be-OI 2022 **Finale - SENIOR** samedi 23 avril 2022	**Remplissez ce cadre en MAJUSCULES et LISIBLEMENT, svp** PRÉNOM : .... NOM : .... ÉCOLE :	301 **Réservé**

Finale de l'Olympiade belge d'Informatique 2022 (durée : **2h au maximum**)

Notes générales (à lire attentivement avant de répondre aux questions)

1. Vérifiez que vous avez bien reçu la bonne série de questions (mentionnée ci-dessus dans l'en-tête):
   - Pour les élèves jusqu'en deuxième année du secondaire: catégorie **cadets**.
   - Pour les élèves en troisième ou quatrième année du secondaire: catégorie **junior**.
   - Pour les élèves de cinquième année du secondaire et plus: catégorie **senior**.
2. N'indiquez votre nom, prénom et école **que sur la première page**.
3. Indiquez **vos réponses** sur les pages prévues à cet effet, **à la fin du formulaire**.
4. Si, suite à une rature, vous êtes amené à écrire hors d'un cadre, répondez obligatoirement **sur la même feuille** (au verso si nécessaire).
5. Écrivez de façon **bien lisible** à l'aide d'un **stylo ou stylo à bille** bleu ou noir.
6. Vous ne pouvez avoir que de quoi écrire avec vous; les calculatrices, GSM, … sont **interdits**.
7. Vous pouvez toujours demander des feuilles de brouillon supplémentaires à un surveillant.
8. Quand vous avez terminé, **remettez la première page (avec votre nom) et les pages avec les réponses**. Vous pouvez conserver les autres.
9. Tous les extraits de code de l'énoncé sont en **pseudo-code**. Vous trouverez, sur les pages suivantes, une **description** du pseudo-code que nous utilisons.
10. Si vous devez répondre en code, vous devez utiliser le **pseudo-code** ou un **langage de programmation courant** (Java, C, C++, Pascal, Python, ...). Les erreurs de syntaxe ne sont pas prises en compte pour l'évaluation.

Bonne chance !

L'Olympiade Belge d'Informatique est possible grâce au soutien de nos membres:

Aide-mémoire pseudo-code

Les données sont stockées dans des variables. On change la valeur d'une variable à l'aide de $\leftarrow$. Dans une variable, nous pouvons stocker des nombres entiers, des nombres réels, ou des tableaux (voir plus loin), ainsi que des valeurs booléennes (logiques): vrai/juste (**`true`**) ou faux/erroné (**`false`**). Il est possible d'effectuer des opérations arithmétiques sur des variables. En plus des quatre opérateurs classiques ($+$, $-$, $\times$ et $/$), vous pouvez également utiliser l'opérateur %. Si a et b sont des nombres entiers, alors a/b et $a\%b$ désignent respectivement le quotient et le reste de la division entière. Par exemple, si $a = 14$ et $b = 3$, alors $a/b = 4$ et $a\%b = 2$.

Voici un premier exemple de code, dans lequel la variable age reçoit 17.

```
anneeNaissance ← 2003
age ← 2020 − anneeNaissance
```

Pour exécuter du code uniquement si une certaine condition est vraie, on utilise l'instruction **`if`** et éventuellement l'instruction **`else`** pour exécuter un autre code si la condition est fausse. L'exemple suivant vérifie si une personne est majeure et stocke le prix de son ticket de cinéma dans la variable $prix$. Notez les commentaires dans le code.

```
if (age ≥ 18)
{
    prix ← 8 // Ceci est un commentaire.
}
else
{
    prix ← 6 // moins cher !
}
```

Parfois, quand une condition est fausse, on doit en vérifier une autre. Pour cela on peut utiliser **`else if`**, qui revient à exécuter un autre **`if`** à l'intérieur du **`else`** du premier **`if`**. Dans l'exemple suivant, il y a 3 catégories d'âge qui correspondent à 3 prix différents pour le ticket de cinéma.

```
if (age ≥ 18)
{
    prix ← 8 // Prix pour une personne majeure.
}
else if (age ≥ 6)
{
    prix ← 6 // Prix pour un enfant de 6 ans ou plus.
}
else
{
    prix ← 0 // Gratuit pour un enfant de moins de 6 ans.
}
```

Pour manipuler plusieurs éléments avec une seule variable, on utilise un tableau. Les éléments individuels d'un tableau sont indiqués par un index (que l'on écrit entre crochets après le nom du tableau). Le premier élément d'un tableau tab est d'indice 0 et est noté $tab[0]$. Le second est celui d'indice 1 et le dernier est celui d'indice $N-1$ si le tableau contient N éléments. Par exemple, si le tableau tab contient les 3 nombres 5, 9 et 12 (dans cet ordre), alors $tab[0]=5$, $tab[1]=9$, $tab[2]=12$. Le tableau est de taille 3, mais l'indice le plus élevé est 2.

Pour répéter du code, par exemple pour parcourir les éléments d'un tableau, on peut utiliser une boucle **for**. La notation **for** ($i \leftarrow a$ **to** b **step** k) représente une boucle qui sera répétée tant que $i \leq b$, dans laquelle i commence à la valeur a, et est augmenté de k à la fin de chaque étape. L'exemple suivant calcule la somme des éléments du tableau tab en supposant que sa taille vaut N. La somme se trouve dans la variable sum à la fin de l'exécution de l'algorithme.

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

On peut également écrire une boucle à l'aide de l'instruction **while** qui répète du code tant que sa condition est vraie. Dans l'exemple suivant, on va diviser un nombre entier positif N par 2, puis par 3, ensuite par 4 ... jusqu'à ce qu'il ne soit plus composé que d'un seul chiffre (c'est-à-dire jusqu'à ce que $N < 10$).

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

Souvent les algorithmes seront dans un cadre et précédés d'explications. Après **Input**, on définit chacun des arguments (variables) donnés en entrée à l'algorithme. Après **Output**, on définit l'état de certaines variables à la fin de l'exécution de l'algorithme et éventuellement la valeur retournée. Une valeur peut être retournée avec l'instruction **return**. Lorsque cette instruction est exécutée, l'algorithme s'arrête et la valeur donnée est retournée.

Voici un exemple en reprenant le calcul de la somme des éléments d'un tableau.

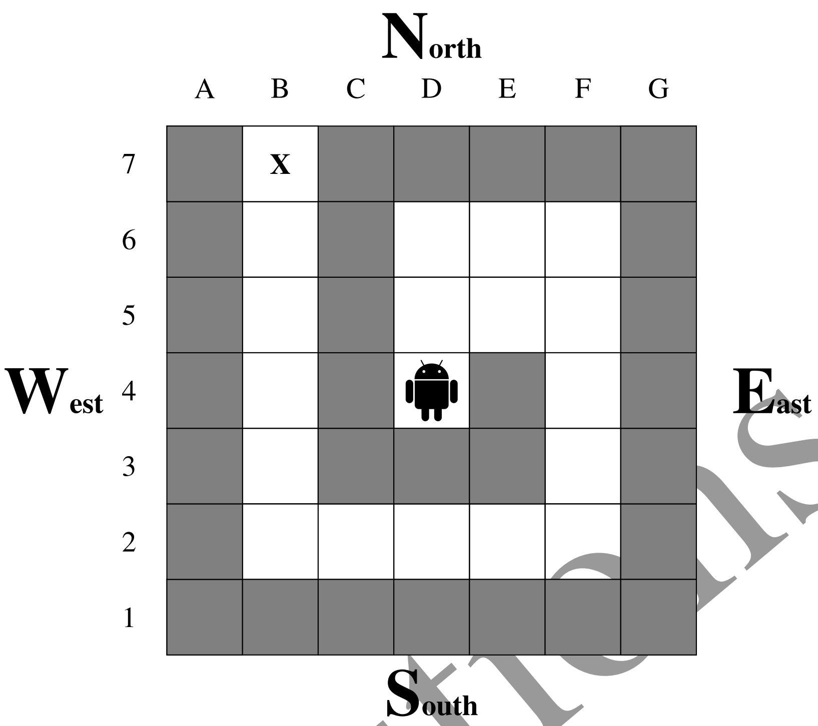
```
Input  : tab, un tableau de N nombres.
         N, le nombre d'éléments du tableau.
Output : sum, la somme de tous les nombres contenus dans le tableau.

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

Remarque: dans ce dernier exemple, la variable i est seulement utilisée comme compteur pour la boucle **for**. Il n'y a donc aucune explication à son sujet ni dans **Input** ni dans **Output**, et sa valeur n'est pas retournée.

Question 1 – Nono le petit robot

Nono le petit robot est bloqué au milieu d'un labyrinthe.



Nono peut se déplacer sur les cases blanches mais pas sur les cases grises qui représentent des murs. Au début, Nono est au centre du labyrinthe dans la case de coordonnées **D4**. Il doit atteindre la sortie: la case de coordonnées **B7** marquée par un **X**.

On déplace Nono d'une case à la fois avec la commande `Go()` qui reçoit un paramètre pouvant être: `N`, `S`, `E` ou `W`. Chaque paramètre correspond à un des quatre points cardinaux qui sont représentés sur la figure ci-dessus, avec le Nord en haut selon la convention habituelle.

On propose sur la page suivante plusieurs algorithmes pour faire sortir Nono du labyrinthe. On veut savoir si Nono est sur la case **X** à la fin de l'exécution de chacun des programmes ou s'il entre en collision avec un mur (et, dans ce cas, on demande les coordonnées du mur avec lequel la collision a lieu).

Certains algorithmes utilisent des tableaux. Par exemple `Dir←[N,E,S,W,N]` dans l'algorithme 4 initialise un tableau à 5 éléments qui seront `Dir[0]=N`, `Dir[1]=E`, `Dir[2]=S`, `Dir[3]=W` et `Dir[4]=N`.

Algorithme 1

```
Go(N)
for (i ← 1 to 2 step 1)
{
  Go(E)
}
for (i ← 1 to 3 step 1)
{
  Go(S)
}
for (i ← 1 to 4 step 1)
{
  Go(W)
}
for (i ← 1 to 5 step 1)
{
  Go(N)
}
```

Algorithme 2

```
Go(N)
i ← 1
while (i≤2)
{
  Go(E)
  i ← i+1
}
while (i≥0)
{
  Go(S)
  i ← i-1
}
while (i≤4)
{
  Go(W)
  i ← i+1
}
i ← 0
while (i<5)
{
  Go(N)
  i ← i+1
}
```

Algorithme 3

```
Go(N)
j ← 2
for (i ← 1 to j step 1)
{
  Go(E)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(S)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(W)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(N)
}
```

Algorithme 4

```
Dir ← [N,E,S,W,N]
Stp ← [1,2,3,4,5]
for (j ← 0 to 4 step 1)
{
  for (i ← 1 to Stp[j] step 1)
  {
    Go(Dir[j])
  }
}
```

Algorithme 5

```
Dir ← [N,E,W,E,S,N,S,W,E,W,N]
Stp ← [1,2,1,1,2,1,2,4,2,1,5]
for (j ← 0 to 10 step 1)
{
  for (i ← 1 to Stp[j] step 1)
  {
    Go(Dir[j])
  }
}
```

Q1(a) [5 pts]	**L'algorithme 1 amène-t-il Nono sur la case X ? Si votre réponse est « non » et que Nono entre en collision avec un mur, indiquez également les coordonnées du premier mur avec lequel Nono entre en collision.**
Solution: Oui	

Q1(b) [5 pts]	**L'algorithme 2 amène-t-il Nono sur la case X ? Si votre réponse est « non » et que Nono entre en collision avec un mur, indiquez également les coordonnées du premier mur avec lequel Nono entre en collision.**
Solution: Non, F1	

Q1(c) [5 pts]	**L'algorithme 3 amène-t-il Nono sur la case X ? Si votre réponse est « non » et que Nono entre en collision avec un mur, indiquez également les coordonnées du premier mur avec lequel Nono entre en collision.**
Solution: Oui	

Q1(d) [5 pts]	**L'algorithme 4 amène-t-il Nono sur la case X ? Si votre réponse est « non » et que Nono entre en collision avec un mur, indiquez également les coordonnées du premier mur avec lequel Nono entre en collision.**
Solution: Oui	

Q1(e) [5 pts]	**L'algorithme 5 amène-t-il Nono sur la case X ? Si votre réponse est « non » et que Nono entre en collision avec un mur, indiquez également les coordonnées du premier mur avec lequel Nono entre en collision.**
Solution: Non, C3	

Question 2 – Positions de pic

Étant donné un tableau $a[\,]$ de n nombres, une *position de pic de* $a[\,]$ est une position i dans ce tableau telle que $a[i]$ est au moins aussi grand que l'élément à sa gauche et que l'élément à sa droite (si ils existent). Notez que les tableaux sont indexés à partir de 0: le premier élément est à la position 0 et le dernier élément à la position $n-1$.

Par exemple, si $a = [2, 7, 4, 5]$, alors les positions de pic de $a[\,]$ sont les positions 1 et 3 (correspondant aux valeurs 7 et 5). Autre exemple, si $a = [6, 6, 6]$, alors toutes les positions (0, 1 et 2) sont des positions de pic. On peut montrer facilement que tout tableau a au moins une position de pic.

Q2(a) [2 pts]	**Donnez *toutes* les positions de pic du tableau $a = [4, 2, 5, 3]$.**
	Solution: 0, 2

Q2(b) [2 pts]	**Donnez *toutes* les positions de pic du tableau $a = [0, 10, 10, 0, 0]$.**
	Solution: 1, 2 et 4

Il existe plusieurs façons de trouver une position de pic, mais il est très facile de se tromper ! Votre ami Alex vous a envoyé les listings de programmes suivants la nuit dernière à 3 heures du matin. Il affirme que ses programmes peuvent trouver une position de pic pour n'importe quel tableau $a[\,]$. Vous pensez qu'Alex était probablement très fatigué lorsqu'il a écrit les programmes et vous décidez de les vérifier.
On suppose que dans chaque programme, le tableau de nombres $a[\,]$ contient $n \geq 2$ éléments.

Programme 1:

```
for (i ← 1 to n-2 step 1)
{
    if (a[i] >= i-1 and a[i] >= i+1)
    {
        return i
    }
}
if (a[0] > a[n-1])
{
    return 0
}
else
{
    return n-1
}
```

Q2(c) [1 pt]	**Quelle valeur retourne le programme 1 si $a = [9, 8, 7, 6, 5]$?**
	Solution: 1

Q2(d) [1 pt]	**Quelle valeur retourne le programme 1 si $a = [0, 1, 3, 4]$?**
	Solution: 2

Q2(e) [1 pt]	**Quelle valeur retourne le programme 1 si $a = [0, 0, 0, 0]$?**
	Solution: 3

Q2(f) [1 pt]	Quelle valeur retourne le programme 1 si $a = [1, 0, 0, 1]$?
	Solution: 3

Q2(g) [2 pts]	Le programme 1 retourne-t-il une position de pic pour n'importe quel tableau $a[\,]$?
	Solution: Non

Programme 2:

```
i ← 0
for (j ← 1 to n-1 step 1)
{
    if (a[j] > a[i])
    {
        i ← j
    }
}
return i
```

Q2(h) [1 pt]	Quelle valeur retourne le programme 2 si $a = [4, 3, 2, 1]$?
	Solution: 0

Q2(i) [1 pt]	Quelle valeur retourne le programme 2 si $a = [0, 1, 0, 5, 0, 5, 0]$?
	Solution: 3

Q2(j) [2 pts]	Le programme 2 retourne-t-il une position de pic pour n'importe quel tableau $a[\,]$?
	Solution: Oui

Programme 3:

```
if (a[0] > a[n-1])
{
    return 0
}
else
{
    return n-1
}
```

Q2(k) [1 pt]	Quelle valeur retourne le programme 3 si $a = [1, 2, 3]$?
	Solution: 2

Q2(l) [1 pt]	**Quelle valeur retourne le programme 3 si $a = [2, 2, 1]$?**
Solution: 0	

Q2(m) [2 pts]	**Le programme 3 retourne-t-il une position de pic pour n'importe quel tableau $a[\,]$?**
Solution: Non	

Programme 4:

```
for (i ← 0 to n-2 step 1)
{
    if (a[i] >= a[i+1])
    {
        return i
    }
}
return n-1
```

Q2(n) [1 pt]	**Quelle valeur retourne le programme 4 si** $a = [1, 2, 3, 2, 1]$**?**
Solution: 2	

Q2(o) [1 pt]	**Quelle valeur retourne le programme 4 si** $a = [0, 1, 0, 5, 0]$**?**
Solution: 1	

Q2(p) [1 pt]	**Quelle valeur retourne le programme 4 si** $a = [2, 2, 1, 0, 5]$**?**
Solution: 0	

Q2(q) [2 pts]	**Le programme 4 retourne-t-il une position de pic pour n'importe quel tableau** $a[\,]$**?**
Solution: Oui	

Programme 5:

```
i = n / 2
while (i > 0 and a[i-1] > a[i])
{
    i ← i-1
}
while (i < n-1 and a[i+1] > a[i])
{
    i ← i+1
}
return i
```

Q2(r) [1 pt]	**Quelle valeur retourne le programme 5 si** $a = [1, 2, 3, 4, 5]$**?**
Solution: 4	

Q2(s) [1 pt]	**Quelle valeur retourne le programme 5 si** $a = [5, 4, 3, 2, 1]$**?**
Solution: 0	

Q2(t) [1 pt]	**Quelle valeur retourne le programme 5 si** $a = [4, 3, 2, 2, 6, 5]$**?**
Solution: 4	

Q2(u) [1 pt]	**Quelle valeur retourne le programme 5 si $a = [0, 4, 2, 5, 3]$?**
Solution: 1	

Q2(v) [2 pts]	**Le programme 5 retourne-t-il une position de pic pour n'importe quel tableau $a[\,]$?**
Solution: Oui	

Le lendemain, Barbara vous envoie un autre programme pour trouver une position de pic. Il semble intrigant, mais vous êtes convaincu que Barbara l'a écrit correctement (notamment parce que, contrairement à Alex, elle ne vous l'a pas envoyé au milieu de la nuit). Cependant, ne manquant jamais une occasion de mettre au défi les capacités de déduction de ses amis, Barbara a laissé quelques blancs à remplir.

```
l = 0
r = ...          // (e1)
while (...)      // (e2)
{
    mid ← (l + r) / 2
    if (...)     // (e3)
    {
        l ← mid + 1
    }
    else
    {
        r ← mid
    }
}
return l
```

Q2(w) [2 pts]	Quelle est l'expression `(e1)` dans le programme de Barbara ?
Solution: `n-1`	

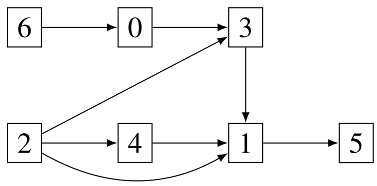
Q2(x) [2 pts]	Quelle est l'expression `(e2)` dans le programme de Barbara ?
Solution: `l<r` (autre solution `l≠r`)	

Q2(y) [2 pts]	Quelle est l'expression `(e3)` dans le programme de Barbara ?
Solution: `a[mid] <= a[mid+1]` ou `a[mid] < a[mid+1]`	

Question 3 – La fabrique d'ordinateurs

L'entreprise be-OI père et fils fabrique des pièces d'ordinateur. Un composant d'ordinateur est un objet complexe et nécessite, pour être fabriqué, d'autres composants, qui sont eux-mêmes assemblés à partir d'autres, *etc*. L'entreprise dispose de toutes les machines pour tout fabriquer, mais ne peut pas les faire fonctionner toutes en même temps. Elle doit donc trouver *l'ordre adéquat pour fabriquer les pièces, en supposant qu'on ne fera fonctionner chaque machine qu'une seule fois*.

Chaque machine porte un numéro de 0 à $n-1$ et l'entreprise a représenté les dépendances entre les machines à l'aide d'un diagramme comme sur l'exemple suivant.



Sur ce schéma, chaque machine est représentée par un rectangle portant le numéro de la machine. Une flèche allant de la machine i à la machine j signifie que la machine i doit fonctionner avant la machine j, car les pièces produites par i sont nécessaires à celles produites par j (on ne se souciera pas des quantités ici: il suffit que i fonctionne une seule fois pour permettre à j de fonctionner). Par exemple, il est obligatoire de faire fonctionner la machine 2 et la machine 0 avant la machine 3. Tout ordre de fonctionnement des machines où 3 est mise en marche avant 2 ou avant 0 est donc impossible. Pour que le diagramme ait du sens, on suppose qu'il ne contient jamais de *cycle*, c'est-à-dire qu'il n'est jamais possible de partir d'une machine et de suivre une série de flèches pour revenir à la même machine.

Comme on le voit, certaines machines peuvent être mises en marche immédiatement.

Q3(a) [1 pt]	**Selon le schéma ci-dessus, quelles sont toutes les machines qu'on peut mettre en marche immédiatement ?**
Solution: 2 et 6	

Q3(b) [1 pt]	**Selon le schéma ci-dessus, quelles sont toutes les machines qu'on peut mettre en marche si on a déjà fait fonctionner la machine 2 uniquement (rappel: on ne fait fonctionner chaque machine qu'une seule fois) ?**
Solution: 4 et 6	

Parmi les ordres suivants, lesquels sont possibles en tenant compte des contraintes données dans le diagramme ci-dessus ?

	Possible	Impossible	Ordre d'exécution des machines
Q3(c) [1 pt]	☐	☒	0,1,2,3,4,5,6
Q3(d) [1 pt]	☒	☐	6,0,2,3,4,1,5
Q3(e) [1 pt]	☐	☒	6,5,4,3,2,1,0
Q3(f) [1 pt]	☒	☐	2,4,6,0,3,1,5
Q3(g) [1 pt]	☐	☒	2,4,1,6,0,3,5

Clairement, il y a une condition bien précise que doivent respecter les machines (dans un diagramme comme celui ci-dessus) pour qu'on puisse les faire fonctionner immédiatement.

Q3(h) [2 pts]		**Parmi les propositions ci-dessous, quelle est celle qui caractérise exactement les machines qu'on peut faire fonctionner immédiatement ?**
	☐	Les machines qui ont un nombre pair de flèches entrantes.
	☐	Les machines qui ont au plus 2 flèches entrantes.
	☒	Les machines qui n'ont aucune flèche entrante.
	☐	Les machines qui ont au moins une flèche entrante.
	☐	Les machines qui n'ont aucune flèche entrante et un numéro (de machine) pair.

On cherche maintenant un algorithme pour trouver un ordre dans lequel faire fonctionner les machines, qui soit compatible avec le diagramme. Le diagramme est donné comme une matrice de booléens $M[n][n]$, où $M[i][j]$ vaut **`true`** si et seulement s'il y a une flèche allant de la machine i à la machine j.

Voici la matrice pour le diagramme donné en exemple.
Les cases valant **`true`** ont été mises en évidence pour améliorer la lisibilité.

	0	1	2	3	4	5	6
0	false	false	false	**true**	false	false	false
1	false	false	false	false	false	**true**	false
2	false	**true**	false	**true**	**true**	false	false
3	false	**true**	false	false	false	false	false
4	false	**true**	false	false	false	false	false
5	false	false	false	false	false	false	false
6	**true**	false	false	false	false	false	false

M[ ][ ]

$M[2][4]$ est entouré en trait plein, tandis que $M[4][2]$ est entouré en pointillé.
$M[2][4] =$ **`true`** car il y a une flèche allant de la machine 2 à la machine 4.
$M[4][2] =$ **`false`** car il n'y a pas de flèche allant de la machine 4 à la machine 2.

L'algorithme qu'on propose repose sur une structure de données appelée *file*. Il s'agit d'une file, avec un début et une fin. On ajoute les éléments à la fin de la file et on traite les éléments en commençant par celui au début de la file. On dispose des fonctions: `Empty()`, qui vide la file (elle ne contient plus aucun élément); `Insert(`v`)`, qui ajoute l'élement v à la **fin** de la file; `Remove()`, qui renvoie l'élement du **début** de file et enlève cet élément de la file; et enfin `IsEmpty()` qui renvoie **`true`** si la file est vide et **`false`** si elle contient au moins un élément.

L'algorithme est le suivant.

1. On commence par insérer dans la file toutes les machines qu'on peut faire fonctionner immédiatement.
2. Ensuite, on répète les mêmes actions jusqu'à ce que la file soit vide:
   (a) prendre une machine en début de file et la faire fonctionner;
   (b) si l'exécution de cette machine permet d'en faire fonctionner d'autres, ajouter ces nouvelles machines dans la file.

Remarques.
Une valeur booléenne **bool** peut être utilisée directement dans une instruction **if** (**bool**){...}.
L'opérateur **not** agit comme suit sur les valeurs booléennes: **not true** est égal à **false** et **not false** est égal à **true**.

Voici une implémentation de l'algorithme où certaines parties du code sont manquantes:

```
Input  : Une matrice M[n][n] de taille n × n, qui représente les dépendances entre
    les n machines.
Output : Fait fonctionner les machines dans un ordre compatible avec M.

Empty()
D ← [0,...,0] // (un tableau de longueur n, initialisé avec des 0)

for (j ← 0 to n−1 step 1)
{
  for (i ← 0 to n−1 step 1)
  {
    if(M[i][j])
    {
      D[...] ← ... // (A) et (B)
    }
  }
  if (...) // (C)
  {
    Insert(j)
  }
}

while (not IsEmpty())
{
  m ← Remove()
  Faire fonctionner la machine m
  for (i ← 0 to n−1 step 1)
  {
    if (M[m][i])
    {
      D[i] ← ... // (D)
      if (...) // (E)
      {
        Insert(i)
      }
    }
  }
}
```

Q3(i) [2 pts]	Quelle est l'expression **(A)** manquante dans l'algorithme ?
Solution: j	

Q3(j) [2 pts]	Quelle est l'expression **(B)** manquante dans l'algorithme ?
Solution: $D[j]+1$	

Q3(k) [2 pts]	**Quelle est la condition `(C)` manquante dans l'algorithme ?**
Solution: $D[j] = 0$	

Q3(l) [2 pts]	**Quelle est l'expression `(D)` manquante dans l'algorithme ?**
Solution: $D[i] - 1$	

Q3(m) [2 pts]	**Quelle est la condition `(E)` manquante dans l'algorithme ?**
Solution: $D[i] = 0$	

Enfin, on aimerait réaliser la file et ses quatre fonctions utilisées dans l'algorithme à l'aide d'un tableau $f[n]$ de taille n et de deux indices s et e. Les éléments qui sont dans la file seront tous les éléments aux indices $s, s+1, \ldots, e-1$. L'élement à l'indice s est celui au début et celui à l'indice $e-1$ est à la fin. Par exemple, si $s = 3$ et $e = 6$, le début de la file sera $f[3]$, puis la file contiendra les éléments $f[4]$ et $f[5]$. Si un des indices atteint la dernière case $n-1$ du tableau, on peut malgré tout étendre la file en traitant le tableau de façon circulaire, ce qui revient à remettre l'indice à 0. Par exemple, si $n = 7$, $s = 5$ et $e = 2$, la file contiendra les éléments $f[5], f[6], f[0], f[1]$, dans cet ordre, avec $f[5]$ au début. On vous demande de compléter le code ci-dessous pour obtenir cette implémentation de la file. Indice: utilisez la fonction mathématique *modulo* (représentée par l'opérateur %): $a\%b$ est le reste de la division de a par b.

```
Empty()
{
  s ← 0
  e ← 0
}

Insert(v)
{
  f[e] ← v
  e ← ... // (F)
}

IsEmpty()
{
  return (...) // (G)
}

Remove()
{
  v ← f[s]
  s ← ... // (H)
  return v
}
```

Q3(n) [2 pts]	**Quelle est l'expression `(F)` manquante dans l'algorithme ?**
Solution: $(e+1)\%n$	

Q3(o) [2 pts]	**Quelle est la condition `(G)` manquante dans l'algorithme ?**
Solution: $e = s$	

Q3(p) [2 pts]	**Quelle est l'expression `(H)` manquante dans l'algorithme ?**
Solution: $(s + 1)\%n$	

Question 4 – LIDAR

Le LIDAR est une technologie qui utilise la lumière ou des lasers pour déterminer les distances ou les hauteurs des objets. Vous faites partie d'une équipe qui veut mesurer la hauteur exacte de différentes parties de la Grande Muraille de Chine. Pendant que vous volez au-dessus de la Muraille, vous remarquez soudain que votre équipement LIDAR a un problème : au lieu de mesurer la hauteur d'un seul segment, il mesure et additionne les hauteurs de plusieurs segments consécutifs. Vous n'avez eu l'autorisation de survoler le Mur qu'aujourd'hui, et vous n'avez pas le temps pour reconfigurer le LIDAR. Comme vous êtes l'expert en algorithmique de l'équipe, vos collègues comptent sur vous pour récupérer les hauteurs réelles des segments à partir des données recueillies par le scan.

Pour simplifier, nous représenterons le mur par une liste $W[\,]$ de n entiers positifs ou nuls ($W[i] \geq 0$).
Chaque $W[i]$ représente la hauteur d'un segment du mur.
Le LIDAR prendra q scans, chacun sera la somme d'éléments consécutifs de $W[\,]$.

Exemple: prenons un mur avec $n = 4$ segments : $W = [4, 3, 5, 8]$.
Comme d'habitude, les indices commencent à 0, donc ici $W[0] = 4$, $W[1] = 3$, $W[2] = 5$ et $W[3] = 8$.
Les résultats de 2 scans par le LIDAR pourraient être (remarquez la notation $W[i \ldots j] = W[i] + \ldots + W[j]$).
$W[1 \ldots 3] = W[1] + W[2] + W[3] = 3 + 5 + 8 = 16$ et $W[0 \ldots 1] = W[0] + W[1] = 4 + 3 = 7$.

Pour les quatre prochaines questions, on considère un mur $W[\,]$ avec 4 segments.
Le LIDAR a mesuré $W[0 \ldots 2] = 31$, $W[0 \ldots 3] = 42$ et $W[2 \ldots 3] = 17$.

Q4(a) [1 pt]	**Quelle est la hauteur de $W[3]$?**
Solution: 11, puisque $W[3] = W[0 \ldots 3] - W[0 \ldots 2] = 42 - 31 = 11$	

Q4(b) [1 pt]	**Quelle est la hauteur de $W[2]$?**
Solution: 6, puisque $W[2] = W[0 \ldots 2] + W[2 \ldots 3] - W[0 \ldots 3] = 31 + 17 - 42 = 6$	

Q4(c) [1 pt]	**Combien vaut $W[0 \ldots 1]$?**
Solution: 25, puisque $W[0 \ldots 1] = W[0 \ldots 3] - W[2 \ldots 3] = 42 - 17 = 25$	

Q4(d) [1 pt]	**Combien de murs différents sont compatibles avec les mesures du LIDAR ?**
Solution: 26, puisque seuls $W[0]$ et $W[1]$ sont inconnus mais que leur somme doit valoir 25 (n'oubliez pas qu'une hauteur peut valoir 0).	

Avant de résoudre le problème réel, nous essayons d'abord de simuler efficacement les mesures. Cela nous permettra de générer des cas de test afin de vérifier si nous avons fait les choses correctement.

Implémentons d'abord un algorithme simple, mais lent. Les entrées sont le mur $W[\,]$ et sa longueur n, le nombre de scans q, et les extrémités gauche et droite des scans dans les tableaux $left[\,]$ et $right[\,]$. La sortie est un tableau $Lidar[\,]$ dont les éléments sont les sommes $W[left[i] \ldots right[i]]$ mesurées par les scans.

```
Input  : n, W[], q, left[], right[]
Output : Lidar[]

Lidar[] is initialized to q zeroes.

for (i ← 0 to ... step 1) // (i)
{
    for (j ← left[i] to ... step 1) // (ii)
```

```
    {
        Lidar[i] ← Lidar[i] + ... //(iii)
    }
}
```

Il y a quelques trous dans cette implémentation, pouvez-vous les combler ?

Q4(e) [1 pt]	**Quelle est l'expression (i) dans l'algorithme ci-dessus ?**
Solution: $q-1$	

Q4(f) [1 pt]	**Quelle est l'expression (ii) dans l'algorithme ci-dessus ?**
Solution: $right[i]$	

Q4(g) [1 pt]	**Quelle est l'expression (iii) dans l'algorithme ci-dessus ?**
Solution: $W[j]$	

Voici un algorithme plus rapide, avec un seul passage sur les valeurs de $W[\,]$ et de $left[\,]$ et $right[\,]$. L'idée est de calculer d'abord une table de préfixes $P[\,]$ telle que $P[0]=0$ et $P[i+1]=W[0\ldots i]$ et de l'utiliser ensuite pour calculer $Lidar[\,]$. Les entrées et les sorties sont les mêmes que dans le premier algorithme.

```
Input  : n, W[], q, left[], right[]
Output : Lidar[]

P[] ← [0,0,---,0] //n+1 "0".

for (i ← 0 to ... step 1) // (i)
{
    P[i + 1] ← ...   // (ii)
}

for (i ← 0 to ... step 1) // (iii)
{
    Lidar[i] ← ... // (iv)
}
```

Vous devez juste remplir quelques blancs.

Q4(h) [1 pt]	**Quelle est l'expression (i) dans l'algorithme ci-dessus ?**
Solution: $n-1$	

Q4(i) [2 pts]	**Quelle est l'expression (ii) dans l'algorithme ci-dessus ?**
Solution: $P[i]+W[i]$	

Q4(j) [1 pt]	**Quelle est l'expression (iii) dans l'algorithme ci-dessus ?**
Solution: $q-1$	

Q4(k) [3 pts]	**Quelle est l'expression (iv) dans l'algorithme ci-dessus ?**
	Solution: $P[right[i]+1]-P[left[i]]$

Si nous pouvions calculer la table $P[\,]$ de l'algorithme précédent à partir des scans, nous pourrions facilement en déduire toutes les valeurs $W[i]$. Essayons de le faire efficacement. Les entrées sont le nombre de segments n dans le mur, le nombre de scans q, les extrémités des scans dans les tableaux $left[\,]$et $right[\,]$, et les valeurs des balayages dans $Lidar[i] = W[left[i]\dots right[i]]$. **Note importante : pour cet algorithme, nous supposons que les balayages sont tous cohérents (c'est-à-dire qu'il n'y a pas de contradictions), et que les balayages déterminent uniquement** $P[\,]$**.**

La matrice $K[\,][\,]$ sera utilisée pour contenir toutes les mesures que nous connaissons pour chaque extrémité de chaque balayage. Par exemple, si un scan est $W[3\dots7] = 61$, alors nous savons que $P[8] = W[0\dots7] = W[0\dots2] + W[3\dots7] = P[3]+61$ et aussi que $P[3] = P[8]-61$. Pour tenir compte de cela, nous enregistrons la contrainte $(8,61)$ dans la liste $K[3]$ et la contrainte $(3,-61)$ dans la liste $K[8]$. Faites bien attention aux indices !

Notez que $K[\,][\,]$ est un tableau de listes et que les éléments de chaque liste $K[i][\,]$ sont des couples de nombres $(indice, valeur)$.
La ligne **foreach** `((index, scan)in K[current])` du code va démarrer une boucle qui sera exécutée une fois pour chaque élément de la liste $K[current][\,]$. Les valeurs de `index` et `scan` peuvent être utilisées à l'intérieur de cette boucle.

L'opération `append x à y` dans le code ajoute x comme nouvel élément à droite de la liste y. Par exemple, si $x = 4$ et $y = [6,2]$, alors y deviendra $[6,2,4]$.
L'opération inverse est `remove last element` : elle transforme $y = [6,2,4]$ en $[6,2]$ et attribue 4 à la variable spécifiée.

```
Input  : n, q, left[], right[], Lidar[]
Output : P[]

Initialize K[][] to n+1 empty arrays.

for (i ← 0 to q - 1 step 1)
{
    append (..., Lidar[i]) to K[left[i]]     // (i)
    append (...) to K[...] // (ii), (iii)
}

Initialize processed[] to n values false.
P[0] ← 0
processed[0] ← true
Initialize todo[] to [0]

while (todo[] is not empty)
{
    current ← remove last element of todo[]
    foreach ((index, scan) in K[current][])
    {
        P[index] ← ... // (iv)
        if (not processed[index])
        {
            processed[index] ← true
            append index to todo
        }
```

```
        }
}
```

Q4(l) [1 pt]	**Quelle est l'expression (i) dans l'algorithme ci-dessus ?**
Solution: $right[i] + 1$	

Q4(m) [3 pts]	**Quelle est l'expression (ii) dans l'algorithme ci-dessus ?**
Solution: $left[i], -Lidar[i]$	

Q4(n) [1 pt]	**Quelle est l'expression (iii) dans l'algorithme ci-dessus ?**
Solution: $right[i] + 1$	

Q4(o) [5 pts]	**Quelle est l'expression (iv) dans l'algorithme ci-dessus ?**
Solution: $P[current] + scan$	

Les scans suivants d'**un mur avec 9 segments** sont cohérents et déterminent de façon unique $P[\,]$ (vous pouvez utiliser l'algorithme précédent) et $W[\,]$.

- $W[0 \ldots 0] = 4$
- $W[0 \ldots 3] = 18$
- $W[0 \ldots 5] = 20$
- $W[1 \ldots 4] = 14$
- $W[2 \ldots 8] = 23$
- $W[3 \ldots 4] = 1$
- $W[4 \ldots 7] = 15$
- $W[5 \ldots 7] = 15$
- $W[5 \ldots 8] = 18$
- $W[7 \ldots 8] = 7$
- $W[8 \ldots 8] = 3$

Q4(p) [4 pts]	**Quelles sont les hauteurs des 9 segments du mur $W[\,]$?**
Solution: $[4, 9, 4, 1, 0, 2, 9, 4, 3]$	

Les scans suivants d'**un mur avec 9 segments** sont cohérents mais ne déterminent pas $W[\,]$ de manière unique.

- $W[0 \ldots 3] = 22$
- $W[0 \ldots 4] = 22$

- $W[1\ldots1] = 2$
- $W[1\ldots4] = 16$
- $W[1\ldots5] = 24$
- $W[2\ldots4] = 14$
- $W[3\ldots6] = 19$
- $W[3\ldots8] = 30$
- $W[7\ldots8] = 11$

Q4(q) [4 pts]	**Combien de murs $W[\,]$ sont compatibles avec les scans donnés ?**
Solution: 144, n'importe quel mur $[6, 2, 14-x, x, 0, 8, 11-x, y, 11-y]$ où x et y peuvent chacun prendre n'importe quelle valeur entre 0 et 11 est possible.	

Question 5 – Binairo

Binairo alias **Takuzu** est un jeu de logique avec des règles simples mais des solutions complexes.
On joue sur une grille carrée de taille paire ($2n$ lignes de $2n$ cellules).
Au départ, certaines cellules contiennent un pion, noir ou blanc. Toutes les autres cellules sont vides.
Le but est de mettre un pion dans chaque cellule en respectant les règles suivantes.

- **Règle 1:** Chaque ligne ou colonne doit contenir n pions blancs et n pions noirs.
- **Règle 2:** Dans chaque ligne ou colonne, on peut avoir deux pions consécutifs de la même couleur, mais pas trois.
- **Règle 3:** Chaque ligne et chaque colonne est unique (nous n'aurons pas besoin de cette règle dans la suite).

Voici un exemple de grille binairo 6x6 (à gauche) et l'unique solution (à droite).

●					
○	○				
				○	
	●		●		●
			●		

●	○	●	○	●	○
○	○	●	●	○	●
●	●	○	○	●	○
●	○	●	○	○	●
○	●	○	●	○	●
○	●	○	●	●	○

Voici 2 exemples de lignes qui ne respectent pas les règles.

○○●○●○○● Règle 1 non respectée (il y a plus de pions blancs que de pions noirs).

○○●●●○○● Règle 2 non respectée (il y a plus de 2 pions noirs consécutifs).

Complétez les lignes suivantes **en respectant les 2 premières règles**.
Si nécessaire répondez ici au brouillon avant de **recopier sur la feuille de réponses**.

Q5(a) [1 pt]	**Complétez la ligne**
	Solution: ●●○○

●	●		

Q5(b) [1 pt]	**Complétez la ligne**
	Solution: ○●○●●○●○

○			●	●			○

Q5(c) [1 pt]	**Complétez la ligne**
	Solution: ●○●○○●○○●●

●		●						●	●

De combien de manières peut-on compléter les lignes suivantes **en respectant les 2 premières règles** ?

Q5(d) [1 pt]	**Nombre de manières de compléter**
	Solution: 2

○	●		

Q5(e) [1 pt]	**Nombre de manières de compléter** ○○◻◻
Solution: 1	

Q5(f) [1 pt]	**Nombre de manières de compléter** ◻○◻◻●◻
Solution: 3	

Q5(g) [1 pt]	**Nombre de manières de compléter** ○◻◻●○◻◻●
Solution: 4	

Q5(h) [1 pt]	**Nombre de manières de compléter** ○○●○◻◻◻◻
Solution: 2	

Q5(i) [1 pt]	**Nombre de manières de compléter** ○◻○◻◻◻◻◻○○
Solution: 1	

Q5(j) [1 pt]	**Nombre de manières de compléter** ○◻◻○◻◻○◻◻○
Solution: 6	

Q5(k) [1 pt]	**Nombre de manières de compléter** ●◻○○◻●◻○◻●
Solution: 0	

Q5(l) [1 pt]	**Nombre de manières de compléter** ◻◻◻◻
Solution: 6	

Q5(m) [1 pt]	**Nombre de manières de compléter** ◻◻◻◻◻◻
Solution: 14	

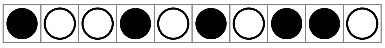
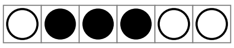
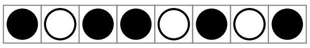
Q5(n) [1 pt]	**Nombre de manières de compléter** ○○●◻◻◻◻◻◻●○○
Solution: 1	

Q5(o) [1 pt]	**Nombre de manières de compléter** ◻○●◻◻○◻◻●◻◻○
Solution: 5	

Le **programme 1** vérifie si une ligne de pions noirs et de pions blancs respecte les 2 premières règles. Certaines parties du code sont manquantes, il faut les compléter.

La ligne à vérifier est représentée par un tableau d'entiers $binairo[\,]$. Les pions blancs sont représentés par des 1 et les pions noirs par des 2. Un nombre n est aussi donné: le tableau $binairo[\,]$ contient $2n$ éléments (numérotés de 0 à $2n-1$) et selon la **Règle 1**, il doit y avoir le même nombre n de pions blancs et de pions noirs. Le programme doit retourner **true** si les deux premières règles sont respectées et **false** sinon.

Exemples.

- Pour  $n = 5$, $binairo = [2, 1, 1, 2, 1, 2, 1, 2, 2, 1]$, le programme retourne **true**.
- Pour  $n = 3$, $binairo = [1, 2, 2, 2, 1, 1]$, le programme retourne **false**.
- Pour  $n = 4$, $binairo = [2, 1, 2, 2, 1, 2, 1, 2]$, le programme retourne **false**.

Le programme parcourt $binairo[\,]$ et utilise la variable $t1$ pour compter le nombre **total** de pions blancs et la variable $c1$ pour compter le nombre de pions blancs **consécutifs** parmi les derniers analysés. Les variables $t2$ et $c2$ font de même pour les pions noirs.

Programme 1: Les solutions sont à la suite des ...

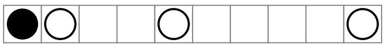
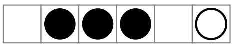
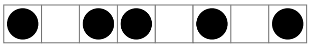
```
Input  : n, binairo[]
Output : true or false
t1 ← 0; t2 ← 0; c1 ← 0; c2 ← 0
for (i ← 0 to 2*n-1 step 1){
    if (binairo[i] = 1) {
        t1 ← ...t1+1
        if (t1 > ...n) {return false}
        c1 ← ...c1+1
        if (c1 = ...3) {return false}
        c2 ← ...0
    }
    else {
        t2 ← ...t2+1
        if (t2 > ...n) {return false}
        c2 ← ...c2+1
        if (c2 = ...3) {return false}
        c1 ← ...0
    }
}
return ...true
```

Q5(p) [6 pts]	**Complétez les ... dans le programme 1.** Note entre 0 et 6. Vous perdez 1 point par faute.
Solution: Les solutions sont à la suite des ...	

Le **programme 2** est une amélioration du programme 1. Il permet de vérifier si une ligne *dont éventuellement certaines cellules sont encore vides* respecte les 2 premières règles. Attention: le programme n'essaie pas de vérifier s'il est possible de remplir la ligne en respectant les règles, il vérifie seulement si les 2 premières règles sont respectées *pour l'instant par les pions qui sont déjà dans la ligne*.

Le tableau $binairo$[ ] peut contenir des 0 qui représentent les cellules vides (rappel: 1 et 2 représentent les pions).
Les variables $t1$ et $t2$ sont remplacées par un tableau t[ ] à 3 éléments.
t[0] est inutilisé, t[1] compte le nombre total de pions blancs, t[2] compte le nombre total de pions noirs.
Les variables $c1$ et $c2$ sont remplacées par un tableau c[ ] à 3 éléments.
c[0] est inutilisé, c[1] compte le nombre de pions blancs consécutifs, c[2] compte le nombre de pions noirs consécutifs.

Exemples.

- Pour  $n = 5$, $binairo = [2, 1, 0, 0, 1, 0, 0, 0, 0, 1]$ et le programme retourne **true**.
- Pour  $n = 3$, $binairo = [0, 2, 2, 2, 0, 1]$, le programme retourne **false** car il y a plus de 2 pions noirs consécutifs.
- Pour  $n = 4$, $binairo = [2, 0, 2, 2, 0, 2, 0, 2]$, le programme retourne **false** car il y a déjà trop de pions noirs.

Programme 2: Les solutions sont à la suite des ...

```
Input  : n, binairo[]
Output : true or false
t ← [0,0,0]
c ← [0,0,0]
for (i ← 0 to 2*n-1 step 1){
    j = binairo[i]
    if (...j ≠ 0) {
        t[j] ← ...t[j]+1
        if (...t[j] > n) {return false}
        c[j] ← ...c[j]+1
        if (...c[j] = 3) {return false}
        c[...3-j] ← ...0
    }
}
return ...true
```

Q5(q) [6 pts]	**Complétez les ... dans le programme 2.** Note entre 0 et 6. Vous perdez 1 point par faute.
Solution: Les solutions sont à la suite des ...	

Le **programme 3** compte le nombre de manières de remplir une ligne avec n pions blancs et n pions noirs (en respectant les 2 premières règles, on ne le rappelera plus).

Illustrons l'idée du programme dans le cas où $n = 2$.
Notation: on écrit # devant une ligne pour signifier le nombre de manières de compléter cette ligne. On a successivement :

\# □□□□ = # ○□□□ + # ●□□□
= (# ○○□□ + # ○●□□) + (# ●○□□ + # ●●□□)
= (# ○○●□ + (# ○●○□ + # ○●●□)) + ((# ●○○□ + # ●○●□) + # ●●○□)
= (# ○○●● + (# ○●○● + # ○●●○)) + ((# ●○○● + # ●○●○) + # ●●○○)
= (1 + (1 + 1)) + ((1 + 1) + 1) = 6

Le programme 3 n'utilise qu'une seule donnée en entrée: le nombre n.
On définit d'abord une fonction $TotalBinairo(i, t1, c1, t2, c2)$ qui calcule le nombre de manières de compléter une ligne, en sachant que :

- les cellules ayant un numéro inférieur à i sont déjà remplies,
- il y a déjà $t1$ pions blancs et $t2$ pions noirs,
- les dernières cellules placées sont $c1$ pions blancs consécutifs (et dans ce cas $c2 = 0$) ou $c2$ pions noirs consécutifs (et dans ce cas $c1 = 0$).

Pour effectuer ce calcul, la fonction vérifie d'abord si $i = 2n$, ce qui signifie que la ligne est remplie. Dans ce cas, la valeur 1 est retournée pour compter cette manière de remplir la ligne. Sinon, la fonction vérifie si on peut placer un pion blanc dans la case i. Si c'est le cas, elle s'appelle elle-même avec des valeurs adaptées des paramètres et reçoit le nombre de manières de compléter les cellules qui restent après ce nouveau pion blanc. Ensuite, la fonction fait de même en essayant de placer un pion noir dans la case i. En cas d'impossibilité (impossible d'ajouter un pion blanc ou un pion noir dans la cellule i), la fonction doit retourner 0.

La dernière ligne du programme appelle la fonction pour calculer le nombre de manières de compléter une ligne vide.

Programme 3: Les solutions sont à la suite des . . .

```
TotalBinairo( i, t1, c1, t2, c2){
    if (i = 2*n){return 1}
    else {
        TotBin ← 0
        if (t1<n and c1<2){
            TotBin ← TotBin + TotalBinairo( ...i+1, ...t1+1, ...c1+1, ...t2, ...0)
        }
        if (...t2<n and c2<2){
            TotBin ← TotBin + TotalBinairo( ...i+1, ...t1, ...0, ...t2+1, ...c2+1)
        }
        return ...TotBin
    }
}
return TotalBinairo(0, 0, 0, 0, 0)
```

Q5(r) [6 pts]	**Complétez les . . . dans le programme 3.** **Note entre 0 et 6. Vous perdez 1 point par faute.**
Solution: Les solutions sont à la suite des . . .	