Script generated by TTT

Title: T?ubig: GAD (24.04.2012)

Date: Tue Apr 24 14:28:34 CEST 2012

Duration: 91:04 min

Pages: 41

Effizienzmessung

Für ein gegebenes Problem sei $\mathcal{I}_n$ die Menge der Instanzen der Größe n.

Effizienzmaße:

- Worst case:
$$t(n) = \max\{T(i) : i \in \mathcal{I}_n\}$$
- Average case:
$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i)$$

Effizienzmaße: Vor- und Nachteile

- worst case:
liefert Garantie für die Effizienz des Algorithmus
(auch wichtig für Robustheit)
- average case:
nicht unbedingt übereinstimmend mit dem "typischen Fall" in der Praxis (evt. schwer formal zu erfassen), kann durch Wahrscheinlichkeitsverteilung noch verallgemeinert werden
- exakte Formeln für $t(n)$ meist sehr aufwendig

$\Rightarrow$ betrachte asymptotisches Wachstum

Asymptotische Notation

- $f(n)$ und $g(n)$ haben gleiche Wachstumsrate, falls für große n das Verhältnis der beiden nach oben und unten durch Konstanten beschränkt ist, d.h.,
$$\exists c, d \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad c \leq \frac{f(n)}{g(n)} \leq d$$
- $f(n)$ wächst schneller als $g(n)$, wenn es für alle positiven Konstanten c ein n_0 gibt, ab dem $f(n) \geq c \cdot g(n)$ für $n \geq n_0$ gilt, d.h.,
$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Beispiel

n^2 und $5n^2 - 7n$ haben das gleiche Wachstum, da für alle $n \geq 2$ $\frac{1}{5} < \frac{(5n^2-7n)}{n^2} < 5$ und $\frac{1}{5} < \frac{n^2}{(5n^2-7n)} < 5$ gilt.

Asymptotische Notation

- Warum die Betrachtung der Wachstumsrate und die Forderung nur für genügend große n?

  Ziel effizienter Algorithmen: Lösung großer Probleminstanzen

  gesucht: Verfahren, die für große Instanzen noch effizient sind

  Für große n sind Verfahren mit kleinerer Wachstumsrate besser.
- Warum Verzicht auf konstante Faktoren?

  Unser Maschinenmodell ist nur eine Abstraktion von echten Computern und kann die eigentliche Rechenzeit sowieso nur bis auf konstante Faktoren bestimmen.

  Deshalb ist es in den meisten Fällen sinnvoll, Algorithmen mit gleichem Wachstum erstmal als gleichwertig zu betrachten.
- Außerdem: Laufzeitangabe durch einfache Funktionen

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

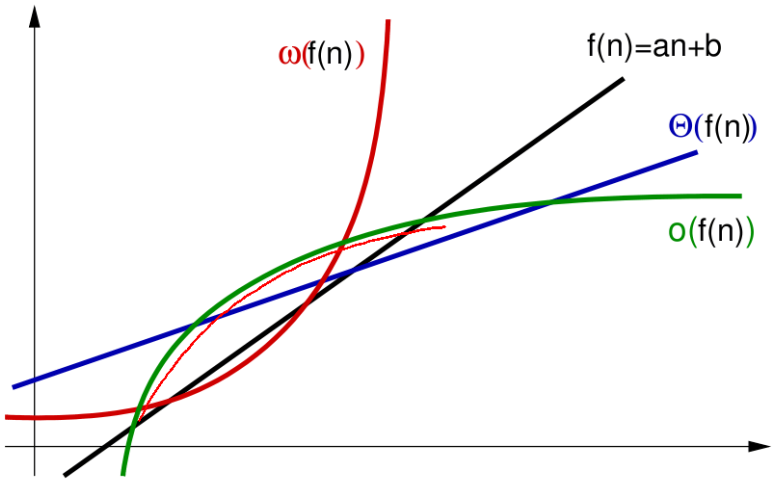
$$\begin{aligned}
O(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\
\Omega(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}
\end{aligned}$$

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

$$\begin{aligned}
o(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\
\omega(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}
\end{aligned}$$

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben
$\Rightarrow$ Forderung: $\exists n_0 > 0 : \forall n \geq n_0 : f(n) > 0$
Manchmal auch: $\forall n : f(n) \geq 0$

Asymptotische Notation



Asymptotische Notation

Beispiel

- $5n^2 - 7n \in O(n^2)$, $n^2/10 + 100n \in O(n^2)$, $4n^2 \in O(n^3)$
- $5n^2 - 7n \in \Omega(n^2)$, $n^3 \in \Omega(n^2)$, $n \log n \in \Omega(n)$
- $5n^2 - 7n \in \Theta(n^2)$
- $\log n \in o(n)$, $n^3 \in o(2^n)$
- $n^5 \in \omega(n^3)$, $2^{2n} \in \omega(2^n)$

Asymptotische Notation als Platzhalter

- statt $g(n) \in O(f(n))$ schreibt man auch oft $g(n) = O(f(n))$
- für $f(n) + g(n)$ mit $g(n) \in o(h(n))$ schreibt man auch $f(n) + g(n) = f(n) + o(h(n))$
- statt $O(f(n)) \subseteq O(g(n))$ schreibt man auch $O(f(n)) = O(g(n))$

Beispiel

$$n^3 + n \quad = \quad n^3 + o(n^3) \quad = \quad (1 + o(1))n^3 \quad = \quad O(n^3)$$

O-Notations"gleichungen" sollten nur von links nach rechts gelesen werden!

Übersicht

Wachstum von Polynomen

Beweis.

Zu zeigen: $p(n) \in O(n^k)$ und $p(n) \in \Omega(n^k)$

$p(n) \in O(n^k)$:

Für $n \geq 1$ gilt:

$$p(n) \quad \leq \quad \sum_{i=0}^{k} |a_i| \cdot n^i \quad \leq \quad n^k \sum_{i=0}^{k} |a_i|$$

Also ist die Definition

$$O(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

mit $c = \sum_{i=0}^{k} |a_i|$ und $n_0 = 1$ erfüllt.

Wachstum von Polynomen

Beweis.

$p(n) \in \Omega(n^k)$:

Sei $A = \sum_{i=0}^{k-1} |a_i|$.

Für positive n gilt dann:

$$p(n) \quad \geq \quad a_k n^k - A n^{k-1} \quad = \quad \frac{a_k}{2} n^k + n^{k-1} \left(\frac{a_k}{2} n - A \right)$$

Also ist die Definition

$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

mit $c = a_k/2$ und $n_0 > 2A/a_k$ erfüllt. □

Rechenregeln für $\mathcal{O}$-Notation

Für Funktionen $f(n)$ (bzw. $g(n)$) mit der Eigenschaft
$\exists n_0 > 0 : \ \forall n \geq n_0 : \ f(n) > 0$ gilt:

Lemma

- $c \cdot f(n) \in \Theta(f(n))$ *für jede Konstante* $c > 0$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n))$ *falls* $g(n) \in \mathcal{O}(f(n))$

Die Ausdrücke sind auch korrekt für Ω *statt* $\mathcal{O}$.

Vorsicht, der letzte heißt dann

- $\Omega(f(n) + g(n)) = \Omega(f(n))$ *falls* $g(n) \in \mathcal{O}(f(n))$

Aber: Vorsicht bei induktiver Anwendung!

Induktions"beweis"

Behauptung:

$$\sum_{i=1}^{n} i = \mathcal{O}(n)$$

"Beweis": Sei $f(n) = n + f(n-1)$ und $f(1) = 1$.

Ind.anfang: $f(1) = \mathcal{O}(1)$

Ind.vor.: Es gelte $f(n-1) = \mathcal{O}(n-1)$

Ind.schritt: Dann gilt

$$f(n) = n + f(n-1) = n + \mathcal{O}(n-1) = \mathcal{O}(n)$$

Also ist

$$f(n) = \sum_{i=1}^{n} i = \mathcal{O}(n)$$

FALSCH!

Rechenregeln für $\mathcal{O}$-Notation

Lemma

Seien f *und* g *differenzierbar.*

Dann gilt

- *falls* $f'(n) \in \mathcal{O}(g'(n))$, *dann auch* $f(n) \in \mathcal{O}(g(n))$
- *falls* $f'(n) \in \Omega(g'(n))$, *dann auch* $f(n) \in \Omega(g(n))$
- *falls* $f'(n) \in o(g'(n))$, *dann auch* $f(n) \in o(g(n))$
- *falls* $f'(n) \in \omega(g'(n))$, *dann auch* $f(n) \in \omega(g(n))$

Umgekehrt gilt das im Allgemeinen nicht!

Rechenbeispiele für $\mathcal{O}$-Notation

Beispiel

- 1. Lemma:
  - $n^3 - 3n^2 + 2n \in \mathcal{O}(n^3)$
  - $\mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2/2 + n/2) = \mathcal{O}(n^2)$
- 2. Lemma:
  Aus $\log n \in \mathcal{O}(n)$ folgt $n \log n \in \mathcal{O}(n^2)$.
- 3. Lemma:
  - $(\log n)' = 1/n$, $(n)' = 1$ und $1/n \in \mathcal{O}(1)$.
    $\Rightarrow \log n \in \mathcal{O}(n)$

Übersicht

Rechenbeispiele für O-Notation

Beispiel

- 1. Lemma:
  - $n^3 - 3n^2 + 2n \in O(n^3)$
  - $O(\sum_{i=1}^{n} i) = O(n^2/2 + n/2) = O(n^2)$
- 2. Lemma:
  Aus $\log n \in O(n)$ folgt $n \log n \in O(n^2)$.
- 3. Lemma:
  - $(\log n)' = 1/n$, $(n)' = 1$ und $1/n \in O(1)$.
    $\Rightarrow \log n \in O(n)$

Rechenbeispiele für O-Notation

Beispiel

- 1. Lemma:
  - $n^3 - 3n^2 + 2n \in O(n^3)$
  - $O(\sum_{i=1}^{n} i) = O(n^2/2 + n/2) = O(n^2)$
- 2. Lemma:
  Aus $\log n \in O(n)$ folgt $n \log n \in O(n^2)$.
- 3. Lemma:
  - $(\log n)' = 1/n$, $(n)' = 1$ und $1/n \in O(1)$.
    $\Rightarrow \log n \in O(n)$

von Neumann / Princeton-Architektur

John von Neumann
Quelle: http://wikipedia.org/

1945 János / Johann / John von Neumann:
Entwurf eines Rechnermodells: EDVAC
(Electronic Discrete Variable Automatic Computer)

Programm und Daten teilen sich
einen gemeinsamen Speicher

Besteht aus

- Arithmetic Logic Unit (ALU):
  Rechenwerk / Prozessor
- Control Unit: Steuerwerk (Befehlsinterpreter)
- Memory: eindimensional adressierbarer
  Speicher
- I/O Unit: Ein-/Ausgabewerk

Random Access Machine (RAM)

Was ist eigentlich ein Rechenschritt?

⇒ Abstraktion mit Hilfe von Rechnermodellen

Bsp. Turingmaschine:
kann aber nicht auf beliebige Speicherzellen zugreifen,
nur an der aktuellen Position des Lese-/Schreibkopfs

1963 John Shepherdson, Howard Sturgis (u.a.):

Random Access Machine (RAM)

(beschränkte Registeranzahl)
Prozessor
⇓
Linear adressierbarer Speicher
(unbeschränkt)

RAM: Speicher

- Unbeschränkt viele Speicherzellen (words) $S[0], S[1], S[2], \ldots$, von denen zu jedem Zeitpunkt nur endlich viele benutzt werden
- aber: Speicherung beliebig großer Zahlen würde zu unrealistischen Algorithmen führen
- Jede Speicherzelle kann bei Eingabegröße n eine Zahl mit $O(\log n)$ Bits speichern (angemessen: für konstant große Zellen würde man nur einen Faktor $O(\log n)$ bei der Rechenzeit erhalten).

⇒ Gespeicherte Werte stellen polynomiell in n (Eingabegröße) beschränkte Zahlen dar.

- sinnvoll für Speicherung von Array-Indizes

Begrenzter Parallelismus:

- sequentielles Maschinenmodell, aber
- Verknüpfung logarithmisch vieler Bits in konstanter Zeit

RAM: Aufbau

Prozessor:

- Beschränkte Anzahl an Registern $R_1, \ldots, R_k$
- Instruktionszeiger zum nächsten Befehl

Programm:

- Nummerierte Liste von Befehlen
  (Adressen in Sprungbefehlen entsprechen dieser Nummerierung)

Eingabe:

- steht in Speicherzellen $S[1], \ldots, S[R_1]$

Modell / Reale Rechner:

- unendlicher / endlicher Speicher
- Abhängigkeit / Unabhängigkeit der Größe der Speicherzellen von der Eingabegröße

RAM: Befehle

Annahme:

- Jeder Befehl dauert genau eine Zeiteinheit.
- Laufzeit ist Anzahl ausgeführter Befehle

Befehlssatz:

- Registerzuweisung:
  $R_i := c$ Registerzuweisung für eine Konstante c
  $R_i := R_j$ Zuweisung von einem Register an ein anderes

- Speicherzugriff: (alle Zugriffe benötigen gleich viel Zeit)
  $R_i := S[R_j]$ lesend: lädt Inhalt von $S[R_j]$ in R_i
  $S[R_j] := R_i$ schreibend: speichert Inhalt von R_i in $S[R_j]$

RAM: Befehle

- Arithmetische / logische Operationen:
  $R_i := R_j \text{ op } R_k$ binäre Rechenoperation
  op $\in \{+, -, \cdot, \oplus, /, \%, \wedge, \vee, \ldots\}$ oder
  op $\in \{<, \leq, =, \geq, >\}$: 1$\hat{=}$wahr, 0$\hat{=}$falsch
  $R_i := \text{op } R_j$ unäre Rechenoperation
  op $\in \{-, \neg\}$
- Sprünge:

  jump x Springe zu Position x

  jumpz x R_i Falls $R_i = 0$, dann springe zu Position x

  jumpi R_j Springe zur Adresse aus R_j

Das entspricht Assembler-Code von realen Maschinen!

Maschinenmodell

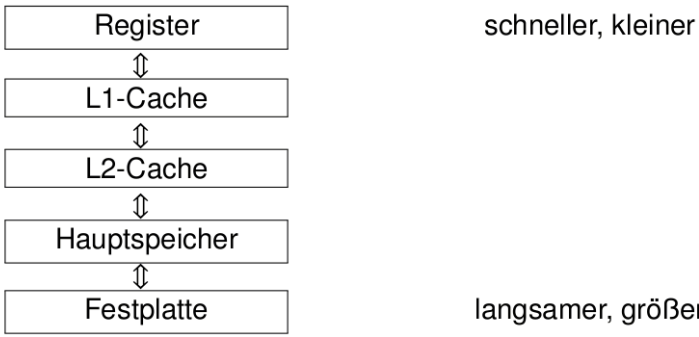
RAM-Modell

- Modell für die ersten Computer
- entspricht eigentlich der Harvard-Architektur (separater Programmspeicher)
- Sein Äquivalent zur Universellen Turing-Maschine, das Random Access Stored Program (RASP) Modell entspricht der von Neumann-Architektur und hat große Ähnlichkeit mit üblichen Rechnern.

Aber: Speicherhierarchien und Multicore-Prozessoren erfordern Anpassung des RAM-Modells

⇒ Algorithm Engineering, z.B. External-Memory Model

Speicherhierarchie



⇒ External-Memory Model

- begrenzter schneller Speicher mit M Zellen
- unbegrenzter (langsamer) externer Speicher
- I/O-Operationen transferieren B aufeinanderfolgende Worte

Pseudocode

Maschinencode / Assembler umständlich
⇒ besser: Programmiersprache wie Pascal, C++, Java, ...

Variablendeklarationen:

T v; Variable v vom Typ T

T v=x; initialisiert mit Wert x

Variablentypen:

- int, boolean, char, double, ...
- Klassen T, Interfaces I
- T[n]: Feld von Elementen vom Typ T (indexiert von 0 bis $n - 1$)

Pseudocode-Programme

Allokation und Deallokation von Speicherobjekten:

- v = new T($v_1, \ldots, v_k$); // implizit wird Konstruktor für T aufgerufen

Sprachkonstrukte: (C: Bedingung, I,J: Anweisungen)

- $v = A$; // Ergebnis von Ausdruck A wird an Variable v zugewiesen
- **if** (C) I **else** J
- **do** I **while** (C); **while**(C) I
- **for**($v = a$; $v < e$; v++) I
- **return** v;

Übersetzung in RAM-Befehle

- Zuweisungen lassen sich in konstant viele RAM-Befehle übersetzen, z.B.
  $a := a + bc$

  $R_1 := R_b * R_C$;
  $R_a := R_a + R_1$
  - R_a, R_b, R_c: Register, in denen a, b und c gespeichert sind

- **if** (C) I **else** J

  eval(C); jumpz sElse R_c; trans(I); jump sEnd; trans(J)
  - eval(C): Befehle, die die Bedingung C auswerten und das Ergebnis in Register R_c hinterlassen
  - trans(I), trans(J): übersetzte Befehlsfolge für I und J
  - sElse: Adresse des 1. Befehls in trans(J)
    sEnd: Adresse des 1. Befehls nach trans(J)

Übersetzung in RAM-Befehle

- Zuweisungen lassen sich in konstant viele RAM-Befehle übersetzen, z.B.
  $a := a + bc$

  $R_1 := R_b * R_C$;
  $R_a := R_a + R_1$
  - R_a, R_b, R_c: Register, in denen a, b und c gespeichert sind

- **if** (C) I **else** J

  eval(C); jumpz sElse R_c; trans(I); jump sEnd; trans(J)
  - eval(C): Befehle, die die Bedingung C auswerten und das Ergebnis in Register R_c hinterlassen
  - trans(I), trans(J): übersetzte Befehlsfolge für I und J
  - sElse: Adresse des 1. Befehls in trans(J)
    sEnd: Adresse des 1. Befehls nach trans(J)

Übersetzung in RAM-Befehle

- **repeat** I **until** C

  trans(I); eval(C); jumpz sI R_c
  - trans(I): übersetzte Befehlsfolge für I
  - eval(C): Befehle, die die Bedingung C auswerten und das Ergebnis in Register R_c hinterlassen
  - sI: Adresse des ersten Befehls in trans(I)

- andere Schleifen lassen sich damit simulieren:

  while(C) I $\longrightarrow$ **if** C **then repeat** I **until** $\neg$C

  for($i = a$; $i \leq b$; i++) I $\longrightarrow$ $i := a$; **while**($i \leq b$) I; i++

Übersicht

worst case

Berechnung der worst-case-Laufzeit:

- $T(I)$ sei worst-case-Laufzeit für Konstrukt I
- $T(\text{elementare Zuweisung}) = O(1)$
- $T(\text{elementarer Vergleich}) = O(1)$
- $T(\textbf{return } x) = O(1)$
- $T(\textbf{new Typ}(\ldots)) = O(1) + O(T(\text{Konstruktor}))$
- $T(I_1; I_2) = T(I_1) + T(I_2)$
- $T(\textbf{if } (C)\ I_1\ \textbf{else}\ I_2) = O(T(C) + \max\{T(I_1), T(I_2)\})$
- $T(\textbf{for}(i = a; i < b; i{+}{+})\ I) = O\left(\sum_{i=a}^{b-1}(1 + T(I))\right)$
- $T(e.m(\ldots)) = O(1) + T(ss)$, wobei ss Rumpf von m

Beispiel: Vorzeichenausgabe

`signum(x)`
Input : Zahl $x \in \mathbb{R}$
Output : -1,0 bzw. 1 entsprechend dem Vorzeichen von x

```
if x < 0 then
  return -1
if x > 0 then
  return 1
return 0
```

Wir wissen:

$T(x < 0) = O(1)$
$T(\textbf{return } \textit{-1}) = O(1)$
$T(\textbf{if } (C)\ I) = O(T(C) + T(I))$

Also: $T(\textbf{if } (x < 0) \textbf{ return } -1) = O(1) + O(1) = O(1)$

Beispiel: Vorzeichenausgabe

`signum(x)`
Input : Zahl $x \in \mathbb{R}$
Output : -1,0 bzw. 1 entsprechend dem Vorzeichen von x

```
if x < 0 then
  return -1          O(1)
if x > 0 then
  return 1           O(1)
return 0             O(1)
```

$O(1 + 1 + 1) = O(1)$

Beispiel: Minimumsuche

`minimum`(A, n)

Input : Zahlenfolge in $A[0], \ldots, A[n-1]$
n: Anzahl der Zahlen
Output : Minimum der Zahlen

min $= A[0]$;	$O(1)$
for $(i = 1; i < n; i{+}{+})$ **do**	$O(\sum_{i=1}^{n-1}(1 + T(I)))$
⌊ **if** $A[i] <$ min **then** min $= A[i]$	$O(1)$ ↗
return min	$O(1)$

$$O\left(1 + \left(\sum_{i=1}^{n-1} 1\right) + 1\right) = O(n)$$

Beispiel: BubbleSort

Sortieren durch Aufsteigen

Vertausche in jeder Runde in der (verbleibenden) Eingabesequenz (hier vom Ende in Richtung Anfang) jeweils zwei benachbarte Elemente, die nicht in der richtigen Reihenfolge stehen

Beispiel

5	10	19	1	14	3
5	10	19	1	3	14
5	10	1	19	3	14
5	1	10	19	3	14
1	5	10	19	3	14

1	5	10	3	19	14
1	5	3	10	19	14
1	**3**	5	10	19	14
1	3	**5**	10	14	19
1	3	5	**10**	**14**	**19**

Beispiel: Sortieren

`BubbleSort`(A, n)

Input : n: Anzahl der Zahlen
$A[0], \ldots, A[n-1]$: Zahlenfolge
Output : Sortierte Zahlenfolge A

for $(i = 0; i < n-1; i{+}{+})$ **do**	$O(\sum_{i=0}^{n-2} T(I))$
⌊ **for** $(j = n-2; j \geq i; j{-}{-})$ **do**	$O(\sum_{j=i}^{n-2} T(I))$
⌊ **if** $A[j] > A[j+1]$ **then**	$O(1 + T(I))$
⌊ $x = A[j]$;	$O(1)$
⌊ $A[j] = A[j+1]$;	$O(1)$
⌊ $A[j+1] = x$;	$O(1)$

$$O\left(\sum_{i=0}^{n-2}\sum_{j=i}^{n-2} 1\right)$$

Beispiel: Sortieren

$$\begin{aligned} O\left(\sum_{i=0}^{n-2}\sum_{j=i}^{n-2} 1\right) &= \sum_{i=0}^{n-2}(n-i-1) \\ &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n-1)}{2} \\ &= \frac{n^2}{2} - \frac{n}{2} \\ &= O\left(n^2\right) \end{aligned}$$

Beispiel: Binäre Suche

`BinarySearch`(A, n, x)

Input : n: Anzahl der (sortierten) Zahlen
$A[0], \ldots, A[n-1]$: Zahlenfolge
x: gesuchte Zahl

Output : Index der gesuchten Zahl

$\ell = 0;$	$O(1)$
$r = n - 1;$	$O(1)$
while $(\ell \leq r)$ **do**	$O(\sum_{i=1}^{k} T(I))$
$\quad m = \lfloor (r + \ell)/2 \rfloor;$	$O(1)$ ↑
$\quad$ **if** $A[m] == x$ **then return** $m;$	$O(1)$
$\quad$ **if** $A[m] < x$ **then** $\ell = m + 1;$	$O(1)$
$\quad$ **else** $r = m - 1;$	$O(1)$
return -1	$O(1)$

$$O\left(\sum_{i=1}^{k} 1\right) = O(k)$$

Beispiel: Binäre Suche

Aber: Wie groß ist die Anzahl der Schleifendurchläufe k?

Größe des verbliebenen Suchintervalls $(r - \ell + 1)$ nach Iteration i:

$$\begin{aligned} s_0 &= n \\ s_{i+1} &\leq \lfloor s_i/2 \rfloor \end{aligned}$$

Bei $s_i < 1$ endet der Algorithmus.

$$\Rightarrow \quad k \leq \log_2 n$$

Gesamtkomplexität: $O(\log n)$