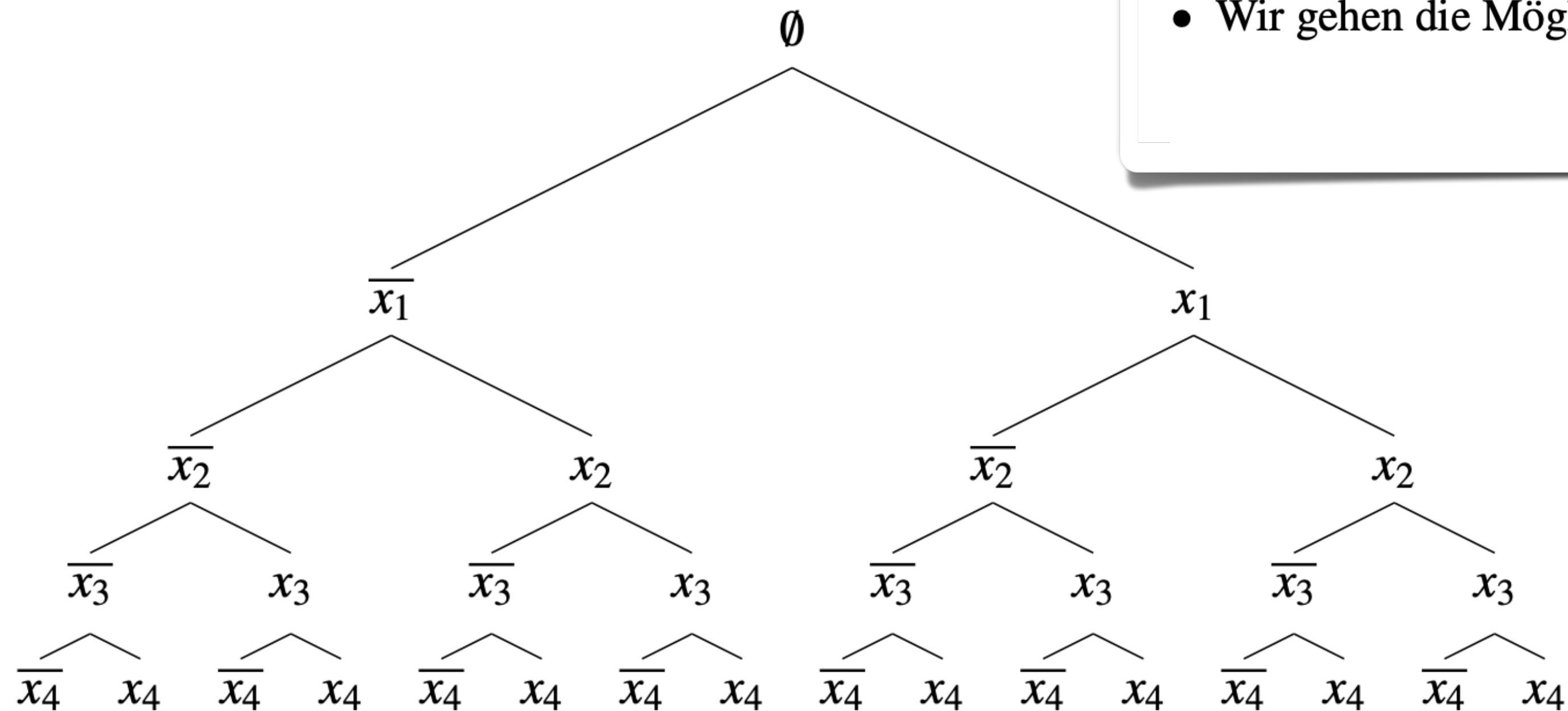
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$



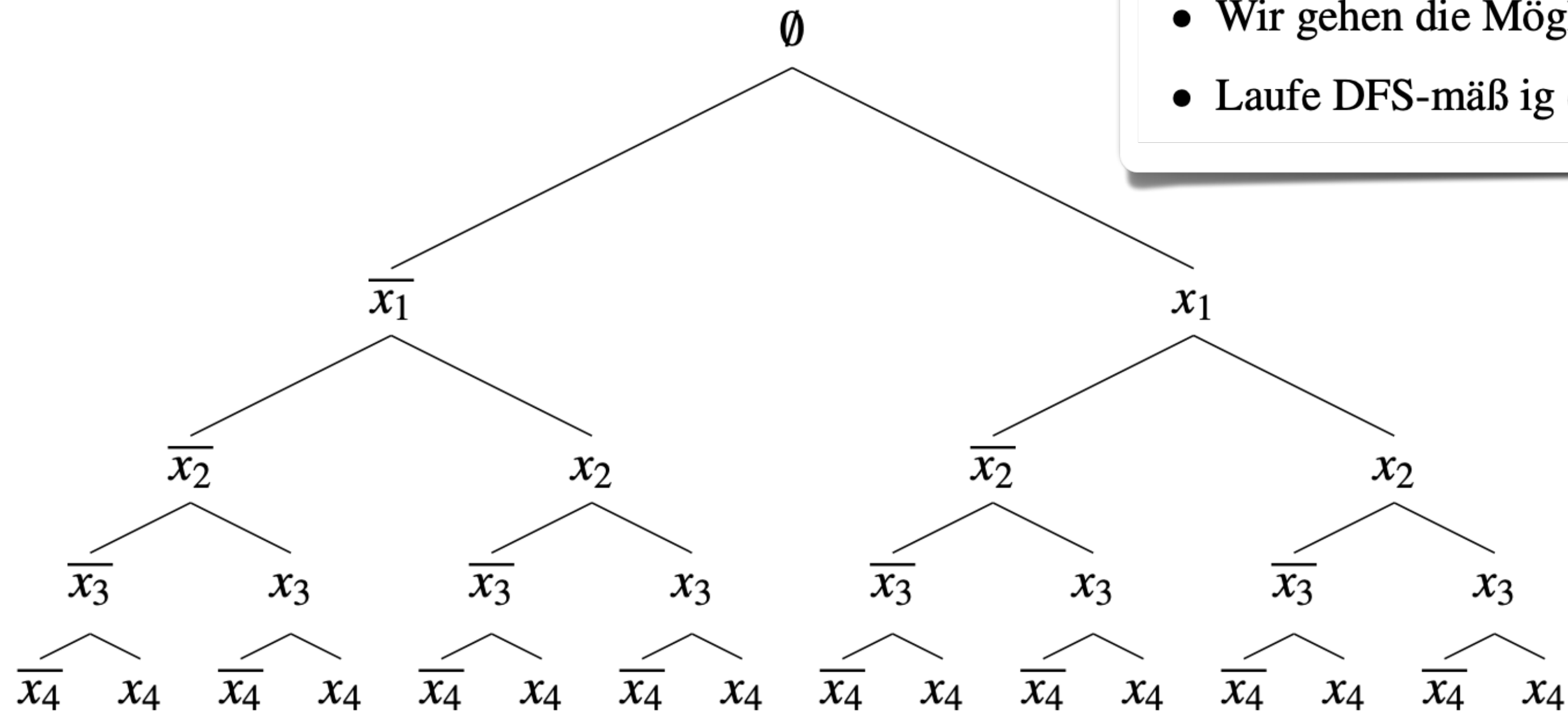
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:



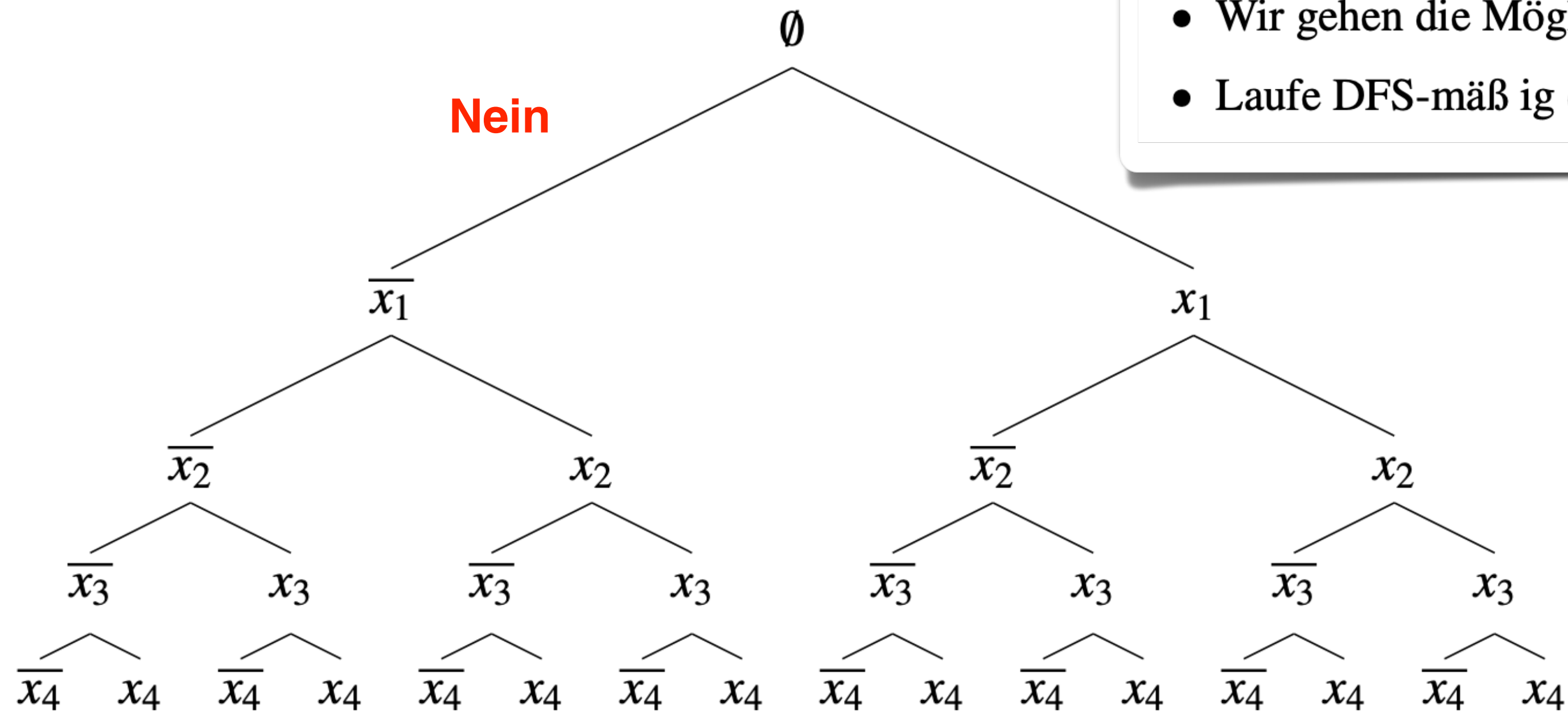
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



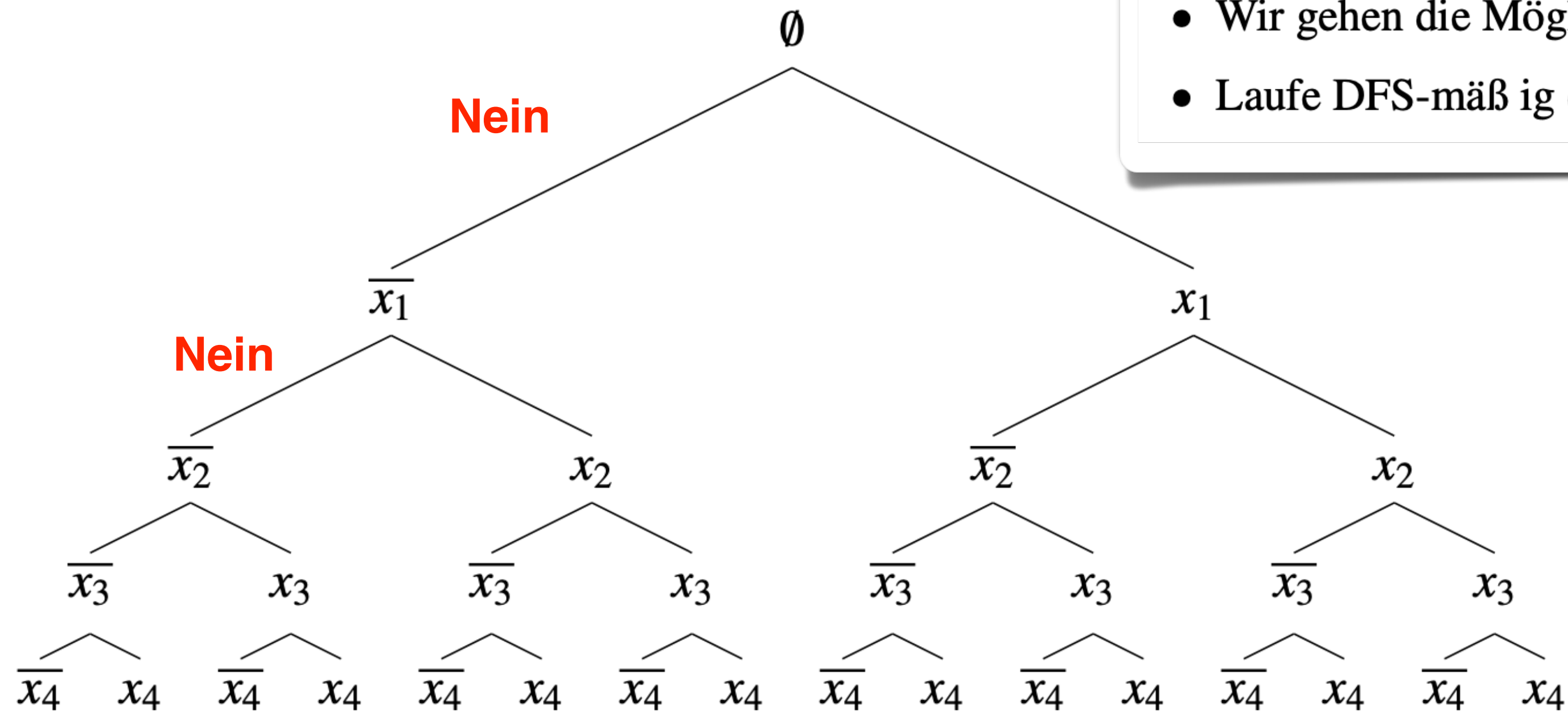
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



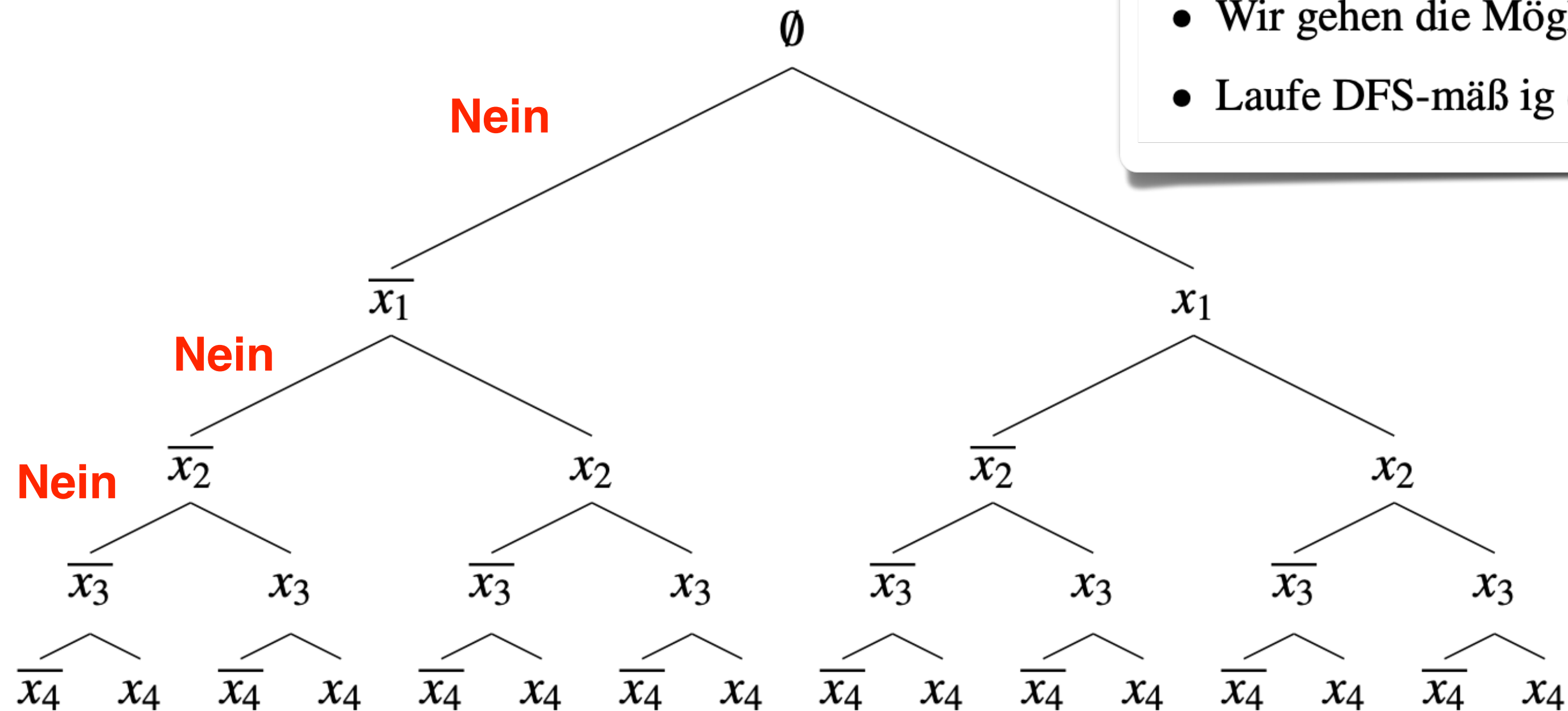
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



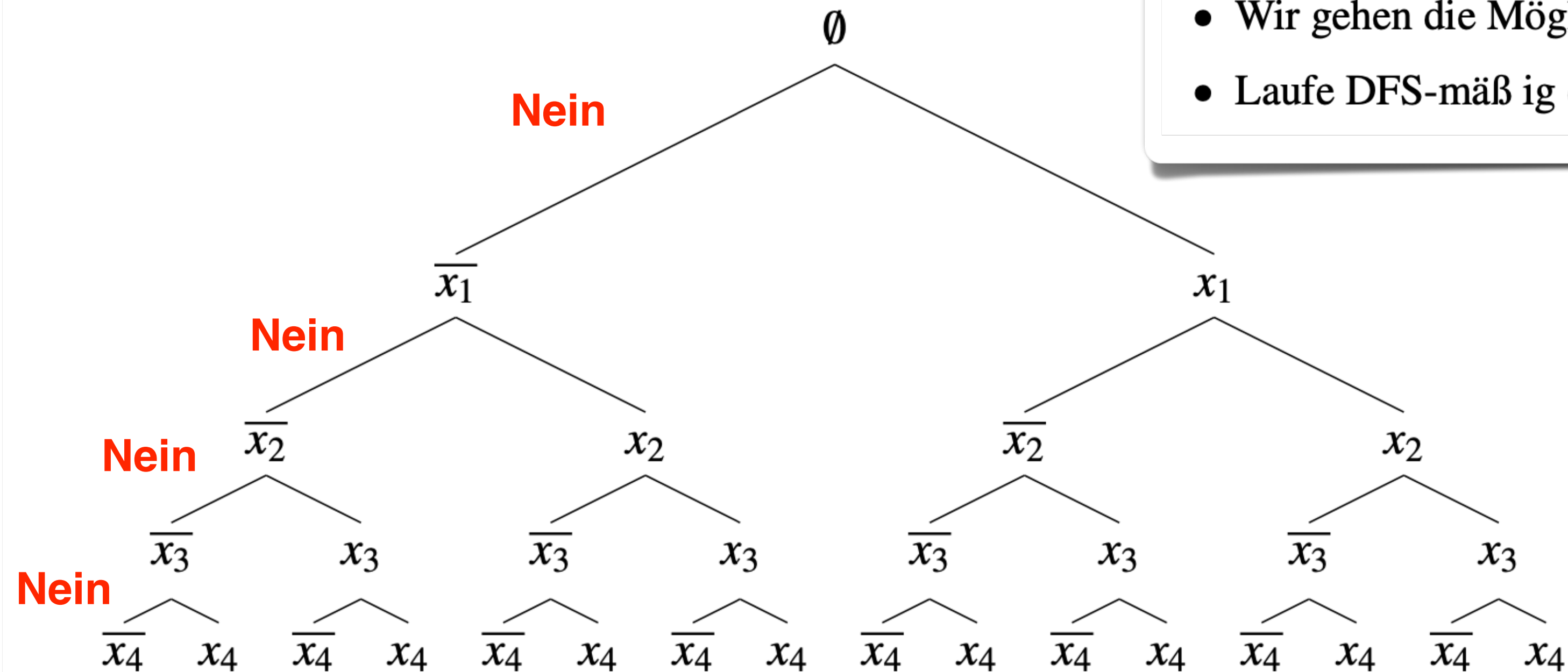
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



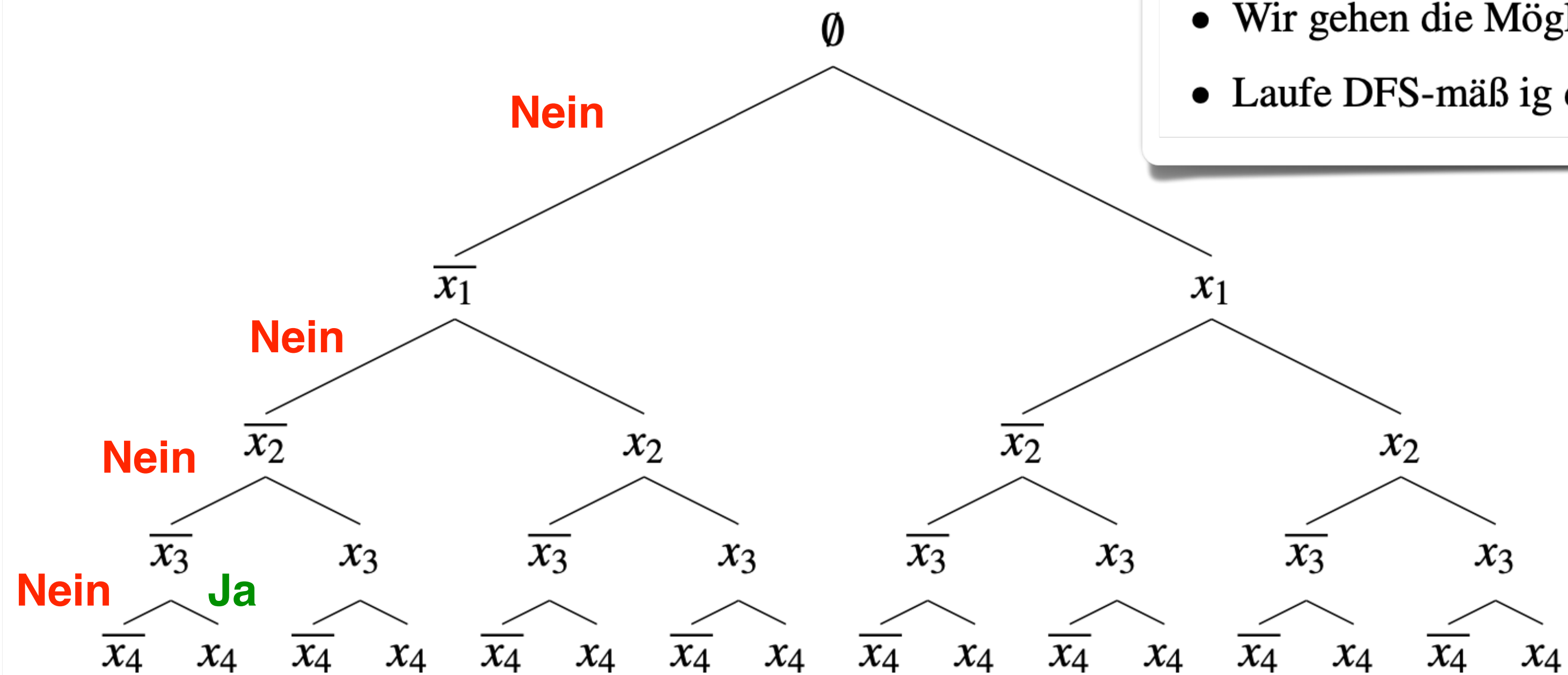
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



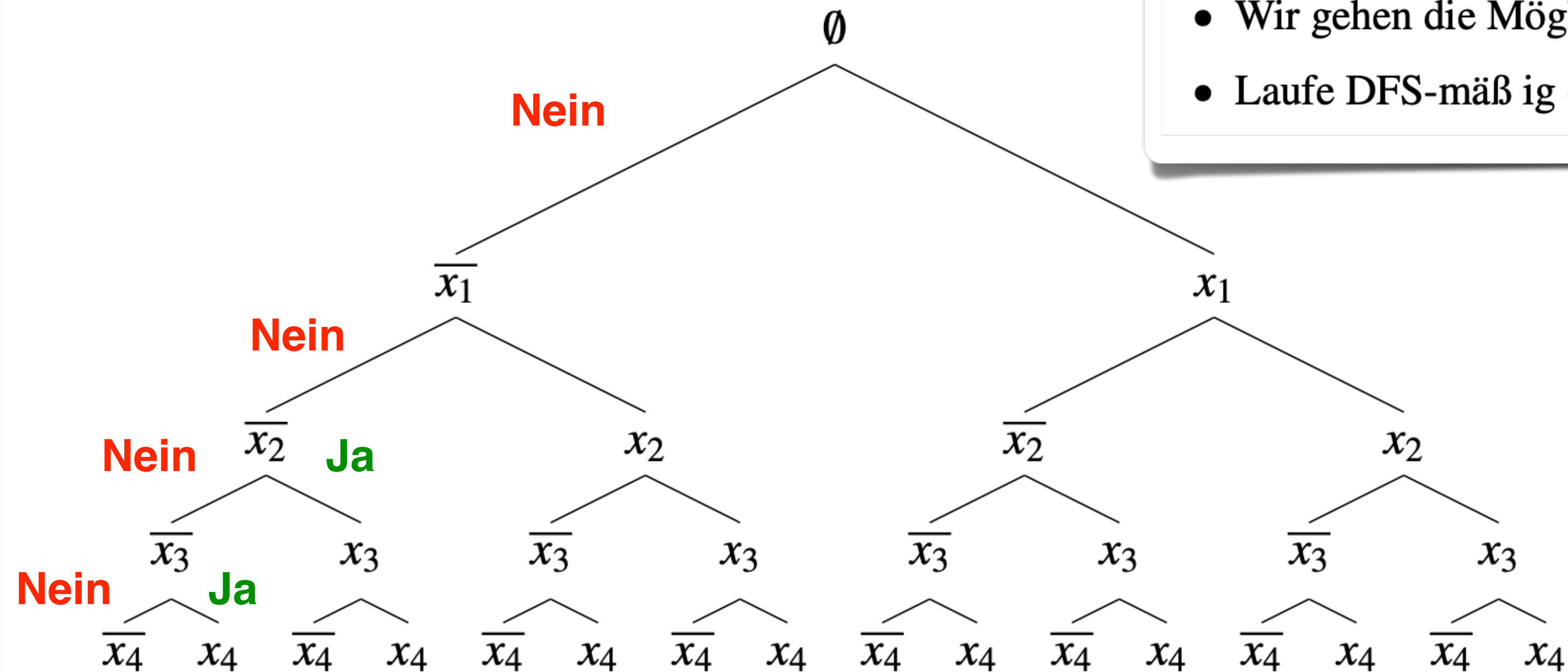
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



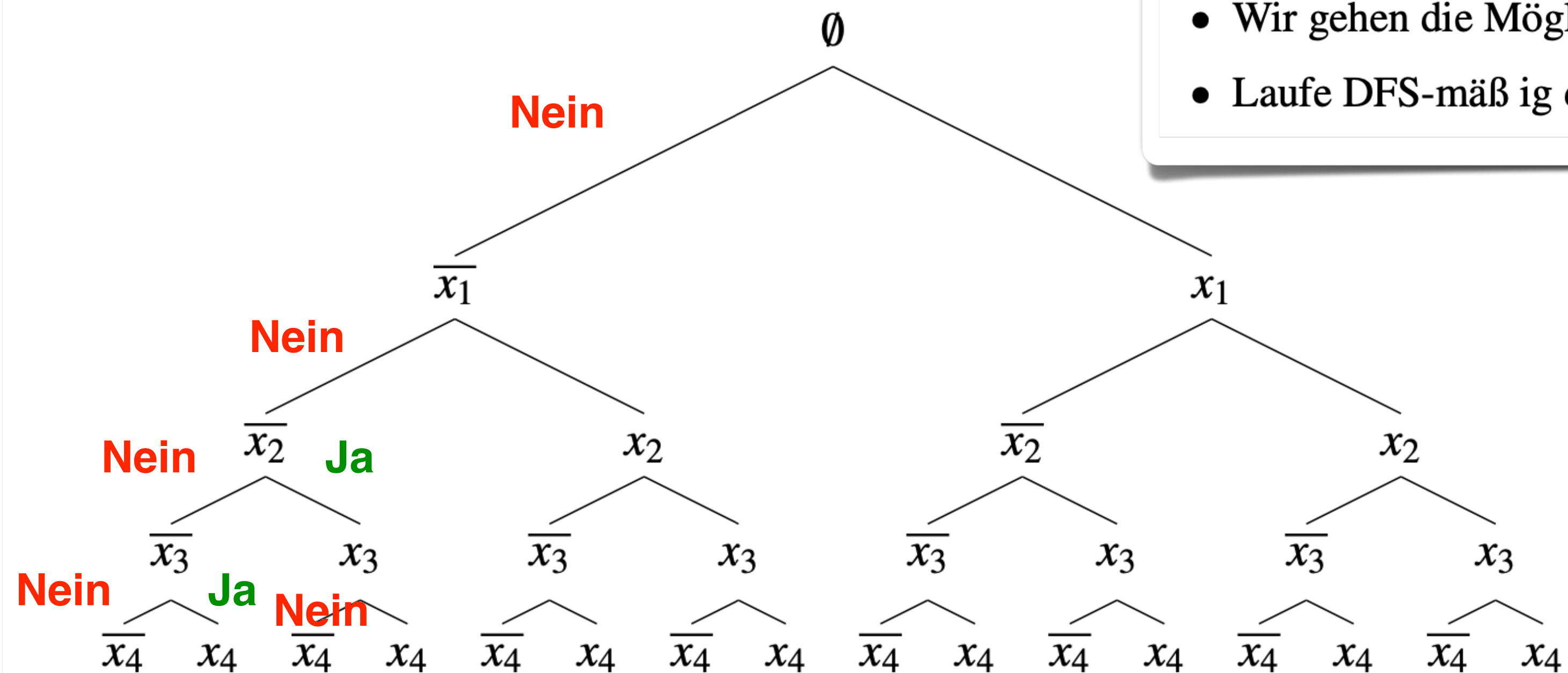
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



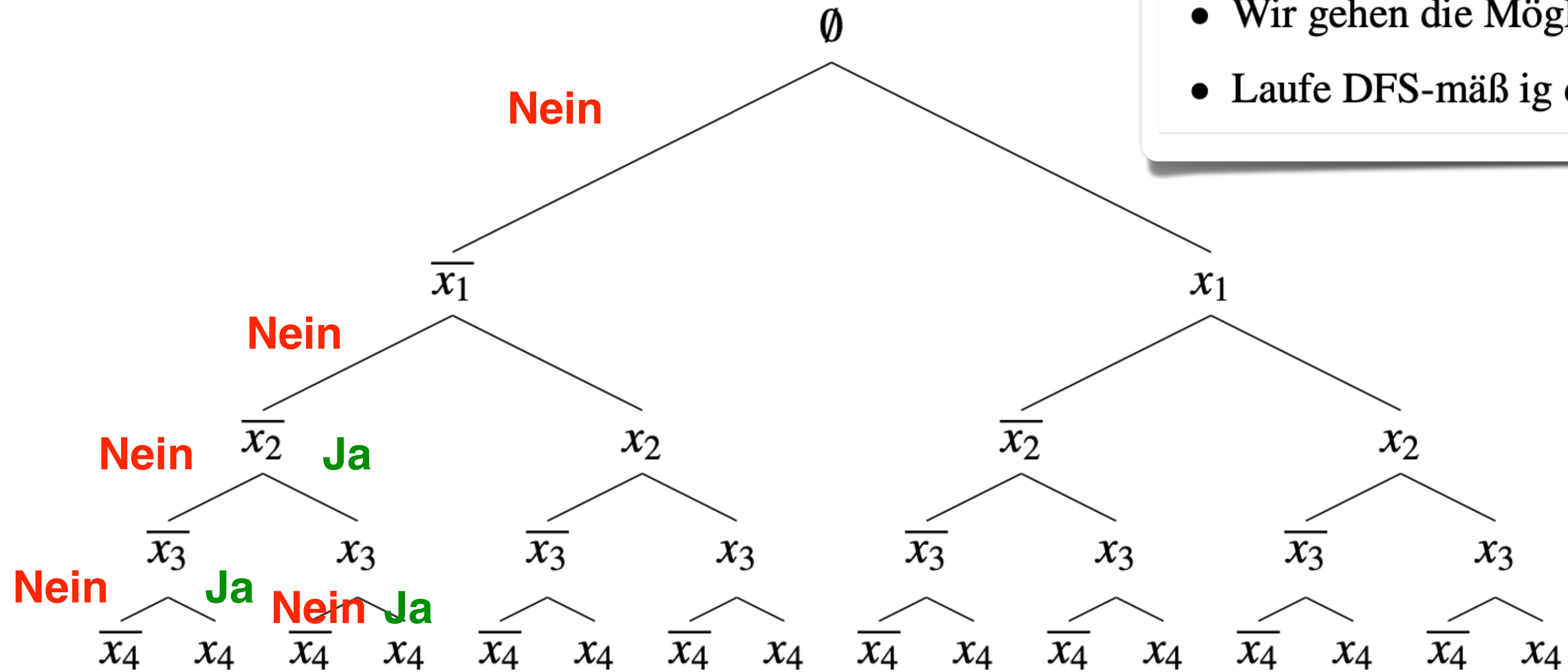
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



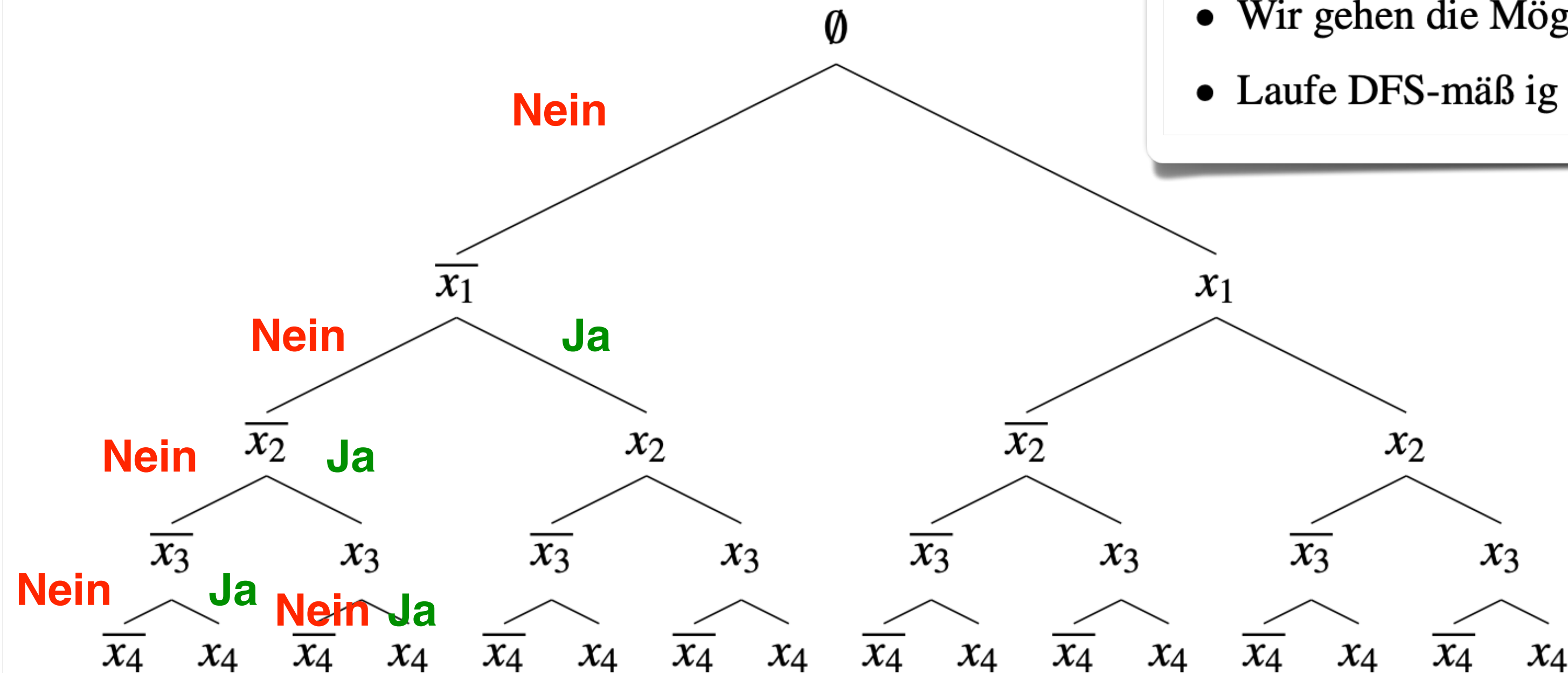
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



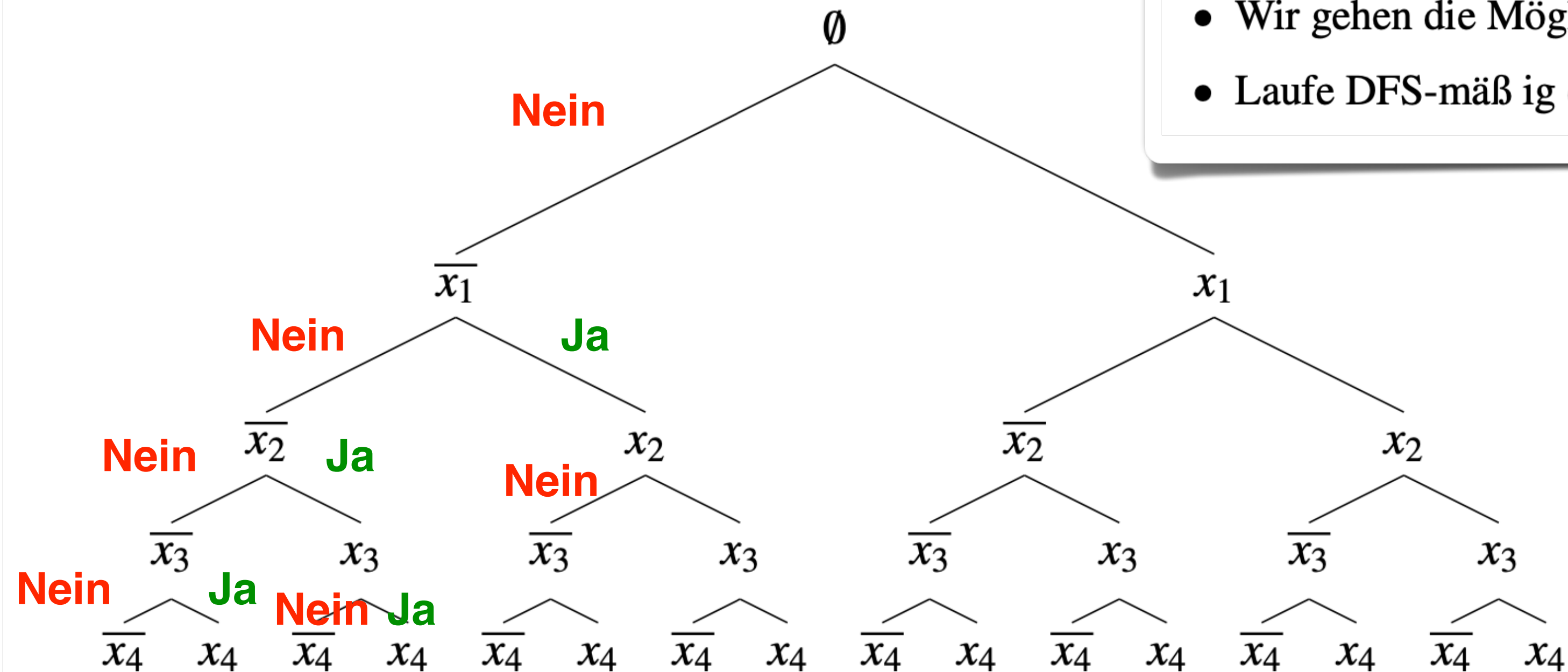
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



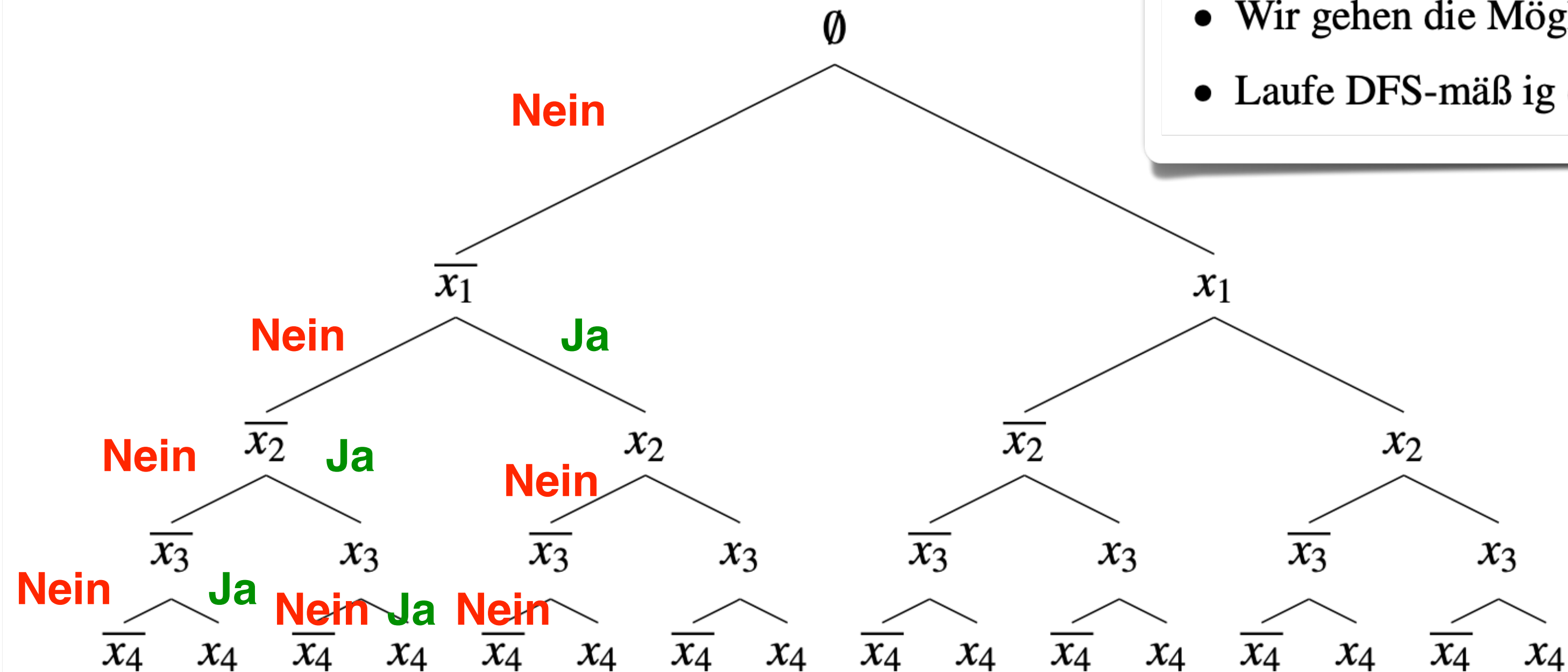
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



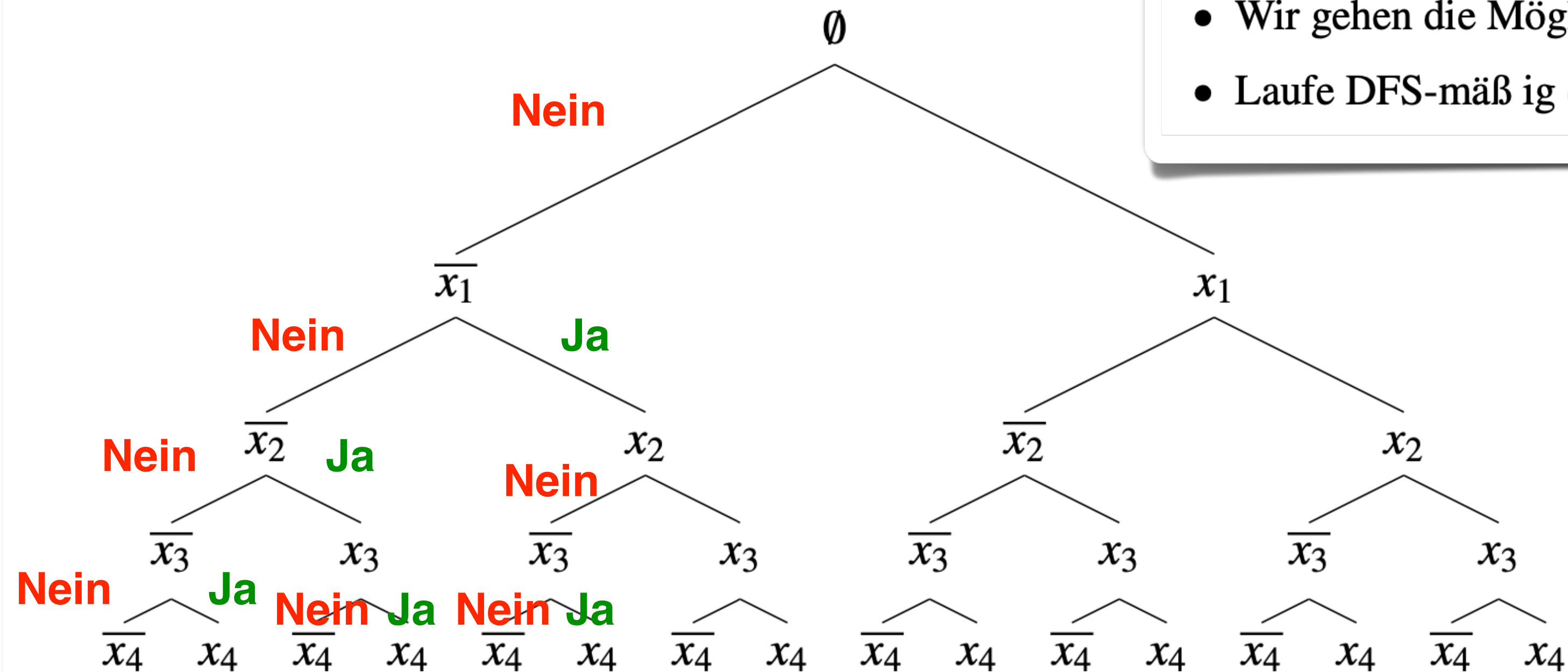
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



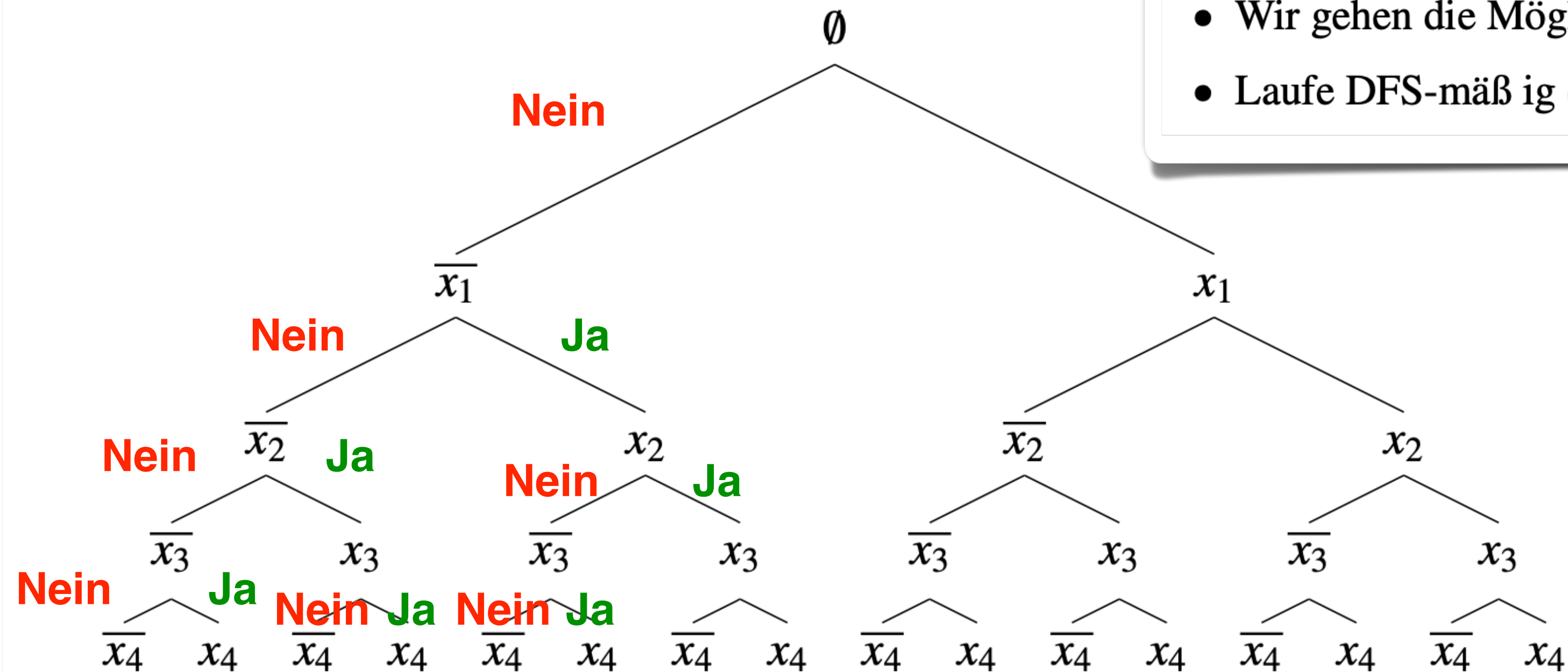
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



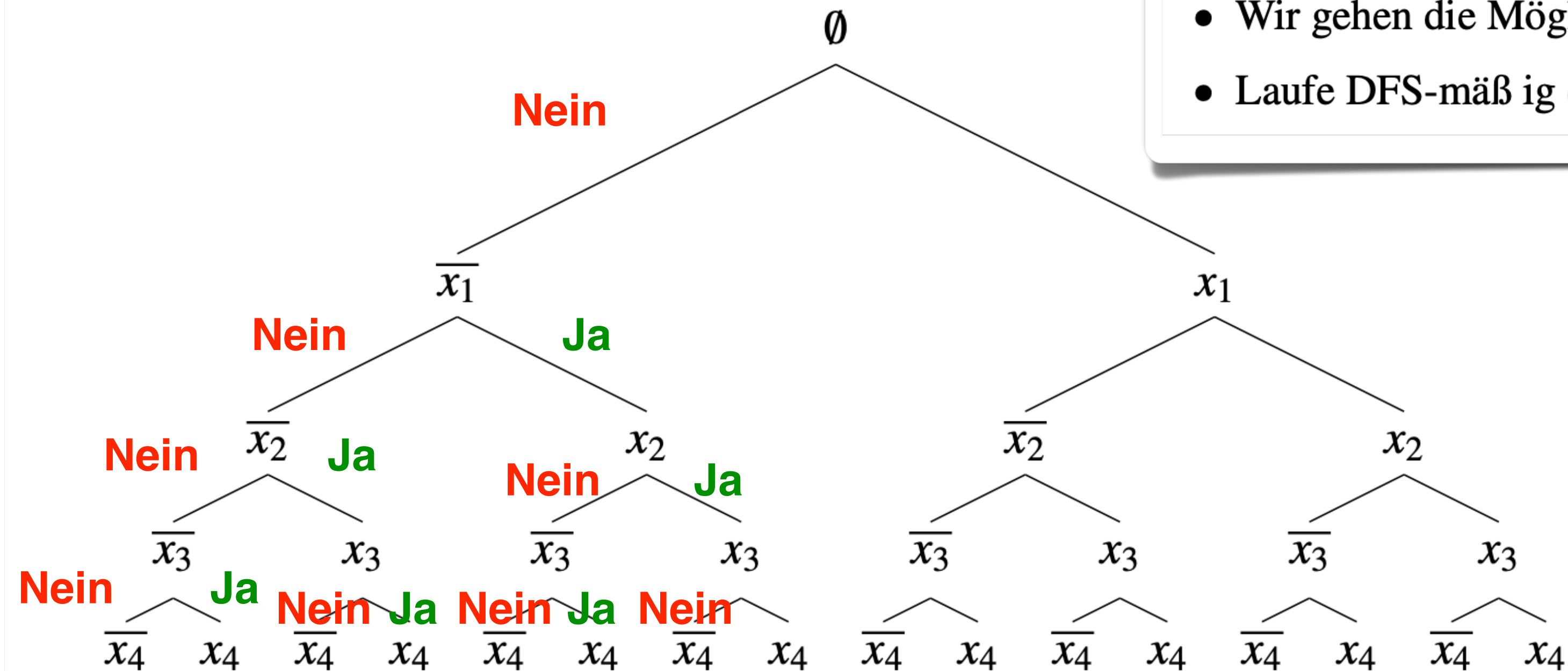
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



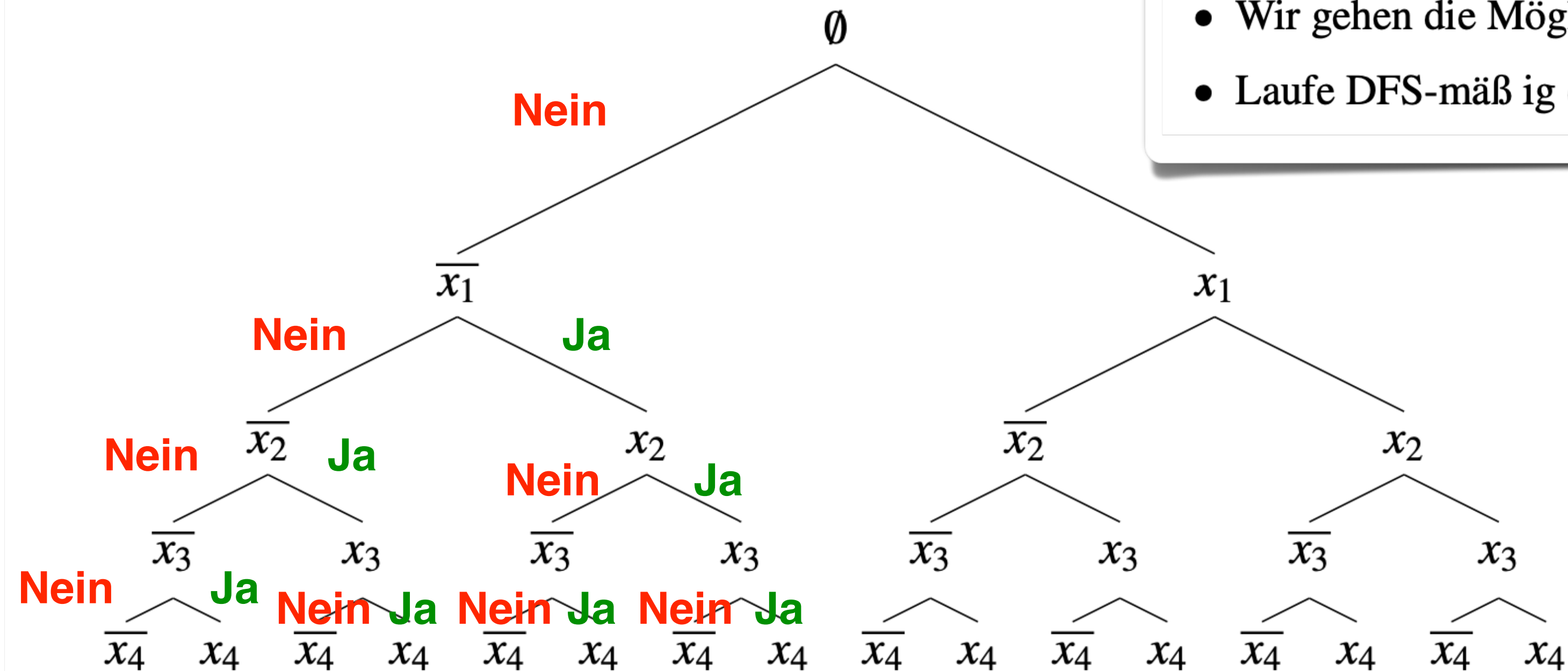
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



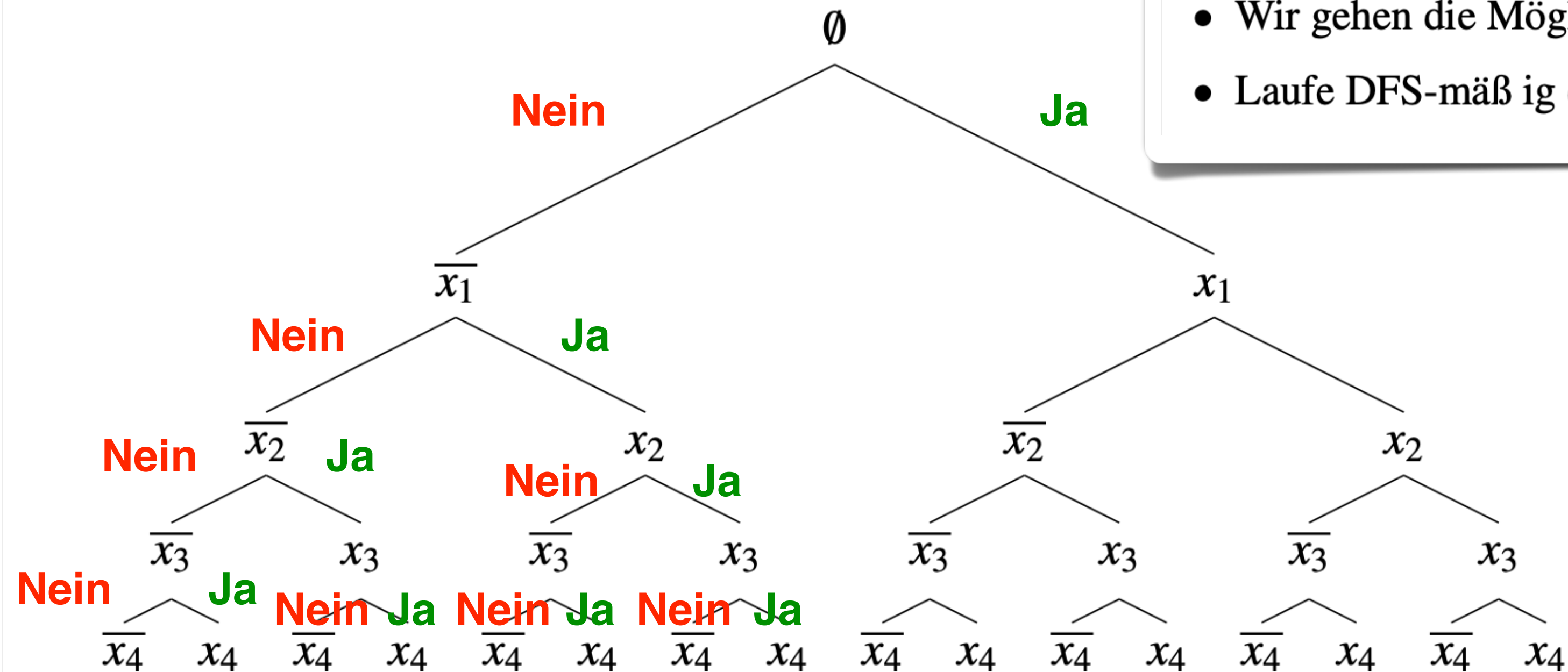
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



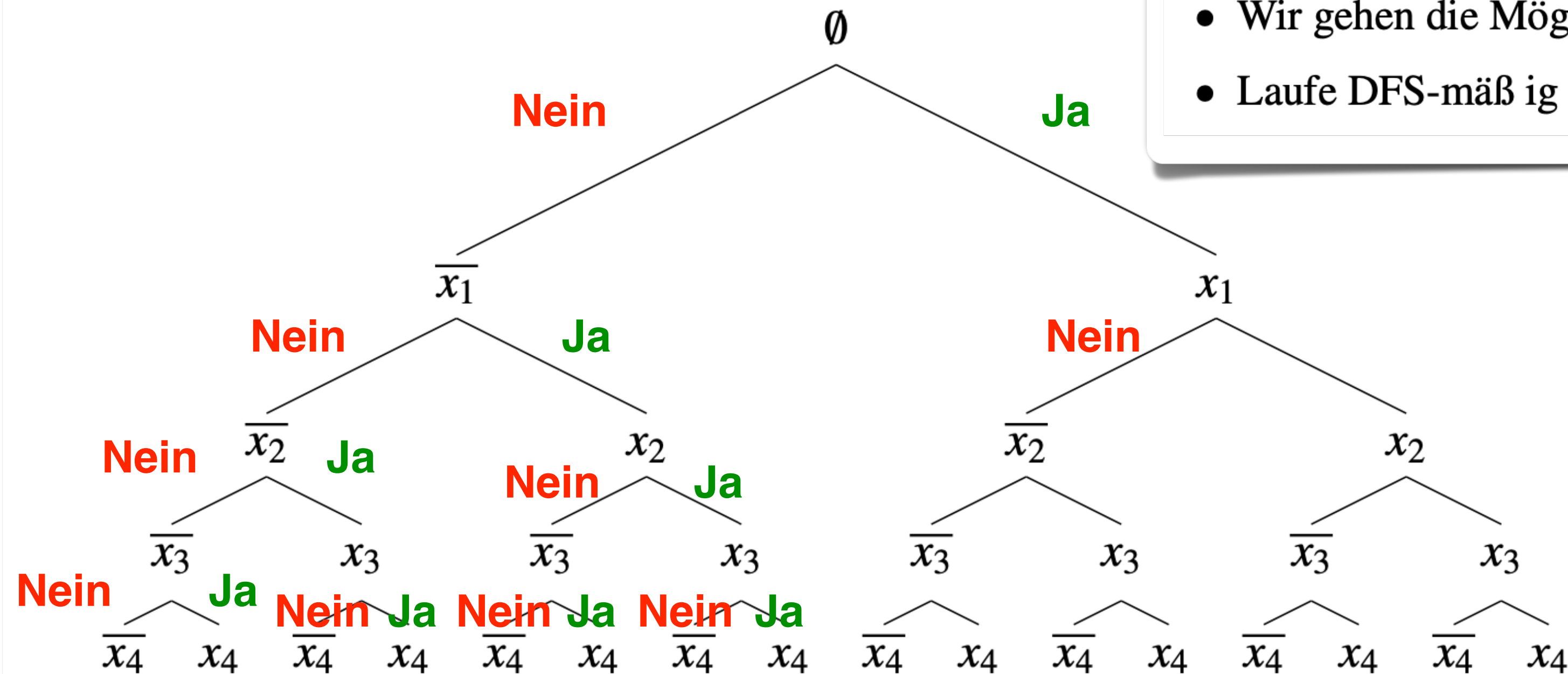
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



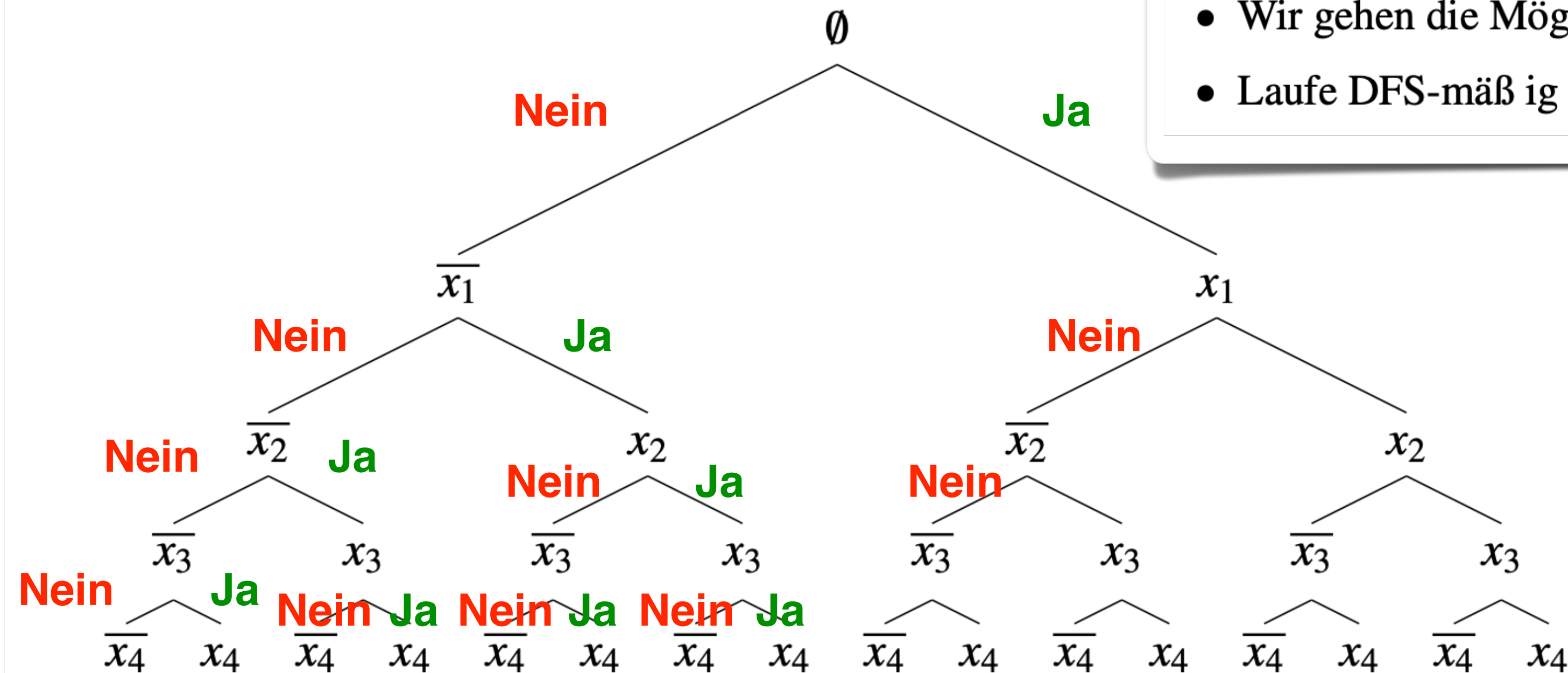
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



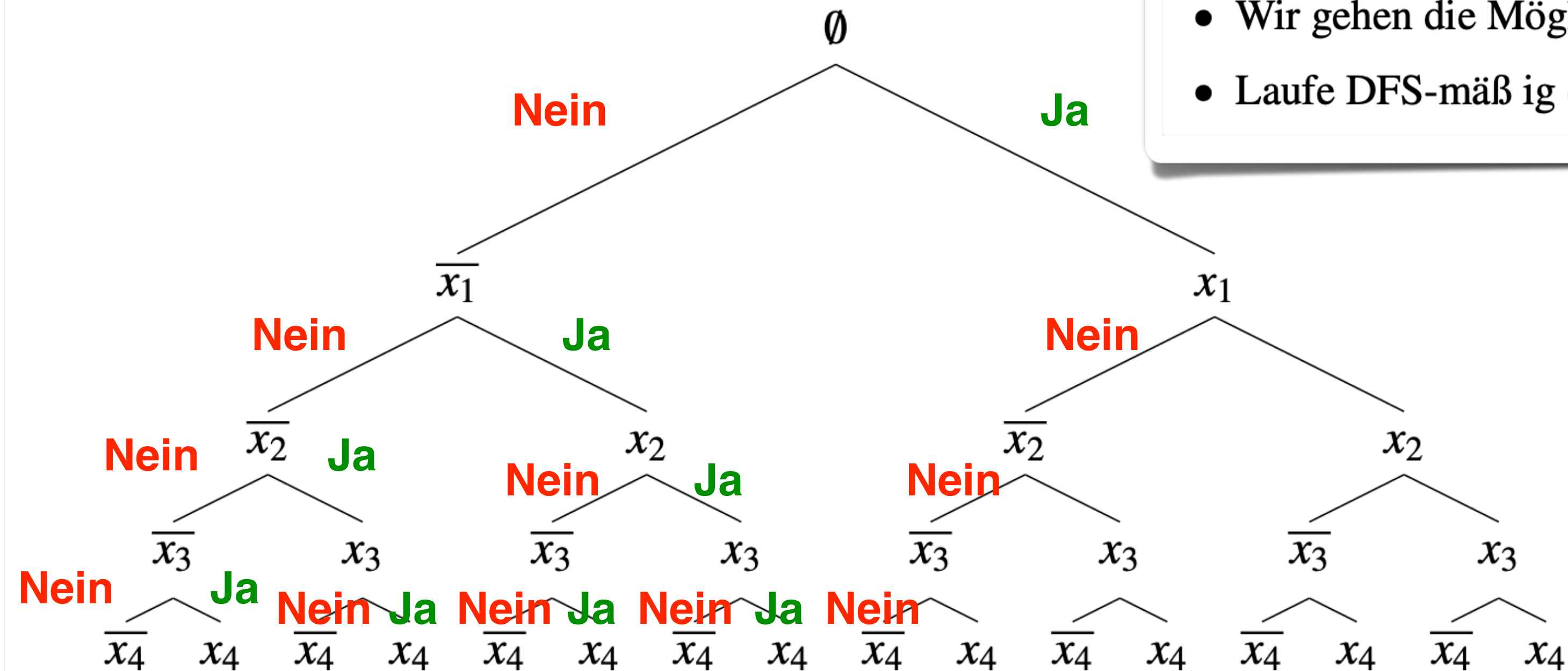
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



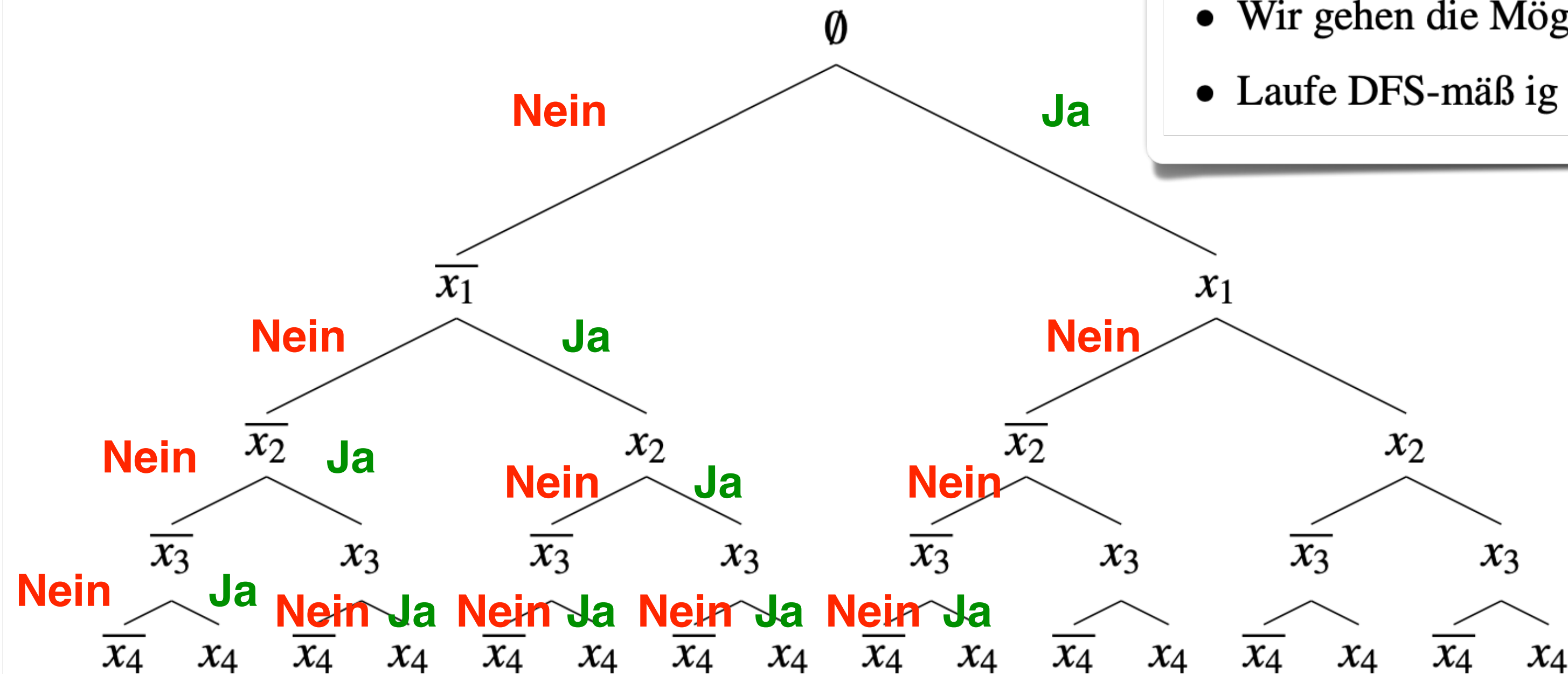
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



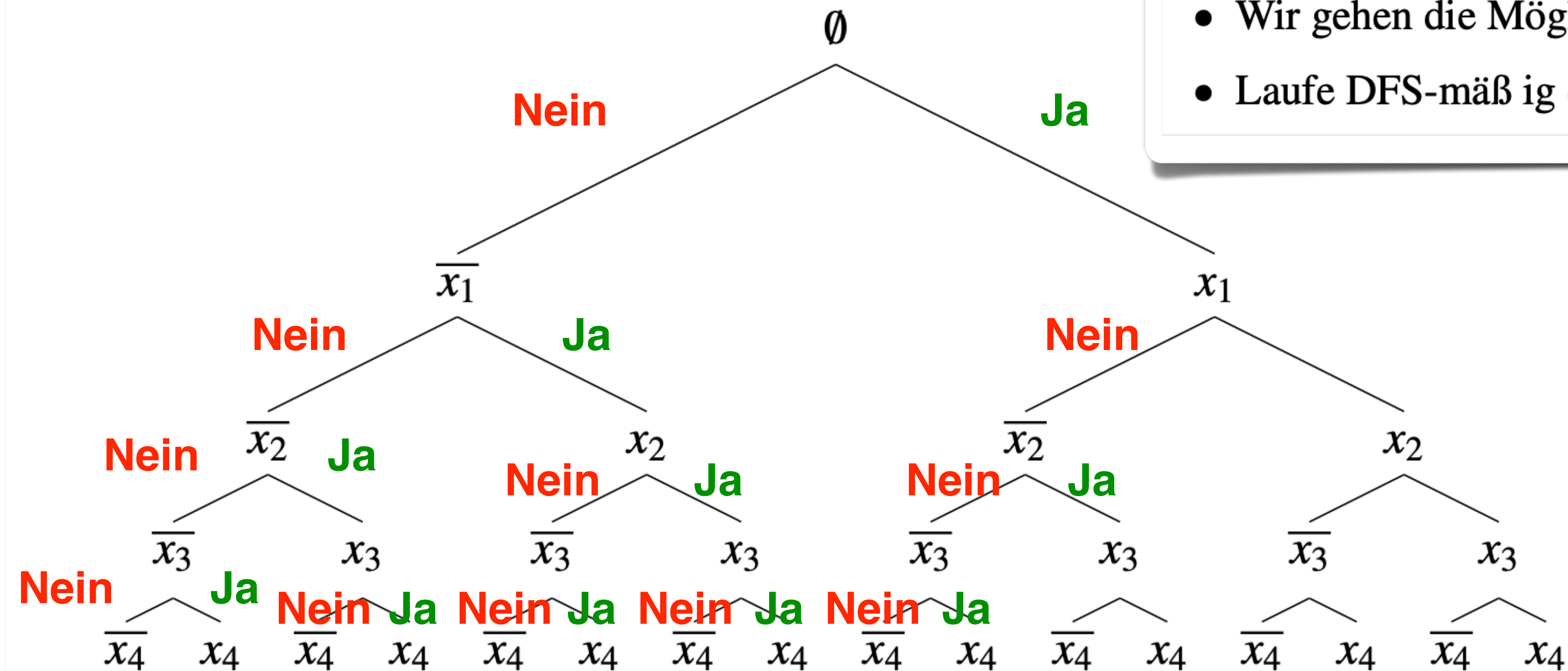
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



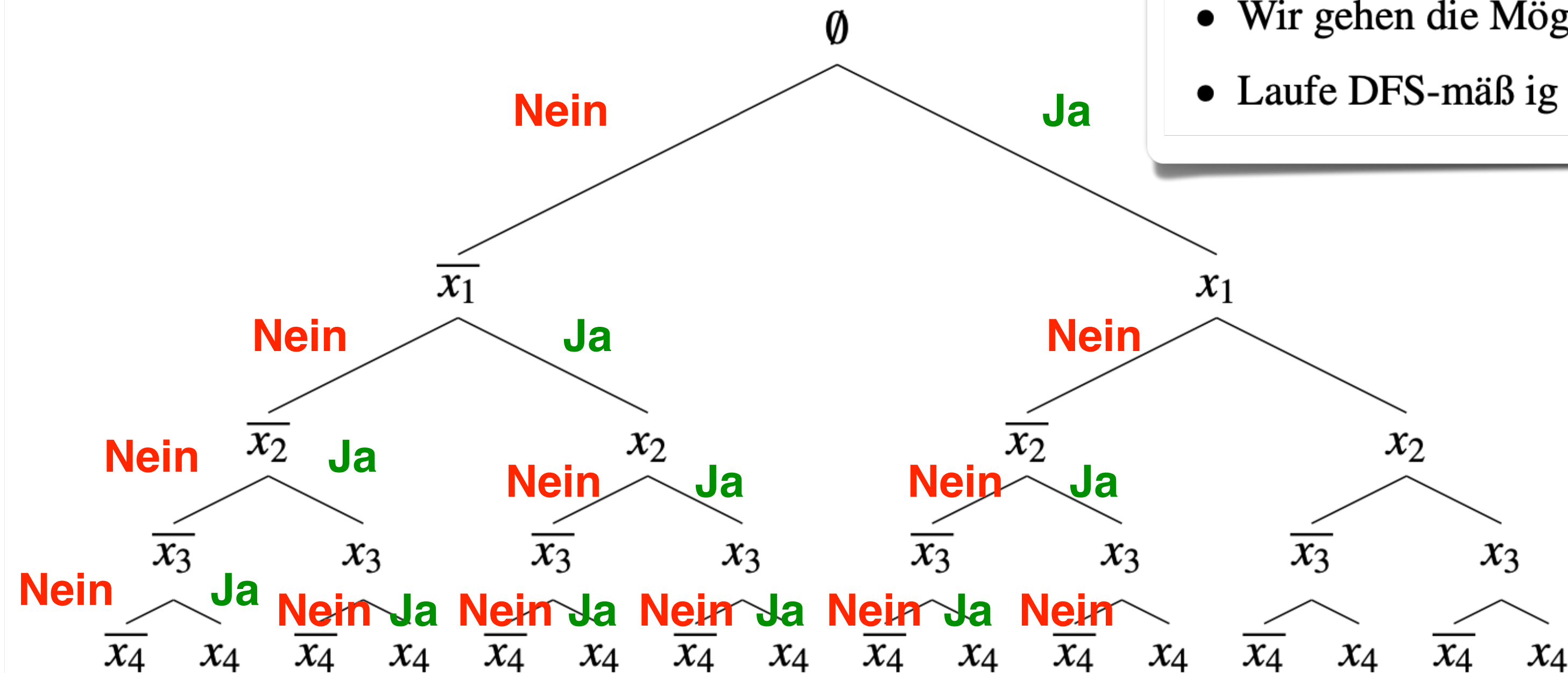
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



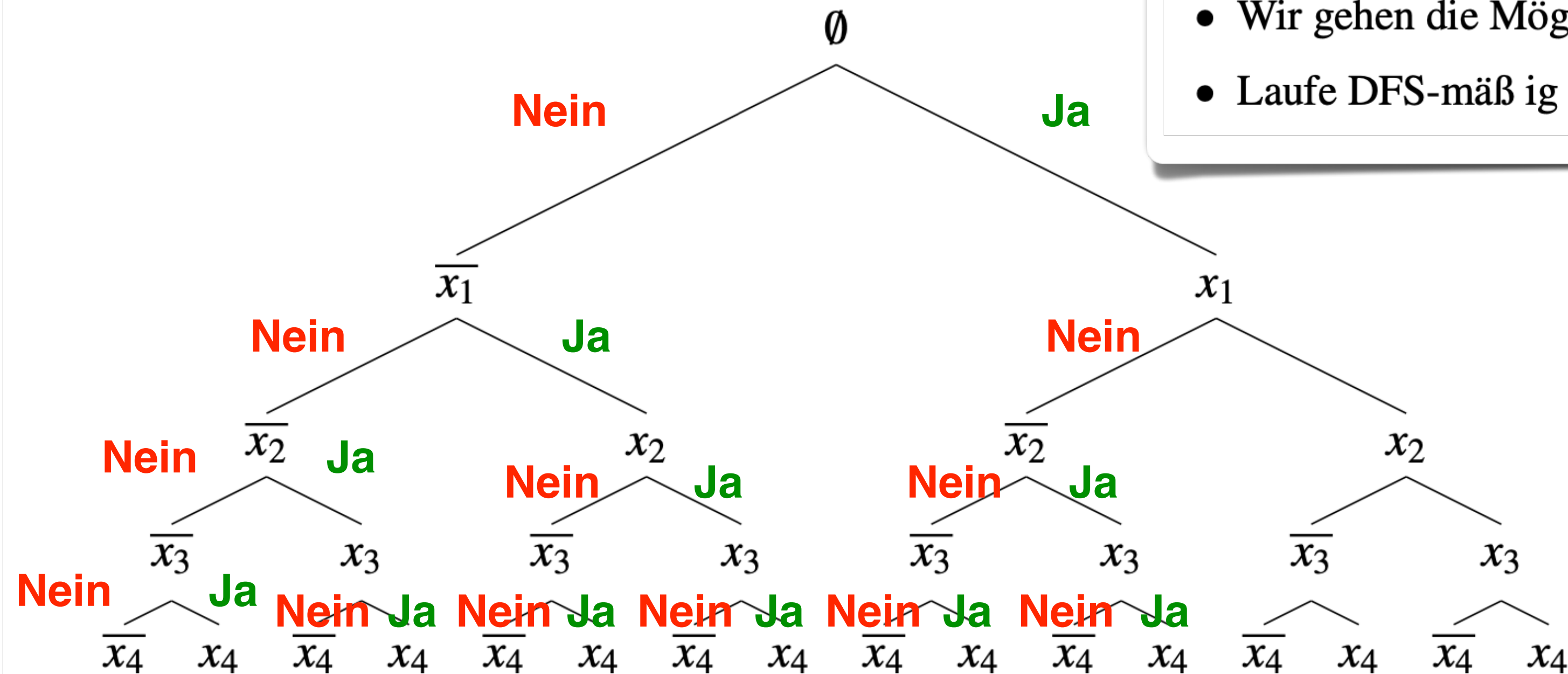
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



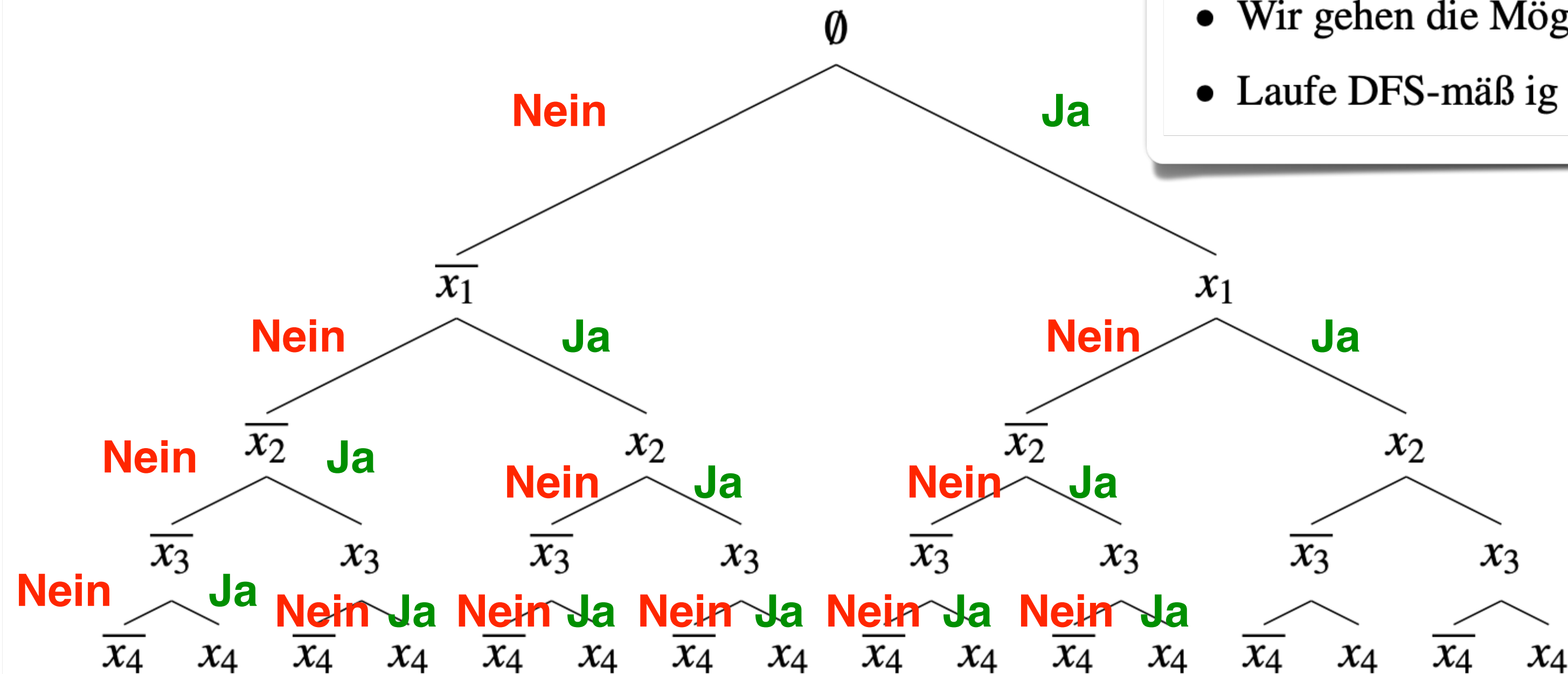
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



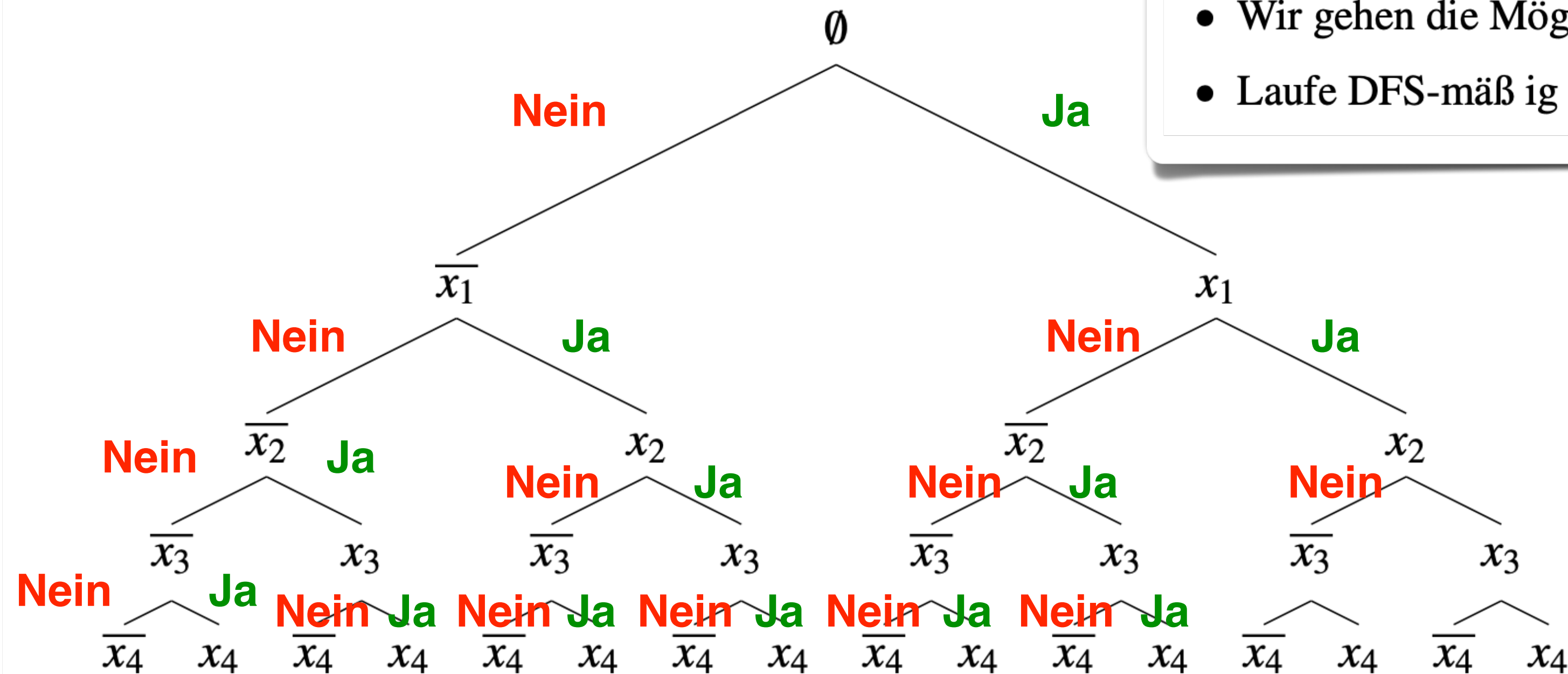
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



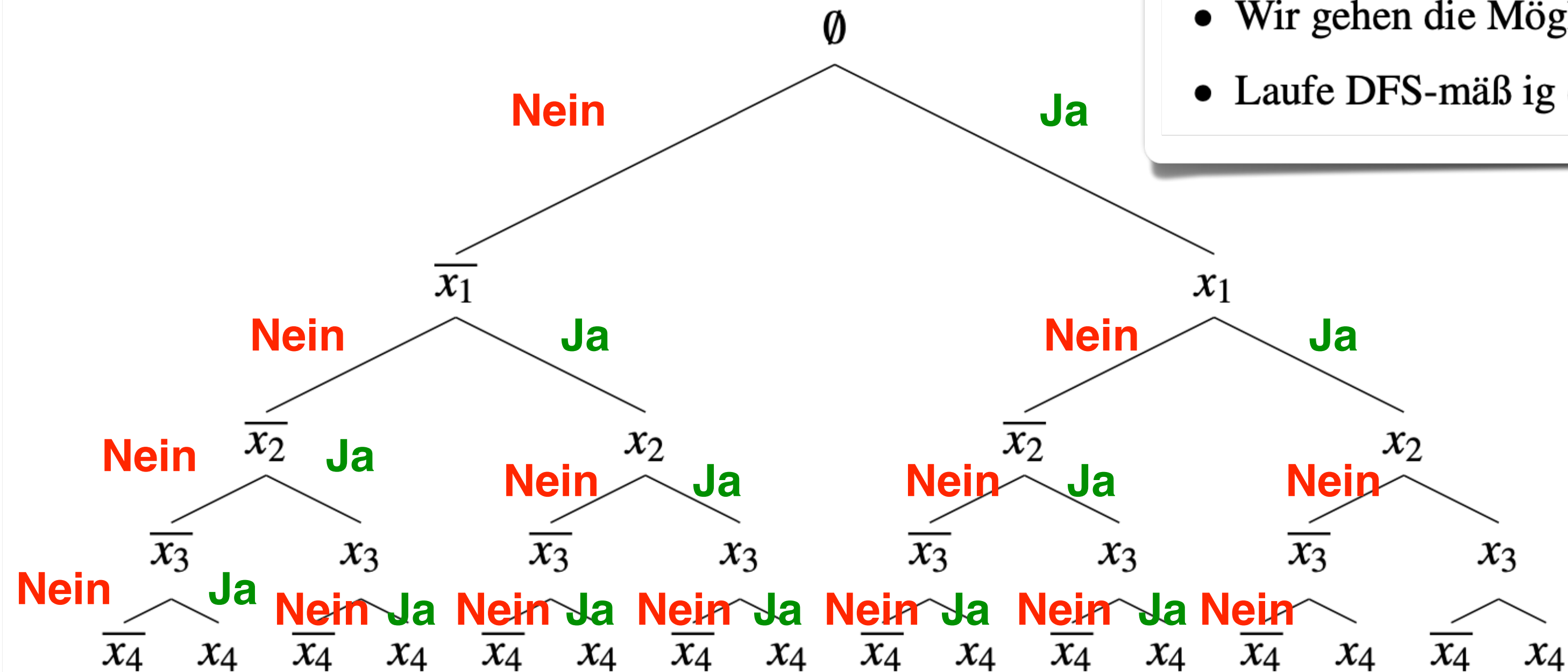
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



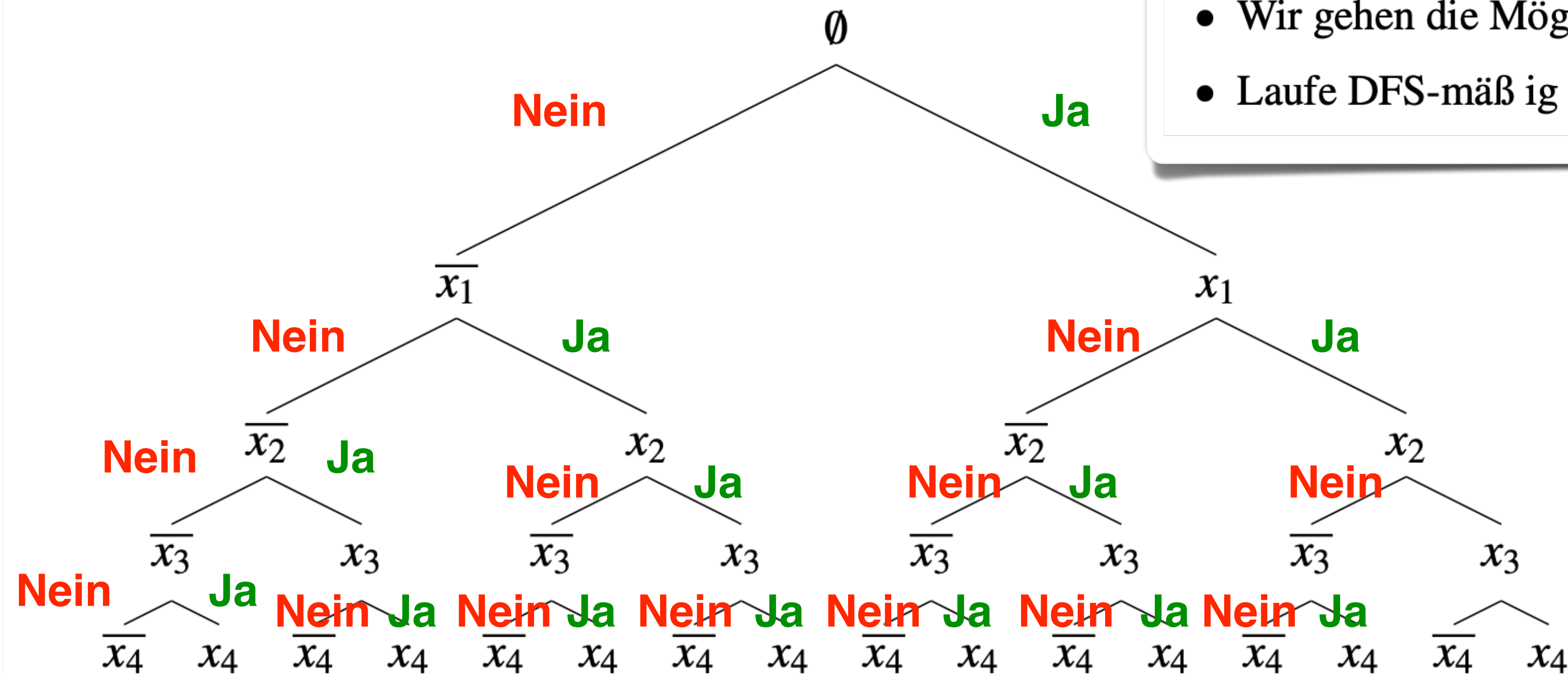
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



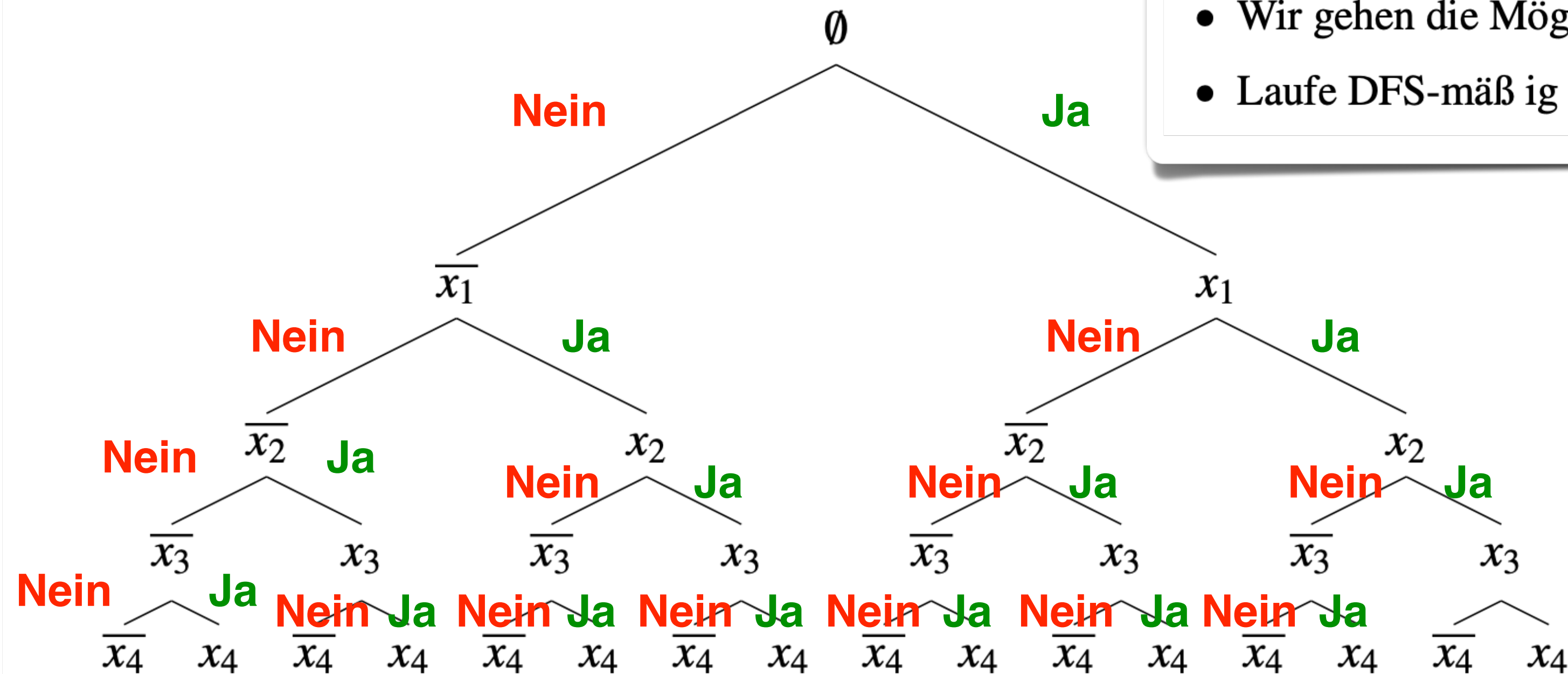
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



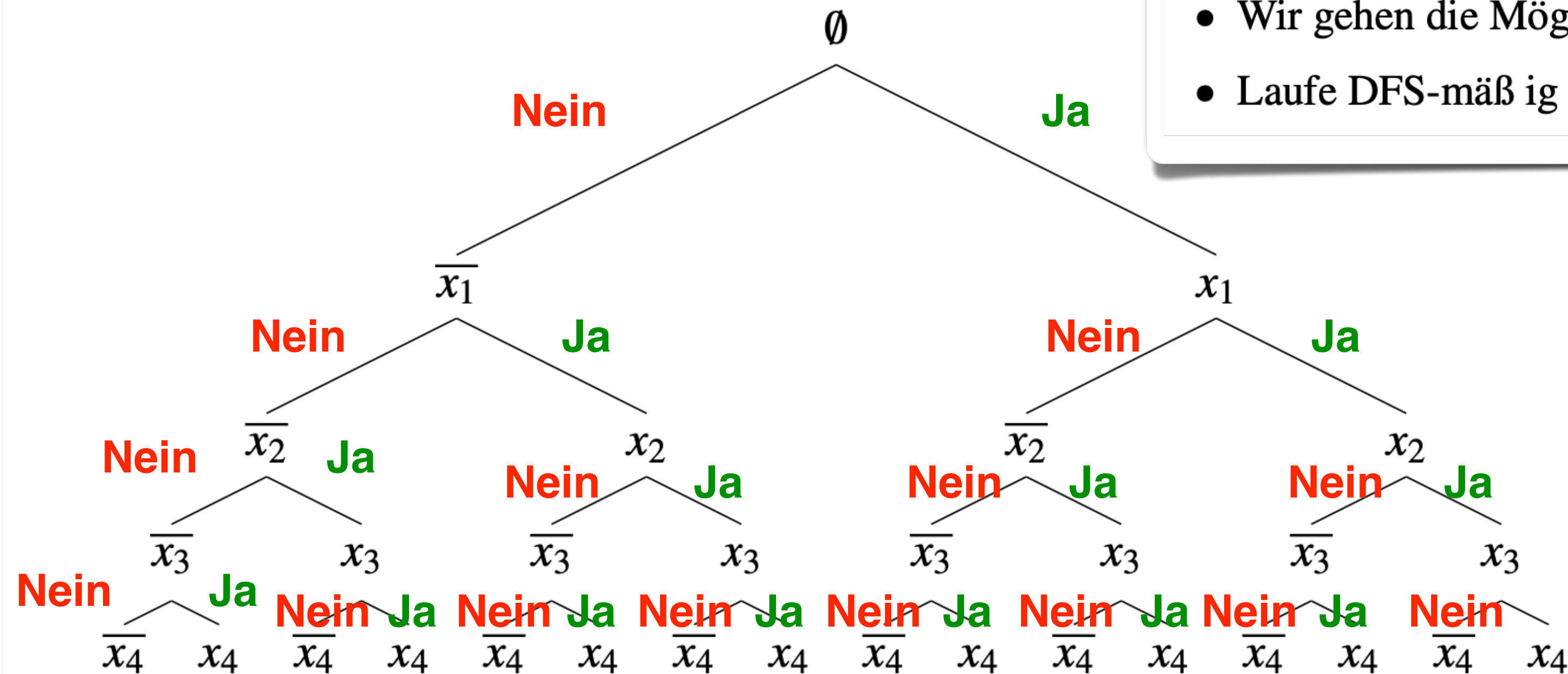
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



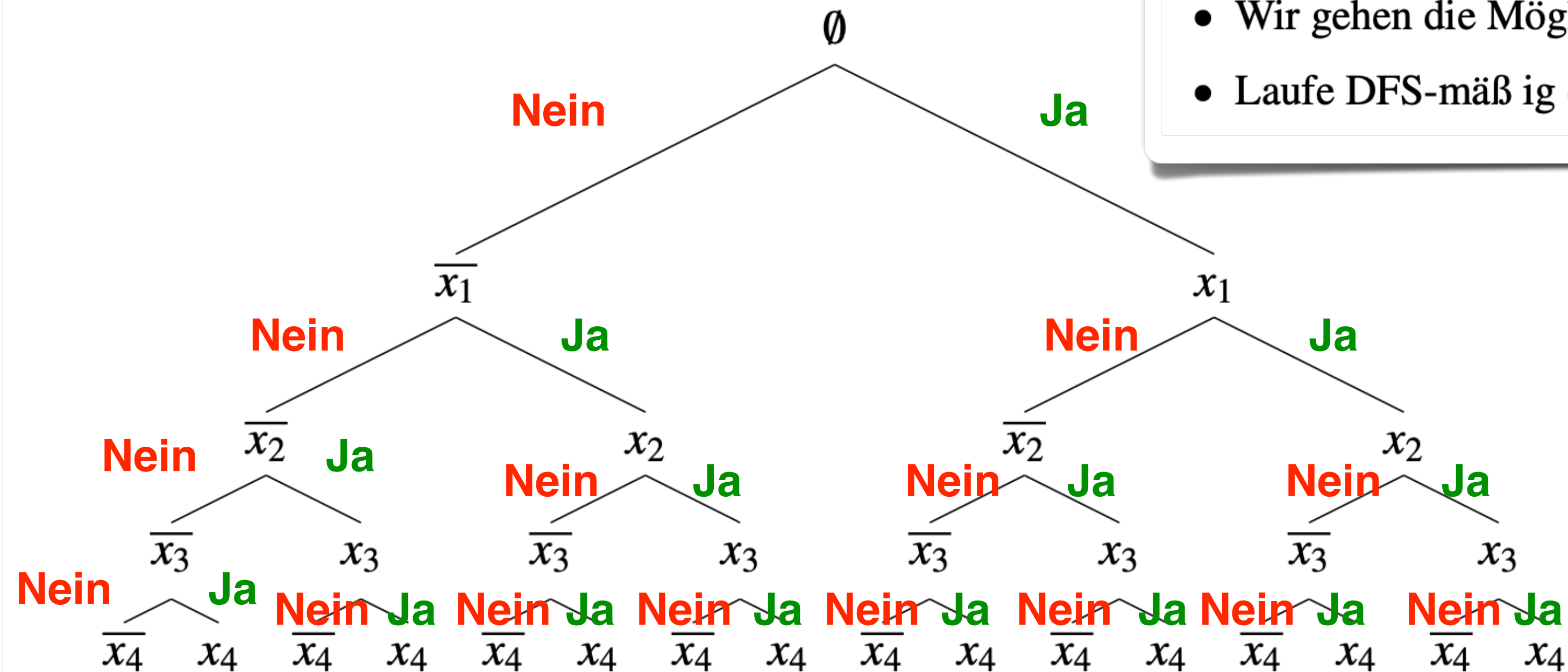
1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.



1. Enumerationsbaum

- Probiere nacheinander für $i = 1, \dots, 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.

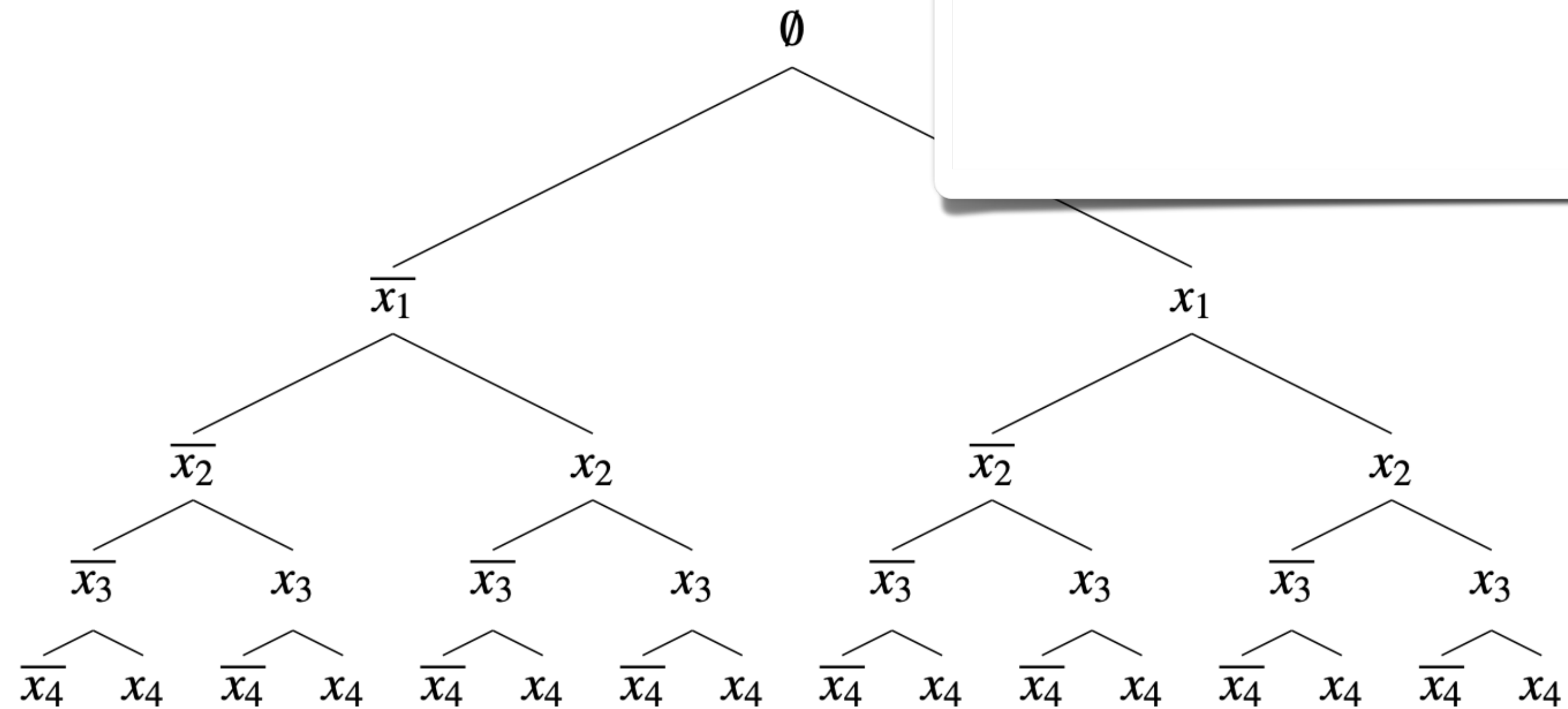


4. Einsatz von Schranken - Grundidee

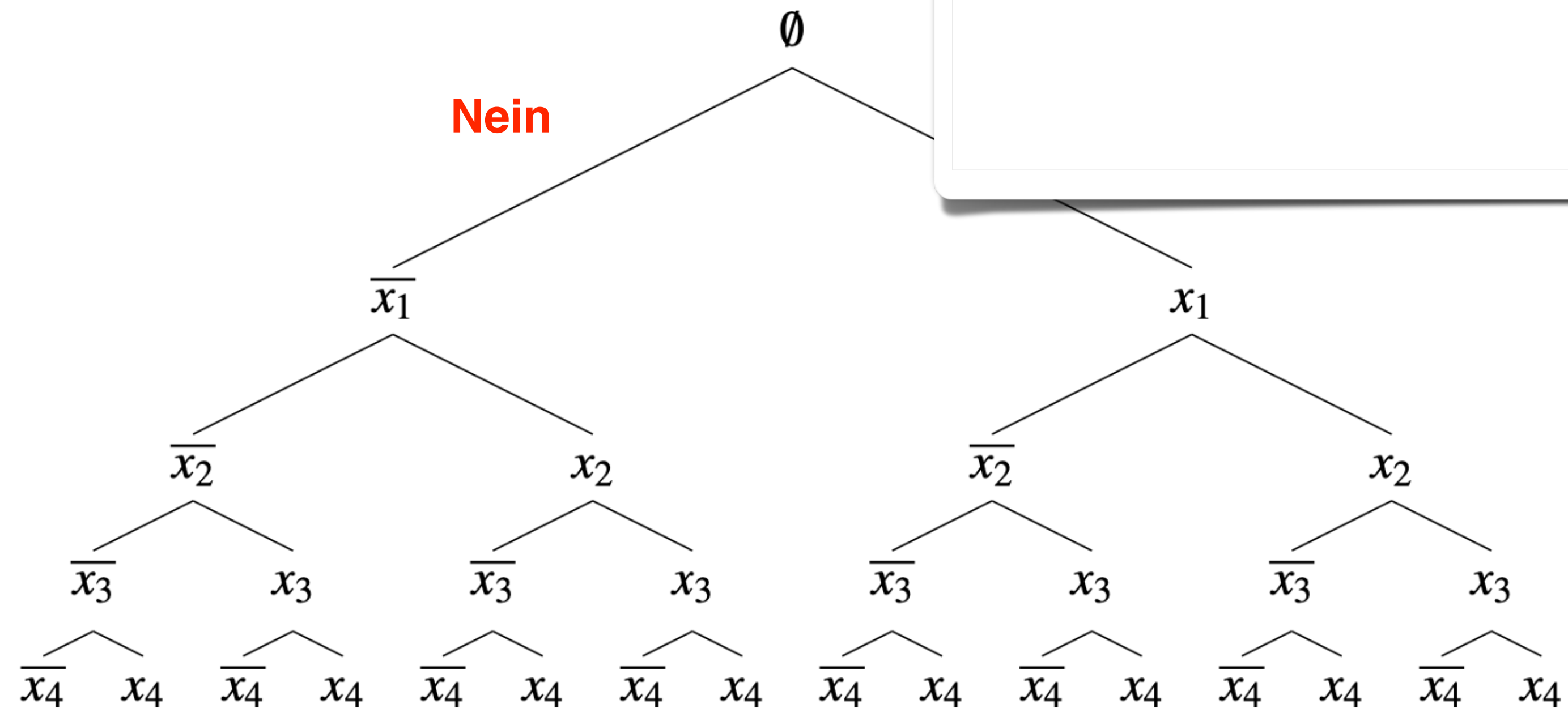
4. Einsatz von Schranken - Grundidee



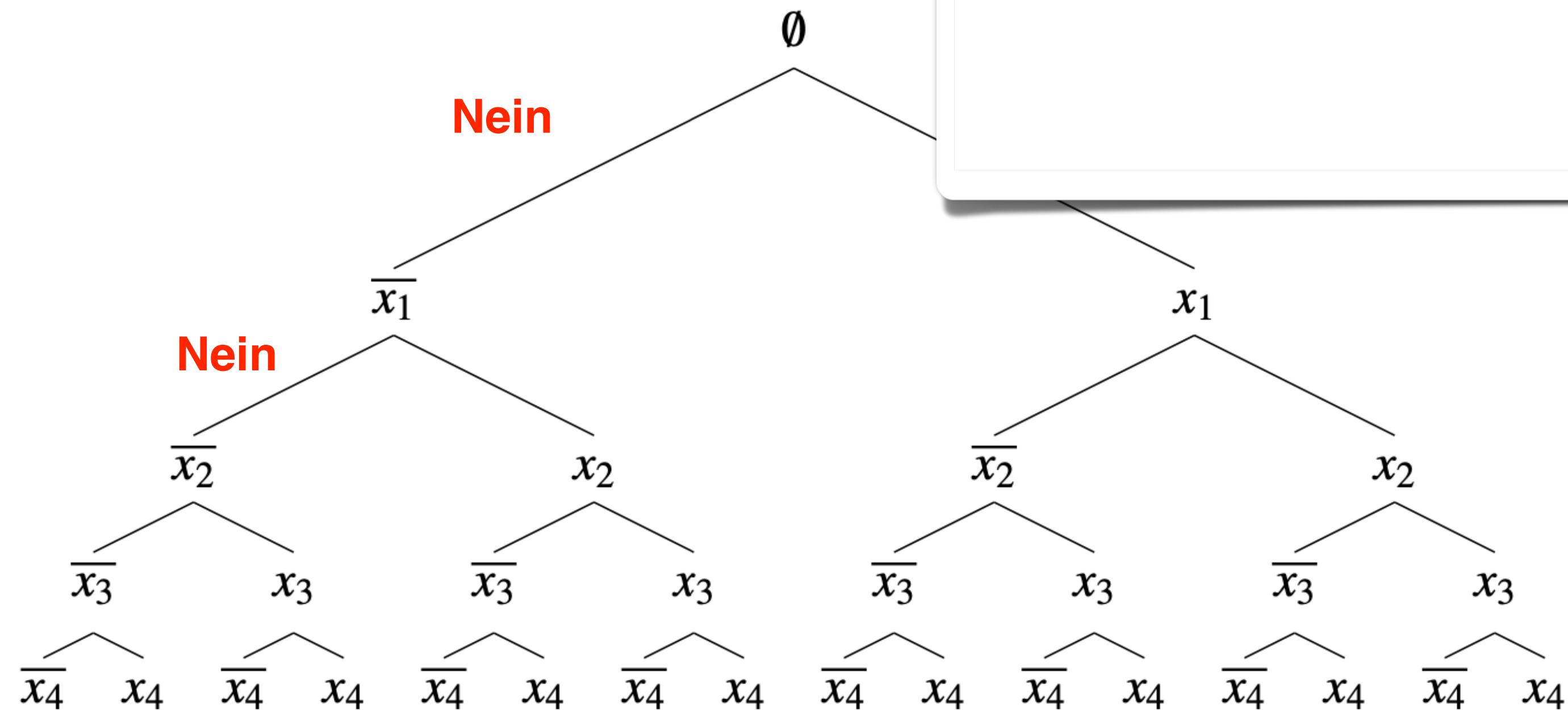
4. Einsatz von Schranken - Grundidee



4. Einsatz von Schranken - Grundidee

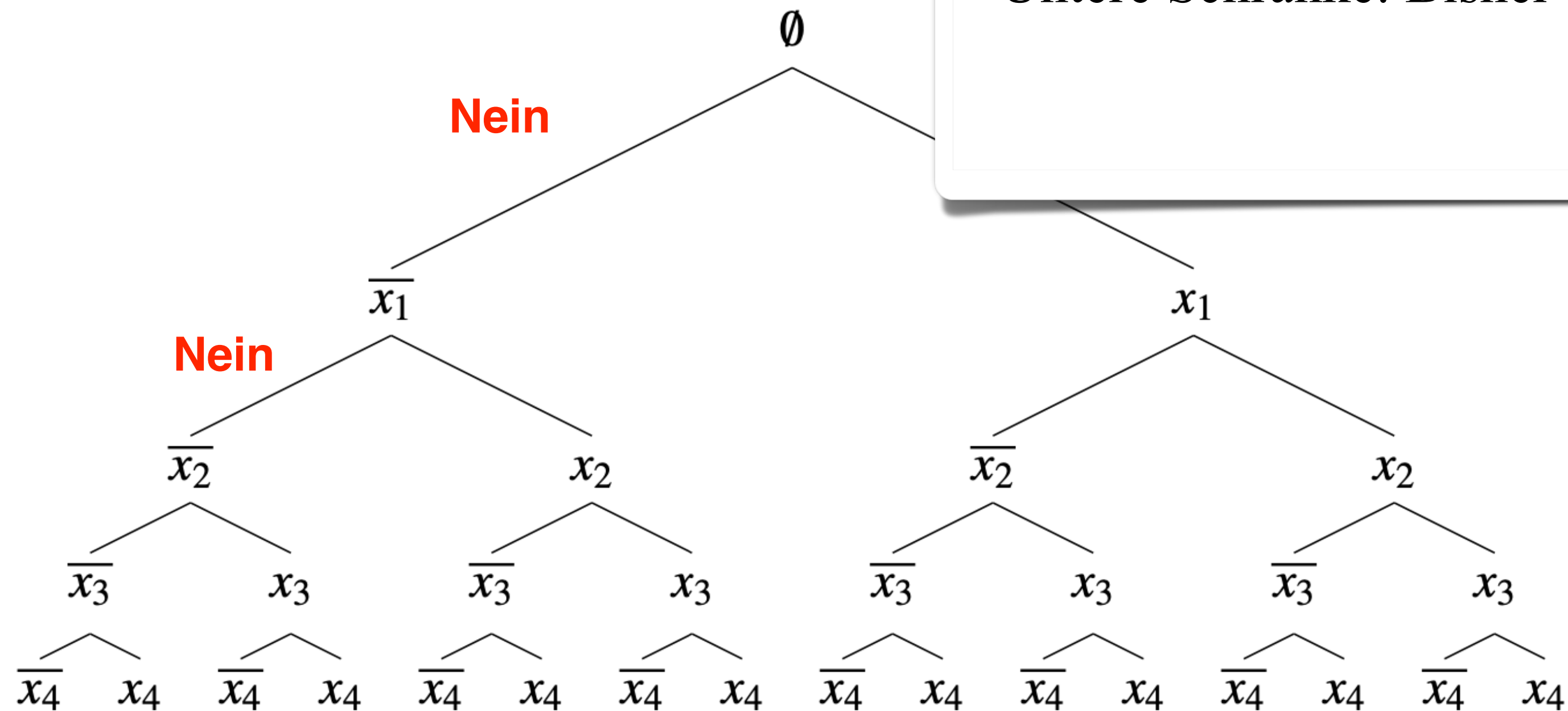


4. Einsatz von Schranken - Grundidee



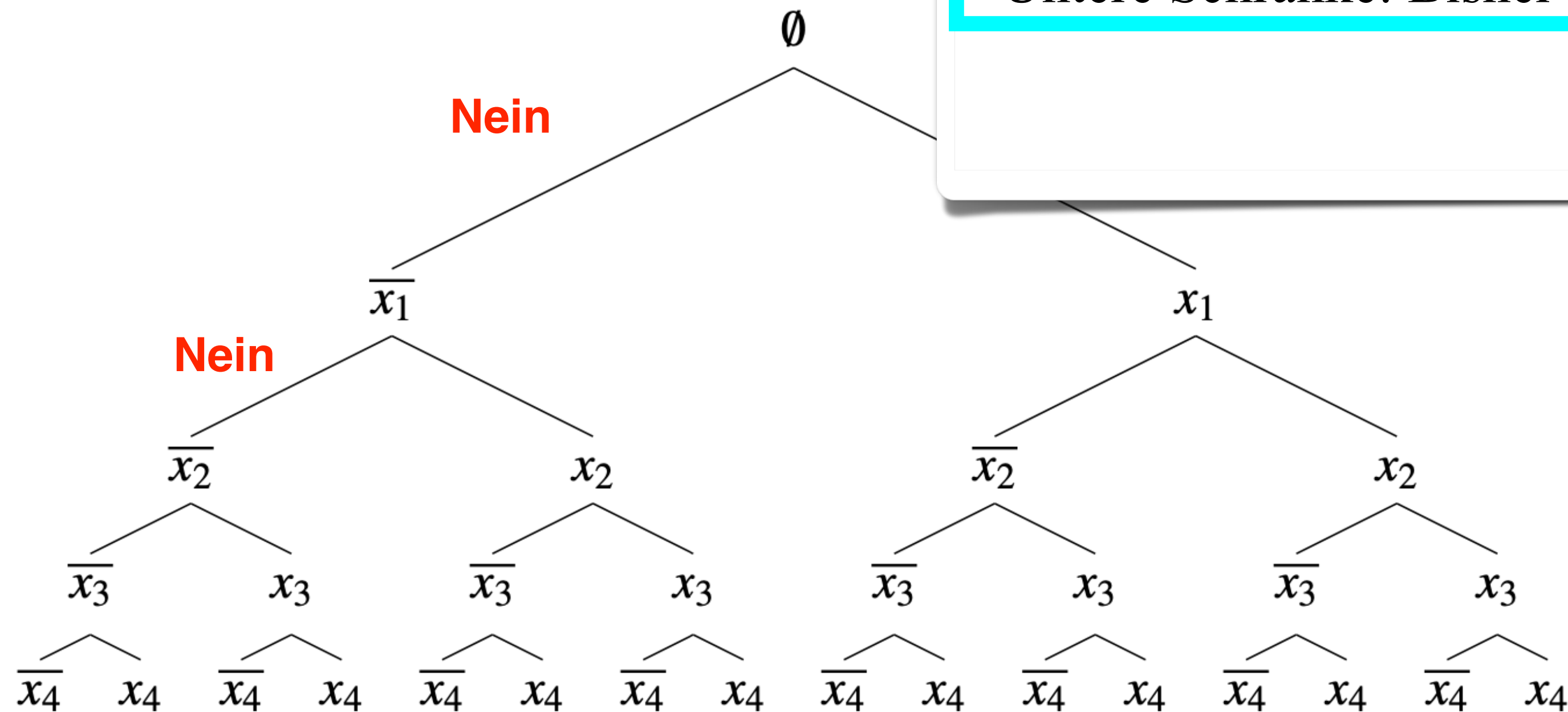
4. Einsatz von Schranken - Grundidee

- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum



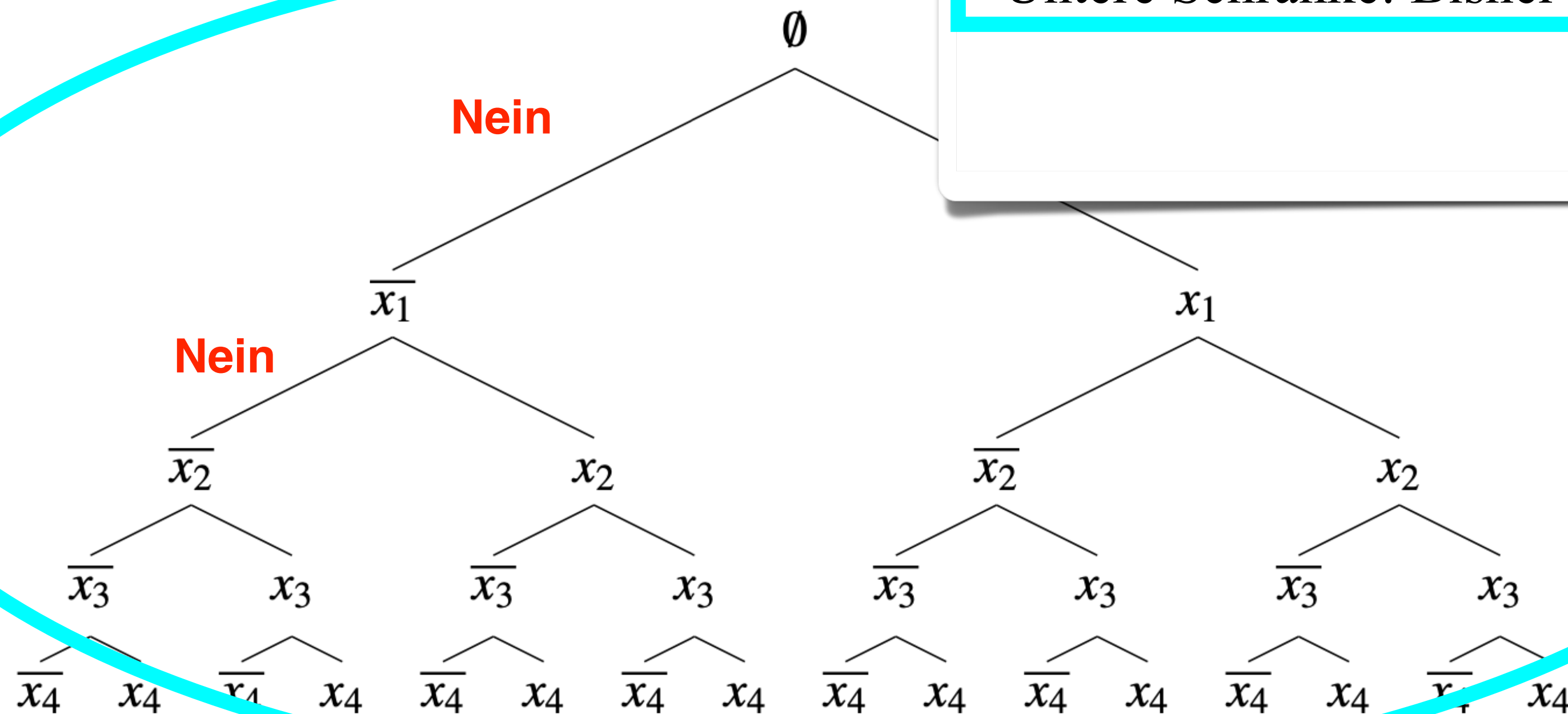
4. Einsatz von Schranken - Grundidee

- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum



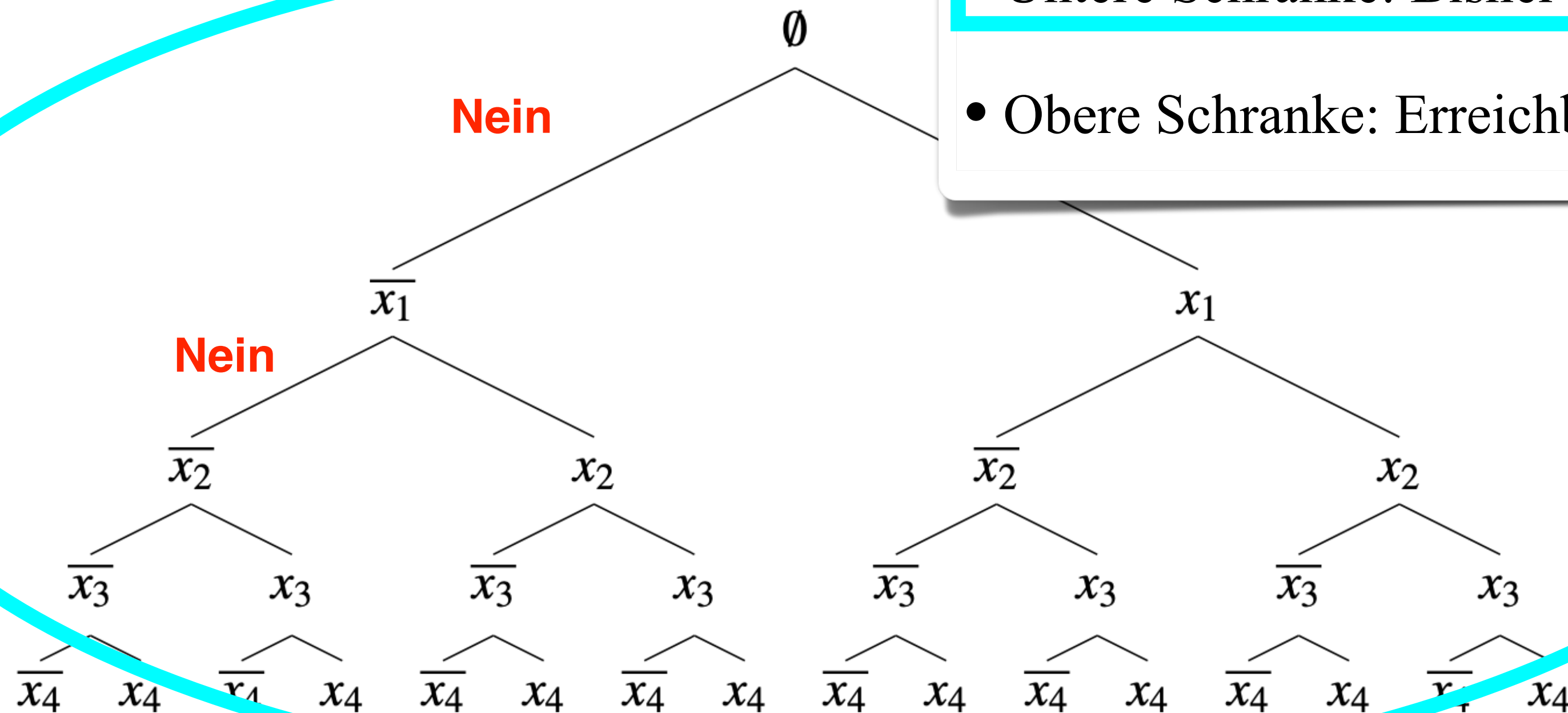
4. Einsatz von Schranken - Grundidee

- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum



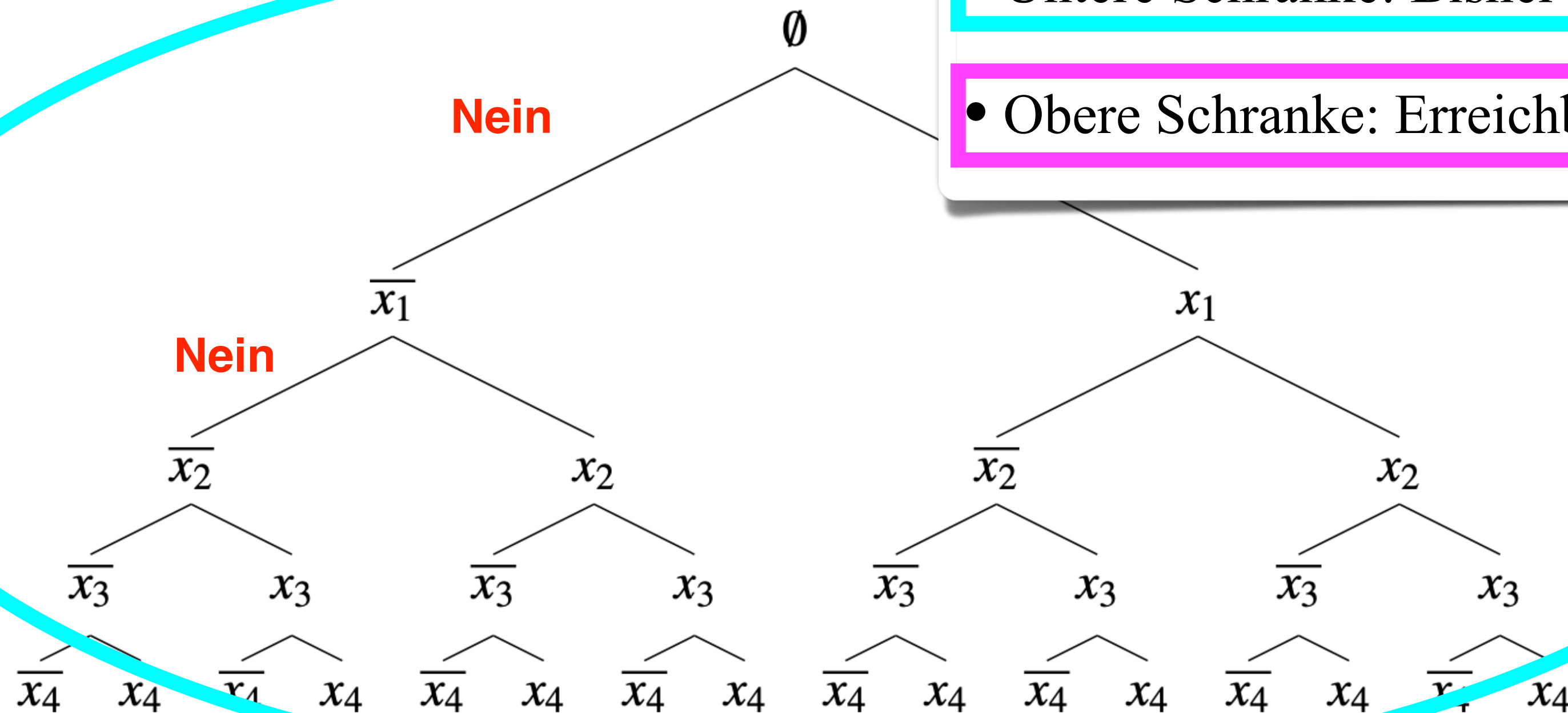
4. Einsatz von Schranken - Grundidee

- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum

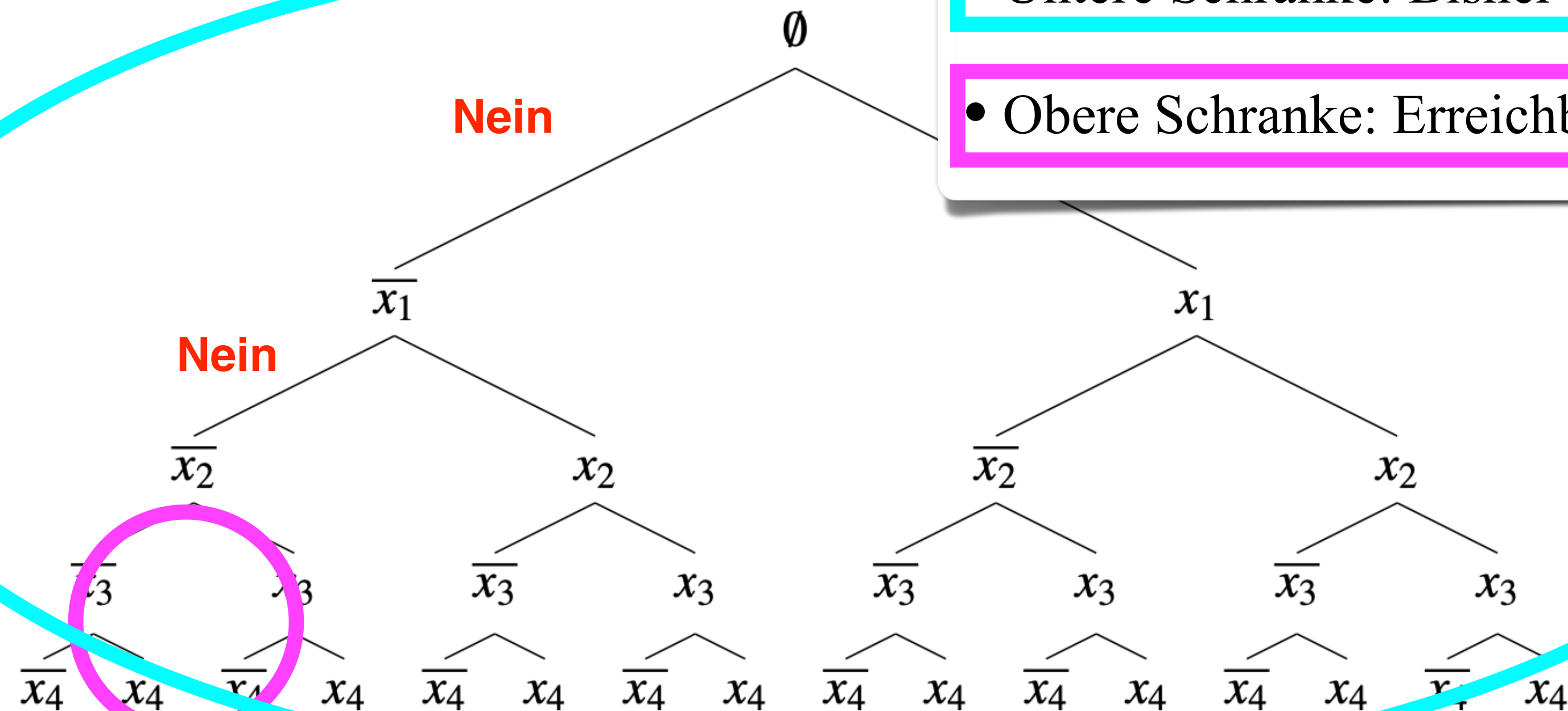


4. Einsatz von Schranken - Grundidee

- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum

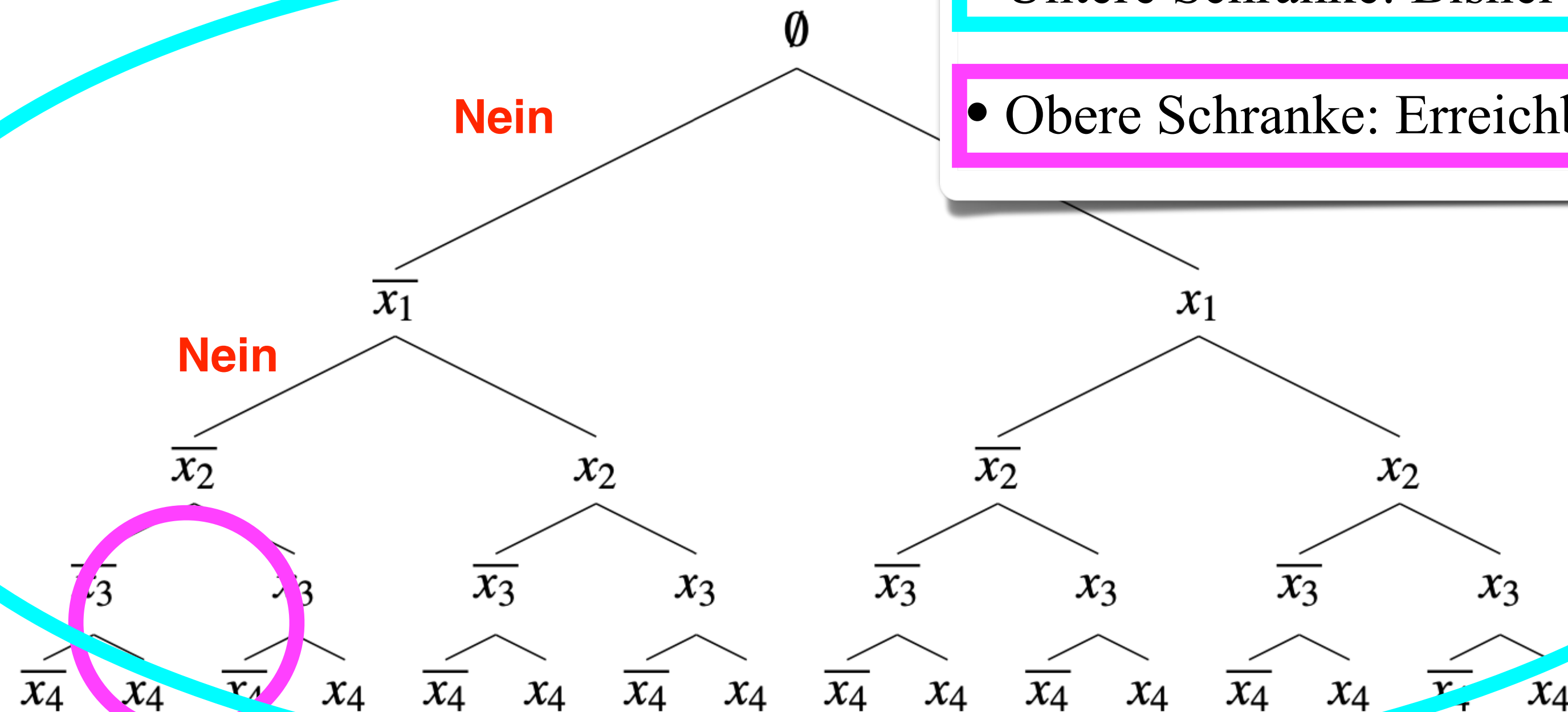


4. Einsatz von Schranken - Grundidee



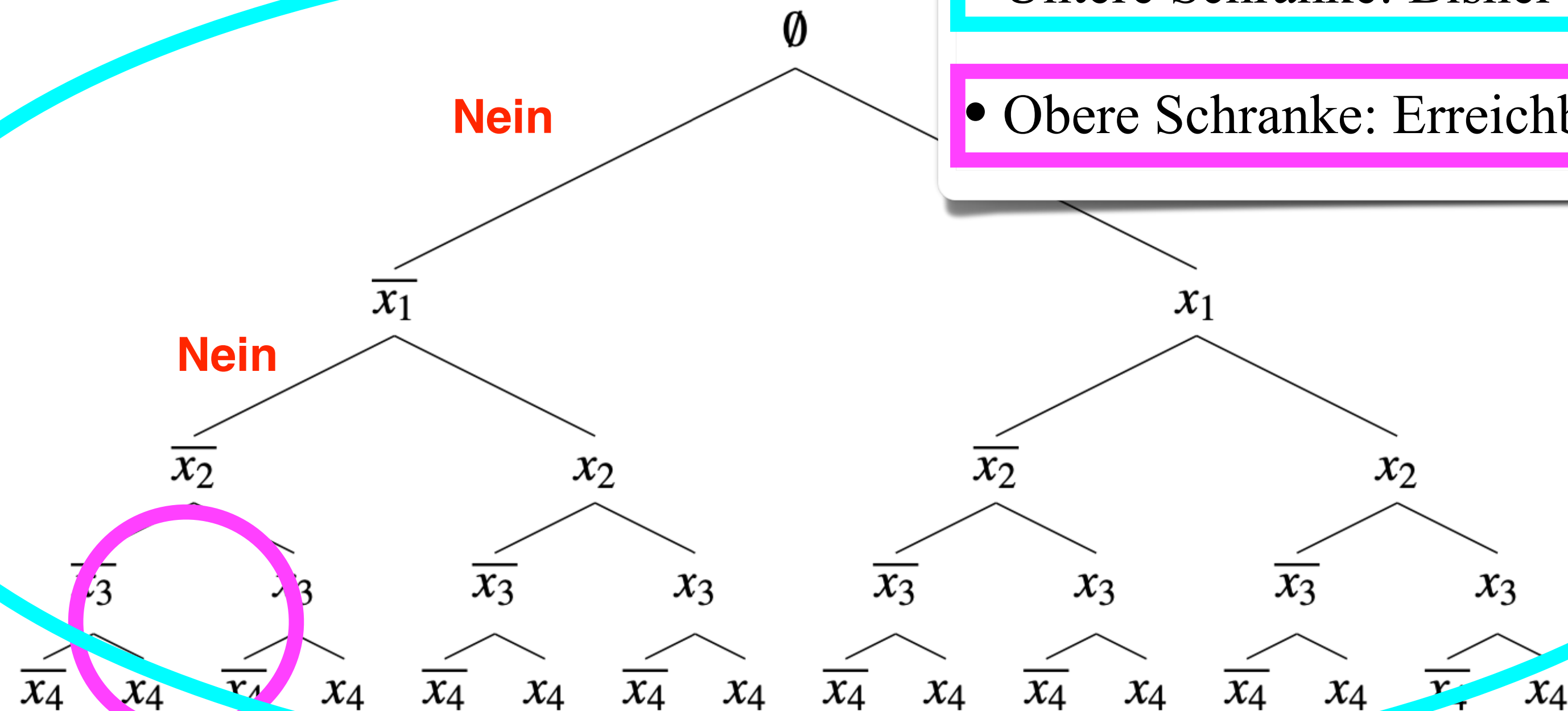
- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum

4. Einsatz von Schranken - Grundidee

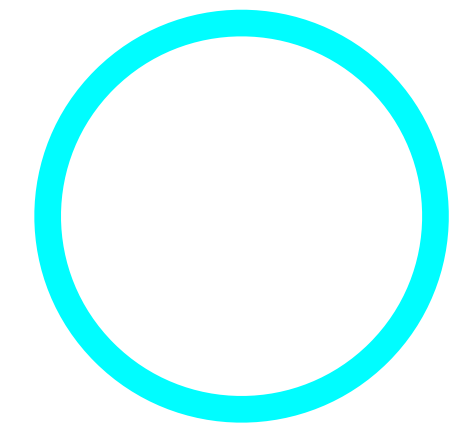
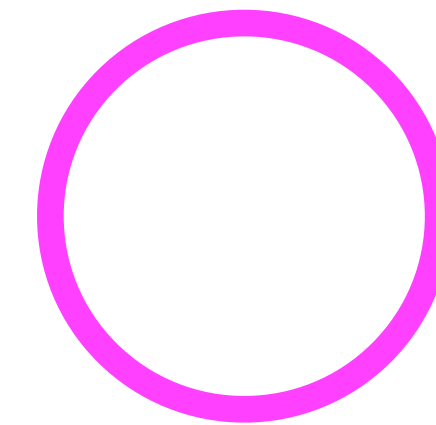


- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum

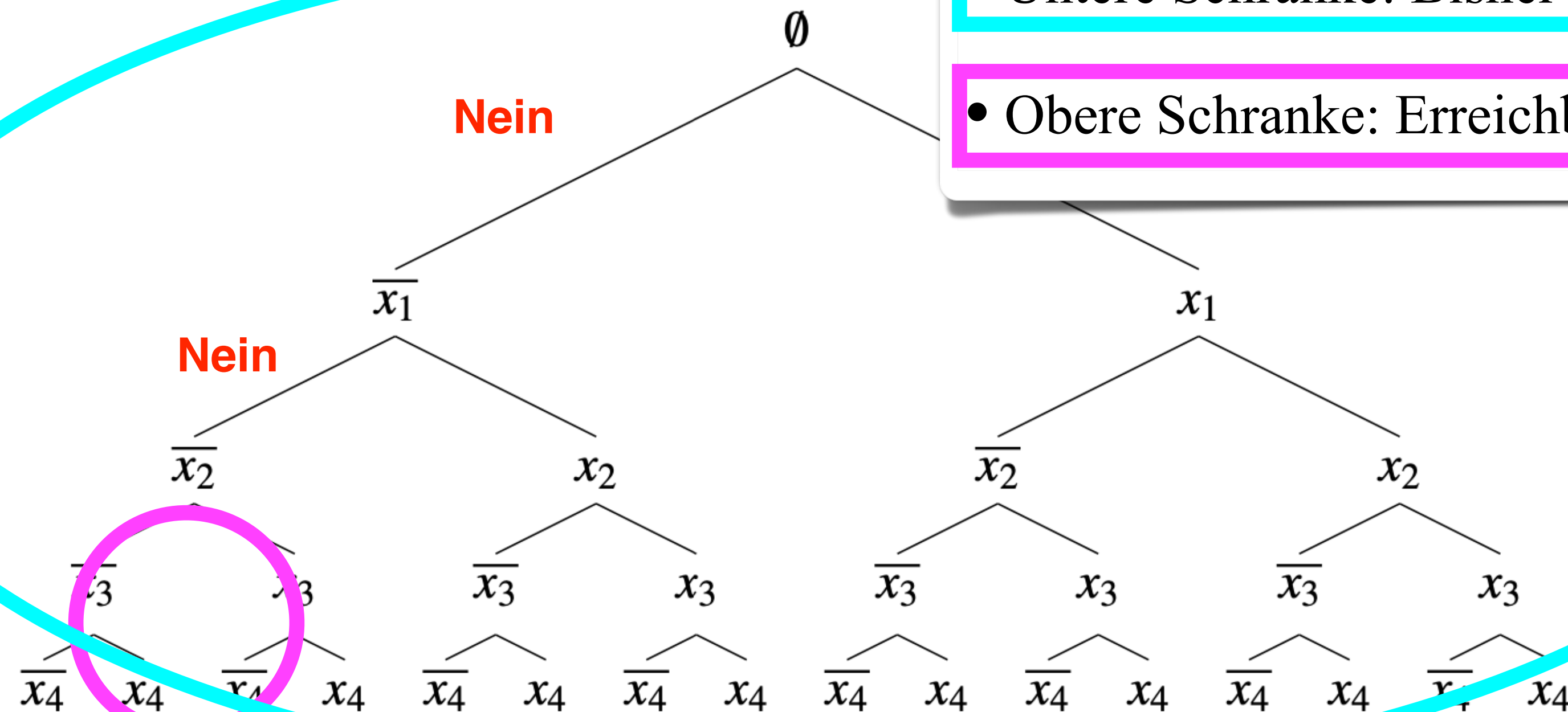
4. Einsatz von Schranken - Grundidee



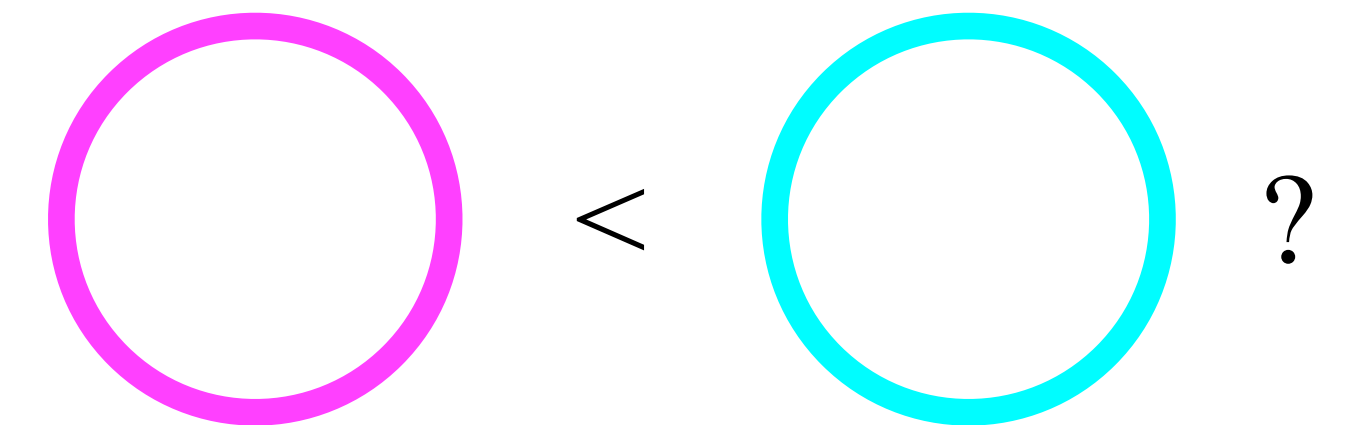
- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum



4. Einsatz von Schranken - Grundidee

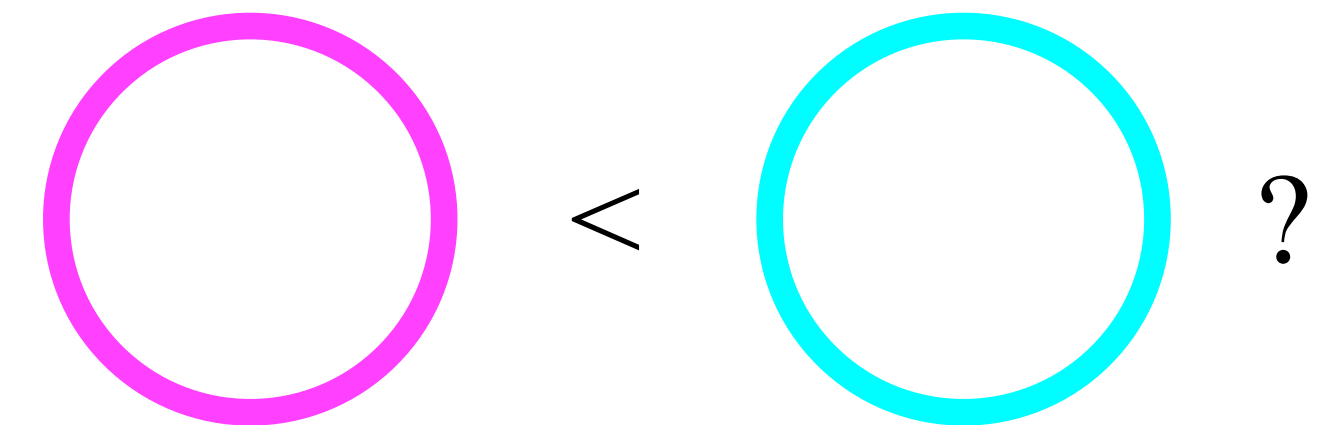
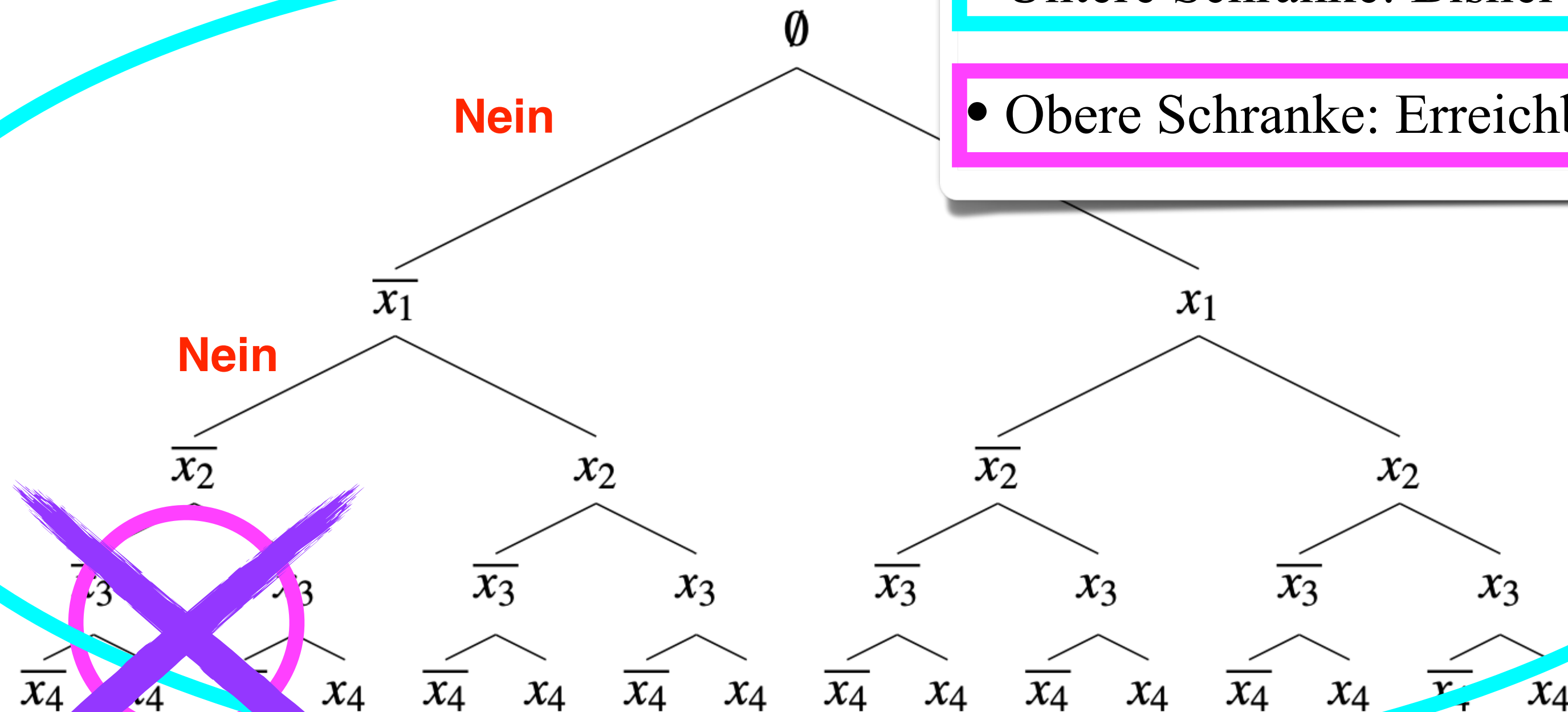


- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum



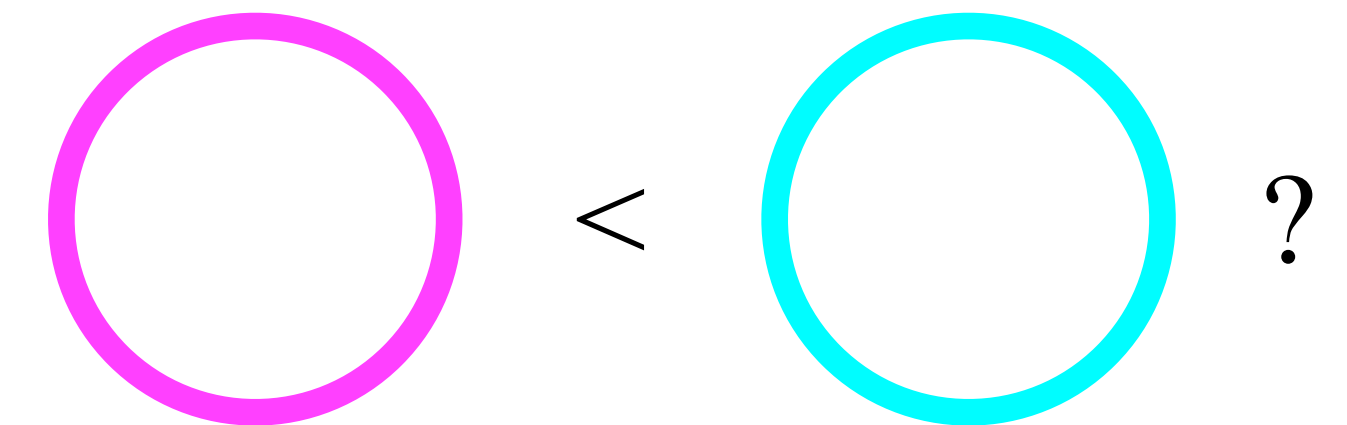
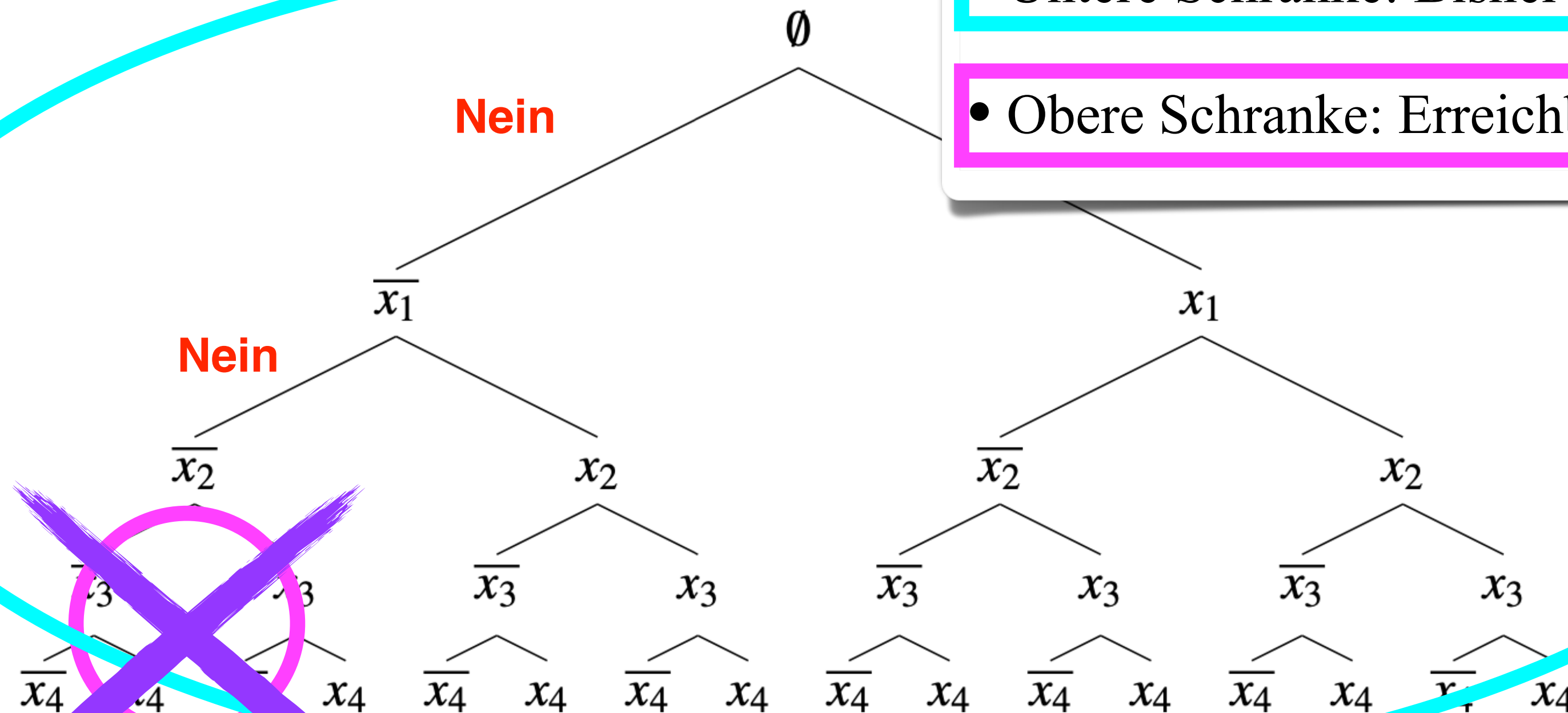
4. Einsatz von Schranken - Grundidee

- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum



4. Einsatz von Schranken - Grundidee

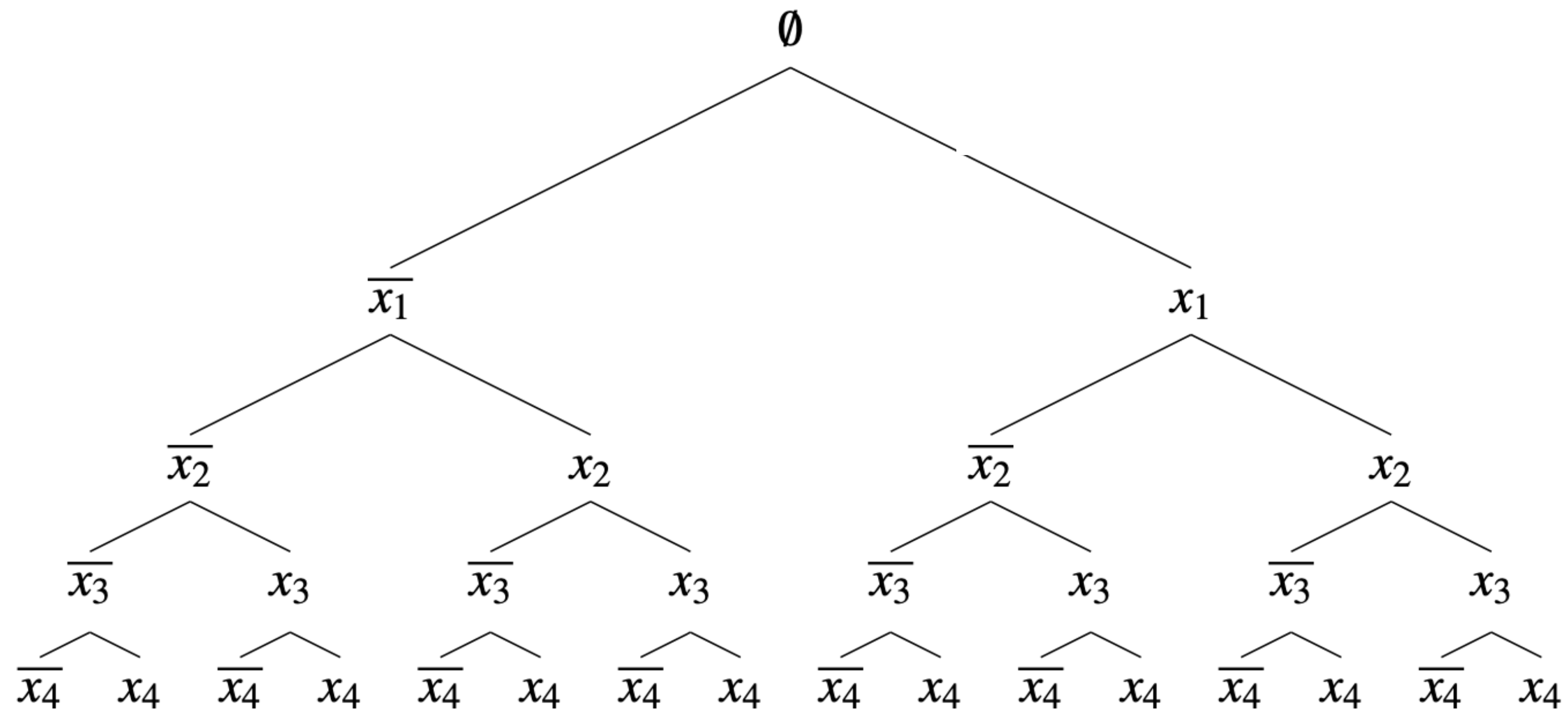
- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum



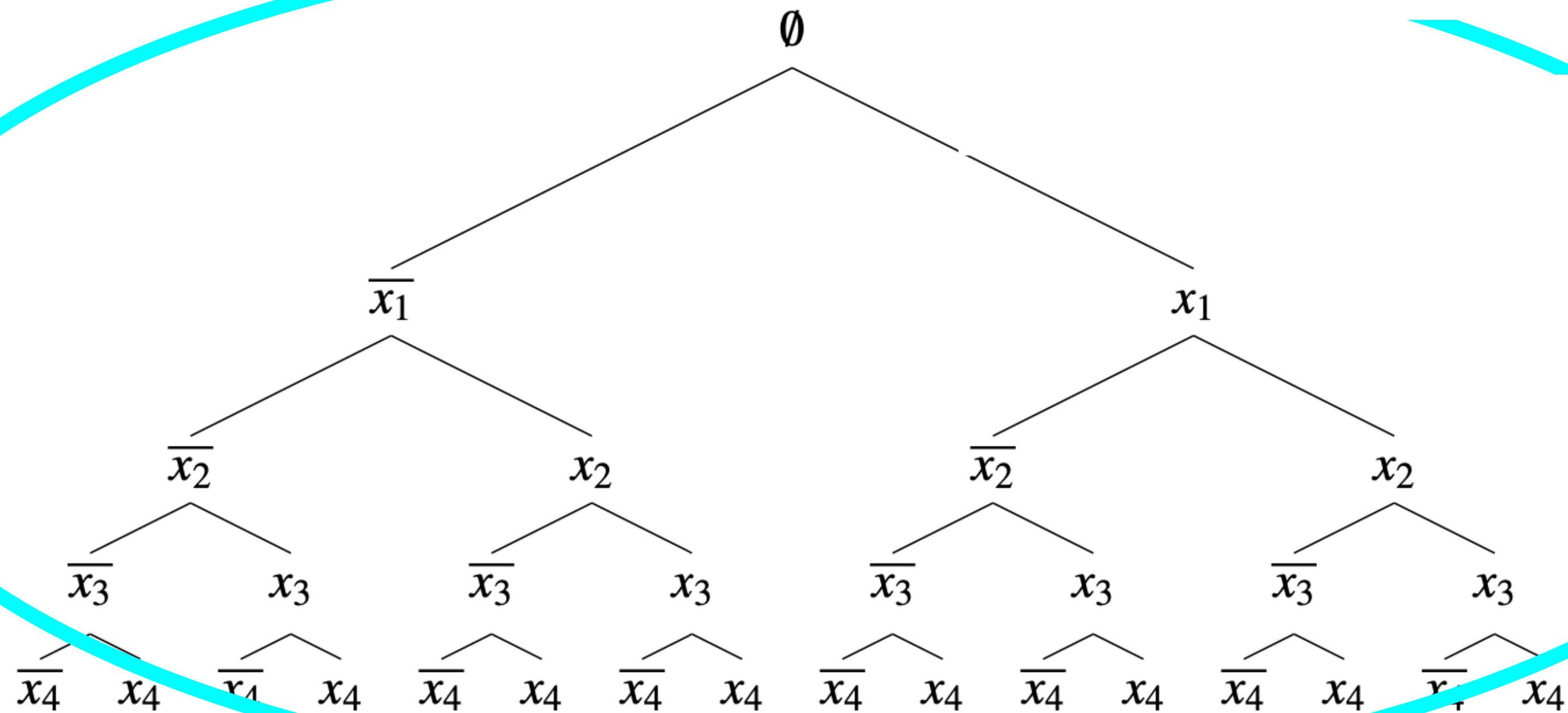
Was sind geeignete Schranken?

2. Geeignete Schranken - Knapsack

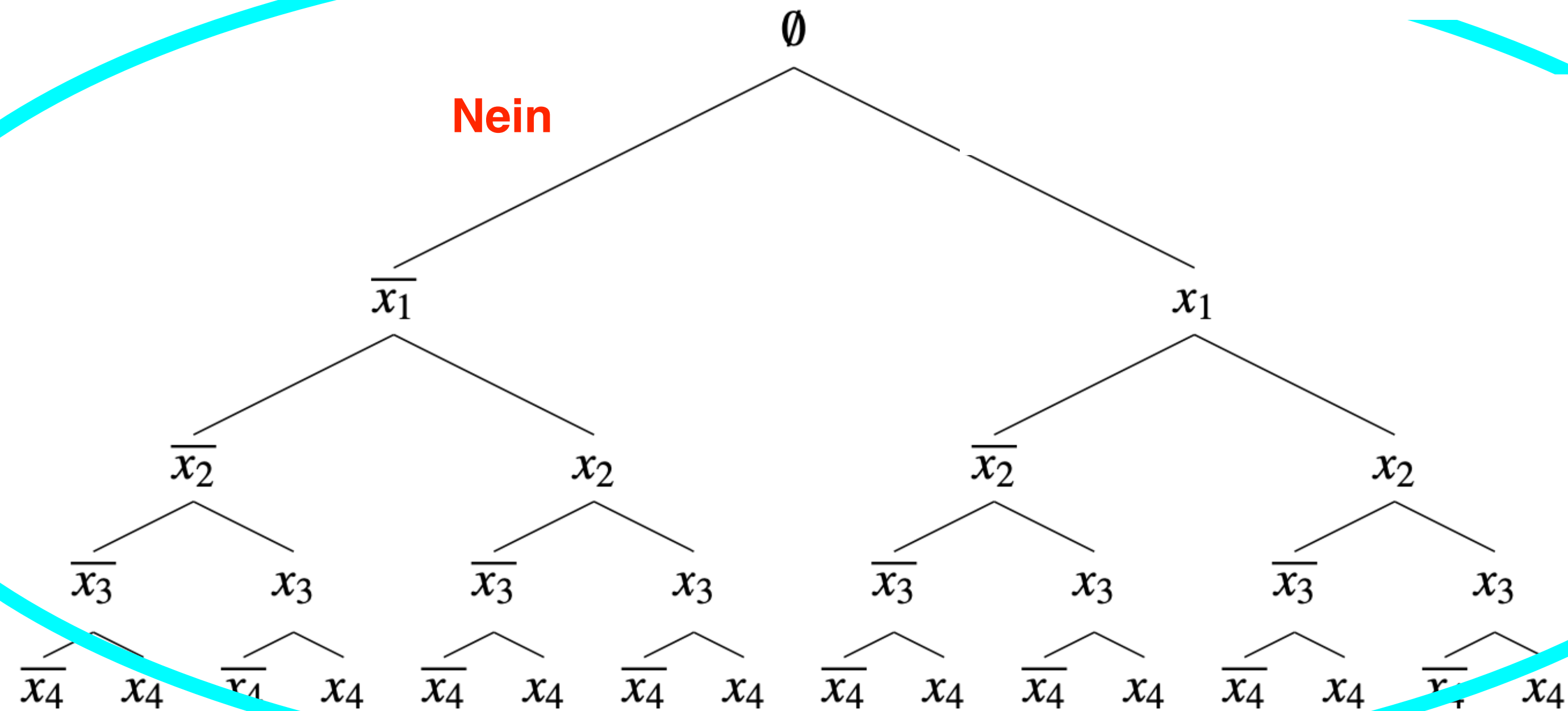
2. Geeignete Schranken - Knapsack



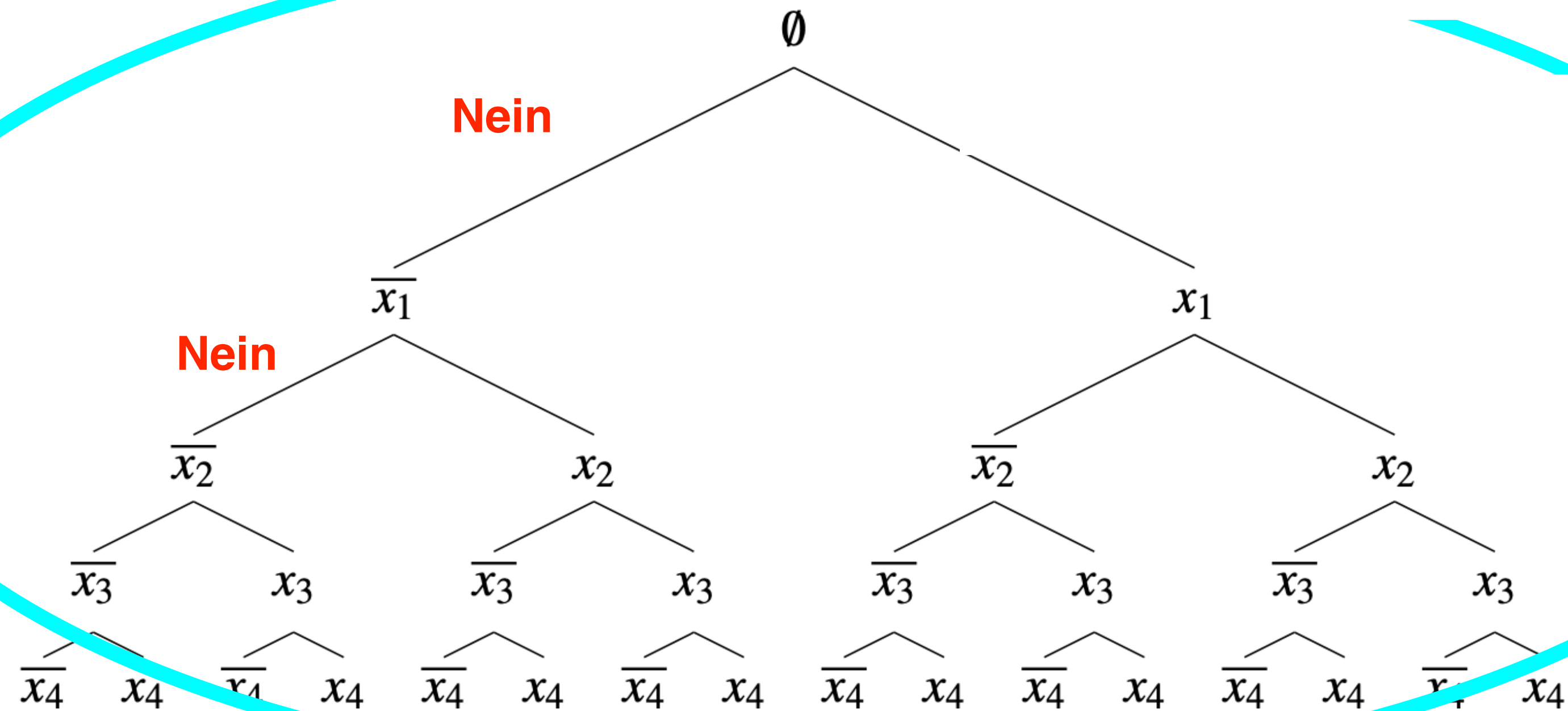
2. Geeignete Schranken - Knapsack



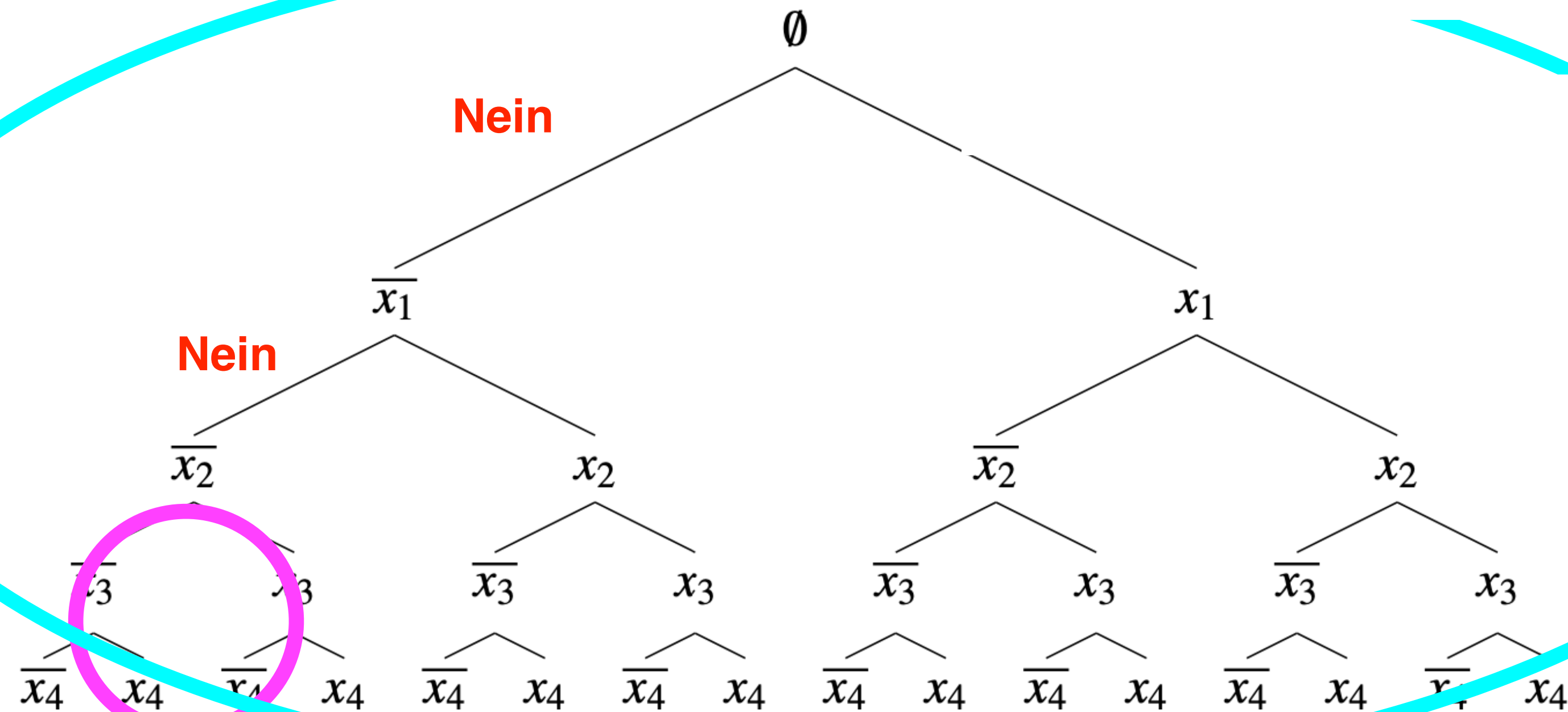
2. Geeignete Schranken - Knapsack



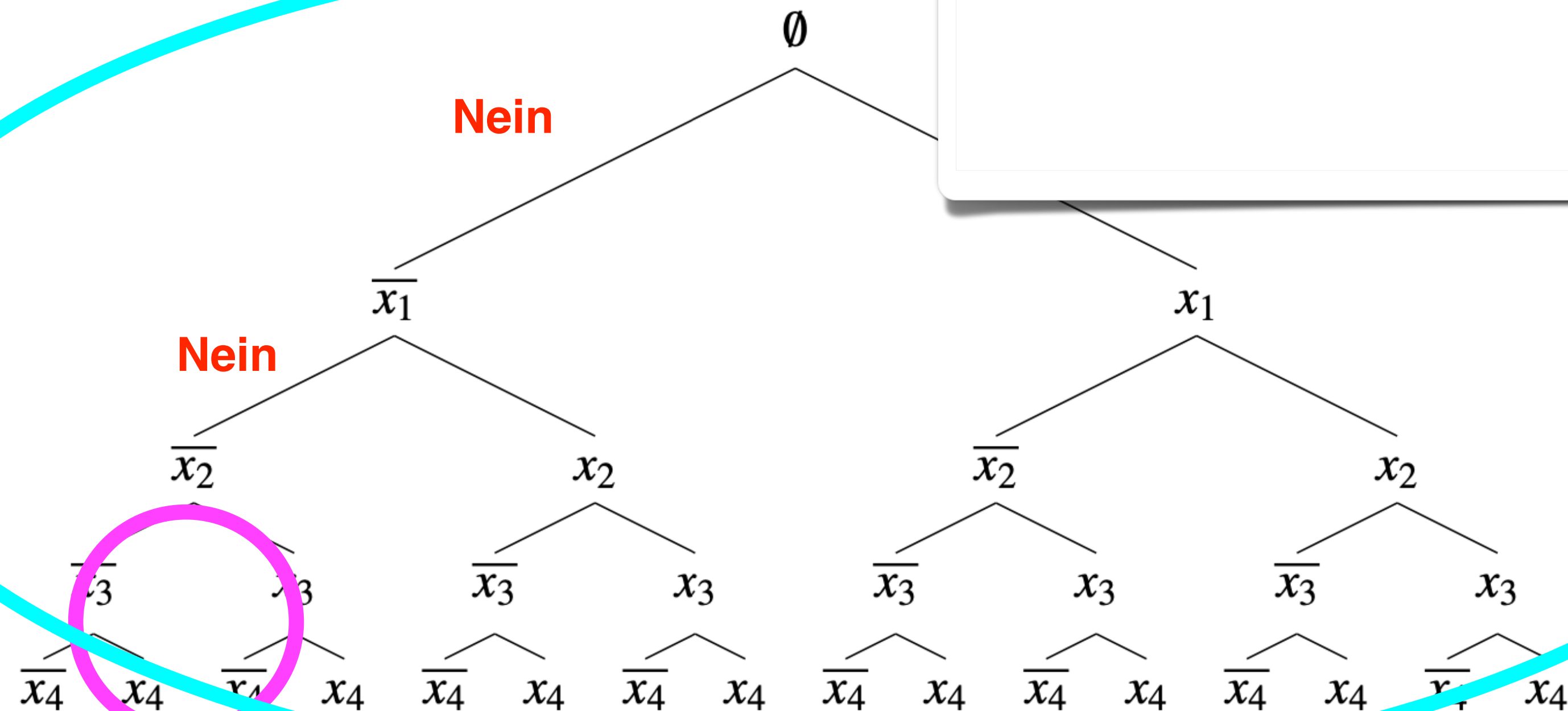
2. Geeignete Schranken - Knapsack



2. Geeignete Schranken - Knapsack

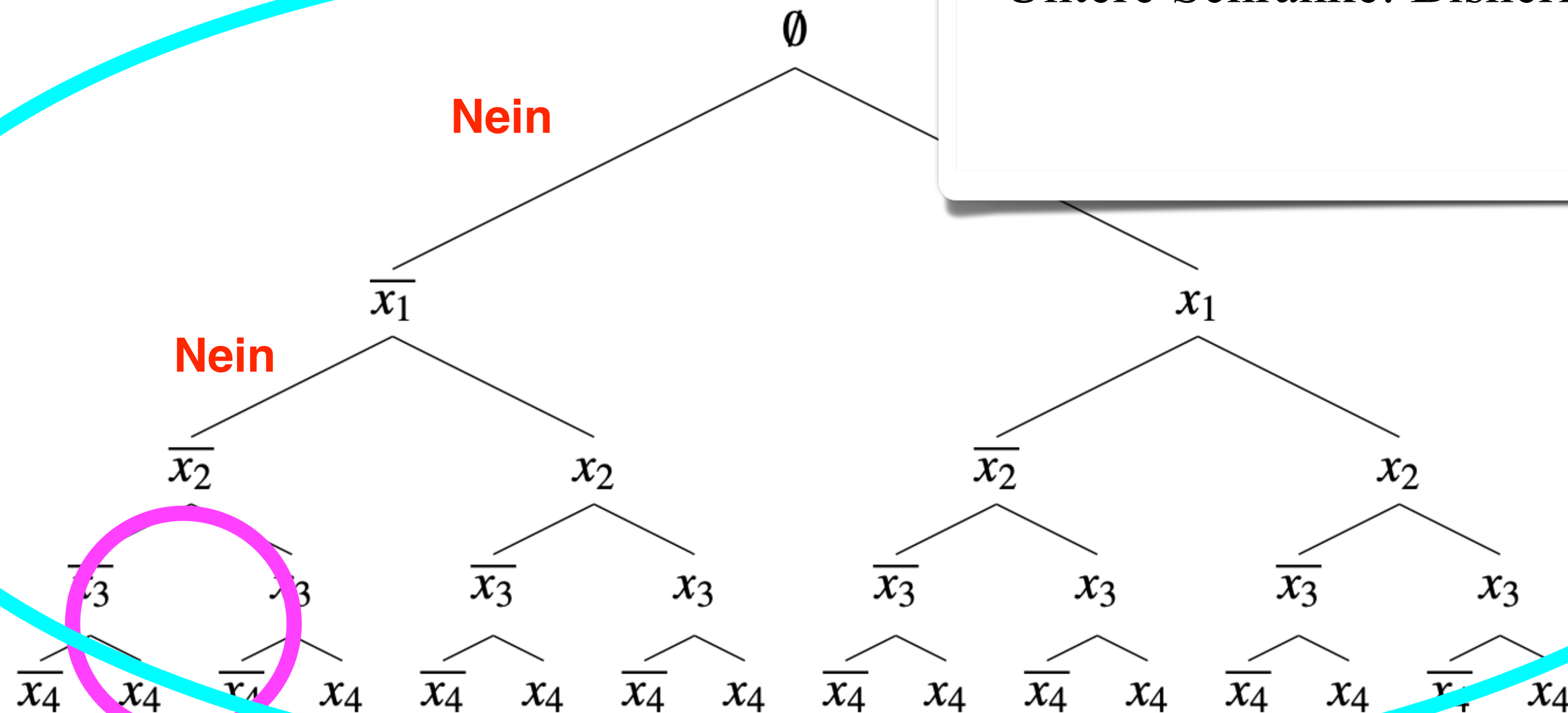


2. Geeignete Schranken - Knapsack



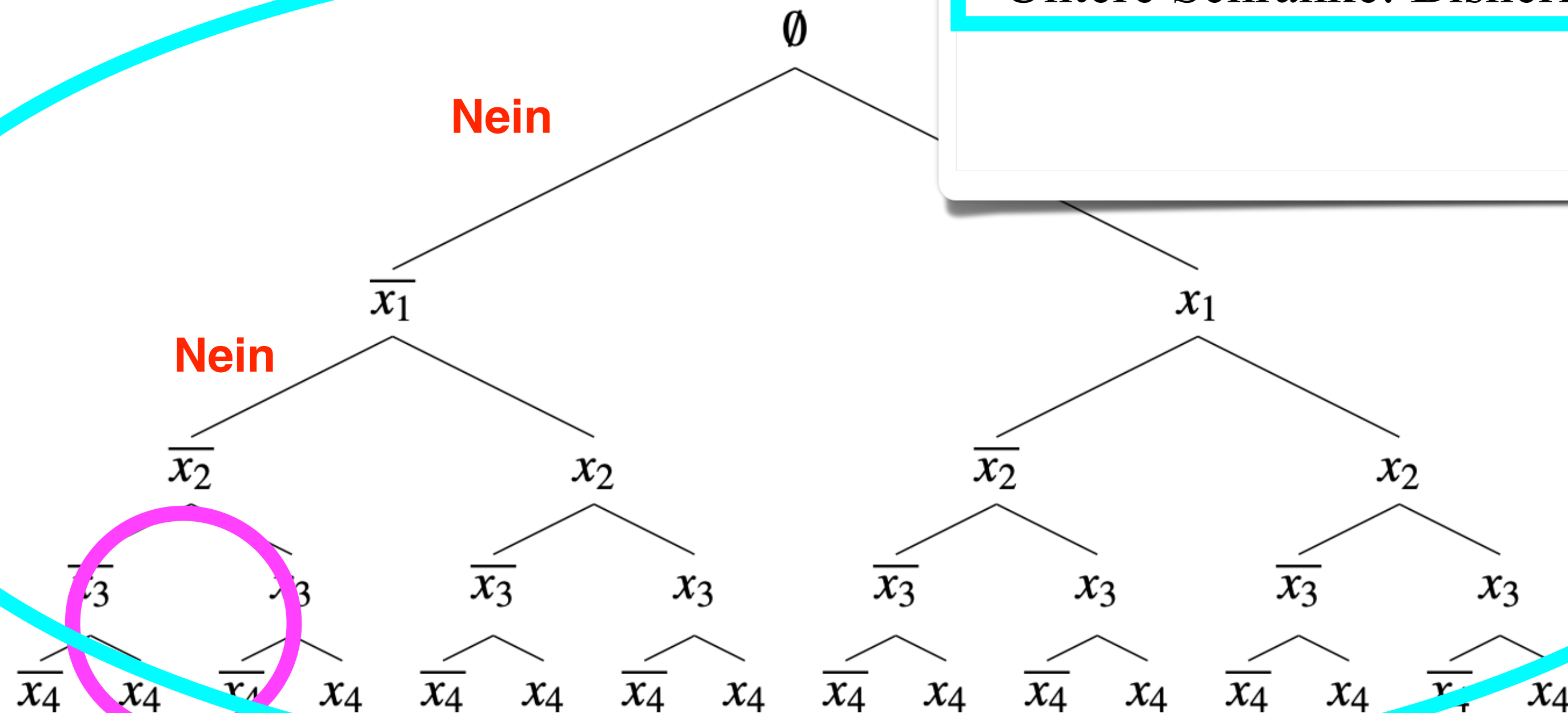
2. Geeignete Schranken - Knapsack

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)



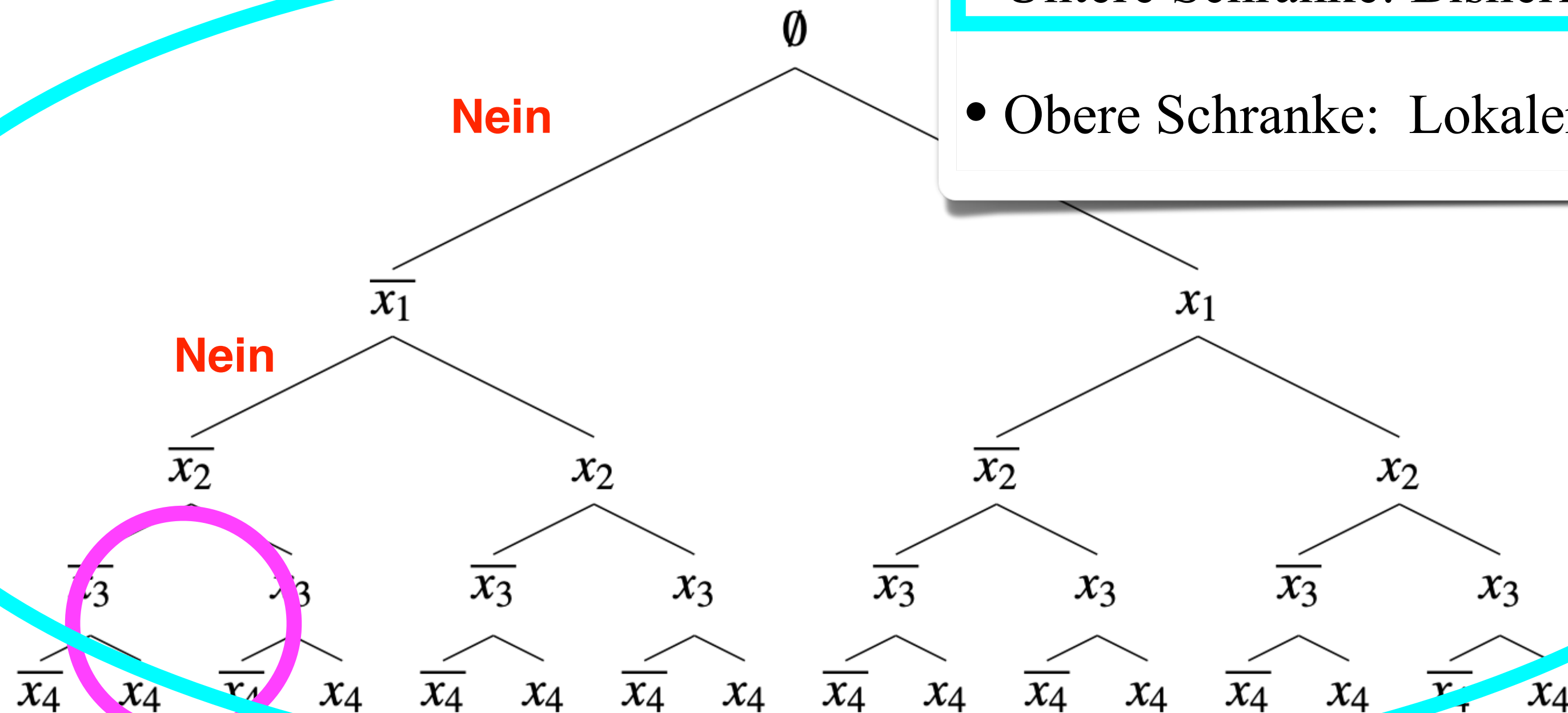
2. Geeignete Schranken - Knapsack

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)



2. Geeignete Schranken - Knapsack

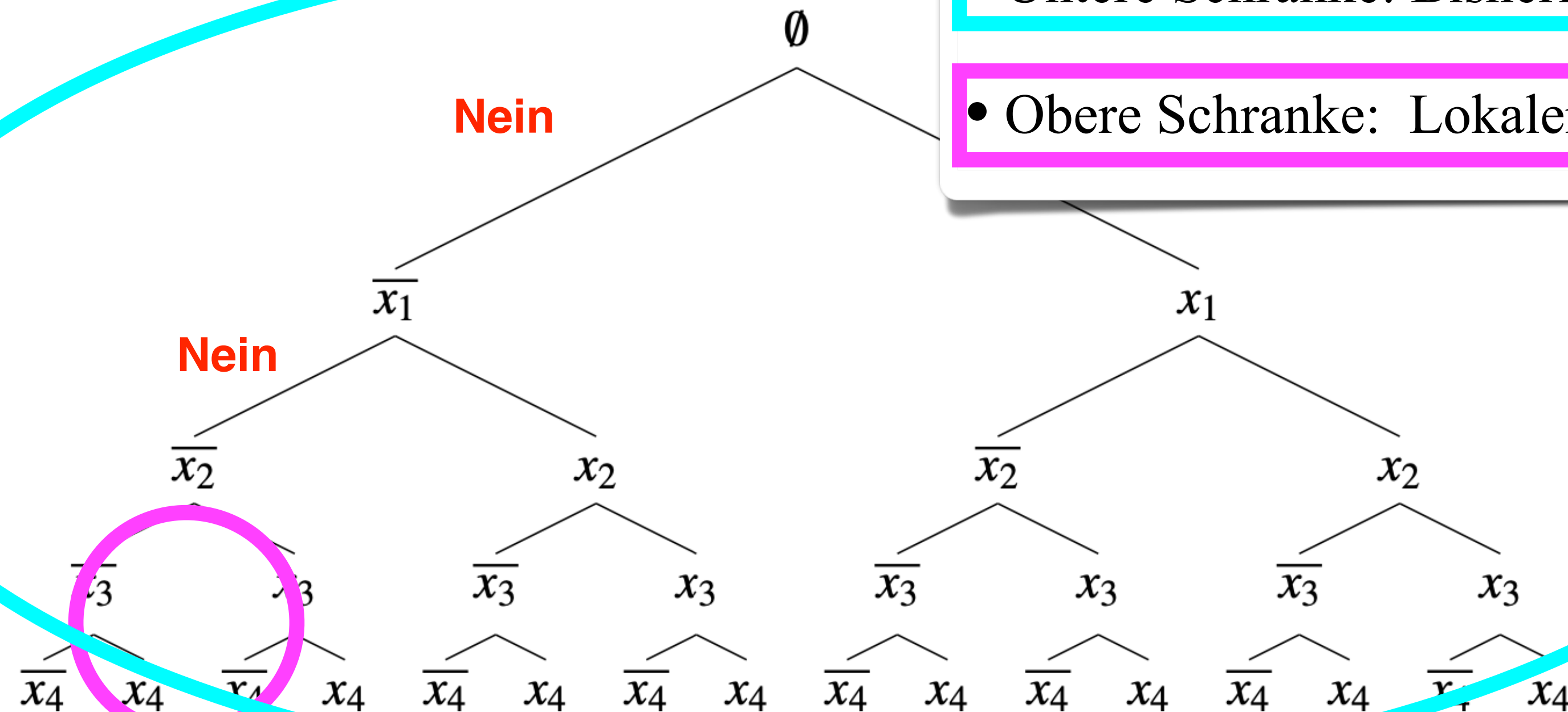
- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)



2. Geeignete Schranken - Knapsack

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)

- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)



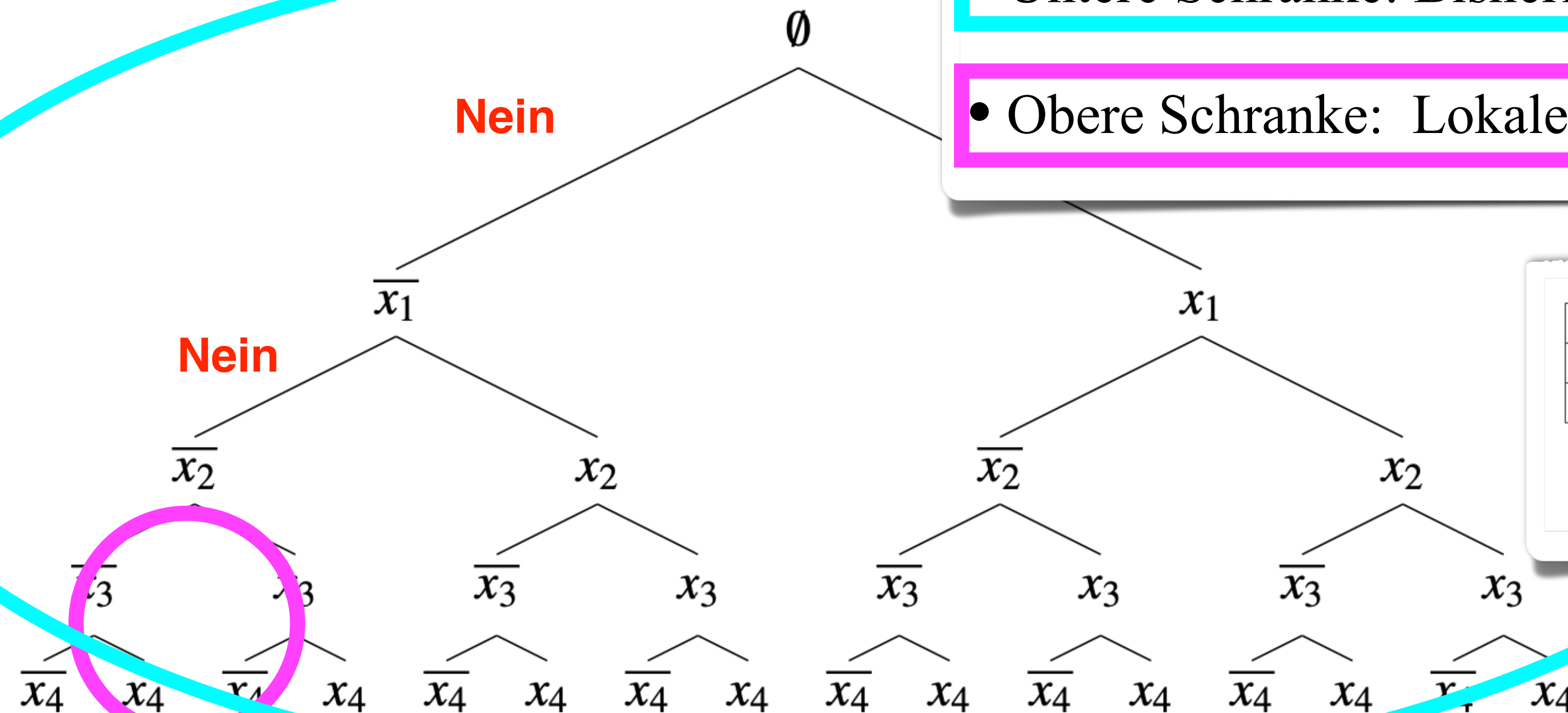
2. Geeignete Schranken - Knapsack

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)

- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^n p_i x_i = 44$$



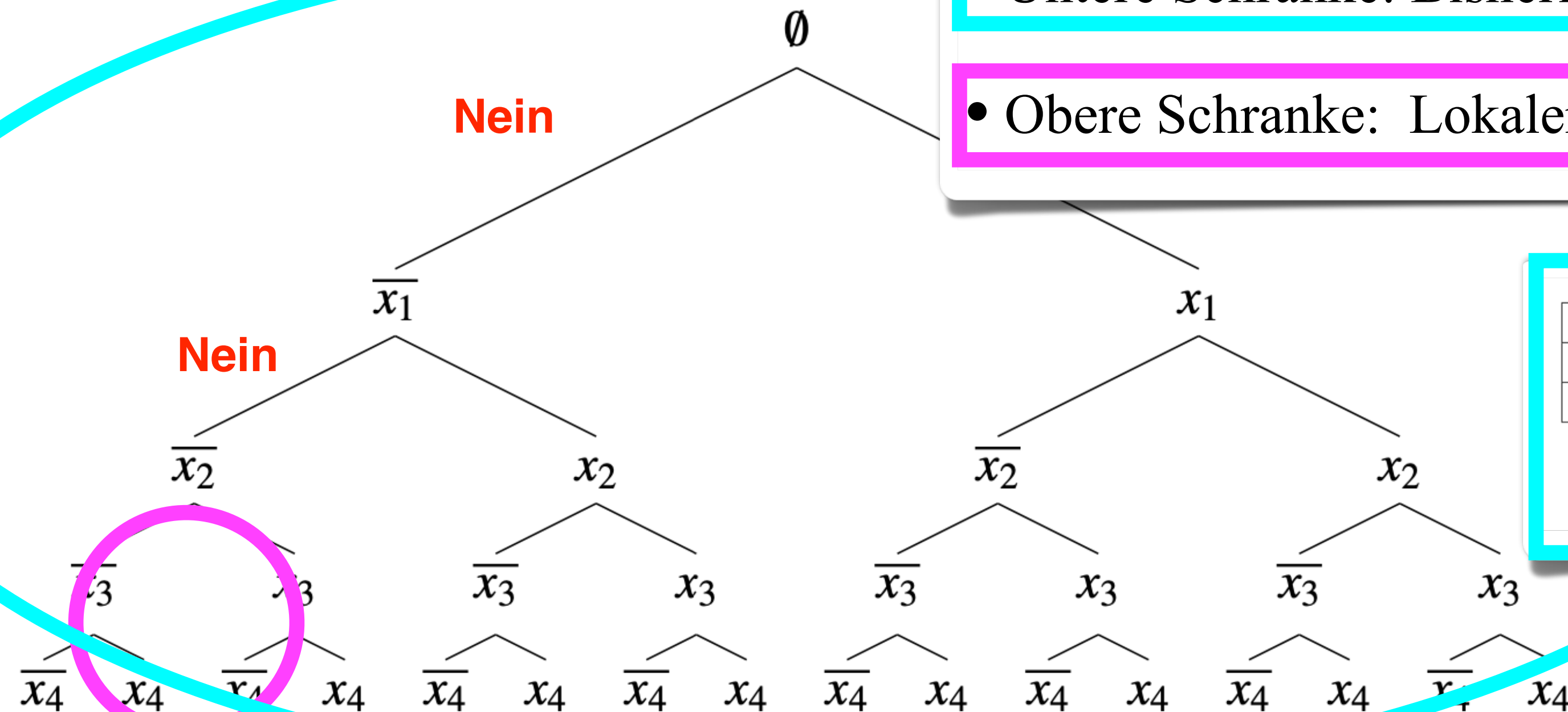
2. Geeignete Schranken - Knapsack

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)

- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

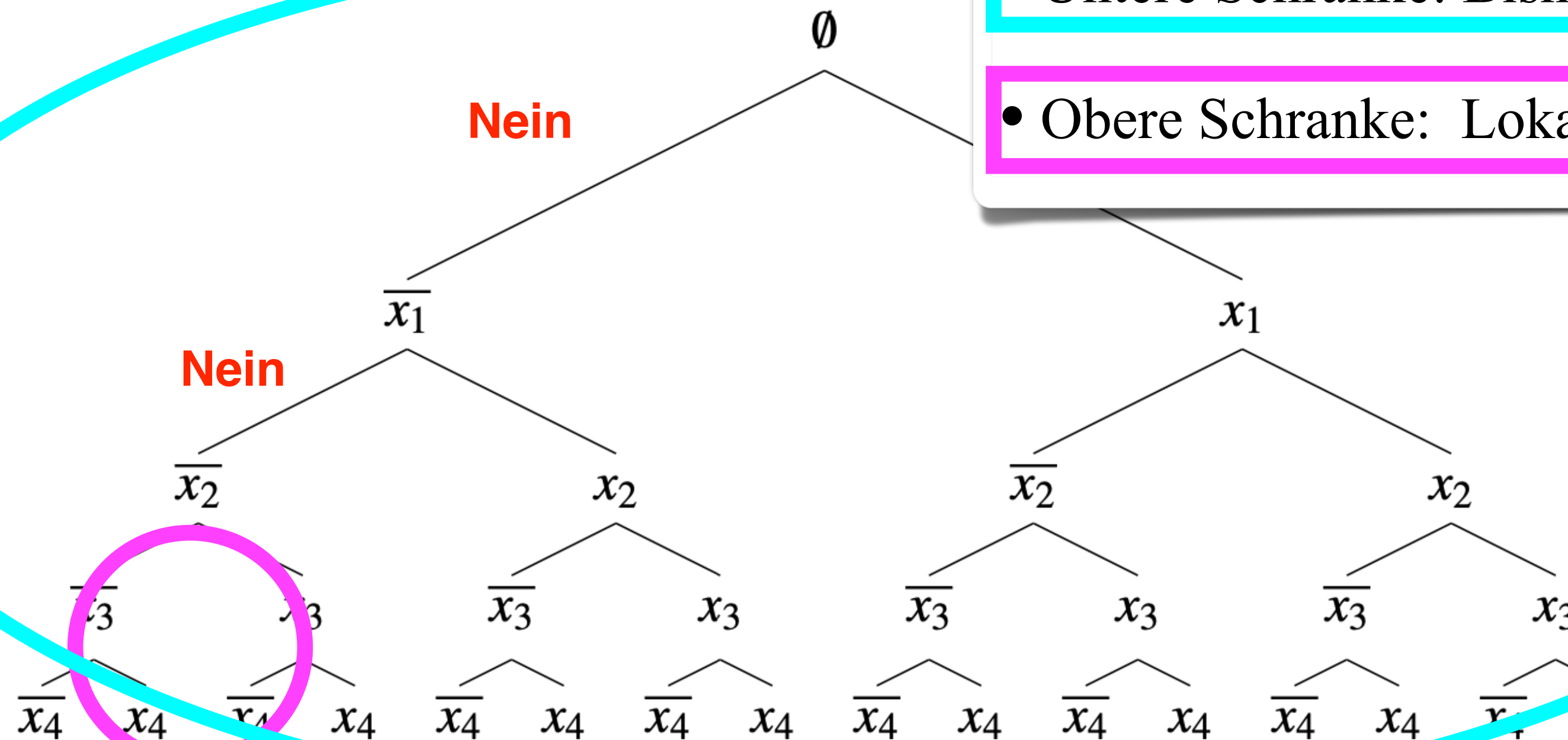
$$\sum_{i=1}^n p_i x_i = 44$$



2. Geeignete Schranken - Knapsack

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)

- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)



i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^n p_i x_i = 44$$

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

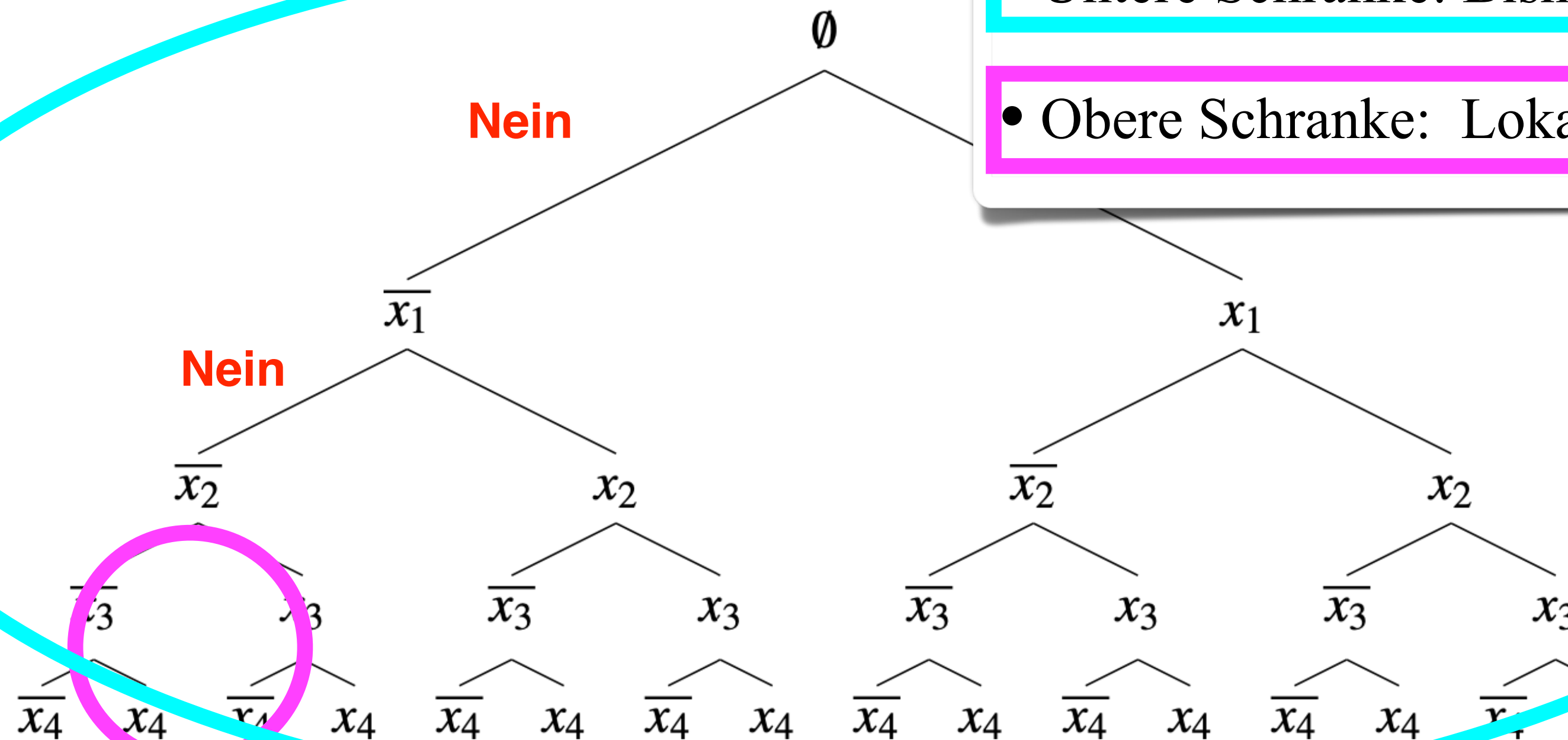
$$\sum_{i=1}^n p_i x_i = 46$$

$$x_{15} = 0,6$$

2. Geeignete Schranken - Knapsack

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)

- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)



i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^n p_i x_i = 44$$

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

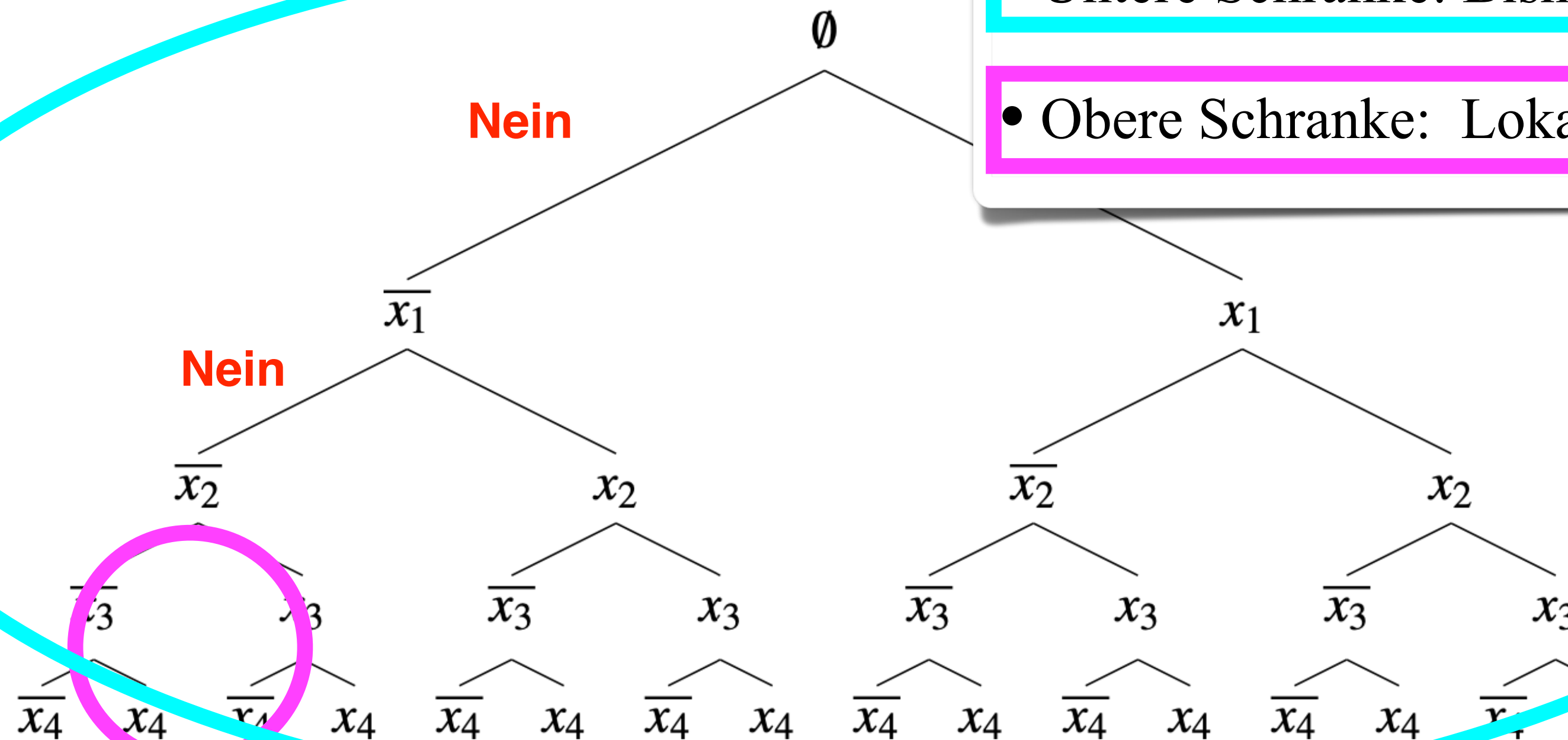
$$\sum_{i=1}^n p_i x_i = 46$$

$$x_{15} = 0,6$$

2. Geeignete Schranken - Knapsack

• Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)

• Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)



i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^n p_i x_i = 44$$

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^n p_i x_i = 46 \quad x_{15} = 0,6$$

Relaxierung: Einfacher durch größere Lösungsmenge

3. Berechnung von Schranken

3. Berechnung von Schranken



3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^n p_i x_i = 44$$

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^n p_i x_i = 44$$

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

<i>i</i>	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
<i>z_i</i>	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
<i>p_i</i>	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^n p_i x_i = 44$

<i>i</i>	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
<i>z_i</i>	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
<i>p_i</i>	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^n p_i x_i = 46$ $x_{15} = 0,6$

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

<i>i</i>	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
<i>z_i</i>	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
<i>p_i</i>	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^n p_i x_i = 44$

<i>i</i>	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
<i>z_i</i>	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
<i>p_i</i>	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^n p_i x_i = 46$ $x_{15} = 0,6$

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

<i>i</i>	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
<i>z_i</i>	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
<i>p_i</i>	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^n p_i x_i = 44$

<i>i</i>	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
<i>z_i</i>	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
<i>p_i</i>	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^n p_i x_i = 46$ $x_{15} = 0,6$

Relaxierung: Schnell und optimal zu lösen durch größere Lösungsmenge

3. Berechnung von Schranken

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

<i>i</i>	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
<i>z_i</i>	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
<i>p_i</i>	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^n p_i x_i = 44$

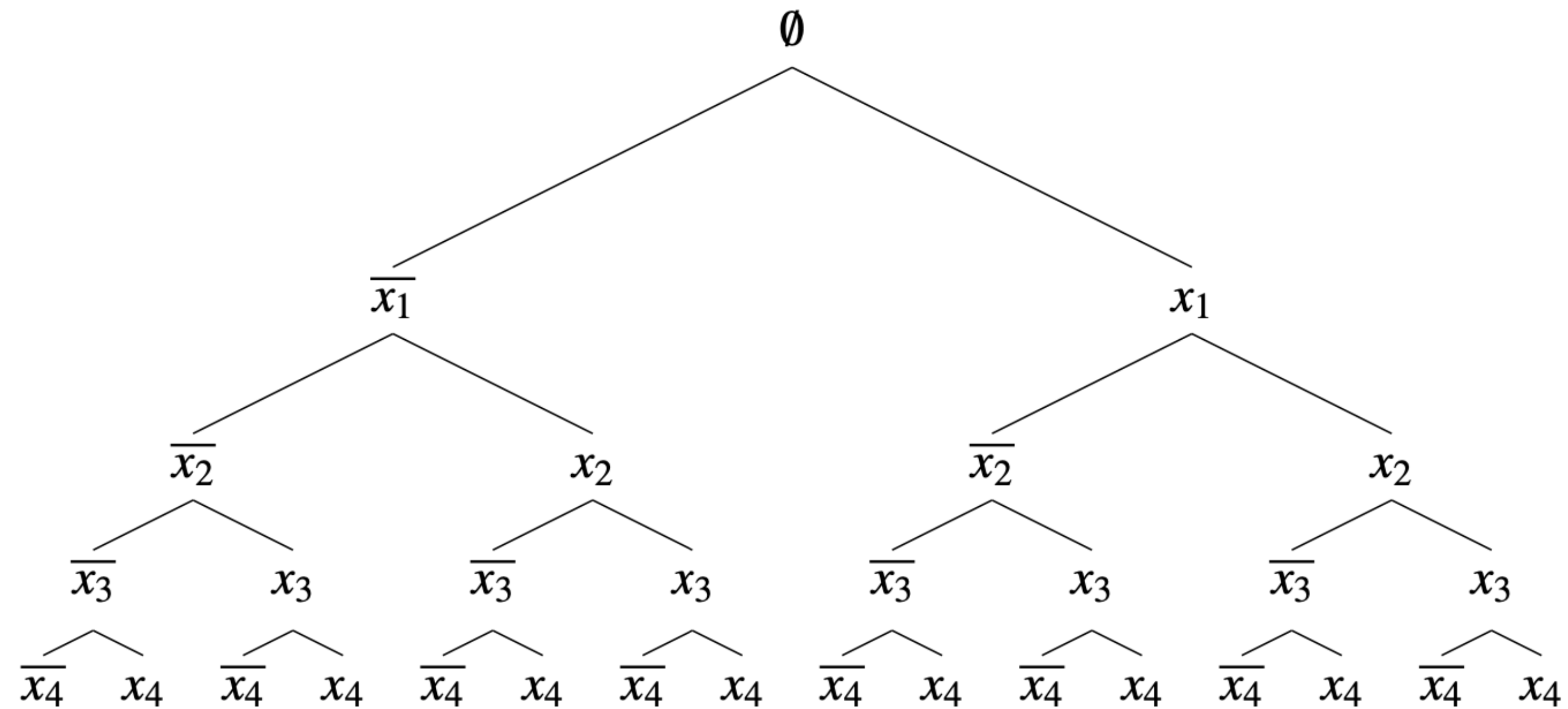
<i>i</i>	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
<i>z_i</i>	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
<i>p_i</i>	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^n p_i x_i = 46$ $x_{15} = 0,6$

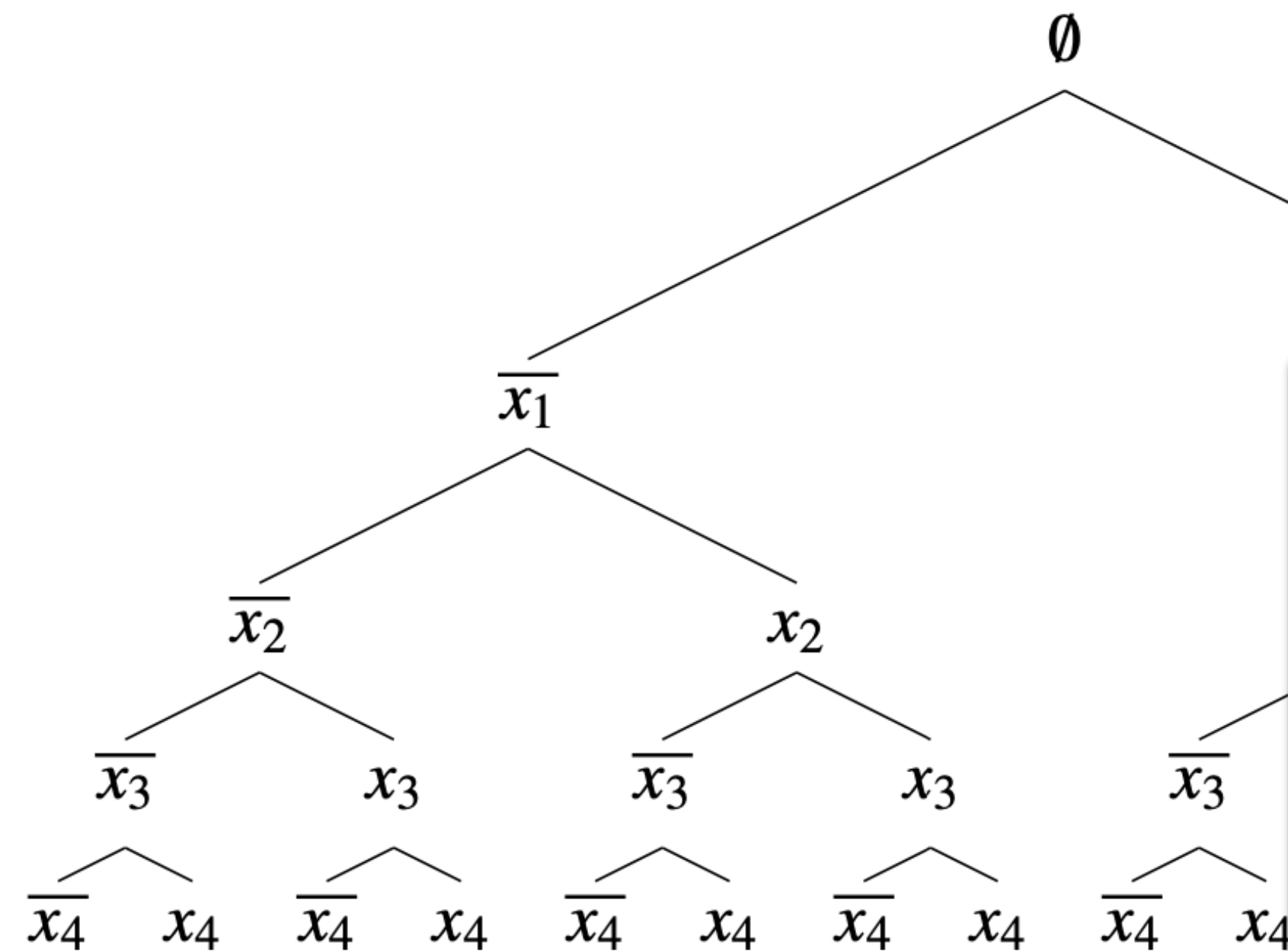
- Heuristik: Schnell zu lösen, aber nicht notwendig optimal
- Relaxierung: Schnell und optimal zu lösen durch größere Lösungsmenge

3. Berechnung von Schranken II

3. Berechnung von Schranken II

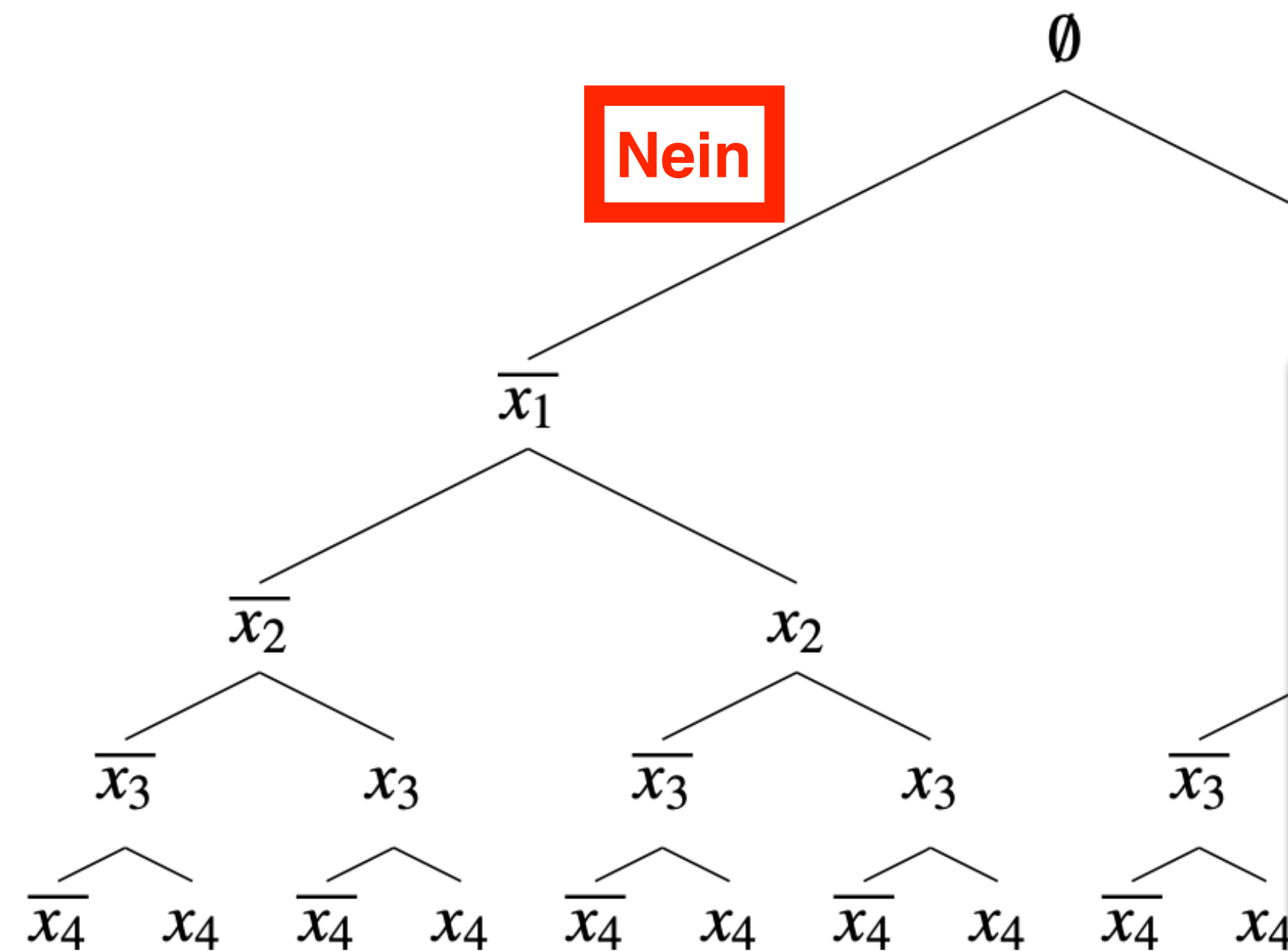


3. Berechnung von Schranken II



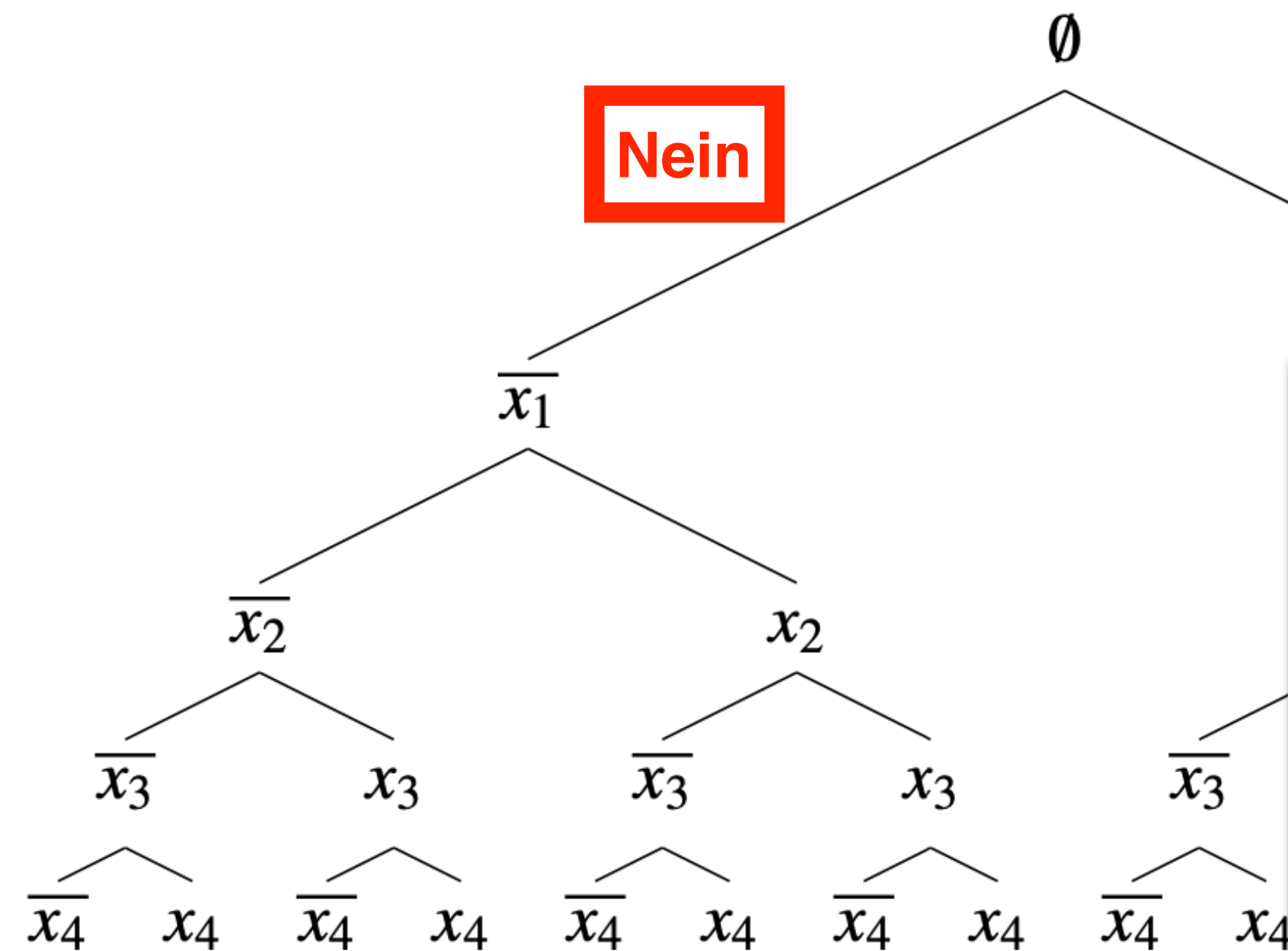
i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3. Berechnung von Schranken II



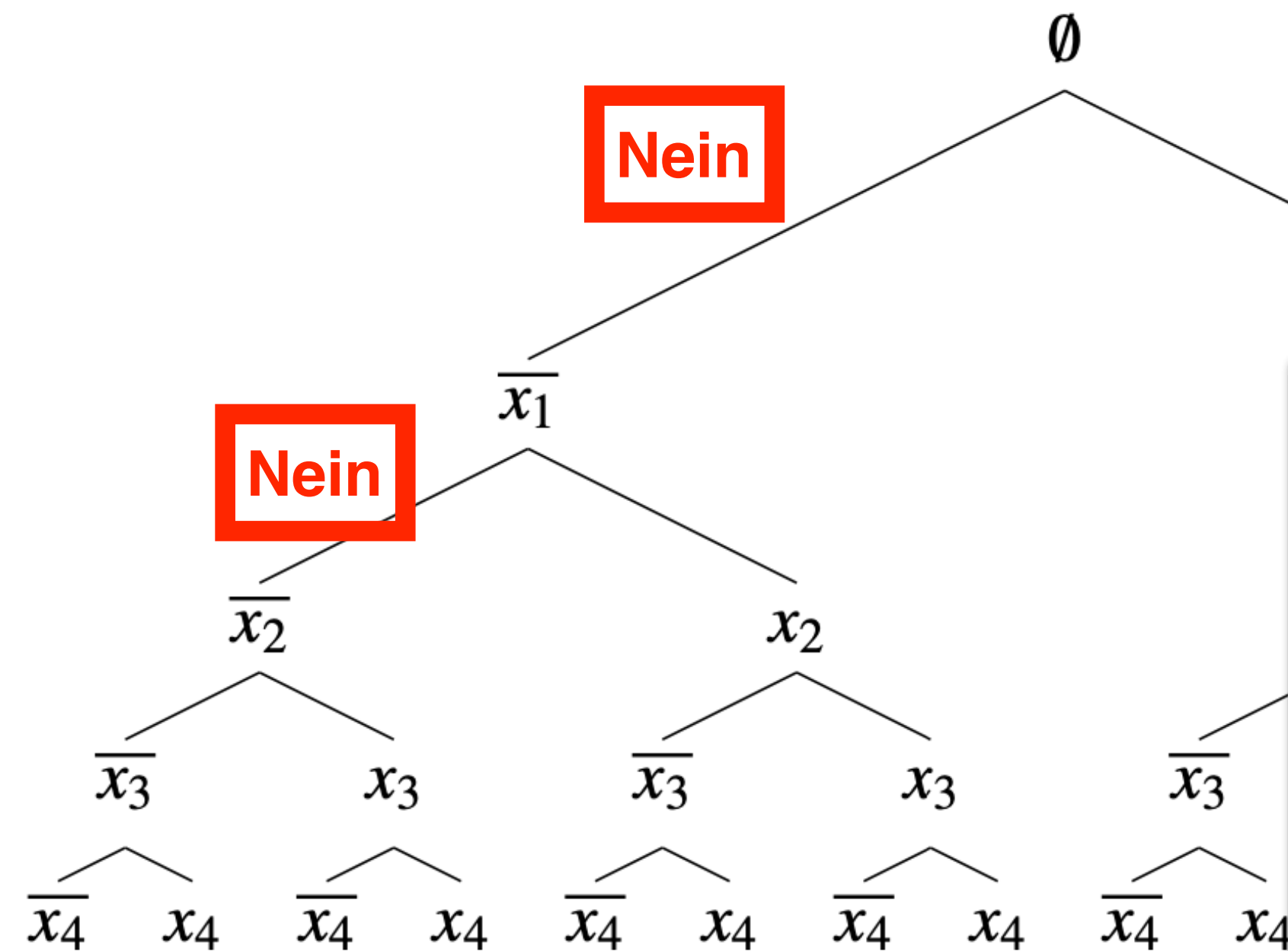
i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3. Berechnung von Schranken II



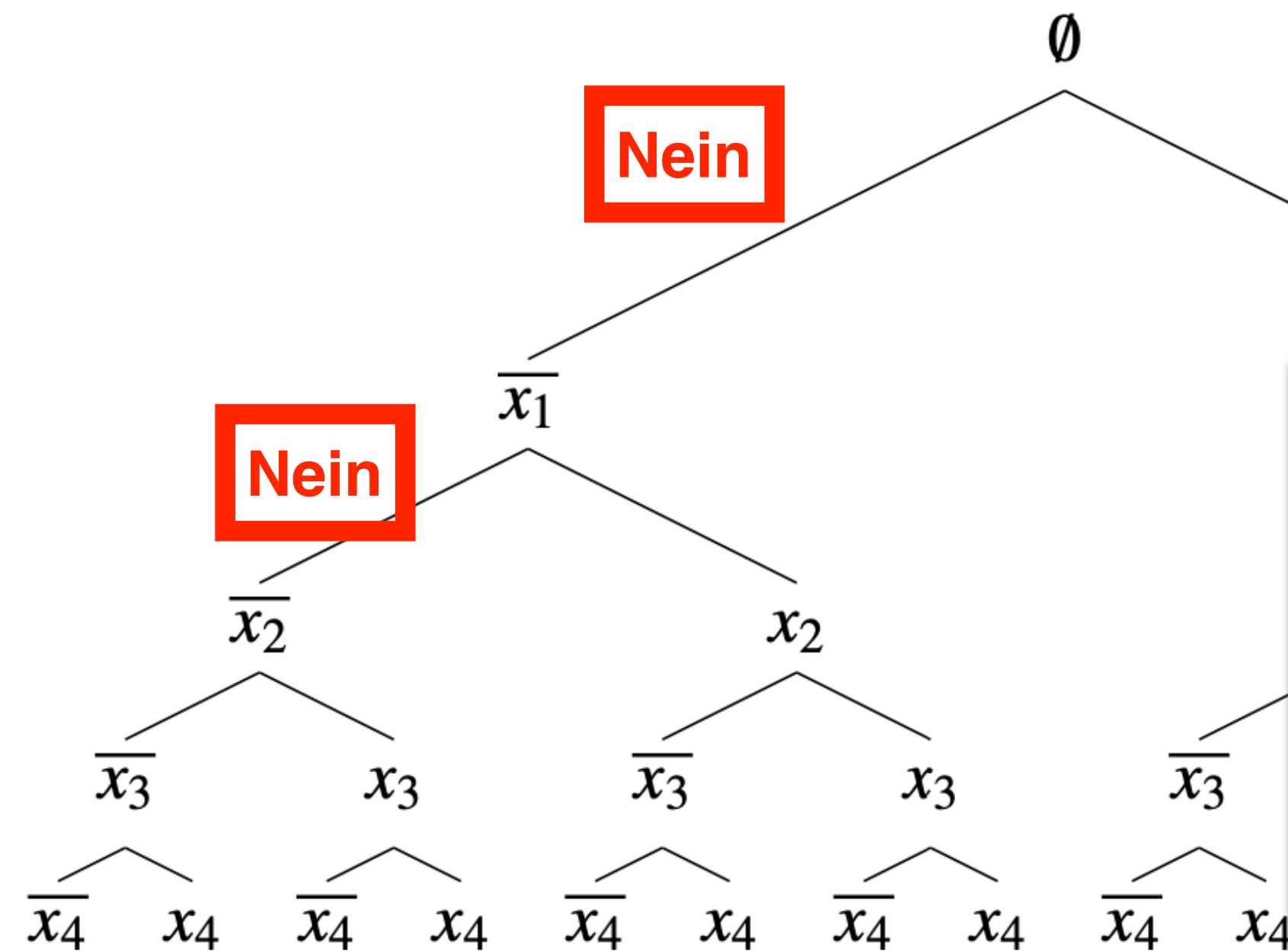
i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3. Berechnung von Schranken II



i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

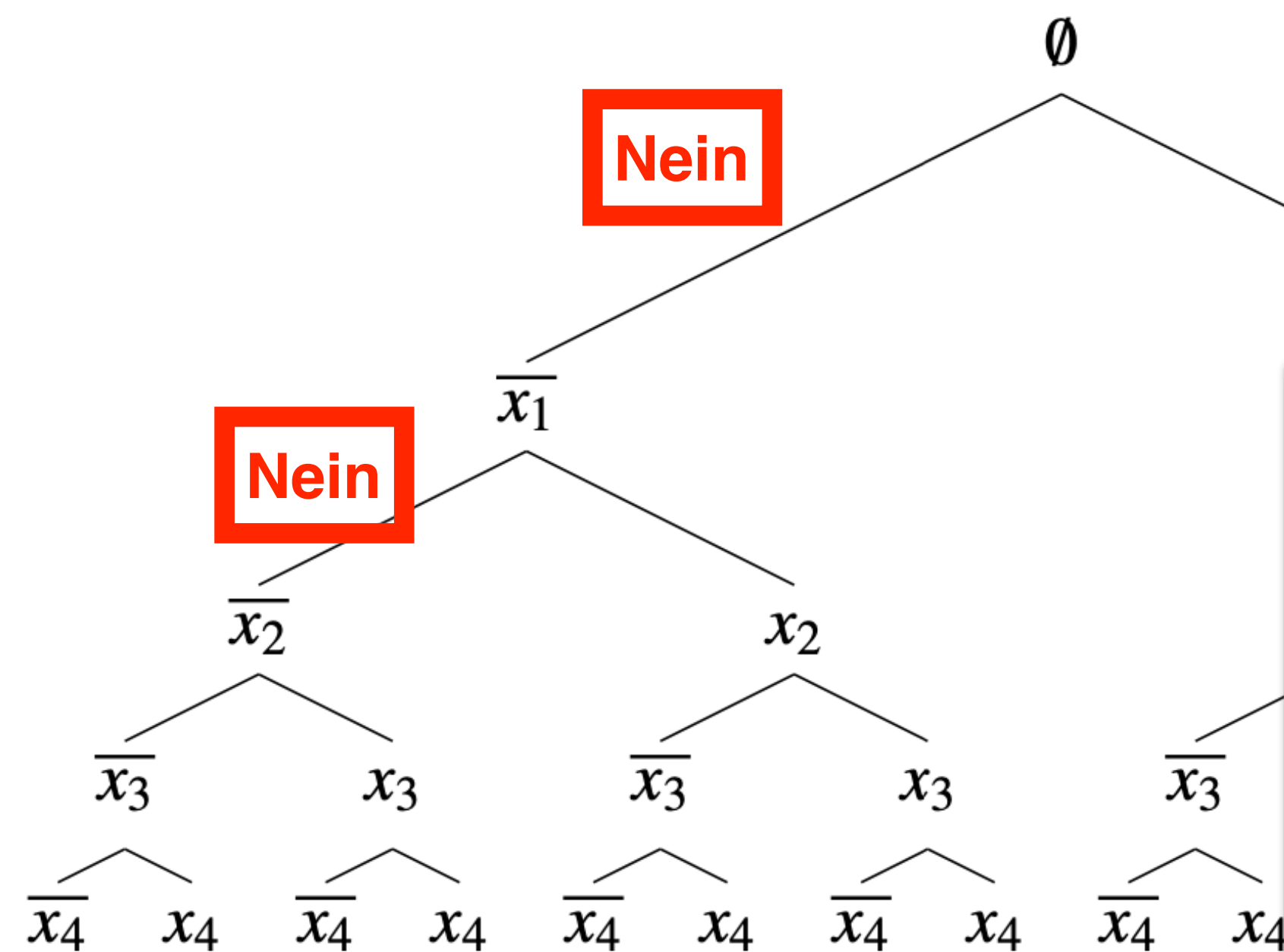
3. Berechnung von Schranken II



i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3. Berechnung von Schranken II

S : positiv fixiert ($x_i = 1$)
 $\bar{S}$: negativ fixiert ($x_i = 0$)

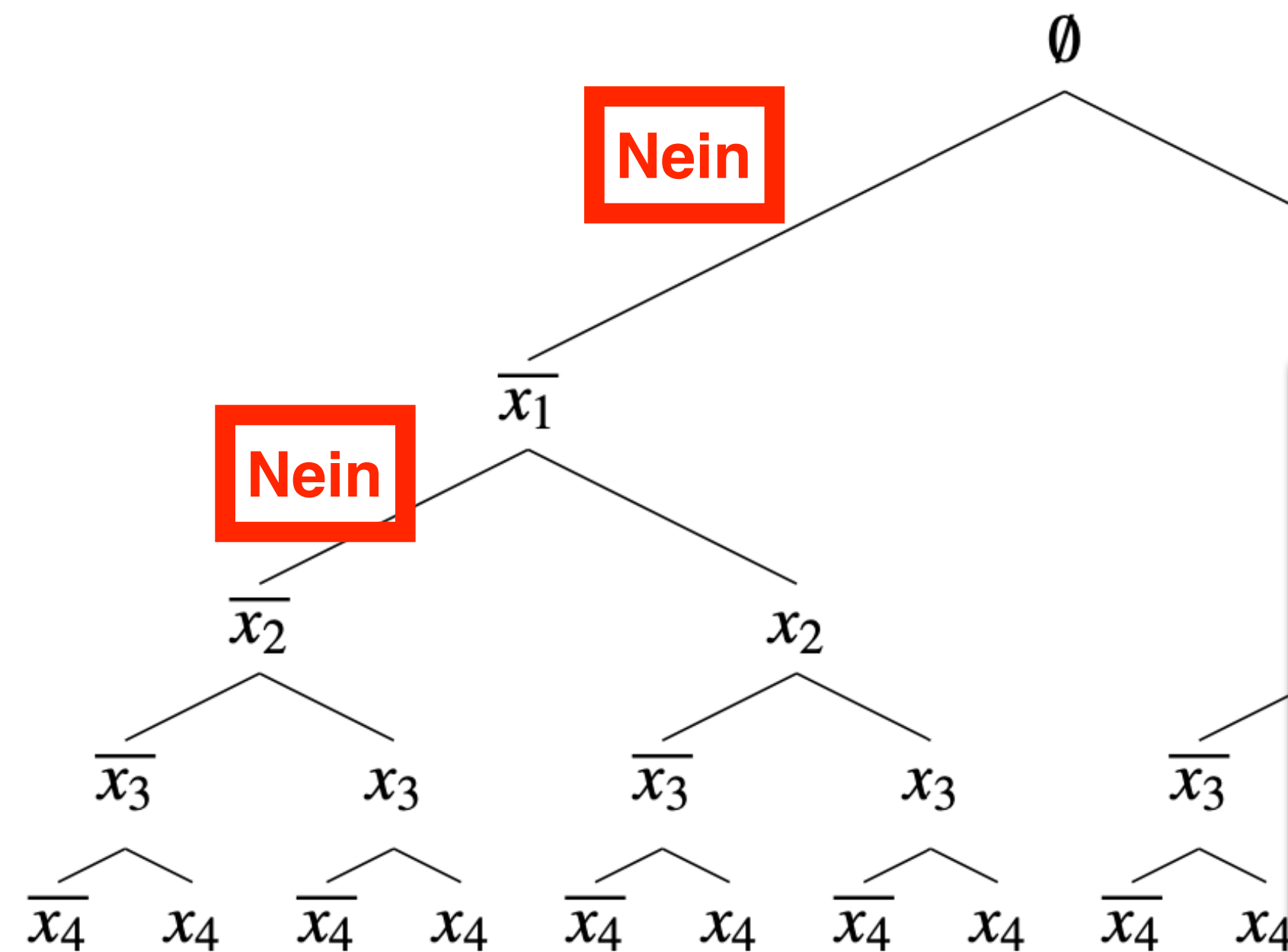


i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3. Berechnung von Schranken II

S: positiv fixiert ($x_i = 1$)

S: negativ fixiert ($x_i = 0$)



i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

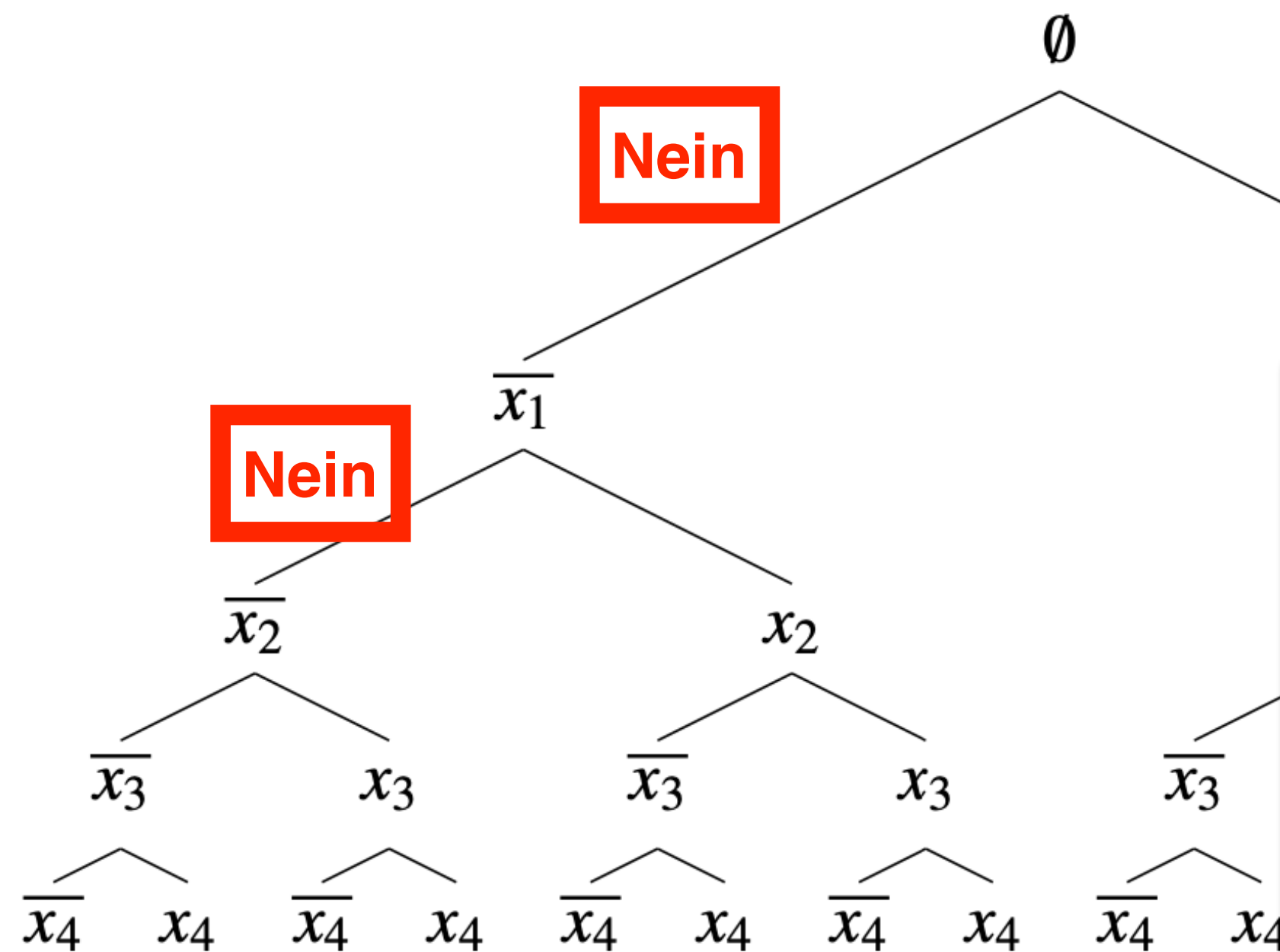
3. Berechnung von Schranken II

S: positiv fixiert ($x_i = 1$)

S: negativ fixiert ($x_i = 0$)

Nein

Nein

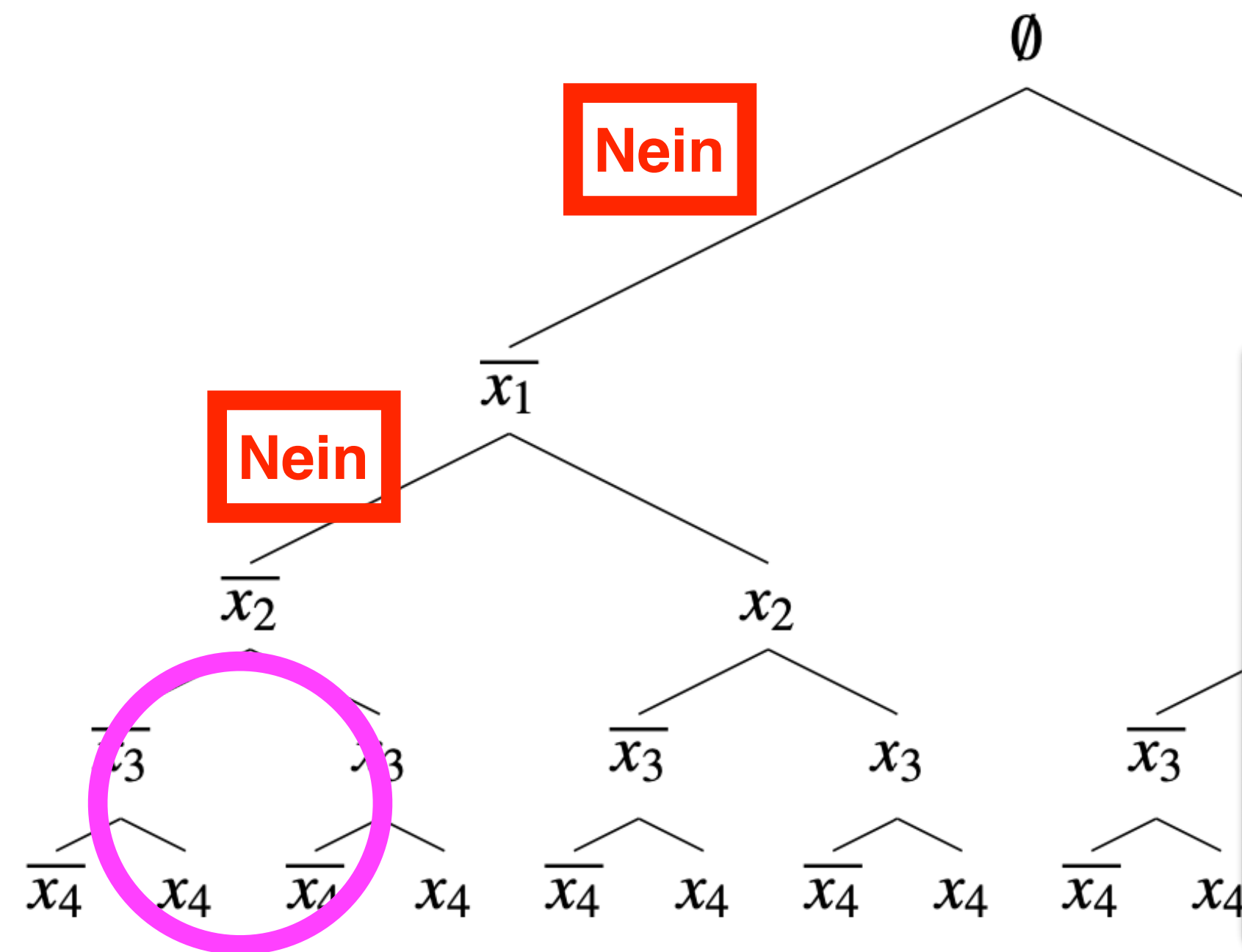


i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3. Berechnung von Schranken II

S: positiv fixiert ($x_i = 1$)

S: negativ fixiert ($x_i = 0$)

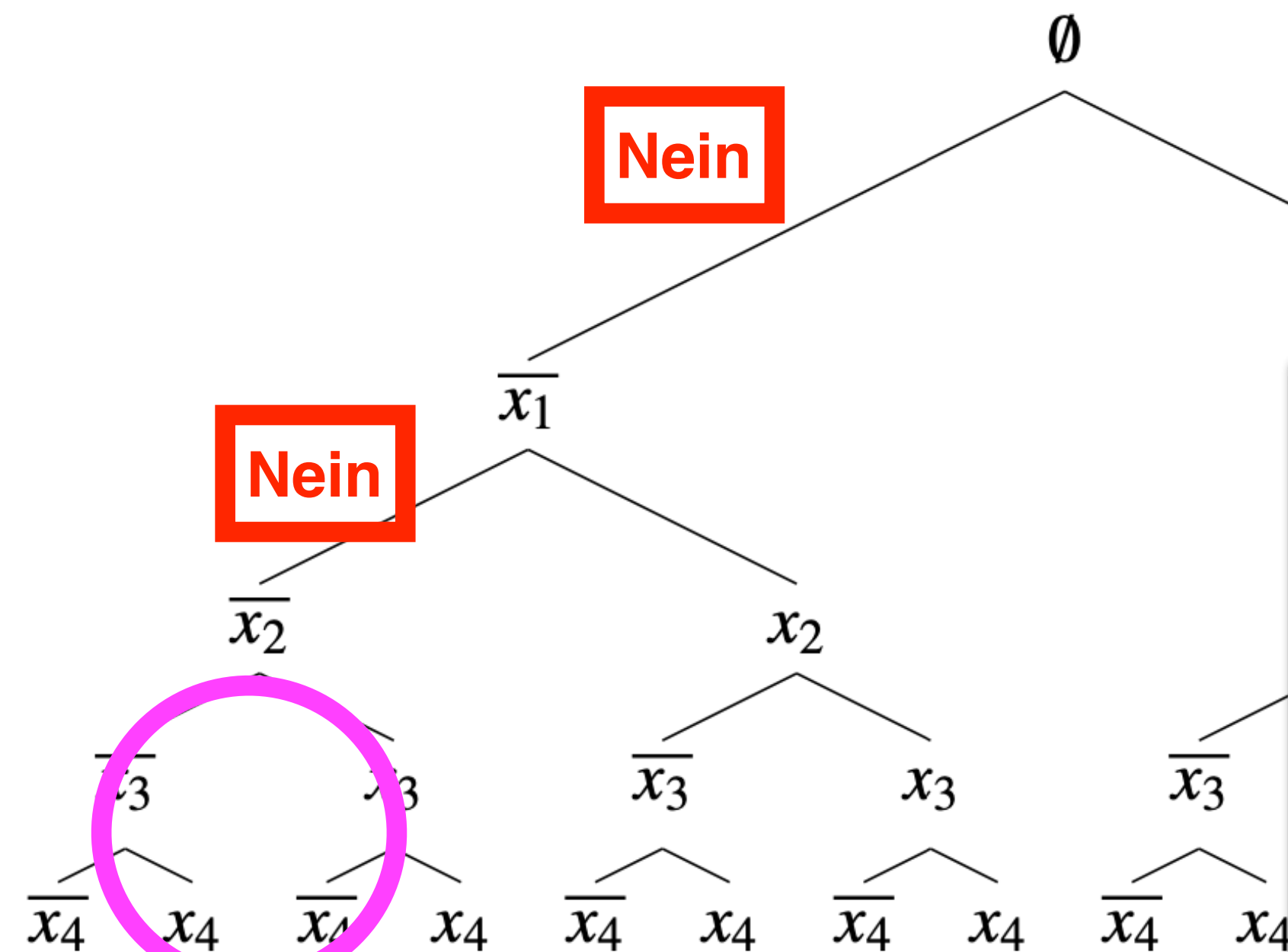


i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3. Berechnung von Schranken II

S: positiv fixiert ($x_i = 1$)

S: negativ fixiert ($x_i = 0$)



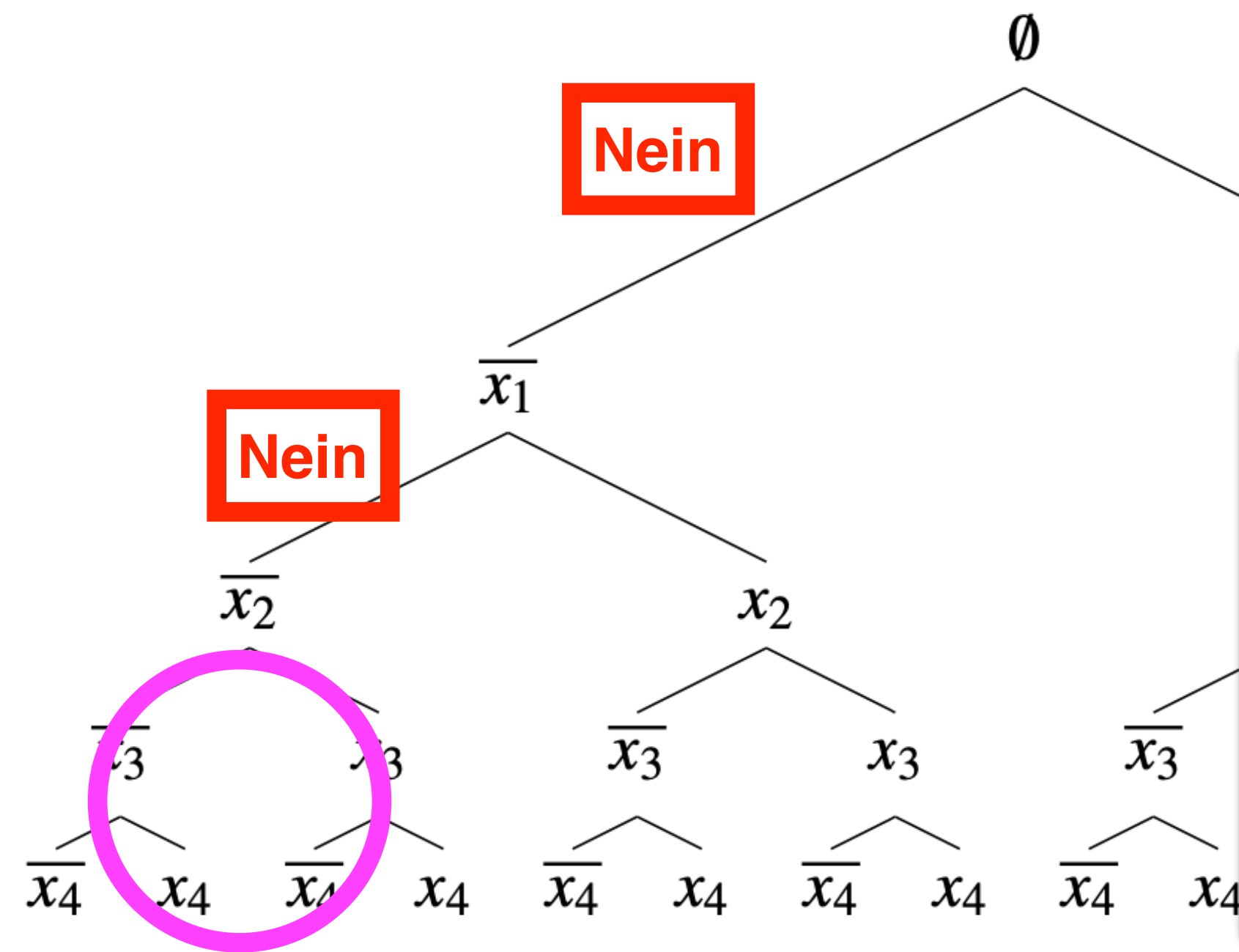
i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3. Berechnung von Schranken II

S: positiv fixiert ($x_i = 1$)

S: negativ fixiert ($x_i = 0$)

Bisherige Entscheidungen fixieren Teilmenge!



i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3.3 Durchführung

Beispiel: Knapsack

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i
z_i
p_i

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1
z_i	2
p_i	6

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2
z_i	2	3
p_i	6	5

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2	3
z_i	2	3	6
p_i	6	5	8

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2	3	4
z_i	2	3	6	7
p_i	6	5	8	9

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2	3	4	5
z_i	2	3	6	7	5
p_i	6	5	8	9	6

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2	3	4	5	6
z_i	2	3	6	7	5	9
p_i	6	5	8	9	6	7

Beispiel: Knapsack

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

Untere Schranke: Greedy (ganzzahlig)

Untere Schranke: Greedy (ganzzahlig)

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

Untere Schranke: Greedy (ganzzahlig)

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$Z=9$

Untere Schranke: Greedy (ganzzahlig)

$S=\emptyset$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$Z=9$

Untere Schranke: Greedy (ganzzahlig)

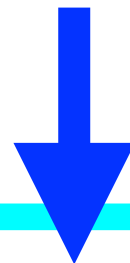
$$S=\emptyset$$

$$\bar{S}=\emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z=9$$

Untere Schranke: Greedy (ganzzahlig)



$$S = \emptyset$$

$$\bar{S} = \emptyset$$

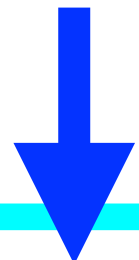
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



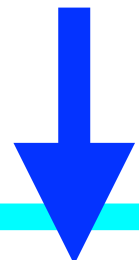
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

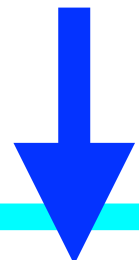
$$Z = 9$$

$$i=1 : \sum_{i \in S} z_i = 2, \sum_{i \in S} p_i = 6$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

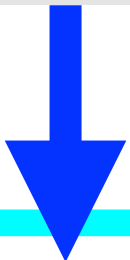
$$Z = 9$$

$$i=1 : \sum_{i \in S} z_i = 2, \sum_{i \in S} p_i = 6$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

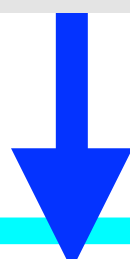
$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$S = \emptyset$

$\bar{S} = \emptyset$

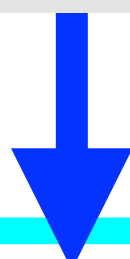
<i>i</i>	1	2	3	4	5	6	7
<i>z_i</i>	2	3	6	7	5	9	4
<i>p_i</i>	6	5	8	9	6	7	3

$Z = 9$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

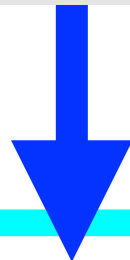


i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z=9$$

$$\mathbf{i=3} : \sum_{i \in S} z_i = 11 > 9, \mathbf{X}$$

Untere Schranke: Greedy (ganzzahlig)



$S = \emptyset$

$\bar{S} = \emptyset$

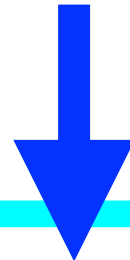
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$Z = 9$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



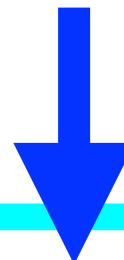
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



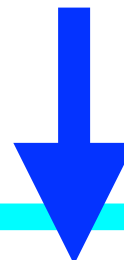
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

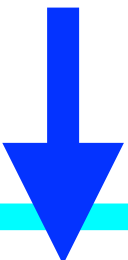


i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=4 : \sum_{i \in S} z_i = 12 > 9, X$$

Untere Schranke: Greedy (ganzzahlig)



$S = \emptyset$

$\bar{S} = \emptyset$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$Z = 9$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$S = \emptyset$

$\bar{S} = \emptyset$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$Z = 9$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=5 : \sum_{i \in S} z_i = 10 > 9, X$$

Untere Schranke: Greedy (ganzzahlig)

$S = \emptyset$

$\bar{S} = \emptyset$

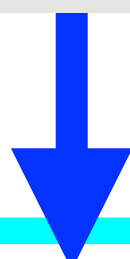
<i>i</i>	1	2	3	4	5	6	7
<i>z_i</i>	2	3	6	7	5	9	4
<i>p_i</i>	6	5	8	9	6	7	3

$Z = 9$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$S = \emptyset$

$\bar{S} = \emptyset$

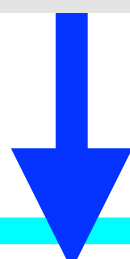
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$Z = 9$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

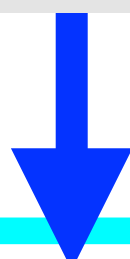
$$Z = 9$$

$$i=6 : \sum_{i \in S} z_i = 14 > 9, X$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Untere Schranke: Greedy (ganzzahlig)

$S = \emptyset$

$\bar{S} = \emptyset$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$Z = 9$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=7 : \sum_{i \in S} z_i = 9, \sum_{i \in S} p_i = 14$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=7 : \sum_{i \in S} z_i = 9, \sum_{i \in S} p_i = 14$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=7 : \sum_{i \in S} z_i = 9, \sum_{i \in S} p_i = 14$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=7 : \sum_{i \in S} z_i = 9, \sum_{i \in S} p_i = 14$$

Untere Schranke: Greedy (ganzzahlig)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=7 : \sum_{i \in S} z_i = 9, \sum_{i \in S} p_i = 14$$

$$LB = 14$$

Obere Schranke: Greedy (fraktional)

Obere Schranke: Greedy (fraktional)

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

Obere Schranke: Greedy (fraktional)

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z=9$$

Obere Schranke: Greedy (fraktional)

$S=\emptyset$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$Z=9$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=1 : \sum_{i \in S} z_i = 2, \sum_{i \in S} p_i = 6$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=1 : \sum_{i \in S} z_i = 2, \sum_{i \in S} p_i = 6$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=1 : \sum_{i \in S} z_i = 2, \sum_{i \in S} p_i = 6$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} \left(= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3} \right)$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} \left(= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3} \right)$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} \left(= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3} \right)$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} \left(= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3} \right)$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} \left(= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3} \right)$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} (= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3})$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} \left(= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3} \right)$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} (= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3})$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} (= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3})$$

$$\text{Damit: } \sum_{i \in S} x_i z_i = 9, \sum_{i \in S} x_i p_i = 11 + \frac{2}{3} * 8 = 16, \bar{3}$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} (= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3})$$

$$\text{Damit: } \sum_{i \in S} x_i z_i = 9, \sum_{i \in S} x_i p_i = 11 + \frac{2}{3} * 8 = 16, \bar{3}$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} (= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3})$$

$$\text{Damit: } \sum_{i \in S} x_i z_i = 9, \sum_{i \in S} x_i p_i = 11 + \frac{2}{3} * 8 = 16, \bar{3}$$

Obere Schranke: Greedy (fraktional)

$$S = \emptyset$$

$$\bar{S} = \emptyset$$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$$Z = 9$$

$$i=2 : \sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$$

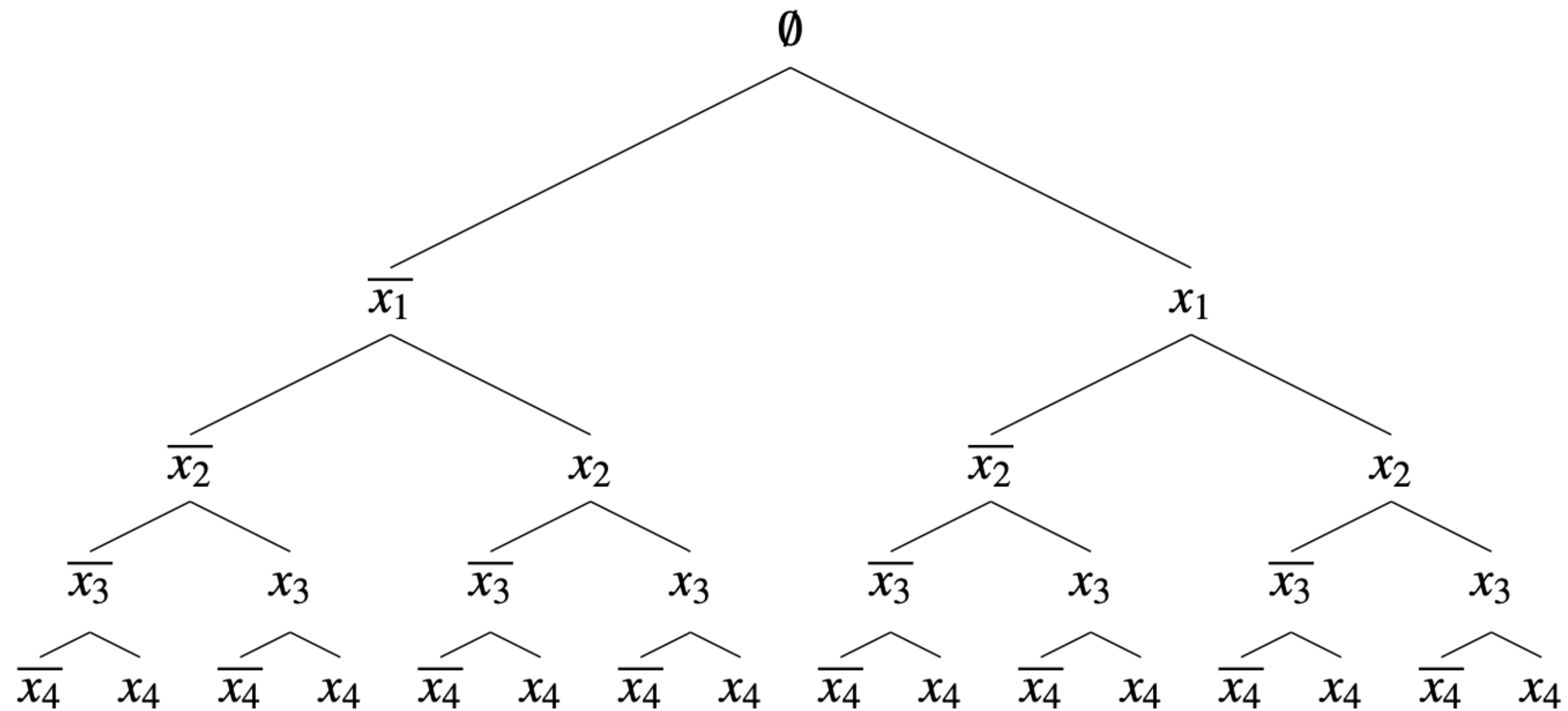
$$\text{Bleibt: } Z - \sum_{i \in S} z_i = 4, \text{ also } x_3 = \frac{2}{3} (= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3})$$

$$\text{Damit: } \sum_{i \in S} x_i z_i = 9, \sum_{i \in S} x_i p_i = 11 + \frac{2}{3} * 8 = 16, \bar{3}$$

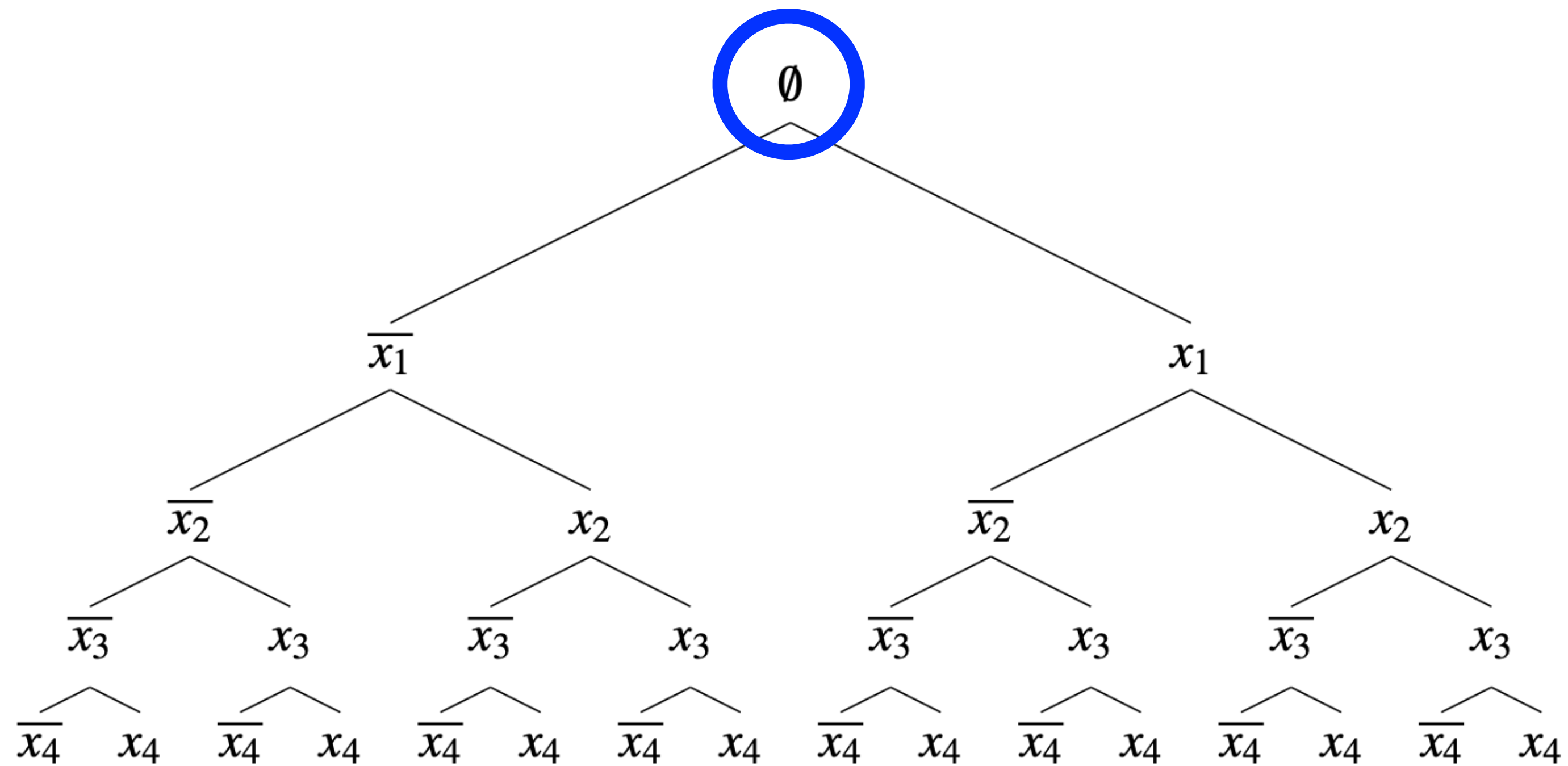
$$UB = 16$$

Gesamtbeispiel

Gesamtbeispiel

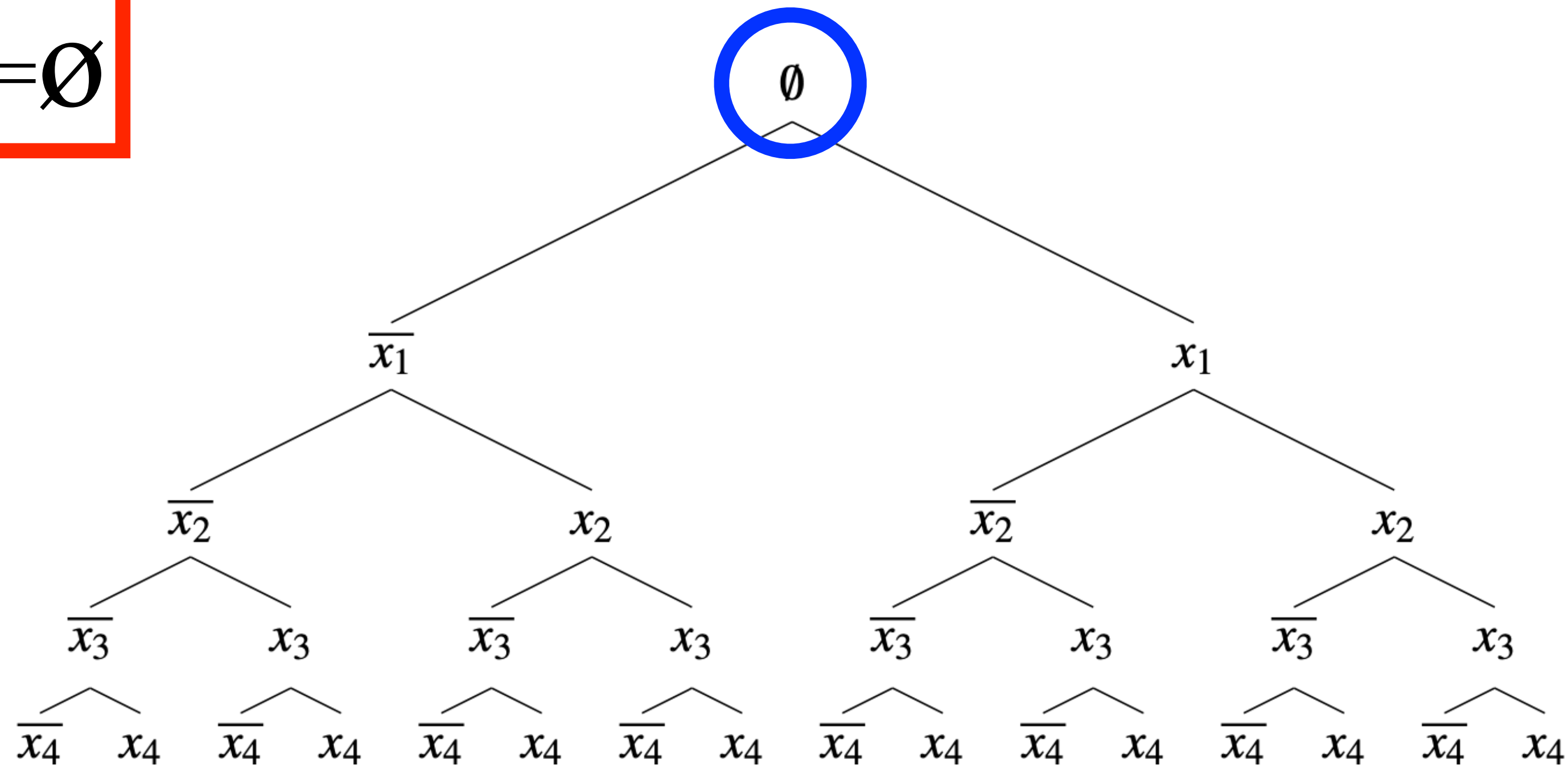


Gesamtbeispiel

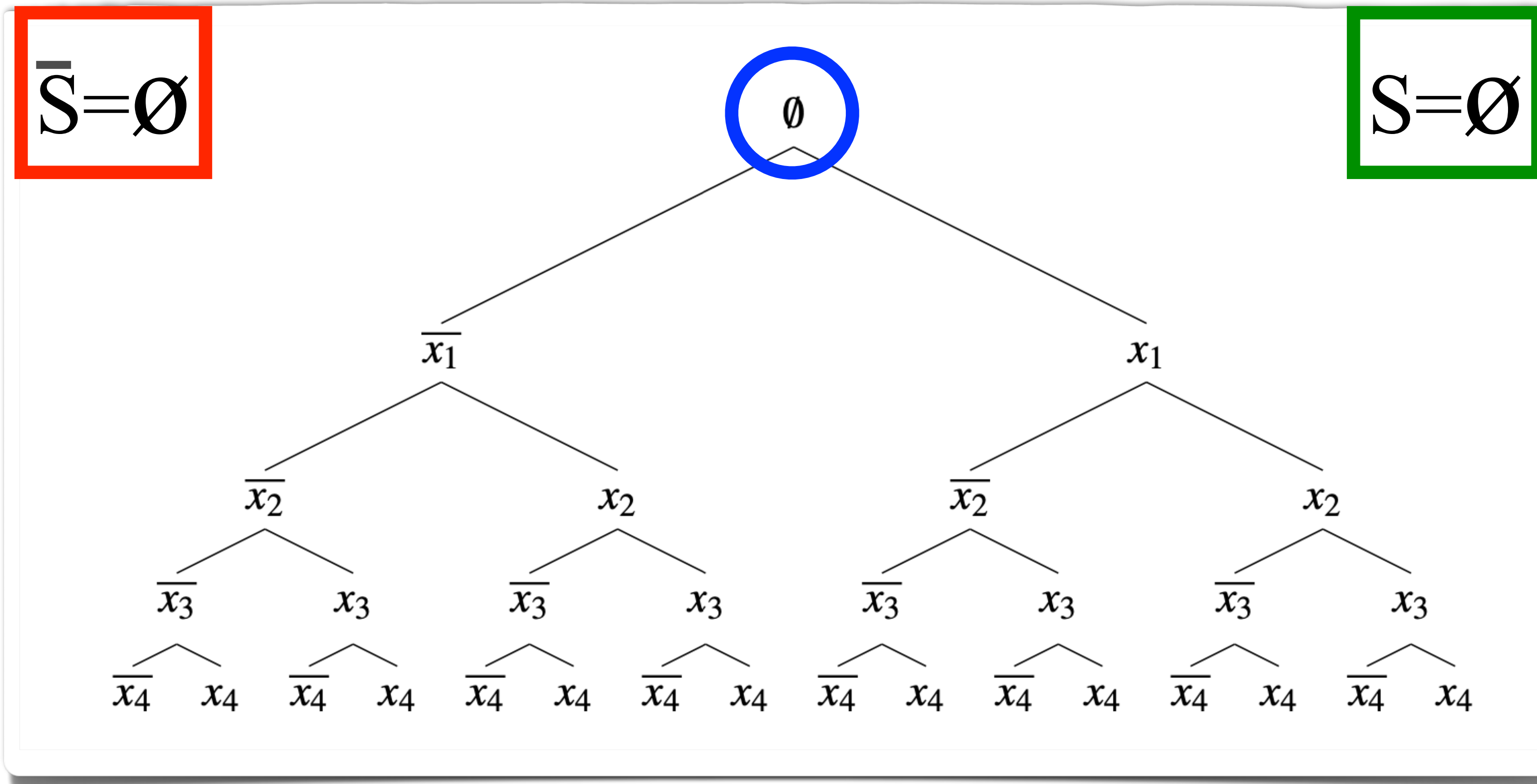


Gesamtbeispiel

$$\bar{S} = \emptyset$$



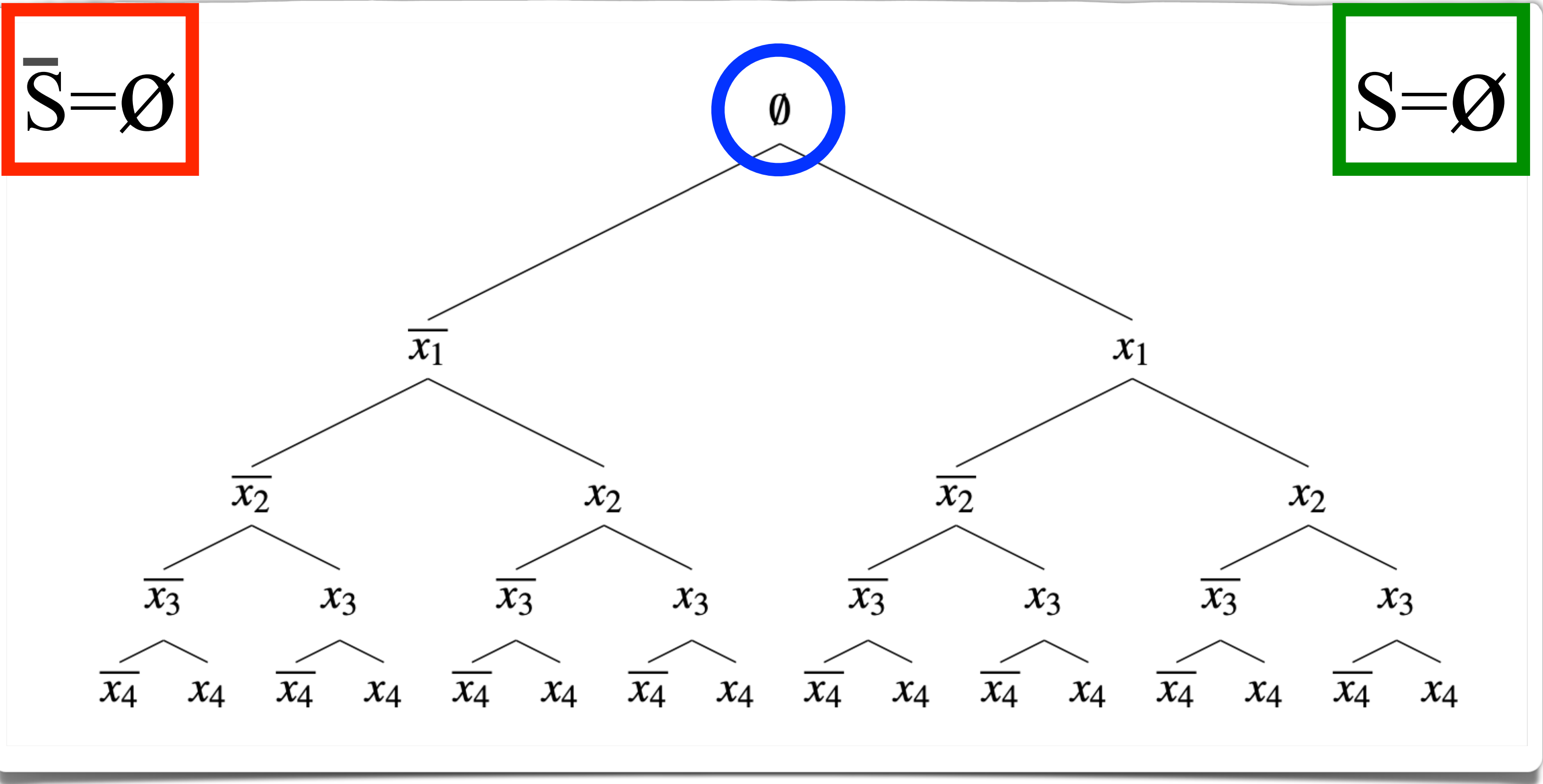
Gesamtbeispiel



Gesamtbeispiel

$\bar{S} = \emptyset$

$S = \emptyset$

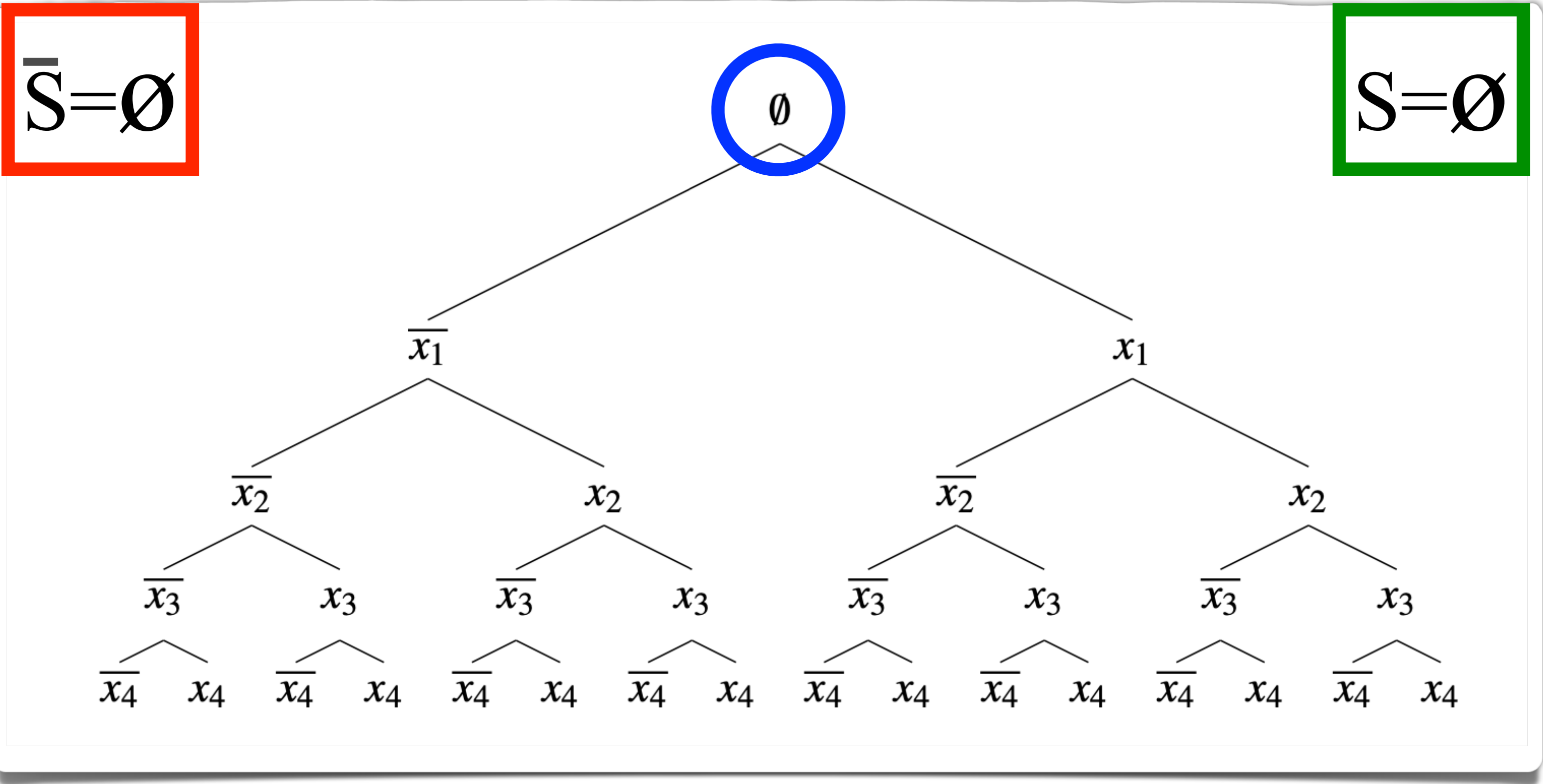


i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

Gesamtbeispiel

$\bar{S} = \emptyset$

$S = \emptyset$

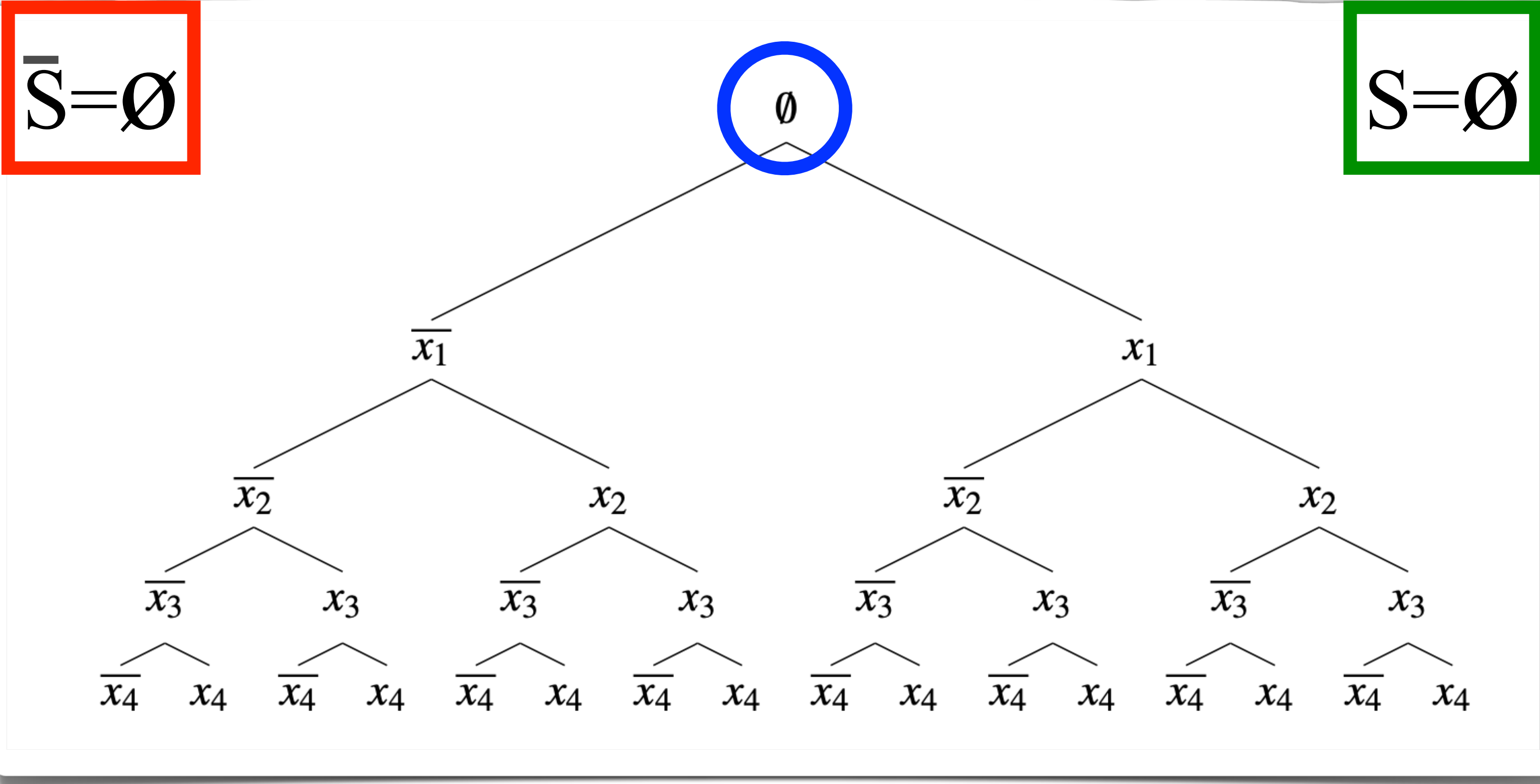


<i>i</i>	1	2	3	4	5	6	7
<i>z_i</i>	2	3	6	7	5	9	4
<i>p_i</i>	6	5	8	9	6	7	3

Gesamtbeispiel

$\bar{S} = \emptyset$

$S = \emptyset$

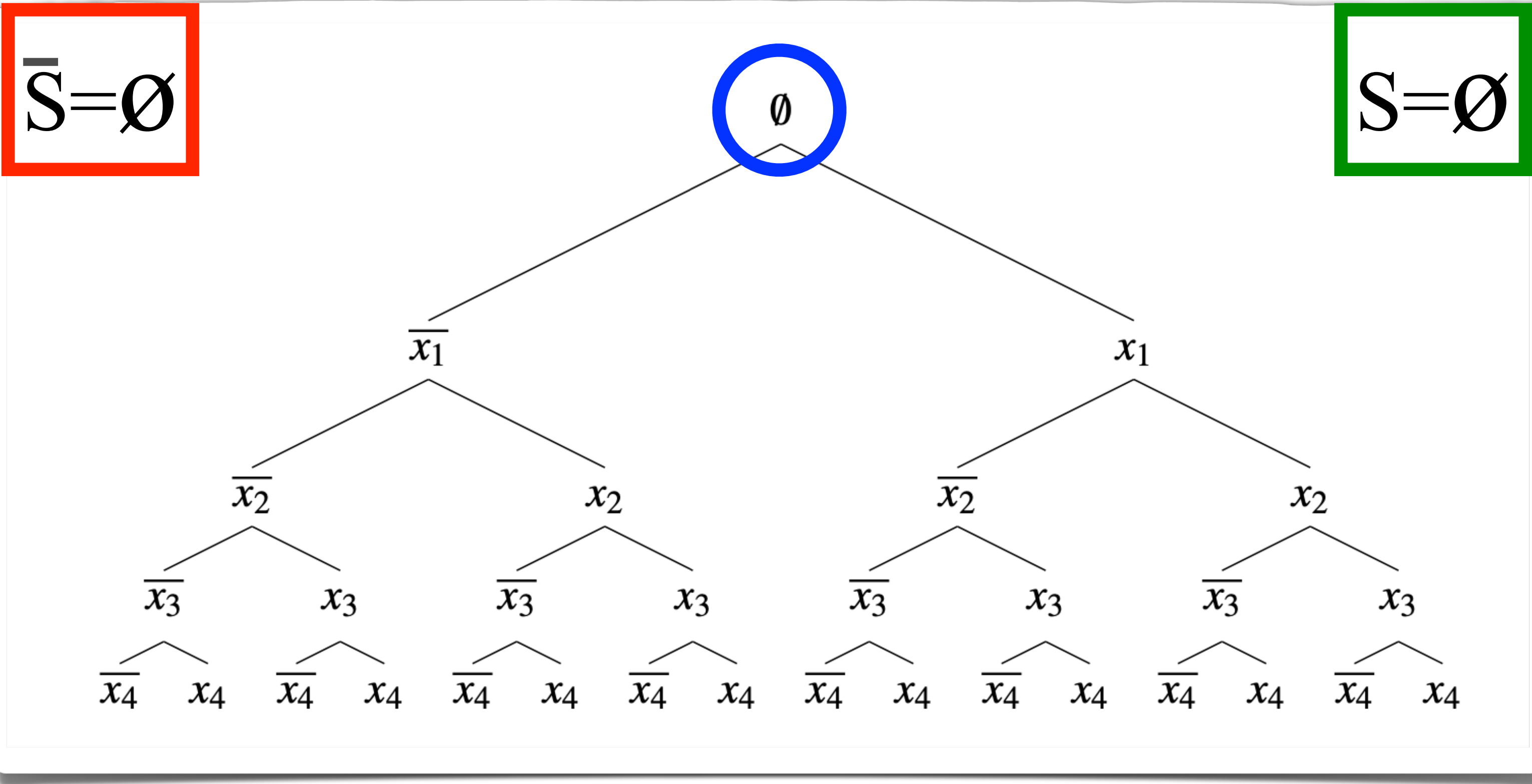


<i>i</i>	1	2	3	4	5	6	7
<i>z_i</i>	2	3	6	7	5	9	4
<i>p_i</i>	6	5	8	9	6	7	3

Gesamtbeispiel

$$\bar{S} = \emptyset$$

$$S = \emptyset$$

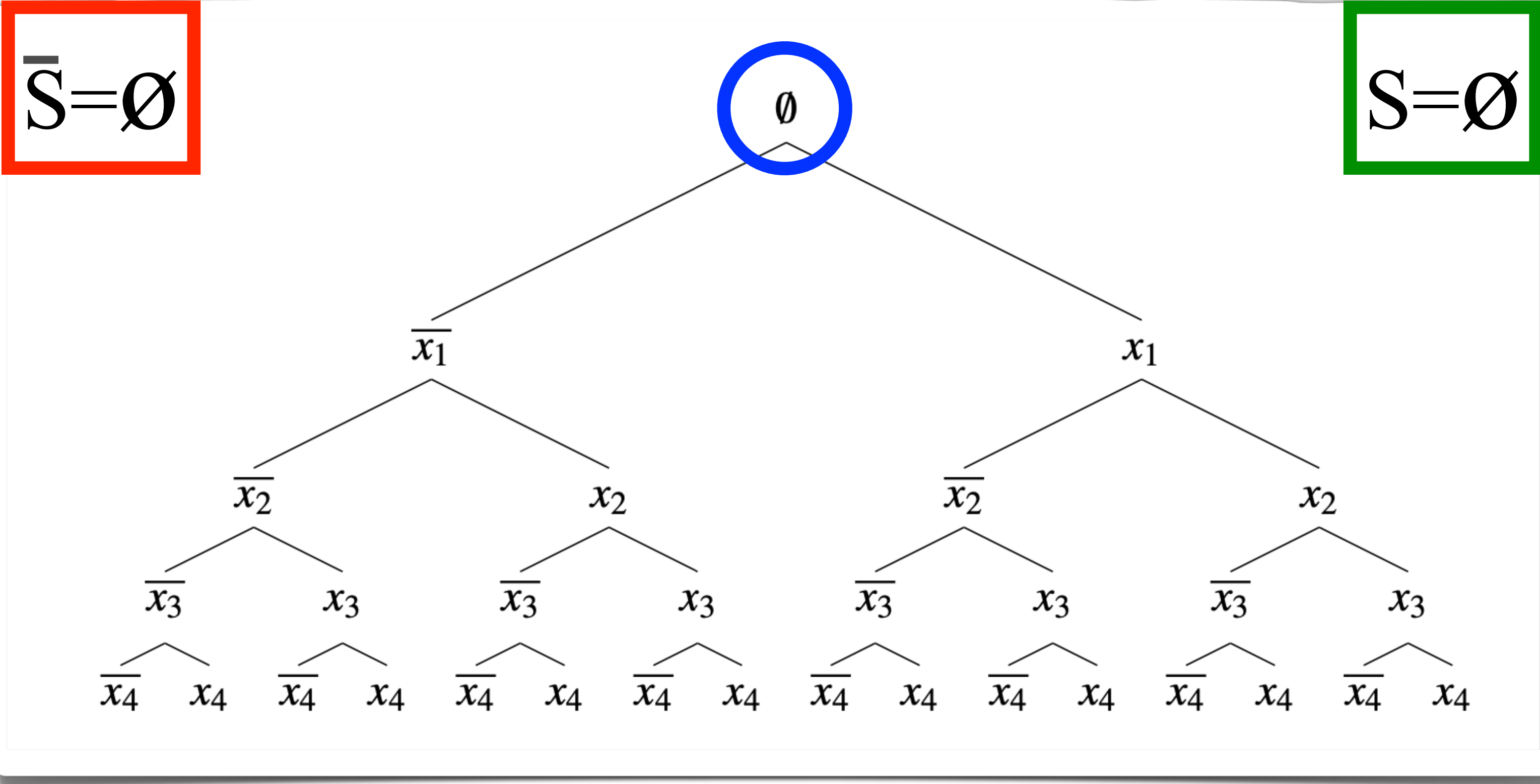


i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

Gesamtbeispiel

$\bar{S} = \emptyset$

$S = \emptyset$



<i>i</i>	1	2	3	4	5	6	7
<i>z_i</i>	2	3	6	7	5	9	4
<i>p_i</i>	6	5	8	9	6	7	3

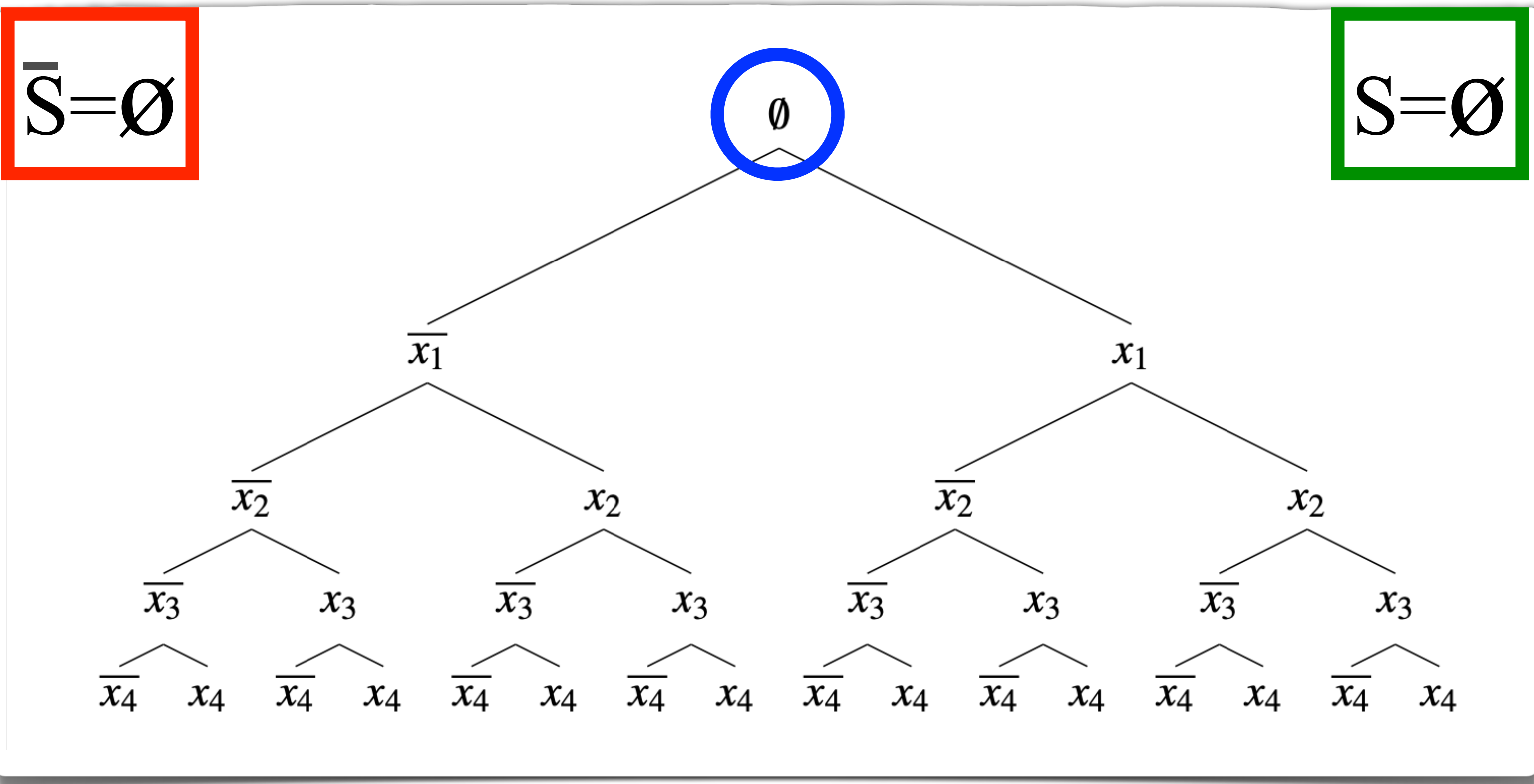
LB=14

Gesamtbeispiel

$$\bar{S} = \emptyset$$



$S = \emptyset$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

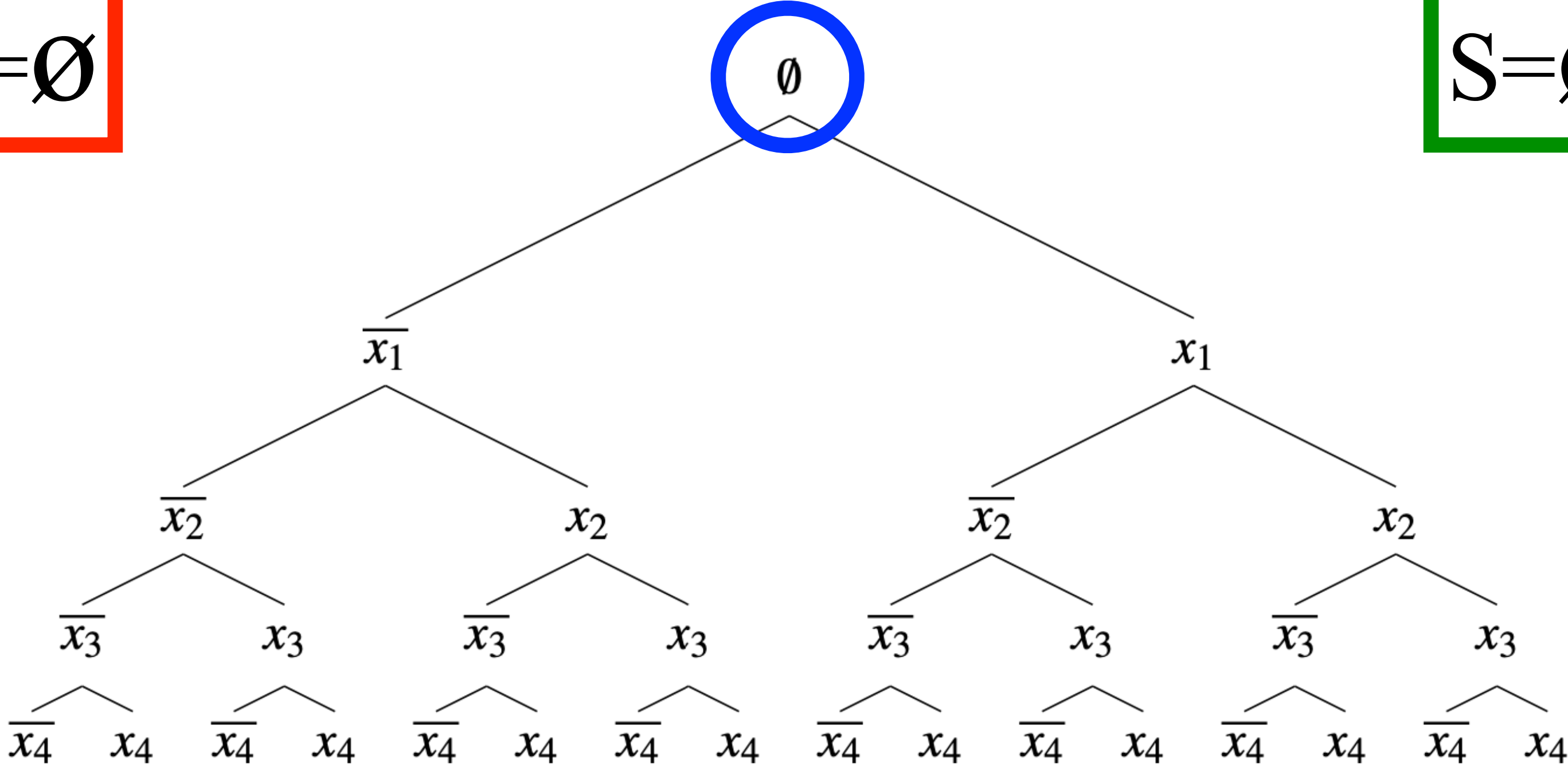


LB=14

Gesamtbeispiel

$$\bar{S} = \emptyset$$

$$S = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

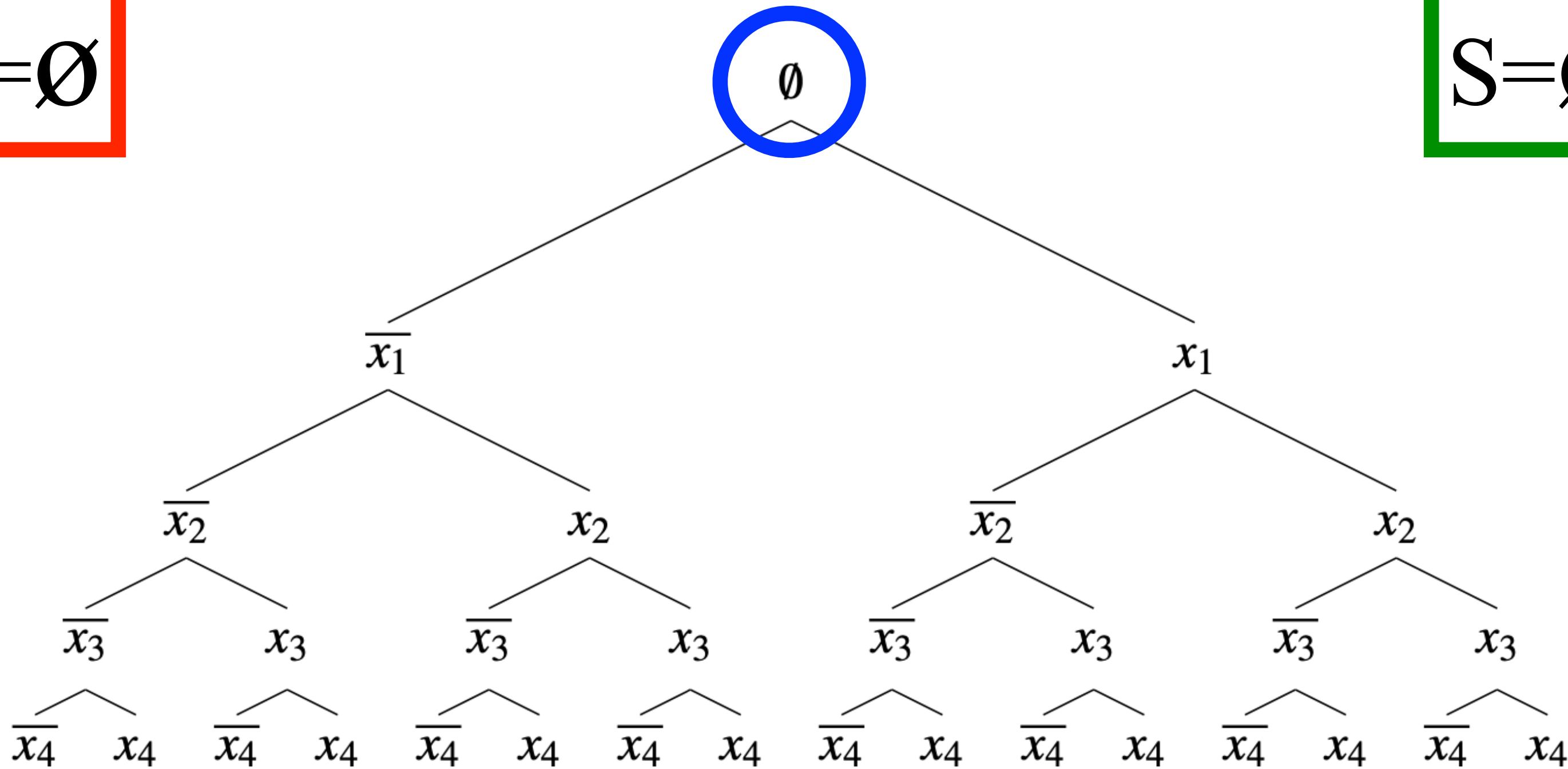
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

LB=14

Gesamtbeispiel

$$\bar{S} = \emptyset$$

$$S = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

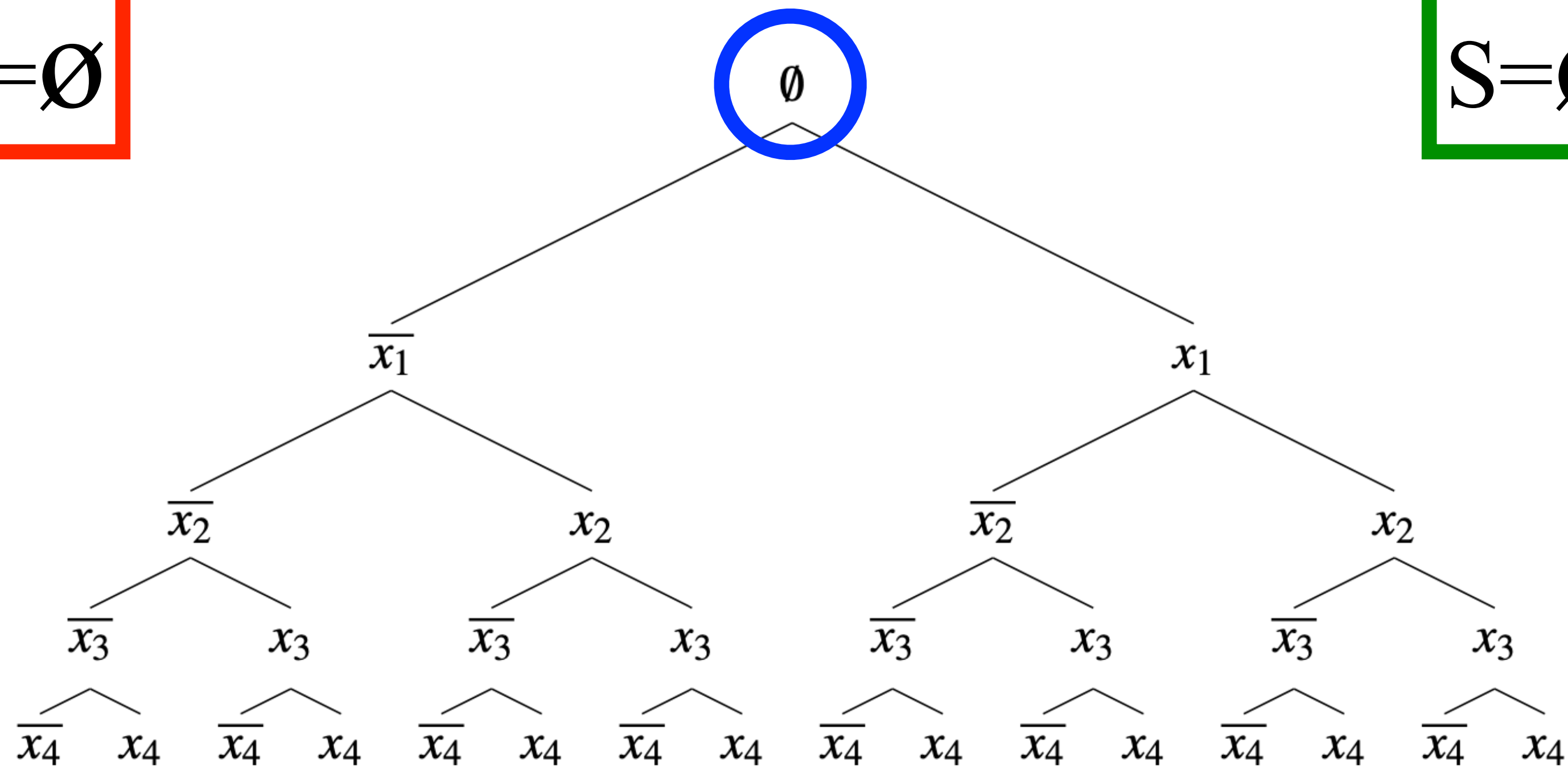
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

LB=14

Gesamtbeispiel

$$\bar{S} = \emptyset$$

$$S = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

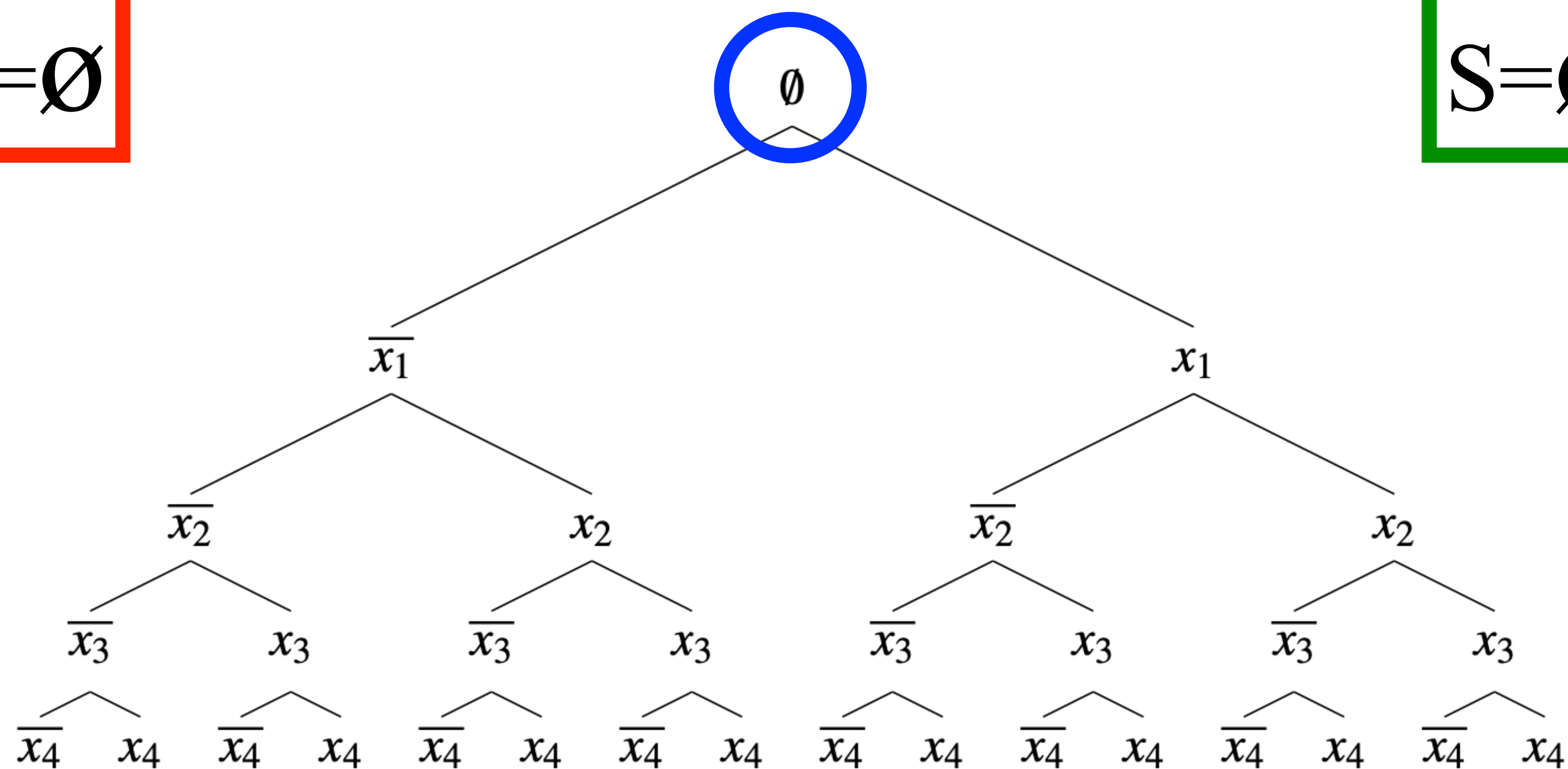
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

LB=14

Gesamtbeispiel

$$\bar{S} = \emptyset$$

$$S = \emptyset$$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

UB=16

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

LB=14

Vielen Dank!

s.fekete@tu-bs.de