4. Grenzen der Berechenbarkeit

4.1 Laufzeiten von Algorithmen

Die grundlegenden Anweisungen einer Programmiersprache, wie das Ausführen einfacher arithmetischer Operationen, die Wertzuweisung von Attributen oder das Erzeugen neuer Objekte benötigen nur einige Nanosekunden.
Deshalb scheinen viele Programme in Echtzeit abzulaufen.

Es gibt aber auch Programme, die wesentlich mehr Operationen ausführen müssen, wie z.B. die Suche nach Elementen, das Sortieren von Elementen oder die Suche kürzester Wege in gewichteten Graphen.

Deshalb haben Überlegungen zum Laufzeitverhalten von Algorithmen in der Softwareentwicklung große Bedeutung.

Messen von Laufzeiten:

Mit der folgenden Methode kann man die Laufzeit einer Methode in Nanosekunden bestimmen:

```
public long laufzeitTesten() {
   long startNanoTime;
   startNanoTime = System.nanoTime();

   //Hier muss die zu testende Methode aufgerufen werden.

   return  System.nanoTime() - startNanoTime;
}
```

Messfehler bei dieser Methode:

- Die Wertzuweisung von startNanoTime und der Aufruf der Methode nanoTime() benötigen ebenfalls eine bestimmte Laufzeit.
- Der Ablauf des Programms kann durch andere Prozesse im Betriebssystem beeinflusst werden. Aus diesem Grund führt man mehrere Messungen durch und bildet nicht den Mittelwert sondern das Minimum aller Laufzeiten.

Qualitative Laufzeitbestimmung durch Zählen der Schritte:

Man verwaltet ein Attribut das die auszuführenden Schritte zählt und erhöht den Attributwert nach jeder Anweisung im Programm um 1.
Auf diese Weise erhält man eine rechnerunabhängige Aussage über die das Laufzeitverhalten.

```
public class Schrittzaehlung{
   public void ggTBerechnen(int a, int b){
      int i; //Anzahl der Schritte
      i = 0;
      i++;
      while(a!=b){
         i++;
         if(a>=b){
            i++;
            a = a-b;
         }
         else{
            i++;
            b = b-a;
         }
      }
      i++;
      System.out.println("Der ggT ist "+a);
      System.out.println("Zur Berechnung waren "+i+" Schritte erforderlich");
   }
}
```

Übung

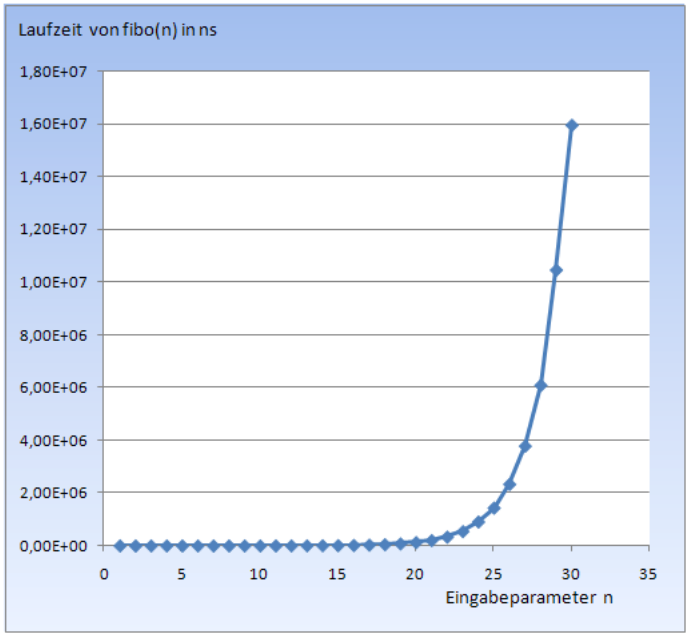
Teste das Programm Laufzeiten_bestimmen für verschiedene Methoden.

4.1 Laufzeiten von Algorithmen

Die Laufzeit eines Programms ist natürlich auch von den Eingabeparametern abhängig.

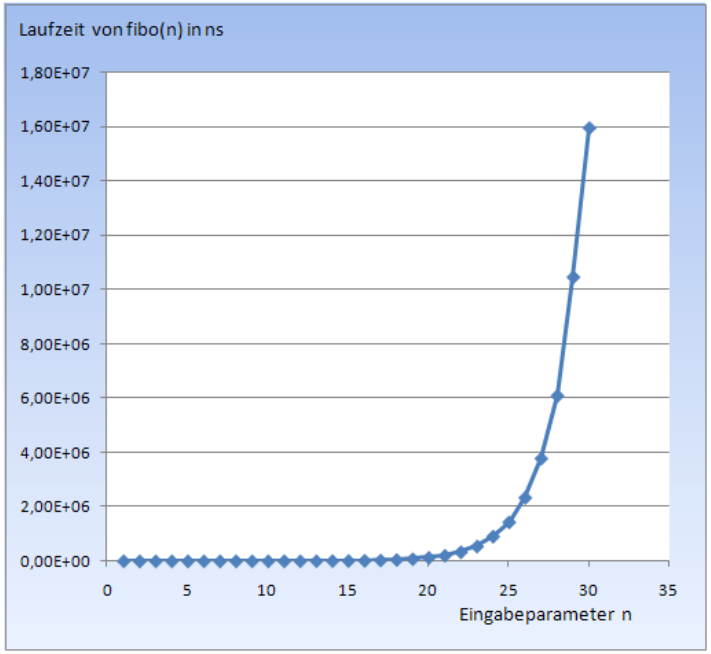
Führt man eine Messreihe durch, kann man die Ergebnisse zum Beispiel mit Hilfe einer Tabellenkalkulation graphisch darstellen.

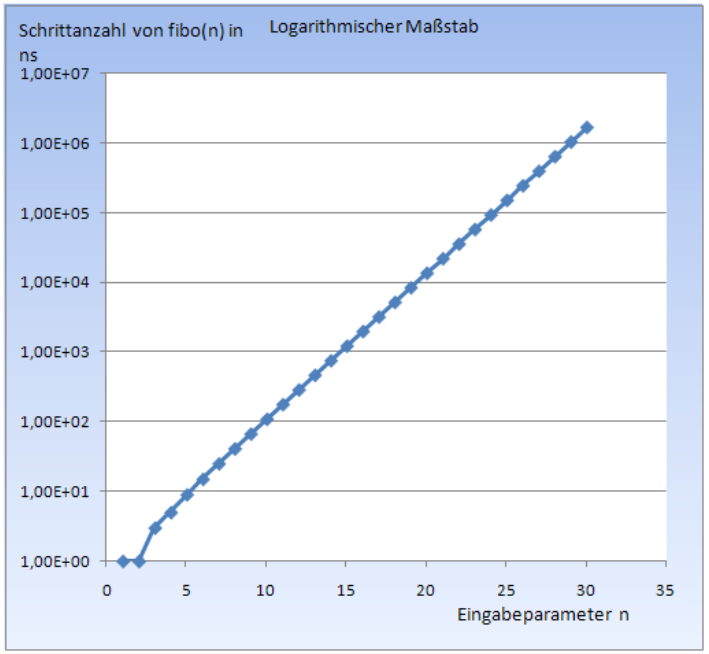
Für die rekursive Methode fibo(int n) zur Bestimmung der n-ten Fibonaccizahl erhält man folgendes Laufzeitverhalten :



Aufgrund der nicht linearen Rekursion (jeder rekursive Aufruf hat zwei weitere Aufrufe zur Folge) steigt die Laufzeit mit zunehmenden n stark, in diesem Fall exponentiell an.

Das exponentielle Laufzeitverhalten lässt sich nachweisen, indem man auf der y-Achse den Logarithmus der Laufzeit auftägt.
Wegen $\log(a \cdot b^x) = \log(a) + x \cdot \log(b)$ ergibt sich in diesem Diagramm eine Gerade.





Übung

Schreibe eine Methode, mit der man eine Messung der Laufzeit für die Fibonaccizahlen durchführen kann und werte die Ergebnisse in einer Tabelle aus. Verwende die Vorlage messung_laufzeit.xls

Schreibe auch ein Programm, das die Anzahl der Schritte bei der Berechnung der Fibonaccizahlen ausgibt.

Übung

Schreibe eine Methode, die die Binomialkoeffizienten $bin(n;k)$ rekursiv berechnet und untersuche das Laufzeitverhalten in Abhängigkeit von n.

Für die Binomialkoeffizienten gilt:

$$bin(n;k) = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Rekursionsvorschrift:

$$bin(n;k) = \begin{cases} 1 & wenn\ \ k = 0 \quad oder\ k = n \\ bin(n-1;k-1) + bin(n-1;k) & sonst \end{cases}$$

Übung

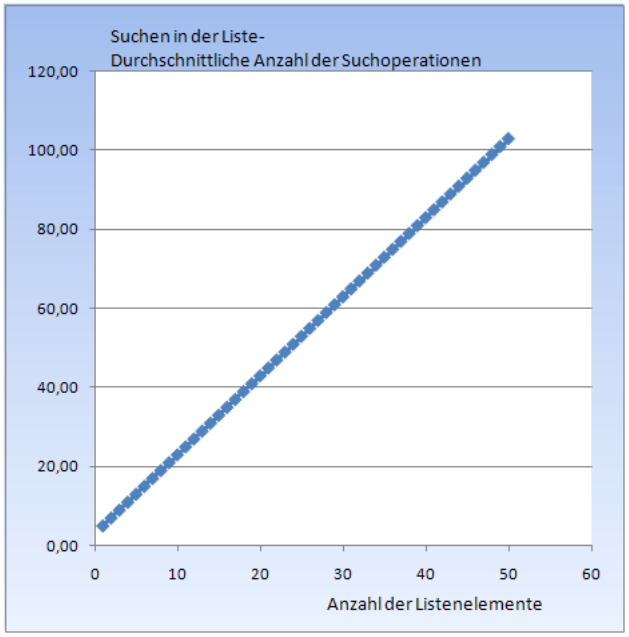
Suche in Liste und Baum (S. 138, Aufgabe 4)

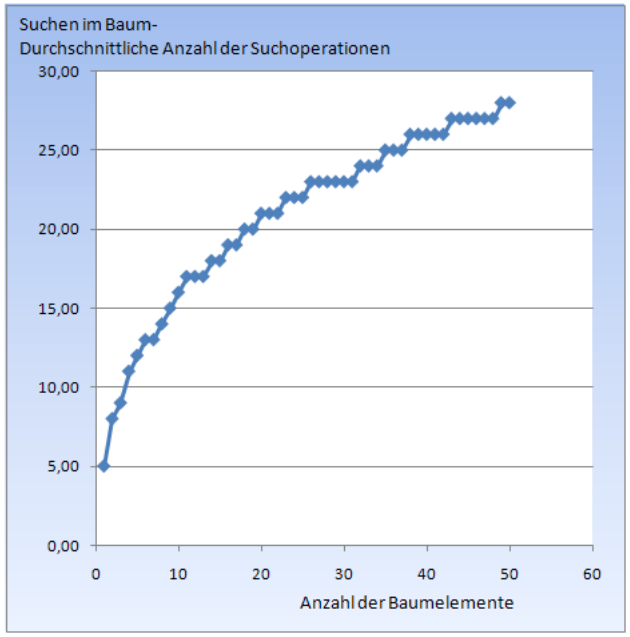
a) Machen Sie sich mit dem gegebenen Projekt „Suche” vertraut.
b) Entscheiden Sie sich für eines der beschriebenen Verfahren zur Laufzeitabschätzung. Wenden Sie das Verfahren auf die Suche in der Liste und die Suche im Baum an.
c) Testen Sie die Suche in der Liste auf den Laufzeitaufwand bei steigender Knotenzahl. Visualisieren Sie in einem Diagramm.
d) Führen Sie entsprechende Messungen für die Suche im geordneten Binärbaum durch.
e) Der Vergleich von Liste und Baum ergibt nur dann einen Sinn, wenn der Baum ausgeglichen und nicht zur Liste entartet ist. Informieren Sie sich, wie ein solcher balancierter Baum erzeugt werden kann (mögliches Stichwort: AVL-Baum).

(Hinweis: Das Programm gibt die Ergebnisse vom Typ float mit Punkt als Dezimalkomma aus.
Dies wird evtl. vom Tabellenkalkulationsprogramm nicht als Zahl erkannt.
In diesem Fall ändert man den Ausgabetyp besser auf integer.)

Ergebnisse

Suche in Liste und Baum (S. 138, Aufgabe 4)





Übung

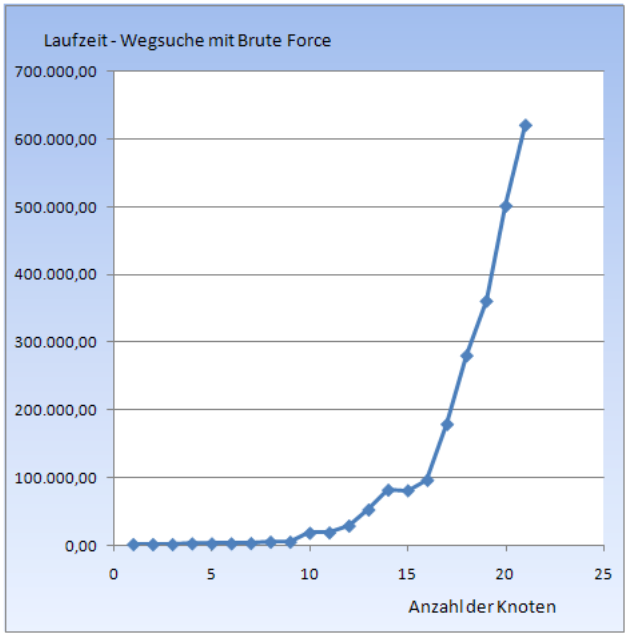
Wegsuche in Graphen(S. 138, Aufgabe 3)

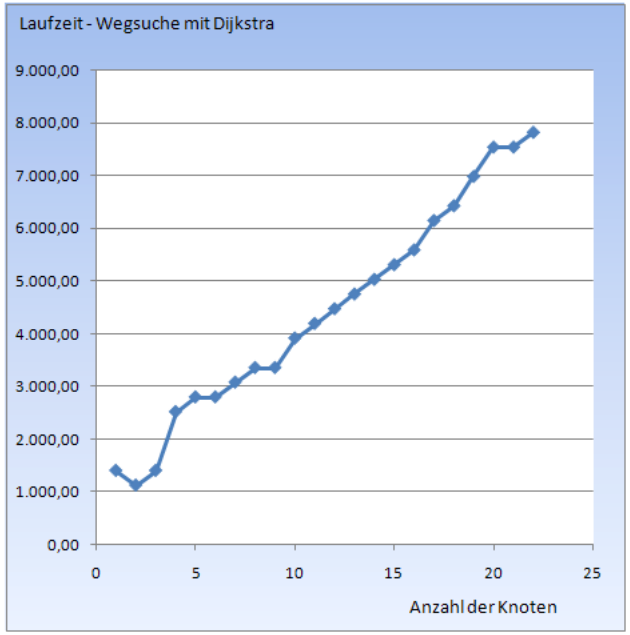
a) Machen Sie sich mit dem gegebenen Projekt „Routenplaner” vertraut.
b) Entscheiden Sie sich für eines der beschriebenen Verfahren zur Laufzeitabschätzung. Wenden Sie das Verfahren auf den Routenplaner an, der nach dem Brute-Force-Verfahren arbeitet. Testen Sie den Routenplaner auf den Laufzeitaufwand bei steigender Knotenzahl. Visualisieren Sie in einem Diagramm.
c) Führen Sie die entsprechende Untersuchung unter Verwendung des Dijkstra-Algorithmus durch.

d) Der einfache Dijkstra-Algorithmus hat bei einem Straßennetz vom Umfang Europas eine zu hohe Laufzeit. Informieren Sie sich, wie der Algorithmus weiter optimiert werden kann (mögliche Stichworte: Highway Hierarchy, Fibonacci Heap, zweiseitige Annäherung).

Ergebnisse

Wegsuche in Graphen(S. 138, Aufgabe 3)





Übung

Passwörter knacken(S. 140, Aufgabe 8)

a) Testen Sie das gegebene Programm zum Knacken von Passwörtern nach dem Brute-Force-Verfahren in Bezug auf die Laufzeit bei variabler Passwortlänge und unterschiedlichem Zeichenvorrat. Stellen Sie die Ergebnisse in geeigneten Diagrammen dar.
b) Informieren Sie sich über weitere Maßnahmen zum Passwortschutz.

c) Suchen Sie im Internet nach Passwort-Wörterbüchern. Untersuchen Sie die darin vorkommenden Begriffe.

Brute-Force-Verfahren

Ein Verfahren, das systematisch alle Möglichkeiten ausprobiert,
heißt **Brute-Force-Verfahren.**

Brute Force Verfahren bei der Wegsuche in einem Graphen:

- Man ermittelt alle zyklenfreien Wege im Graphen vom Start- zum Zielknoten.
- Aus dieser Menge bestimmt man den Weg mit der kürzesten Weglänge.

Beim Verschlüsseln von Texten und bei Passwörtern achtet man darauf, dass eine Entschlüsselung mit Hilfe des Brute Force Verfahrens aufgrund eines hohen Laufzeitaufwands nicht realisierbar ist.

Die **Landau-Notation** für das Laufzeitverhalten:
n steht dabei für einen Eingabeparameter, z.B. die Anzahl der Knoten in einem Baum.

$\mathcal{O}(a^n)$: exponentielles Laufzeitverhalten (Bsp.: Fibonacci-rekursiv, Brute Force bei Graphen)

$\mathcal{O}(n^2)$: quadratisches Laufzeitverhalten (Bsp.: Bubblesort oder Insertionsort Algorithmus)

$\mathcal{O}(n \cdot \log(n))$: super lineares Laufzeitverhalten (Bsp.: Mergesort Algorithmus)

$\mathcal{O}(n)$: lineares Laufzeitverhalten (Bsp.: Suchen in einer Liste)

$\mathcal{O}(\log(n))$: logarithmisches Laufzeitverhalten (Bsp.: Suchen in einem Baum)

$\mathcal{O}(p(n))$: polynomiales Laufzeitverhalten. $p(n)$ steht dabei für eine Polynomfunktion (d.h. ganzrationale Funktion)

Die Schreibweise $\mathcal{O}(f(n))$ beschreibt dabei eine so genannte Komplexitätsklasse.
Man sagt, ein Algorithmus hat die Komplexität $\mathcal{O}(f(n))$, wenn die Funktion $f(n)$, die dessen Laufzeitverhalten für große n beschreibt, zu dieser Klasse gehört.

Bedeutung der Schreibweise $\mathcal{O}(f(n))$ für die Laufzeit eines Algorithmus:

Das Laufzeitverhalten $\mathcal{O}(f(n))$ bedeutet, dass die Laufzeit $t(n)$ proportional zu $f(n)$ ist.
Es gilt also:

$$t(n) = k \cdot f(n)$$

oder auch:

$$\frac{t(n_2)}{t(n_1)} = \frac{f(n_2)}{f(n_1)}$$

$$t(n_2) = t(n_1) \cdot \frac{f(n_2)}{f(n_1)}$$

Kennt man die Laufzeit $t(n_1)$ für einen Wert n_1, lassen sich auf diese Weise weitere Laufzeiten abschätzen.

Übung

Rechnen mit Laufzeiten (S. 140, Aufgabe 7)

Ein Algorithmus benötigt für die Bearbeitung bei $n = 100$ Elementen die Zeit $t = 0{,}1\ s$.

a) Wie lange benötigt er für $110, 120, 130, 140, 150, 200, 300, \ldots, 1000$ Elemente, wenn der Laufzeitaufwand gemäß dem folgenden Term zunimmt:
i) $\log_2(n)$ ii) n iii) $n \cdot \log_2(n)$ iv) 3^n v) $n!$
Hinweis: Tabellenkalkulation verwenden!

b) Schätzen Sie ab, bei wie vielen Elementen der Algorithmus mit einer Laufzeitzunahme gemäß 3^n die folgende Laufzeit benötigt:
$0{,}1\ s$ $1\ s$ $10\ s$ $100\ s$
eine Stunde einen Tag ein Jahrhundert 13,7 Mrd. Jahre (Alter des Universums)

c) Versuchen Sie zum Vergleich eine analoge Berechnung aus b) für einen Algorithmus mit logarithmischem Laufzeitaufwand $\log_2(n)$.

Übung

Laufzeiten bei Sortieralgorithmen – Teil 1

1. Öffne das BlueJ-Projekt SelectionSort und überlege anhand der Ausgabe und des Quelltextes, wie der Algorithmus das Array sortiert.

2. Öffne das Programm VisualSort und führe den Sortieralgorithmus SelectionSort (Sortieren durch Auswahl) aus. Welche Vorbelegung ist für den Algortihmus bezüglich der Laufzeit der „best case“ und welche ist der „worst case“?

3. Führe im Programm VisualSort einige Laufzeitmessreihen durch. Welche Funktion beschreibt das Laufzeitverhalten?

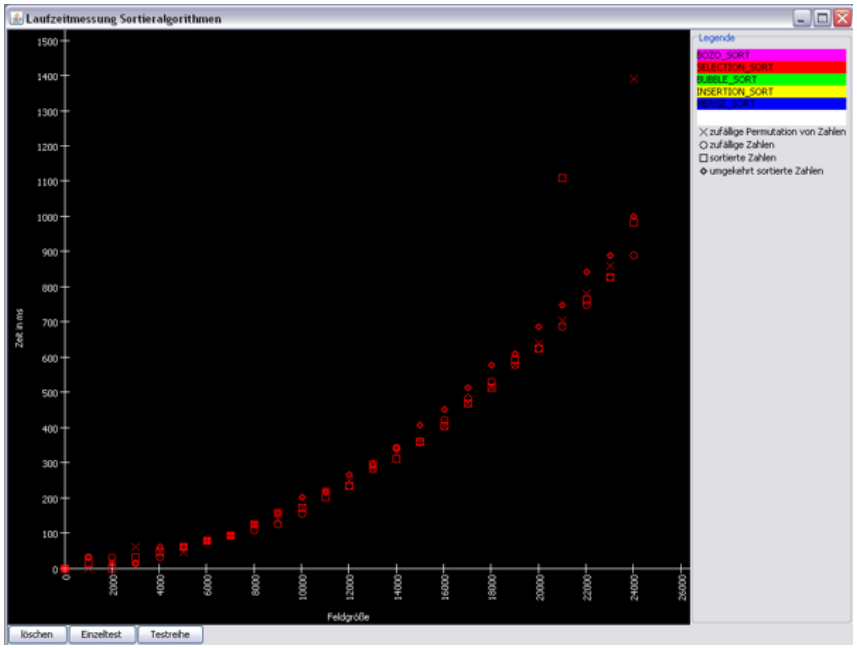
4. Nur bei Interesse (kein Abi-Stoff): Auf den Seiten von MathePrisma findest du eine Herleitung für die Abschätzung des Laufzeitverhaltens.

5. Führe im Programm VisualSort die Sortieralgorithmen InsertSort (Sortieren durch Einfügen) und BubbleSort aus und untersuche wie bei SelectionSort mit einer Laufzeitmessreihe das Laufzeitverhalten.

6. Schreibe in Java jeweils eine Klasse, welche die Suchalgorithmen InsertSort und BubbleSort umsetzt.

Laufzeiten bei Sortieralgorithmen - Ergebnisse

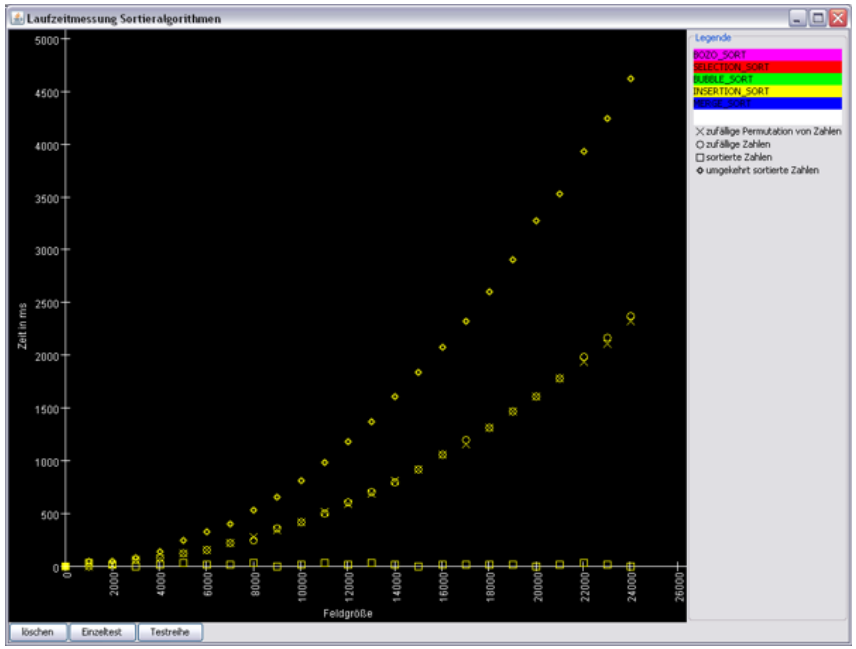
SelectionSort:



worst case: $\mathcal{O}(n^2)$

best case: $\mathcal{O}(n^2)$

InsertSort:

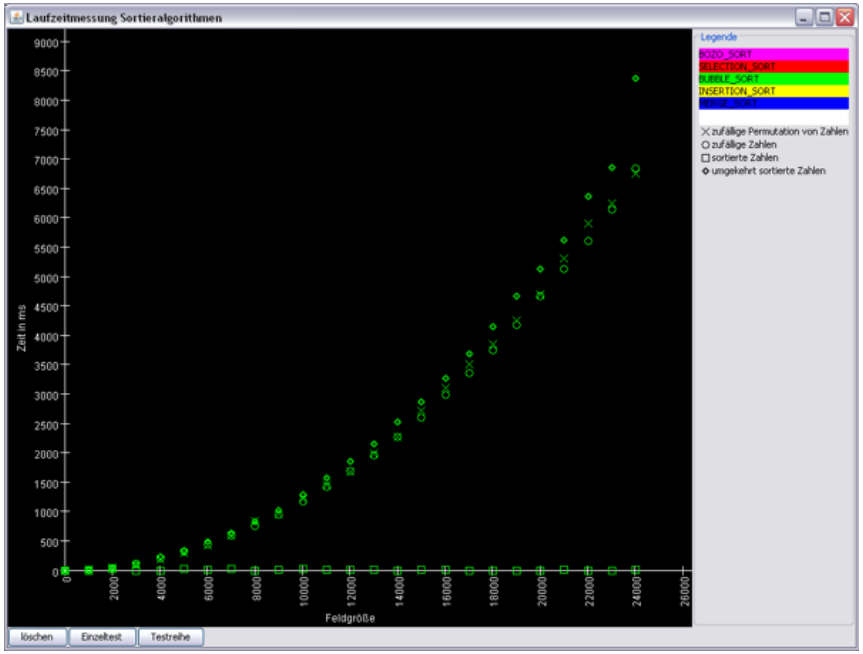


worst case: $\mathcal{O}(n^2)$

best case: $\mathcal{O}(n)$

Laufzeiten bei Sortieralgorithmen - Ergebnisse

BubbleSort:



worst case: $\mathcal{O}(n^2)$

best case: $\mathcal{O}(n)$

Übung

Laufzeiten bei Sortieralgorithmen – Teil 2

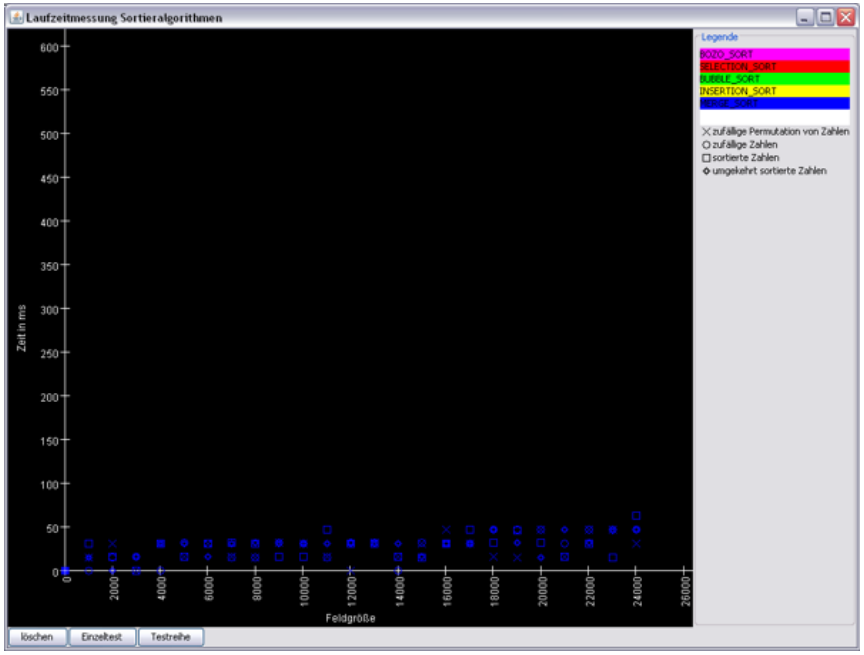
1. Öffne das Programm VisualSort und führe den Sortieralgorithmus MergeSort aus.
   Beschreibe, wie der Algorithmus das Feld sortiert.

2. Führe im Programm VisualSort einige Laufzeitmessreihen durch und vergleiche mit SelectionSort, InsertSort und BubbleSort.

3. Untersuche auf den Seiten von MathePrisma den Sortieralgorithmus QuickSort, der ähnlich wie MergeSort nach dem Prinzip **divide and conquer** arbeitet. http://www.matheprisma.uni-wuppertal.de/Module/Sortieren/index.htm

Laufzeiten bei Sortieralgorithmen - Ergebnisse

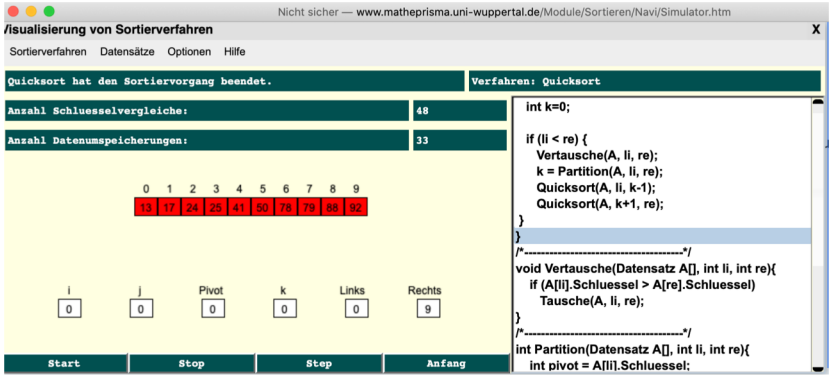
MergeSort:



worst case: $\mathcal{O}(n \cdot \log(n))$

best case: $\mathcal{O}(n \cdot \log(n))$

QuickSort:



http://www.matheprisma.uni-wuppertal.de/Module/Sortieren/index.htm

worst case: $\mathcal{O}(n^2)$

best case: $\mathcal{O}(n \cdot \log(n))$