Rechtlicher Hinweis

Legal notice

Informatik I (B.Sc. Physik)

Rekursion

Dr. Paul Bodesheim
(Paul.Bodesheim@uni-jena.de)

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Fakultät für Mathematik und Informatik
Lehrstuhl für Digitale Bildverarbeitung

SoSe 2020

Was ist Rekursion? 递归

- Interessantes Kapitel in Bezug auf Funktionen allgemein
- Unabhängig von C++
- Allgemeines Prinzip in der Informatik und Mathematik

- Zurückführen von Problemen auf “sich selbst”
- Genauer: auf ein einfacheres Problem der selben Art

- Beispiel: Fakultätsfunktion
- Rückführen der Fakultät einer natürlichen Zahl auf die Fakultät des Vorgängers
- Mathematische Formulierung:

$$n! = \begin{cases} 1 & n \leq 1 \\ (n-1)! \cdot n & \text{sonst} \end{cases}$$

- Kann man so die Fakultät berechnen?

Voraussetzungen für Rekursion

- Man kann nicht beliebig (unendlich) oft das Problem auf sich selbst zurückführen
- Irgendwann muss eine Lösung bestimmt werden
- Im Beispiel für die Fakultät: für den Wert 1 ist das Ergebnis (trivialerweise) bekannt

- Wann kann man Rekursion verwenden?
  - Es existiert eine bekannte (triviale) Lösung für einfache Fälle (bestimmte, oft auch kleine Eingaben)
  - "Komplexe" Probleme lassen sich auf einfachere Probleme derselben Art zurückführen (z.B. Berechnung für Vorgänger)

- Rekursion lässt sich auch in C++ umsetzen!

Rekursion in C++

- Beispiel: Fakultätsfunktion

```
int fakultaet(int n)
{
    if (n<=1)
        return 1;
    else
        return fakultaet(n-1)*n;
        // return n*fakultaet(n-1);
}
```

- Eine Funktion kann sich selbst aufrufen!
- Rekursiver Funktionsaufruf muss veränderte Parameterwerte beinhalten, oft kleiner werdende Werte "in Richtung" Triviallösung
- Triviallösung ist Abbruchbedingung, wird üblicherweise mit Bedingung (if) abgefragt
- Ist die Abbruchbedingung erreicht, wird die eigentliche Berechnung ausgeführt
- Parallelen zum Vorgehen beim Beweis durch Induktion, nur in umgekehrter Reihenfolge

Aufrufgraph

fakultaet(3) $\Rightarrow$ 3*fakultaet(2) $\Rightarrow$ 3*2*fakultaet(1) $\Rightarrow$ 3*2*1=6

```
fakultaet(3)                                    ⇓    Rekursiver Abstieg
        3*fakultaet(2)                          ⇓    (hin zur
                2*fakultaet(1)                  ⇓    Abbruchbedingung)
                        fakultaet(1)=1
                2*1=2
        3*2=6
6
```

Rekursive vs. iterative Lösung

```
int fakultaet(int n)
{
    if (n<=1)
        return 1;
    else
        return fakultaet(n-1)*n;
}
```

```
int fakultaet2(int n)
{
    int result = 1;
    for (int i=2; i<=n; i++)
        result *=i;
    return result;
}
```

- Jede rekursive Funktion lässt sich in eine iterative Funktion (Verwendung von Schleifen) überführen und umgekehrt
- Rekursive Funktion wird oft als eleganter bezeichnet

Aufrufgraph, rekursiver Abstieg und Aufstieg

fakultaet(4)=4*fakultaet(3)
fakultaet(3)=3*fakultaet(2)
fakultaet(2)=2*fakultaet(1)
fakultaet(1)=1
fakultaet(2)=2*1=2
fakultaet(3)=3*2=6
fakultaet(4)=4*6=24

- Nach dem rekursiven Abstieg folgt der Aufstieg
- Letzte Berechnung ist die Multiplikation mit 4

- Problem: bei jedem rekursiven Schritt wird zusätzlicher Speicher angefordert (für Zwischenergebnisse und Rücksprungadresse)
  $\Rightarrow$ Speicherplatzbedarf steigt mit der Rekursionstiefe

- Beim Erreichen der Abbruchbedingung sind alle Zahlen (Faktoren) bereits bekannt
- Kann man auf den Aufstieg verzichten bzw. effizienter gestalten?

Endrekursion

- Funktionsaufruf ist der letzte Schritt bei der Berechnung
- Zwischenergebnisse müssen beim rekursiven Selbstaufruf übergeben werden

- Speicherplatzbedarf unabhängig von der Rekursionstiefe, da nur Ergebnis nach oben gereicht wird
- (Ggf. durch Optimierung des Compilers und Umwandlung in iterative Funktion)

```
int fakultaet(int n, int m)
{
    if (n<=1)
        return m;
    else
        return fakultaet(n-1,m*n);
}

int main()
{
    int n = 4;
    int m = fakultaet(n,1);
}
```

Endrekursion: Verwendung von default-Parametern

- Oft werden die Zwischenergebnisse im ersten Parameter gespeichert
- Speichern der Zwischenergebnisse im letzten Parameter erlaubt Verwendung von default-Parametern

```
int fakultaet(int n, int m=1)
{
    if (n<=1)
        return m;
    else
        return fakultaet(n-1,m*n);
}

int main()
{
    int n;
    cin >> n;
    int m = fakultaet(n);
}
```

Alternative ohne default-Parameter: Verwendung einer zusätzlichen (Hilfs-) Funktion

```
int fakultaet(int n)
{
    return berechne_fakultaet(n,1);
}
int berechne_fakultaet(int n, int m)
{
    if (n<=1)
        return m;
    else
        return berechne_fakultaet(n-1,m*n);
}

int main()
{
    int n;
    cin >> n;
    int m = fakultaet(n);
}
```

Aufrufgraph und Berechnung bei Endrekursion

fakultaet(4,1)=fakultaet(3,4)
fakultaet(3,4)=fakultaet(2,12)
fakultaet(2,12)=fakultaet(1,24)
fakultaet(1,24)=24

Berechnung bei Rekursion: 4*(3*(2*1))

Berechnung bei Endrekursion: ((1*4)*3)*2

Zusammenfassung: Rekursion

- Triviallösung als Abbruchbedingung (Anfangswerte)
- Rekursiver Selbstaufruf, um Problem "auf sich selbst" zurückzuführen

  $\Rightarrow$ Vereinfachen des Problems (Rekursionsschema)

  (Beispiel Fakultät: Produkt aus n-1 Zahlen ist "einfacher" als Produkt aus n Zahlen)
- Sonderform: Endrekursion (speichereffizient)

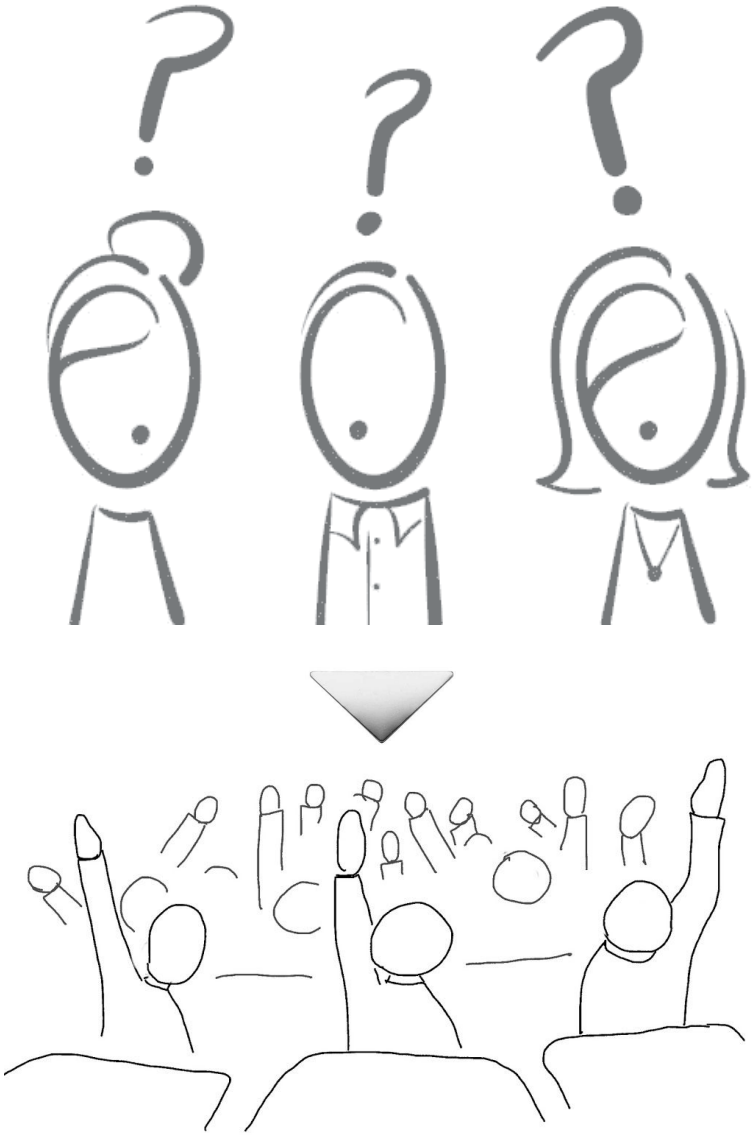
Herangehensweise:

1. Zuerst Triviallösung/Abbruchbedingung für das Problem suchen (und umsetzen)
2. Danach Rekursionsschritt (Selbstaufruf, Vereinfachung) überlegen

Rekursion: Ideen und Beispiele

- Ideen für Rekursion:
  - Erstes/Letztes Element und Rest
  - Menge halbieren ("Teile und Herrsche")
  - ...
- Weitere Beispiele:
  - Summe der ersten n Zahlen
  - Sortieren (später!)
  - Divisionsrestverfahren zur Zahlenkonvertierung
  - Pascalsches Dreieck
  - Türme von Hanoi
  - ...
- Rekursion mit mehreren Variablen, Beispiel: Ackermannfunktion
  - a(0,m) = m+1
  - a(n+1,0) = a(n,1)
  - a(n+1,m+1) = a(n,a(n+1,m))

Gibt es Fragen? (Es gibt keine dummen Fragen!)



"Excuse me, is this the Society for Asking Stupid Questions?"