Technische Universität München
Fakultät für Informatik, I-14
Prof. Harald Räcke
Robin Münk

SoSe 2025
Übungsblatt 4
16. Mai 2025

Algorithmen und Datenstrukturen

Aufgabe 1 Rekursion

Gegeben sind die folgenden zwei Funktionen in Pseudocode.

```
func1(n):
    i = 1
    while (i <= n)
        print(i)
        i += 1
```

```
func2(n):
    i = n
    while (i > 0)
        print(i)
        i -= 1
```

Schreiben Sie beide Funktionen so um, dass keine Schleife (Wiederholung) mehr verwendet wird!

Beispiellösung:

```
func1(n):
    if (n > 0)
        func1(n-1)
        print(n)
```

```
func2(n):
    if (n > 0)
        print(n)
        func2(n-1)
```

Aufgabe 2 Komplexitätsschätzung

Schätzen Sie die Komplexitäten der folgenden Algorithmen asymptotisch ab!

a) **rot13**(*string*):
   encrypted = *string*;
   for $i = 0$ **to size**(*string*)-1 {
      if (*string*$[i] \geq$ 'a' $\wedge$ *string*$[i] \leq$ 'z') {
         abstand = *string*$[i]$ $-$ 'a';
         rotierter_abstand = $(abstand + 13) \bmod 26$;
         encrypted$[i]$ = 'a' + *rotierter_abstand*;
      }
   }

   return *encrypted*;

b) **MatrixMultiplikation**(A, B):
   if (**cols**(A)$\neq$**rows**(B)) {
      return $-\infty$;
   }
   n = **cols**(A);
   C = **matrix**(**rows**(A), **cols**(B));

```
for i = 0 to rows(A)−1 {
    for j = 0 to cols(B)−1 {
        C[i][j] = 0;
        for k = 0 to n − 1 {
            C[i][j] = C[i][j] + A[i][k] · B[k][j];
        }
    }
}
return C;
```

Beispiellösung:

a) $O(n)$ (linear), dabei ist n die Länge des Strings.
Wir haben eine For-Schleife, die von $i = 0$ bis $n-1$ läuft. Das Innere der Schleife wird also n mal ausgeführt. Die Anzahl der Anweisungen, die während eines Schleifendurchgangs ausgeführt werden, ist aber konstant (eine if-Anweisung, einige Zuweisungen, einige Additionen und Modulorechnung). Dann ist die Laufzeit in $O(n)$.

b) $O(n \cdot m \cdot l)$ (kubisch in Anzahl der Spalten, wenn $n = m = l$), dabei ist A eine $n \times m$ und B eine $m \times l$ Matrix.
Wir haben drei verschachtelte Schleifen, die jeweils n, m, bzw. l mal ausgeführt werden. Die Berechnung von $C[i][j]$ wird in konstanter Zeit durchgeführt, da nur eine Addition und eine Multiplikation benötigt werden.

Aufgabe 3 Komplexität der Polynom-Auswertung

In allgemeiner Form ist ein *Polynom* gegeben durch

$$p(x) := \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0 \ ,$$

wobei $a_i \in \mathbb{R}$ der Koeffizient für die i-te Potenz x^i ist. Die höchste Potenz n ist dabei der *Grad des Polynoms*. Einen einzigen Summanden $a_i x^i$ bezeichnet man als *Monom*.

Wir untersuchen nun drei verschiedene Methoden, ein Polynom auszuwerten. Gegeben sei dazu ein Array a der Länge $n+1$, wobei $a[i]$ den Koeffizienten der Potenz x^i enthält. Die allgemeine Signatur der Funktion in Pseudo-Code ist also wie folgt:

Input: Koeffizienten-Array a der Länge $n+1$, Auswertungspunkt x
Output: Ergebnis $p = a[n] \cdot x^n + ... + a[1] \cdot x + a[0]$
EvaluatePolynomial(a,x)**:**

```
p = ...
return p;
```

a) In einer ersten Implementation gehen wir vor, wie dies auf dem Papier erfolgen würde. Wir beginnen also bei der höchsten Potenz, berechnen jeweils die Monome und addieren diese dann auf:

```
p = 0;
for i = n down to 0 {
    // Berechne i-te Potenz
    m = 1;
```

```
        for j = 1 to i {
            m = m · x;
        }

        // Berechne i-tes Monom durch Multiplikation der Potenz mit dem Koeffizienten
        m = a[i] · m;

        // Aktualisiere das Polynom durch Addition des Monoms
        p = p + m;
    }
```

Bestimmen Sie die Laufzeitfunktion dieser Implementierung! In welcher Effizienzklasse in O-Notation ist dieser Code?

b) Offensichtlich ist in diesem Code die Berechnung der Potenzen redundant. Wir stellen den Code nun so um, dass die Potenzen inkrementell auf Basis vorheriger Iterationen errechnet werden:

```
p = 0;
h = 1;
for i = 0 to n {
    // Berechne i-tes Monom durch Multiplikation der (vorberechneten) Potenz mit
    // dem Koeffizienten
    m = a[i] · h;

    // Aktualisiere die Potenz für den nächsten Schritt i + 1
    h = h · x;

    // Aktualisiere das Polynom durch Addition des Monoms
    p = p + m;
}
```

Bestimmen Sie auch die Laufzeitfunktion dieser neuen Implementierung, sowohl exakt als auch in O-Notation!

c) Zuletzt untersuchen wir das sogenannte *Horner-Schema*. Die Idee ist, das Polynom umzuformen:

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0 = (((...) \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$$

Dies kommt dadurch zustande, dass man immer wieder jeweils x aus den höheren Potenzen ausklammert. In Pseudo-Code lautet eine entsprechende Implementierung dann wie folgt:

```
p = a[n];
for i = n − 1 down to 0 {
    // Multipliziere ausgeklammertes x an den bereits ausgewerteten Teil
    p = p · x;

    // Addiere den nächsten Koeffizienten
    p = p + a[i]
}
```

Bestimmen Sie zuletzt auch die Laufzeitfunktion dieser Implementierung, sowohl genau wie auch in O-Notation!

Beispiellösung:

In dieser Aufgabe zählen wir nicht alle Rechenschritte, sondern begnügen uns damit „teure“ arithmetische Operationen zu zählen. Im vorliegenden Fall sind dies vor allem die Multiplikationen.

Vergegenwärtigen Sie sich, dass wir bei einem Polynom des Grads n von $n+1$ Koeffizienten ausgehen (auch wenn diese dann den Wert 0 haben können).

a) Betrachten wir nochmals den Code der einfachen Implementierung:

```
1       p = 0;
2       for i = n down to 0 {
3           m = 1;
4           for j = 1 to i {
5               m = m · x;
            }

6           m = a[i] · m;
7           p = p + m;
        }
```

Im Folgenden vernachlässigen wir die beiden Initialisierungen in den Zeilen 1 und 3. Für jeden Durchlauf i der Schleife (ab Zeile 2) haben wir also folgenden Aufwand:

- i Multiplikationen für die i-te Potenz in Zeilen 4 und 5
  Beachten Sie dabei, dass die letzte Iteration für $i = 0$ diesen Schritt überspringt!
- 1 Multiplikation für den Koeffizienten in Zeile 6
- 1 Addition für die Summierung in Zeile 7

Damit kommen wir bei $n+1$ Iterationen also auf folgenden Gesamt-Aufwand:

- Multiplikationen:
$$\left(\sum_{i=1}^{n} i\right) + (n+1) = \left(\frac{n(n+1)}{2}\right) + (n+1) = \frac{1}{2}n^2 + \frac{3}{2}n + 1$$
- Additionen:
$$n+1$$
- Die Laufzeitfunktion ist damit mindestens so groß wie die Summe dieser beiden Zahlen, da wir ja zusätzlichen Aufwand wie hauptsächlich die Schleifenverwaltung (vom Aufwand $O(n)$) ignorieren. Also folgern wir:
$$T(n) = O(n^2)$$

b) Dies bringt uns zur nächsten Implementierung, bei der die Berechnung der Potenz inkrementell über die Hilfsvariable h erfolgt:

```
1       p = 0;
2       h = 1;
3       for i = 0 to n {
4           m = a[i] · h;
5           h = h · x;
6           p = p + m;
        }
```

Auch hier vernachlässigen wir wieder die Initialisierungen in Zeilen 1 und 2. Für jeden Durchlauf der Schleife (ab Zeile 3) ergibt sich also:

- 1 Multiplikation für den Koeffizienten in Zeile 4
- 1 Multiplikation für die Potenzierung in Zeile 5
- 1 Addition für die Summierung in Zeile 6

Somit lässt sich also der Gesamt-Aufwand bei $n+1$ Iterationen errechnen:

- Multiplikationen:
$$2(n+1) = 2n+2$$
Nachdem die Zeile 5 *ganz am Schluss einmal zu oft ausgeführt wird, ließe sich der Aufwand um eine Multiplikation reduzieren. Außerdem liessen sich zwei weitere Multiplikationen vermeiden, da ja im ersten Durchlauf (mit* $h = 1$*) zwei überflüssige Multiplikationen mit* 1 *stattfinden. Die minimale Anzahl an Multiplikationen ist daher* $2n-1$.
- Additionen:
$$n+1$$
- Die Laufzeitfunktion ist hier wieder mindestens so groß wie die Summe dieser beiden Zahlen, und wieder käme zusätzlicher Aufwand für die Schleifenverwaltung hinzu. Also:
$$T(n) = O(n)$$

c) Betrachten wir zuletzt das Horner-Schema:

```
1    p = a[n];
2    for i = n − 1 down to 0 {
3        p = p · x;
4        p = p + a[i]
     }
```

Dieser Code ist vollständig optimiert und kann durch die Zuweisung in Zeile 1 eine komplette Schleifeniteration einsparen. In einem einzigen Durchlauf haben wir dabei folgenden Aufwand:

- 1 Multiplikation mit einem ausgeklammerten Faktor der Potenz in Zeile 3
- 1 Addition für die Summierung in Zeile 4

Bei nur n Iterationen ergibt sich also dieser Gesamtaufwand:

- Multiplikationen:
$$n$$
- Additionen:
$$n$$
- Wie zuvor ergibt sich die Laufzeitfunktion als Summe der beiden Zahlen und muss durch Schleifenverwaltungsaufwand der Größe $O(n)$ ergänzt werden. Zusammen also:
$$T(n) = O(n)$$

Zum Schluss möchten wir die drei Implementationen vergleichen. Offensichtlich ist der naïve Ansatz wegen seiner $O(n^2)$ Komplexität der eindeutige Verlierer gegen die beiden anderen Methoden. Interessanter ist dagegen der Vergleich zwischen inkrementeller Berechnung der Potenz und dem Horner-Schema, da beide ja in der gleichen Komplexitätsklasse $O(n)$ sind.

Wie Sie sich erinnern, unterscheiden sich die Elemente einer Komplexitätsklasse durchaus um konstante Faktoren, lediglich das generelle Wachstumsverhalten ist ab einer bestimmten Problemgröße vergleichbar.

Wenn man sich also auf die Zählung von Multiplikationen beschränkt – die auf klassischen Systemen den größten Aufwand erzeugen – so wird das Horner-Schema mit seinen n Multiplikationen gegenüber der inkrementellen Berechnung mit ihren minimal $2n - 1$ Multiplikationen stets etwa doppelt so schnell sein. Damit ist im Hinblick auf Geschwindigkeitsoptimalität das Horner-Schema der eindeutige Gewinner.

Aufgabe 4 (P) Iteratoren

Betrachten Sie erneut die folgenden Aufgaben der vorherigen Übungsblätter. Schreiben Sie jeweils eine Funktion um das Problem zu lösen, die zum Auslesen/Bearbeiten der Elemente des Containers Iteratoren verwendet.

Hinweis: Vielleicht finden Sie den Reverse Iterator `.rbegin()` des Containers und die Methode `.insert(`*iterator*`, `*value*`)` dabei hilfreich.

a) **PlusOne.** Gegeben ist ein `vector<int> digits`, der einen positiven Integer repräsentiert. Ziffer *i* ist dabei an Stelle `digits[i]`, geordnet von der höchst- zur niedrigstwertigen Ziffer, die höchstwertige Ziffer steht also am Anfang der Liste. Es gibt keine führenden Nullen.

   Inkrementieren Sie die repräsentierte Zahl und geben Sie den zugehörigen `vector<int>` zurück.

   Bsp.: [3, 1, 2, 6] → [3, 1, 2, 7]

b) **Base-*b*-Addition**. Gegeben sind eine Basis *b* und zwei `vector<int> x, y`, die zwei positive base-*b* Integer repräsentieren. Ziffer *i* ist dabei an Stelle `i` mit Wertigkeit b^i, geordnet von der höchst- zur niedrigstwertigen Ziffer, die höchstwertige Ziffer steht also am Anfang der Liste. Es gibt keine führenden Nullen.

   Addieren Sie die repräsentierten Zahlen und geben Sie die Summe als Zahl zur Basis *b* in einem `vector<int>` zurück.

   Bsp.: [42, 53, 28], [37], 55 → [42, 54, 10]