Einführung und Hügel

- Algorithmen und Datenstrukturen
- Hügel
  - Algorithmus 1
  - Algorithmus 2
  - Algorithmus 3

Holger Dell
Folien adaptiert von Philip Bille

Einführung und Hügel

- Algorithmen und Datenstrukturen
- Hügel
  - Algorithmus 1
  - Algorithmus 2
  - Algorithmus 3

Algorithmen und Datenstrukturen

- Algorithmisches Problem.
  Präzise definierte Relation zwischen Eingabe und Ausgabe.

- Algorithmus.
  Methode, um ein algorithmisches Problem zu lösen.
  - Diskrete and eindeutige Anweisungen.
  - Mathematische Abstraktion eines Programms.

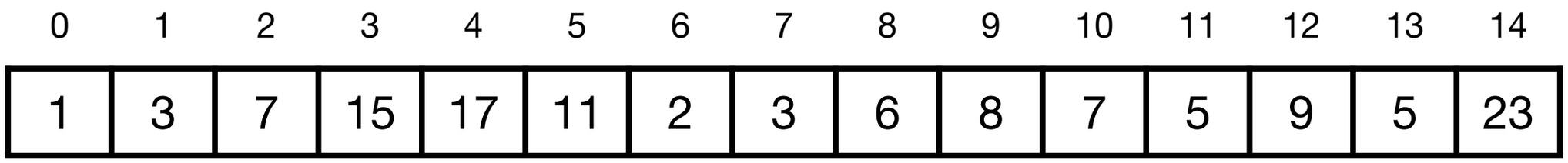
- abstrakte Datenstruktur.
  Präzise definierte Schnittstelle, um Daten zu lesen oder schreiben.

- Datenstruktur.
  Methode, um eine abstrakte Datenstruktur zu verwirklichen.
  - Datenorganisation.
  - Lese- und Schreibzugriffe durch Algorithmen.

Beispiel: Maximum finden

- Maximum finden.
  Gegeben ein Feld A[0..n-1], finde einen Index i, so dass A[i] maximal ist.
  - Eingabe. Feld A[0..n-1].
  - Ausgabe. Ein Index i, so dass A[i] ≥ A[j] für alle Indizes j.

- Algorithmus.
  - Durchlaufe A von links nach rechts und merke dir dabei den Index des bisher maximalen Werts.
  - Gib den Index zurück.



Beschreibung von Algorithmen

- Natürliche Sprache.
  - Durchlaufe A von links nach rechts und merke dir dabei den Index des bisher maximalen Werts.
  - Gib den Index zurück.

- Programm.

```
public static int findMax(int[] A) {
   int max = 0;
   for(int i = 0; i < A.length; i++)
        if (A[i] > A[max]) max = i;
   return max;
}
```

- Pseudocode.

```
FindMax(A, n)
   max = 0
   for i = 0 to n-1
      if (A[i] > A[max]) max = i
   return max
```

Einführung und Hügel

- Algorithmen und Datenstrukturen
- Hügel
  - Algorithmus 1
  - Algorithmus 2
  - Algorithmus 3

Hügel

- Hügel. A[i] ist ein Hügel wenn A[i] mindestens so groß wie seine Nachbarn ist:
  - Für alle i ∈ {1, …, n-2} ist A[i] ein Hügel, wenn A[i-1] ≤ A[i] ≥ A[i+1].
  - A[0] ist ein Hügel, wenn A[0] ≥ A[1].
  - A[n-1] ist ein Hügel, wenn A[n-2] ≤ A[n-1].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

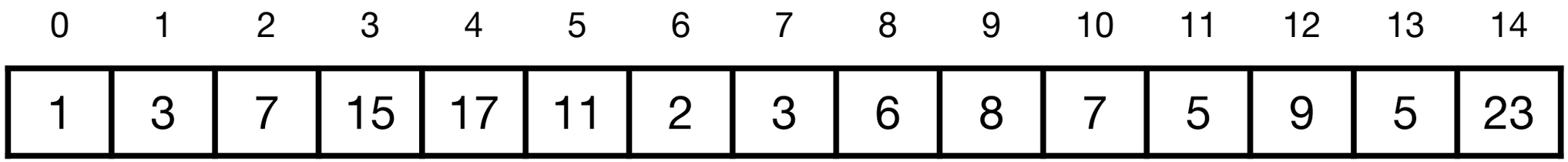
- Hügel finden. Gegeben ein Feld A[0..n-1], finde einen Index i, so dass A[i] Hügel ist.
  - Eingabe. Ein Feld A[0..n-1].
  - Ausgabe. Ein Index i, so dass A[i] ein Hügel ist.

Einführung und Hügel

- Algorithmen und Datenstrukturen
- Hügel
  - Algorithmus 1
  - Algorithmus 2
  - Algorithmus 3

Algorithmus 1

- Algorithmus 1.
  Teste für jeden Eintrag, ob er ein Hügel ist. Gib den Index des ersten Hügels zurück.



- Pseudocode.

```
PEAK1(A, n)
   if A[0] ≥ A[1] return 0
   for i = 1 to n-2
      if A[i-1] ≤ A[i] ≥ A[i+1] return i
   if A[n-2] ≤ A[n-1] return n-1
```

- Frage. Wie analysieren wir den Algorithmus?

Theoretische Analyse

- Laufzeit / Zeitkomplexität.
  - T(n) = Anzahl der Schritte, die der Algorithmus auf Eingabelänge n ausführt.

- Schritte.
  - Lesen/Schreiben im Speicher (`x := y`, `A[i]`, `i = i + 1`, ...)
  - Arithmetische/Boolsche Operationen (`+ - * / % && || & | ^ ~`)
  - Vergleiche (`<` `>` `=<` `=>` `=` ≠)
  - Programmfluss (if-then-else, while, for, goto, function call, ...)

- Laufzeit im schlechtesten Fall (*worst-case*).
  - Maximale Laufzeit über alle Eingaben der Größe n.

Theoretische Analyse

- Laufzeit. Was ist die Laufzeit T(n) von Algorithmus 1?

```
PEAK1(A, n)
   if A[0] ≥ A[1] return 0
   for i = 1 to n-2
      if A[i-1] ≤ A[i] ≥ A[i+1] return i
   if A[n-2] ≤ A[n-1] return n-1
```

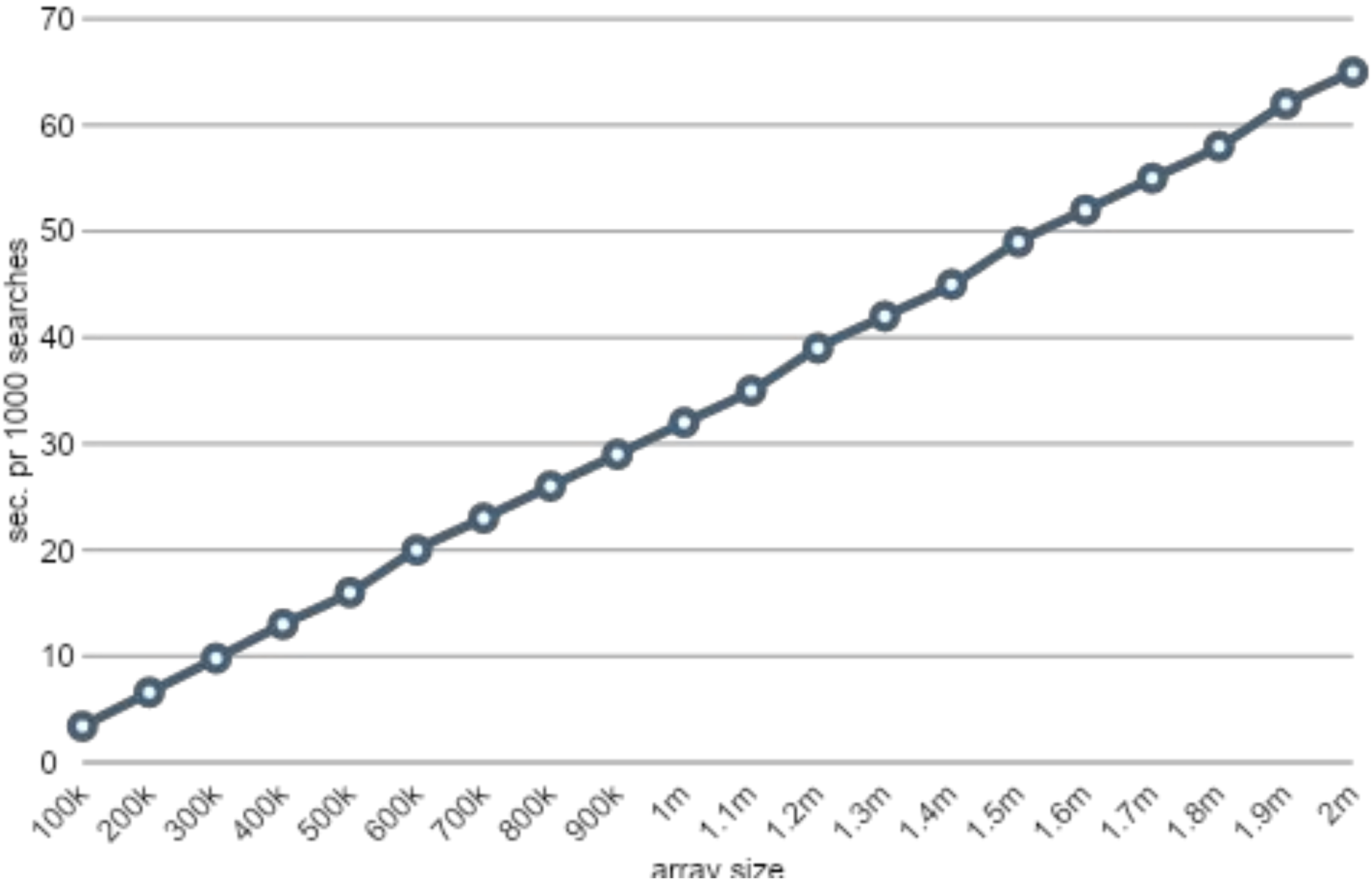
c_1

$(n-2)\cdot c_2$

c_3

$$T(n) = c_1 + (n-2)\cdot c_2 + c_3$$

- T(n) ist eine lineare Funktion in n: $T(n) = an + b$
- Asymptotische Notation. $T(n) = O(n)$
- Experimentelle Analyse.
  - Bestimme die experimentelle Laufzeit von Algorithmus 1.
  - Was hat die experimentelle Analyse mit der theoretischen Analyse zu tun?


70
60
50
40
30
20
10
0
sec. pr 1000 searches
100k
200k
300k
400k
500k
600k
700k
800k
900k
1m
1.1m
1.2m
1.3m
1.4m
1.5m
1.6m
1.7m
1.8m
1.9m
2m
array size

Hügel

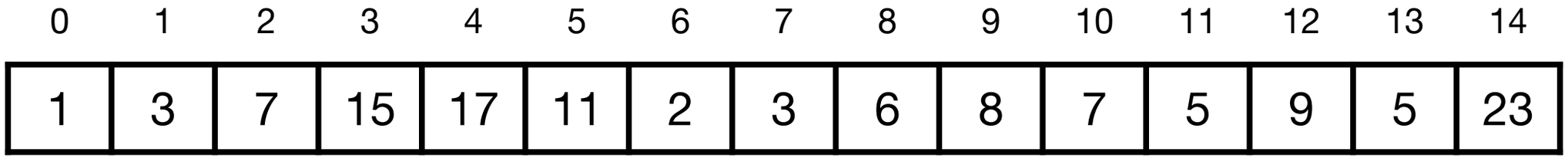
- Algorithmus 1 findet einen Hügel in Zeit O(n).
- Theoretische und experimentelle Analyse sind konsistent.
- Frage. Können wir es besser machen?

Einführung und Hügel

- Algorithmen und Datenstrukturen
- Hügel
  - Algorithmus 1
  - Algorithmus 2
  - Algorithmus 3

Algorithmus 2

- Beobachtung. Jeder maximale Eintrag A[i] ist auch ein Hügel.
- Algorithmus 2. Finde einen maximalen Eintrag in A mit FindMax(A, n).




```
FindMax(A, n)
   max = 0
   for i = 0 to n-1
      if (A[i] > A[max]) max = i
   return max
```

Theoretische Analyse

- Laufzeit. Was ist die Laufzeit T(n) für Algorithmus 2?

```
FindMax(A, n)
   max = 0
   for i = 0 to n-1
      if (A[i] > A[max]) max = i
   return max
```

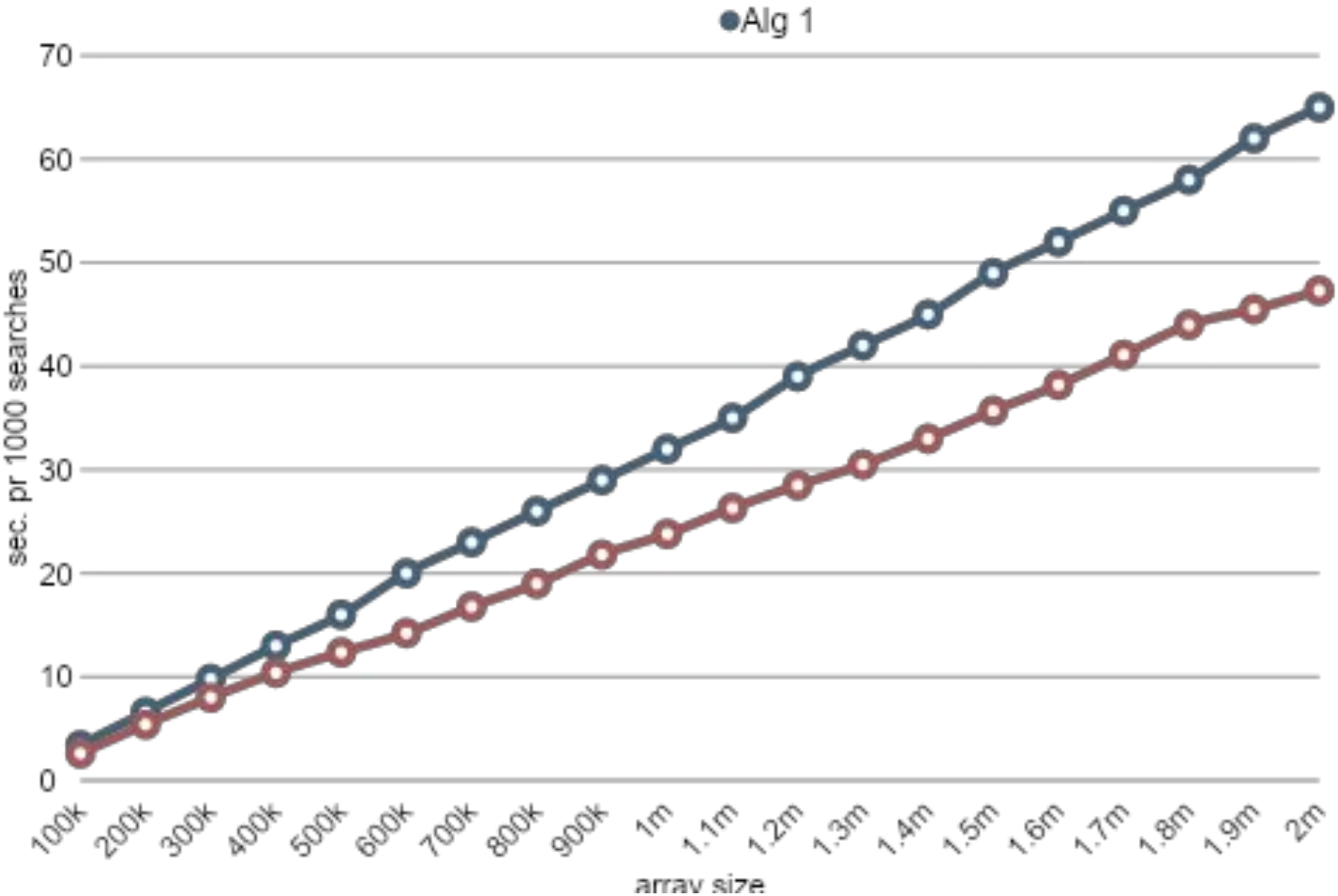
c_4

$n \cdot c_5$

c_6

$$T(n) = c_4 + n \cdot c_5 + c_6 = O(n)$$

- Experimentelle Analyse. Sind die Konstanten besser?


Alg 1
70
60
50
40
30
20
10
0
sec. pr 1000 searches
100k
200k
300k
400k
500k
600k
700k
800k
900k
1m
1.1m
1.2m
1.3m
1.4m
1.5m
1.6m
1.7m
1.8m
1.9m
2m
array size

Hügel

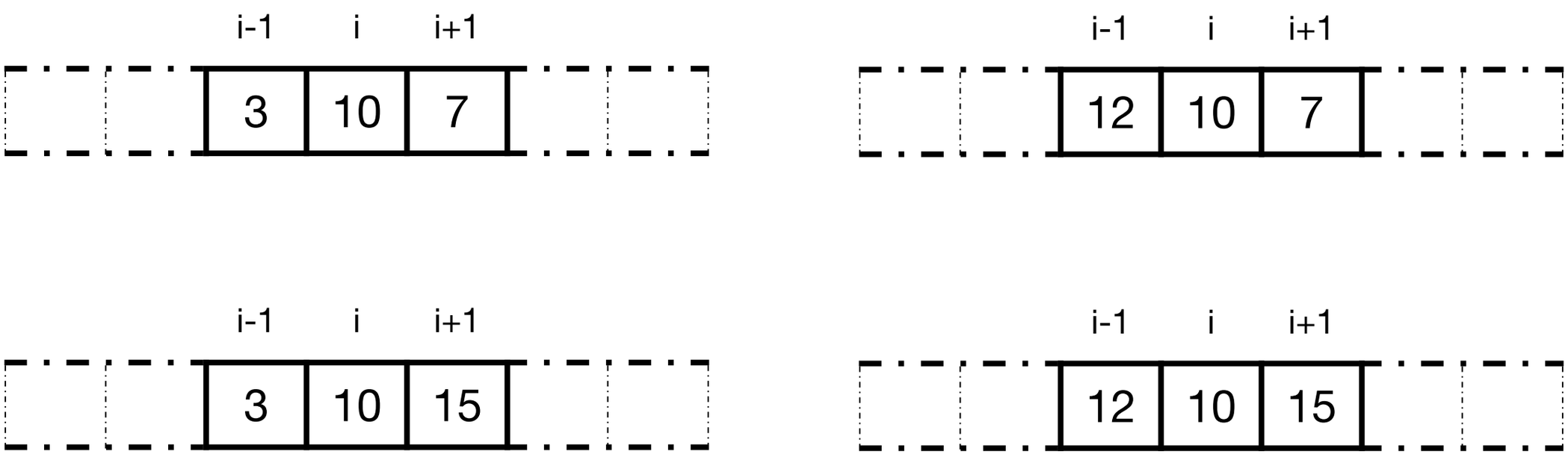
- Theoretische Analyse.
  - Algorithmus 1 und 2 finden einen Hügel in Zeit O(n).
- Experimentelle Analyse.
  - Algorithmus 1 und 2 laufen auch in der Praxis in Zeit O(n).
  - Algorithmus 2 ist um einen konstanten Faktor schneller als Algorithmus 1.
- Frage. Können wir das signifikant besser machen?

Einführung und Hügel

- Algorithmen und Datenstrukturen
- Hügel
  - Algorithmus 1
  - Algorithmus 2
  - Algorithmus 3

Algorithmus 3

- Clevere Idee.
  - Betrachte einen Eintrag A[i] und seine Nachbarn A[i-1] und A[i+1].
  - Wo kann ein Hügel sich relativ zu A[i] befinden?
    - Nachbarn sind ≤ A[i] ⟹ A[i] ist ein Hügel.
    - Sonst ist A wachsend in mindestens eine Richtung ⟹ ein Hügel muss in dieser Richtung existieren.



- Frage. Wie machen wir daraus einen schnellen Algorithmus?

Algorithmus 3

- Algorithmus 3.
  - Betrachte den mittleren Eintrag A[m] und Nachbarn A[m-1] und A[m+1].
  - Wenn A[m] ein Hügel ist, gib m zurück.
  - Sonst suche rekursiv in der Hälfte mit dem größeren Nachbarn.

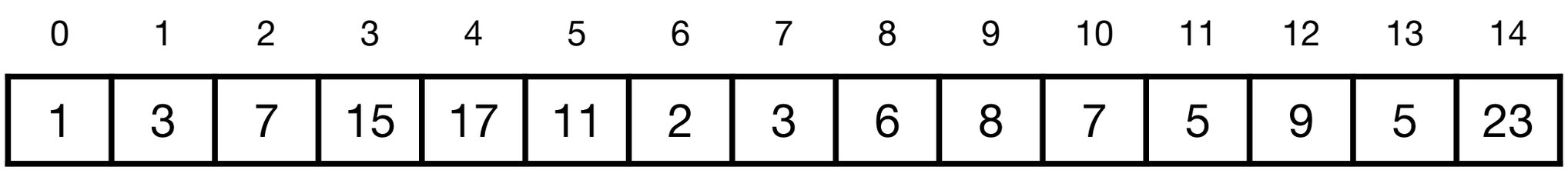
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

Algorithmus 3

- Algorithmus 3.
  - Betrachte den mittleren Eintrag A[m] und Nachbarn A[m-1] und A[m+1].
  - Wenn A[m] ein Hügel ist, gib m zurück.
  - Sonst suche rekursiv in der Hälfte mit dem größeren Nachbarn.

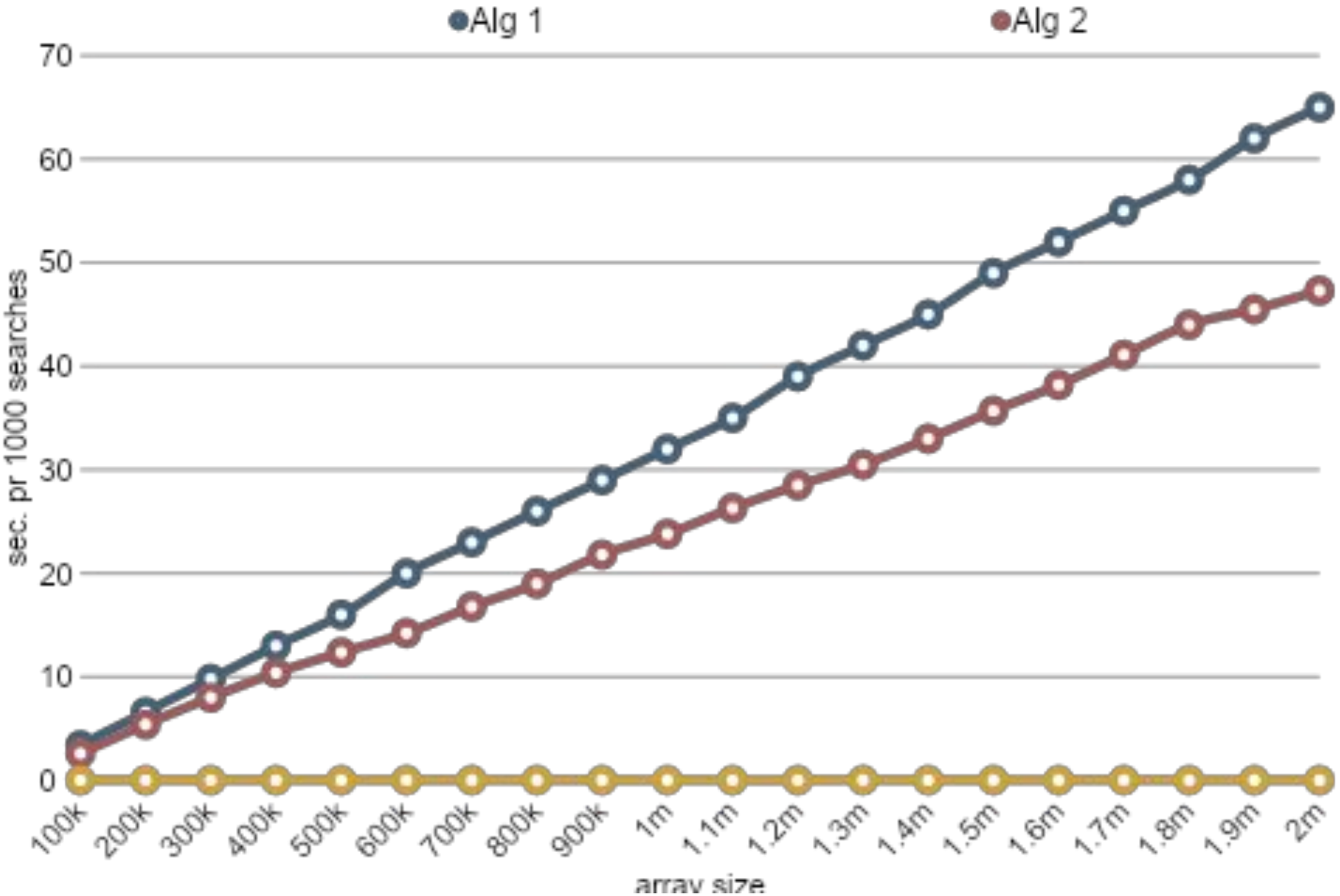
```
PEAK3(A,i,j)
   m = ⌊(i+j)/2)⌋
   if A[m] ≥ neighbors return m
   elseif A[m-1] > A[m]
      return PEAK3(A,i,m-1)
   elseif A[m] < A[m+1]
      return PEAK3(A,m+1,j)
```



Algorithmus 3

```
PEAK3(A,i,j)
   m = ⌊(i+j)/2)⌋
   if A[m] ≥ neighbors return m
   elseif A[m-1] > A[m]
      return PEAK3(A,i,m-1)
   elseif A[m] < A[m+1]
      return PEAK3(A,m+1,j)
```

- Theoretische Laufzeitanalyse.
  - Ein rekursiver Aufruf braucht konstante Zeit.
  - Wie viele rekursive Aufrufe?
  - Ein rekursiver Aufruf halbiert die Intervalllänge. Ende, wenn die Länge 1 ist.
    - 1. rekursiver Aufruf: n/2
    - 2. rekursiver Aufruf: n/4
    - ….
    - k. rekursiver Aufruf: $n/2^k$
    - ….
  - $\Longrightarrow$ Nach ~$\log_2$ n rekursiven Aufrufen: Intervalllänge ≤1.
  - $\Longrightarrow$ Laufzeit ist O(log n)
- Experimentelle Analyse. Ist’s wirklich besser?


Alg 1
Alg 2
70
60
50
40
30
20
10
0
sec. pr 1000 searches
100k
200k
300k
400k
500k
600k
700k
800k
900k
1m
1.1m
1.2m
1.3m
1.4m
1.5m
1.6m
1.7m
1.8m
1.9m
2m
array size

Hügel

- Theoretische Analyse.
  - Algorithmus 1 und 2 finden einen Hügel in Zeit O(n).
  - Algorithmus 3 findet einen Hügel in Zeit O(log n).
- Experimentelle Analyse.
  - Algorithmus 1 und 2 laufen in der Praxis in Zeit O(n).
  - Algorithmus 2 ist um einen konstanten Faktor schneller als Algorithmus 1.
  - Algorithmus 3 ist viel, viel schneller als Algorithmus 1 und 2.

Einführung und Hügel

- Algorithmen und Datenstrukturen
- Hügel
  - Algorithmus 1
  - Algorithmus 2
  - Algorithmus 3