

Hochschule Deggendorf Prof. Dr. Peter Jüttner	
Vorlesung: Algorithmen und Datenstrukturen	SS 2012
Probeklausur	Termin 14.6.12
Matrikelnummer	

Probeklausur

Lösungshinweis zur Aufgabe 1:

Zuerst sollte die Lösungsidee zu den Teilaufgaben 1.1 und 1.2 plausibel erklärt werden, also welche Funktion ist größer bzgl. der O-Notation?

Um dann einen formalen Beweis vorzulegen, gibt es zwei Strategien:

1. Wollen Sie zeigen, dass die Aussage $f(n) \in O(g(n))$ wahr ist, so müssen Sie zeigen, dass gilt:

$$\exists c > 0, \exists n_0 > 0 : \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

es muss also ein Paar (c, n_0) angegeben werden, für das diese Aussage wahr ist.

2. Wollen Sie zeigen, dass die Aussage nicht gilt, so können Sie zeigen, dass gilt:

$$\lim_{n \rightarrow \infty} f(n)/g(n) = \infty$$

Insbesondere, wenn Zähler und Nenner gegen $+\infty$ streben, hilft oft die Regel von L'Hospital:

$$\lim_{n \rightarrow \infty} f(n)/g(n) = \lim_{n \rightarrow \infty} f'(n)/g'(n)$$

Lösungshinweis zur Aufgabe 2: Plausible Begründung genügt

1. Aufgabe: O-Notation

Gelten folgende Aussagen? Geben Sie eine Begründung an. (6 Punkte, jeweils 3)

Einfache Argumentation: $\log n$ „wächst langsamer“ als n , n „wächst langsamer als

$n \cdot \log n$, $n \cdot \log n$ „wächst langsamer“ als n^2 , n^2 wächst langsamer als 2^n , ...

- $100 \cdot n \in O(\sqrt{n})$
Falsch, denn $\lim_{n \rightarrow \infty} 100n / \sqrt{n} = \infty$
Richtig, für $c=1$ und für $n \geq n_0=100$ gilt $100 \cdot n \leq n^2$
- $100 \cdot n \in O(n \cdot \sqrt{n})$
Richtig, für $c=1$ und für $n \geq n_0=10000$ gilt $100 \cdot n \leq n \cdot \sqrt{n}$

2. Aufgabe Teilmengen Addieren

Gegeben sei eine Folge von Zahlen a_0, \dots, a_n .

Gesucht ist für alle Teilfolgen a_1, \dots, a_i mit $i = 0, \dots, n$ die Summe der Teilfolge

Beispiel: Der Algorithmus soll aus einem Eingabefeld
[7, 2, 6, 5, 1, 9] die Ausgabe [7, 9, 15, 20, 21, 30] erzeugen, da
7 die Summe der Teilsequenz [7],
9 die Summe der Teilsequenz [7, 2],
15 die Summe der Teilsequenz [7, 2, 6],
20 die Summe der Teilsequenz [7, 2, 6, 5],
21 die Summe der Teilsequenz [7, 2, 6, 5, 1],
30 die Summe der Teilsequenz [7, 2, 6, 5, 1, 9]
sind.

Gegeben ist folgender Lösungsalgorithmus:

```
void teilsomme(int input[], int output[], int n)
{ int i,j;
  int summe;
  output[0] = input[0];
  for (i=1; i<n; i++)
  { /* Berechne Summe der Teilsequenz input[0, ... , i] */
    summe = 0;
    for (j=0; j<=i; j++)
      summe = summe + input[j];
    output[i] = summe;
  };
};
```

- Analysieren Sie den obigen Algorithmus und geben Sie seine Komplexität in der O-Notation an. Begründen Sie Ihre Aussage (4 Punkte)
Der („etwas ungünstige“) Algorithmus hat zwei geschachtelte for-Schleifen, die (vereinfacht) jeweils $O(n)$ habe. Aus der Schachtelung ergibt sich eine Gesamtkomplexität von $O(n^2)$
- Finden Sie ein Verfahren dass die Komplexität $O(n)$ hat. Begründen Sie Ihre Aussage. Nennen Sie nur die Verbesserung, ein Algorithmus muss nicht angegeben werden. (4 Punkte)
Der verbesserte Algorithmus nutzt die Tatsache, dass im Ausgabefeld schon die

Summe der vorhergehenden Folge steht. Nur zu dieser muss das aktuelle Element addiert werden.

```
void teilsomme_verbessert(int input[], int output[], int n)
{ int i,j;
  output[0] = input[0];
  for (i=1; i<n; i++)
  { /* Berechne Summe der teilsequenz input[0, ... , i] */
    output[i] = input[i] + output[i-1];
  };
};
```

3. Aufgabe: Rekursives Suchen in einer Zeichenkette:


Entwickeln Sie eine C-Funktion, die rekursiv in einer Zeichenkette vom Typ char[] feststellt, wie oft ein bestimmtes Zeichen in der Zeichenkette vorhanden ist. Dies soll als Ergebnis der Funktion zurückgegeben werden. Sowohl die Zeichenkette als auch das gesuchte Zeichen und die Länge der Zeichenkette (Anzahl der relevanten Zeichen in der Zeichenkette) sollen als Parameter übergeben werden. Fehlerfälle z.B. nicht plausible Länge müssen nicht betrachtet werden. Gross- und Kleinbuchstaben dürfen als unterschiedliche Zeichen betrachtet werden. (14 Punkte)

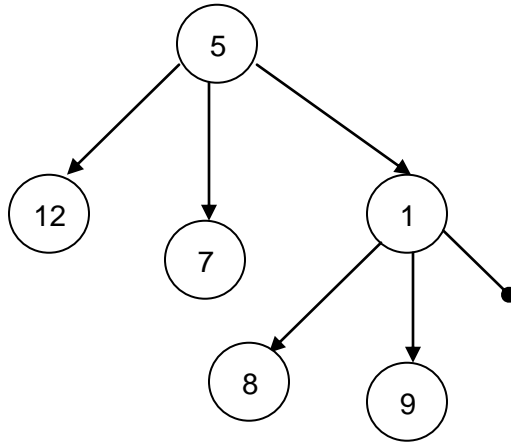
Tipps:

- Überlegen Sie, in welchen Fällen Ihr Algorithmus zu einem Ergebnis kommt. Formulieren Sie diese Fälle als Terminierungsfälle.
- Fügen Sie bei Bedarf einen weiteren Parameter hinzu.

```
unsigned int zeichenzaehlen(char s[], unsigned int laenge, unsigned int start, char c)
{ if (laenge == 0)
  return 0;
  else if (start == laenge)
  return 0;
  else if (s[start] == c)
    return zeichenzaehlen(s,laenge,start+1,c) + 1;
  else return zeichenzaehlen(s,laenge,start+1,c);
}
```

4. Ternärbäume

- a) Schreiben Sie eine C-Datenstruktur für Ternärbäume folgender Art. Ein Ternärbaum ist ein Baum, der in einem Knoten eine Integer Zahl speichert und bei dem von einem Knoten maximal drei Teilbäume ausgehen (s. beispielhafte grafische Darstellung,  steht für einen leeren Teilbaum). Die Verzweigung soll über Pointer realisiert werden. Ein Nullpointer steht für einen leeren Baum. (3 Punkte)



```

typedef struct ternaerbaum
{ int knoten;
  ternaerbaum *links;
  ternaerbaum *mitte;
  ternaerbaum *rechts;
}

```

- b) Schreiben Sie eine C-Funktion, die die Zahlen in allen Knoten eines Ternärbaums aufaddiert. Der Ternärbaum soll per Pointer als Parameter an die Funktion übergeben werden. Ein leerer Baum wird durch einen Nullpointer dargestellt. (11 Punkte)

Tipp: Lösen Sie die Aufgabe rekursiv.

```

int knotensumme(ternaerbaum * t)
{ if (t == NULL) return 0;
  else return t->knoten + knotensumme(t->links) + + knotensumme(t->mitte) +
    knotensumme(t->rechts);
}

```

- c) Schreiben Sie eine C-Funktion, die aus einer Zahl i und drei Ternärbäumen einen neuen Ternärbaum erzeugt. Die Zahl i wird direkt und die drei Ternärbaume als Pointer-Parameter an die Funktion übergeben. Dabei soll i in der Wurzel des Baums stehen und die drei Ternärbaume an die Wurzel angehängt werden. Der Wurzelknoten soll auf dem Heap angelegt werden und der neue Baum soll per Pointer als Funktionsergebnis zurückgegeben werden. Fehlerfälle (z.B: kein Platz auf dem Heap) müssen nicht abgefangen werden. Für die Speicherplatzanforderung darf malloc() oder new verwendet werden. (10 Punkte)

```

ternaerbaum* neuer_Baum(int i, ternaerbaum *l, ternaerbaum *m, ternaerbaum
*r)
{
  ternaerbaum *b = (ternaerbaum*)malloc(sizeof(ternaerbaum));
  b->knoten = i;
  b->links = l;
  b->mitte = m;
}

```

```
b->rechts = r;  
return b;  
};
```

4 Punkte Kopfzeile (1 Ergebnis, je 1P pro Parameter)

3 Punkte für new oder malloc

1 Punkt für Variable

1 Punkt für Zuweisen der Parameter an Variable

1 Punkt für return

5. Denksport

Erläutern Sie,

- a) wann es besser ist, bei der Erstellung von Programmen, die mit einer Datenstruktur arbeiten, nur Standardfunktionen der Datenstruktur (z.B. Liste head, tail, append, isempty, ...) zu verwenden, anstatt direkt auf die Implementierung der Datenstruktur (z.B. Liste über Pointer) zuzugreifen. (5 Punkte)
Zugriff über Funktionen, falls die Implementierung der Datenstruktur sich ändern kann, z.B. bei einer Portierung in eine andere Umgebung.
- b) wann es besser ist, bei der Erstellung von Programmen, die mit einer Datenstruktur arbeiten, direkt auf die Implementierung der Datenstruktur (z.B. (z.B. Liste über Pointer) zuzugreifen, anstatt Zugriffsfunktionen zu verwenden. (5 Punkte)
Zugriff über Struktur, falls Zeitverhalten oder Speicherplatz kritisch sind.