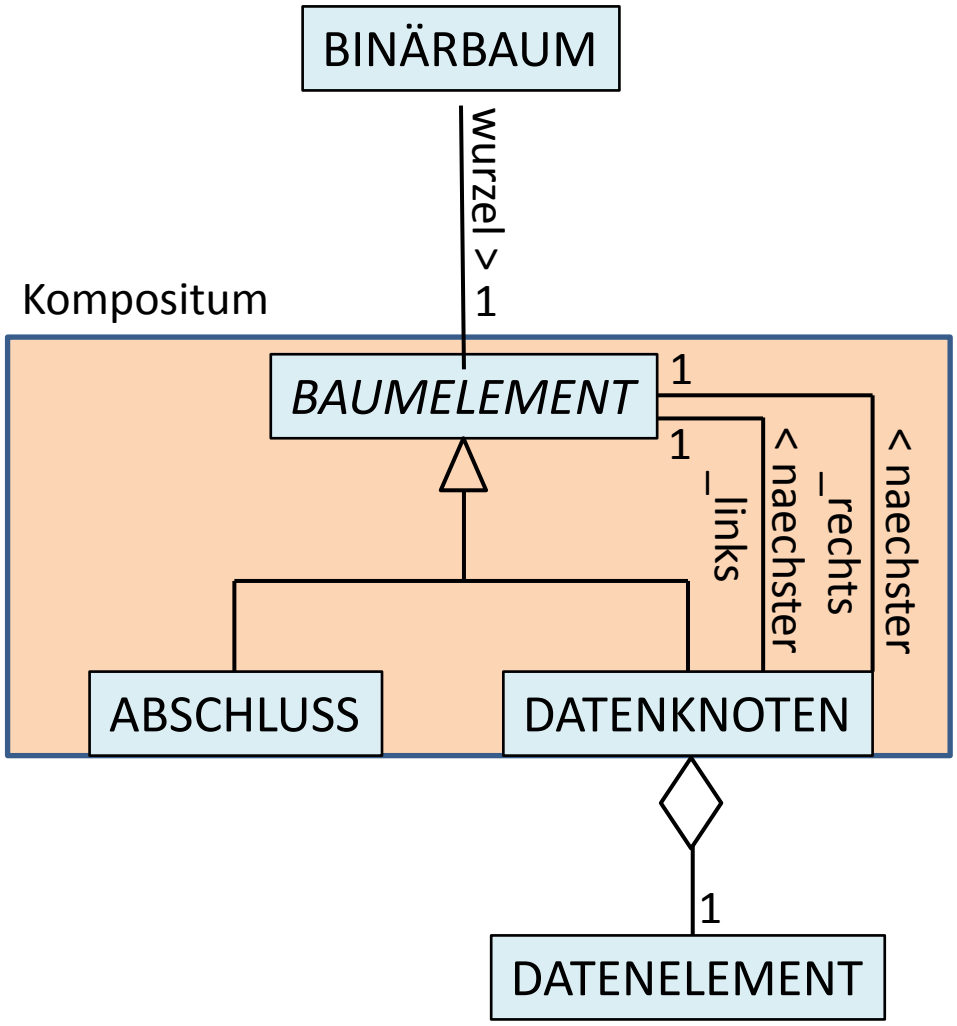
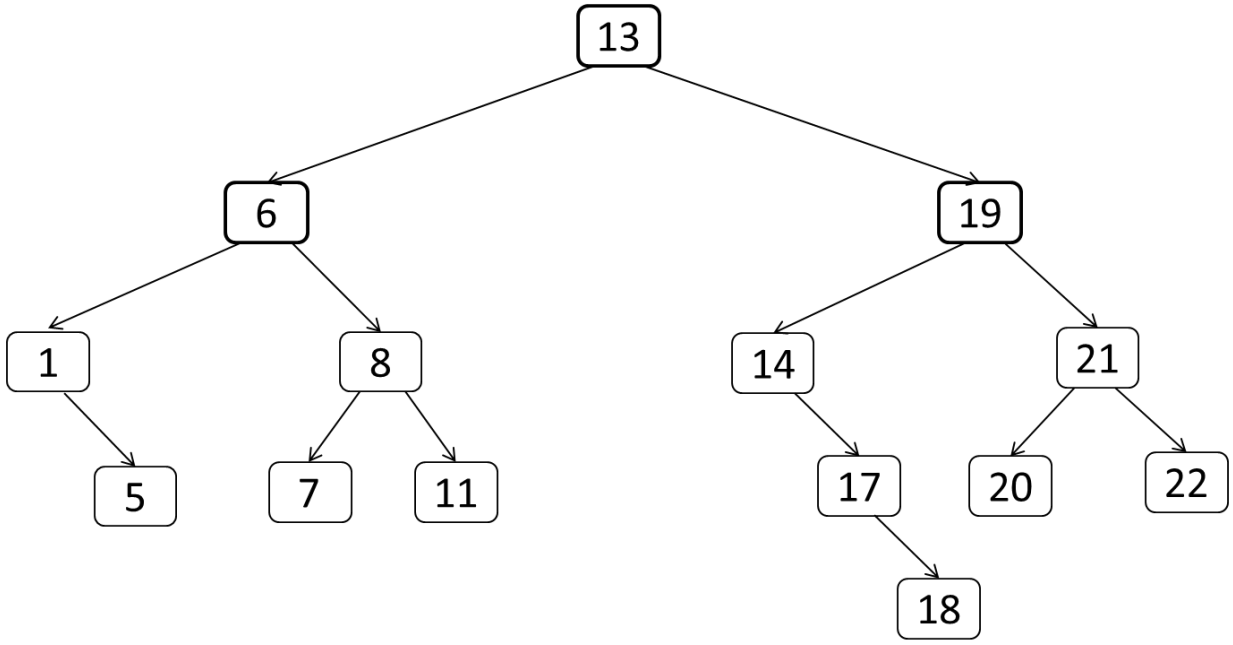
S. 131/3 Suchen in binären Bäumen



Geordneter Binärbaum:

Ein Binärbaum heißt geordnet, wenn die Knoten des linken Teilbaums eines Knotens nur kleinere Schlüssel und die Knoten des rechten Teilbaums eines Knotens nur größere Schlüssel als der Knoten selbst besitzen.

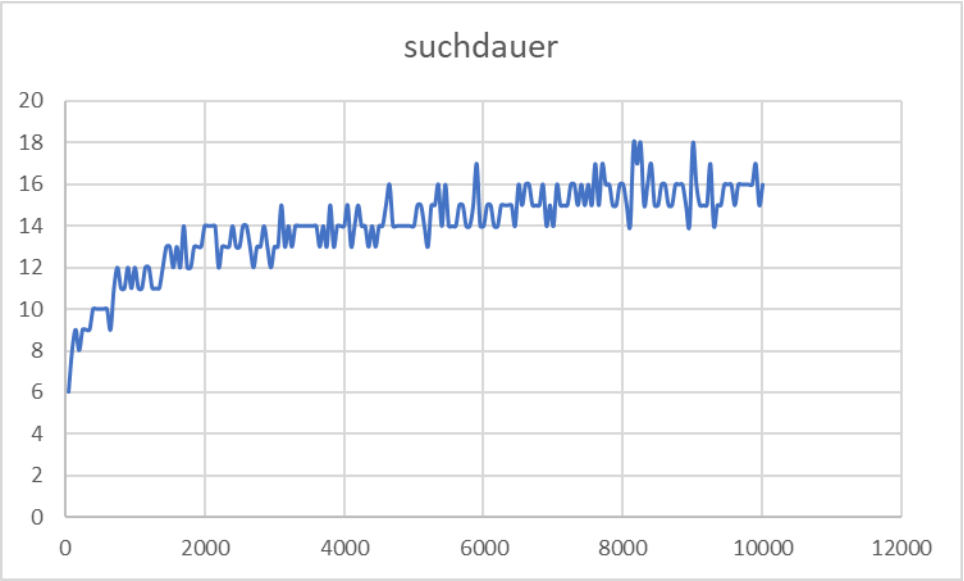


Öffne die Vorlage S131-A3.
Warum können hier nicht die ersten eingefügten Elemente als Suchelemente benutzt werden?
Wie wird hier die Suchliste gefüllt?
Ergänze die Methode suchdauer() in Test. Benutze dazu die Methode inhaltSuchenZaehlen(…) der Klasse Suchbaum.

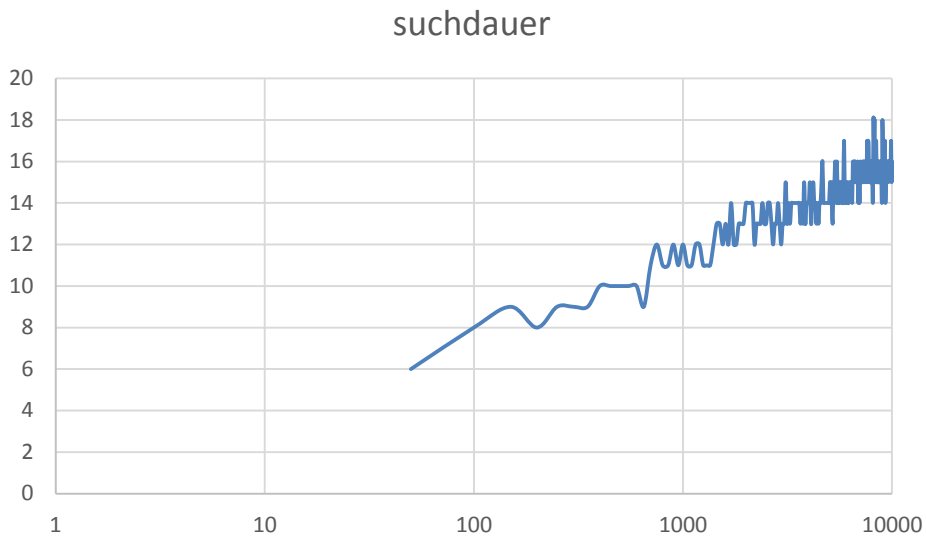
```
public int suchdauer(){
        int suchdauer = 0;
        for (int i = 0; i< suchliste.length;i++){
            suchdauer = suchdauer +
                              sbaum.inhaltSuchenZaehlen
                                        (new Eintrag("", suchliste[i]));
        }
        return (int) (suchdauer/suchliste.length);
 }
```

Variiere n und miss die jeweilige Suchdauer für s Einträge.
Trage die Ergebnisse in eine Tabelle ein.
Welche Laufzeitordnung ergibt sich?
Welche günstigen und ungünstigen Fälle können beim Aufbau des Baums vorkommen?

n	suchdauer
50	5
100	7
150	7
200	9
250	8
300	8
350	10
400	10
450	11
500	10
550	11
600	10
650	11
700	10
750	11
800	10
850	12
900	10
950	11
1000	12
1050	11
1100	12



Laufzeitordnung: O(log n)
günstig: ausbalanciert O(log n)
ungünstig: zu Liste entartet O(n)



Skaliert man die x-Achse logarithmisch, erkennt man einen linearen Zusammenhang!

Die Ackermann-Funktion

- sehr schnelles Wachstum
  (noch schneller: Busy Beaver)
- Test: Was ist berechenbar?

http://www.youtube.com/watch?v=VZz7m91XxoM

$$ack(n,m) = \begin{cases} m+1, & falls\ n = 0 \\ ack(n-1,1), & falls\ m = 0 \\ ack(n-1, ack(n, m-1)) & sonst \end{cases}$$

$$ack(n,m) = \begin{cases} m+1, & falls\ n = 0 \\ ack(n-1,1), & falls\ m = 0 \\ ack(n-1, ack(n,m-1)) & sonst \end{cases}$$

```
public class Ackermann
{
    public int ack(int n, int m){
        if(n==0) {
            return m+1;
        }
        else if (m==0){
            return ack(n-1,1);
        }
        else {
            return ack(n-1, ack(n, m-1));
        }
    }
```

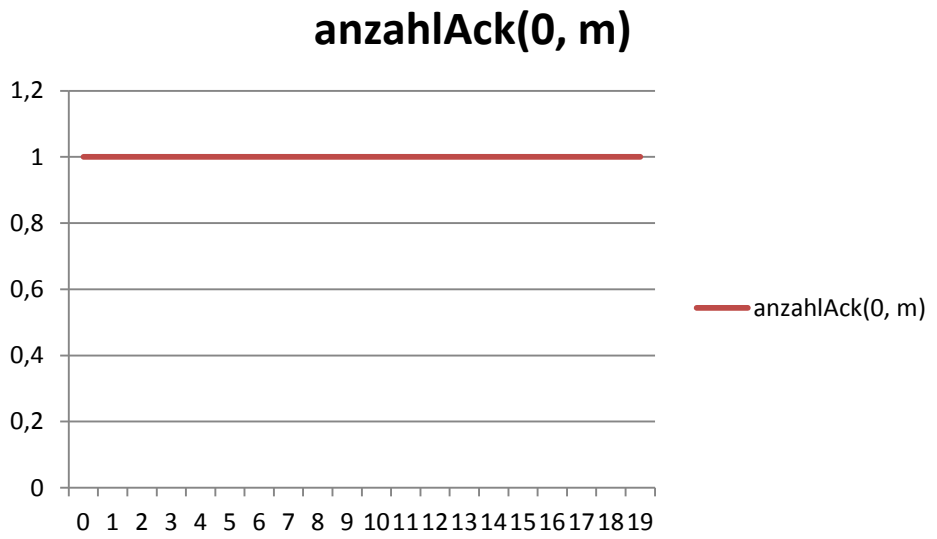
```
public long anzahlAck(int n, int m){
   if(n==0) {
      return 1;
    }
    else if (m==0){
      return 1+ anzahlAck(n-1,1);
    }
    else {
      return 1 + anzahlAck(n-1, ack(n, m-1))+ anzahlAck(n, m-1);
    }
 }

public void schleife (int n, int m){
    for (int i=0; i<m; i++){
      System.out.println( i + "; " +anzahlAck(n, i));
    }
 }
```

$$ack(n,m) = \begin{cases} m+1, & falls\, n = 0 \\ ack(n-1,1), & falls\, m = 0 \\ ack(n-1, ack(n,m-1)) & sonst \end{cases}$$

Laufzeitverhalten:

- ack(0, m)

m	anzahlAck(0, m)
0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	1
18	1
19	1

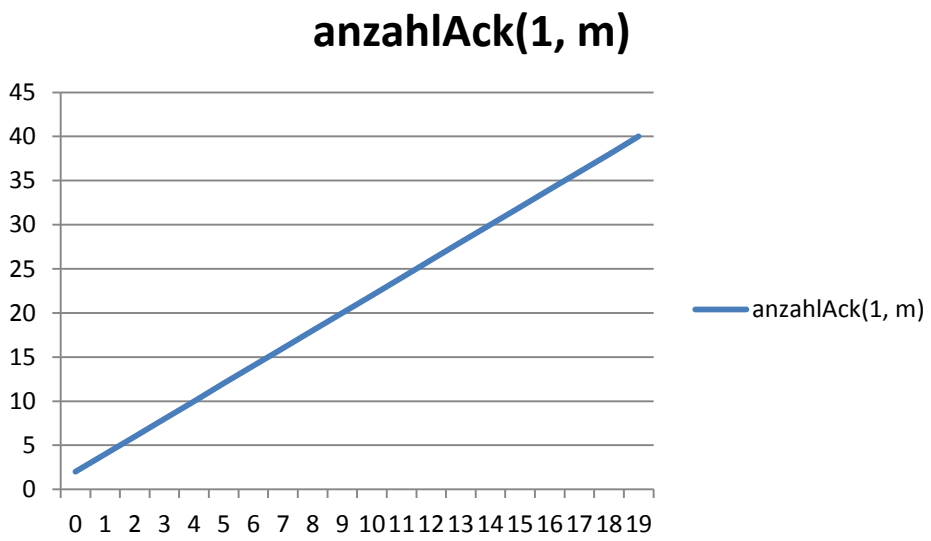


konstantes Laufzeitverhalten:
O(1)

Laufzeitverhalten:

- ack(1, m)

m	anzahlAck(1, m)
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20
10	22
11	24
12	26
13	28
14	30
15	32
16	34
17	36
18	38
19	40



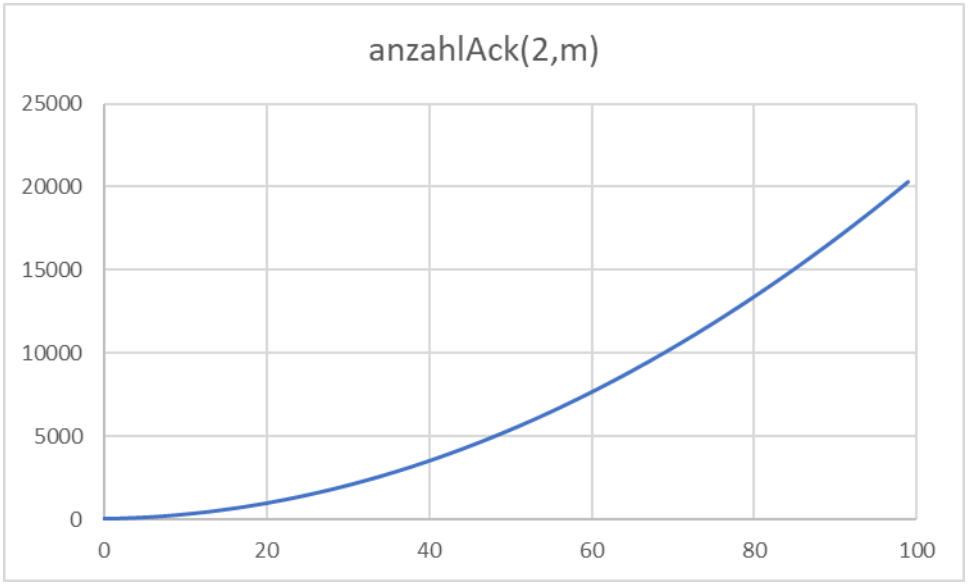
y =2m+2

lineares Wachstum: O(n)

Laufzeitverhalten:

- ack(2, m) (Messung bis m=100)

m	anzahlAck(2, m)
0	5
1	14
2	27
3	44
4	65
5	90
6	119
7	152
8	189
9	230
10	275
11	324
12	377
13	434
14	495
15	560
16	629
17	702
18	779
19	860



Vermutung: polynomiale Laufzeit!

Maßgeblich ist die höchste Potenz:

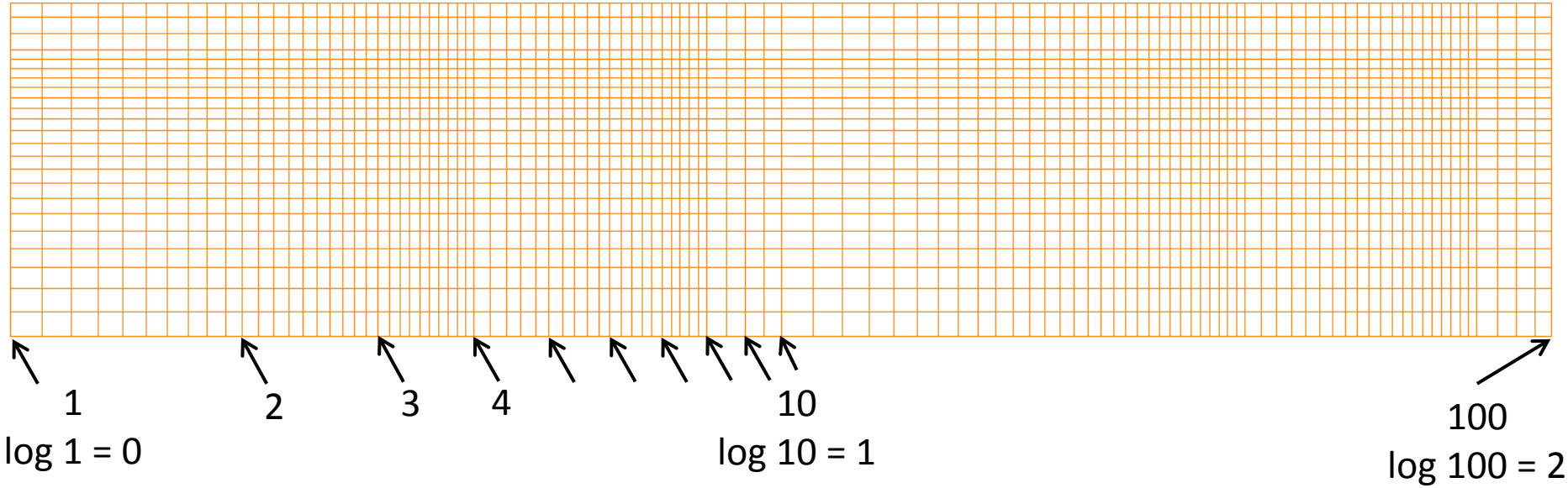
$$y = c \cdot x^n$$

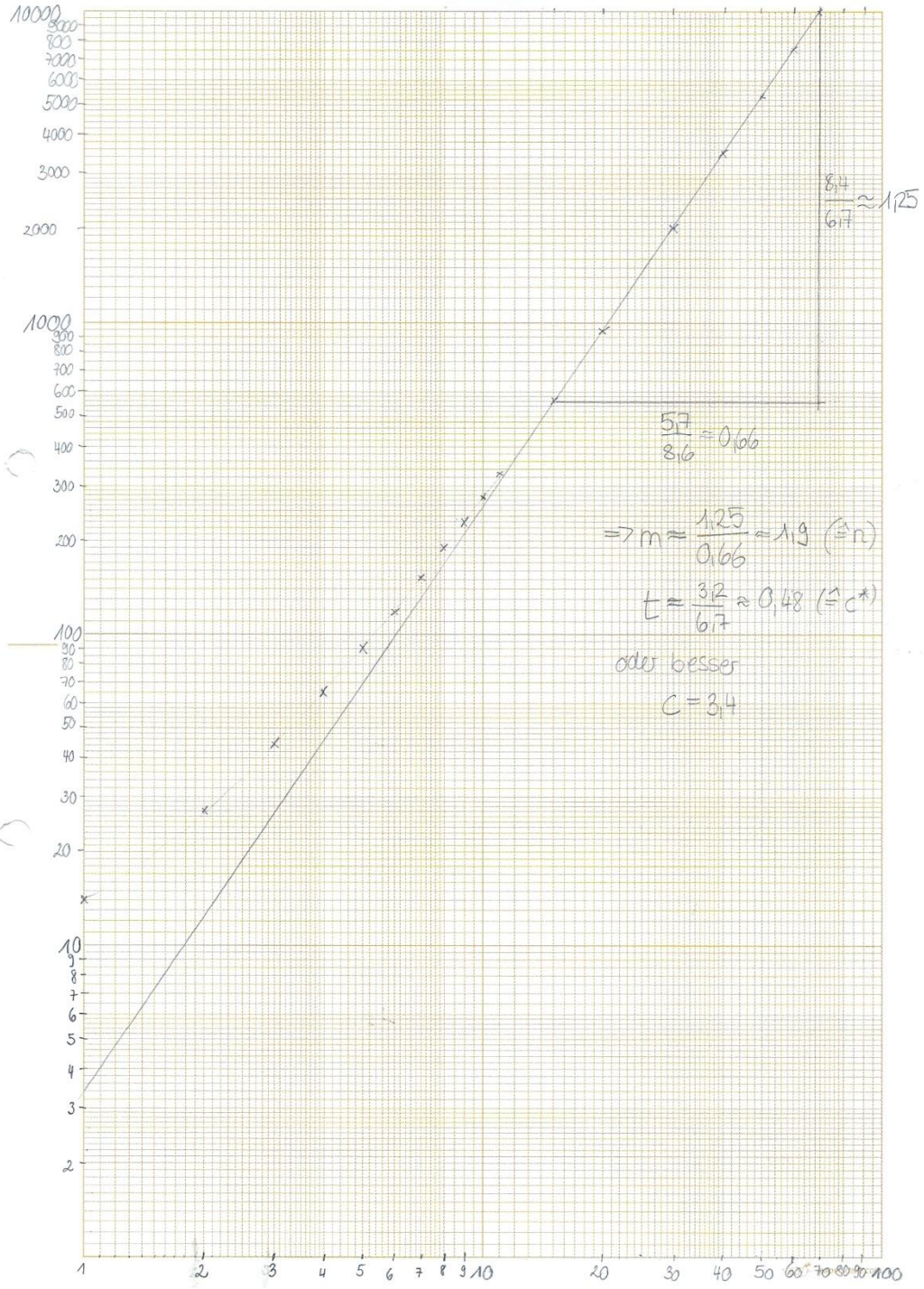
$$\log y = \log(c \cdot x^n)$$

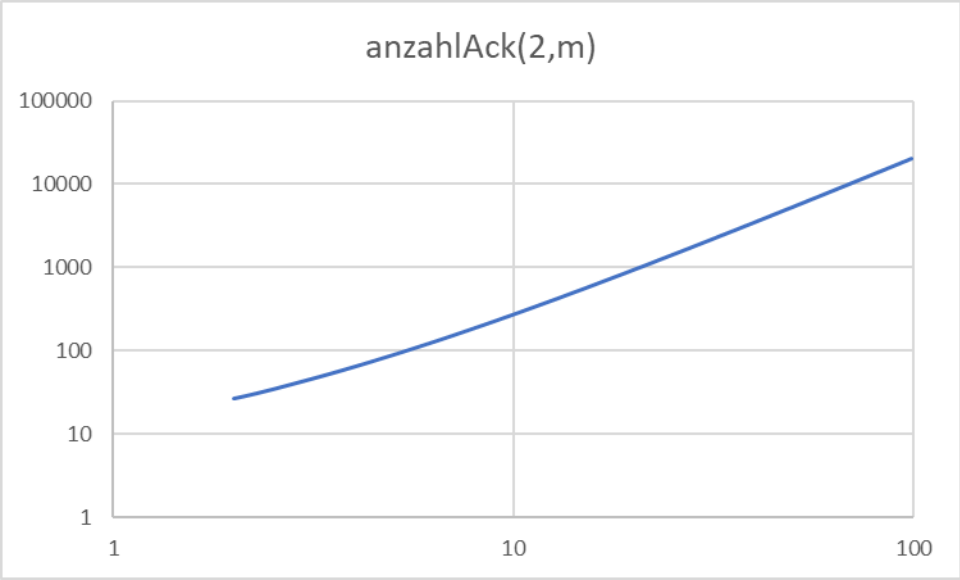
$$\log y = \log c + n \cdot \log x$$

$$y^* = c^* + n \cdot x^*$$

Doppelt logarithmisch aufgetragen ist das eine Gerade mit Steigung n und y-Abschnitt c*.




10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
900
800
700
600
500
400
300
200
100
90
80
70
60
50
40
30
20
10
9
8
7
6
5
4
3
2
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
8,4 / 6,7 ≈ 1,25
57 / 8,6 ≈ 0,66
⇒ m ≈ 1,25 / 0,66 ≈ 1,9 (≙ n)
t ≈ 3,2 / 6,7 ≈ 0,48 (≙ c*)
oder besser
c = 3,4



Doppelt logarithmisch aufgetragen ist ein linearer Zusammenhang erkennbar.
Dies ist ein Nachweis für polynomiales Laufzeitverhalten!

Bestimme aus zwei Wertepaaren der Wertetabelle c und n für anzahlAck(2, m) (z.B. m=70 und m= 15):

(1) $\log 10295 = n \cdot \log 70 + \log c$

(2) $\log 560 = n \cdot \log 15 + \log c$

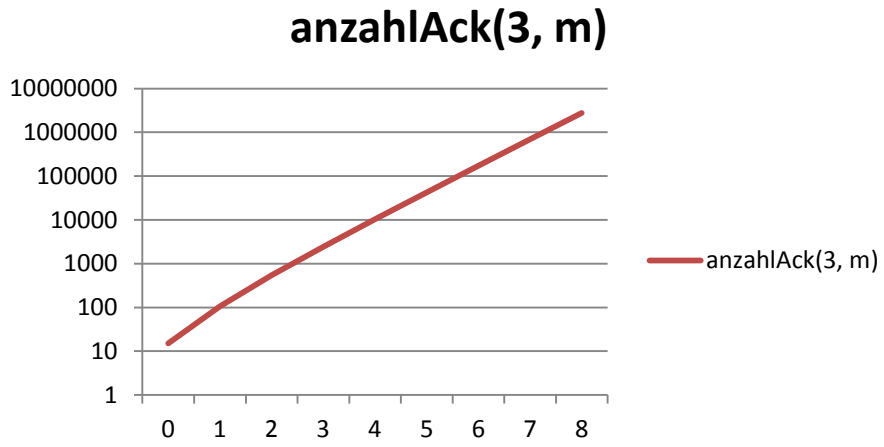
$$n = \frac{\log 10295 - \log 560}{\log 70 - \log 15} \approx 1,89$$

$$\log c = \log 10295 - n \cdot \log 70 \approx 0,525348$$

$$c \approx 3,4$$

d.h. Laufzeitverhalten von ack(2, m) ist polynomial,
es verhält sich asymptotisch wie $3,4 \cdot m^{1,89}$,
d.h. ungefähr $O(n^2)$.

- ack (3, m): exponentielles Laufzeitverhalten



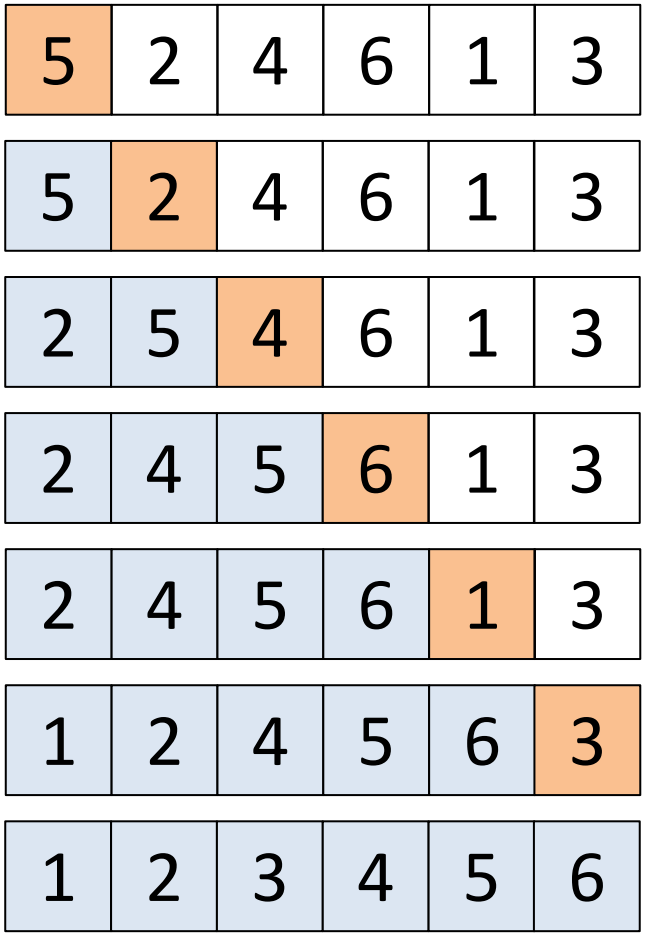
- ack (4, m): unbekanntes Laufzeitverhalten

Vergleiche die folgenden Sortieralgorithmen

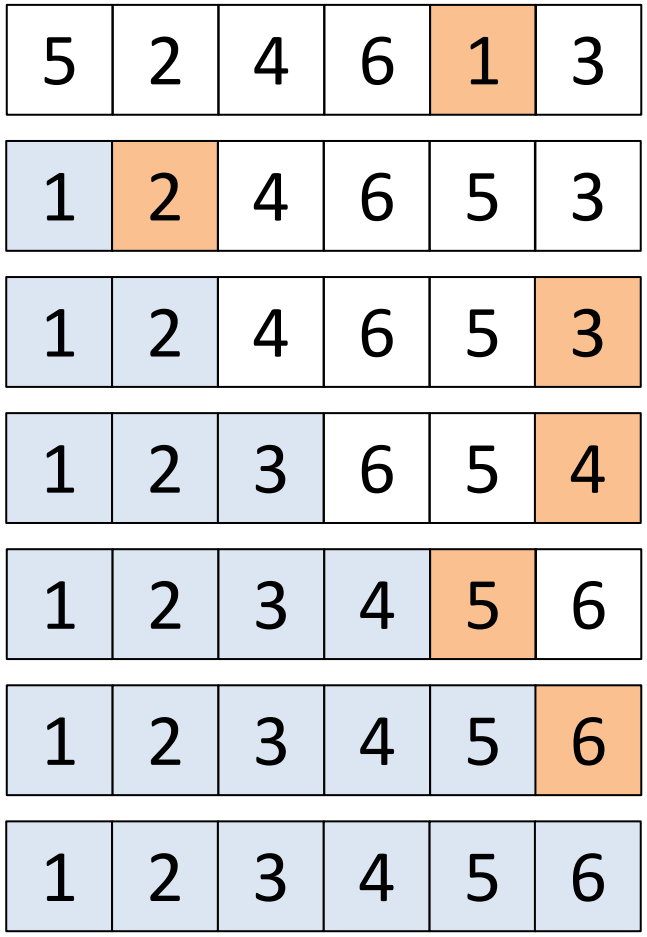
- InsertionSort
- SelectionSort
- BubbleSort
- MergeSort
- QuickSort

Einschub: Sortieralgorithmen

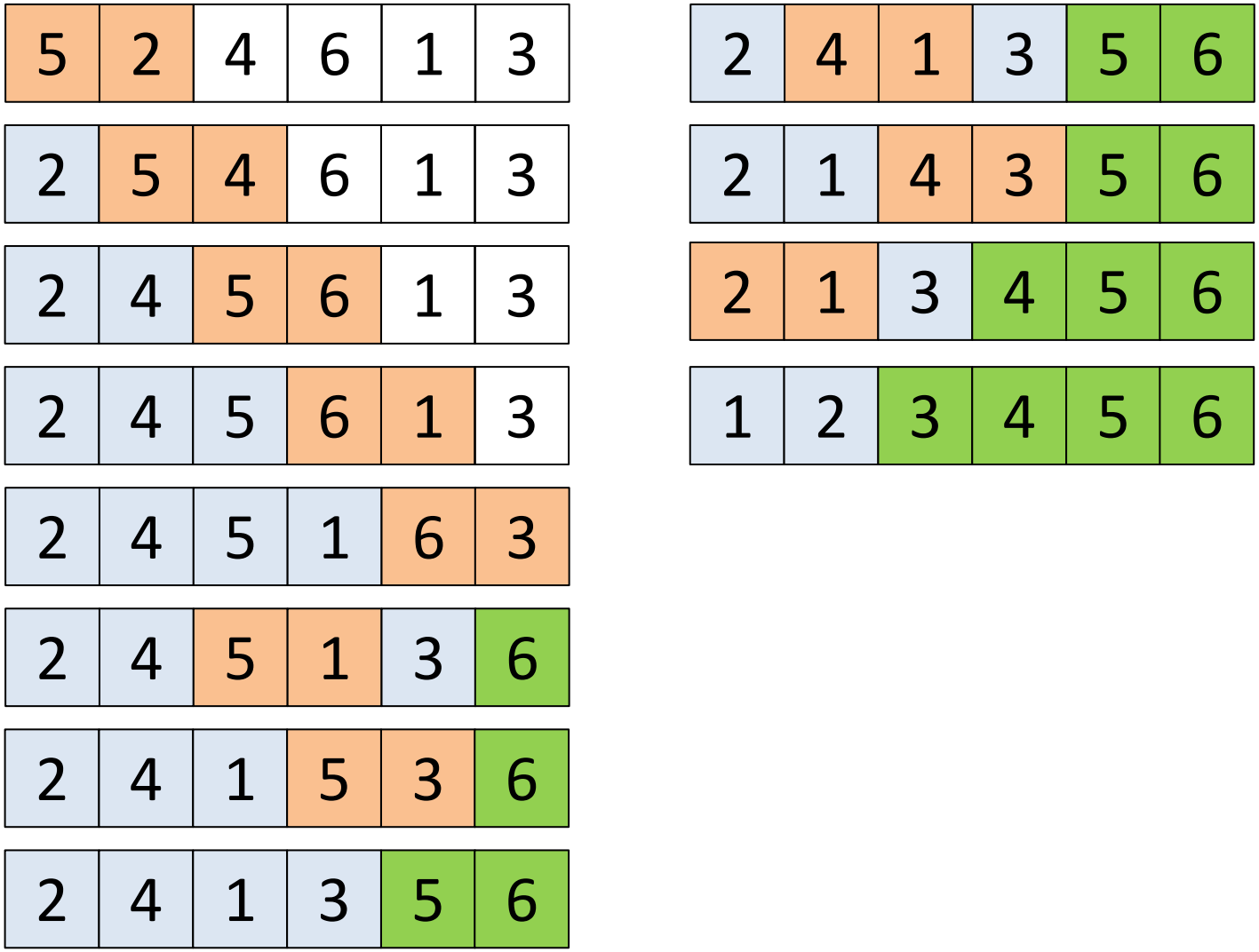
- Insert Sort:
  Elemente des noch nicht sortierten 2. Teils der Liste werden in den ersten Teil der Liste einsortiert.



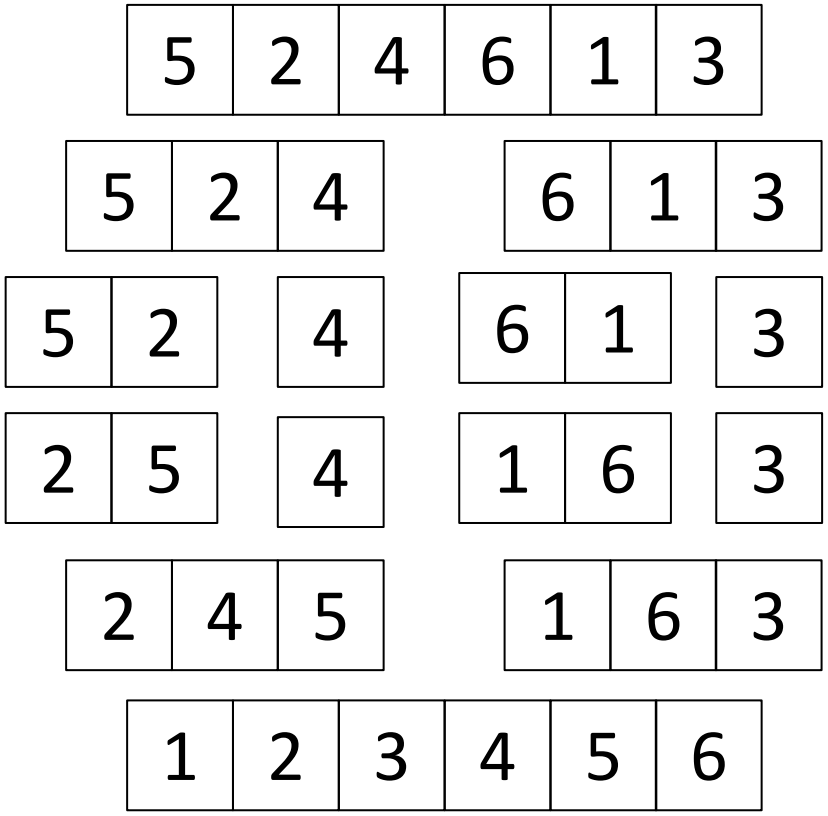
- Selection Sort:
  Das jeweils kleinste Element des noch nicht sortierten 2. Teils der Liste wird in den ersten Teil der Liste einsortiert.



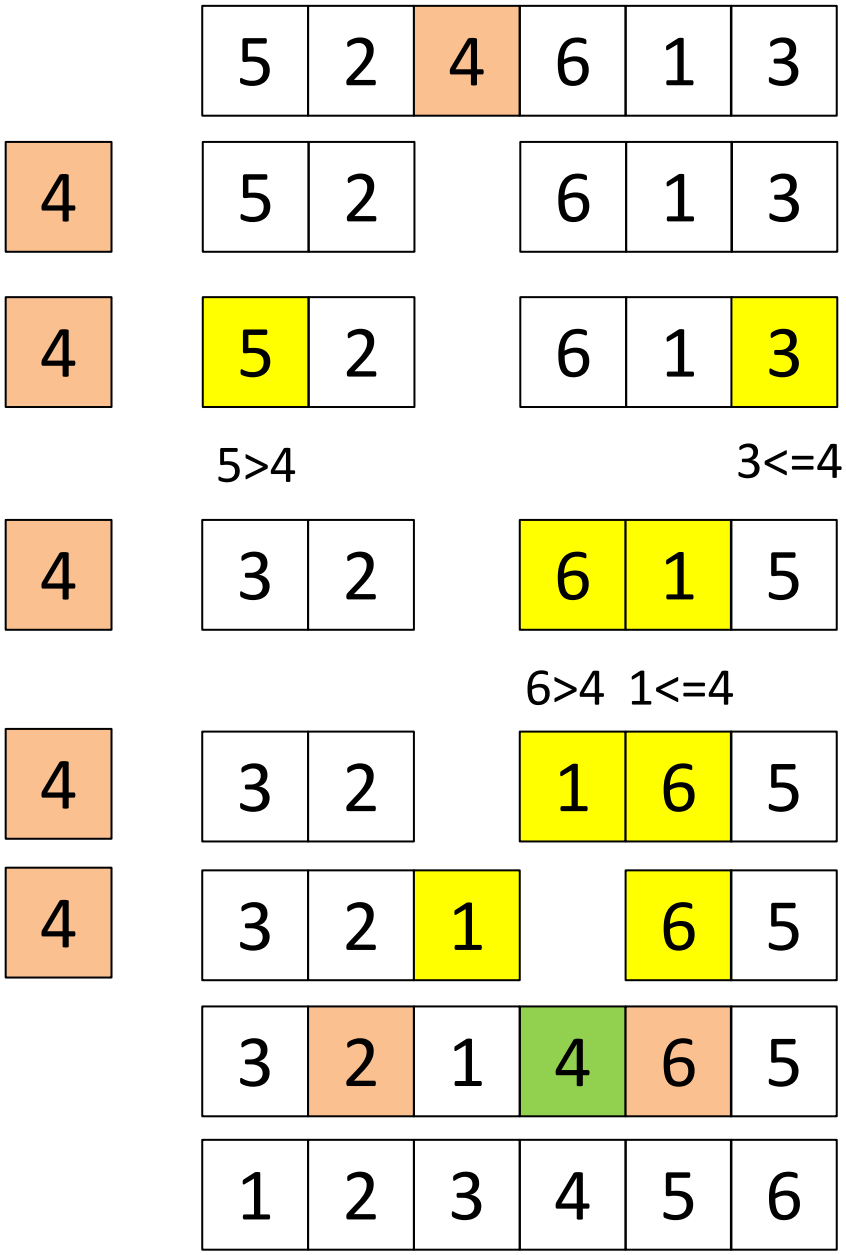
- Bubble Sort:
  Nachbarelemente, die in der falschen Reihenfolge stehen, werden so lange vertauscht, bis das Feld sortiert ist.



- Merge Sort:
  Teile in zwei gleich große Teillisten und sortiere diese getrennt.



- Quick Sort:
  Aus der unsortierten Liste wird ein beliebiges Element p ausgewählt und die übrigen Elemente auf zwei Listen verteilt, von denen die eine nur Elemente kleiner gleich p, die andere nur Elemente größer gleich p enthalten darf.


5 2 4 6 1 3
4
5 2 6 1 3
4
5 2 6 1 3
5>4
3<=4
4
3 2 6 1 5
6>4 1<=4
4
3 2 1 6 5
4
3 2 1 6 5
3 2 1 4 6 5
1 2 3 4 5 6

Schätze das Laufzeitverhalten aufgrund der eigenen Sortierversuche ab.

Lade VisualSort herunter:

BlueJ Projekte - mebis | Infoportal (bayern.de)

Mit VisualSort Veranschaulichung können mehrere Sortieralgorithmen veranschaulicht werden, mit VisualSort Laufzeitmessung kann die Laufzeit gemessen werden.

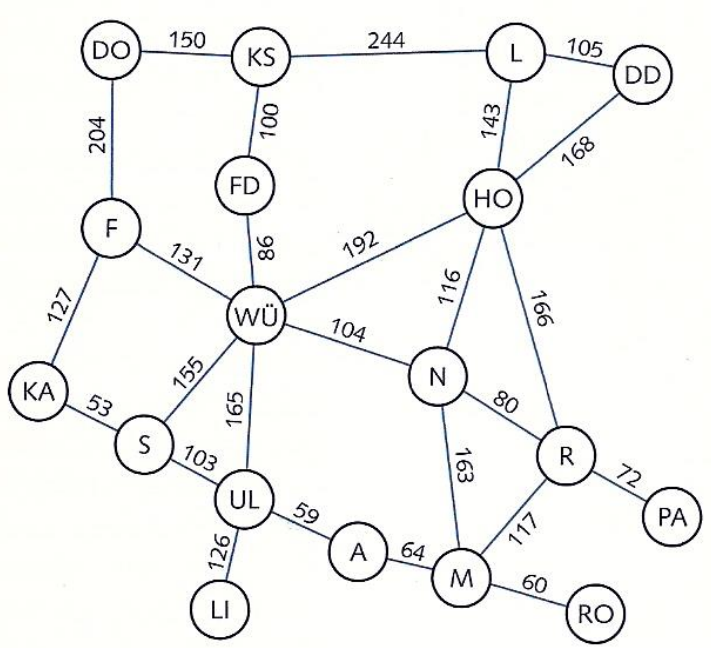
Vergleiche die Laufzeiten (außer BozoSort).

Laufzeit	Best Case	Average Case	Worst Case
Insert Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$

15 Sortieralgorithmen in 6 Minuten
https://www.youtube.com/watch?v=kPRA0W1kECg

Routenplaner:
Brute-Force-Verfahren:
Bestimme alle zyklenfreien Wege vom Start (München) zum Ziel (Frankfurt) und wähle den kürzesten.

Öffne das BlueJ-Projekt Routenplaner. Vervollständige in Arbeit die Methode BruteForceMessreihe().
In der Schleife soll die Zeit für die Wegesuche weitere 1000 mal bestimmt werden und davon das Minimum berechnet werden.

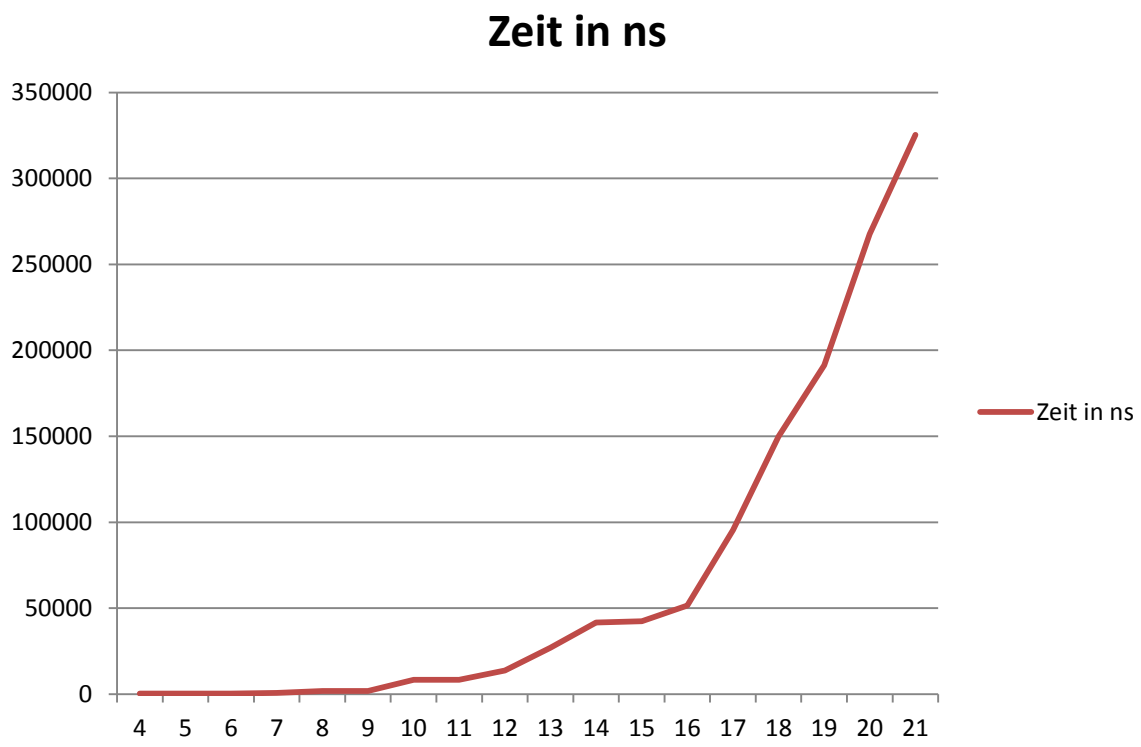


4 Ausschnitt aus Autobahnnetz

aus Informatik – Oberstufe 2
Oldenbourg Verlag

```
public void BruteForceMessreihe() {
      long startzeit, minDauer, dauer;
      for (int i=4; i<22; i++){
        GRAPH_MATRIX g = GraphErzeugen(i);
        startzeit = System.nanoTime();
        g.WegeSuchen("M","F");
        minDauer = System.nanoTime()-startzeit;
        for (int j=0; j<1000; j++){
           startzeit = System.nanoTime();
           g.WegeSuchen("M","F");
           dauer = System.nanoTime()-startzeit;
           if (dauer < minDauer) {
              minDauer = dauer;
            }
         }
         System.out.println(i + "; " + minDauer);
      }
   }
```

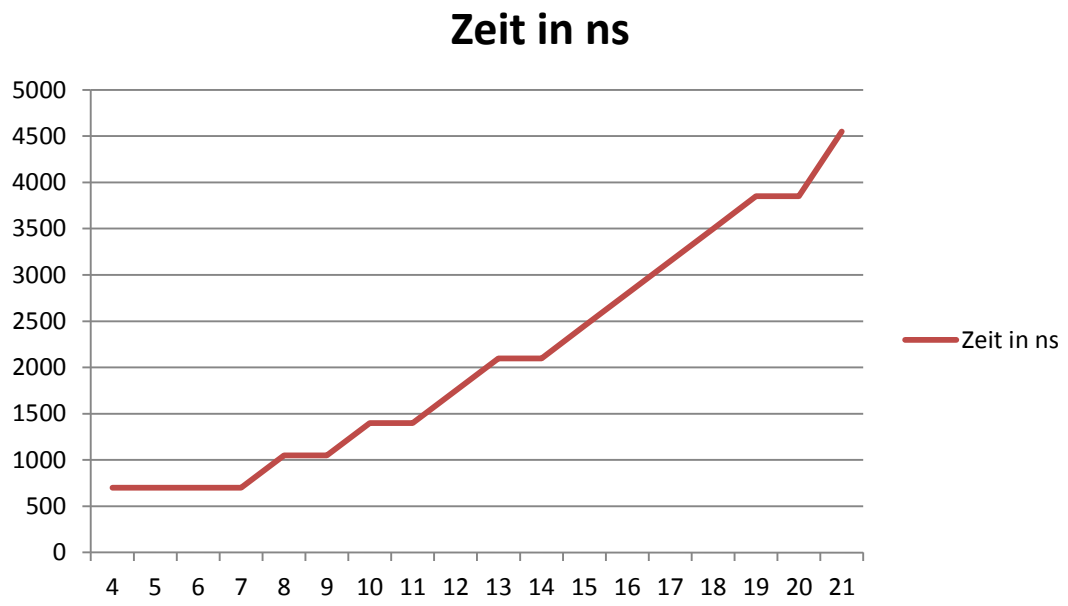
Anzahl Knoten	Zeit in ns
4	349
5	349
6	349
7	699
8	1749
9	1749
10	8398
11	8399
12	13646
13	26944
14	41640
15	42340
16	51438
17	95529
18	149767
19	191408
20	267691
21	325428



exponentieller Zeitbedarf:
$O(e^n)$

Kopiere den Inhalt der Methode BruteForceMessreihe in die Methode DijkstraMessreihe und ersetze g.WegeSuchen("M", "F") durch g.KuerzesterWeg("M", "F"). Nimm einen Messreihe auf und untersuche das Laufzeitverhalten.

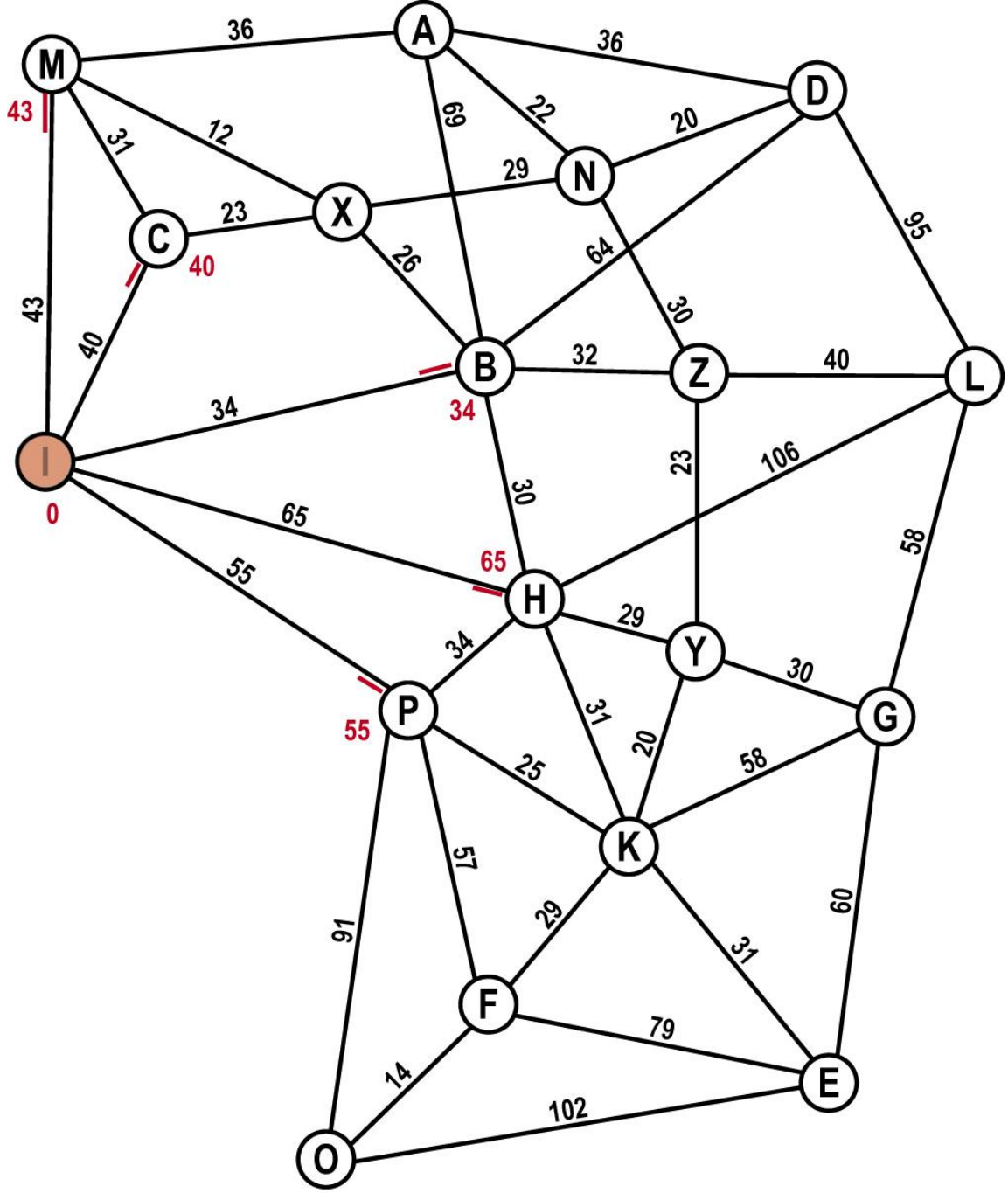
Anzahl Knoten	Zeit in ns
4	699
5	699
6	699
7	699
8	1049
9	1049
10	1399
11	1399
12	1749
13	2099
14	2099
15	2449
16	2799
17	3149
18	3499
19	3849
20	3849
21	4548



Laufzeitordnung von Dijkstra: $O(n^2)$

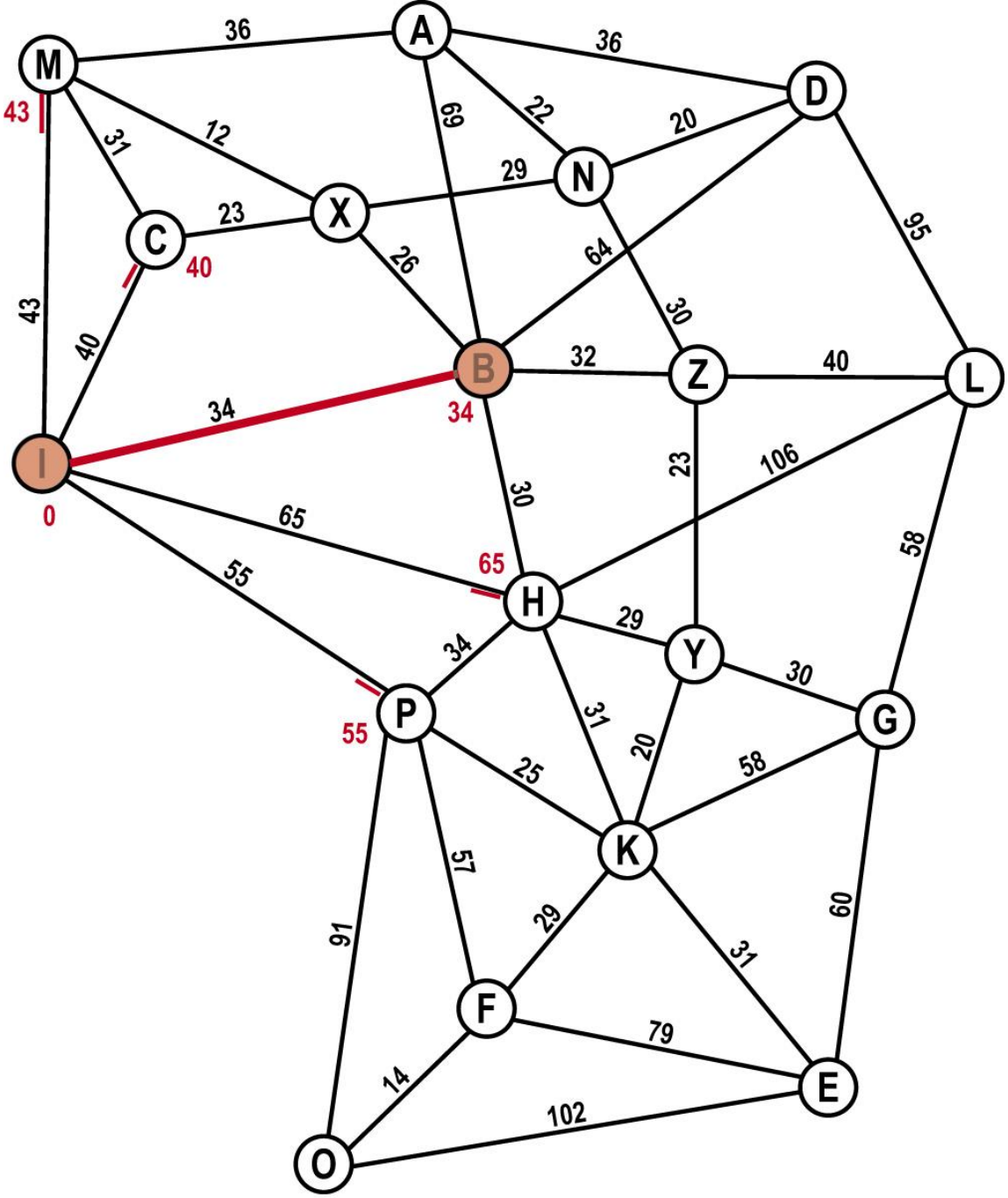
Europäisches Straßennetz: 10 Millionen Knoten

➔ Verbesserung:
Aufteilung in verschiedene Hierarchieebenen

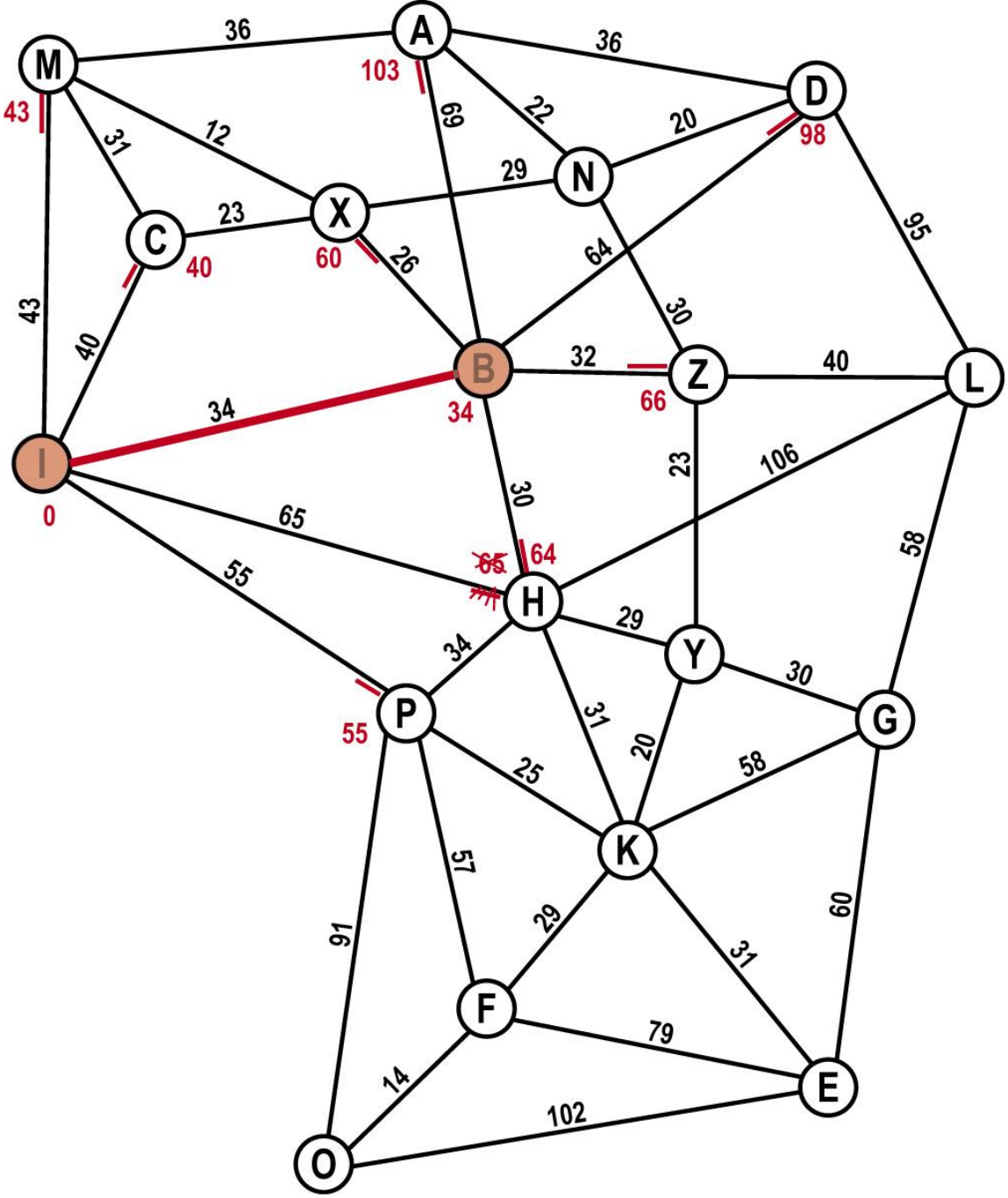


Aus: Gallenbacher, *Abenteuer Informatik, 2. Aufl.*

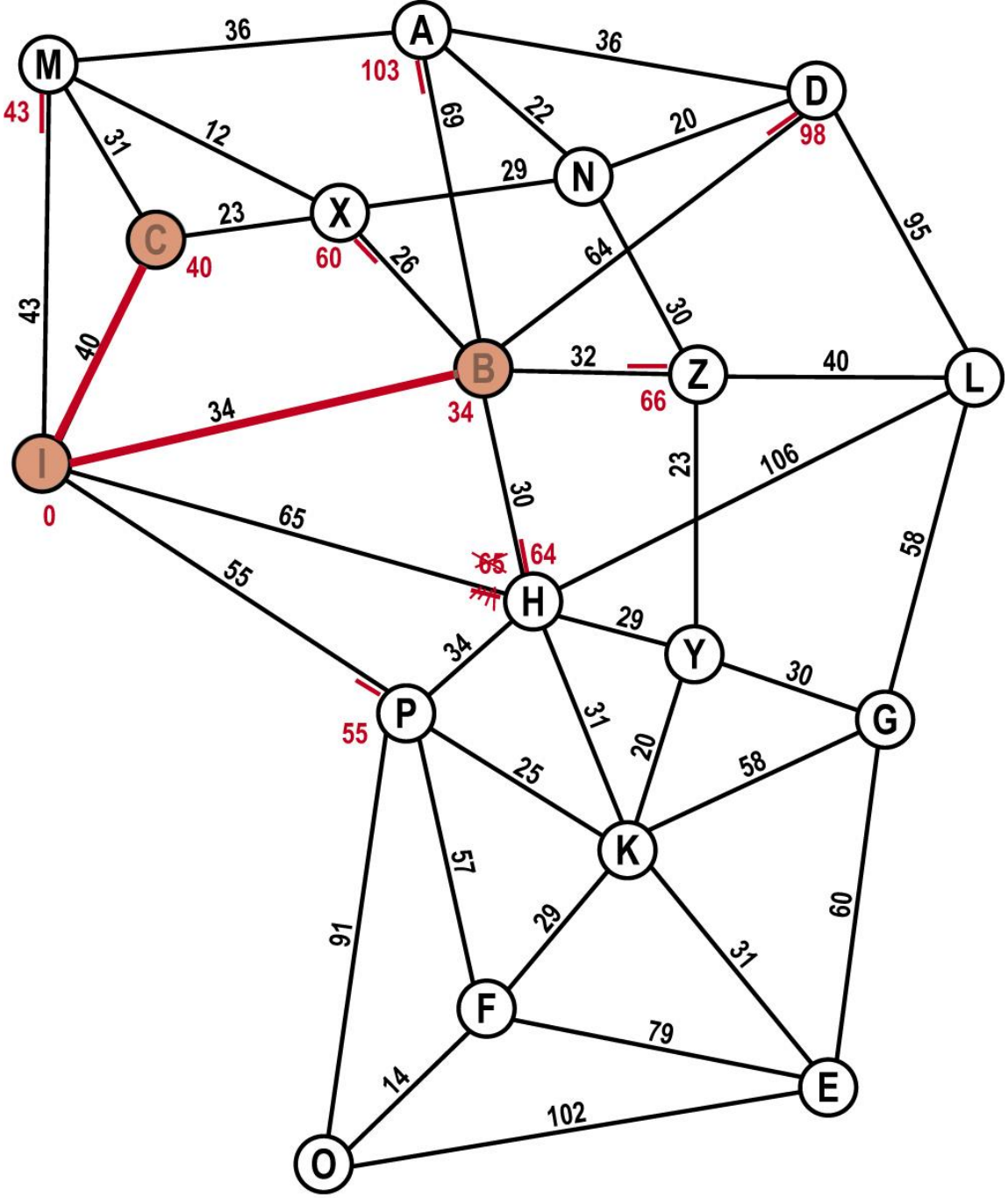
© Spektrum Akademischer Verlag GmbH 2008



Aus: Gallenbacher, *Abenteuer Informatik*, 2. Aufl.
© Spektrum Akademischer Verlag GmbH 2008



Aus: Gallenbacher, *Abenteuer Informatik, 2. Aufl.*
© Spektrum Akademischer Verlag GmbH 2008

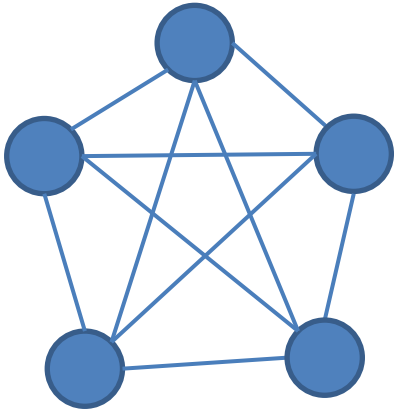


Aus: Gallenbacher, *Abenteuer Informatik, 2. Aufl.*
© Spektrum Akademischer Verlag GmbH 2008

Abschätzung:

- Extremfall 1: Linearer Graph

  1 Möglichkeit um von einem Knoten zu einem anderen zu gelangen.
  n Knoten werden besucht.



- Extremfall 2: Vollständiger Graph
  (n – 1)! mögliche Wege per Tiefensuche

- Realität:
  Von jedem Knoten gehen drei oder mehr Kanten aus.