Algorithmen und Komplexität
Vorlesung 1

Wolfgang Globke

3. April 2019

Dozent

Dr. Wolfgang Globke
Email: wolfgang.globke@univie.ac.at

Vita

- 2007: Diplom Informatik, Universität Karlsruhe (TH)
- 2011: Dr. rer. nat. Mathematik, Karlsruhe Institute of Technology
- 2012-2018: Wissenschaftlicher Mitarbeiter, School of Mathematical Sciences, University of Adelaide
- 2018-heute: Wissenschaftlicher Mitarbeiter, Fakultät für Mathematik, Universität Wien

Motivation – Was sind Algorithmen?

Ein Algorithmus ist eine eindeutige Vorschrift zur Lösung eines Problems (oder Problemklasse) in endlich vielen Schritten („Kochrezept”).

Ein Algorithmus ist eine eindeutige Vorschrift zur Lösung eines Problems (oder Problemklasse) in endlich vielen Schritten („Kochrezept").

Das Wort „Algorithmus" ist vom Namen des persischen Mathematikers Muhammad ibn Musa Al-Khwarizmi (≈ 780–850 n.Chr.) abgeleitet.

- Übersetzung wissenschaftlicher Werke aus dem Griechischen.
- Schrieb ein Lehrbuch über Astronomie und das erste Lehrbuch der Algebra (zu praktischen Zwecken).
- Auf ihn geht auch das Wort „Algebra" zurück, da es seine Methode des Lösens quadratischer Gleichungen beschreibt („al-jabr").

Motivation – Wozu Algorithmen (und diese Vorlesung)?

Der Lösung eines jeden nicht-trivialen Problems in der Informatik liegt ein Algorithmus zugrunde.

Motivation – Wozu Algorithmen (und diese Vorlesung)?

Der Lösung eines jeden nicht-trivialen Problems in der Informatik liegt ein Algorithmus zugrunde.

- Probleme müssen „hinreichend schnell" gelöst werden
  (je nach Problem: Mikrosekunden, Sekunden, Minuten, Tage…)

Motivation – Wozu Algorithmen (und diese Vorlesung)?

Der Lösung eines jeden nicht-trivialen Problems in der Informatik liegt ein Algorithmus zugrunde.

- Probleme müssen „hinreichend schnell" gelöst werden
  (je nach Problem: Mikrosekunden, Sekunden, Minuten, Tage...)
- Algorithmenentwurf muss Lösungen finden, die diesem Anspruch gerecht werden.

Motivation – Wozu Algorithmen (und diese Vorlesung)?

Der Lösung eines jeden nicht-trivialen Problems in der Informatik liegt ein Algorithmus zugrunde.

- Probleme müssen „hinreichend schnell" gelöst werden
  (je nach Problem: Mikrosekunden, Sekunden, Minuten, Tage…)
- Algorithmenentwurf muss Lösungen finden, die diesem Anspruch gerecht werden.
  - Kenntnis der gängigen Algorithmen und Entwurfsmethoden hilft bei der Auswahl bzw. Entwicklung des geeigneten Verfahrens.
  - Theoretische und experimentelle Analyse, ob der Algorithmus den Anforderungen genügt.

Motivation – Wozu Algorithmen (und diese Vorlesung)?

Der Lösung eines jeden nicht-trivialen Problems in der Informatik liegt ein Algorithmus zugrunde.

- Probleme müssen „hinreichend schnell" gelöst werden
  (je nach Problem: Mikrosekunden, Sekunden, Minuten, Tage...)
- Algorithmenentwurf muss Lösungen finden, die diesem Anspruch gerecht werden.
  - Kenntnis der gängigen Algorithmen und Entwurfsmethoden hilft bei der Auswahl bzw. Entwicklung des geeigneten Verfahrens.
  - Theoretische und experimentelle Analyse, ob der Algorithmus den Anforderungen genügt.
- Geeignete Datenypen fördern die Leistungsfähigkeit der Algorithmen.

Motivation – Wozu Algorithmen (und diese Vorlesung)?

Der Lösung eines jeden nicht-trivialen Problems in der Informatik liegt ein Algorithmus zugrunde.

- Probleme müssen „hinreichend schnell" gelöst werden
  (je nach Problem: Mikrosekunden, Sekunden, Minuten, Tage...)
- Algorithmenentwurf muss Lösungen finden, die diesem Anspruch gerecht werden.
  - Kenntnis der gängigen Algorithmen und Entwurfsmethoden hilft bei der Auswahl bzw. Entwicklung des geeigneten Verfahrens.
  - Theoretische und experimentelle Analyse, ob der Algorithmus den Anforderungen genügt.
- Geeignete Datenypen fördern die Leistungsfähigkeit der Algorithmen.
- Verständnis für grundsätzliche Schwierigkeiten bei der Lösung gewisser Problemklassen (z.B. P vs. NP, Worst-Case-Laufzeiten).

Was wir behandeln:

- Einführung, motivierendes Beispiel

Überblick über die Vorlesung

Was wir behandeln:

- Einführung, motivierendes Beispiel
- Methoden der Algorithmenanalyse (O-Kalkül, Master-Theorem, Invarianten...)

Überblick über die Vorlesung

Was wir behandeln:

- Einführung, motivierendes Beispiel
- Methoden der Algorithmenanalyse (O-Kalkül, Master-Theorem, Invarianten…)
- Einführung in Komplexitätstheorie (P, NP…)

Was wir behandeln:

- Einführung, motivierendes Beispiel
- Methoden der Algorithmenanalyse (O-Kalkül, Master-Theorem, Invarianten…)
- Einführung in Komplexitätstheorie (P, NP…)
- Grundlegende Datenstrukturen: Von Arrays zu Listen, Stacks, Queues, etc.

Überblick über die Vorlesung

Was wir behandeln:

- Einführung, motivierendes Beispiel
- Methoden der Algorithmenanalyse (O-Kalkül, Master-Theorem, Invarianten...)
- Einführung in Komplexitätstheorie (P, NP...)
- Grundlegende Datenstrukturen: Von Arrays zu Listen, Stacks, Queues, etc.
- Grundlegende Algorithmen: Hashing, Sortieren, Suchen, Auswählen

Überblick über die Vorlesung

Was wir behandeln:

- Einführung, motivierendes Beispiel
- Methoden der Algorithmenanalyse (O-Kalkül, Master-Theorem, Invarianten…)
- Einführung in Komplexitätstheorie (P, NP…)
- Grundlegende Datenstrukturen: Von Arrays zu Listen, Stacks, Queues, etc.
- Grundlegende Algorithmen: Hashing, Sortieren, Suchen, Auswählen
- Datenstrukturen: Bäume und Graphen

Überblick über die Vorlesung

Was wir behandeln:

- Einführung, motivierendes Beispiel
- Methoden der Algorithmenanalyse (O-Kalkül, Master-Theorem, Invarianten…)
- Einführung in Komplexitätstheorie (P, NP…)
- Grundlegende Datenstrukturen: Von Arrays zu Listen, Stacks, Queues, etc.
- Grundlegende Algorithmen: Hashing, Sortieren, Suchen, Auswählen
- Datenstrukturen: Bäume und Graphen
- Die wichtigsten Graphenalgorithmen

Überblick über die Vorlesung

Was wir behandeln:

- Einführung, motivierendes Beispiel
- Methoden der Algorithmenanalyse (O-Kalkül, Master-Theorem, Invarianten…)
- Einführung in Komplexitätstheorie (P, NP…)
- Grundlegende Datenstrukturen: Von Arrays zu Listen, Stacks, Queues, etc.
- Grundlegende Algorithmen: Hashing, Sortieren, Suchen, Auswählen
- Datenstrukturen: Bäume und Graphen
- Die wichtigsten Graphenalgorithmen
- Codierung und Kompression/Komprimierung von Daten

Jenseits dieser Vorlesung

Interessante Dinge, die wir hier nicht behandeln:

- Stochastische Algorithmen.
- Hochoptimierte Algorithmen für spezifische Prozessortypen
- Sprachen und Automaten
- Maschinelles Lernen, künstliche Intelligenz
- Kryptographie
- Internetalgorithmen (allerdings teilweise in Graphenalgorithmen)
- Datenbankalgorithmen
- PageRank
- Graphische/geometrische Algorithmen
- Mathematische Algorithmen (Numerik, Computeralgebra)
- Softwaretechnik (keine Algorithmen)
- Programmiersprachen, Compiler

Mathematik

- Vollständige Induktion
- Wachstumsverhalten von Funktionen
- Elementare Wahrscheinlichkeitstheorie
- Elementare Matrizenrechnung

Informatik

- Programmierspache C (inkl. Strukturen und Zeiger)
- Umgang mit GNU C-Compiler gcc

Ablauf von Vorlesung und Übung

Vorlesung:
Mittwoch, 11:00-12:30 (wöchentlich), Raum 179 C
Donnerstag, 12:30-14:00 (wöchentlich), Raum 179 C

Übung:
Dienstag, 11:00-12:30 (14-tägig), Raum 179 C
(d.h. am 9. April, 23. April, 7. Mai, 21. Mai, 4. Juni)

Übungsaufgaben

- 5 Übungsblätter mit jeweils 20 Punkten,
- Abgabe zu Beginn der jeweils nächsten Übung,
- neue Übungsaufgaben nach der jeweiligen Übung,
- enthalten Theorie- und Programmieraufgaben (in C),
- lesbar schreiben/coden!!!

Programmieraufgaben

- Programmiersprache C (nicht C++).
- Programme müssen mit gcc übsersetzbar sein.
- Programmgerüst wird vorgegeben, das an klar gekennzeichneten Stellen mit dem eigenen Code gefüllt wird.
- Der Code außerhalb dieser Stellen muss unverändert bleiben.
- Keine Bibliotheken einbinden, die nicht vom Programmgerüst schon vorgegeben sind.
- Eigenen Namen in den abgegebenen Quellcode schreiben.

- Programmiersprache C (nicht C++).
- Programme müssen mit gcc übsersetzbar sein.
- Programmgerüst wird vorgegeben, das an klar gekennzeichneten Stellen mit dem eigenen Code gefüllt wird.
- Der Code außerhalb dieser Stellen muss unverändert bleiben.
- Keine Bibliotheken einbinden, die nicht vom Programmgerüst schon vorgegeben sind.
- Eigenen Namen in den abgegebenen Quellcode schreiben.

Beispiel

```
#include <stdio.h>
int fibonacci(int k) {
/*** BEGIN eigener Code ***/

/*** END eigener Code ***/
}
int main() {
  int n;
  printf(" Enter number n = ");
  scanf("%d", &n);
  printf("%d-th Fibonacci number equals: %d", n, fibonacci(n));
  return 0;
}
```

Auf
`https://moodle.dhbw-mannheim.de`
ist dieser Kurs zu finden unter

- Studiengang Informationstechnik
- SS19
- Algorithmen und Komplexität (MA-TINF18ITNS).

Bitte alle diesem Kurs beitreten!

Auf
`https://moodle.dhbw-mannheim.de`
ist dieser Kurs zu finden unter

- Studiengang Informationstechnik
- SS19
- Algorithmen und Komplexität (MA-TINF18ITNS).

Bitte alle diesem Kurs beitreten! Dort gibt es

- Forum
- Übungsblätter
- Programmieraufgaben
- Abgabe der Programmieraufgaben online
- Gruppenbenachrichtigungen

Klausur am 13. Juni, 12:30-14:30
Raum 036 B

Gesamtnote = (0.7× Klausurnote) + (0.3× Übungsnote)

Zum Bestehen der Klausur sind höchstens 50% der erreichbaren Punkte notwendig.

Literatur

- *T.H. Cormen, C.E. Leiserson, R. Rivest, C. Stein*,
  Algorithmen – Eine Einführung, vierte Auflage, De Gruyter Oldenbourg, 2013
  Der Klassiker unter den Algorithmenlehrbüchern. Gewaltiger Umfang, geht weit über die Vorlesung hinaus. Gut zum Lernen und zum Nachschlagen.
- *D.E. Knuth*,
  The Art of Computer Programming 1–4, Addison-Wesly 2011
  Die Bibel der mathematischen Algorithmentheorie. Geht weit über das für praktischen Anwendungen nötige hinaus.
- *K. Mehlhorn, P. Sanders*,
  Algorithms and Data Structures – The Basic Toolbox, Springer 2008 (auch auf deutsch erhältlich)
  Moderne und recht formale Einführung in die wichtigsten Algorithmen und ihre mathematische Analyse. Nicht immer anfängerfreundlich geschrieben.

Ein Beispiel: Binäre Suche

Binäre Suche: Problemstellung

Die binäre Suche ist ein praktisches Verfahren, um innerhalb einer sortierten Liste die Position eines gesuchten Eintrags zu finden.

Binäre Suche: Problemstellung

Die binäre Suche ist ein praktisches Verfahren, um innerhalb einer sortierten Liste die Position eines gesuchten Eintrags zu finden.

Konkreter:

- Gegeben:
  Ein sortierter Array $A[0, \ldots, n-1]$ und ein Element x vom selben Datentyp wie die Einträge von A.
- Gesucht:
  Der Index $i \in \{0, \ldots, n-1\}$, für den $A[i] = x$ gilt,
  oder eine Nachricht, dass x nicht in A enthalten ist.

(Der Einfachheit wegen nehmen wir an, dass kein Eintrag in A mehrfach vorkommt, also $A[0] < A[1] < \ldots < A[n-1]$.)

Binäre Suche: Algorithmus

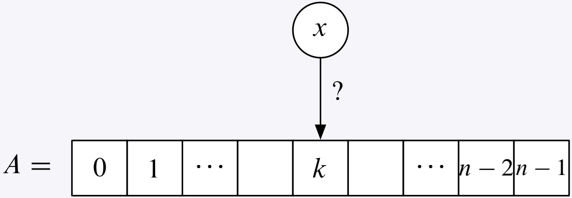
Idee: Suchen wie im Telefonbuch.

Idee: Suchen wie im Telefonbuch.

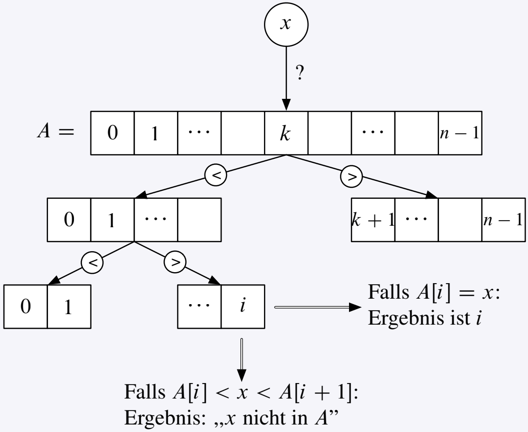
- In der Mitte anfangen.
- Eintrag gefunden? $\leadsto$ fertig.
- Nachsehen, ob der gesuchte Eintrag vor oder hinter der Mitte steht.
- In der vorderen bzw. hinteren Hälfte des Telefonbuchs die Prozedur wiederholen.

Idee: Suchen wie im Telefonbuch.

- In der Mitte anfangen.
- Eintrag gefunden? $\rightsquigarrow$ fertig.
- Nachsehen, ob der gesuchte Eintrag vor oder hinter der Mitte steht.
- In der vorderen bzw. hinteren Hälfte des Telefonbuchs die Prozedur wiederholen.



Prinzip Teile und Herrsche (engl. Divide and Conquer)



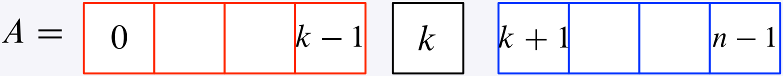
Erste Skizze:

$\textsc{BinarySearchSketch}(A, x)$
  $n :=$ Length(A)
  $k := \lfloor \frac{n}{2} \rfloor$
  if $A[k] = x$ **then return** k
  if $n = 1$ **then return** "x not in A"
  if $x > A[k]$ **then** $\textsc{BinarySearchSketch}(A[k+1, \ldots, n-1], x)$
  else if $x < A[k]$ **then** $\textsc{BinarySearchSketch}(A[0, \ldots, k-1], x)$

Binäre Suche: Algorithmus

Erste Skizze:

$\textsc{BinarySearchSketch}(A, x)$
  $n :=$ Length(A)
  $k := \lfloor \frac{n}{2} \rfloor$
  if $A[k] = x$ **then return** k
  if $n = 1$ **then return** "x not in A"
  if $x > A[k]$ **then** $\textsc{BinarySearchSketch}(A[k+1, \ldots, n-1], x)$
  else if $x < A[k]$ **then** $\textsc{BinarySearchSketch}(A[0, \ldots, k-1], x)$



Eine besser durchdachte Version:

- Keine Kopien von Teilen des Array anlegen!
- Rekursion vermeiden.
- Benutze Zähler ℓ und r, um im Auge zu behalten, in welchen Teil des Array A wir suchen (wie „Finger im Telefonbuch").

Binäre Suche: Algorithmus

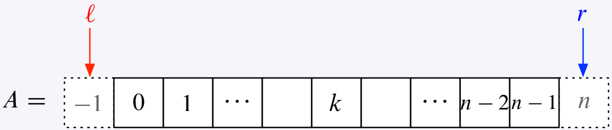
Eine besser durchdachte Version:

- Keine Kopien von Teilen des Array anlegen!
- Rekursion vermeiden.
- Benutze Zähler ℓ und r, um im Auge zu behalten,
  in welchen Teil des Array A wir suchen (wie „Finger im Telefonbuch”).

	ℓ										r
$A =$	-1	0	1	$\cdots$		k		$\cdots$	$n-2$	$n-1$	n

Eine besser durchdachte Version:

- Keine Kopien von Teilen des Array anlegen!
- Rekursion vermeiden.
- Benutze Zähler ℓ und r, um im Auge zu behalten,
  in welchen Teil des Array A wir suchen (wie „Finger im Telefonbuch”).



BinarySearch(A, x)

    $\ell := -1$
    $r := n$
    if $x < A[0]$ **then return** “$x < A[0]$”
    if $x > A[n-1]$ **then return** “$x > A[n-1]$”
    while true **do**
        if $\ell + 1 = r$ **then return** “$A[\ell] < x < A[\ell+1]$”
        $k := \lfloor \frac{\ell+r}{2} \rfloor$
        if $A[k] = x$ **then return** “$A[k] = x$”
        if $A[k] < x$ **then** $\ell := k$ **else** $r := k$

Binäre Suche: Beispiel 1

BINARYSEARCH($A = [2, 3, 5, 7, 11, 13], x = 2$)

  $\ell := -1$

  $r := n$

  if $x < A[0]$ **then return** "$x < A[0]$"

  if $x > A[n-1]$ **then return** "$x > A[n-1]$"

  while true **do**

    if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$"

    $k := \lfloor \frac{\ell + r}{2} \rfloor$

    if $A[k] = x$ **then return** "$A[k] = x$"

    if $A[k] < x$ **then** $\ell := k$ **else** $r := k$

$\textsc{BinarySearch}(A = [2, 3, 5, 7, 11, 13], x = 2)$

$\quad \ell := -1$

$\quad r := n$

$\quad$ **if** $x < A[0]$ **then return** "$x < A[0]$"

$\quad$ **if** $x > A[n-1]$ **then return** "$x > A[n-1]$"

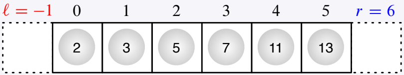
$\quad$ **while** true **do**

$\quad\quad$ **if** $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$"

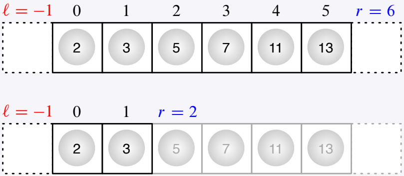
$\quad\quad k := \lfloor \frac{\ell + r}{2} \rfloor$

$\quad\quad$ **if** $A[k] = x$ **then return** "$A[k] = x$"

$\quad\quad$ **if** $A[k] < x$ **then** $\ell := k$ **else** $r := k$



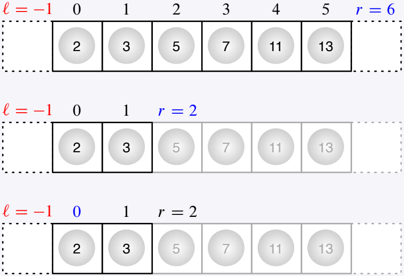
BinarySearch($A = [2, 3, 5, 7, 11, 13], x = 2$)
  $\ell := -1$
  $r := n$
  if $x < A[0]$ **then return** "$x < A[0]$"
  if $x > A[n-1]$ **then return** "$x > A[n-1]$"
  while true **do**
    if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell+1]$"
    $k := \lfloor \frac{\ell+r}{2} \rfloor$
    if $A[k] = x$ **then return** "$A[k] = x$"
    if $A[k] < x$ **then** $\ell := k$ **else** $r := k$



Binäre Suche: Beispiel 1

BINARYSEARCH($A = [2, 3, 5, 7, 11, 13], x = 2$)

  $\ell := -1$
  $r := n$
  if $x < A[0]$ **then return** "$x < A[0]$"
  if $x > A[n-1]$ **then return** "$x > A[n-1]$"
  while true **do**
    if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$"
    $k := \lfloor \frac{\ell+r}{2} \rfloor$
    if $A[k] = x$ **then return** "$A[k] = x$"
    if $A[k] < x$ **then** $\ell := k$ **else** $r := k$



Binäre Suche: Beispiel 2

BINARYSEARCH($A = [2, 3, 5, 7, 11, 13], x = 10$)

```
ℓ := −1
r := n
if x < A[0] then return "x < A[0]"
if x > A[n − 1] then return "x > A[n − 1]"
while true do
    if ℓ + 1 = r then return "A[ℓ] < x < A[ℓ + 1]"
    k := ⌊(ℓ+r)/2⌋
    if A[k] = x then return "A[k] = x"
    if A[k] < x then ℓ := k else r := k
```

Binäre Suche: Beispiel 2

BinarySearch($A = [2, 3, 5, 7, 11, 13], x = 10$)

    $\ell := -1$

    $r := n$

    if $x < A[0]$ **then return** "$x < A[0]$"

    if $x > A[n-1]$ **then return** "$x > A[n-1]$"

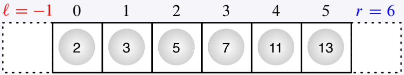
    while true **do**

        if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$"

        $k := \lfloor \frac{\ell + r}{2} \rfloor$

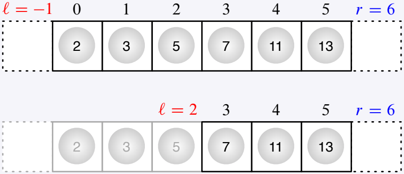
        if $A[k] = x$ **then return** "$A[k] = x$"

        if $A[k] < x$ **then** $\ell := k$ **else** $r := k$



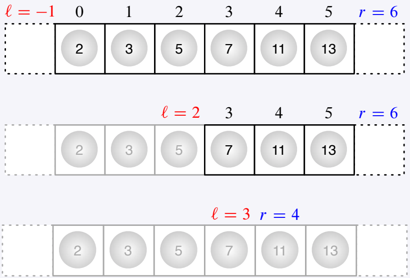
Binäre Suche: Beispiel 2

BinarySearch($A = [2, 3, 5, 7, 11, 13], x = 10$)
  $\ell := -1$
  $r := n$
  if $x < A[0]$ **then return** "$x < A[0]$"
  if $x > A[n-1]$ **then return** "$x > A[n-1]$"
  while true **do**
    if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$"
    $k := \lfloor \frac{\ell + r}{2} \rfloor$
    if $A[k] = x$ **then return** "$A[k] = x$"
    if $A[k] < x$ **then** $\ell := k$ **else** $r := k$



Binäre Suche: Beispiel 2

$\textsc{BinarySearch}(A = [2, 3, 5, 7, 11, 13], x = 10)$
  $\ell := -1$
  $r := n$
  if $x < A[0]$ **then return** "$x < A[0]$"
  if $x > A[n-1]$ **then return** "$x > A[n-1]$"
  while true **do**
    if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$"
    $k := \lfloor \frac{\ell + r}{2} \rfloor$
    if $A[k] = x$ **then return** "$A[k] = x$"
    if $A[k] < x$ **then** $\ell := k$ **else** $r := k$



Binäre Suche: Analyse des Algorithmus

Der Algorithmus BinarySearch mag recht leicht zu verstehen sein, aber grundsätzlich müssen wir beim Entwickeln eines Algorithmus gewisse Eigenschaften gewährleisten:

Binäre Suche: Analyse des Algorithmus

Der Algorithmus BINARYSEARCH mag recht leicht zu verstehen sein, aber grundsätzlich müssen wir beim Entwickeln eines Algorithmus gewisse Eigenschaften gewährleisten:

1. Terminierung
   Für jede beliebige Eingabe endet der Algorithmus nach endlich vielen Schritten. (Dies kann für spezielle Algorithmen entfallen, die in einer Dauerschleife laufen sollen.)

Binäre Suche: Analyse des Algorithmus

Der Algorithmus BINARYSEARCH mag recht leicht zu verstehen sein, aber grundsätzlich müssen wir beim Entwickeln eines Algorithmus gewisse Eigenschaften gewährleisten:

1. Terminierung
   Für jede beliebige Eingabe endet der Algorithmus nach endlich vielen Schritten. (Dies kann für spezielle Algorithmen entfallen, die in einer Dauerschleife laufen sollen.)
2. Korrektheit
   Der Algorithmus tut tatsächlich, was er tun soll.
   - Hierzu müssen Spezifikationen gemacht werden, was als korrekte Eingabe und korrektes Ergebnis aufgefasst werden soll.
   - Sogenannte (Schleifen-)Invarianten (sie gelten vor und nach jedem Durchlauf einer Schleife) helfen, Zwischenzustände des Algorithmus zu untersuchen.
   - Den meisten Korrektheitsbeweisen liegt (explizit oder implizit) vollständige Induktion zugrunde.

Binäre Suche: Analyse des Algorithmus

Der Algorithmus BINARYSEARCH mag recht leicht zu verstehen sein, aber grundsätzlich müssen wir beim Entwickeln eines Algorithmus gewisse Eigenschaften gewährleisten:

1. Terminierung
   Für jede beliebige Eingabe endet der Algorithmus nach endlich vielen Schritten. (Dies kann für spezielle Algorithmen entfallen, die in einer Dauerschleife laufen sollen.)
2. Korrektheit
   Der Algorithmus tut tatsächlich, was er tun soll.
   - Hierzu müssen Spezifikationen gemacht werden, was als korrekte Eingabe und korrektes Ergebnis aufgefasst werden soll.
   - Sogenannte (Schleifen-)Invarianten (sie gelten vor und nach jedem Durchlauf einer Schleife) helfen, Zwischenzustände des Algorithmus zu untersuchen.
   - Den meisten Korrektheitsbeweisen liegt (explizit oder implizit) vollständige Induktion zugrunde.
3. Aufwand
   Der Algorithmus läuft für Eingaben beliebiger Größe effektiv, d.h. in angemessener Zeit, und mit angemessenem Speicherverbrauch.
   Gemessen wird der Aufwand in der Anzahl gewisser „Schritte” oder Elementaroperationen, die der Algorithmus ausführen muss.

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \leq \ell < r \leq n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \leq \ell < r \leq n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Beweis durch Induktion über die Anzahl m der Schleifendurchläufe:

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \leq \ell < r \leq n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Beweis durch Induktion über die Anzahl m der Schleifendurchläufe:

- $m = 0$:

  Unmittelbar vor dem ersten Schleifendurchlauf gilt $\ell = -1$ und $r = n$, also (I1), und die zwei **if** -Bedingungen vor der Schleife erzwingen (I2).

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \leq \ell < r \leq n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Beweis durch Induktion über die Anzahl m der Schleifendurchläufe:

- $m = 0$:

  Unmittelbar vor dem ersten Schleifendurchlauf gilt $\ell = -1$ und $r = n$, also (I1), und die zwei **if** -Bedingungen vor der Schleife erzwingen (I2).

- $m \rightsquigarrow m + 1$:

  Nach Induktionsannahme beginnt der $m + 1$-te Schleifendurchlauf mit Bedingung (I1) und (I2) wahr (vom Ende des m-ten Schleifendurchlaufs).

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \leq \ell < r \leq n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Beweis durch Induktion über die Anzahl m der Schleifendurchläufe:

- $m = 0$:

  Unmittelbar vor dem ersten Schleifendurchlauf gilt $\ell = -1$ und $r = n$, also (I1), und die zwei **if** -Bedingungen vor der Schleife erzwingen (I2).

- $m \rightsquigarrow m + 1$:

  Nach Induktionsannahme beginnt der $m + 1$-te Schleifendurchlauf mit Bedingung (I1) und (I2) wahr (vom Ende des m-ten Schleifendurchlaufs).

- Falls $\ell + 1 = r$, so terminiert der Algorithmus, und es ist nichts zu zeigen.

- Andernfalls muss also $\ell + 2 \leq r$ gelten.

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \leq \ell < r \leq n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Beweis durch Induktion über die Anzahl m der Schleifendurchläufe:

- $m = 0$:

  Unmittelbar vor dem ersten Schleifendurchlauf gilt $\ell = -1$ und $r = n$, also (I1), und die zwei **if** -Bedingungen vor der Schleife erzwingen (I2).

- $m \rightsquigarrow m + 1$:

  Nach Induktionsannahme beginnt der $m + 1$-te Schleifendurchlauf mit Bedingung (I1) und (I2) wahr (vom Ende des m-ten Schleifendurchlaufs).

- Falls $\ell + 1 = r$, so terminiert der Algorithmus, und es ist nichts zu zeigen.

- Andernfalls muss also $\ell + 2 \leq r$ gelten. Ist $k = \lfloor \frac{\ell+r}{2} \rfloor$ der Mittelwert, so gilt demnach $\ell < k < r$.

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \le \ell < r \le n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Beweis durch Induktion über die Anzahl m der Schleifendurchläufe:

- $m = 0$:

  Unmittelbar vor dem ersten Schleifendurchlauf gilt $\ell = -1$ und $r = n$, also (I1), und die zwei **if** -Bedingungen vor der Schleife erzwingen (I2).
- $m \rightsquigarrow m + 1$:

  Nach Induktionsannahme beginnt der $m + 1$-te Schleifendurchlauf mit Bedingung (I1) und (I2) wahr (vom Ende des m-ten Schleifendurchlaufs).
- Falls $\ell + 1 = r$, so terminiert der Algorithmus, und es ist nichts zu zeigen.
- Andernfalls muss also $\ell + 2 \le r$ gelten. Ist $k = \lfloor \frac{\ell+r}{2} \rfloor$ der Mittelwert, so gilt demnach $\ell < k < r$.
- Falls $A[k] = x$, so terminiert der Algorithmus, und es ist nichts zu zeigen.

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \leq \ell < r \leq n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Beweis durch Induktion über die Anzahl m der Schleifendurchläufe:

- $m = 0$:
  Unmittelbar vor dem ersten Schleifendurchlauf gilt $\ell = -1$ und $r = n$, also (I1), und die zwei **if** -Bedingungen vor der Schleife erzwingen (I2).
- $m \rightsquigarrow m + 1$:
  Nach Induktionsannahme beginnt der $m + 1$-te Schleifendurchlauf mit Bedingung (I1) und (I2) wahr (vom Ende des m-ten Schleifendurchlaufs).
- Falls $\ell + 1 = r$, so terminiert der Algorithmus, und es ist nichts zu zeigen.
- Andernfalls muss also $\ell + 2 \leq r$ gelten. Ist $k = \lfloor \frac{\ell+r}{2} \rfloor$ der Mittelwert, so gilt demnach $\ell < k < r$.
- Falls $A[k] = x$, so terminiert der Algorithmus, und es ist nichts zu zeigen.
- Falls $A[k] < x$, so wird $\ell := k$ gesetzt, so dass wieder $A[\ell] < x$ gilt, und $x < A[r]$ gilt noch vom letzten Schleifendurchlauf. Also gelten (I1) und (I2) am Ende der Schleife.

Binäre Suche: Analyse des Algorithmus

Für die Analyse der binären Suche, führen wir die folgenden Invarianten ein.

$$-1 \leq \ell < r \leq n, \tag{I1}$$

$$A[\ell] < x < A[r]. \tag{I2}$$

Diese Invarianten gelten zu Beginn und am Ende jedes Schleifendurchlaufs.

Beweis durch Induktion über die Anzahl m der Schleifendurchläufe:

- $m = 0$:

  Unmittelbar vor dem ersten Schleifendurchlauf gilt $\ell = -1$ und $r = n$, also (I1), und die zwei **if** -Bedingungen vor der Schleife erzwingen (I2).
- $m \rightsquigarrow m + 1$:

  Nach Induktionsannahme beginnt der $m + 1$-te Schleifendurchlauf mit Bedingung (I1) und (I2) wahr (vom Ende des m-ten Schleifendurchlaufs).
- Falls $\ell + 1 = r$, so terminiert der Algorithmus, und es ist nichts zu zeigen.
- Andernfalls muss also $\ell + 2 \leq r$ gelten. Ist $k = \lfloor \frac{\ell+r}{2} \rfloor$ der Mittelwert, so gilt demnach $\ell < k < r$.
- Falls $A[k] = x$, so terminiert der Algorithmus, und es ist nichts zu zeigen.
- Falls $A[k] < x$, so wird $\ell := k$ gesetzt, so dass wieder $A[\ell] < x$ gilt, und $x < A[r]$ gilt noch vom letzten Schleifendurchlauf. Also gelten (I1) und (I2) am Ende der Schleife.
- Ähnlich, falls $A[k] > x$. $\square$

Binäre Suche: Terminierung

Behauptung

Der Algorithmus BinarySearch *terminiert nach endlich vielen Schleifendurchläufen.*

Binäre Suche: Terminierung

Behauptung

Der Algorithmus BINARYSEARCH *terminiert nach endlich vielen Schleifendurchläufen.*

Beweis:

- Falls in einem Schleifendurchlauf der Algorithmus nicht durch ein „return" endet, so wird entweder ℓ erhöht oder r verkleinert.

Binäre Suche: Terminierung

Behauptung

Der Algorithmus BinarySearch *terminiert nach endlich vielen Schleifendurchläufen.*

Beweis:

- Falls in einem Schleifendurchlauf der Algorithmus nicht durch ein „return" endet, so wird entweder ℓ erhöht oder r verkleinert.
- In jedem Fall wird, wegen unserer Invariante (I1), $-1 \leq \ell < r \leq n$, der Abstand $|\ell - r|$ kleiner.

Behauptung

Der Algorithmus BinarySearch *terminiert nach endlich vielen Schleifendurchläufen.*

Beweis:

- Falls in einem Schleifendurchlauf der Algorithmus nicht durch ein „return" endet, so wird entweder ℓ erhöht oder r verkleinert.
- In jedem Fall wird, wegen unserer Invariante (I1), $-1 \leq \ell < r \leq n$, der Abstand $|\ell - r|$ kleiner.
- Dies kann nur endlich oft geschehen. $\square$

Behauptung

Angenommen $A[0] \leq x \leq A[n-1]$. *Dann:*

1. *Ist* x *in* A *enthalten, so liefert* BINARYSEARCH *den Index* k *mit* $A[k] = x$.
2. *Ist* x *nicht in* A *enthalten, so liefert* BINARYSEARCH *den Index* k *mit* $A[k] < x < A[k+1]$.

Behauptung

Angenommen $A[0] \leq x \leq A[n-1]$. *Dann:*

1. *Ist x in A enthalten, so liefert* BINARYSEARCH *den Index k mit $A[k] = x$.*
2. *Ist x nicht in A enthalten, so liefert* BINARYSEARCH *den Index k mit $A[k] < x < A[k+1]$.*

Beweis:

- Wir wissen inzwischen, dass BINARYSEARCH immer terminiert.
  Die zwei möglichen Ausgaben sind (1) "$A[k] = x$" und (2) "$A[\ell] < x < A[\ell+1]$".

Binäre Suche: Korrektheit des Algorithmus

Behauptung

Angenommen $A[0] \leq x \leq A[n-1]$. *Dann:*

1. *Ist* x *in* A *enthalten, so liefert* BinarySearch *den Index* k *mit* $A[k] = x$.
2. *Ist* x *nicht in* A *enthalten, so liefert* BinarySearch *den Index* k *mit* $A[k] < x < A[k+1]$.

Beweis:

- Wir wissen inzwischen, dass BinarySearch immer terminiert.
  Die zwei möglichen Ausgaben sind (1) "$A[k] = x$" und (2) "$A[\ell] < x < A[\ell + 1]$".
- Die Ausgabe (1) tritt offensichtlich nur ein, wenn $A[k] = x$ ist (für $0 \leq k \leq n-1$).

Behauptung

Angenommen $A[0] \leq x \leq A[n-1]$. Dann:

1. *Ist x in A enthalten, so liefert* BinarySearch *den Index k mit $A[k] = x$.*
2. *Ist x nicht in A enthalten, so liefert* BinarySearch *den Index k mit $A[k] < x < A[k+1]$.*

Beweis:

- Wir wissen inzwischen, dass BinarySearch immer terminiert.
  Die zwei möglichen Ausgaben sind (1) "$A[k] = x$" und (2) "$A[\ell] < x < A[\ell + 1]$".
- Die Ausgabe (1) tritt offensichtlich nur ein, wenn $A[k] = x$ ist (für $0 \leq k \leq n-1$).
- Zu Beginn der Schleife gilt stets $A[\ell] < x < A[r]$ nach Invariante (I2).

Behauptung

Angenommen $A[0] \leq x \leq A[n-1]$. Dann:

1. *Ist x in A enthalten, so liefert* BinarySearch *den Index k mit $A[k] = x$.*
2. *Ist x nicht in A enthalten, so liefert* BinarySearch *den Index k mit $A[k] < x < A[k+1]$.*

Beweis:

- Wir wissen inzwischen, dass BinarySearch immer terminiert.
  Die zwei möglichen Ausgaben sind (1) “$A[k] = x$” und (2) “$A[\ell] < x < A[\ell+1]$”.
- Die Ausgabe (1) tritt offensichtlich nur ein, wenn $A[k] = x$ ist (für $0 \leq k \leq n-1$).
- Zu Beginn der Schleife gilt stets $A[\ell] < x < A[r]$ nach Invariante (I2).
- Bei Ausgabe (2) gilt $\ell + 1 = r$, und somit $A[\ell] < x < A[\ell+1]$.

Behauptung

Angenommen $A[0] \leq x \leq A[n-1]$. Dann:

1. *Ist x in A enthalten, so liefert* BinarySearch *den Index k mit $A[k] = x$.*
2. *Ist x nicht in A enthalten, so liefert* BinarySearch *den Index k mit $A[k] < x < A[k+1]$.*

Beweis:

- Wir wissen inzwischen, dass BinarySearch immer terminiert.
  Die zwei möglichen Ausgaben sind (1) "$A[k] = x$" und (2) "$A[\ell] < x < A[\ell+1]$".
- Die Ausgabe (1) tritt offensichtlich nur ein, wenn $A[k] = x$ ist (für $0 \leq k \leq n-1$).
- Zu Beginn der Schleife gilt stets $A[\ell] < x < A[r]$ nach Invariante (I2).
- Bei Ausgabe (2) gilt $\ell + 1 = r$, und somit $A[\ell] < x < A[\ell+1]$.
  Es ist also kein weiterer Index mehr verfügbar, an dem x liegen könnte.
  Das bedeutet, x ist nicht in A enthalten.

Binäre Suche: Korrektheit des Algorithmus

Behauptung

Angenommen $A[0] \leq x \leq A[n-1]$. *Dann:*

1. *Ist x in A enthalten, so liefert* BINARYSEARCH *den Index k mit $A[k] = x$.*
2. *Ist x nicht in A enthalten, so liefert* BINARYSEARCH *den Index k mit $A[k] < x < A[k+1]$.*

Beweis:

- Wir wissen inzwischen, dass BINARYSEARCH immer terminiert.
  Die zwei möglichen Ausgaben sind (1) "$A[k] = x$" und (2) "$A[\ell] < x < A[\ell + 1]$".
- Die Ausgabe (1) tritt offensichtlich nur ein, wenn $A[k] = x$ ist (für $0 \leq k \leq n-1$).
- Zu Beginn der Schleife gilt stets $A[\ell] < x < A[r]$ nach Invariante (I2).
- Bei Ausgabe (2) gilt $\ell + 1 = r$, und somit $A[\ell] < x < A[\ell + 1]$.
  Es ist also kein weiterer Index mehr verfügbar, an dem x liegen könnte.
  Das bedeutet, x ist nicht in A enthalten.
- Somit tritt Ausgabe (1) genau dann ein, wenn $A[k] = x$ ist (für $0 \leq k \leq n-1$).

Binäre Suche: Korrektheit des Algorithmus

Behauptung

Angenommen $A[0] \leq x \leq A[n-1]$. Dann:

1. *Ist x in A enthalten, so liefert* BinarySearch *den Index k mit $A[k] = x$.*
2. *Ist x nicht in A enthalten, so liefert* BinarySearch *den Index k mit $A[k] < x < A[k+1]$.*

Beweis:

- Wir wissen inzwischen, dass BinarySearch immer terminiert.
  Die zwei möglichen Ausgaben sind (1) "$A[k] = x$" und (2) "$A[\ell] < x < A[\ell+1]$".
- Die Ausgabe (1) tritt offensichtlich nur ein, wenn $A[k] = x$ ist (für $0 \leq k \leq n-1$).
- Zu Beginn der Schleife gilt stets $A[\ell] < x < A[r]$ nach Invariante (I2).
- Bei Ausgabe (2) gilt $\ell + 1 = r$, und somit $A[\ell] < x < A[\ell+1]$.
  Es ist also kein weiterer Index mehr verfügbar, an dem x liegen könnte.
  Das bedeutet, x ist nicht in A enthalten.
- Somit tritt Ausgabe (1) genau dann ein, wenn $A[k] = x$ ist (für $0 \leq k \leq n-1$).
- Dann muss zwangsläufig Ausgabe (2) eintreten, wenn x nicht in A enthalten ist. $\square$

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ *Schleifendurchläufen.*

Binäre Suche: Aufwand

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?
- Falls der Algorithmus in einem Schleifendurchlauf nicht terminiert, wird am Ende entweder $\ell := \lfloor \frac{\ell + r}{2} \rfloor$ oder $r := \lfloor \frac{\ell + r}{2} \rfloor$ gesetzt.

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?
- Falls der Algorithmus in einem Schleifendurchlauf nicht terminiert, wird am Ende entweder $\ell := \lfloor \frac{\ell+r}{2} \rfloor$ oder $r := \lfloor \frac{\ell+r}{2} \rfloor$ gesetzt.
- Nun gilt:

$$
\begin{aligned}
\max\{r - \lfloor \tfrac{\ell+r}{2} \rfloor - 1,\ \lfloor \tfrac{\ell+r}{2} \rfloor - \ell - 1\} &\leq \max\{r - \tfrac{\ell+r}{2} - \tfrac{1}{2},\ \tfrac{\ell+r}{2} - \ell - 1\} \\
&= \max\{\tfrac{r-\ell-1}{2},\ \tfrac{r-\ell}{2} - 1\} \\
&= \tfrac{r-\ell-1}{2}.
\end{aligned}
$$

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?
- Falls der Algorithmus in einem Schleifendurchlauf nicht terminiert, wird am Ende entweder $\ell := \lfloor \frac{\ell+r}{2} \rfloor$ oder $r := \lfloor \frac{\ell+r}{2} \rfloor$ gesetzt.
- Nun gilt:

$$\begin{aligned} \max\{r - \lfloor \tfrac{\ell+r}{2} \rfloor - 1,\ \lfloor \tfrac{\ell+r}{2} \rfloor - \ell - 1\} &\leq \max\{r - \tfrac{\ell+r}{2} - \tfrac{1}{2},\ \tfrac{\ell+r}{2} - \ell - 1\} \\ &= \max\{\tfrac{r-\ell-1}{2},\ \tfrac{r-\ell}{2} - 1\} \\ &= \tfrac{r-\ell-1}{2}. \end{aligned}$$

  Also wird die Anzahl der zu durchsuchenden Elemente mindestens halbiert.

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?
- Falls der Algorithmus in einem Schleifendurchlauf nicht terminiert, wird am Ende entweder $\ell := \lfloor \frac{\ell+r}{2} \rfloor$ oder $r := \lfloor \frac{\ell+r}{2} \rfloor$ gesetzt.
- Nun gilt:

$$\begin{aligned}
\max\{r - \lfloor \tfrac{\ell+r}{2} \rfloor - 1,\ \lfloor \tfrac{\ell+r}{2} \rfloor - \ell - 1\} &\leq \max\{r - \tfrac{\ell+r}{2} - \tfrac{1}{2},\ \tfrac{\ell+r}{2} - \ell - 1\} \\
&= \max\{\tfrac{r-\ell-1}{2},\ \tfrac{r-\ell}{2} - 1\} \\
&= \tfrac{r-\ell-1}{2}.
\end{aligned}$$

  Also wird die Anzahl der zu durchsuchenden Elemente mindestens halbiert.
- Also nach k Durchläufen: $r - \ell - 1 \leq \lfloor \frac{n}{2^k} \rfloor$.

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?
- Falls der Algorithmus in einem Schleifendurchlauf nicht terminiert, wird am Ende entweder $\ell := \lfloor \frac{\ell+r}{2} \rfloor$ oder $r := \lfloor \frac{\ell+r}{2} \rfloor$ gesetzt.
- Nun gilt:

$$\begin{aligned} \max\{r - \lfloor \tfrac{\ell+r}{2} \rfloor - 1,\ \lfloor \tfrac{\ell+r}{2} \rfloor - \ell - 1\} &\leq \max\{r - \tfrac{\ell+r}{2} - \tfrac{1}{2},\ \tfrac{\ell+r}{2} - \ell - 1\} \\ &= \max\{\tfrac{r-\ell-1}{2},\ \tfrac{r-\ell}{2} - 1\} \\ &= \tfrac{r-\ell-1}{2}. \end{aligned}$$

  Also wird die Anzahl der zu durchsuchenden Elemente mindestens halbiert.
- Also nach k Durchläufen: $r - \ell - 1 \leq \lfloor \frac{n}{2^k} \rfloor$.
- Algorithmus endet sicher (sagen wir im $(k+1)$-ten Durchlauf), falls $\ell + 1 = r$.

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?
- Falls der Algorithmus in einem Schleifendurchlauf nicht terminiert, wird am Ende entweder $\ell := \lfloor \frac{\ell+r}{2} \rfloor$ oder $r := \lfloor \frac{\ell+r}{2} \rfloor$ gesetzt.
- Nun gilt:

$$\begin{aligned}\max\{r - \lfloor \tfrac{\ell+r}{2} \rfloor - 1,\ \lfloor \tfrac{\ell+r}{2} \rfloor - \ell - 1\} &\leq \max\{r - \tfrac{\ell+r}{2} - \tfrac{1}{2},\ \tfrac{\ell+r}{2} - \ell - 1\} \\ &= \max\{\tfrac{r-\ell-1}{2},\ \tfrac{r-\ell}{2} - 1\} \\ &= \tfrac{r-\ell-1}{2}.\end{aligned}$$

  Also wird die Anzahl der zu durchsuchenden Elemente mindestens halbiert.
- Also nach k Durchläufen: $r - \ell - 1 \leq \lfloor \frac{n}{2^k} \rfloor$.
- Algorithmus endet sicher (sagen wir im $(k+1)$-ten Durchlauf), falls $\ell + 1 = r$.
  Dies ist garantiert, falls $\frac{n}{2^k} < 1$ bzw. $\log_2(n) < k$.

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?
- Falls der Algorithmus in einem Schleifendurchlauf nicht terminiert, wird am Ende entweder $\ell := \lfloor \frac{\ell+r}{2} \rfloor$ oder $r := \lfloor \frac{\ell+r}{2} \rfloor$ gesetzt.
- Nun gilt:

$$\begin{aligned}\max\{r - \lfloor \tfrac{\ell+r}{2} \rfloor - 1,\ \lfloor \tfrac{\ell+r}{2} \rfloor - \ell - 1\} &\leq \max\{r - \tfrac{\ell+r}{2} - \tfrac{1}{2},\ \tfrac{\ell+r}{2} - \ell - 1\} \\ &= \max\{\tfrac{r-\ell-1}{2},\ \tfrac{r-\ell}{2} - 1\} \\ &= \tfrac{r-\ell-1}{2}.\end{aligned}$$

  Also wird die Anzahl der zu durchsuchenden Elemente mindestens halbiert.
- Also nach k Durchläufen: $r - \ell - 1 \leq \lfloor \frac{n}{2^k} \rfloor$.
- Algorithmus endet sicher (sagen wir im $(k+1)$-ten Durchlauf), falls $\ell + 1 = r$.
  Dies ist garantiert, falls $\frac{n}{2^k} < 1$ bzw. $\log_2(n) < k$.
- Da k ganzzahlig, $1 + \lfloor \log_2(n) \rfloor \leq k$.

Satz

Die binäre Suche findet ein Element x in A nach höchstens $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufen.

Beweis:

- Zu gegebenen Zeitpunkt im Ablauf des Algorithmus:
  Die Anzahl der zu durchsuchenden Elemente ist $r - \ell - 1$.
  Frage: Wie verändert sich $r - \ell - 1$ nach jedem Durchlauf?
- Falls der Algorithmus in einem Schleifendurchlauf nicht terminiert, wird am Ende entweder $\ell := \lfloor \frac{\ell+r}{2} \rfloor$ oder $r := \lfloor \frac{\ell+r}{2} \rfloor$ gesetzt.
- Nun gilt:

$$\begin{aligned}\max\{r - \lfloor \tfrac{\ell+r}{2} \rfloor - 1,\ \lfloor \tfrac{\ell+r}{2} \rfloor - \ell - 1\} &\leq \max\{r - \tfrac{\ell+r}{2} - \tfrac{1}{2},\ \tfrac{\ell+r}{2} - \ell - 1\} \\ &= \max\{\tfrac{r-\ell-1}{2},\ \tfrac{r-\ell}{2} - 1\} \\ &= \tfrac{r-\ell-1}{2}.\end{aligned}$$

  Also wird die Anzahl der zu durchsuchenden Elemente mindestens halbiert.
- Also nach k Durchläufen: $r - \ell - 1 \leq \lfloor \frac{n}{2^k} \rfloor$.
- Algorithmus endet sicher (sagen wir im $(k+1)$-ten Durchlauf), falls $\ell + 1 = r$.
  Dies ist garantiert, falls $\frac{n}{2^k} < 1$ bzw. $\log_2(n) < k$.
- Da k ganzzahlig, $1 + \lfloor \log_2(n) \rfloor \leq k$.
- Da der Algorithmus im $(k+1)$-ten Durchlauf endet, folgt der Satz. $\square$

Was genau bestimmt eigentlich den Aufwand eines Algorithmus?

Was genau bestimmt eigentlich den Aufwand eines Algorithmus?

Wir legen ein vereinfachtes Rechnermodell zugrunde, das RAM-Modell.

Was genau bestimmt eigentlich den Aufwand eines Algorithmus?

Wir legen ein vereinfachtes Rechnermodell zugrunde, das RAM-Modell.

- RAM steht für Random Access Machine.
  Auf jedes Speicherelement kann also in der gleichen Zeit zugegriffen werden.

RAM-Modell

Was genau bestimmt eigentlich den Aufwand eines Algorithmus?

Wir legen ein vereinfachtes Rechnermodell zugrunde, das RAM-Modell.

- RAM steht für Random Access Machine.
  Auf jedes Speicherelement kann also in der gleichen Zeit zugegriffen werden.
- Anweisungen werden sequentiell (nacheinander) ausgeführt,
  nicht parallel.

Was genau bestimmt eigentlich den Aufwand eines Algorithmus?

Wir legen ein vereinfachtes Rechnermodell zugrunde, das RAM-Modell.

- RAM steht für Random Access Machine.
  Auf jedes Speicherelement kann also in der gleichen Zeit zugegriffen werden.
- Anweisungen werden sequentiell (nacheinander) ausgeführt, nicht parallel.
- Elementaroperationen in Registern beinhalten, ähnlich einem realen Rechner, Addieren, Subtrahieren, Multiplizieren, Dividieren, Reste, Runden, Datenblöcke laden, speichern und kopieren, Kontrollanweisungen (if, while,...), Rückgabe etc.

Was genau bestimmt eigentlich den Aufwand eines Algorithmus?

Wir legen ein vereinfachtes Rechnermodell zugrunde, das RAM-Modell.

- RAM steht für Random Access Machine.
  Auf jedes Speicherelement kann also in der gleichen Zeit zugegriffen werden.
- Anweisungen werden sequentiell (nacheinander) ausgeführt, nicht parallel.
- Elementaroperationen in Registern beinhalten, ähnlich einem realen Rechner, Addieren, Subtrahieren, Multiplizieren, Dividieren, Reste, Runden, Datenblöcke laden, speichern und kopieren, Kontrollanweisungen (if, while,...), Rückgabe etc.
- Wir nehmen an, dass jeder dieser Befehle eine feste Zeit („einen Takt") benötigt.

Was genau bestimmt eigentlich den Aufwand eines Algorithmus?

Wir legen ein vereinfachtes Rechnermodell zugrunde, das RAM-Modell.

- RAM steht für Random Access Machine.
  Auf jedes Speicherelement kann also in der gleichen Zeit zugegriffen werden.
- Anweisungen werden sequentiell (nacheinander) ausgeführt, nicht parallel.
- Elementaroperationen in Registern beinhalten, ähnlich einem realen Rechner, Addieren, Subtrahieren, Multiplizieren, Dividieren, Reste, Runden, Datenblöcke laden, speichern und kopieren, Kontrollanweisungen (if, while,...), Rückgabe etc.
- Wir nehmen an, dass jeder dieser Befehle eine feste Zeit („einen Takt") benötigt.
- Zahlen: Ganze Zahlen (Integer) und Fließkommazahlen (Float).

Was genau bestimmt eigentlich den Aufwand eines Algorithmus?

Wir legen ein vereinfachtes Rechnermodell zugrunde, das RAM-Modell.

- RAM steht für Random Access Machine.
  Auf jedes Speicherelement kann also in der gleichen Zeit zugegriffen werden.
- Anweisungen werden sequentiell (nacheinander) ausgeführt, nicht parallel.
- Elementaroperationen in Registern beinhalten, ähnlich einem realen Rechner, Addieren, Subtrahieren, Multiplizieren, Dividieren, Reste, Runden, Datenblöcke laden, speichern und kopieren, Kontrollanweisungen (if, while,...), Rückgabe etc.
- Wir nehmen an, dass jeder dieser Befehle eine feste Zeit („einen Takt") benötigt.
- Zahlen: Ganze Zahlen (Integer) und Fließkommazahlen (Float).
- Unterschiede bei (modernen) realen Rechnern:
  - Komplexere Befehle in (fast) konstanter Zeit.
  - Speicherhierachie (Festplatte, RAM, Cache).
  - Parallelisierung.
  - Endlicher Speicher.

Nochmal binäre Suche

Wir haben $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufe.
Was genau zählen wir damit?

Wir haben $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufe.
Was genau zählen wir damit?

$\text{BINARYSEARCH}(A, x)$
  [...]
  while true **do**   1 Operation
    if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$" ≥ 3 Operationen
    $k := \lfloor \frac{\ell + r}{2} \rfloor$ ≥ 4 Operationen
    if $A[k] = x$ **then return** "$A[k] = x$" ≥ 2 Operationen
    if $A[k] < x$ **then** $\ell := k$ **else** $r := k$ ≥ 2 Operationen

Wir haben $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufe.
Was genau zählen wir damit?

$\textsc{BinarySearch}(A, x)$
[...]
while true **do** — 1 Operation
if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$" — ≥ 3 Operationen
$k := \lfloor \frac{\ell + r}{2} \rfloor$ — ≥ 4 Operationen
if $A[k] = x$ **then return** "$A[k] = x$" — ≥ 2 Operationen
if $A[k] < x$ **then** $\ell := k$ **else** $r := k$ — ≥ 2 Operationen

Jeder Schleifendurchlauf enspricht also ≥ 12 Prozessortakten,
Gesamtaufwand $\geq 12 \cdot (2 + \lfloor \log_2(n) \rfloor)$.

Wir haben $2 + \lfloor \log_2(n) \rfloor$ Schleifendurchläufe.
Was genau zählen wir damit?

$\textsc{BinarySearch}(A, x)$
  $[\ldots]$
  while true **do** 1 Operation
    if $\ell + 1 = r$ **then return** "$A[\ell] < x < A[\ell + 1]$" ≥ 3 Operationen
    $k := \lfloor \frac{\ell+r}{2} \rfloor$ ≥ 4 Operationen
    if $A[k] = x$ **then return** "$A[k] = x$" ≥ 2 Operationen
    if $A[k] < x$ **then** $\ell := k$ **else** $r := k$ ≥ 2 Operationen

Jeder Schleifendurchlauf enspricht also ≥ 12 Prozessortakten,
Gesamtaufwand $\geq 12 \cdot (2 + \lfloor \log_2(n) \rfloor)$.

- Genaue Zahl schwierig, hängt von Speicherstruktur ab (Daten bereits im Register?)
- Bei realen Rechnern stark maschinenabhängig durch unterschiedliche Speicherhierarchien und Befehlssätze.
- Daher ist man in der Regel nur am asymptotischen Aufwand interessiert („Aufwand bis auf konstante Faktoren").
- Ziel ist nur, die Größenordnung der Laufzeit vorherzusagen.

Größenordnung

Heuristik:
Ein konstanter Faktor macht für große Eingabewerte weniger Unterschied als eine andere Funktionsklasse.

Heuristik:
Ein konstanter Faktor macht für große Eingabewerte weniger Unterschied als eine andere Funktionsklasse.

Betrachte

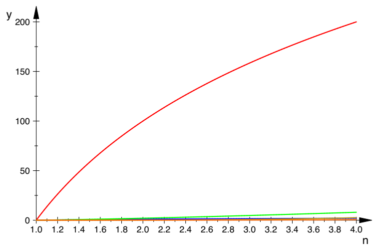
$$
\begin{aligned}
f_1(n) &= \log_2(n), \\
f_2(n) &= 100 \cdot \log_2(n), \\
f_3(n) &= n \cdot \log_2(n), \\
f_4(n) &= \frac{1}{100} \cdot n^4, \\
f_5(n) &= \frac{1}{100} \cdot \exp(n).
\end{aligned}
$$

Heuristik:
Ein konstanter Faktor macht für große Eingabewerte weniger Unterschied als eine andere Funktionsklasse.

Betrachte

$$\begin{aligned} f_1(n) &= \log_2(n), \\ f_2(n) &= 100 \cdot \log_2(n), \\ f_3(n) &= n \cdot \log_2(n), \\ f_4(n) &= \frac{1}{100} \cdot n^4, \\ f_5(n) &= \frac{1}{100} \cdot \exp(n). \end{aligned}$$

$n = 4$

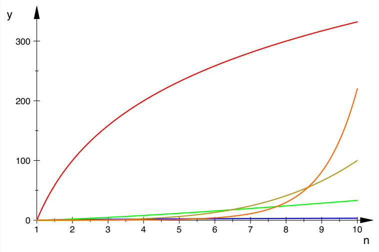


Heuristik:
Ein konstanter Faktor macht für große Eingabewerte weniger Unterschied als eine andere Funktionsklasse.

Betrachte

$$f_1(n) = \log_2(n),$$
$$f_2(n) = 100 \cdot \log_2(n),$$
$$f_3(n) = n \cdot \log_2(n),$$
$$f_4(n) = \frac{1}{100} \cdot n^4,$$
$$f_5(n) = \frac{1}{100} \cdot \exp(n).$$

$n = 10$

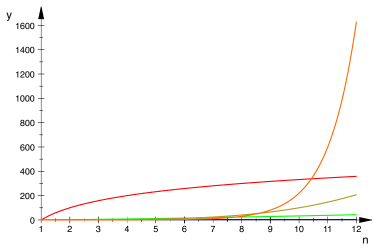


Heuristik:
Ein konstanter Faktor macht für große Eingabewerte weniger Unterschied als eine andere Funktionsklasse.

Betrachte

$$f_1(n) = \log_2(n),$$
$$f_2(n) = 100 \cdot \log_2(n),$$
$$f_3(n) = n \cdot \log_2(n),$$
$$f_4(n) = \frac{1}{100} \cdot n^4,$$
$$f_5(n) = \frac{1}{100} \cdot \exp(n).$$

$n = 12$



Heuristik:
Ein konstanter Faktor macht für große Eingabewerte weniger Unterschied als eine andere Funktionsklasse.

Betrachte

$$f_1(n) = \log_2(n),$$
$$f_2(n) = 100 \cdot \log_2(n),$$
$$f_3(n) = n \cdot \log_2(n),$$
$$f_4(n) = \frac{1}{100} \cdot n^4,$$
$$f_5(n) = \frac{1}{100} \cdot \exp(n).$$

$n = 20$