Prüfung — Informatik D-MATH/D-PHYS
24. 1. 2014 09:00–11:00

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 (vier doppelseitige Blätter mit insgesamt 6 Aufgaben und ein leeres Notizblatt)! **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 10:45 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 C++-Typ und Wert an! Zwischenschritte der Auswertung geben keine Punkte. Nehmen Sie an, dass x vom Typ **int** ist, und *zu Beginn jeder Teilaufgabe* den Wert 3 hat.

Solution

Expression	Type	Value
`0 < 1 && 1 != 1 || ++x > 3`	`bool`	*true*
`12.0 / 3u / 2u`	`double`	2.0
`2 + x++`	`int`	5
`1.4e3 * 0.2e-2`	`double`	2.8
`2014 % 4 + x % 2`	`int`	3
`x / 2 + 9.0f / 6`	`float`	2.5

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=0; i<100; i*=2)
  std::cout << ++i << " ";
```

Ausgabe:

---

b)
```
int i=0;
int j=5;
while (i != j)
  std::cout << (i+=2) - j++ << " ";
```

Ausgabe:

---

c)
```
void f(unsigned int x) {
  if (x == 0)
    std::cout << "-";
  else {
    std::cout << "*";
    f(x-1);
    std::cout << "*";
  }
}
f(5);
```

Ausgabe:

Solution

a) `1 3 7 15 31 63` ($2^k - 1$, for $k = 1, ...6$)

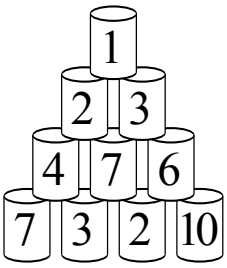
b) `-3 -2 -1 0 1`

c) `*****-*****`

Aufgabe 3. (23 Punkte) Beim Büchsenwerfen sind Büchsen pyramidenförmig angeordnet. Jeder Büchse ist eine bestimmte Punktzahl zugeordnet. Diese Punktzahlen sind in einem zweidimensionalen Feld (Array) gespeichert. Der erste Index des Feldes bestimmt die Reihe in der die Büchse steht und der zweite Index die Position innerhalb der Reihe. Wir zählen Reihen von oben nach unten, beginnend bei 0, und Positionen (Kolonnen) von links nach rechts, beginnend bei 0. Im Beispiel unten hat die Büchse in Reihe 2 an Position 0 den Wert 4 (`D[2][0] == 4`).

Wird eine Büchse getroffen, fällt diese herunter. Es fallen auch alle Büchsen die direkt oberhalb einer fallenden Büchse stehen herunter. Wird zum Beispiel die Büchse mit der Punktzahl 7 in der Mitte der Reihe 2 getroffen, fallen die Büchsen mit den Punkzahlen 7, 2, 3 und 1.

```
int D[4][4] = {{1, 0, 0, 0},
               {2, 3, 0, 0},
               {4, 7, 6, 0},
               {7, 3, 2, 10}};
```



Im folgenden finden Sie ein Skelett für eine Funktion `hit`, die die Summe der Punktzahlen auf den gefallenen Büchsen berechnet, falls mit dem Wurf die Büchse an Position `c` in der Reihe `r` getroffen wurde. Ergänzen Sie das Skelett zu einer korrekten Funktionsdefinition, indem Sie die vier Ausdrücke `expr1`, `expr2`, `expr3` und `expr4` angeben! Sie können davon ausgehen, dass sie in der Funktion direkt auf das Feld `D` zugreifen können, d.h. `D` ist eine globale Variable.

```
// PRE:  the array D has size at least (n+1) x (n+1) (n = max(r, c)),
//       D[r][c] denotes the value of the can in row r, column c
//       where c <= r
// POST: returns the sum of the values of all cans that fall
//       if the can at position/column c in row r gets hit
int hit(int r, int c)
{
  if (r == 0)
    return expr1;
  if (c == 0)
```

```
    return expr2;
  if (c == r)
    return expr3;
  return expr4;
}
```

Solution For `expr2-4` one has to call the function `hit` *recursively*.

```
expr1: D[0][0];

expr2: D[r][c] + hit(r-1, c);

expr3: D[r][c] + hit(r-1, c-1);

expr4: D[r][c] + hit(r-1, c-1) + hit(r-1, c) - hit(r-2, c-1);
```

Aufgabe 4. (24 Punkte) Ihre Aufgabe ist es, eine Funktion zu implementieren, die eine Fliesskommazahlen-Approximation der Exponentialfunktion

$$e^x := \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

liefert, für eine gegebene reelle Zahl x. Die Implementierung soll auf der Definition von e^x als unendliche Reihe basieren. Um eine Approximation von e^x zu erhalten, soll die Funktion die ersten Terme dieser Reihe aufsummieren (mit `double`-Arithmetik), *bis* die Addition des aktuellen Terms den bisher erhaltenen Wert nicht mehr ändert. Sie dürfen *keine* externen Bibliotheken benutzen. Die *gesamte* Anzahl an Anweisungen in Ihrer Lösung darf die vorgegebene Anzahl an Zeilen nicht überschreiten.

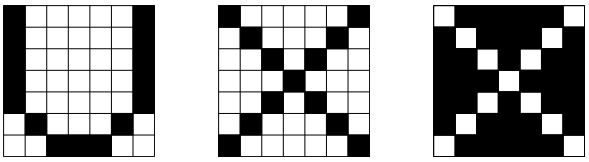
Solution

```
double exp (double x) {
   double t = 1.0;     // 1/i!
   double ex = 1.0;    // i-th approximation of e
   double xi = 1.0;    // x^i
   double ex_old = 0.0;
   for (unsigned int i = 1; ex != ex_old; ++i) {
     ex_old = ex;
     xi *= x;
     t /= i;
     ex += xi * t;
   }
   return ex;
}
```

Important ingredients of the solution are

- a variable `t` to store $\frac{1}{i!}$. (or $i!$)
- a variable `xi` to store x^i
- a variable `ex` to store the i-th approximation of e^x.
- a variable `ex_old` to store the previous ($(i-1)$-th) approximation of e^x.
- a loop updating the variables. Repeat as long as `ex != ex_old`.

Aufgabe 5. (14 / 12 Punkte) Betrachten Sie die folgende partielle Deklaration einer Klasse zur Repräsentierung von Schwarz-Weiss Bildern mit 7×7 Pixeln. Hier sind drei Beispiele für solche Bilder.


← Bild für Aufgabenteil b)

```
// a class for 7x7 black-and-white pixel images
class image {
private:
  // bitmap[r][c] = true if and only if the pixel
  // in row r and column c is black
  bool bitmap[7][7];

public:
  // POST: *this is the empty image (all pixels are white)
  image ();

  // PRE: 0 <= r,c < 7
  // POST: the pixel in row r and column c in *this is set to black
  void set_pixel (int r, int c);

  // POST: the inverted image of *this is returned (black pixels
  //       become white and white pixels become black)
  image operator! () const;
};
```

a) Geben Sie korrekte Definitionen für die drei öffentlichen Mitgliedsfunktionen an! Die *gesamte* Anzahl an Anweisungen in Ihrer Lösung darf die jeweils vorgegebene Anzahl an Zeilen nicht überschreiten.

b) Nehmen Sie an, dass ein Ausgabeoperator für die Klasse `image` existiert und definieren Sie die Hauptfunktion so, dass das Bild ganz rechts auf der vorhergehenden Seite ausgegeben wird!

Solution Here are the code snippets for part a). We present two slightly different solutions for `operator!`.

```
image::image()
{
  for (int r=0; r<7; ++r)
    for (int c=0; c<7; ++c)
      bitmap[r][c] = false;
}

void image::set_pixel (int r, int c)
{
  bitmap[r][c] = true;
}

// version 1
image image::operator! () const
{
  image result;
  for (int r=0; r<7; ++r)
    for (int c=0; c<7; ++c)
      result.bitmap[r][c] = !bitmap[r][c];
  return result;
}

// version 2
image image::operator! () const
{
  image result;
  for (int r=0; r<7; ++r)
    for (int c=0; c<7; ++c)
      if (!bitmap[r][c])
        result.set_pixel(r,c);   //or result.bitmap[r][c] = true;
  return result;
}
```

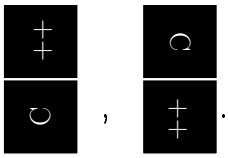
For part b) there are as well several possible solutions. An elegant way would be to use `operator!` that was defined in part a). Another option is to iterate over all pixels and set them to black if they are not on any of the two diagonals.

```
// version 1
```

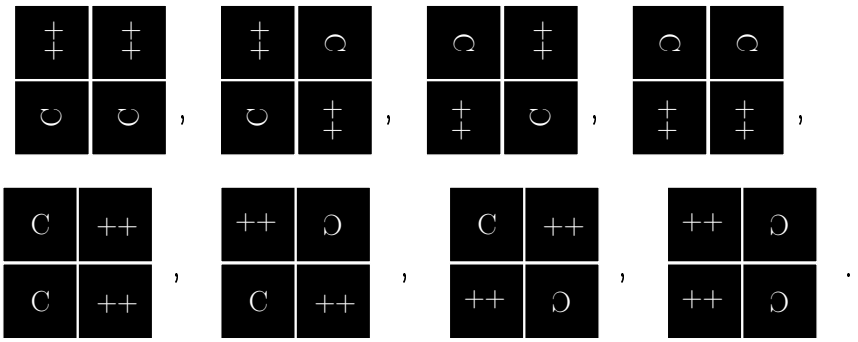
```
int main()
{
  image cross;
  for (int r=0; r<7; ++r) {
    cross.set_pixel(r,r);
    cross.set_pixel(r,6-r);
  }
  std::cout << !cross << std::endl;
  return 0;
}

// version 2
int main()
{
  image cross;
  for (int r=0; r<7; ++r)
    for (int c=0; c<7; ++c)
      if (c != r && c != (6-r))
        cross.set_pixel(r, c);
  std::cout << cross << std::endl;
  return 0;
}
```

Aufgabe 6. (10 / 4 Punkte) Sei n eine positive natürliche Zahl. Ein Spielfeld der Grösse $2 \times n$ soll vollständig mit Dominosteinen der Grössse 1×2 belegt werden, wobei jeder Dominostein wie folgt aussieht:

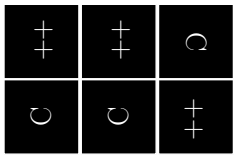
Sei $T(n)$ die Anzahl der verschiedenen Möglichkeiten, das Spielfeld zu belegen. Wir erhalten zum Beispiel $T(1) = 2$:



Für $n = 2$ ergibt sich $T(2) = 8$:



Für $n = 3$ zeigen wir hier nur eine der vielen Möglichkeiten:



a) Geben Sie eine rekursive Formel für $T(n)$ an, für $n \geq 3$. Sie müssen die Korrektheit der Formel nicht beweisen.

$T(n) =$, falls $n \geq 3$.

b) Berechnen Sie unter Benutzung von a), oder auf irgendeine andere Weise, den Wert $T(4)$.

$T(4) =$

Solution. We have two mutually exclusive options: either we fill the two leftmost columns with two horizontal dominos (there are 4 ways to do this), and then fill the remaining $n-2$ columns; or we fill the leftmost column with a vertical domino (there are 2 ways to do it), and then fill the remaining $n-1$ columns. This yields

$$T(n) = 4T(n-2) + 2T(n-1), \quad n \geq 3.$$

We obtain $T(3) = 4T(1)+2T(2) = 4\cdot 2+2\cdot 8 = 24$ and $T(4) = 4T(2)+2T(3) = 4\cdot 8+2\cdot 24 = 80$. It is in fact not hard to see that $T(n) = 2^n F_{n+1}$, where F_n is the n-th Fibonacci number ($F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$). This is easily obtained from the recurrence, or seen as follows: if the dominos are unlabeled, there are F_{n+1} different possibilities (a known application of Fibonacci numbers). Taking the labels into account, each of the n dominos can be in one of two orientations, for a total of 2^n.