be-OI 2023 **Finale - BELOFTE** Zaterdag 18 maart 2023	**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 daar voor voorziene antwoordbladen. Schrijf **duidelijk leesbaar** met blauwe of zwarte **bic of pen**.
4. Gebruik een potlood en een gom wanneer 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 ← 2006
leeftijd ← 2023 − 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.

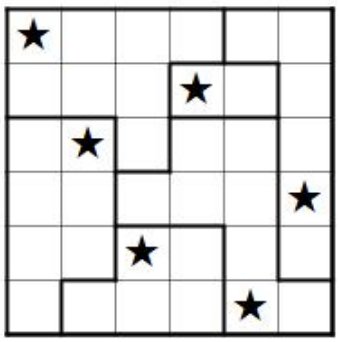
Vraag 1 – Star Battle

Star Battle is een logisch spel waarin je **n** sterren moet plaatsen in een raster van **n** rijen en **n** kolommen verdeeld in **n** zones gescheiden door muren die worden weergegeven door dikke lijnen.

De volgende regels moeten in acht worden genomen.

- **Regel 1:** elke rij moet precies 1 ster bevatten.
- **Regel 2:** elke kolom moet precies 1 ster bevatten.
- **Regel 3:** elke zone moet precies 1 ster bevatten.
- **Regel 4:** vakjes rondom een ster kunnen geen andere ster bevatten.

Voorbeeld: het rooster aan de rechterkant is ingevuld volgens de 4 regels.



Hier is een voorbeeld met **n=6** waar slechts één ster is geplaatst.
De rijnummers (van 0 t/m 5) staan links van het raster.
De kolomnummers (van 0 t/m 5) staan boven het raster.
De ster werd geplaatst in het vakje op de kruising van rij 2 en kolom 1.
Vakjes waarin geen sterren meer kunnen worden geplaatst, zijn gemarkeerd met een "x":
• de vakjes van rij 2 (regel 1),
• de vakjes van kolom 1 (regel 2),
• de (grijze) vakjes van de zone met de ster (regel 3),
• de 8 vakjes rond de ster (regel 4).

	0	1	2	3	4	5
0		x				
1	x	x	x			
2	x	★	x	x	x	x
3	x	x	x			
4	x	x				
5	x	x				

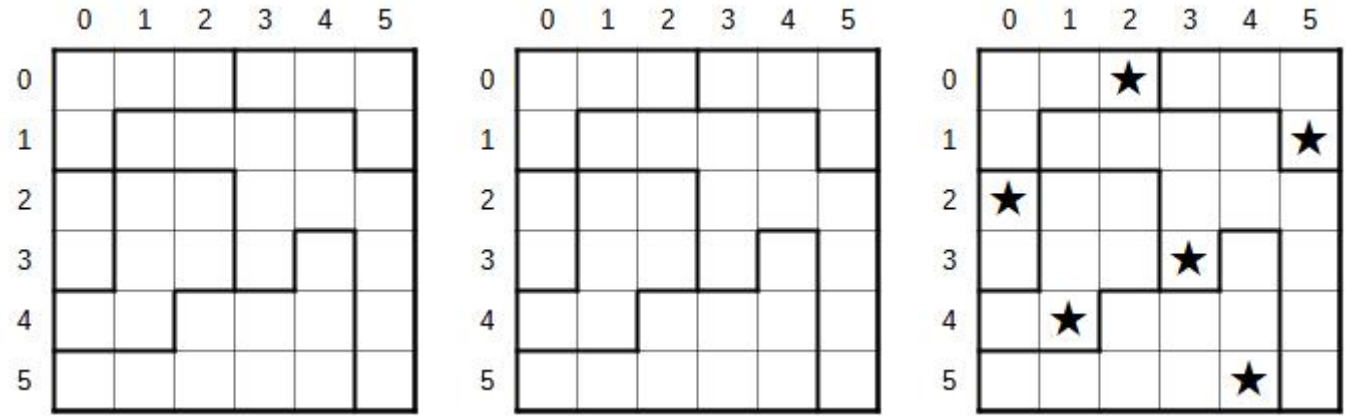
We nummeren de zones van **0** tot **n-1** door ons van vakje tot vakje te verplaatsen “in de leesrichting” vanaf de linkerbovenhoek van het raster.
We schrijven **0** in alle vakjes van de eerste zone, we schrijven **1** in alle vakjes van de volgende zone die nog geen nummers bevat, we gaan door met de volgende getallen totdat we **n-1** invoeren in de laatste zone (zone **5** in het voorbeeld).
De eerste vakjes van elke zone “in de leesrichting” zijn vetgedrukt.
In de programma’s wordt het rooster voorgesteld door een tabel **t[ ][ ]** met zonenummers.
• **t[2][1]=3**, omdat het vakje op rij 2 en kolom 1 (in het grijs) zich in zone 3 bevindt.
• **t[4][5]=1**, omdat het vakje op rij 4 en kolom 5 (ook in het grijs) zich in zone 1 bevindt.

	0	1	2	3	4	5
0	**0**	0	0	0	**1**	1
1	0	0	0	**2**	2	1
2	**3**	3	0	**4**	4	1
3	3	3	4	4	4	1
4	3	3	**5**	5	4	1
5	3	5	5	5	4	4

De speler stelt een oplossing voor met behulp van een array **col[ ]**.
Voor elk rijnummer **i** moet de speler het kolomnummer **col[i]** opgeven waar hij een ster in deze rij wil plaatsen.
In het voorbeeld rechts, **col[ ]=[2,3,0,4,2,5]**
(afgekorte notatie voor **col[0]=2**, **col[1]=3**, **col[2]=0**, **col[3]=4**, **col[4]=2** en **col[5]=5**).
Dit voorstel is erg slecht omdat alleen regel 1 wordt gerespecteerd.
• Regel 2 wordt overtreden omdat er meerdere sterren in kolom 2 staan.
• Regel 3 wordt overtreden omdat er meerdere sterren in zone 4 staan.
• Regel 4 wordt overtreden door de sterren op rij 0 en 1.

	0	1	2	3	4	5
0	0	0	★	0	1	1
1	0	0	0	★	2	1
2	★	3	0	4	4	1
3	3	3	4	4	★	1
4	3	3	★	5	4	1
5	3	5	5	5	4	★

De vragen op deze pagina gebruiken het **Star Battle**-raster dat hieronder 3 keer wordt weergegeven.
(De oplossing wordt weergegeven in het rechterraster.)



Q1(a) /4	**Wat zijn de waarden van de volgende elementen van de tabel t[ ][ ] voor dit raster?** `t[1][5]=..., t[2][3]=..., t[3][0]=..., t[4][3]=...`
Oplossing: `t[1][5]`=1, `t[2][3]`=2, `t[3][0]`=3, `t[4][3]`=5	

Voor de volgende 4 vragen moet je de vakjes van de regels die **gerespecteerd** zijn **aanvinken** en de vakjes van de regels die *niet gerespecteerd* zijn *niet aanvinken*. Je scoort alleen punten als je het juiste antwoord geeft voor de 4 regels.

Q1(b) /2	**Vink de vakjes aan van de regels die gerespecteerd worden als col[ ]=[1, 5, 0, 3, 1, 4].**
	☒ regel 1 ☐ regel 2 ☒ regel 3 ☒ regel 4

Q1(c) /2	**Vink de vakjes aan van de regels die gerespecteerd worden als col[ ]=[1, 5, 3, 0, 2, 4].**
	☒ regel 1 ☒ regel 2 ☐ regel 3 ☒ regel 4

Q1(d) /2	**Vink de vakjes aan van de regels die gerespecteerd worden als col[ ]=[2, 3, 0, 4, 1, 5].**
	☒ regel 1 ☒ regel 2 ☐ regel 3 ☐ regel 4

Q1(e) /2	**Vink de vakjes aan van de regels die gerespecteerd worden als col[ ]=[2, 5, 3, 0, 1, 4].**
	☒ regel 1 ☒ regel 2 ☒ regel 3 ☐ regel 4

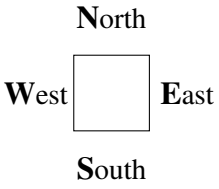
Vind de oplossing !

Q1(f) /2	**Plaats 6 sterren in het rooster volgens de 4 regels.**
Oplossing: De oplossing wordt weergegeven in het raster rechtsboven.	

Q1(g) /2	**Wat is de waarde van de array col[ ] voor jouw oplossing ?** `col[]=[..., ..., ..., ..., ..., ...]`
Oplossing: col[ ]=[2, 5, 0, 3, 1, 4]	

Je moet het onderstaande **Programma SB1** vervolledigen dat wordt gebruikt om een spelraster weer te geven op het scherm zoals op het eerste voorbeeld van de eerste pagina van deze uitleg.
Het programma ontvangt de grootte **n**, de tabel **t[ ][ ]** en de array **col[ ]**.

Het programma maakt gebruik van een robot die over het scherm kan bewegen en lijnen, muren en sterren kan tekenen. Het programma begint met de robot in de juiste plaats zodat de robot het vakje linksboven in het raster kan tekenen, wat overeenkomt met **t[0][0]**.
We kunnen de instructies `Go()`, `line()`, `wall()` en `star()` gebruiken, die hieronder worden uitgelegd.
`Go()`, `line()` en `wall()` hebben een richting W, E, N of S nodig.



- `Go(W,n)`, `Go(E,n)`, `Go(N,n)` et `Go(S,n)` dienen om **n** vakjes in de overeenkomstige richting te bewegen.
  Voorbeelden: `Go(E,1)` verplaatst de robot één vakje naar rechts, `Go(S,2)` verplaatst hem 2 vakjes naar omlaag.
- De instructie `wall()` wordt gebruikt om een muur te tekenen op een van de randen van het vakje waar de robot zich bevindt.
  `wall(W)` tekent een verticale muur aan de linkerkant en `wall(E)` een verticale muur aan de rechterkant.
  `wall(N)` tekent een horizontale muur bovenaan en `wall(S)` een horizontale muur onderaan het vakje.
  Het rooster en de verschillende gebieden moeten omgeven zijn door muren.
- `line()` werkt als `wall()` maar tekent een dunne lijn in plaats van een muur.
  Aan elke zijde van elk vakje moet een muur of een dunne lijn zijn.
- `star()` tekent een ster in het vakje waar de robot zich bevindt.

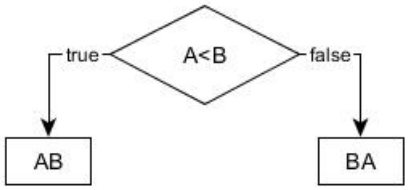
Als een test meerdere voorwaarden bevat die zijn gekoppeld door **`or`**, is de test **`true`** als ten minste één van de voorwaarden **`true`** is. Bovendien, als de eerste voorwaarde van een **`or true`** is, wordt de tweede voorwaarde niet eens geëvalueerd (dat is nutteloos).

Q1(h) /10	**Vervolledig de ____ in Programma SB1.** **Punten tussen 0 en 10. Je verliest 1 punt per foutief of ontbrekend antwoord.**
Oplossing: De oplossingen worden hieronder weergegeven op een grijze achtergrond.	

```
Input  : n, t[][], col[]
for (i ← 0 to n-1 step 1) {
   for (j ← 0 to n-1 step 1) {
      if (i=0 or t[i][j] ≠ t[i-1][j]) {wall(N)}
      else { line(N) }

      if (i=n-1) {wall(S)}

      if (j=0 or {t[i][j] ≠ t[i][j-1]) {wall(W)}
      else {line(W)}

      if (j=n-1) {wall(E)}

      if (j=col[i]) {star()}

      Go(E, 1)
   }
   Go(W, n)
   Go(S , 1)
}
```

Vraag 2 – Flowchart

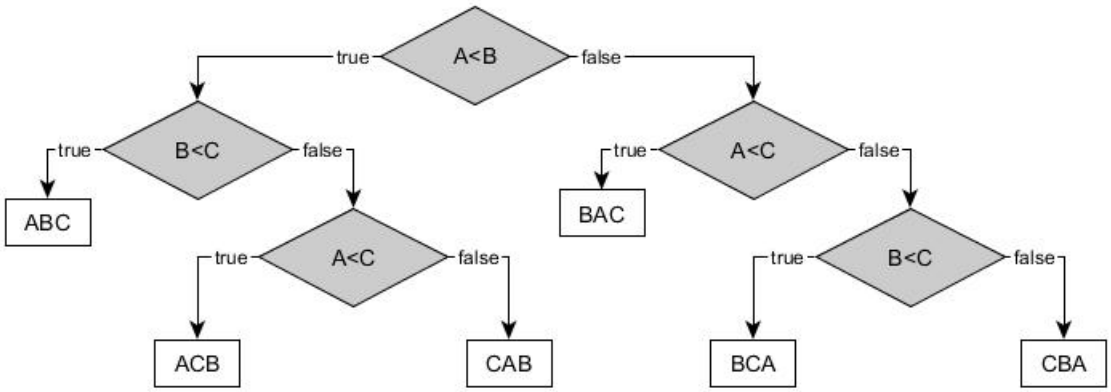
We kunnen een test voorstellen door een ruit waarin we de te verifiëren voorwaarde schrijven.
Afhankelijk van het resultaat volgen we de pijl **`true`** of de pijl **`false`**.
In onderstaand voorbeeld zijn `A` en `B` variabelen die worden vergeleken met elkaar.
Als `A<B`, moet het bericht "**AB**" worden getoond, anders moet "**BA**" getoond worden.
De bewerkingen voor het weergeven van berichten worden weergegeven door de rechthoeken.



Bijvoorbeeld, als `A=7` en `B=4`, volgt het bovenstaand algoritme de pijl **`false`** van de test en toont "**BA**".

Laten we verder gaan met een soortgelijk probleem met drie variabelen `A`, `B` en `C` die elk **een verschillende waarde** hebben.
We moeten een algoritme grafisch weergeven dat de 3 letters toont "**A**", "**B**" en "**C**" in de volgorde van de waarden van de variabelen, van klein naar groot. Bijvoorbeeld, als `A=11`, `B=16` en `C=7`, dan moet het algoritme "**CAB**" tonen, aangezien `C` de kleinste waarde heeft en `B` de grootste.
In elke ruit moet een test worden geplaatst. Je mag enkel de volgende 6 testen gebruiken: `A<B`, `B<A`, `A<C`, `C<A`, `B<C` en `C<B`.

Q2(a) /5	**Vervolledig onderstaande . . . . Je krijgt één punt per juist ingevulde zone**
Oplossing: De antwoorden worden getoond op een grijze achtergrond.	



Stel nu dat de variabelen **identieke waarden** kunnen hebben.

Q2(b) /3	**Welk bericht wordt weergegeven als de 3 variabelen gelijk zijn?**
Oplossing: CBA	

Q2(c) /3	**Welk bericht verschijnt nooit als 2 variabelen gelijk zijn?**
Oplossing: ABC	

Vraag 3 – Kruising van lijsten

Het is nodig om alle elementen te vinden die gemeenschappelijk voorkomen in 2 lijsten met n elementen $L1[\,]$ en $L2[\,]$.
Het is bekend dat de elementen van L1 allemaal verschillend zijn, evenals de elementen van L2.
We moeten de lijst $Li[\,]$ maken van alle elementen die zowel in $L1[\,]$ als in $L2[\,]$ staan.
Bijvoorbeeld, als $n = 8$, $L1[\,] = [1, 3, 6, 10, 15, 21, 28, 36]$ en $L2[\,] = [3, 7, 15, 19, 23, 29, 36, 41]$ dan $Li[\,] = [3, 15, 36]$.
Zoals gewoonlijk nummeren we de elementen vanaf 0.

Eerste idee : het programma P1.
Het gebruikt 2 geneste **`for`** lussen.
De eerste lus selecteert bij elke doorgang een nieuw element $x1$ uit $L1[\,]$.
De tweede lus, binnen de eerste, vergelijkt achtereenvolgens $x1$ met alle elementen van $L2[\,]$.
Als er een gelijkheid wordt gevonden, voegen we $x1$ toe aan de lijst $Li[\,]$.
P1 gebruikt `Li[]←[]` om een lege $Li[\,]$ lijst te maken.
Het gebruikt de methode `append()` om **een item toe te voegen aan het einde van een lijst**.
Bijvoorbeeld, als $Li[\,]$ leeg is en we voeren `Li[].append(3)` uit, dan $Li[\,] = [3]$.
Als we dan `Li[].append(15)` uitvoeren, vervolgens `Li[].append(36)`, dan $Li[\,] = [3, 15, 36]$.

Let op de regels (1) en (2) in **P1** gemarkeerd met een nummer in een grijze rechthoek.

```
Input : L1[], L2[], n        Output : Li[]
Li[] ← []
for i1 ← 0 to n-1 step 1 {
    x1 ← L1[i1]          // (1)
    for i2 ← 0 to n-1 step 1 {
        if (L2[i2]=x1)     // (2)
            {Li[].append(x1)}
    }
}
```

Q3(a) /1	**Als $n = 100$, hoe vaak wordt regel (1) dan uitgevoerd?**
Oplossing: 100	

Q3(b) /1	**Als $n = 100$, hoe vaak wordt regel (2) dan uitgevoerd?**
Oplossing: 10000	

Q3(c) /1	**In het algemeen, voor een gegeven n, hoe vaak zal regel (1) uitgevoerd worden?**
Oplossing: n	

Q3(d) /1	**In het algemeen, voor een gegeven n, hoe vaak zal regel (2) uitgevoerd worden?**
Oplossing: n^2	

In de volgende vragen testen we **P1** met lijsten van verschillende lengte.
We nemen aan dat de totale uitvoeringstijd alleen afhangt van het aantal keren dat de **regel (2)** wordt uitgevoerd (het is bij benadering, maar voldoende nauwkeurig voor ons).

Q3(e) /1	**Als de uitvoeringstijd 3 seconden is voor een bepaalde n, hoe lang, in *seconden*, duurt het voor 5 keer langere lijsten?**
Oplossing: $3 \cdot 5 \cdot 5 = 75$ seconden	

Q3(f) /1	**Als de uitvoeringstijd** 6 **seconden is voor een bepaalde** n**, hoe lang, in** *minuten*, **duurt het voor** 10 **keer langere lijsten?**
Oplossing: $6 \cdot 10 \cdot 10 = 600$ seconden = 10 minuten	

Q3(g) /1	**Als de uitvoeringstijd** 4 **seconden is voor een bepaalde** n**, hoe lang, in** *uren*, **duurt het voor** 60 **keer langere lijsten ?**
Oplossing: $4 \cdot 60 \cdot 60$ seconden = $4 \cdot 60$ minuten = 4 uren	

Tweede idee : het programma P2.
P1 werkt met alle lijsten, gesorteerd of niet, maar het is traag.
P2 is sneller, maar het werkt alleen als de lijsten in stijgende lijn gesorteerd zijn.

Hier is de lijst van **P2** zonder uitleg.
Let op de lijn (3) gemarkeerd met een nummer in een grijze rechthoek.

```
Input : L1[], L2[], n        Output : Li[]
Li[] ← []
i1 ← 0
i2 ← 0
while (i1 < n and i2 < n) {
    x1 ← L1[i1]        // (3)
    x2 ← L2[i2]
    if (x1 < x2) {i1 ← i1 + 1}
    else {
        if (x1 = x2) {Li[].append(x1)}
        i2 ← i2 + 1
    }
}
```

Q3(h) /1	**Als** $n = 10$**, wat is dan het MINimaal aantal keren dat regel (3) wordt uitgevoerd?**
Oplossing: 10	

Q3(i) /1	**Als** $n = 10$**, wat is dan het MAXimaal aantal keren dat regel (3) wordt uitgevoerd?**
Oplossing: 19	

Q3(j) /1	**Als** $n = 10$ **en** $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$**, geef dan een lijst** $L2[\,]$ **van verschillende gehele getallen** > 0 **zodanig dat regel (3) een MINimaal aantal keren wordt uitgevoerd.**
Oplossing: Elke lijst waarvan het eerste element bijvoorbeeld groter is dan 19, `L2[] = [20,21,22,23,24,25,26,27,28,29]`	

Q3(k) /1	**Als** $n = 10$ **en** $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$**, geef dan een lijst** $L2[\,]$ **van verschillende gehele getallen** > 0 **zodanig dat regel (3) een MAXimaal aantal keren uitgevoerd wordt.**
Oplossing: Er zijn veel oplossingen: `L2[]=[2,4,6,8,10,12,14,16,18,n]` met $n \geq 20$ of `L2[]=L1[]=[1,3,5,7,9,11,13,15,17,19]`	

Q3(l) /1	**Voor een gegeven** n**, wat is het MINimum aantal keren dat regel (3) uitgevoerd zal worden?**
Oplossing: n	

Q3(m) /1	**Voor een gegeven n, wat is het MAXimum aantal keren dat regel (3) uitgevoerd zal worden?**
Oplossing: $2n - 1$	

In de volgende vragen testen we **P2** met lijsten van verschillende lengte.
We nemen aan dat de totale uitvoeringstijd alleen afhangt van het aantal keren dat de **regel (3)** wordt uitgevoerd (het is bij benadering, maar voldoende nauwkeurig voor ons).

Q3(n) /1	**Als de MAXimale uitvoeringstijd 3 seconden is voor een bepaalde n, wat is dan de MAXimale tijd, in *seconden*, voor lijsten die 5 keer langer zijn?**
Oplossing: $3 \cdot 5 = 15$ seconden	

Q3(o) /1	**Als de MAXimale uitvoeringstijd 6 seconden is voor een bepaalde n, wat is dan de MAXimale tijd, in *minuten*, voor lijsten die 10 keer langer zijn?**
Oplossing: $6 \cdot 10 = 60$ seconden = 1 minuut	

Q3(p) /1	**Als de MAXimale uitvoeringstijd 4 seconden is voor een bepaalde n, wat is dan de MAXimale tijd, in *minuten*, voor lijsten die 60 keer langer zijn?**
Oplossing: $4 \cdot 60$ seconden = 4 minuten	

Vraag 4 – Blokjes

$L[\,]$ is een lijst met n getallen $L[0], \ldots, L[n-1]$.
De **lengte** van een lijst is het aantal elementen dat deze bevat.
Een **blokje** van $L[\,]$ is een zo lang mogelijke sub-lijst van opeenvolgende en gelijke elementen van $L[\,]$.
Voorbeeld: de blokjes van $L[\,] = [4, 2, 2, 2, 1, 1, 2]$ zijn $[4]$, $[2, 2, 2]$, $[1, 1]$ en $[2]$, met respectieve lengtes 1, 3, 2 en 1.

De **bijbehorende lijst** van $L[\,]$ is de lijst met de lengtes van de blokjes van $L[\,]$.
Voortbouwend op bovenstaand voorbeeld: de bijbehorende lijst van $[4, 2, 2, 2, 1, 1, 2]$ is $[1, 3, 2, 1]$.

Q4(a) /1	**Geef de bijbehorende lijst van** $[1, 2, 3, 4, 5]$ **?**
Oplossing: $[1, 1, 1, 1, 1]$	

Q4(b) /1	**Geef de bijbehorende lijst van** $[3, 3, 3, 3, 3]$ **?**
Oplossing: $[5]$	

Q4(c) /2	**Wat is de lengte van** $L[\,]$ **als je weet dat de bijbehorende lijst** $[3, 2, 2, 3]$ **is?**
Oplossing: 10	

Q4(d) /2	**Welke lijst is gelijk aan haar eigen bijbehorende lijst?**
Oplossing: $[1]$	

In alle volgende vragen beschouwen we de lijsten $L[\,]$ **die enkel bestaan uit de getallen** 1 **en** 2 **en die beginnen met** 1.
Bijvoorbeeld, $L[\,] = [1, 2, 2, 2, 1, 2]$ of $L[\,] = [1, 1, 1, 1]$.
$A[\,]$ is de bijbehorende lijst van $L[\,]$.
Voor de twee bovenstaande voorbeelden, hebben we respectievelijk $A[\,] = [1, 3, 1, 1]$ en $A[\,] = [4]$.

Q4(e) /2	**Wat is** $A[\,]$ **als** $L[\,]$ **en** $A[\,]$ **allebei een lengte** 6 **hebben?**
Oplossing: $A[\,] = [1, 1, 1, 1, 1, 1]$	

Q4(f) /2	**Wat is** $L[\,]$ **als** $L[\,]$ **en** $A[\,]$ **allebei een lengte** 6 **hebben?**
Oplossing: $L[\,] = [1, 2, 1, 2, 1, 2]$	

Vul het onderstaande **Programma B1** aan dat als input een lijst $L[\,]$ heeft en diens lengte n (met $n > 0$) en dat als output een lijst $A[\,]$ genereert, bijbehorend aan $L[\,]$.

Het programma gebruikt `A[]←[]` om een lege lijst $A[\,]$ te creëren.
Het programma gebruikt de methode `append()` om **een element toe te voegen aan het einde van een lijst**.
Bijvoorbeeld, als $A[\,]$ leeg is en `A[].append(3)` wordt uitgevoerd, dan wordt $A[\,] = [3]$.
Als nadien `A[].append(1)` wordt uitgevoerd, dan $A[\,] = [3, 1]$.

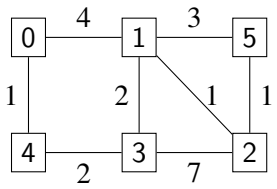
Q4(g) /6	**Vervolledig de _____ in Programma B1.** **Punten tussen 0 en 6. Je verliest 1 punt per fout of ontbrekend antwoord.**
Oplossing: De antwoorden worden hieronder getoond op een grijze achtergrond.	

```
Input  : n, L[]
Output : A[]
A[] ← []
value ← L[0]
length ← 0
for (i ← 0 to n-1 step 1)
{
    if (L[i] = value)
    {
        length ← length + 1
    }
    else
    {
        A[].append(length)
        value ← L[i]
        length ← 1
    }
}
A[].append(length)
return A[]
```

Vraag 5 – Smurfen op reis

Smurfen zijn reizigers, dat weten we allemaal ! Ze gaan vaak op tocht, onder leiding van Grote Smurf. Maar die tochten duren erg lang (« Zijn we er bijna, Grote Smurf ? ») en op zijn oude dag laat Grote Smurf al eens een steekje vallen, hij is immers al 542 jaar oud . . .

Computersmurf beslist om hem bij te staan. Hij haalt er Geosmurf bij die helpt door op onderstaande kaart alle bekende paden in het smurfenbos te tekenen en de tijd aan te duiden die nodig is om van het ene punt naar het andere te stappen.



De vierkante vakjes stellen de verschillende plaatsen voor. Elke plaats heeft een nummer.
0 is het dorp, vertrekpunt van elke reis; 1 is de grote eik; 2, Gargamels bouwvallige kasteel, *etc*.
De lijnen tussen de plaatsen stellen de paden voor.
De getallen op de paden geven de tijd aan die nodig is om van de ene plaats naar de andere te stappen.
Bijvoorbeeld, het duurt 4 uur om van 0 naar 1 te gaan.

Maar de kaart van Geosmurf stelt enkel de *rechtstreekse* paden voor.
Computersmurf wil de kaart gebruiken om de snelste *route* te berekenen naar hun bestemming.
Een *route* is een rij met rechtstreeks verbonden plaatsen die kunnen worden bereikt vanuit het smurfendorp als vertrekpunt.
Bijvoorbeeld 0, 4, 3, 2 is een route, maar 0, 3, 2 is er geen aangezien er geen rechtstreekse verbinding is tussen 0 en 3.
De *duur* van een route is de som van de duurtijden van de paden die de route vormen.
Bijvoorbeeld, de duur van route 0, 4, 3, 2 is $1 + 2 + 7 = 10$ uren.

Q5(a) /1	**Hoe lang duurt route** 0, 1, 3, 2, 5 **?**
Oplossing: 14	

Q5(b) /1	**Is het mogelijk om van** 0 **naar** 5 **te stappen in exact 7 uren ?** **Als je antwoord "ja" is, toon dan de gevolgde weg, anders antwoord je "neen".**
Oplossing: 0, 1, 5	

Q5(c) /1	**Is het mogelijk om van** 0 **naar** 5 **te stappen in minder dan 5 uren ?** **Als je antwoord "ja" is , toon dan de gevolgde weg, anders antwoord je "neen"..**
Oplossing: non	

Q5(d) /1	**Wat is de snelste route om van** 0 **naar** 5 **te stappen ?**
Oplossing: 0, 1, 2, 5	

Computersmurf legt hieronder zijn algoritme uit om de snelste routes te berekenen.

Mijn algoritme gebruikt een array `D[]`, met als indices de plaatsnummers. Aan het einde van het algoritme zal `D[x]` voor elke plaats `x` de duur bevatten van de snelste route tussen het smurfendorp en `x`.
Aan het begin initialiseer ik (in een lus **`for`**) `D[x]` met $+\infty$ voor alle plaatsen `x`.
Maar meteen daarna verander ik `D[0]` al in waarde 0 aangezien we altijd vanuit het dorp vertrekken.

Het symbool $+\infty$ betekent « waarde oneindig ».
Dit is zo'n beetje het wiskundige equivalent van « Het is heel ver, beste Smurfen ! ».
Het is een speciale waarde waarvan we veronderstellen dat ze groter is dan eender welk ander getal.
Daarenboven, als we een geheel getal optellen bij $+\infty$, blijft dit nog steeds $+\infty$.
bijvoorbeeld, $+\infty + 5 = +\infty$

Mijn algoritme maakt, tijdens het uitvoeren ervan, onderscheid tussen 2 soorten plaatsen.

- De plaatsen waarvoor de snelste route vanuit het dorp al gekend is.
  Voor deze plaatsen verandert de waarde van `D[x]` niet meer tot het einde van de uitvoering.
  We zeggen eenvoudigweg dat dit de *gekende* plaatsen zijn.
- De plaatsen waarvoor de snelste route vanuit het dorp nog niet gekend is.
  Voor deze plaatsen kan de waarde van `D[x]` nog veranderen tijdens de uitvoering van de code.
  We zeggen eenvoudigweg dat dit de *ongekende* plaatsen zijn.

In het begin veronderstelt het algoritme dat alle plaatsen *ongekend* zijn.
Op het einde, wanneer alle snelste routes berekend zijn, zullen alle plaatsen *gekend* zijn.
In de praktijk gebruikt het algoritme een array `K[]` met booleaanse waarden, zodanig dat `K[x]` de waarde **`true`** heeft voor de *gekende* plaatsen en **`false`** heeft voor de *ongekende* plaatsen.

Wanneer het vaststaat dat `D[x]` niet meer zal veranderen, betekent dit dat plaats `x` *gekend* is en bewaren we deze informatie door **`true`** in te vullen in `K[x]`.

Dé belangrijkste bewerkingen in het algoritme vinden plaats in een **`while`** lus.
Hieronder zie je wat er gebeurt bij elke herhaling van deze lus.

- Uit alle *ongekende* plaatsen, kies plaats `m` zodat `D[m]` minimaal is.
- Verander `m` van *ongekende* plaats naar *gekende* plaats.
- Onderzoek alle *opvolgers* van `m` om te checken of er voor deze een snellere route kan gevonden worden dan de route die al gekend is. De opvolgers van `m` zijn alle plaatsen die bereikbaar zijn vanaf `m` door een rechtstreekse verbinding.
  Bijvoorbeeld, de opvolgers van 0 zijn 1 en 4.
  Ik bekijk wat de duur `D[s]` is van de snelst gekende route naar elke opvolger `s` en ik probeer deze te verbeteren met een route die loopt via `m`.

Tot slot moet ik je nog uitleggen hoe ik de kaart met de paden weergeef.
Ik gebruik een array `G[][]` zodanig dat `G[i][j]` de duur is van het rechstreekse pad tussen plaats `i` en plaats `j` als dit pad bestaat.
Als er geen rechtstreekse verbinding is tussen plaats `i` en plaats `j`, duid ik dit aan met een $+\infty$ in `G[i][j]`.
Bijvoorbeeld, met de kaart van Geosmurf, hebben we `G[0][1]=4` en `G[0][3]=`$+\infty$.

Het algoritme gebruikt de functie `minFalse(D[],K[])` met als output het getal (tussen 0 en $n-1$) van de *ongekende* plaats dat de kleinste waarde heeft in `D[]`.
Als er verschillende *ongekende* plaatsen zijn met hetzelfde resultaat, geeft de functie de plaats met het kleinste getal.
Als alle plaatsen gekend zijn, geeft de functie dit aan met de output -1.
Bijvoorbeeld : als de nog ongekende plaatsen 2, 4 en 5 zijn en als `D[]=[0,7,9,15,17,9]`,
dan zoekt `minFalse(D[],K[])` het minimum tussen `D[2]=9`, `D[4]=17` en `D[5]=9`. Dit is dus 9.
Aangezien de plaatsen 2 en 5 hetzelfde resultaat geven, zal `minFalse(D[],K[])` het kleinste getal als output weergeven, namelijk 2.

Ik hoop dat je mijn uitleg goed hebt gesmurfd, pardon « begrepen » !

Hier is het algoritme van Computersmurf :
Input : n, het aantal plaatsen; G[][], de kaart met paden.
Output : D[], zodat D[m] de duur is van de snelste route van 0 naar m.

```
for (i ← 0 to n-1 step 1) {
    D[i] ← +∞
    K[i] ← false
}
D[0] ← 0
m ← minFalse(D[],K[])

while(m ≠ -1) {
   K[m] ← true
   for(s ← 0 to n-1 step 1) {
       if ((not K[s]) and (G[m][s] ≠ +∞)) {
           if (D[m] + G[m][s] < D[s]) {
               D[s] ← D[m] + G[m][s]
           }
       }
   }
   m ← minFalse(D[],K[])
}
return D[]
```

Hier zijn de eerste stappen om het algoritme uit te voeren met behulp van de kaart van Geosmurf.
De tabellen stellen D[] en K[] voor. Als afkorting geven we **true** weer met een T en **false** met een F.

- Initialisatie:

	0	1	2	3	4	5
D:	0	+∞	+∞	+∞	+∞	+∞

	0	1	2	3	4	5
K:	F	F	F	F	F	F

- Tijdens de eerste passage door de lus kiest het algoritme m= 0 en checkt het diens opvolgers 1 en 4.
  We hebben G[0][1]= 4 en G[0][4]= 1.
  Aangezien D[0]+G[0][1] = 0+4 = 4 < +∞, wordt D[1] aangepast met deze waarde.
  Evenzo wordt D[4] aangepast en krijgen we:

	0	1	2	3	4	5
D:	0	4	+∞	+∞	1	+∞

	0	1	2	3	4	5
K:	T	F	F	F	F	F

Welke zijn de volgende stappen die het algoritme gaat berekenen ?

Q5(e) /4	**Geef de arrays D[] en K[] op het einde van de tweede iteratie van de while lus.**
Oplossing: D[] = [0,4,+∞,3,1,+∞] K[] = [T,F,F,F,T,F]	

Q5(f) /4	**Geef de arrays D[] en K[] op het einde van de derde iteratie van de while lus.**
Oplossing: D[] = [0,4,10,3,1,+∞] K[] = [T,F,F,T,T,F]	

Computersmurf stelt nu vragen over de efficiëntie van het algoritme voor het berekenen van de snelste routes en over de mogelijke antwoorden...

Q5(g) /2	**Hoeveel iteraties van de `while` lus zal het algoritme uitvoeren met de kaart van Geo-Smurf?**
Oplossing: 6	

Vervolgens laat Computersmurf zijn algoritme aan Grote Smurf zien.
Grote Smurf zegt tegen hem: « Dat is heel mooi werk, mijn kleine Smurf, maar je algoritme is niet erg bruikbaar als het alleen de *duur* van de beste routes berekent! Ik wil ook weten welke routes dat zijn! ».

Computersmurf gaat dan weer aan het werk en realiseert zich dat hij voor elke plaats `x` moet berekenen welke plaats `y` direct voorafgaat aan `x` in de snelste route naar `x`. Hij noemt `y` de *voorloper* van `x`.
Bijvoorbeeld, de snelste route naar 3 is 0, 4, 3.
We moeten dus bewaren dat de voorloper van 3 in dit geval 4 is en dat de voorloper van 4 in dit geval 0 is.
Computersmurf wijzigt zijn algoritme en voegt een array toe, `P[]`, die zal dienen om voor elke plaats `x`, diens voorloper `P[x]` te bewaren.
In het begin moet `P[x]` gelijk zijn aan een speciale waarde om aan te duiden dat er nog geen berekeningen zijn gebeurd.
Computersmurf kiest als speciale waarde: -1.
Vervolgens moet hij wijzigingen aanbrengen aan de **`while`** lus om `P[x]` aan te kunnen passen.
Deze instructies zijn toegevoegd met een grijze achtergrond.

```
for (i ← 0 to n-1 step 1) {
    D[i] ← +∞
    K[i] ← false
    P[i] ← -1
}
D[0] ← 0
m ← minFalse(D[],K[])
while(m ≠ -1) {
   K[m] ← true
   for(s ← 0 to n-1 step 1) {
       if ((not K[s]) and (G[m][s] ≠ +∞)) {
           if (D[m] + G[m][s] < D[s]) {
               D[s] ← D[m] + G[m][s]
               P[s] ← m
           }
       }
   }
   m ← minFalse(D[],K[])
}
```

Je moet voorspellen wat het effect van deze lijnen zal zijn.

Q5(h) /2	**Wat is de inhoud van array `P[]` nadat het algritme uitgevoerd is op de kaart van Geosmurf ?**
Oplossing: `P[] = [-1, 0, 1, 4, 0, 2]`	