Algorithmen und Datenstrukturen 1

Prof. Dr. Carsten Lecon

Wiederholung

- Algorithmusbegriff
  - präzise, eindeutig, endliche Beschreibung, …
  - Eigenschaften von Algorithmen
    - Determinismus
    - Statische Finitheit
    - Dynamische Finitheit
    - Komplexität
      - Zeiteffizienz
      - Speichereffizienz
    - Korrektheit
    - Terminiertheit
  - Beispiele (Kürzen eines Bruches)

Wiederholung

- Notation von Algorithmen
  - Natürliche Sprache
  - Mathematische Formeln
  - Programmablaufplan
  - Struktogramm
  - Pseudocode

Inhalt I

- Analyse von Algorithmen
  - Einführung
  - Beispiel
  - Analyse
  - Wachstum von Funktionen
- Entwurf von Algorithmen
  - Einführung
  - Teile und herrsche
  - Greedy-Verfahren
  - Backtracking

Inhalt I

- Analyse von Algorithmen
  - Einführung
  - Beispiel
  - Analyse
  - Wachstum von Funktionen
- Entwurf von Algorithmen
  - Einführung
  - Teile und herrsche
  - Greedy-Verfahren
  - Dynamische Programmierung
  - Vollständige Enumeration
  - Backtracking

Lernziele

- Herausforderungen:
  - Wie zeigt man die Korrektheit eines Algorithmus?
  - Wie kann man die Laufzeit abschätzen?
  - Wie kann man den Speicherbedarf abschätzen?
  - Asymptotische Laufzeit

- Untersuchung am Beispiel eines Sortieralgorithmus

Inhalt I

- Analyse von Algorithmen
  - Einführung
  - Beispiel
  - Analyse
  - Wachstum von Funktionen
- Entwurf von Algorithmen
  - Einführung
  - Teile und herrsche
  - Greedy-Verfahren
  - Dynamische Programmierung
  - Vollständige Enumeration
  - Backtracking

Das Sortierproblem

- Das Sortierproblem wird beschrieben durch
  - Eingabe: Folge von n ganzen Zahlen $(a_1, a_2, \ldots, a_n)$
  - Ausgabe: Eine Permutation $(a'_1, a'_2, \ldots, a'_n)$ mit

    $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

  - Beispiel:
    - Eingabe:
      - 31, 41, 59, 26, 58
    - Ausgabe:
      - 26, 31, 41, 58, 59

Anwendungen des Sortierproblems

- Sortierproblem ist fundamentales Problem in der Informatik:
  - Es kommt als Teilproblem in vielen Anwendung vor.
  - Es gibt viele Lösungen (Algorithmen).
  - Die unterschiedlichen Lösungen basieren z.T. auf unterschiedliche Entwurfstechniken.
  - Das Problem an sich ist gut verstanden. Man kennt eine untere Schranke für die Laufzeit.
  - Das Problem ist optimal lösbar, d.h., die besten Algorithmen haben eine Laufzeit in der Größenordnung der unteren Schranke.

Anwendungen des Sortierproblems

- Beispiel: *Insertionsort*
- Einfach, wenig effizient; aber
  - leicht zu implementieren
  - gute Ergebnisse bei vorsortierten oder kleinen Folgen

- Entspricht bspw. dem Kartenaufnehmen in Kartenspielen

Bildquelle: http://mathbits.com

Algorithmus (in Pseudocode)

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x = A[i] // einzufügendes Element
4      j = i    // einzufügende Position
5      while( (j>1) and (A[j-1] > x) )
6        A[j] = A[j-1]
7        j = j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

Indizierung bei Feldern (Arrays)

- Obacht!
  - In Programmiersprachen in der Regel **0 … n-1**
  - Bei der Beschreibung von Algorithmen in der Regel **1 … n**

Algorithmus (als Struktogramm)

- Wie sieht es aus?
  → Hörsaalübung ☺

Inhalt I

- Analyse von Algorithmen
  - Einführung
  - Beispiel
  - Analyse
  - Wachstum von Funktionen
- Entwurf von Algorithmen
  - Einführung
  - Teile und herrsche
  - Greedy-Verfahren
  - Dynamische Programmierung
  - Vollständige Enumeration
  - Backtracking

Analyse von Algorithmen

- Wichtige Kriterien:
  - Korrektheit:
    - Es ist zu zeigen, dass der Algorithmus für alle Instanzen des Problems eine korrekte Lösung berechnet.
  - Komplexität:
    - Der Ressourcenbedarf des Algorithmus an Laufzeit und Speicher wird untersucht. In der Praxis relevant ist vor allem die Laufzeit.
  - Wichtig ist ein aussagekräftiges Maß für diese Größen.

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x = A[i] // einzufügendes Element
4      j = i    // einzufügende Position
5      while( (j>1) and (A[j-1] > x) )
6        A[j] = A[j-1]
7        j = j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

Korrektheit von Algorithmen

- In Analogie zur Beweistechnik der vollständigen Induktion wird mit Invarianten (Schleifeninvarianten) gearbeitet, indem man folgende Punkte zeigt:
  - **Initialisierung**:
    - Die Invariante ist vor der ersten Iteration der Schleife wahr.
  - **Aufrechterhaltung**:
    - Wenn die Invariante vor einer Iteration der Schleife erfüllt ist, dann ist sie auch vor Beginn der nächsten Iteration erfüllt.
  - **Terminierung**:
    - Wenn die Schleife endet, dann liefert die Invariante einen nützlichen Hinweis, um die Korrektheit des Algorithmus zu zeigen.

Korrektheit des insertionsort-Algorithmus

- Beobachtung: Berechnung erfolgt in Schleife (2..n)
- **Schleifeninvariante**:
  - Die Teilfolge $a_1, \ldots, a_{i-1}$ ist sortiert.

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x = A[i] // einzufügendes Element
4      j = i    // einzufügende Position
5      while( (j>1) and (A[j-1] > x) )
6        A[j] = A[j-1]
7        j = j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

Korrektheit des insertionsort-Algorithmus

- Initialisierung:
  - Vor dem ersten Schleifendurchlaufs (Zeile 2) gilt:
    - i wird auf 2 gesetzt
    - Teilfolge ist $a_1, \ldots, a_{2-1}$ (also nur a_1)

      → das Feld ist sortiert, die Invariante ist erfüllt.

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x = A[i] // einzufügendes Element
4      j = i    // einzufügende Position
5      while( (j>1) and (A[j-1] > x) )
6        A[j] = A[j-1]
7        j = j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

Korrektheit des insertionsort-Algorithmus

- Aufrechterhaltung:
  - Die **for**-Schleife bewegt die Elemente a_{j-1}, a_{j-2}, …jeweils um eine Position nach rechts (Zeilen 4-8), bis der richtige Einfügeplatz für a_i gefunden ist.
  - Dann wird a_i an diese Stelle geschrieben (Zeile 9).

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x = A[i] // einzufügendes Element
4      j = i    // einzufügende Position
5      while( (j>1) and (A[j-1] > x) )
6        A[j] = A[j-1]
7        j = j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

Korrektheit des insertionsort-Algorithmus

- Aufrechterhaltung (2):
  - Der Schlüssel von a_i wird zwischengespeichert.
  - In der vorigen Iteration wurde eine sortierte Teilfolge $(a_1, \ldots, a_{i-1})$ hergestellt.
  - In der **while**-Schleife wird das erste Element $a_j \leq a_i < a_{j+1}$ gesucht.
  - a_i wird durch Verschieben der größeren Elemente zwischen a_j und a_{j+1} einsortiert.

    → somit sind die Elemente a_j, a_i sortiert.

    → die Schleifeninvariante ist zum Start der nächsten Iteration wieder erfüllt.

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x = A[i] // einzufügendes Element
4      j = i    // einzufügende Position
5      while( (j>1) and (A[j-1] > x) )
6        A[j] = A[j-1]
7        j = j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

Korrektheit des insertionsort-Algorithmus

- Terminierung:
  - Algorithmus terminiert, wenn i=n+1 (`länge(A)`).
  - Da die Invariante „$a_1, \ldots, a_{i-1}$ ist geordnet“ gilt, gilt auch „$a_1, \ldots, a_n$ ist geordnet“ (da i=n+1).

→ Der Algorithmus ist korrekt, die Liste ist geordnet.

```
1 function insertionsort(A)
2   for i=2 to länge(A)
3     x = A[i] // einzufügendes Element
4     j = i    // einzufügende Position
5     while( (j>1) and (A[j-1] > x) )
6       A[j] = A[j-1]
7       j = j-1
8     end while
9     A[j] = x
10  end for
11end function
```

Noch ein Beispiel (Klausur WS 2012/2013)

Zeigen Sie mit vollständiger Induktion, dass die Schleifeninvariante $r = 2^i$ lautet.

```
1. public static int pot2 (int n) {
2.   int r = 1;
3.   int i = 0;
4.   while (i < n) {
5.     r *= 2;
6.     i++;
7.   }
8.   return r;
9. }
```

Analyse von Algorithmen

- Wichtige Kriterien:
  - Korrektheit:
    - Es ist zu zeigen, dass der Algorithmus für alle Instanzen des Problems eine korrekte Lösung berechnet.
  - Komplexität:
    - Der Ressourcenbedarf des Algorithmus an Laufzeit und Speicher wird untersucht. In der Praxis relevant ist vor allem die Laufzeit.
  - Wichtig ist ein aussagekräftiges Maß für diese Größen.

Analyse von Algorithmen: Komplexität

- Eine Analyse erlaubt, den „besten“ Algorithmus auszusuchen, bzw. die „ungünstigen“ auszusortieren.
  - Absolute Qualität: Aufwandsmessung und/oder -abschätzung unabhängig von anderen Verfahren
  - Relative Qualität: relativ zur Qualität von „Konkurrenzverfahren“
- Analyse: Vorhersage des Ressourcenverbrauchs eines Algorithmus
  - Rechenzeit
  - Speicher
  - Kommunikationsbreite
  - …
- Meist ist Rechenzeit wichtig

Analyse von Algorithmen

- Rechenzeit hängt ab von
  - Probleminstanz, d.h. Eingabe
  - Eingabelänge (ist problemabhängig)
  - Bei vielen Algorithmen: Anzahl der Elemente der Eingabe (Sortieren, Fouriertransformation, …)
  - Bei machen Algorithmen Anzahl der Bits der Eingabe (binäre Addition, Multiplikation)

Analyse von Algorithmen

- Rechenzeit kann beschrieben werden durch
  - Anzahl der einfachen Operationen, das heißt durch Zählen von Schritten, und
  - Zuweisen von Rechenzeit für bestimmte Operationen

Analyse von Algorithmen: Beispiel

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x := A[i]
4      j := i
5      while( (j>1) and (A[j-1] > x) )
6        A[j] := A[j-1]
7        j := j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

Analyse von Algorithmen

- Was sind hier die relevanten Operationen?
  - Herausnehmen/Einfügen eines Objekts: Pro Iteration nur einmal
  - Größenvergleich mit Objekten der linken (sortierten) Teilfolge: variabel
  - Verschieben von Objekten nach rechts: variabel

Analyse von Algorithmen

- Arten der Analyse:
  - ***best case***
  - ***worst case***
  - ***average case***

Analyse von Algorithmen

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x := A[i]
4      j := i
5      while( (j>1) and (A[j-1] > x) )
6        A[j] := A[j-1]
7        j := j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

- Laufzeit: *best case:*
  - Bereits sortierte Folge
    - Aufwand: Pro Iteration ein Vergleich
  - 1. Objekt wird nicht verglichen (j>1)

    → n-1 Iterationsrunden

    → Gesamtaufwand in der **Größenordnung** von n-1, da der absolute Aufwand für einen Einzelvergleich nicht spezifiziert ist.

Analyse von Algorithmen

- Aufwand (allgemein):
  - Abstraktion auf Potenzen oder andere Funktionen der Gesamtzahl **n** der zu sortierenden Elemente (**n**, **n^2**, **log n**, **n^3 * log n**, etc.)
  - Vernachlässigung von konstanten Anteilen (z.B. n**-1**)
  - „O-Notation“:
    - $O(n^2)$: „In der Ordnung von n^2“

- In unserem Fall:
  - O(n)
  - **Die Komplexität von insertionsort im *best case* ist O(n).**

Analyse von Algorithmen

```
1  function insertionsort(A)
2    for i=2 to länge(A)
3      x := A[i]
4      j := i
5      while( (j>1) and (A[j-1] > x) )
6        A[j] := A[j-1]
7        j := j-1
8      end while
9      A[j] = x
10   end for
11 end function
```

- Laufzeit: *worst case:*
  - Pro Iteration Vergleich und Verschieben von allen linken Nachbarn
  - Hochrechnung:
    - 1. Iteration (i=2): ein Vergleich, einmal Verschieben
    - 2. Iteration (i=3): 2 Vergleiche, zwei Verschiebungen
    - …
    - (n-1). Iteration (i=n): je n-1 Vergleiche und Verschiebungen
    - → 2*(1+2+…n-1)

      = 2*((n-1)*n)/2) = (n-1)*n = n^2-n

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Analyse von Algorithmen

- Laufzeit: *worst case:*
  - Pro Iteration Vergleich und Verschieben von allen linken Nachbarn
  - Hochrechnung:
    - 1. Iteration (i=2): ein Vergleich, einmal Verschieben
    - 2. Iteration (i=3): 2 Vergleiche, zwei Verschiebungen
    - …
    - (n-1). Iteration (i=n): je n-1 Vergleiche und Verschiebungen
    - → $2*(1+2+\ldots n-1)$
      $= 2*((n-1)*n)/2) = (n-1)*n = n^2-n$

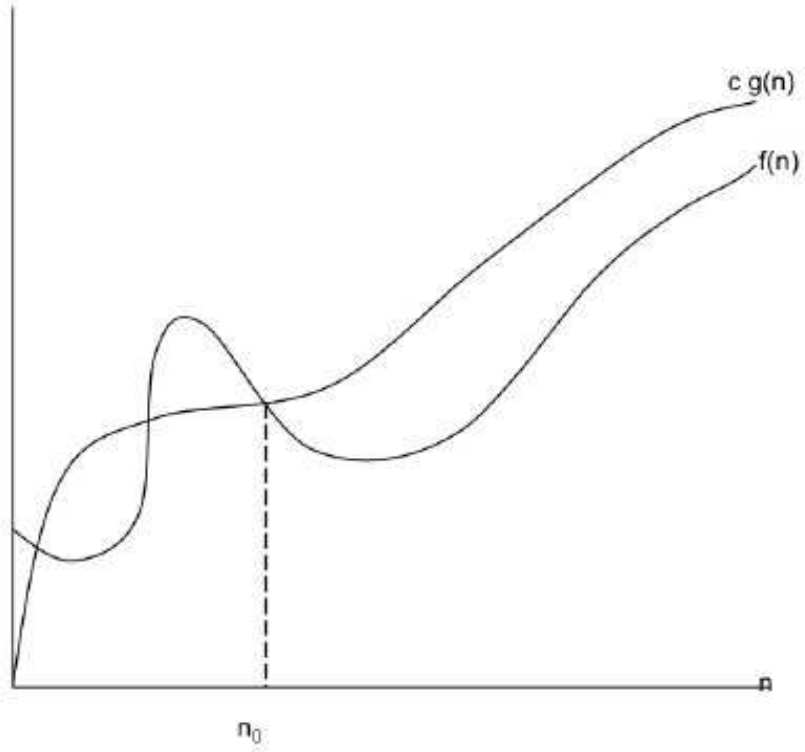
Die Komplexität von insertionsort im *worst case* ist $O(n^2)$.

Analyse von Algorithmen

- O-Notation, formal:
  - Für eine gegebene Funktion g(n) ist O(g(n)) die folgende Menge von Funktionen:
    - $O(g(n)) = \{f(n): \exists$ positive Konstanten c_1, n_0 mit $0 \leq f(n) \leq c_1 g(n)$ für alle $n \geq n_0\}$
  - Man schreibt auch: **f ε O(g)** oder **f=O(g)**
  - **g** ist eine asymptotische obere Schranke für **f(n)**

Analyse von Algorithmen

- Veranschaulichung O-Notation



Analyse von Algorithmen

- O-Notation: Beispiel 1
- Zeigen Sie, dass $\frac{1}{2} n^2 - 3n \in O(n^2)$

- $\frac{1}{2} n^2 - 3n \leq c_1 n^2$
- $\frac{1}{2} - 3/n \leq c_1$

- Wähle $n_0 = 1$ und $c_1 = \frac{1}{2}$

Analyse von Algorithmen

- O-Notation: Beispiel 2
- Zeigen Sie, dass $\mathbf{2n^2 + 3n \in O(n^2)}$

- $\mathbf{2n^2 + 3n \leq 2n^2 + 3n^2 = 5n^2}$

- Wähle $\mathbf{n_0=1}$ und $\mathbf{c_1= 5}$

Analyse von Algorithmen

- O-Notation: Größenordnungen

- $O(1)$ konstant
- $O(\log n)$ logarithmisch
- $O(\log_2 n)$ quadratisch logarithmisch
- $O(n)$ linear
- $O(n \log n)$ n log n
- $O(n^2)$ quadratisch
- $O(n^3)$ kubisch
- $O(n^k)$ polynomiell
- $O(k^n)$ exponentiell

Rechenregeln

$f \in \Theta(f)$

$f \in \Omega(f)$

$f \in O(f)$

$O(O(f)) \in O(f)$

$c \cdot O(f) \in O(f)$

$O(f + c) \in O(f)$

$O(f + g) \in O(\max(f, g))$

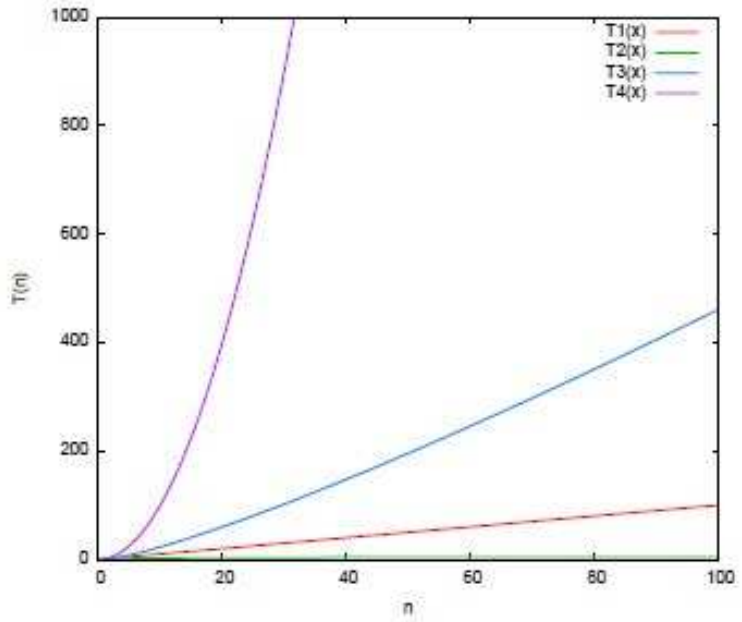
$O(f) \cdot O(g) \in O(f \cdot g)$

$O(f + g)$ Hintereinanderausführung von Programmteilen (Sequenz)

$O(f) \cdot O(g)$ Schachtelung von Programmteilen

Analyse von Algorithmen

- O-Notation: Größenordnungen



T1 linear, T2 logarithmisch, T3 $n \log n$, T4 quadratisch

Analyse von Algorithmen

Annahme: Ein Rechenschritt benötigt 1ms → 1000 Schritte/s

T(n)	1s	1 min	1 h
O(n)	1.000	60.000	3.600.00
O(n log n)	140	4.895	204.094
$O(n^2)$	31	244	1.897
$O(n^3)$	10	39	153
$O(2^n)$	9	15	21

Wachstum der Laufzeit von Algorithmen

- Beispiel 1: Laufzeit?

```
for (int j = 0; j < N; j++) {
    if (j == N / 2)
        break;
    for (int i = 0; i < N; i++) {
        if (i <= N / 2) {
            System.out.println(i);
        } else {
            break;
        } // if
    } // for
} // for (j)
for (int j = 0; j < 3; j++) {
    for (int i = 0; i < N; i++) {
        System.out.println(i * i);
    } // for (i)
} // for (j)
```

- Vermutung: ½ N^2 + 3N

Wachstum der Laufzeit von Algorithmen

- Vermutung: ½ N^2 + 3N
- Zeigen Sie, dass **½ N^2 + 3N ε $O(N^2)$**

$$\tfrac{1}{2} N^2 + 3N \leq c_1 N^2$$
$$\tfrac{1}{2} + 3/N \leq c_1$$

Wähle $n_0=1$ und $c_1=3,5$

Wachstum der Laufzeit von Algorithmen

- Beispiel 2: Laufzeit?

```
for (int j=0; j<N; j++) {
    for (int i=0; i<N; i++) {
        System.out.print(j-i);
    } // for (i)
} // for (j)
for (int j=0; j<3; j++) {
    for (int i=0; i<N; i++) {
        System.out.println(i);
    } // for (i)
} // for(j)
```

- Vermutung: $2N^2 + 3N = O(N^2)$

Wachstum der Laufzeit von Algorithmen

- Vermutung: $\mathbf{2N^2 + 3N = O(N^2)}$
- Zeigen Sie, dass $\mathbf{2N^2 + 3N = O(N^2)}$

  $\mathbf{2N^2 + 3N \leq 2N^2 + 3N^2 = 5N^2}$

  Wähle $\mathbf{n_0=1}$ und $\mathbf{c_1= 5}$

Analyse von Algorithmen - Zusammenfassung

- Beweis der Korrektheit:
  - Z.B. mittels Invarianten
- Komplexität
  - Laufzeit wird größenordnungsmäßig beschrieben
    - Meist O-Notation