be-OI 2024 **Finale - BELOFTE** Zaterdag 23 maart 2024	**Invullen in HOOFDLETTERS en LEESBAAR aub** VOORNAAM : ..... NAAM : ..... SCHOOL :	**O** **Gereserveerd**

Finale van de Belgische Informatica-olympiade (duur : **maximum 2u**)

Algemene opmerkingen (lees dit aandachtig voor je begint)

1. Controleer of je de juiste versie van de vragen hebt gekregen (die staat hierboven in de hoofding).
   - De categorie **belofte** is voor leerlingen tot en met het 2e middelbaar,
   - de categorie **junior** is voor het 3e en 4e middelbaar,
   - de categorie **senior** is voor het 5e middelbaar en hoger.
2. Vul duidelijk je voornaam, naam en school in, **alleen op dit blad**.
3. **Jouw antwoorden** moet je invullen op de daarvoor voorziene antwoordbladen. Schrijf **duidelijk leesbaar** met blauwe of zwarte **bic of pen**.
4. Gebruik een potlood en een gom wanneer je in het klad werkt.
5. Je mag alleen schrijfgerief bij je hebben. Rekentoestel, mobiele telefoons, ... zijn **verboden**.
6. Je mag altijd extra kladpapier vragen aan de toezichthouder of leerkracht.
7. Wanneer je gedaan hebt, geef je deze eerste bladzijde terug (met jouw naam erop) en de pagina's met jouw antwoorden, al de rest mag je bijhouden.
8. Voor alle code in de opgaven werd **pseudo-code** gebruikt. Op de volgende bladzijde vind je een **beschrijving** van de pseudo-code die we hier gebruiken.
9. Als je moet antwoorden met code, mag dat in **pseudo-code** of in eender welke **courante programmeertaal** (zoals Java, C, C++, Pascal, Python, ...). We trekken geen punten af voor syntaxfouten.

Veel succes!

De Belgische Informatica-olympiade wordt mogelijk gemaakt dankzij de steun van onze leden:

Overzicht pseudo-code

Gegevens worden opgeslagen in variabelen. Je kan de waarde van een variabele veranderen met $\leftarrow$. In een variabele kunnen we gehele getallen, reële getallen of arrays opslaan (zie verder), en ook booleaanse (logische) waarden : waar/juist (**`true`**) of onwaar/-fout (**`false`**). Op variabelen kan je wiskundige bewerkingen uitvoeren. Naast de klassieke operatoren $+$, $-$, $\times$ en $/$, kan je ook $\%$ gebruiken: als a en b allebei gehele getallen zijn, dan zijn a/b en $a\%b$ respectievelijk het quotiënt en de rest van de gehele deling (staartdeling).
Bijvoorbeeld, als $a = 14$ en $b = 3$, dan geldt: $a/b = 4$ en $a\%b = 2$.

In het volgende stukje code krijgt de variabele $leeftijd$ de waarde 17.

```
geboortejaar ← 2007
leeftijd ← 2024 − geboortejaar
```

Als we een stuk code alleen willen uitvoeren als aan een bepaalde voorwaarde (conditie) is voldaan, gebruiken we de instructie **`if`**. We kunnen eventueel code toevoegen die uitgevoerd wordt in het andere geval, met de instructie **`else`**. Het voorbeeld hieronder test of iemand meerderjarig is, en bewaart de prijs van zijn/haar cinematicket in een variabele $prijs$. De code is bovendien voorzien van commentaar.

```
if (leeftijd ≥ 18)
{
  prijs ← 8    // Dit is een stukje commentaar
}
else
{
  prijs ← 6    // Goedkoper!
}
```

Soms, als een voorwaarde onwaar is, willen we er nog een andere controleren. Daarvoor kunnen we **`else if`** gebruiken, wat neerkomt op het uitvoeren van een andere **`if`** binnen in de **`else`** van de eerste **`if`**. In het volgende voorbeeld zijn er 3 leeftijdscategorieën voor cinematickets.

```
if (leeftijd ≥ 18)
{
    prijs ← 8 // Prijs voor een volwassene.
}
else if (leeftijd ≥ 6)
{
    prijs ← 6 // Prijs voor een kind van 6 of ouder.
}
else
{
    prijs ← 0 // Gratis voor kinderen jonger dan 6.
}
```

Wanneer we in één variabele tegelijk meerdere waarden willen stoppen, gebruiken we een array. De afzonderlijke elementen van een array worden aangeduid met een index (die we tussen vierkante haakjes schrijven achter de naam van de array). Het eerste element van een array $arr[\,]$ heeft index 0 en wordt genoteerd als $arr[0]$. Het volgende element heeft index 1, en het laatste heeft index $n - 1$ als de array n elementen bevat. Dus als de array $arr[\,]$ de drie getallen 5, 9 en 12 bevat (in die volgorde) dan is $arr[0] = 5$, $arr[1] = 9$ en $arr[2] = 12$. De lengte van $arr[\,]$ is 3, maar de hoogst mogelijke index is slechts 2.

Voor het herhalen van code, bijvoorbeeld om de elementen van een array af te lopen, kan je een **`for`**-lus gebruiken. De notatie **`for (`** $i \leftarrow a$ **`to`** b **`step`** k**`)`** staat voor een lus die herhaald wordt zolang $i \leq b$, waarbij i begint met de waarde a en telkens verhoogd wordt met k aan het eind van elke stap. Het onderstaande voorbeeld berekent de som van de elementen van de array $arr[\,]$, veronderstellend dat de lengte ervan n is. Nadat het algoritme werd uitgevoerd, zal de som zich in de variabele sum bevinden.

```
sum ← 0
for (i ← 0 to n − 1 step 1)
{
    sum ← sum + arr[i]
}
```

Een alternatief voor een herhaling is een **`while`**-lus. Deze herhaalt een blok code zolang er aan een bepaalde voorwaarde is voldaan. In het volgende voorbeeld delen we een positief geheel getal n door 2, daarna door 3, daarna door 4 ... totdat het getal nog maar uit 1 decimaal cijfer bestaat (d.w.z., kleiner wordt dan 10).

```
d ← 2
while (n ≥ 10)
{
    n ← n/d
    d ← d + 1
}
```

We tonen algoritmes vaak in een kader met wat extra uitleg. Na **`Input`**, definiëren we alle parameters (variabelen) die gegeven zijn bij het begin van het algoritme. Na **`Output`**, definiëren we de staat van bepaalde variabelen nadat het algoritme is uitgevoerd, en eventueel de waarde die wordt teruggegeven. Een waarde teruggeven doe je met de instructie **`return`**. Zodra **`return`** wordt uitgevoerd, stopt het algoritme en wordt de opgegeven waarde teruggegeven.

Dit voorbeeld toont hoe je de som van alle elementen van een array kan berekenen.

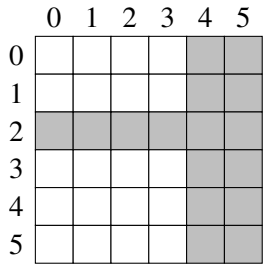
```
Input  : arr[ ], een array van n getallen.
         n, het aantal elementen van de array.
Output : sum, de som van alle getallen in de array.

sum ← 0
for (i ← 0 to n − 1 step 1)
{
    sum ← sum + arr[i]
}
return sum
```

Opmerking: in dit laatste voorbeeld wordt de variabele i enkel gebruikt om de tel bij te houden van de **`for`**-lus. Er is dus geen uitleg voor nodig bij **`Input`** of **`Output`**, en de waarde ervan wordt niet teruggegeven.

Vraag 1 – Tests

Je moet de vakjes van een tabel inkleuren door een logische test over de lijn- en kolomnummers te evalueren. De variabele `i` bevat een lijnnummer en de variabele `j` een kolomnummer. Als de test waar is voor de waardes `i` en `j`, dan moet je het vakje op de kruising tussen lijn `i` en kolom `j` inkleuren.

Hier zie je een voorbeeld met de voorwaarde `(i==2) or (j>3)`
De test slaagt als het lijnnummer gelijk is aan 2, of als het kolomnummer groter is dan 3.
De lijnnummers zijn links van de tabel aangegeven en de kolomnummers bovenaan.
De vakjes moet dus als volgt ingekleurd worden:



De logische tests gebruiken de vergelijkingen van deze voorbeelden.

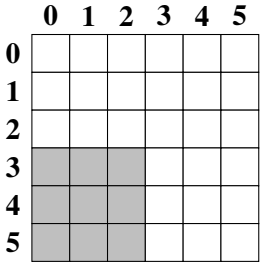
`i==1`	Waar als `i` gelijk is aan 1.
`j>3`	Waar als `j` groter is dan 3.
`j>=4`	Waar als `j` groter of gelijk is aan 4.
`i+j<5`	Waar als `i+j` kleiner is dan 5.
`i<=j+2`	Waar als `i` kleiner of gelijk is aan `j+2`.
`(i==3) or (j==2)`	Waar als minstens een van de 2 voorwaarden waar is.
`(i<2) and (j>3)`	Waar als beide voorwaarden waar zijn.

Sommige vragen gebruiken ook de volgende begrippen:

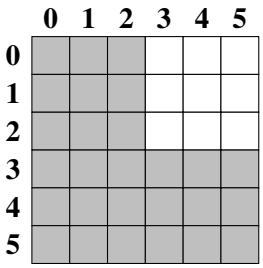
`max(14,18)`	De maximum functie. Geeft de grootste waarde terug, 18 dus.
`min(i,j)`	De minimum functie. Geeft de kleinste waarde terug.

De oplossingen staan hieronder afgebeeld.

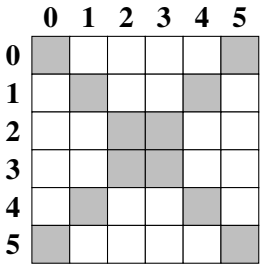
Q1(a) /2 **Kleur de vakjes in voor de voorwaarde `(i>=3) and (j<3)`**



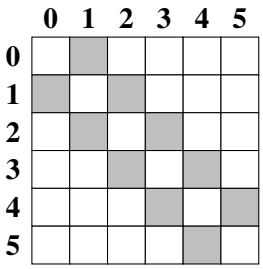
Q1(b) /2	**Kleur de vakjes in voor de voorwaarde `(i>=3) or (j<3)`**



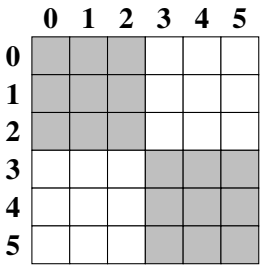
Q1(c) /4	**Kleur de vakjes in voor de voorwaarde `(i==j) or (i+j==5)`**



Q1(d) /4	**Kleur de vakjes in voor de voorwaarde `(i==j+1) or (i==j-1)`**



Q1(e) /4	**Kleur de vakjes in voor de voorwaarde `min(i,j)>=3 or max(i,j)<=2`**

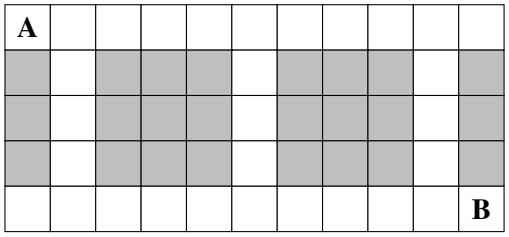


Vraag 2 – Wegen

Hoeveel wegen?

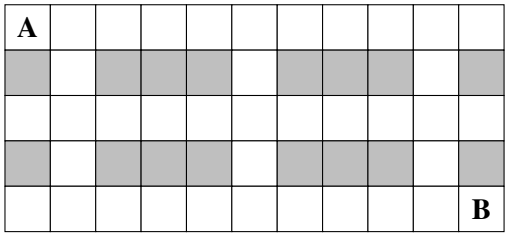
De tekeningen hieronder tonen zalen met vierkante tegels (witte vakjes) en obstakels (grijze vakjes). Een robot vertrekt van **A** links bovenaan en moet zich verplaatsen naar **B** rechts onderaan. Bij elke stap kan die zich enkel voortbewegen naar onder of naar rechts vanaf het vakje, nooit naar links or naar boven. De robot kan enkel op witte vakjes staan, en kan nooit over een grijs vakje bewegen. Voor elk geval, hoeveel verschillende wegen kan de robot gebruiken om van **A** naar **B** te gaan?

Q2(a) /1 **Hoeveel verschillende wegen kan de robot gebruiken om van A naar B te gaan?**



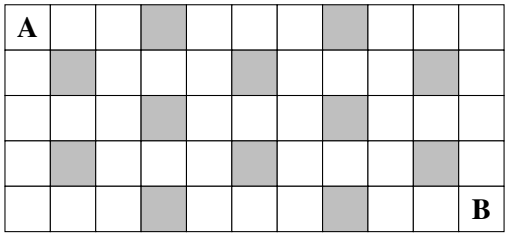
Oplossing: 3

Q2(b) /1 **Hoeveel verschillende wegen kan de robot gebruiken om van A naar B te gaan?**



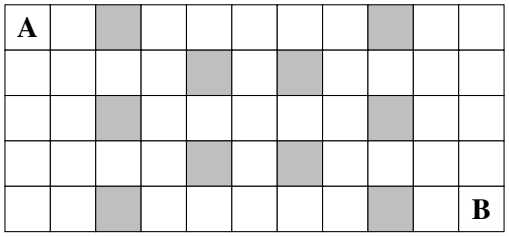
Oplossing: 6

Q2(c) /2 **Hoeveel verschillende wegen kan de robot gebruiken om van A naar B te gaan?**



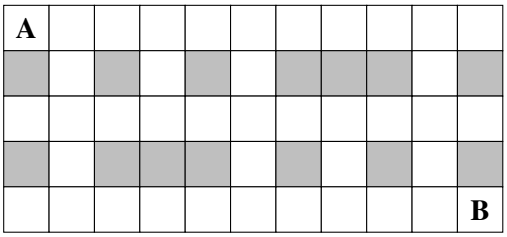
Oplossing: 1

Q2(d) /2 **Hoeveel verschillende wegen kan de robot gebruiken om van A naar B te gaan?**



Oplossing: 4

Q2(e) /3	**Hoeveel verschillende wegen kan de robot gebruiken om van A naar B te gaan?**

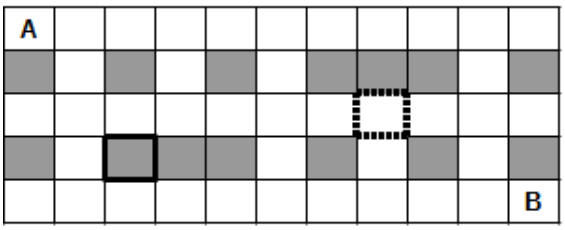


Oplossing: 11

Een programma om de wegen te tellen

Er bestaan verscheidene manieren om de wegen die van **A** naar **B** leiden te tellen. Een DP (dynamisch programmeren)-algoritme laat je toe het voor elk geval te doen, en wordt onmisbaar voor de moeilijke gevallen.

De onvolledige pseudo-code van een DP-algoritme is iets verderop gegeven. De zaal wordt voorgesteld met een tabel `t[][]` die een **0** bevat voor elk wit vakje en een **1** voor elk grijs vakje.



Zaal

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	1	0	1	1	1	0	1
2	0	0	0	0	0	0	0	0	0	0	0
3	1	0	1	1	1	0	1	0	1	0	1
4	0	0	0	0	0	0	0	0	0	0	0

Overeenkomstige tabel `t[][]`

In dit voorbeeld bestaat de tabel `t[][]` uit 5 rijen en 11 kolommen. De rijnummers staan links van de tabel aangegeven en de kolomnummers bovenaan. We verwijzen naar een vakje met zijn rij- en kolomnummer. Het vakje met een vette lijn is grijs, dus het overeenkomstige element `t[3][2]` is gelijk aan **1**. Het vakje met een stippellijn is wit, dus het overeenkomstige element `t[2][7]` is gelijk aan **0**. De vakjes **A** en **B** zijn natuurlijk wit en voorgesteld door **0** in `t[][]`.

Het DP-algoritme berekent voor elk vakje hoeveel verschillende wegen van **A** naar dat vakje gaan. Het aantal wegen om **A** te bereiken is gelijk aan **1**, gezien je er start en het onmogelijk is er later terug te keren. Het aantal wegen is gelijk aan **0** voor de grijze vakjes (omdat je er nooit kan komen). Het aantal wegen om een wit vakje te bereiken kan berekend worden op basis van de wegen om zijn buren links en boven te bereiken (dat is de essentiële eigenschap waarop DP-programma's zich baseren). Pas op voor de vakjes van de eerste kolom (die geen buur links hebben) en de eerste rij (die geen bovenbuur hebben)!

Het aantal wegen wordt opgeslagen in een tabel `dp[][]` met hetzelfde aantal rijen en kolommen als de tabel `t[][]`. Bij het begin van het algoritme wordt de tabel `dp[][]` gevuld met **0**. Het programma vertrekt bij het vakje **A** en gaat verder in de leesrichting (rij per rij, van links naar rechts in elke rij). Voor de voorbeeldzaal, na uitvoering van het algoritme, heeft `dp[3][2]` de waarde **0** want geen enkele weg laat toe een grijs vakje te bereiken, en `dp[2][7]` bevat het aantal wegen van **A** naar het vakje met een stippellijn.

Het programma krijgt het aantal rijen (het zijn er niet altijd 5!) in de variabele `ni`, het aantal kolommen in de variabele `nj`, de tabel `t[][]` die de zaal voorstelt, en de tabel `dp[][]` gevuld met **0**
Het programma moet het aantal verschillende wegen van **A** naar **B** teruggeven.

Q2(f) /6	**Vul de ____ in voor het DP-programma.** **Score tussen 0 en 6. Je verliest 1 punt per fout of ontbrekend antwoord.**
Oplossing : De oplossingen zijn weergegeven met een grijze achtergrond.	

```
dp[0][0] ← 1
for (i ← 0 to ni-1 step 1) {
   for (j ← 0 to nj-1 step 1) {

      if (t[i][j] = 1) { dp[i][j] ← 0 }
      else {
         if (i ≠ 0) { dp[i][j] ← dp[i-1][j] }

         if (j ≠ 0) { dp[i][j] ← dp[i][j] + dp[i][j-1] }
      }
   }
}
return dp[ni-1][nj-1]
```

Onbereikbare vakjes

Sommige witten vakjes zijn onbereikbaar. Dat wil zeggen dat de robot er niet kan geraken vanaf **A** omwille van obstakels (grijze vakjes). We veronderstellen dat het DP-programma uitgevoerd is en we dus beschikken over het aantal rijen in de variabele `ni`, het aantal kolommen in de variabele `nj`, de correcte tabel `t[][]` voor de zaal, en de tabel `dp[][]` ingevuld door het DP-programma.
Vervolledig het programma NoPath hieronder, dat het aantal onbereikbare witte vakjes moet teruggeven

Q2(g) /5	**Vul de ____ in voor het programma NoPath, dat het aantal witte onbereikbare vakjes telt.** **Score tussen 0 en 5. Je verliest 1 punt per fout of ontbrekend antwoord.**
Oplossing : De oplossingen zijn weergegeven met een grijze achtergrond.	

```
NoPath ← 0
for (i ← 0 to ni-1 step 1) {
   for (j ← 0 to nj-1 step 1) {

      if (dp[i][j]=0 and t[i][j]=0) { NoPath ← NoPath + 1 }
   }
}
return NoPath
```

Tussenliggende vakjes

We willen nu het aantal wegen tellen die door een tussenliggend vakje **X** gaan. De robot vertrekt steeds vanaf **A** links bovenaan en moet zich verplaatsen naar **B** rechts onderaan, maar moet onderweg door **X** gaan.

Zaal 1: We weten dat er 12 verschillende wegen zijn tussen **A** en **X** en 7 verschillende wegen tussen **X** en **B**.

Q2(h) /1	**In zaal 1, hoeveel wegen tussen A en B gaan er door X?**
Oplossing : $12 \cdot 7 = 84$	

Voor elk van de volgende vragen zijn er 2 tussenliggende vakjes **X** en **Y**.
We kennen telkens een onvolledig plan van de zaal waar **X** en **Y** op staan, en we kennen
N(A,X) het aantal verschillende wegen tussen **A** en **X**, **N(X,B)** het aantal verschillende wegen tussen **X** en **B**,
N(A,Y) het aantal verschillende wegen tussen **A** en **Y**, **N(Y,B)** het aantal verschillende wegen tussen **Y** en **B**.

Zaal 2: We weten N(A,X)=5, N(X,B)=12, N(A,Y)=8, N(Y,B)=10 en hebben het onvolledige plan

X				
				Y

.

Q2(i) /2	**In zaal 2, hoeveel wegen van A naar B gaan door X of door Y?**
Oplossing : $(5 \cdot 12) + (8 \cdot 10) = 140$	

Zaal 3: We weten N(A,X)=6, N(X,B)=16, N(A,Y)=10, N(Y,B)=8 en hebben het onvolledige plan

X		Y

.

Q2(j) /1	**In zaal 3, hoeveel wegen van A naar B gaan door beide vakjes X en Y?**
Oplossing : $6 \cdot 8 = 48$	

Q2(k) /2	**In zaal 3, hoeven wegen van A naar B gaan door X of Y (of beide)?**
Oplossing : $(6 \cdot 16) + (10 \cdot 8) - (6 \cdot 8) = 128$	

Vraag 3 – Lussen

Beschouw het programma **LoopA** hieronder. Het bestaat uit 3 geneste **`for`**-lussen en een **`if`**-opdracht met 3 voorwaarden die waar moeten zijn opdat een afdruk-opdracht uitgevoerd zou worden.

```
for (i ← 1 to 100 step 1) {
   for (j ← 1 to 100 step 1) {
      for (k ← 1 to 100 step 1) {
         if (i<j and j<k and i+j+k=100) {
            print(i,j,k)
         }
      }
   }
}
```

LoopA drukt alle verzamelingen van drie gehele getallen groter dan nul af waarvan de som gelijk is aan 100.
De 3 getallen van elke verzameling worden in oplopende volgorde afgedrukt.
Zo drukt **LoopA** het drietal `(20,30,50)` af maar niet het drietal `(50,20,30)`.

Q3(a) /1	**Welk drietal wordt eerst afgedrukt?**
Oplossing : $(1, 2, 97)$	

Q3(b) /1	**Wat is het tiende afgedrukte drietal?**
Oplossing : $(1, 11, 88)$	

Q3(c) /1	**Wat is het laatste afgedrukte drietal?**
Oplossing : $(32, 33, 35)$	

Q3(d) /1	**Hoeveel keer wordt de voorwaarde van de `if`-opdracht in LoopA geëvalueerd?**
Oplossing : $100 \cdot 100 \cdot 100 = 1000000$ keer	

Het programma **LoopB** drukt precies hetzelfde af als **LoopA**.
Maar het gebruikt een eenvoudigere voorwaarde in de **`if`**-opdracht: de voorwaarden `i<j` en `j<k` zijn niet meer nodig.

Q3(e) /3	**Vul de ____ in LoopB in zodat het hetzelfde als LoopA afdrukt.** **Score tussen 0 en 3. Je verliest 1 punt per fout of ontbrekend antwoord.**
Oplossing : De oplossingen worden hieronder op een grijze achtergrond weergegeven.	

```
for (i ← 1 to 100 step 1) {
   for (j ← i+1 to 100 step 1) {
      for (k ← j+1 to 100 step 1) {
         if (i+j+k=100) {
            print(i,j,k)
         }
      }
   }
}
```

Q3(f) /1	**Wordt de voorwaarde van de `if`-opdracht in LoopB vaker, minder vaak of even vaak geëvalueerd als in LoopA?** ☐ **Vaker** ☒ **Minder vaak** ☐ **Even vaak** **De voorwaarde wordt in LoopB 161700 keer geëvalueerd.**

Q3(g) /1	**Wordt de `print`-opdracht in LoopB vaker, minder vaak, of even vaak uitgevoerd als in LoopA?** ☐ **Vaker** ☐ **Minder vaak** ☒ **Even vaak** **De 2 programma's drukken precies hetzelfde af!**

Het programma **LoopC** drukt precies hetzelfde af als **LoopA** en **LoopB**.
De voorwaarde van de **`if`**-opdracht is opnieuw veranderd en er zijn nog maar 2 **`for`**-lussen.

Q3(h) /3	**Vul de ____ in LoopC in zodat het hetzelfde afdrukt als LoopA en LoopB.** **Score tussen 0 en 3. Je verliest 1 punt per fout of ontbrekend antwoord.**
Oplossing : De oplossingen worden hieronder op een grijze achtergrond weergegeven.	

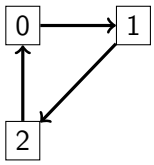
```
for (i ← 1 to 100 step 1) {
   for (j ← i+1 to 100 step 1) {
      k = 100-i-j
      if (j<k) {
         print(i,j,k)
      }
   }
}
```

Q3(i) /1	**Wordt de voorwaarde van de `if`-opdracht in LoopC vaker, minder vaak, of even vaak geëvalueerd als in LoopB?** ☐ **Vaker** ☒ **Minder vaak** ☐ **Even vaak** **De voorwaarde wordt 4950 keer geëvalueerd in LoopC.**

Vraag 4 – Teleportaties

Professor Spock heeft verschillende *teleportatienetwerken* opgesteld. Zo'n netwerk bestaat uit n teleportatiecabines, genummerd van 0 tot $n-1$. Elke cabine kan je teleporteren naar exact één **andere** cabine (het gaat om prototypes, elke cabine kan voorlopig nog maar naar één vaste bestemming teleporteren). Om veiligheidscontroles uit te voeren, wil Spock nagaan dat herhaaldelijk gebruik van de teleportators niet gevaarlijk is. Kan je hem helpen bij deze taak?

Het **eerste netwerk** dat de professor opgesteld heeft bestaat slechts uit 3 cabines.



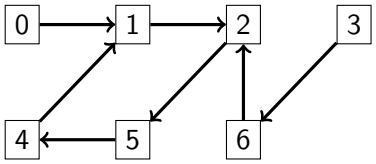
De professor test het netwerk door een object verschillende keren te teleporteren, vertrekkend vanaf cabine 0. Het object komt aan in cabine 1 na een teleportatie, in cabine 2 na twee teleportaties, en komt terug in cabine 0 na drie teleportaties, en zo verder.

Q4(a) /1	**Vertrekkend vanaf cabine 0, in welke cabine komt het object aan na 10 teleportaties?**
Oplossing : 1	

Q4(b) /2	**Vertrekkend vanaf cabine 0, in welke cabine komt het object aan na 50 teleportaties?**
Oplossing : 2	

Q4(c) /2	**Vertrekkend vanaf cabine 0, in welke cabine komt het object aan na 1000 teleportaties?**
Oplossing : 1	

Tevreden met de tests, bekijken we nu het **tweede netwerk**, bestaande uit 7 cabines.

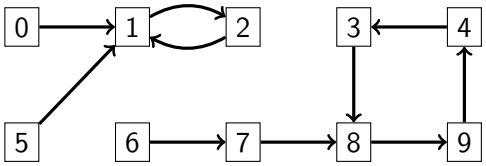


Q4(d) /2	**Vertrekkend vanaf cabine 4, in welke cabine kome je aan na 50 teleportaties?**
Oplossing : 2	

Q4(e) /2	**Vertrekkend vanaf cabine 3, in welke cabine kome je aan na 1000 teleportaties?**
Oplossing : 4	

Je merkt op dat door een groot aantal teleportaties uit te voeren, we telkens in dezelfde cabines uitkomen. In het vorige netwerk, vertrekkend vanaf cabine 0, komen we opeenvolgend in de cabines 0, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4,$\cdots$. We zeggen dat een cabine die na een zeker aantal teleportaties terug bij zichzelf kan uitkomen *deel uitmaakt van een cyclus*. Een *cyclus* is een lijst van cabines waar elke cabine naar de volgende gaat, en de laatste terug naar de eerste gaat. De lengte van een cyclus is het aantal cabines van de cyclus. Het **tweede netwerk** bevat een cyclus van lengte 4, gevormd door de cabines 1, 2, 5, 4.

Hieronder vind je het **derde netwerk**, bestaande uit 10 cabines en met 2 cycli, één van lengte 2, de andere van lengte 4.



De professor heeft bewezen dat in al zijn *teleportatienetwerken*, waar elke cabine naar exact één andere cabine teleporteert, je altijd in een cyclus uitkomt na een bepaald aantal teleportaties, onafhankelijk van de cabine waar je vertrekt.

Q4(f) /3	**In een netwerk met `n` cabines, vertrekkend van eender welke cabine, na hoeveel teleportaties ben je zeker dat je in een cyclus bent?**
Oplossing : `n-2`	

Het is some nuttig om te kunnen bepalen in welke cyclus je zal uitkomen vanaf een gegeven cabine.
Een teleportatienetwerk kan voorgesteld worden door een tabel `T[]` die aangeeft, voor elke cabine, welke bestemming een teleportatie heeft. Zodus is de tabel die overeenkomt met het **derde netwerk** met `n = 10` cabines gegeven als `T=[1, 2, 1, 8, 3, 1, 7, 8, 9, 4]`.
De cyclus van lengte 2 kan afgelezen worden uit de verbanden `T[1]=2` en `T[2]=1`.
De cyclus van lengte 4 kan afgelezen worden uit de verbanden `T[3]=8`, `T[8]=9`, `T[9]=4` en `T[4]=3`.
De cabines 0, 1, 2 en 5 komen uit in de cyclus van lengte 2.
De andere cabines komen uit in de cyclus van lengte 4 (na 2 teleportaties bij vertrek vanaf cabine 6).

Je moet hier **Programma A** vervolledigen. Dit krijgt als invoer een tabel `T[]` die het te bestuderen netwerk voorstelt, het aantal cabines `n` en het nummer van de vertrekcabine `c`.
Als uitvoer geeft het een tabel van logische waardes (booleans) `cyc[]` van lengte `n`, zodat `cyc[i]` **`true`** is als cabine `i` deel uitmaakt van de cyclus waarin je uitkomt bij vertrek vanaf cabine `c`, en anders **`false`**.
Het programma gebruikt een interne tabel `check[]` met `n` logische waarden.
Je kan de logische operator **`not`** gebruiken (**`not true`** is gelijk aan **`false`** en **`not false`** is **`true`**).

Q4(g) /5	**Vul de ____ in voor Programma A.** **Score tussen 0 en 5. Je verliest 1 punt per fout of ontbrekend antwoord.**
Oplossing : De oplossingen worden op een grijze achtergrond getoond.	

```
Input : n, T[], c       Output : cyc[]
for (i ← 0 to n-1 step 1) {
   check[i] ← false
   cyc[i] ← false
}
pos ← c
while (not check[pos]) {
    check[pos] ← true
    pos ← T[pos]
}
while (not cyc[pos]) {
    cyc[pos] ← true
    pos ← T[pos]
}
return cyc[]
```

Het is evenzeer belangrijk om te weten welke cabines de cycli in het netwerk vormen. **Programma B** moet de tabel met booleans `cy[]` invullen zodat `cy[i]` **`true`** is als cabine `i` deel uitmaakt van een cyclus en anders **`false`**. Het programma gebruikt een interne tabel `pos[]` van `n` gehele getallen.

Q4(h) /4	**Vul de ____ in voor Programma B.** **Score tussen 0 en 4. Je verliest 2 punten per fout of ontbrekend antwoord.**
Oplossing : De oplossingen worden op een grijze achtergrond getoond.	

```
Input : n, T[]    Output : cy[]
for (i ← 0 to n-1 step 1) {
    pos[i] ← i
    cy[i] ← false
}
for(j ← 0 to n-1 step 1) {
    for (i ← 0 to n-1 step 1) {
        pos[i] ← T[pos[i]]
    }
}
for (i ← 0 to n-1 step 1) {
    cy[pos[i]] ← true
}
return cy[]
```

Ook de lengtes van de cycli zijn belangrijk. **Programma C** moet de tabel `lency[]` invullen zodat `lency[i]` 0 is als cabine `i` geen deel uitmaakt van een cyclus en anders de lengte van die cyclus. **Programma C** gebruikt de tabel `cy[]` die eerder door **Programma B** berekend is.

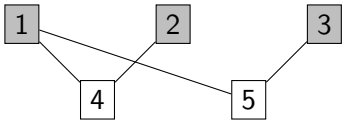
Q4(i) /5	**Vul de ____ in voor Programma C.** **Score tussen 0 en 5. Je verliest 1 punt per fout of ontbrekend antwoord.**
Oplossing : De oplossingen worden op een grijze achtergrond getoond.	

```
Input : n, T[], cy[]    Output : lency[]
for (i ← 0 to n-1 step 1) {
    if(cy[i]) {
        pos ← T[i]
        len ← 1
        while(pos ≠ i) {
            pos ← T[pos]
            len ← len+1
        }
        lency[i] ← len
    }
   else { lency[i] ← 0 }
}
return lency[]
```

Vraag 5 – Zwerkbal

Professor Perkamentus zit met een probleempje: hij moet een Zwerkbalmatch (de favoriete sport van tovenaars) organiseren, en daarvoor moet hij de leerlingen in twee groepen onderverdelen om teams te kunnen vormen. Het probleem is dat vele leerlingen niet goed met elkaar overweg kunnen. Zo wil Ron bijvoorbeeld niet meer met Draco spreken sinds die slobberige slakken in zijn bed heeft doen verschijnen.

Professor Perkamentus heeft besloten om alle relaties tussen leerlingen in een graaf op te nemen zoals die hieronder. Die bestaat uit "knopen" met de namen van leerling (hier vervangen door getallen voor de leesbaarheid) en verbindingen (genaamd "zijden") tussen deze knopen die aangeven wanneer twee leerlingen niet met elkaar overweg kunnen.



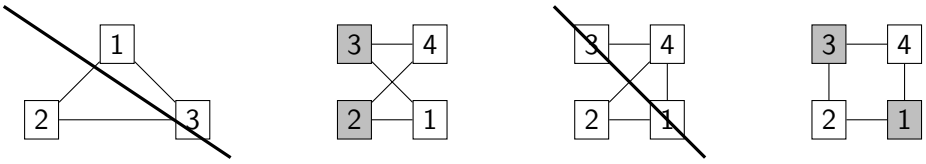
Eerste voorbeeld: een graaf met 5 knopen en 4 zijden.

Je kan zo dus bijvoorbeeld zien dat 1 en 4 niet met elkaar overweg kunnen en dus niet in dezelfde groep kunnen zitten. Daarentegen kan 1 het goed vinden met 2 en 3.

Perkamentus gaat de leerlingen nu onderverdelen in twee groepen. Deze groepen moeten niet noodzakelijk dezelfde grootte hebben, maar **twee leerlingen die het niet met elkaar kunnen vinden kunnen niet in dezelfde groep zitten**.

In het bovenstaand voorbeeld is het gemakkelijk te zien dat we twee groepen hebben: $\{1, 2, 3\}$ en $\{4, 5\}$. Om dit beter zichtbaar te maken heeft Perkamentus de leerlingen van de eerste groep grijs gekleurd en de anderen wit gelaten.

Hier krijg je andere grafen. Voor elke graaf willen we weten of het mogelijk is zijn knopen in twee groepen te splitsen zoals uitgelegd. Als het mogelijk is, kleur dan de knopen van een van de twee groepen. Als het onmogelijk is, doorkruis dan de graaf.
Je kan hier in het klad werken. Vergeet niet om **je antwoorden over te schrijven naar het antwoordblad**.



(a) (b) (c) (d)

Q5(a) /2	**Kleur de knopen van een groep in graaf (a) of doorkruis de graaf als dat onmogelijk is.**
Oplossing : Zie boven	

Q5(b) /2	**Kleur de knopen van een groep in graaf (b) of doorkruis de graaf als dat onmogelijk is.**
Oplossing : Zie boven	

Q5(c) /2	**Kleur de knopen van een groep in graaf (c) of doorkruis de graaf als dat onmogelijk is.**
Oplossing : Zie boven	

Q5(d) /2	**Kleur de knopen van een groep in graaf (d) of doorkruis de graaf als dat onmogelijk is.**
Oplossing : Zie boven	

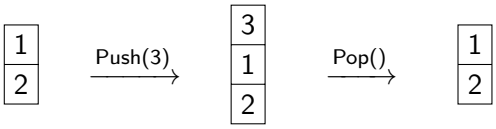
Nu wil professor Perkamentus een algoritme vinden om dit opsplitsen in groepen automatisch te doen, of anders gezegd, om zijn grafen te kleuren.

Hiertoe raadpleegt hij zijn lievelingsgrimoire: “De Algorithmica”, geschreven door magister Cormenius (en diens collega’s) in de dertiende eeuw. Helaas is er onuitwisbare inkt gemorst op de relevante pagina, en ontbreken er stukken van het algoritme.

Perkamentus beseft dat hij nood zal hebben aan een datastructuur onder de naam “stapel”. Een stapel laat toe om knopen bij te houden terwijl hij de graaf doorkruist, alsof het om een stapel borden zou gaan, met een knoop onderaan de stapel, een andere daarboven, *etc* totdat de top van de stapel bereikt is. We kunnen de volgende operaties uitvoeren op een stapel (zie de voorbeelden iets verderop):

1. Push(x) voegt een knoop toe aan de top van de stapel;
2. Pop() geeft de knoop bovenaan de stapel terug en neemt die weg van de stapel;
3. IsEmpty() geeft **`true`** terug als de stapel leeg is, en anders **`false`**.

Hier zie je een voorbeeld waar we beginnen met een stapel met twee elementen, we 3 toevoegen en terug verwijderen (de laatste Pop()-operatie geeft dus 3 terug).



Q5(e) /1	**Veronderstel dat we met een lege stapel beginnen. Geef een opeenvolging van** Push()**-operaties die deze stapel geeft:** 
Oplossing : Push(87), Push(11), Push(42)	

Q5(f) /1	**Veronderstel dat we deze stapel hebben:**  **en dat we de volgende operaties uitvoeren:** Pop(), Pop(), Push(24), Push(37), Push(71), Pop(), Pop(). **Welke stapel is dan het resultaat?**

Solution: 

Q5(g) /1	**Welke waarde geeft de laatste** Pop() **van de vorige vraag terug?**
Oplossing : de laatste Pop() geeft 37 terug.	

Professor Perkamentus kan nu een eerste *graafdoorloopalgoritme* bestuderen.
De te doorlopen graaf bestaat uit `n` knopen en wordt beschreven door een tabel met logische waarden (booleans) `Edge[][]`. `Edge[x][y]` en `Edge[y][x]` zijn **`true`** als er een zijde tussen de knopen `x` en `y` is, en anders **`false`**.
Voor de graaf van het eerste voorbeeld, zijn `Edge[1][4]` en `Edge[4][1]`, `Edge[1][5]` en `Edge[5][1]`, `Edge[2][4]` en `Edge[4][2]`, `Edge[3][5]` en `Edge[5][3]` **`true`**. Alle andere waarden in de tabel `Edge[][]` zijn **`false`**.

Het algoritme krijgt het aantal knopen `n`, de tabel `Edge[][]` en een beginknoop `s` vanaf waar de doorloop begint.
Het gebruikt een tabel `color[]` die toelaat de knopen te *kleuren* tijdens het doorlopen.
Aan het begin van het programma initialiseren we `color[i]` tot -1 voor alle knopen `i`.
Om een knoop `i` te kleuren zetten we `color[i]` naar 0, en deze waarde verandert later niet meer.
Het algoritme gebruikt ook een stapel die initieel leeg is om de knopen te onthouden die bezocht moeten worden.
Merk op dat, in tegenstelling tot wat gebruikelijk is in de informatica, de indices op 1 beginnen in dit programma.

```
Input : n, Edge[][], s

for (i ← 1 to n step 1){
   color[i] ← -1
}

Push(s)
color[s] ← 0

while(not IsEmpty()) {
   x ← Pop()
   for (y ← 1 to n step 1){
      if (Edge[x][y] and color[y] = -1){
         Push(y)
         color[y] ← 0
      }
   }
}
```

Q5(h) /3	**In welke volgorde kleurt het algoritme de knopen van de eerste voorbeeldgraaf (met 5 knopen en 4 zijden) als de startknoop `s=4` is?**
Oplossing : 4, 1, 2, 5, 3	

Professor Perkamentus probeert nu eindelijk het algoritme te vinden waarmee hij de knopen van de graaf in twee groepen kan verdelen.
Hij begrijpt dat hij twee verschillende kleuren moet gebruiken, dat worden 0 en 1, overeenkomstig met de twee groepen.
Hij snapt ook dat als een knoop de kleur 0 krijgt, alle knopen die daarmee verbonden zijn met een zijde de andere kleur 1 moeten krijgen, en vice versa.
Om hem bij deze taak te helpen gebruikt professor Perkamentus een functie `invert` zodanig dat `invert(0)=1` en `invert(1)=0`. Kan hij het algoritme hieronder vervolledigen, met deze ideeën?
Het moet een tabel `color[]` teruggeven met de kleuren van elke knoop als de verdeling mogelijk is, en anders **`false`** teruggeven als het onmogelijk is.

Q5(i) /6	**Vul de ____ in voor het algoritme.** **Score tussen 0 en 6. Je verliest 2 punten per fout of ontbrekend antwoord.**
Oplossing : De oplossingen zijn weergegeven met een grijze achtergrond.	

```
Input : n, Edge[][], s          Output: color[] or false
for (i ← 1 to n step 1){
   color[i] ← -1
}
Push(s)
color[s] ← 0

while(not IsEmpty()) {
   x ← Pop()
   for (y ← 1 to n step 1){
       if (Edge[x][y]){
          if (color[y] = color[x] ) {
             return false
          } else if (color[y] = -1) {
             Push(y)
             color[y] = invert(color[x])
          }
       }
   }
}
return color[]
```