

Script generated by TTT

Title: Täubig: GAD (15.04.2014)

Date: Tue Apr 15 13:50:43 CEST 2014

Duration: 124:56 min

Pages: 45

Übersicht

- 3 Effizienz
 - Effizienzmaße
 - Rechenregeln für O -Notation
 - Maschinenmodell
 - Laufzeitanalyse
 - Durchschnittliche Laufzeit
 - Erwartete Laufzeit

Effizienzmessung

Ziel:

- Beschreibung der Performance von Algorithmen
- möglichst genau, aber in kurzer und einfacher Form

Exakte Spezifikation der Laufzeit eines Algorithmus (bzw. einer DS-Operation):

- Menge \mathcal{I} der Instanzen
- Laufzeit des Algorithmus $T : \mathcal{I} \mapsto \mathbb{N}$

Problem: T sehr schwer exakt bestimmbar bzw. beschreibbar

Lösung: **Gruppierung** der Instanzen (meist nach **Größe**)

Eingabekodierung

Bei Betrachtung der **Länge** der Eingabe:

Vorsicht bei der **Kodierung**!

Beispiel (Primfaktorisation)

Gegeben: Zahl $x \in \mathbb{N}$

Gesucht: Primfaktoren von x (Primzahlen p_1, \dots, p_k mit $x = \prod_{i=1}^k p_i^{e_i}$)

Bekannt als hartes Problem (wichtig für RSA-Verschlüsselung!)

Eingabekodierung - Beispiel Primfaktorisierung

Beispiel (Primfaktorisierung)

Trivialer Algorithmus

Teste von $y = 2$ bis $\lfloor \sqrt{x} \rfloor$ alle Zahlen, ob diese x teilen und wenn ja, dann bestimme wiederholt das Ergebnis der Division bis die Teilung nicht mehr ohne Rest möglich ist

Laufzeit: \sqrt{x} Teilbarkeitstests und höchstens $\log_2 x$ Divisionen

- **Unäre** Kodierung von x (x Einsen als Eingabe):
Laufzeit **polynomiell** bezüglich der Länge der Eingabe
- **Binäre** Kodierung von x ($\lceil \log_2 x \rceil$ Bits):
Laufzeit **exponentiell** bezüglich der Länge der Eingabe

Eingabekodierung

Betrachtete Eingabegröße:

- Größe von Zahlen: **Anzahl Bits** bei binärer Kodierung
- Größe von Mengen / Folgen: **Anzahl Elemente**

Beispiel (Sortieren)

Gegeben: Folge von Zahlen $a_1, \dots, a_n \in \mathbb{N}$

Gesucht: sortierte Folge der Zahlen

Größe der Eingabe: n

Manchmal Betrachtung von mehr Parametern:

- Größe von Graphen: Anzahl Knoten und Anzahl Kanten

Effizienzmessung

Sei \mathcal{I}_n die Menge der Instanzen der **Größe** n eines Problems.

Effizienzmaße:

- Worst case:

$$t(n) = \max \{ T(i) : i \in \mathcal{I}_n \}$$

- Average case:

$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i)$$

- Best case:

$$t(n) = \min \{ T(i) : i \in \mathcal{I}_n \}$$

(Wir stellen sicher, dass max und min existieren und dass \mathcal{I}_n endlich ist.)

Vor- und Nachteile der Maße

- worst case:
liefert **Garantie** für die Effizienz des Algorithmus, evt. aber sehr pessimistische Abschätzung
- average case:
beschreibt durchschnittliche Laufzeit, aber nicht unbedingt übereinstimmend mit dem "typischen Fall" in der Praxis, ggf. Verallgemeinerung mit Wahrscheinlichkeitsverteilung
- best case:
Vergleich mit worst case liefert Aussage über die Abweichung innerhalb der Instanzen gleicher Größe, evt. sehr optimistisch

Exakte Formeln für $t(n)$ sind meist sehr aufwendig bzw. nicht möglich!

⇒ betrachte **asymptotisches Wachstum** ($n \rightarrow \infty$)

Wachstumsrate / -ordnung

- $f(n)$ und $g(n)$ haben **gleiche** Wachstumsrate, falls für große n das Verhältnis durch Konstanten beschränkt ist:

$$\exists c, d \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad c \leq \frac{f(n)}{g(n)} \leq d$$

- $f(n)$ 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.,

$$\forall c \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : \quad f(n) \geq c \cdot g(n)$$

anders ausgedrückt: $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$

Beispiel

n^2 und $5n^2 - 7n$ haben gleiche Wachstumsrate, da für alle $n \geq 2$ $1 \leq \frac{5n^2 - 7n}{n^2} \leq 5$ gilt. Beide wachsen schneller als $n^{3/2}$.

Asymptotische Notation

Mengen zur Formalisierung des asymptotischen Verhaltens:

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

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

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

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

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

Funktionen sollen Laufzeit bzw. Speicherplatz beschreiben

⇒ Forderung:

$$\exists n_0 : \forall n \geq n_0 : \underline{f(n)} > 0$$

Manchmal auch:

$$\forall n : \underline{f(n)} \geq 0$$

Wachstumsrate / -ordnung

- 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

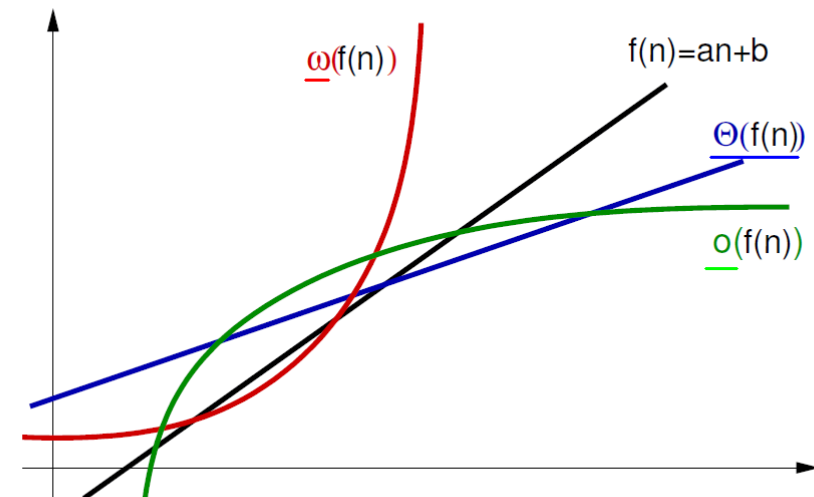
Computern und kann die reale Laufzeit sowieso nur bis auf

- ~~konstante Faktoren bestimmen.~~

Daher ist es meistens sinnvoll, Algorithmen mit gleicher Wachstumsrate erstmal als gleichwertig zu betrachten.

- außerdem: Laufzeitangabe durch ~~einfache Funktionen~~

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 oft auch $g(n) = O(f(n))$
- 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

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

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

Übersicht

- 3 Effizienz
 - Effizienzmaße
 - Rechenregeln für O -Notation
 - Maschinenmodell
 - Laufzeitanalyse
 - Durchschnittliche Laufzeit
 - Erwartete Laufzeit

Wachstumsrate von Polynomen

Lemma

Sei p ein Polynom der Ordnung k bzgl. der Variable n , also

$$p(n) = \sum_{i=0}^k a_i \cdot n^i \quad \text{mit} \quad a_k > 0.$$

Dann ist

$$p(n) \in \Theta(n^k).$$

Wachstumsrate 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:

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

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.



Wachstumsrate von Polynomen

Beweis.

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

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

Für positive n gilt dann:

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

Also ist die Definition

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

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



Rechenregeln für O-Notation

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

Lemma

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

Die Ausdrücke sind auch korrekt für Ω statt O .

Vorsicht, der letzte heißt dann

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

Aber: **Vorsicht bei induktiver Anwendung!**



Induktions"beweis"

Behauptung:

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

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

Ind.anfang: $f(1) = O(1)$

Ind.vor.: Es gelte $f(n-1) = O(n-1)$

Ind.schritt: Dann gilt

$$f(n) = n + f(n-1) = n + O(n-1) = O(n)$$

Also ist

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

FALSCH!



Ableitungen und O -Notation

Lemma

Seien f und g **differenzierbar**.

Dann gilt

- falls $f'(n) \in O(g'(n))$, dann auch $f(n) \in O(g(n))$
- falls $f'(n) \in \Omega(g'(n))$, dann auch $f(n) \in \Omega(g(n))$
- 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 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 $1/n \in O(1)$.
 - ⇒ $\log n \in O(n)$

Übersicht

- ③ Effizienz
 - Effizienzmaße
 - Rechenregeln für O -Notation
 - **Maschinenmodell**
 - Laufzeitanalyse
 - Durchschnittliche Laufzeit
 - Erwartete Laufzeit

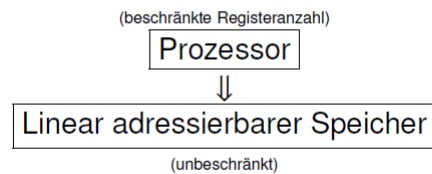
Übersicht

- ③ Effizienz
 - Effizienzmaße
 - Rechenregeln für O -Notation
 - **Maschinenmodell**
 - Laufzeitanalyse
 - Durchschnittliche Laufzeit
 - Erwartete Laufzeit

Abstraktion durch Maschinen-/Rechnermodelle

- 1936 Turing-Maschine: kann nicht auf beliebige Speicherzellen zugreifen, nur an der aktuellen Position des Lese-/Schreibkopfs
- 1945 J. von Neumann u.a.: Entwurf des Rechners EDVAC (Electronic Discrete Variable Automatic Computer)
Programm und Daten teilen sich einen **gemeinsamen** Speicher
- 1963 John Shepherdson, Howard Sturgis (u.a.):

Random Access Machine (RAM)



RAM: Aufbau

Prozessor:

- beschränkte Anzahl an Registern R_1, \dots, R_k
- Instruktionszeiger zum nächsten Befehl

Programm:

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

Eingabe:

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

Modell / Reale Rechner:

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

RAM: Aufbau

Prozessor:

- beschränkte Anzahl an Registern R_1, \dots, R_k
- Instruktionszeiger zum nächsten Befehl

Programm:

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

Eingabe:

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

Modell / Reale Rechner:

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

RAM: Speicher

- unbeschränkt viele Speicherzellen (words) $S[0], S[1], S[2], \dots$, von denen zu jedem Zeitpunkt nur endlich viele benutzt werden
- beliebig große Zellen führen zu unrealistischen Algorithmen
⇒ Jede Speicherzelle darf bei Eingabelänge n eine Zahl mit $O(\log n)$ Bits speichern.
(Für konstant große Zellen würde man einen Faktor $O(\log n)$ bei der Rechenzeit erhalten.)
- ⇒ gespeicherte Werte stellen polynomiell in Eingabelänge n beschränkte Zahlen dar (sinnvoll für Array-Indizes; bildet auch geschichtliche Entwicklung $4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64$ Bit ab)

Begrenzter Parallelismus:

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

RAM: Befehle

Annahme:

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

Befehlssatz:

- Registerzuweisung:
 $R_i := c$ (Konst. an Register), $R_i := R_j$ (Register an Register)
- Speicherzugriff:
 $R_i := S[R_j]$ (lesend), $S[R_j] := R_i$ (schreibend)
- Arithmetische / logische Operationen:
 $R_i := R_j \text{ op } R_k$ (binär: $\text{op} \in \{+, -, \cdot, \oplus, /, \%, \wedge, \vee, <, \leq, =, \geq, >\}$),
 $R_i := \text{op } R_j$ (unär: $\text{op} \in \{-, \neg\}$)
- Sprünge:
 $\text{jump } x$ (zu Adresse x), $\text{jumpz } x R_i$ (bedingt, falls $R_i = 0$),
 $\text{jumpi } R_j$ (zu Adresse aus R_j)

Das entspricht **Assembler-Code** von realen Maschinen!

Maschinenmodell

RAM-Modell

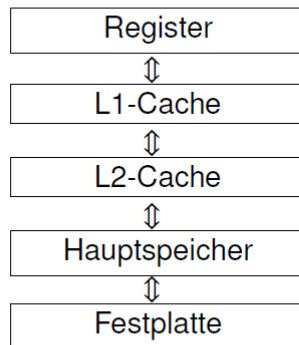
- Modell für die ersten Computer
- entspricht eigentlich der Harvard-Architektur (separater Programmspeicher)
- Random Access Stored Program (RASP) Modell entspricht der von Neumann-Architektur und hat große Ähnlichkeit mit üblichen Rechnern

Aber: Speicherhierarchie erfordert ggf. Anpassung des Modells

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

Speicherhierarchie

schnell, klein

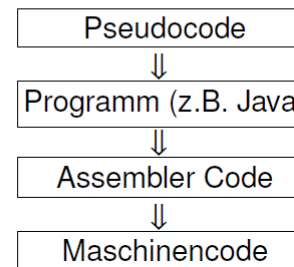


langsam, groß

External-Memory Model

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

Pseudocode / Maschinencode



- Assembler/Maschinencode schwer überschaubar
- besser: Programmiersprache wie Pascal, C++, Java, ...
- oder: informal als Pseudocode in verständlicher Form

$a := a + bc \quad \Rightarrow \quad R_1 := R_b * R_c; \quad R_a := R_a + R_1$

R_a, R_b, R_c : Register, in denen a, b und c gespeichert sind

if (C) | else J \Rightarrow 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, sEnd: Adresse des 1. Befehls in trans(J) bzw. des 1. Befehls nach trans(J)

Übersicht

3 Effizienz

- Effizienzmaße
- Rechenregeln für O -Notation
- Maschinenmodell
- Laufzeitanalyse
- Durchschnittliche Laufzeit
- Erwartete Laufzeit

Laufzeitanalyse / worst case

Berechnung der worst-case-Laufzeit:

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

Beispiel: Vorzeichenausgabe

Funktion `signum(x)`

Eingabe : Zahl $x \in \mathbb{R}$

Ausgabe : $-1, 0$ bzw. 1
entsprechend dem
Vorzeichen von x

Wir wissen:

$$T(x < 0) = O(1)$$

$$T(\text{return } -1) = O(1)$$

$$T(\text{if } (C) I) = O(T(C) + T(I))$$

```

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

```

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

Beispiel: Vorzeichenausgabe

Funktion `signum(x)`

Eingabe : Zahl $x \in \mathbb{R}$

Ausgabe : $-1, 0$ bzw. 1
entsprechend dem
Vorzeichen von x

```

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

```

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

Beispiel: Minimumsuche

Funktion `minimum(A, n)`

Eingabe : Zahlenfolge in $A[0], \dots, A[n-1]$

n : Anzahl der Zahlen

Ausgabe : Minimum der Zahlen

`min = A[0];`

`for (i = 1; i < n; i++) do`

`if A[i] < min then min = A[i];`

`return min`

$$\begin{array}{l} O(1) \\ O(\sum_{i=1}^{n-1} (1 + T(i))) \\ O(1) \\ O(1) \end{array}$$

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

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

Algorithmus 1 : `BubbleSort(A, n)`

Input : n : Anzahl der Zahlen

$A[0], \dots, A[n-1]$: Zahlenfolge

Output : Sortierte Zahlenfolge A

`for (i = 0; i < n - 1; i++) do`

`for (j = n - 2; j >= i; j--) do`

`if A[j] > A[j + 1] then`

`x = A[j];`

`A[j] = A[j + 1];`

`A[j + 1] = x;`

$$\begin{array}{l} O(\sum_{i=0}^{n-2} T(i_1)) \\ O(\sum_{j=i}^{n-2} T(i_2)) \\ O(1 + T(i_3)) \\ O(1) \\ O(1) \\ O(1) \end{array}$$

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

Beispiel: Sortieren

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

Beispiel: Binäre Suche

Prozedur BinarySearch(A, n, x)

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

Output : Index der gesuchten Zahl

```

ℓ = 0;
r = n - 1;
while (ℓ ≤ r) do
    m = ⌊(r + ℓ)/2⌋;
    if A[m] == x then return m;
    if A[m] < x then ℓ = m + 1;
    else r = m - 1;
return -1
    
```

$O(1)$
 $O(1)$
 $O(\sum_{i=1}^k T(l))$
 $O(1) \uparrow$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$

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

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.

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

Gesamtkomplexität: $O(\log n)$

Beispiel: Bresenham-Algorithmus

Algorithmus Bresenham1: zeichnet einen Kreis

```

x = 0; y = R;
plot(0, R); plot(R, 0); plot(0, -R); plot(-R, 0);
F = 5/4 - R;
while x < y do
    if F < 0 then
        F = F + 2 * x + 1;
    else
        F = F + 2 * x - 2 * y + 2;
        y = y - 1;
    x = x + 1;
    plot(x, y); plot(-x, y); plot(-y, x); plot(-y, -x);
    plot(y, x); plot(y, -x); plot(x, -y); plot(-x, -y);
    
```

$O(1)$
 $O(1)$
 $O(1)$
 $O(\sum_{i=1}^k T(l))$

alles $O(1)$

Wie groß ist Anzahl Schleifendurchläufe k ?

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

Beispiel: Bresenham-Algorithmus

- Betrachte dazu die Entwicklung der Werte der Funktion

$$\varphi(x, y) = y - x$$

- Anfangswert: $\varphi_0(x, y) = R$
- Monotonie: verringert sich pro Durchlauf um mindestens 1
- Beschränkung: durch die **while**-Bedingung $x < y$
 bzw. $0 < y - x$

\Rightarrow maximal R Runden

Beispiel: Fakultätsfunktion

Funktion fakultaet(n)

Eingabe : $n \in \mathbb{N}_+$

Ausgabe : $n!$

```

if ( $n == 1$ ) then            $O(1)$ 
  return 1                  $O(1)$ 
else
  return  $n * fakultaet(n - 1)$     $O(1 + \dots?)$ 

```

- $T(n)$: Laufzeit von fakultaet(n)
 - $T(1) = O(1)$
 - $T(n) = T(n - 1) + O(1)$
- $\Rightarrow T(n) = \underline{O(n)}$

Übersicht

- 3 Effizienz
 - Effizienzmaße
 - Rechenregeln für O -Notation
 - Maschinenmodell
 - Laufzeitanalyse
 - **Durchschnittliche Laufzeit**
 - Erwartete Laufzeit