```
double[ ] average(int k, double a[ ]) {
  int n = a.length;
  assert n > k;
  double sum[ ] = new double[n];
  sum[0] = a[0];
  for (int i = 1; i < n; i++) {
    sum[i] = sum[i − 1] + Math.abs(a[i]);
  }
  double mittel[ ] = new double[n];
  for (int i = k; i < n; i++) {
    mittel[i] = (sum[i] − sum[i − k])/k;
  }
  for (int i = 0; i < k; i++) {
    mittel[i] = mittel[k];
  }
  return mittel;
}
```

```
double[ ] threshold(double a[ ], double mean[ ]) {
  assert mean.length == 2;
  int n = a.length;
  double result[ ] = new double[n];
  int diff = 0;
  for (int i = 0; i < n; i++) {
    if (Math.abs(mean[0] − a[i]) < Math.abs(mean[1] − a[i])) {
      diff = diff − 1;
    } else {
      diff = diff + 1;
    }
    final int d = 10;
    if (diff > d) {diff = d; result[i] = 1; }
    if (diff < −d) {diff = −d; result[i] = 0; }
  }
  return result;
}
```

```
List<Integer> cummulate(double a[ ]) {
  int n = a.length, count = 0;
  List<Integer> result = new ArrayList<Integer>();
  for (int i = 1; i < n; i++) {
    if((a[i] > 0) != (a[i − 1] > 0)) {
      result.add(count);
      count = 0;
    }
    if(a[i] > 0) {
      count = count + 1;
    }
    else {
      count = count − 1;
    }
  }
  result.add(count);
  return result;
}
```

```
String symbolize(List<Integer> len) {
  String s = "";
  double pulse[ ] = kMeans.getCenters(2, signFilter(+1, len));
  double pause[ ] = kMeans.getCenters(2, signFilter(−1, len));
  for(int dur : len) {
    if(dur < 0) {
      if(Math.abs(pause[0] − dur) < Math.abs(pause[1] − dur)) {
        s = s + "  ";
      }
    }
    else {
      if(Math.abs(pulse[0] − dur) < Math.abs(pulse[1] − dur)) {
        s = s + ".";} else {s = s + "−";}
    }
  }
  return s;
}
```

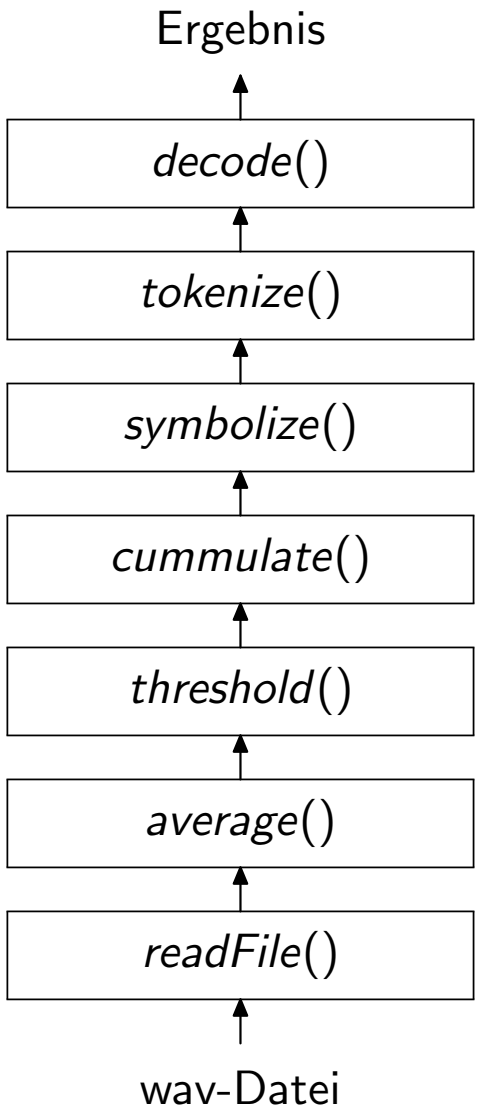
```
double[ ] signFilter(int sign, List<Integer> list) {
  List<Integer> result = new ArrayList<Integer>();
  for (int n : list) {
    if ((sign > 0) == (n > 0)) {
      result.add(n);
    }
  }
  double r[ ] = new double[result.size()];
  for(int i = 0; i < r.length; i++) {
    r[i] = result.get(i);
  }
  return r;
}
```

```
List<String> tokenize(String dotdashs) {
  List<String> tokens = new ArrayList<String>();
  String z = "";
  for (int i = 0; i < dotdashs.length(); i++) {
    if (dotdashs.charAt(i) == ' ') {
      tokens.add(z);
      z = "";
    } else {
      z = z + dotdashs.charAt(i);
    }
  }
  tokens.add(z);
  return tokens;
}
```

```
String decode(List<String> tokens) {
    String result = "";
    for (String morseToken : tokens) {
        result += translateMorseCodeToChar(morseToken);
    }
    return result;
}
```

```
char translateMorseCodeToChar(String code) {
    Character c = morseTable.get(code);
    if(c == null) {
        return '?';
    }
    else {
        return c;
    }
}
```

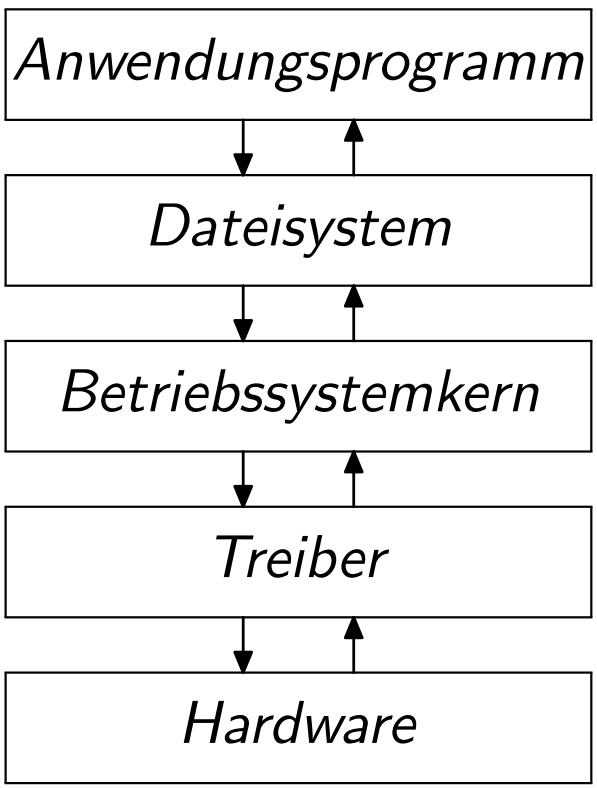
Pipelines, um schrittweise zum Ergebnis zu kommen.



Programmieren in Schichten

Pipeline: Die Daten laufen in einer Richtung.

Verallgemeinerung: Viele Schichten, die immer mehr abstrahieren.



Neues Kapitel

1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. Objektorientiertes Design
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

5 Effiziente Programme

- Schnelle und langsame Programme
- Speicherplatz

Wann ist ein Programm schnell?

Die Geschwindigkeit von Programmen ist schwer zu vergleichen.

- Welcher Computer?
- Welcher Compiler?
- Größe des Speichers?
- Cache-Speicher?
- Welche anderen Programme laufen?
- etcetera, etcetera.

Die Geschwindigkeit eines Programms kann auch vom Wetter abhängen.

Schnelle Programme

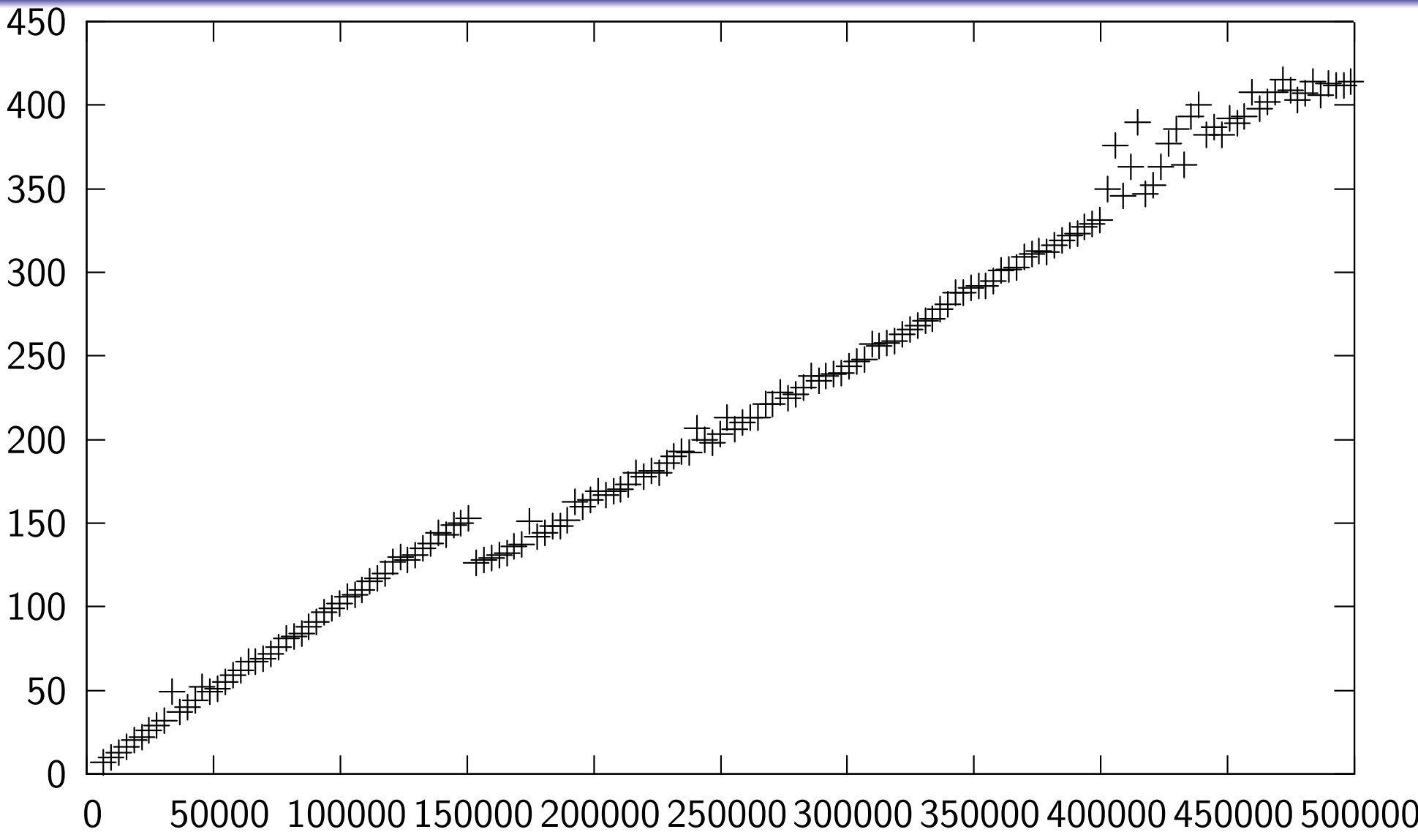
Wir können Programme durch „Tuning“ verbessern.

Dies lohnt sich aber nur bei den Teilen, welche die meiste Zeit verbrauchen: „innere Schleifen“

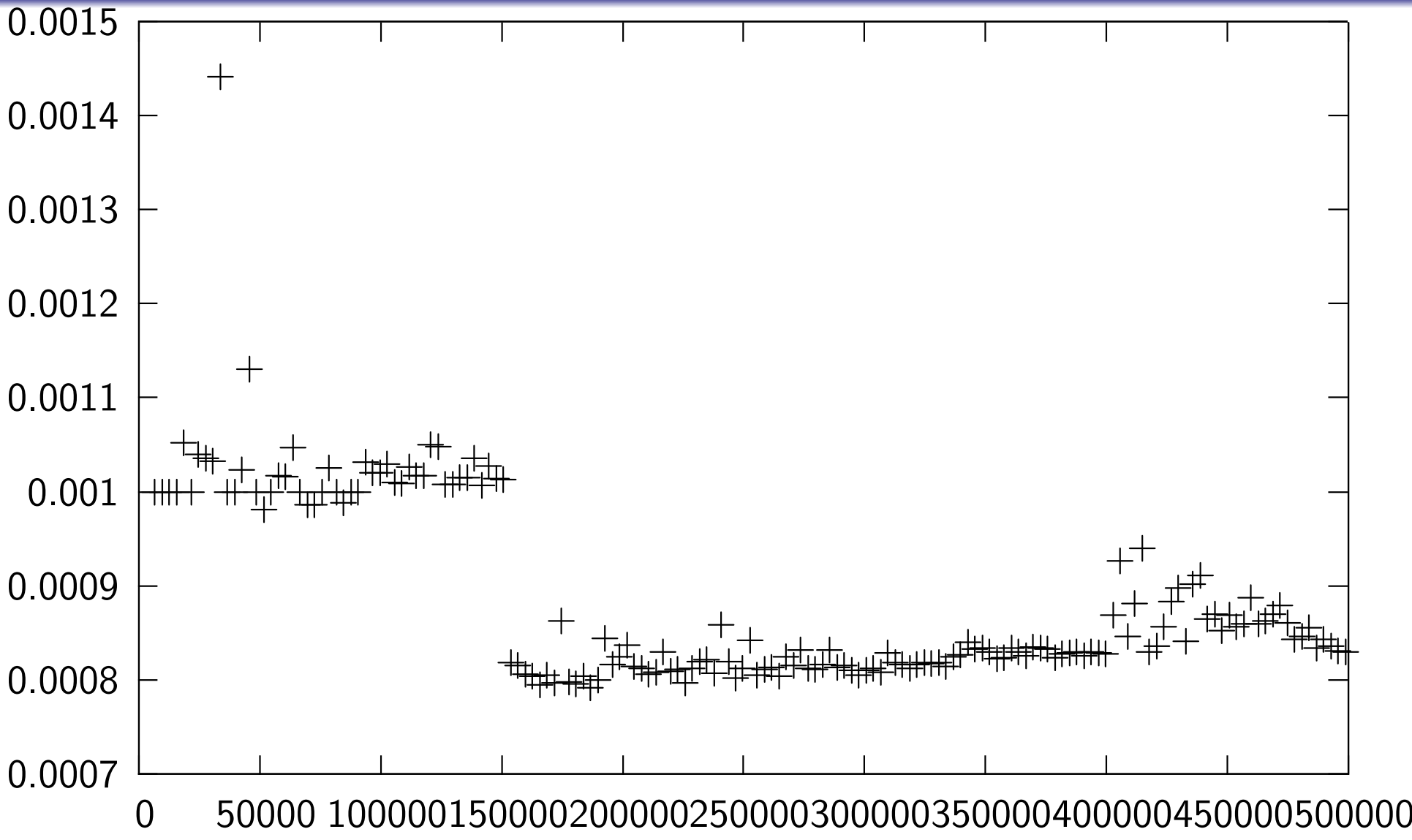
Und:

> *Premature optimization is the root of all evil.*
> *Donald Knuth*

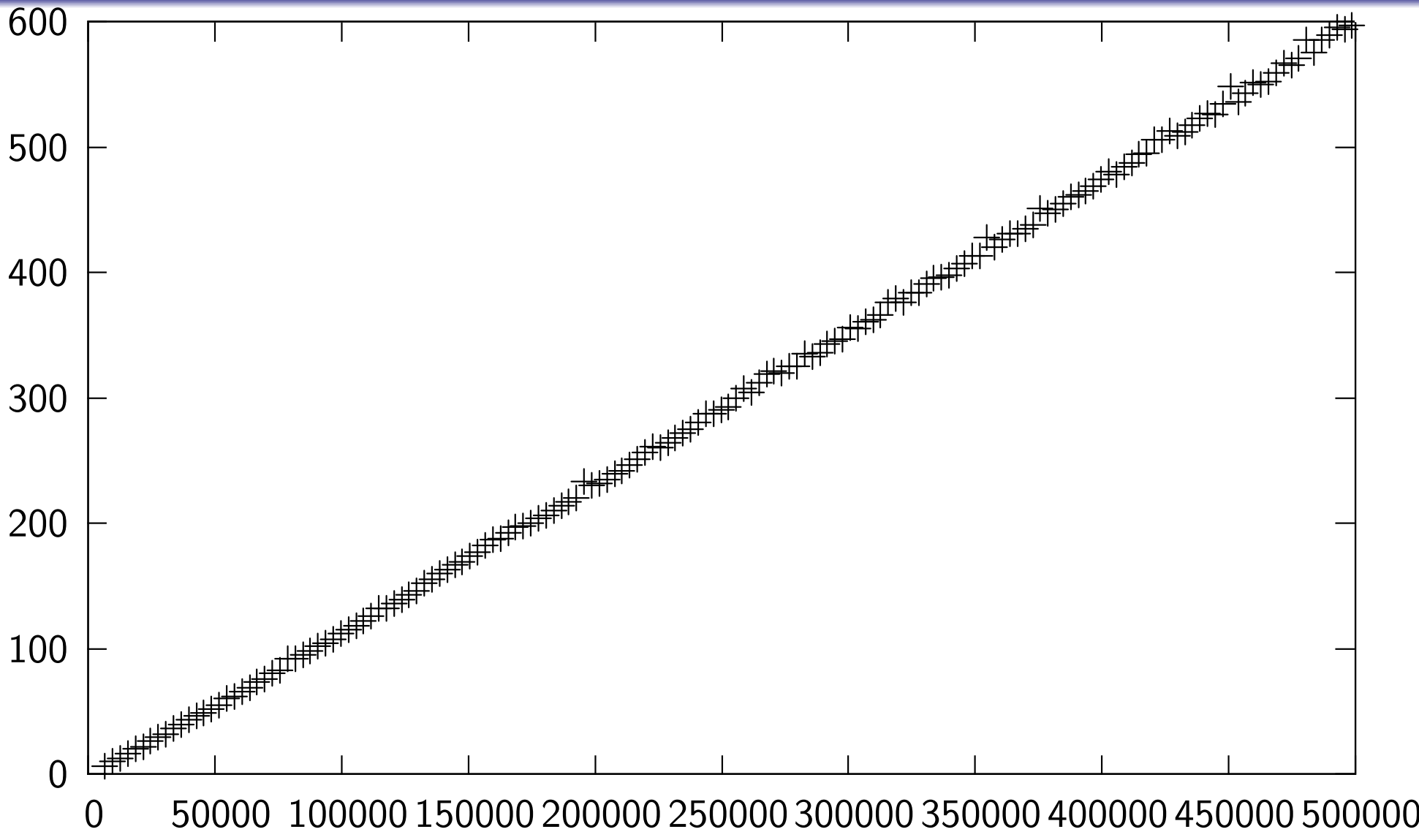
Ein überlegener Algorithmus kann die bessere Wahl sein.



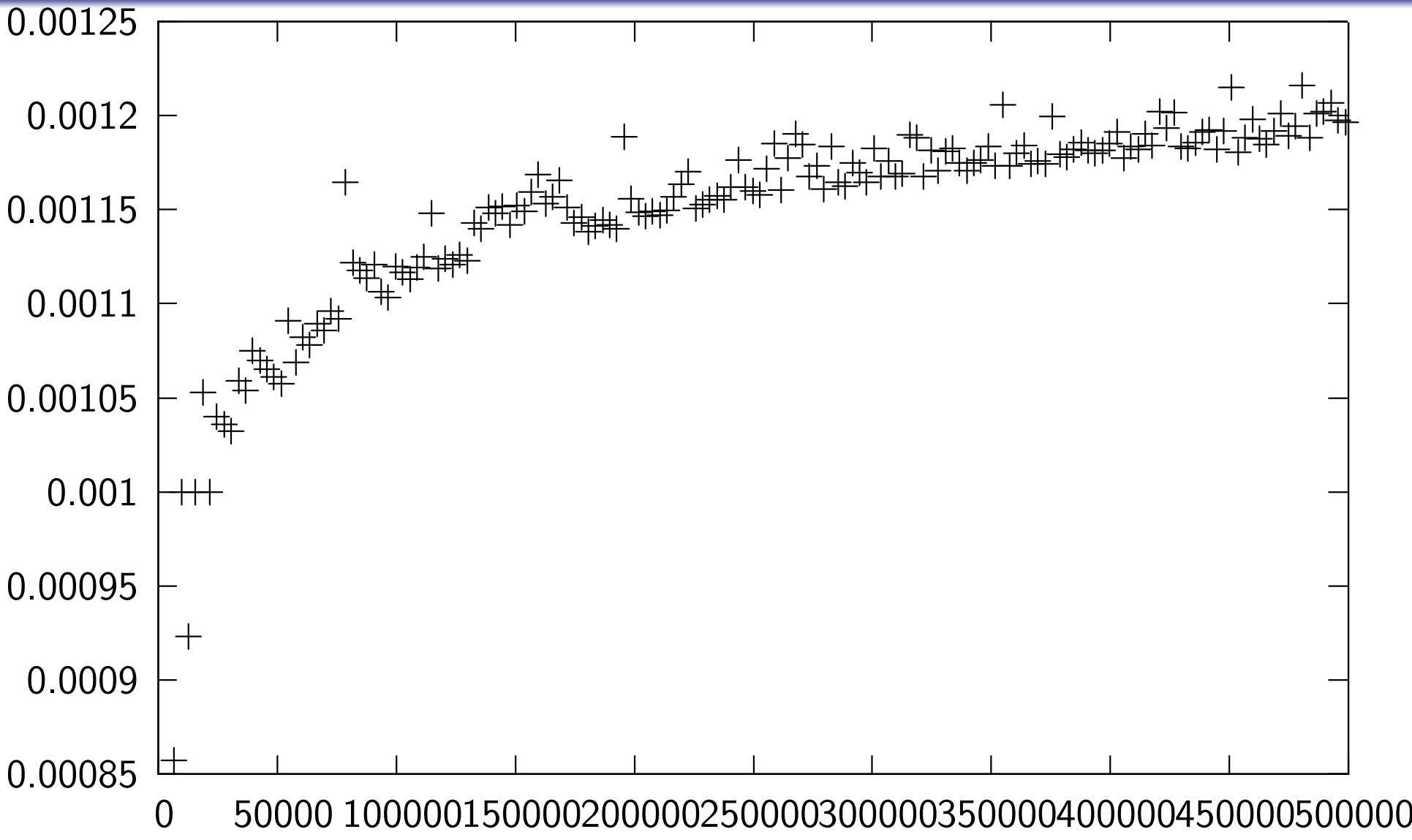
Zehnmalige Wiederholung von Quicksort.



Laufzeit *pro* Datenelement.

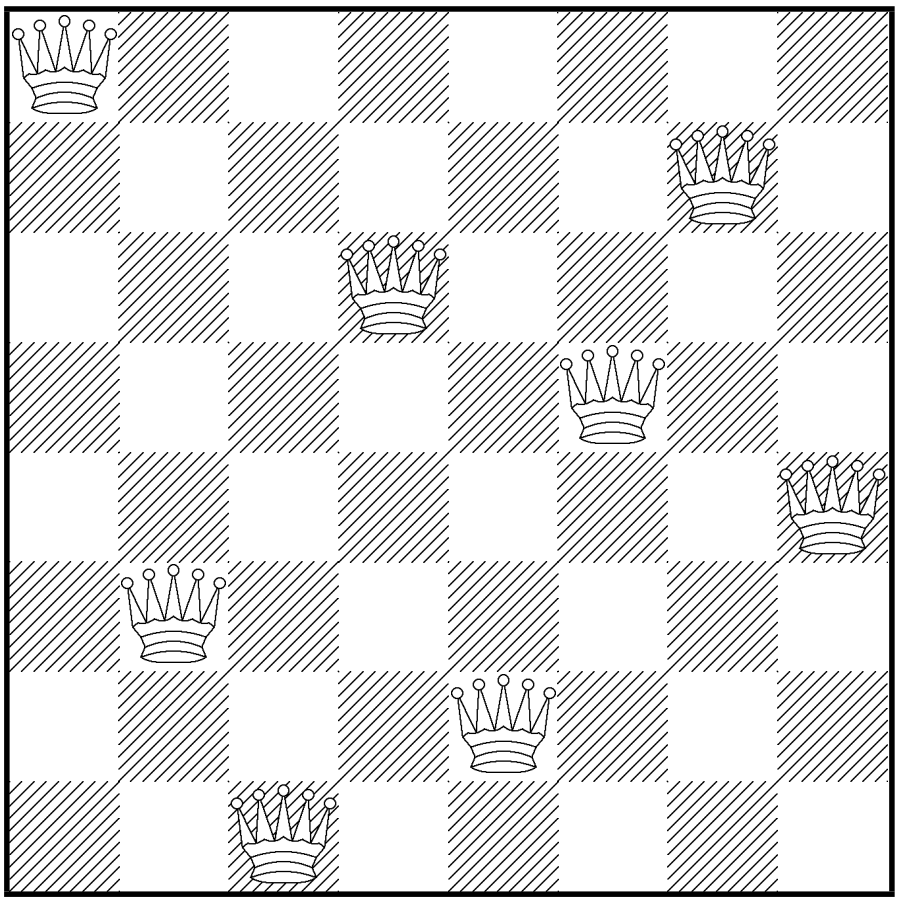


Zehnmalige Wiederholung von Heapsort.



Laufzeit *pro* Datenelement.

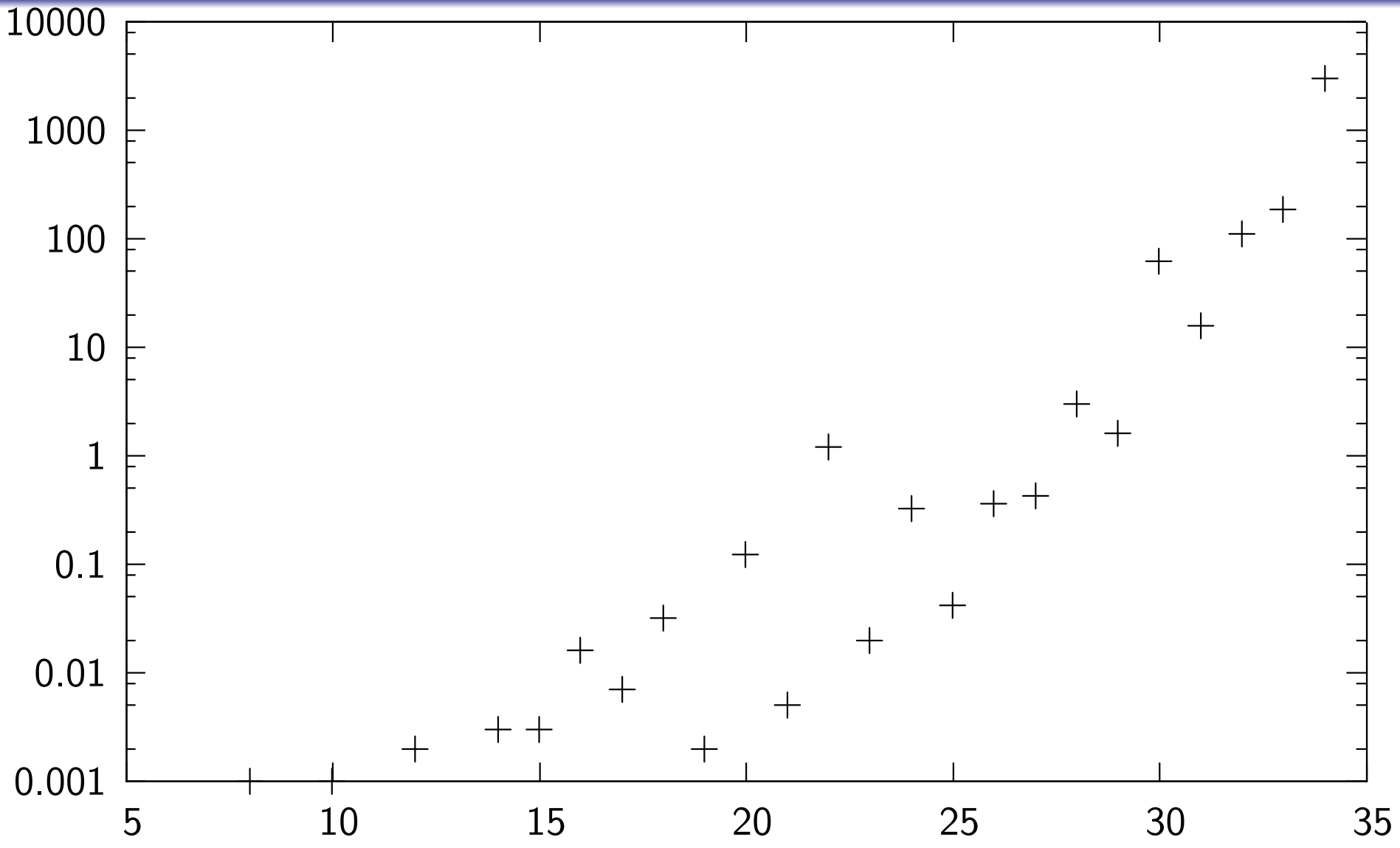
Das *n*-Damenproblem



Wie können wir das *n*-Damenproblem *schneller* lösen?

Dies war unser Programm:

```
void setzeDamenAbZeile(int y) {
  if(y >= n) {
    geloest = true;
    return;
  }
  for(int x = 0; x < n; x++) {
    if(safePosition(y, x)) {
      brett[y][x] = true;
      setzeDamenAbZeile(y + 1);
      if(geloest) {
        return;
      }
      brett[y][x] = false;
    }
  }
}
```



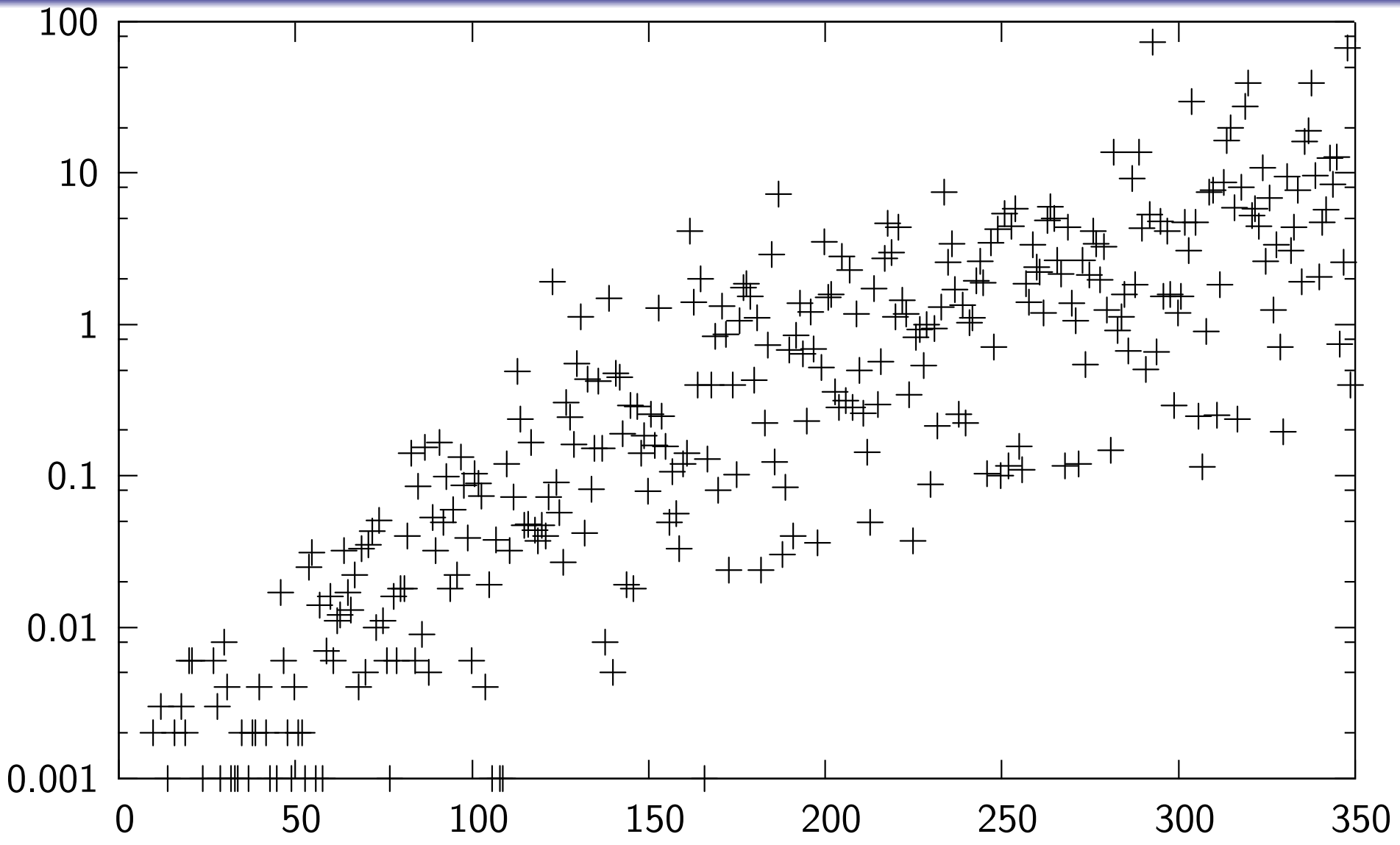
Laufzeit des n-Damenproblems.

```
void setzeDamenAbZeile(int y, int totallives) {
  lives = totallives;
  if(y >= n) {
    geloest = true;
    return;
  }
  lives = lives - 1;
  if(lives < 0) {return;}
  int offset = generator.nextInt(n);
  for(int k = 0; k < n; k++) {
    int x = (k + offset)%n;
    if(safePosition(y, x)) {
      brett[y][x] = true;
      setzeDamenAbZeile(y + 1, lives);
      if(geloest) {return;}
      brett[y][x] = false;
    }
  }
}
```

Zwei Innovationen:

- Nach gewisser Zeit aufgeben und neu anfangen.
- Die Damen in zufälliger Reihenfolge setzen.

```
void setzeDamenAbZeile(int y) {
  int totallives = 1000;
  while(!geloest) {
    for(int yy = y; yy < n; yy++) {
      for(int x = 0; x < n; x++) {
        brett[yy][x] = false;
      }
    }
    setzeDamenAbZeile(y, totallives);
    totallives += 100;
  }
}
```



Laufzeit des n-Damenproblems mit Wiederstart.