Prüfung — Informatik D-MATH/D-PHYS
9. 8. 2013 **15:00–17:00**

Prof. Bernd Gärtner

Kandidat/in:

Name: .....

Vorname: .....

Stud.-Nr.: .....

Ich bezeuge mit meiner Unterschrift, dass ich die Prüfung unter regulären Bedingungen ablegen konnte und dass ich die allgemeinen Bemerkungen gelesen und verstanden habe.

Unterschrift: .....

Allgemeine Bemerkungen und Hinweise:

1. Überprüfen Sie die Vollständigkeit der ausgeteilten Prüfungsunterlagen (drei doppelseitige Blätter mit insgesamt 6 Aufgaben)! **Tragen Sie auf dem Deckblatt gut lesbar Namen, Vornamen und Stud.-Nr. ein.**
2. Erlaubte Hilfsmittel: **Keine. Einzige Ausnahme sind Wörterbücher**.
3. Betrugsversuche führen zu sofortigem Ausschluss und können rechtliche Folgen haben.
4. **Schreiben Sie Ihre Lösungen direkt auf die Aufgabenblätter!** Pro Aufgabe ist höchstens eine gültige Version eines Lösungsversuchs zulässig. **Tipp:** Lösungsentwürfe auf separaten Blättern vorbereiten und die fertige Lösung auf die Aufgabenblätter übertragen. Falls Sie eine Lösung ändern wollen, streichen Sie den alten Lösungsversuch klar erkennbar durch. Falls auf dem Aufgabenblatt nicht mehr genug Platz für Ihre neue Lösung vorhanden ist, benutzen Sie ein separates Blatt, das mit Ihrem Namen und der Aufgabennummer beschriftet ist.
5. Wenn Sie frühzeitig abgeben möchten, übergeben Sie Ihre Unterlagen bitte einer Aufsichtsperson und verlassen Sie den Raum.
6. **Ab 16:50 Uhr kann nicht mehr frühzeitig abgegeben werden. Bleiben Sie an Ihrem Platz sitzen, bis die Prüfung beendet ist und ihre Unterlagen von einer Aufsichtsperson eingesammelt worden sind.**
7. Die Prüfung ist bestanden, wenn Sie 60 von 120 Punkten erzielen. **Viel Erfolg!**

1	2	3	4	5	6	$\sum$

Aufgabe 1. (18 Punkte) Geben Sie für jeden der folgenden 6 Ausdrücke Typ und Wert an! Zwischenschritte der Auswertung geben keine Punkte und müssen nicht aufgeschrieben werden.

Ausdruck	Typ	Wert
`11u + 6u / 3u`		
`23 / 7`		
`10 / 4.0f / 2`		
`3.0e4 * 2.0e-3`		
`1612795 % 5`		
`double(1/2) == 0.5`		

Solution

Expression	Type	Value
`11u + 6u / 3u`	`unsigned int`	13
`23 / 7`	`int`	3
`10 / 4.0f / 2`	`float`	1.25
`3.0e4 * 2.0e-3`	`double`	60
`1612795 % 5`	`int`	0
`double(1/2) == 0.5`	`bool`	*false*

Aufgabe 2. (15 Punkte) Geben Sie für jedes der drei folgenden Code-Fragmente die Folge von Zahlen an, die das Fragment ausgibt!

---

a)
```
for (int i=1; i<1000; i*=3)
  std::cout << i << " ";
```

Ausgabe:

---

b)
```
int a=0;
int b=1;
for (int h; (h=a)<50; b+=h)
  std::cout << (a=b) << " ";
```

Ausgabe:

---

c)
```
unsigned int x = 1;
do {
  std::cout << x << " ";
  x = (7 * x + 4) % 9;
} while (x != 1);
```

Ausgabe:

Solution

a) `1 3 9 27 81 243 729` (cube numbers)

b) `1 1 2 3 5 8 13 21 34 55` (Fibonacci numbers)

c) `1 2 0 4 5 3 7 8 6` (pseudorandom numbers)

Aufgabe 3. (4 / 16 Punkte) Eine natürliche Zahl $n \geq 1$ heisst *einfach*, wenn sie von der Form $n = 2^k \cdot 3^\ell$ ist, für natürliche Zahlen $k, \ell \geq 0$. In anderen Worten, n ist einfach, wenn n ein Produkt einer Zweier- und einer Dreierpotenz ist. Zum Beispiel sind alle Zweier-, Dreier- und Sechserpotenzen einfach. Andere einfache Zahlen sind $18 = 2 \cdot 3^2$ und $24 = 2^3 \cdot 3$.

Im folgenden finden Sie ein Skelett für eine Funktion `is_simple`, die testet, ob eine natürliche Zahl einfach ist. Ergänzen Sie das Skelett zu einer korrekten Funktionsdefinition, indem Sie die Ausdrücke `expr1` und `expr2` angeben!

```
// PRE: n > 0
// POST: returns true if and only if n is simple, that means n is of
//       the form 2^k * 3^l for some natural numbers k,l >= 0
bool is_simple (unsigned int n)
{
  if (n == 1)
    return expr1;
  else
    return expr2;
}
```

```
expr1:
```

```
expr2:
```

Solution

```
expr1: true

expr2:(n % 2 == 0) && is_simple (n / 2) ||
      (n % 3 == 0) && is_simple (n / 3)
```

Aufgabe 4. (25 Punkte) Eine *Permutation* der Menge $\{1, 2, \ldots, n\}$ ist eine bijektive Funktion $\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$. Hier ist als Beispiel eine tabellarische Darstellung einer Permutation von $\{1, 2, 3, 4, 5\}$:

i	1	2	3	4	5
$\pi(i)$	5	3	2	4	1

Eine *Inversion* in einer Permutation π ist ein Paar (i, j) mit $1 \leq i < j \leq n$ und $\pi(i) > \pi(j)$. Im obigen Beispiel ist $(1, 2)$ eine Inversion, denn $\pi(1) = 5 > 3 = \pi(2)$. Das Paar $(3, 4)$ ist keine Inversion, denn $\pi(3) = 2 < 4 = \pi(4)$. Geben Sie eine Definition für die folgende Funktion `inversions` an, die die Anzahl der Inversionen in einer gegebenen Permutation zählt! (Die Permutation im Beispiel hat 8 Inversionen.) Die Gesamtanzahl der Anweisungen darf die vorgegebene Anzahl von Zeilen nicht überschreiten.

```
// PRE: the range [begin, end) stores a permutation
//      pi of the set {1,2,...,end-begin}, i.e. the
//      i-th element of the range is pi(i), for all
//      i in {1,2,...,end-begin}
// POST: the number of inversions of pi is returned
int inversions (const int* begin, const int* end)
{
```

```
                                                    // 1
                                                    // 2
                                                    // 3
                                                    // 4
                                                    // 5
                                                    // 6
}
```

Solution

```
int inversions (const int* begin, const int* end) {
  int result = 0;
  for (const int* ip = begin; ip < end; ++ip)
    for (const int* jp = ip+1; jp < end; ++jp)
      if (*ip > *jp) ++result;
  return result;
}
```

Aufgabe 5. (30 Punkte) Aus der Vorlesung kennen wir die *Liste* als Datenstruktur zum Speichern einer Folge von Schlüsseln des gleichen Typs. Anders als ein Feld erlaubt eine Liste effizientes Einfügen und Löschen "in der Mitte". Für den Typ `int` ist hier der Teil der Klassendefinition, der relevant für diese Aufgabe ist.

```
class List{
public:
  ...
  // POST: returns const pointer to the first node
  const ListNode* get_head() const;
  ...
private:
  ...
  ListNode* head_;
  ...
};
```

Eine Liste speichert also einen (möglicherweise Null-) Zeiger auf den Kopf-Knoten, dessen Schlüssel das erste Element der Liste ist. Ausgehend vom Kopf-Knoten können wir die Liste durchlaufen, indem wir den `get_next()`-Zeigern der Klasse `ListNode` folgen, deren relevanter Teil der Definition hier angegeben ist:

```
class ListNode {
```

```
 public:
  ...
  int get_key() const;
  ListNode* get_next() const;
  ...
 private:
  int key_;
  ListNode* next_;
};
```

Implementieren Sie einen Gleichheitstest für zwei Listen als globalen `operator==`! Zwei Listen sind gleich, wenn sie die gleiche Folge von Schlüsseln speichern.

```
// POST: returns true if and only if l1 and l2
//       store the same sequence of keys
bool operator== (const List& l1, const List& l2)
{









}
```

Solution

```
bool operator== (const List& l1, const List& l2) {
  const ListNode* p1 = l1.get_head();
  const ListNode* p2 = l2.get_head();
  while (p1 != 0 && p2 != 0) {
    if (p1->get_key() != p2->get_key()) return false;
    p1 = p1->get_next();
    p2 = p2->get_next();
  }
  return (p1 == 0 && p2 == 0);
}
```

Aufgabe 6. (9 / 3 Punkte) Ein *binärer Suchbaum* ist entweder leer, or er hat einen *Wurzel*-Knoten mit einem *Schlüssel* (eine relle Zahl), einen *linken* Teilbaum, und einen *rechten* Teilbaum, mit den folgenden Eigenschaften.

- Der linke Teilbaum ist ein binärer Suchbaum, der nur Knoten enthält, deren Schlüssel *kleiner* sind als der Schlüssel der Wurzel.
- Der rechte Teilbaum ist ein binärer Suchbaum, der nur Knoten enthält, deren Schlüssel *grösser* sind als der Schlüssel der Wurzel.

Sei T_N die Menge der binären Suchbäume, die alle Schlüssel aus $\{1, \ldots, N\}$ enthalten, für eine gegebene natürliche Zahl $N \geq 0$. Das heisst, jeder Baum in T_N hat N Knoten. Abbildung 1 zeigt die Mengen T_1, T_2 und T_3.

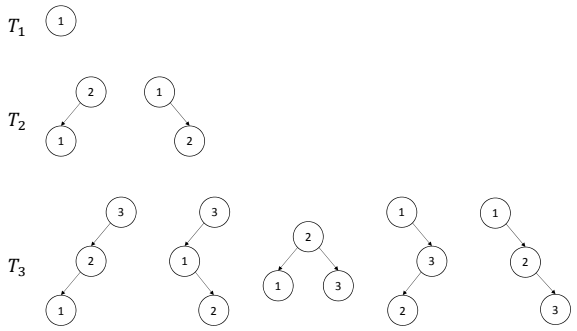

Figure 1: Die Mengen T_1, T_2 und T_3

a) Geben Sie eine *rekursive* Definition der Function $f : \mathbb{N} \to \mathbb{N}$ mit $f(N) = |T_N|$ an! Das heisst, $f(N)$ ist die Anzahl der binären Suchbäume mit Schlüsselmenge genau $\{1, \ldots, N\}$. Sie müssen nicht argumentieren, warum Ihre Definition korrekt ist. **Hinweis:** Sie sollten mit Hilfe Ihrer Definition die folgenden Werte erhalten: $f(0) = f(1) = 1, f(2) = 2, f(3) = 5$.

b) Berechnen Sie $f(4) = |T_4|$, unter Verwendung von a) (oder irgendeiner anderen Methode)! Sie müssen nicht argumentieren, warum Ihr Wert stimmt.

$$f(4) =$$

Solution Our base case is $f(0) = |T_0| = 1$.

Suppose, that we want to compute $f(N)$ for some positive N. We know that each key from the set $\{1, \ldots, N\}$ can be the root of a tree in T_N. Let us fix $k \in \{1, \ldots, N\}$ to be the key of the root node. Then, we know that the left subtree consists of nodes with keys from $\{1, \ldots, k-1\}$ and the right subtree has nodes with keys from $\{k+1, \ldots, N\}$. Therefore, we know that there are $f(k-1)$ possible left subtrees and $f(N-k)$ possible right subtrees. Furthermore, left and right subtrees are independent. Hence, we can combine any possible left subtree with any possible right subtree. Therefore, there are $f(k-1)f(N-k)$ trees rooted on the node with key k in T_N. Finally,

$$f(N) = \sum_{k=1}^{N} f(k-1)f(N-k).$$

For b), we obtain from this formula $f(4) = 14$.