Tentamen Imperatief Programmeren

Vrijdag 1 februari 2008, 9:00–12:00 uur, zaal 5412.0025 (WSN 25)

- Schrijf boven ieder blad je naam, studentnummer, en volgnummer van het blad. Schrijf op het eerste blad het aantal ingeleverde bladen.
- Lees eerst een opgave volledig door alvorens deze te maken.
- Schrijf netjes en zorgvuldig met een pen (geen potlood).
- Je hebt 3 uur de tijd. Gebruik deze nuttig. Als je snel klaar bent, gebruik dan de resterende tijd om je antwoorden nog eens te controleren.
- Succes!

Opgave 1: Toekenningen
Bepaal voor ieder van de onderstaande annotaties de keuze die op de plaats van de lege regel (.....) ingevuld kan worden. Per onderdeel is er precies één keuze mogelijk. De variabelen `x` en `y` zijn van het type `int`. Let erop dat `X` en `Y` (met hoofdletter!) specificatie-constanten (en dus geen variabelen) zijn.

```
1.1  // x==X+1
     .....
     // x==7*X + 12

     (a) x=7*(X-1)+12;
     (b) x=7*x+5;
     (c) x=7*x+11;
```

```
1.2 // 7*X-2<=x<7*X+5
    .....
    // x==X

    (a) x=x/7;
    (b) x=(x-5)/7;
    (c) x=(x+2)/7;
```

```
1.3 // x==X+Y, y==2*X-7
    .....
    // x==X+Y, y=Y

    (a) y=(2*x-7-y)/2;
    (b) y=(2*y-7-x)/2;
    (c) y=(y+7)/2;
```

```
1.4 // x==X, y==Y
        y=x; x=y;
        .....

    (a) // x==Y, y==X
    (b) // x==X, y==X
    (c) // x==Y, y==Y
```

```
1.5 // x==X, y==Y
        x=x+y; y=x-y; x=x-y;
        // .....

    (a) // x==Y, y==X
    (b) // x==X, y==Y
    (c) // x==X, y==X
```

```
1.6 // x==X+1, y==Y;
        x=x+y; y=x-y; x=x-y;
        .....

    (a) // x==X+1, y==X
    (b) // x==Y, y==X+1
    (c) // x==Y+1, y==X
```

Opgave 2: Zeef van Eratosthenes
De Zeef van Eratosthenes (circa 240 v.Chr.) is een al zeer lang bekend algoritme om priemgetallen te vinden. Beginnende met een gesorteerde lijst van alle getallen van 2 tot een bepaald maximum, is het kleinste niet-doorgestreepte getal (2) een priemgetal. Je streept alle veelvouden van dit priemgetal door in de lijst, en herhaalt deze stappen voor elk volgend kleinste niet-doorgestreepte getal, zolang het volgende getal kleiner of gelijk is aan de wortel van het maximum. De getallen die op deze manier overblijven zijn alle priemgetallen tot het maximum.

Het onderstaande programma (zie volgende vel) bepaalt alle priemgetallen tussen 0 en 1000 en drukt deze op volgorde af. Het programma bevat echter 5 fouten. Geef voor iedere fout een correctie.

```
1   class Priemlijst {
2       int [] lijst;
3
4       void maakLegeLijst(int max) {
5           int bovengrens;
6           bovengrens = max;
7           lijst = new boolean [max + 1];
8           for (int i = 0; i <= bovengrens; i++) {
9               lijst[i] = true;
10          }
11      }
12
13      void nietPriem(int index) {
14          lijst[index] = false;
15      }
16
17      boolean isPriem(int p) {
18          return lijst[p];
19      }
20
21      void drukAf() {
22          for (int i = 0; i <= bovengrens; i++) {
23              if (isPriem(i)) {
24                  System.out.print(i + "\t");
25              }
26          }
27          System.out.println();
28      }
29
30      void berekenPriemLijst(int max) {
31          maakLegeLijst();
32
33          int startpunt = 2;
34          while (startpunt*startpunt <= bovengrens) {
35              int factor = 2;
36              while (factor*startpunt <= bovengrens) {
37                  nietPriem(factor*startpunt);
38              }
39              do {
40                  startpunt++;
41              } while (startpunt*startpunt <= bovengrens && !isPriem(startpunt));
42          }
43      }
44
45      public Priemlijst() {
46          berekenPriemLijst(1000);
47          drukAf();
48      }
49
50      public static void main(String[] args) {
51          new Priemlijst();
52      }
53  }
```

Opgave 3: Tijdscomplexiteit: time-management is ingewikkeld ...
Geef van ieder van de volgende programmafragmenten aan wat de scherpste bovengrens is (in termen van `N`, waarbij `N` > 0) voor het aantal rekenstappen dat het fragment uitvoert. M.a.w. een algoritme dat N stappen doet is $O(N)$ en niet $O(N^2)$ omdat $O(N)$ de scherpste bovengrens is.

3.1
```
bits=0;
while (N!=0) {
 bits++;
 N=N/2;
}
```

(a) $O(1)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(\log N)$ (e) $O(N^2)$

3.2
```
cijfers=0;
while (N!=0) {
 cijfers++;
 N=N/10;
}
```

(a) $O(1)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(\log N)$ (e) $O(N^2)$

3.3
```
som=0;
i=0;
j=N;
while (i<j) {
 som += i+j;
 i++;
 j--;
}
```

(a) $O(1)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(\log N)$ (e) $O(N^2)$

3.4
```
som=N;
som+=N/2;
som+=N/4;
som+=N/8;
som+=N/16;
```

(a) $O(1)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(\log N)$ (e) $O(N^2)$

3.5
```
for (int i=0; i<N; i++) {
 c[i]=0;
 for (int j=0; j<N; j++) {
  c[i] += a[i][j]*b[j];
 }
}
```

(a) $O(1)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(\log N)$ (e) $O(N^2)$

3.3
```
som=0;
i=0;
while (som<N) {
 som += i;
 i++;
}
```

(a) $O(1)$ (b) $O(\sqrt{N})$ (c) $O(N)$ (d) $O(\log N)$ (e) $O(N^2)$

Opgave 4: Sorteren en Zoeken
De onderstaande methode `invoegSort(int[] rij)` sorteert de rij `rij` in oplopende volgorde m.b.v. het z.g.n. insertion sort algoritme.

```
void recursiefInvoegSort(int i, int[] rij) {
     if (i==rij.length) {
       return;
     }

     for (int j=i; (j>0) && (rij[j]<rij[j-1]); j--) {
        int h = rij[j];
        rij[j] = rij[j-1];
        rij[j-1] = h;
     }

     recursiefInvoegSort(i+1, rij);
}

void invoegSort(int[] rij) {
     recursiefInvoegSort(0, rij);
}
```

(a) De methode maakt gebruik van recursie (via `recursiefInvoegSort`. Schrijf een iteratieve (niet-recursieve) versie van de methode `invoegSort(int[] rij)`.

(b) Schrijf een iteratieve methode `zoek(int[] rij, int waarde)` die de positie van `waarde` in een ongesorteerde array `rij` oplevert. Als de waarde niet voorkomt dient de methode -1 op te leveren.
(c) De onderstaande iteratieve methode `snelZoek(int[] rij, int waarde)` levert de positie van `waarde` in een gesorteerde array `rij`. Als de waarde niet voorkomt retourneert de methode -1. Schrijf een recursieve versie van deze methode.

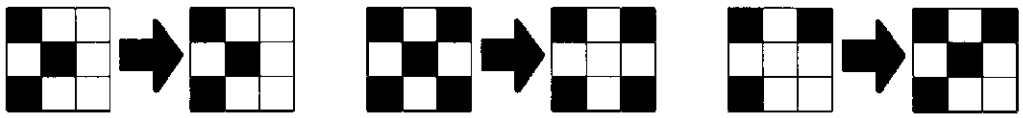
```
int snelZoek(int[] rij, int waarde) {
     int idx=0;
     int i=1;
     while (idx < rij.length) {
        if (rij[idx] == waarde) {
           return idx;
        }
        if ((idx+i < rij.length) && (rij[idx+i] < waarde)) {
           i++;
        } else {
           i = 1;
        }
        idx += i;
     }
     return -1;
}
```

Opgave 5: Conway's game of Life
De *Game of Life* is geen echt spel zoals bijvoorbeeld Mens-Erger-Je-Niet of Schaken; er zijn geen spelers en men kan niet winnen of verliezen. Het 'spel' werd in 1970 uitgevonden door de wiskundige John Conway. Hij was geïnteresseerd in de evolutie van kunstmatige (computer gegenereerde) celkolonies.

Het 'spel' speelt zich af op een rechthoekig stuk ruitjespapier. Ieder vakje wordt een cel genoemd. Nadat een aantal cellen is ingekleurd, begint het spel. Het spel bestaat uit het simuleren van het 'leefgedrag' van de ingekleurde cellen. Game of Life werkt met 'generaties'. Om te bepalen of een cel in de volgende generatie gekleurd is of juist niet wordt de onderstaande verzameling regels gebruikt.

Een ingekleurde cel heet 'levend' en een niet ingekleurde cel heet 'dood'. Elke cel heeft 8 buurcellen.

- Als een levende cel door 2 of 3 levende buurcellen omgeven wordt, dan blijft deze cel leven.
- Als een levende cel door 4 of meer levende buurcellen omgeven wordt, dan gaat deze cel dood door overbevolking.



Figuur 1: Drie voorbeelden van het evolutiegedrag van de middelste cel. Links: De cel overleeft omdat hij twee buren heeft. Midden: De cel sterft uit (overbevolking). Rechts: Geboorte.

- Als een levende cel door minder dan twee levende buurcellen omgeven wordt, dan gaat deze cel ook dood, maar dan door eenzaamheid.
- Als een dode cel wordt omgeven door precies 3 levende buurcellen, dan wordt deze dode cel levend ('geboren').

Al deze transformaties geschieden gelijktijdig voor alle cellen. Men zou dit kunnen doen op de volgende manier.

1. Maak het beginpatroon met zwarte stenen op een groot dambord.
2. Inspecteer alle stenen. Heeft een steen twee of drie buren, leg er dan een witte steen onder.
3. Inspecteer alle lege cellen. Heeft een cel precies drie zwarte buren, leg er dan een witte steen neer.
4. Verwijder alle zwarte stenen en vervang de witte stenen door zwarte.
5. Dit is de tweede generatie. Ga terug naar stap 2.

In 1970 kon lang niet iedereen over een computer beschikken. Thans spreekt het vanzelf dat men voor het spel een computer gebruikt, zodat er geen fouten worden gemaakt. In deze opgave gaan we een programma maken waarmee we dit spel kunnen spelen.

Een deel van het programma `Life` is al voor je geïmplementeerd. Zie hiervoor de volgende pagina. Het programma leest m.b.v. de methode `leesBeginGeneratie` een startgeneratie. De invoer van het programma ziet er als volgt uit: twee gehele getallen groter dan 0, respectievelijk de `hoogte` en `breedte` van het bord. Vervolgens volgen er `hoogte` regels van `breedte` nullen of enen. Een nul betekent dat een cel dood is, terwijl een 1 betekent dat een cel levend is.

(a) De methode `int aantalBuren(int rij, int kolom)` bepaalt voor cel `(rij,kolom)` het aantal levende buren. Schrijf deze methode.

(b) De methode `void berekenGeneratie()` berekent de volgende generatie op basis van de bovenstaande regels. Schrijf deze methode. Maak hierbij gebruik van de methode `aantalBuren`.

(c) Maak het hoofdprogramma, m.a.w. `public Life()`, af. Het programma moet de gebruiker vragen hoeveel generaties hij wil berekenen. Vervolgens bepaalt het programma de reeks van generaties. Toon in iedere iteratie de generatie op het scherm.

Het programma Life:

```
1   import java.util.Scanner;
2
3   class Life {
4       Scanner sc = new Scanner(System.in);
5       int hoogte, breedte;
6       boolean[][] generatie;
7
8       void leesBeginGeneratie() {
9         hoogte  = sc.nextInt();
10        breedte = sc.nextInt();
11        generatie = new boolean [hoogte][breedte];
12        for (int i=0; i<hoogte; i++) {
13            for (int j=0; j<breedte; j++) {
14                generatie[i][j] = (sc.nextInt() == 1);
15            }
16        }
17      }
18
19      void toonGeneratie() {
20        for (int rij=0; rij<hoogte; rij++) {
21            for (int kolom=0; kolom<breedte; kolom++) {
22                if (generatie[rij][kolom]) {
23                    System.out.print ("*");
24                } else {
25                    System.out.print (".");
26                }
27            }
28            System.out.println ();
29        }
30        System.out.println();
31      }
32
33      int aantalBuren(int rij, int kolom) {
34        .......... // bereken het aantal buren van cel (rij,kolom)
35      }
36
37      void berekenGeneratie() {
38        .......... // bereken de volgende generatie
39      }
40
41      public Life () {
42         leesBeginGeneratie();
43         toonGeneratie();
44         .......... // maak dit zelf af
45      }
46
47      public static void main(String [] args) {
48          new Life();
49      }
50  }
```