be-OI 2023 **Finale - SENIOR** samedi 18 mars 2023	**Remplissez ce cadre en MAJUSCULES et LISIBLEMENT svp** PRÉNOM : ..... NOM : ..... ÉCOLE :	O **Réservé**

Finale de l'Olympiade belge d'Informatique 2023 (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 **cadet**.
   - 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 cette page**.
3. Indiquez **vos réponses** sur les pages prévues à cet effet. Écrivez de façon **bien lisible** à l'aide d'un **bic ou stylo** bleu ou noir.
4. Utilisez un crayon et une gomme lorsque vous travaillez au brouillon sur les feuilles de questions.
5. Vous ne pouvez avoir que de quoi écrire avec vous; les calculatrices, GSM, smartphone, ... sont **interdits**.
6. Vous pouvez toujours demander des feuilles de brouillon supplémentaires à un surveillant.
7. Quand vous avez terminé, **remettez la première page (avec votre nom) et les pages avec les réponses**, vous pouvez conserver les autres pages.
8. 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.
9. 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 ← 2006
age ← 2023 − 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 $prix$. Observez 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.

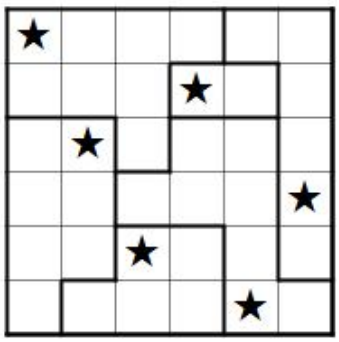
Question 1 – Star Battle

Star Battle est un jeu de logique où il faut placer **n** étoiles dans une grille de **n** lignes et **n** colonnes divisée en **n** zones séparées par des murs représentés par des traits épais.

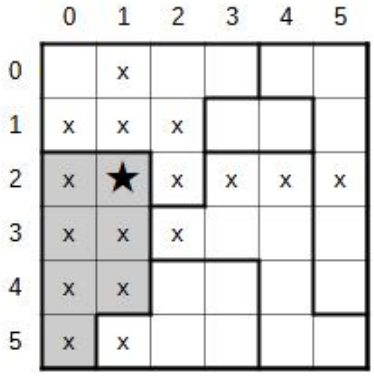
Il faut respecter les règles suivantes.

- **Règle 1:** chaque ligne doit contenir exactement 1 étoile.
- **Règle 2:** chaque colonne doit contenir exactement 1 étoile.
- **Règle 3:** chaque zone doit contenir exactement 1 étoile.
- **Règle 4:** les cases qui entourent une étoile ne peuvent pas contenir une autre étoile.

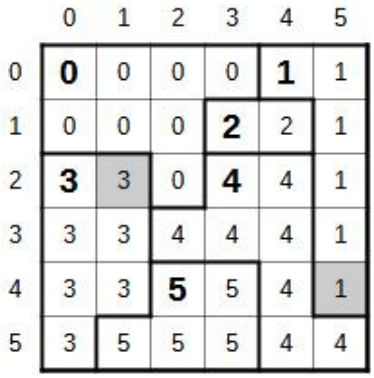
Exemple: la grille à droite est remplie en respectant les 4 règles.



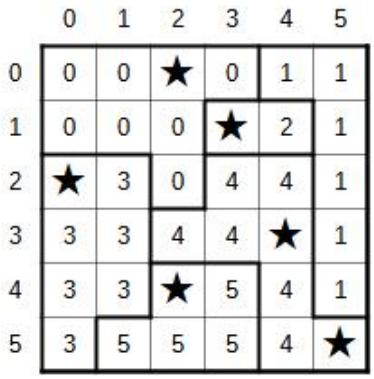
Voici un exemple avec **n=6** où on a seulement placé une étoile.
Les numéros des lignes (de 0 à 5) sont inscrits à gauche de la grille.
Les numéros des colonnes (de 0 à 5) sont inscrits au-dessus de la grille.
L'étoile a été placée à l'intersection de la ligne 2 et de la colonne 1.
Les cases où l'on ne peut plus placer d'autres étoiles sont marquées par un "x":
• les cases de la ligne 2 (règle 1),
• les cases de la colonne 1 (règle 2),
• les cases (en gris) de la zone contenant l'étoile (règle 3),
• les 8 cases autour de l'étoile (règle 4).



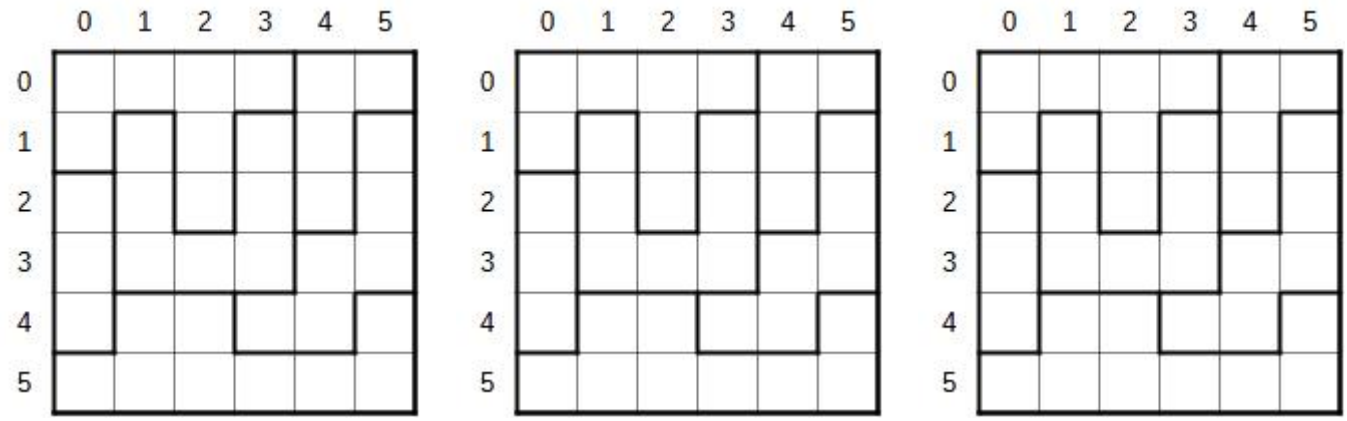
On numérote les zones de **0** à **n-1** en se déplaçant case par case "dans le sens de la lecture" en partant du coin supérieur gauche de la grille.
On inscrit **0** dans toutes les cases de la première zone, on inscrit **1** dans toutes les cases de la prochaine zone sans numéros, on continue avec les numéros suivants jusqu'à inscrire **n-1** dans la dernière zone (la zone **5** dans l'exemple).
Les premières cases de chaque zone "dans le sens de la lecture" sont en gras.
Dans les programmes, la grille est codée par un tableau **t[ ][ ]** des numéros de zones.
• **t[2][1]=3**, car la case en ligne 2 et colonne 1 (en gris) est dans la zone 3.
• **t[4][5]=1**, car la case en ligne 4 et colonne 5 (en gris aussi) est dans la zone 1.



Le joueur propose une solution à l'aide d'un tableau **col[ ]**.
Pour chaque numéro de ligne **i**, le joueur doit donner le numéro de la colonne **col[i]** où il veut placer une étoile dans cette ligne.
Dans l'exemple à droite, **col[ ]=[2,3,0,4,2,5]**
(notation abrégée pour **col[0]=2**, **col[1]=3**, **col[2]=0**, **col[3]=4**, **col[4]=2** et **col[5]=5**).
Cette proposition est très mauvaise car seule la règle 1 est respectée.
• La règle 2 n'est pas respectée car il y a plusieurs étoiles dans la colonne 2.
• La règle 3 n'est pas respectée car il y a plusieurs étoiles dans la zone 4.
• La règle 4 n'est pas respectée par les étoiles des lignes 0 et 1.



Les questions de cette page utilisent la grille de **Star Battle** affichée 3 fois ci-dessous.
Utilisez ces grilles pour chercher vos réponses au brouillon **avec un crayon et une gomme**.
N'oubliez pas de **recopier vos réponses sur les feuilles prévues**.



Q1(a) /4	**Que valent les éléments suivants du tableau t[ ][ ] pour cette grille ?** `t[1][5]=..., t[2][3]=..., t[3][0]=..., t[4][3]=...`

Pour les 4 questions suivantes, vous devez **cocher** les cases des règles qui sont **respectées** et *ne pas cocher* les cases des règles *non-respectées*. Vous marquez les points uniquement si vous donnez la bonne réponse pour les 4 règles .

Q1(b) /2	**Cochez les cases des règles qui sont respectées si col[ ]=[4, 1, 5, 2, 0, 3].** ☐ règle 1 ☐ règle 2 ☐ règle 3 ☐ règle 4

Q1(c) /2	**Cochez les cases des règles qui sont respectées si col[ ]=[2, 5, 4, 0, 3, 1].** ☐ règle 1 ☐ règle 2 ☐ règle 3 ☐ règle 4

Q1(d) /2	**Cochez les cases des règles qui sont respectées si col[ ]=[5, 3, 2, 0, 4, 1].** ☐ règle 1 ☐ règle 2 ☐ règle 3 ☐ règle 4

Q1(e) /2	**Cochez les cases des règles qui sont respectées si col[ ]=[3, 1, 4, 0, 3, 5].** ☐ règle 1 ☐ règle 2 ☐ règle 3 ☐ règle 4

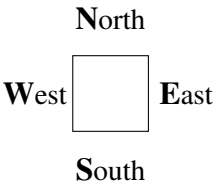
Trouvez une solution !

Q1(f) /2	**Placez 6 étoiles dans la grille en respectant les 4 règles.**

Q1(g) /2	**Que vaut le tableau col[ ] pour votre solution ?** `col[]=[..., ..., ..., ..., ..., ...]`

Vous devez compléter le **Programme SB1** ci-dessous qui sert à afficher à l'écran une grille de jeu, comme sur le premier exemple de la première page de l'énoncé.
Le programme dispose de la taille **n**, du tableau **t[ ][ ]** et du tableau **col[ ]**.

Le programme utilise un robot qu'on peut déplacer à l'écran et qui peut tracer des lignes, des murs et des étoiles. Le programme démarre avec le robot au bon endroit pour dessiner la case en haut à gauche de la grille, celle qui correspond à **t[0][0]**.
On dispose des instructions `Go()`, `line()`, `wall()` et `star()` expliquées ci-dessous.
`Go()`, `line()` et `wall()` ont besoin d'une direction W, E, N ou S.



- `Go(W,n)`, `Go(E,n)`, `Go(N,n)` et `Go(S,n)` sert à avancer de **n** cases dans la direction correspondante.
  Exemples: `Go(E,1)` déplace le robot d'une case vers la droite, `Go(S,2)` le déplace de 2 cases vers le bas.
- L'instruction `wall()` sert à tracer un mur sur un des bords de la case où il se trouve.
  `wall(W)` trace un mur vertical à gauche et `wall(E)` un mur vertical à droite.
  `wall(N)` trace un mur horizontal en haut et `wall(S)` un mur horizontal en bas.
  La grille et les différentes zones doivent être entourées par des murs.
- `line()` fonctionne comme `wall()` mais trace une fine ligne au lieu d'un mur.
  Il doit y avoir un mur ou une fine ligne sur chaque côté de chaque case.
- `star()` trace une étoile dans la cellule où se trouve le robot.

Si un test contient plusieurs conditions reliées par des **`or`**, le test est **`true`** si au moins une des conditions est **`true`**.
De plus, si la première condition d'un **`or`** est **`true`**, la deuxième condition n'est même pas évaluée (c'est inutile).

Q1(h) /10	**Complétez les ____ dans le Programme SB1.** **Note entre 0 et 10. Vous perdez 1 point par faute ou absence de réponse.**

Vous pouvez compléter ci-dessous au brouillon. N'oubliez pas de **recopier sur la feuille de réponses**.

```
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] ≠ ____________________ ) { wall(N) }
      else {line(N)}

      if (____________) {wall(S)}

      if (_________ or ____________________________) {wall(W)}
      else {line(W)}

      if (________________) {wall(____)}

      if (____________________) {star()}

      Go(E, 1)
   }
   Go(W, _____)
   Go(_____ , _____)
}
```

Vous devez compléter le **Programme SB2** ci-dessous qui sert à vérifier si une proposition du joueur vérifie les 4 règles. Le programme dispose de la taille **n**, du tableau **t[ ][ ]** et de la proposition dans le tableau **col[ ]**.

Le programme utilise un tableau **bcol[ ]** de **n** valeurs logiques (des Booléens). Au départ, ce tableau est rempli de valeurs **false** mais chaque fois qu'une étoile est placée dans une colonne, la valeur correspondante dans le tableau est changée en **true**. Pour tester si une étoile a déjà été placée dans la colonne 3 (par exemple), on peut utiliser le test `if (bcol[3]){...} else {...}` puisque **bcol[3]** a pour valeur **true** ou **false**.

Le programme utilise aussi un tableau **bzone[ ]** de **n** valeurs logiques qui joue le même rôle que **bcol[ ]** mais pour les zones au lieu des colonnes.

Le programme commence par vérifier la règle 2. Si plusieurs étoiles sont placées dans la même colonne, le programme s'arrête en renvoyant une valeur **false**. Ensuite, le programme vérifie la règle 3 et s'arrête en renvoyant **false** s'il y a plusieurs étoiles dans la même zone. Si les règles 2 et 3 sont vérifiées, le programme peut vérifier la règle 4 et s'arrête en renvoyant **false** si 2 étoiles se touchent. La dernière ligne s'exécute uniquement si les 4 règles sont respectées et le signale en retournant la valeur **true**.

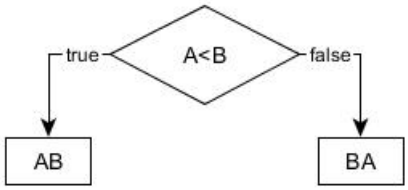
Q1(i) /10	**Complétez les _____ dans le Programme SB2.** **Note entre 0 et 10. Vous perdez 1 point par faute ou absence de réponse.**

Vous pouvez compléter ci-dessous au brouillon. N'oubliez pas de **recopier sur la feuille de réponses**.

```
Input  : n, t[][], col[]
Output : true or false
for (i ← 0 to n-1) {
   bcol[i] ← false
   bzone[i] ← false
}
//Rule 2 - Regle 2 - Regel 2
for (i ← 0 to _____ step 1) {
   j ← col[i]
   if (bcol[_____]) {return false}
   else {_____ ← true}
}
//Rule 3 - Regle 3 - Regel 3
for (i ← 0 to _____ step 1) {
   z ← _____
   if (bzone[_____]) {return false}
   else {_____ ← true}
}
//Rule 4 - Regle 4 - Regel 4
for (i ← 0 to _____ step 1) {
   if (col[i]=_____ or col[i]=_____){return false}
}
return true
```

Question 2 – Flowchart

On peut représenter un test par un losange dans lequel on écrit la condition à vérifier.
Selon le résultat, on suit soit la flèche **`true`** soit la flèche **`false`**.
Dans l'exemple ci-dessous, `A` et `B` sont des variables à comparer.
Si `A<B`, il faut afficher le message "**AB**", sinon il faut afficher "**BA**".
Les opérations d'affichage des messages sont représentées par les rectangles.

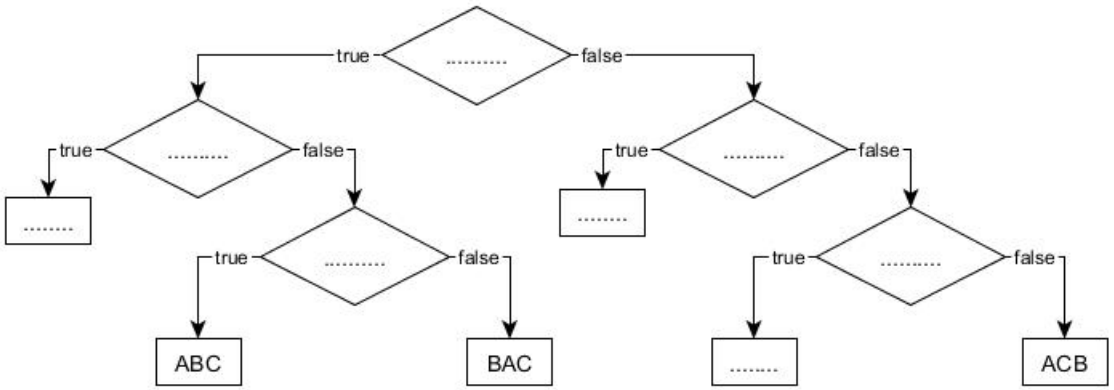


Par exemple, si `A=7` et `B=4`, l'algorithme ci-dessus suivra la flèche **`false`** du test et affichera "**BA**".

Passons à un problème similaire avec trois variables `A`, `B` et `C` qui ont des **valeurs différentes**.
Il faut représenter gaphiquement un algorithme qui affiche les 3 lettres "**A**", "**B**" et "**C**" dans l'ordre des valeurs des variables, de la plus petite à la plus grande. Par exemple, si `A=11`, `B=16` et `C=7`, l'algorithme doit afficher "**CAB**" puisque `C` a la plus petite valeur et `B` la plus grande.
Dans chaque losange vous devez placer un test. Vous pouvez uniquement utiliser les 6 tests suivants: `A<B`, `B<A`, `A<C`, `C<A`, `B<C` et `C<B`. Vous devez aussi écrire dans les rectangles avec des ... les 3 lettres dans le bon ordre.

Q2(a) /8	**Complétez les ... ci-dessous. Vous marquez un point par zone correctement remplie.**

Vous pouvez compléter sur cette feuille au brouillon. N'oubliez pas de **recopier sur la feuille de réponses**.



Supposons maintenant que les variables peuvent avoir des **valeurs identiques**.

Q2(b) /2	**Quel message s'affiche si les 3 variables sont égales ?**

Q2(c) /2	**Quels messages ne s'affichent jamais si 2 variables sont égales ?**

Question 3 – Croisement de listes

Il faut trouver tous les éléments communs à 2 listes de n éléments $L1[\,]$ et $L2[\,]$.
On sait que les éléments de $L1[\,]$ sont tous différents, de même pour les éléments de $L2[\,]$.
Il faut créer la liste $Li[\,]$ de tous les éléments qui sont à la fois dans $L1[\,]$ et dans $L2[\,]$.
Par exemple, si $n = 8$, $L1[\,] = [1, 3, 6, 10, 15, 21, 28, 36]$ et $L2[\,] = [3, 7, 15, 19, 23, 29, 36, 41]$ alors $Li[\,] = [3, 15, 36]$.
Comme d'habitude, on numérote les éléments à partir de 0.

Première idée : le programme P1.
Il utilise 2 boucles **`for`** imbriquées.
La première boucle sélectionne un nouvel élément $x1$ de $L1[\,]$ lors de chaque passage.
La deuxième boucle, à l'intérieur de la première, compare successivement $x1$ à tous les éléments de $L2[\,]$.
Si une égalité est trouvée, on ajoute $x1$ dans la liste $Li[\,]$.
P1 utilise `Li[]←[]` pour créer une liste $Li[\,]$ vide.
Il utilise la méthode `append()` pour **ajouter un élément à la fin d'une liste**.
Par exemple, si $Li[\,]$ est vide et qu'on exécute `Li[].append(3)` alors $Li[\,] = [3]$.
Ensuite, si on exécute `Li[].append(15)` puis `Li[].append(36)` alors $Li[\,] = [3, 15, 36]$.

Remarquez dans **P1** les lignes (1) et (2) mises en évidence avec un numéro dans un rectangle gris.

```
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	**Si $n = 100$, combien de fois sera exécutée la ligne (1) ?**

Q3(b) /1	**Si $n = 100$, combien de fois sera exécutée la ligne (2) ?**

Q3(c) /1	**De manière générale, pour un n donné, combien de fois sera exécutée la ligne (1) ?**

Q3(d) /1	**De manière générale, pour un n donné, combien de fois sera exécutée la ligne (2) ?**

Dans les questions qui suivent, on teste **P1** avec des listes de différentes longueurs.
On admet que le temps total d'exécution dépend uniquement du nombre de fois que la **ligne (2)** est exécutée (c'est approximatif, mais suffisament précis pour nous).

Q3(e) /1	**Si le temps d'exécution vaut 3 secondes pour un certain n, combien de temps, en *secondes*, faudra-t-il pour des listes 5 fois plus longues ?**

Q3(f) /1	**Si le temps d'exécution vaut 6 secondes pour un certain n, combien de temps, en *minutes*, faudra-t-il pour des listes 10 fois plus longues ?**

Q3(g) /1	**Si le temps d'exécution vaut 4 secondes pour un certain n, combien de temps, en *heures*, faudra-t-il pour des listes 60 fois plus longues ?**

Seconde idée : le programme P2.
P1 fonctionne avec n'importe quelles listes, qu'elles soient triées ou pas, mais il est lent.
P2 est plus rapide, mais il ne fonctionne que si les listes sont triées en ordre croissant.

Voici le listing de **P2** sans explications.
Remarquez la ligne (3) mise en évidence avec un numéro dans un rectangle gris.

```
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	**Si $n = 10$, combien de fois au MINimum sera exécutée la ligne (3) ?**

Q3(i) /1	**Si $n = 10$, combien de fois au MAXimum sera exécutée la ligne (3) ?**

Q3(j) /1	**Si $n = 10$ et $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$, donnez une liste $L2[\,]$ de nombres entiers > 0 différents telle que la ligne (3) soit exécutée un MINimum de fois.**

Q3(k) /1	**Si $n = 10$ et $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$, donnez une liste $L2[\,]$ de nombres entiers > 0 différents telle que la ligne (3) soit exécutée un MAXimum de fois.**

Q3(l) /1	**Pour un n donné, combien de fois au MINimum sera exécutée la ligne (3) ?**

Q3(m) /1	**Pour un n donné, combien de fois au MAXimum sera exécutée la ligne (3) ?**

Dans les questions qui suivent, on teste **P2** avec des listes de différentes longueurs.
On admet que le temps total d'exécution dépend uniquement du nombre de fois que la **ligne (3)** est exécutée (c'est approximatif, mais suffisament précis pour nous).

Q3(n) /1	**Si le temps d'exécution MAXimal vaut 3 secondes pour un certain n, quel sera le temps MAXimal, en $secondes$, pour des listes 5 fois plus longues ?**

Q3(o) /1	**Si le temps d'exécution MAXimal vaut 6 secondes pour un certain n, quel sera le temps MAXimal, en $minutes$, pour des listes 10 fois plus longues ?**

Q3(p) /1	**Si le temps d'exécution MAXimal vaut 4 secondes pour un certain n, quel sera le temps MAXimal, en $minutes$, pour des listes 60 fois plus longues ?**

Troisième idée : le programme P3.
P2 est bien plus rapide que **P1**, mais il faut que les listes soient triées.
Si les listes $L1[\,]$ et $L2[\,]$ ne sont pas triées, laquelle des 2 procédures suivantes est la plus rapide ?
Utiliser **P1** ou bien trier les 2 listes et ensuite utiliser **P2** ? Cela dépend de la vitesse à laquelle on peut trier !

P3 utilise la méthode `sort()` qui permet de trier une liste.
Par exemple, si $L[\,] = [36, 16, 4, 25, 9, 1]$ et qu'on exécute `L[].sort()` alors $L[\,] = [1, 4, 9, 16, 25, 36]$.

La méthode utilise un algorithme qui exécute environ $n \cdot \log_2(n)$ instructions pour trier une liste de n éléments.

Voici quelques exemples de valeurs de $\log_2(n)$ arrondies à l'entier le plus proche:
$\log_2(100) \simeq 7$ $\log_2(1000) \simeq 10$ $\log_2(1000000) \simeq 20$

P3 consiste à trier les listes $L1[\,]$ et $L2[\,]$, puis à utiliser **P2**.

Dans les questions suivantes, vous devez calculer le nombre approximatif d'instructions exécutées par **P3** avec des listes dont la longueur n est donnée.
Pour le calcul approximatif, on tiendra compte des 2 tris (en utilisant la formule sur fond gris et les valeurs fournies de $\log_2(n)$) et du nombre MAXimal de fois que l'instruction (3) de **P2** est exécutée.

Q3(q) /1	**Si $n = 100$, combien P3 exécutera-t-il d'instructions environ ?**

Q3(r) /1	**Si $n = 1000$, combien P3 exécutera-t-il d'instructions environ ?**

Q3(s) /1	**Si $n = 1000000$, combien P3 exécutera-t-il d'instructions environ ?**

Dans les questions suivantes, calculez le rapport $\dfrac{\text{temps d'exécution de P3}}{\text{temps d'exécution de P1}}$.

En pratique, vous devez utilisez les nombres d'instructions calculés dans les questions précédentes.
Exemple:
Si **P1** et **P3** exécutent respectivement 20000 et 300 instructions, vous calculez le rapport $\dfrac{300}{20000} = \dfrac{3}{200} = 0,015$.

Dans cet exemple, le temps d'exécution de **P3** vaut donc seulement $0,015 = 1,5\%$ du temps d'exécution de **P1**.
Donnez des réponses arrondies à 2 chiffres significatifs (par exemple, il faut arrondir $0,03681$ à $0,037$).

Q3(t) /1	**Si $n = 100$, quel est le rapport $\dfrac{\text{temps d'exécution de P3}}{\text{temps d'exécution de P1}}$?**

Q3(u) /1	**Si $n = 1000$, quel est le rapport $\dfrac{\text{temps d'exécution de P3}}{\text{temps d'exécution de P1}}$?**

Q3(v) /1	**Si $n = 1000000$, quel est le rapport $\dfrac{\text{temps d'exécution de P3}}{\text{temps d'exécution de P1}}$?**

Q3(w) /1	**Conclusion: si les listes sont non-triées, quel est le programme le plus rapide, P1 ou P3 ?**

Question 4 – Blocks

$L[\,]$ est une liste de n nombres $L[0], \ldots, L[n-1]$.
La **longueur** d'une liste est le nombre d'éléments qu'elle contient.
Un **bloc** de $L[\,]$ est une sous-liste, aussi longue que possible, d'éléments consécutifs et égaux entre eux de $L[\,]$.
Exemple: les blocs de $L[\,] = [4, 2, 2, 2, 1, 1, 2]$ sont $[4]$, $[2, 2, 2]$, $[1, 1]$ et $[2]$, respectivement de longueurs 1, 3, 2 et 1.

La **liste associée** à $L[\,]$ est la liste des longueurs des blocs de $L[\,]$.
Suite de l'exemple: la liste associée à $[4, 2, 2, 2, 1, 1, 2]$ est $[1, 3, 2, 1]$.

Q4(a) /1	**Quelle est la liste associée à** $[1, 2, 3, 4, 5]$ **?**

Q4(b) /1	**Quelle est la liste associée à** $[3, 3, 3, 3, 3]$ **?**

Q4(c) /2	**Quelle est la longueur de** $L[\,]$ **si sa liste associée est** $[3, 2, 2, 3]$ **?**

Q4(d) /2	**Quelle liste est égale à sa propre liste associée ?**

Dans toutes les questions suivantes, on considère des listes $L[\,]$ **consituées uniquement de** 1 **et de** 2 **et qui commencent par** 1.
Par exemple, $L[\,] = [1, 2, 2, 2, 1, 2]$ ou $L[\,] = [1, 1, 1, 1]$.
$A[\,]$ sera la liste associée à $L[\,]$.
Pour les 2 exemples précédents, on a respectivement $A[\,] = [1, 3, 1, 1]$ et $A[\,] = [4]$.

Q4(e) /2	**Que vaut** $A[\,]$ **si** $L[\,]$ **et** $A[\,]$ **sont de longueur** 6 **?**

Q4(f) /2	**Que vaut** $L[\,]$ **si** $L[\,]$ **et** $A[\,]$ **sont de longueur** 6 **?**

Dans les questions suivantes, $Z[\,]$ est la liste associée à $A[\,]$.
Par exemple, si $L[\,] = [1, 2, 2, 2, 1, 1, 1, 2, 2]$ alors $A[\,] = [1, 3, 3, 2]$ et $Z[\,] = [1, 2, 1]$.

Q4(g) /3	**Que vaut** $L[\,]$ **si elle a** 7 **éléments et si** $Z[\,] = [2, 3]$ **?**

Q4(h) /3	**Quelle est la plus longue liste** $L[\,]$ **telle que** $A[\,]$ **ne contient que des** 1 **et des** 2 **et** $Z[\,] = [1, 3, 1]$ **?**

Vous devez compléter le **Programme B1** ci-dessous qui prend comme entrées une liste $L[\,]$ et sa longueur n (où $n > 0$) et génère en sortie la liste $A[\,]$ associée à $L[\,]$.

Le programme utilise `A[]←[]` pour créer une liste $A[\,]$ vide.
Le programme utilise la méthode `append()` qui sert à **ajouter un élément à la fin d'une liste**.
Par exemple, si $A[\,]$ est vide et qu'on exécute `A[].append(3)` alors $A[\,] = [3]$.
Si ensuite on exécute `A[].append(1)` alors $A[\,] = [3, 1]$.

Q4(i) /6	**Complétez les ____ dans le Programme B1.** **Note entre 0 et 6. Vous perdez 1 point par faute ou absence de réponse.**

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

Vous pouvez compléter ci-dessus au brouillon. N'oubliez pas de **recopier sur la feuille de réponses**.

On peut également s'intéresser à certaines listes infinies.
Par exemple, la liste infinie $[1, 2, 1, 2, 1, 2, 1, 2, \ldots]$ a pour liste associée $[1, 1, 1, 1, 1, 1, 1, 1, \ldots]$ qui est également infinie.

Il existe une unique liste "magique" infinie ayant les trois propriétés suivantes:

- Elle commence par 1.
- Elle est consituée uniquement de 1 et de 2.
- Elle est égale à sa propre liste associée.

Nous appellerons $M[\,]$ cette liste magique.

Q4(j) /4	**Quels sont les 6 premiers termes de la liste magique $M[\,]$?**

Vous devez compléter le **Programme B2** ci-dessous qui génère les premiers éléments de la liste magique.
L'entrée de l'algorithme est un entier n, le nombre d'éléments que l'algorithme doit générer. La sortie est un tableau $M[\,]$ de longueur n contenant les n premiers éléments de la liste magique.

Le programme utilise `M[]←[1]` pour créer une liste $M[\,]$ qui contient un seul élément $M[0]$ qui vaut 1.
En plus de la méthode `append()` déjà expliquée, ce programme utilise la métode `pop()` qui sert à **supprimer le dernier élément d'une liste**.
Par exemple, si $L[\,] = [1, 2, 2, 4]$ et qu'on exécute `L[].pop()` alors $L[\,] = [1, 2, 2]$.
On doit parfois utiliser cette méthode à la fin du programme car il lui arrive, dans la boucle **`while`**, de générer un élément de plus que les n éléments demandés.

Q4(k) /8	**Complétez les ____ dans le Programme B2.** **Note entre 0 et 8. Vous perdez 2 points par faute ou absence de réponse.**

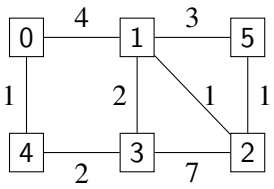
Vous pouvez compléter ci-dessous au brouillon. N'oubliez pas de **recopier sur la feuille de réponses**.

```
Input  : n
Output : M[]

M[] ← [1]
iL ← 1
iA ← 1
value ← 2

while (iL < n)
{
    M[].append(value)

    if (M[iA] = 2) {                    }

    iL ← iL +
    iA ← iA +
    value ←
}
if (iL = n+1) {M.pop()}
return M[]
```

Question 5 – Le voyage des Schtroumpfs

Les Schtroumpfs sont de grands voyageurs, c'est bien connu ! Ils doivent régulièrement partir en expédition, guidés par le Grand Schtroumpf. Mais ces voyages sont très longs (« C'est encore loin, Grand Schtroumpf ? ») et le Grand Schtroumpf perd un peu la tête avec ses 542 ans…

Le Schtroumpf informaticien décide de l'aider. Il fait appel au Schtroumpf géographe qui représente, dans la carte ci-dessous, les chemins connus dans la Forêt Magique et les temps nécessaires pour se rendre d'un point à un autre.



Les carrés représentent les différents lieux.
0 est le village, point de départ de tout voyage; 1 est le grand chêne; 2, la masure de Gargamel, *etc*.
Les lignes entre les lieux représentent les chemins.
Les nombres sur les chemins donnent le temps nécessaire pour se rendre d'un lieu à l'autre.
Par exemple, il faut 4 heures pour aller de 0 à 1.

Mais la carte du Schtroumpf géographe ne représente que les chemins *directs*.
Le Schtroumpf informaticien veut calculer l'*itinéraire* le plus rapide pour se rendre dans chaque lieu.
Un *itinéraire* est une séquence de lieux qu'on peut visiter en partant du village et en suivant les chemins directs.
Par exemple $0, 4, 3, 2$ est un itinéraire, mais $0, 3, 2$ n'en est pas un car il n'y a pas de chemin direct entre 0 et 3.
La *durée* d'un itinéraire est la somme des durées des chemins qui le composent.
Par exemple, la durée de l'itinéraire $0, 4, 3, 2$ est $1 + 2 + 7 = 10$ heures.

Q5(a) /1	**Quelle est la durée de l'itinéraire** $0, 1, 3, 2, 5$ **?**

Q5(b) /1	**Peut-on se rendre de** 0 **à** 5 **en exactement 7 heures ?** **Si votre réponse est "oui", indiquez l'itinéraire, sinon indiquez "non".**

Q5(c) /1	**Peut-on se rendre de** 0 **à** 5 **en moins de 5 heures ?** **Si votre réponse est "oui", indiquez l'itinéraire, sinon indiquez "non".**

Q5(d) /1	**Quel est l'itinéraire le plus rapide pour se rendre de** 0 **à** 5 **?**

Le Schtroumpf informaticien explique ci-dessous son algorithme pour calculer les durées des itinéraires les plus rapides.

Mon algorithme utilise un tableau `D[]`, indexé par les numéros des lieux. À la fin de l'algorithme, pour chaque lieu `x`, `D[x]` contiendra la durée de l'itinéraire le plus rapide entre le village et `x`.
Au départ, j'initialise (dans une boucle **`for`**) `D[x]` à $+\infty$ pour tous les lieux `x`.
Mais juste après, je modifie `D[0]` qui vaut 0 puisqu'on part toujours du village.

> Le symbole $+\infty$ signifie « valeur infinie ».
> C'est un peu l'équivalent mathématique de « C'est très loin mes petits Schtroumpfs ! ».
> C'est une valeur spéciale qui est considérée plus grande que n'importe quel nombre.
> En outre, quand on ajoute une valeur entière à $+\infty$, on obtient toujours $+\infty$.
> Par exemple, $+\infty + 5 = +\infty$

Durant son exécution, mon algorithme distingue 2 types de lieux.

- Les lieux pour lesquels on connaît déjà l'itinéraire le plus rapide depuis le village.
  Pour ces lieux, la valeur de `D[x]` ne changera plus jusqu'à la fin de l'exécution.
  On dira, pour simplifier, que ce sont des lieux *connus*.
- Les lieux pour lesquels on ne connaît pas encore l'itinéraire le plus rapide depuis le village.
  Pour ces lieux, la valeur de `D[x]` peut encore changer au cours de l'exécution.
  On dira, pour simplifier, que ce sont des lieux *inconnus*.

Au début, l'algorithme considère que tous les lieux sont *inconnus*.
À la fin, quand tous les itinéraires les plus rapides auront été calculés, tous les lieux seront *connus*.
En pratique, l'algorithme utilise un tableau de Booléens `K[]` tel que `K[x]` vaut **`true`** pour les lieux *connus* et **`false`** pour les lieux *inconnus*.

Quand on est certain que `D[x]` ne changera plus, cela veut dire que le lieu `x` devient *connu* et on retient cette information en mettant **`true`** dans `K[x]`.

Les opérations les plus importantes de l'algorithme ont lieu dans une boucle **`while`**.
Voici ce qui est effectué à chaque itération de cette boucle.

- Parmi tous les lieux *inconnus*, choisir le lieu `m` tel que `D[m]` est minimum.
- Faire passer `m` de lieu *inconnu* à lieu *connu*.
- Examiner tous les *successeurs* de `m` pour voir si je peux leur trouver un itinéraire plus rapide que celui que je connais déjà. Les successeurs de `m` sont tous les lieux auxquels on accède depuis `m` en suivant un chemin direct.
  Par exemple, les successeurs de 0 sont 1 et 4.
  Je regarde quelle est la durée `D[s]` de l'itinéraire le plus rapide connu pour chaque successeur `s` et j'essaie de l'améliorer à l'aide d'un itinéraire passant par `m`.

Enfin, il faut que je vous explique comment je représente la carte des chemins.
J'utilise un tableau `G[][]` tel que `G[i][j]` est la durée du chemin direct entre le lieu `i` et le lieu `j` si ce chemin existe.
S'il n'y a pas de chemin direct entre le lieu `i` et le lieu `j`, on l'indique en mettant $+\infty$ dans `G[i][j]`.
Par exemple, avec la carte du Schtroumpf géographe, on a `G[0][1]=4` et `G[0][3]=`$+\infty$.

L'algorithme utilise la fonction `minFalse(D[],K[])` qui renvoie le numéro (compris entre 0 et $n-1$) du lieu *inconnu* qui a la plus petite valeur dans `D[]`.
S'il y a plusieurs lieux *inconnus* ex-aequo, la fonction renvoie celui qui a le plus petit numéro.
Si tous les lieux sont connus, la fonction le signale en renvoyant -1.
Exemple : si les lieux encore inconnus sont 2, 4 et 5 et si `D[]=[0,7,9,15,17,9]`,
alors `minFalse(D[],K[])` cherche le minimum parmi `D[2]=9`, `D[4]=17` et `D[5]=9`. C'est 9.
Comme les lieux 2 et 5 sont ex-aequo, `minFalse(D[],K[])` renvoie le plus petit numéro, c'est-à-dire 2.

J'espère que vous avez bien schtroumpfé, pardon « compris », mes explications !

Voici l'algorithme du Schtroumpf informaticien :
Input : n, le nombre de lieux et G[][], la carte des chemins.
Output : D[], tel que D[m] est la durée de l'itinéraire le plus rapide de 0 à 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[]
```

Voici les premières étapes d'exécution de l'algorithme en utilisant le plan du Schtroumpf géographe.
Les tableaux représentent D[] et K[]. Pour raccourcir, on représente **true** par T et **false** par F.

- Initialisation:

	0	1	2	3	4	5
D:	0	+∞	+∞	+∞	+∞	+∞

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

- Durant le premier tour de boucle, l'algorithme sélectionne m= 0 et inspecte ses successeurs 1 et 4.
  On a G[0][1]= 4 et G[0][4]= 1.
  Puisque D[0]+G[0][1] = 0+4 = 4 < +∞, on met D[1] à jour avec cette valeur.
  De même, D[4] sera mis à jour et on aura:

	0	1	2	3	4	5
D:	0	4	+∞	+∞	1	+∞

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

Quelles sont les prochaines étapes que l'algorithme va calculer ?

Q5(e) /4	**Donnez les tableaux D[] et K[] à la fin de la seconde itération de la boucle while.**

Q5(f) /4	**Donnez les tableaux D[] et K[] à la fin de la troisième itération de la boucle while.**

Q5(g) /4	**Donnez les tableaux D[] et K[] à la fin de la quatrième itération de la boucle while.**

Le Schtroumpf informaticien a écrit la fonction `minFalse`, mais il a renversé de jus de salsepareille sur son listing et n'arrive plus à compléter les parties effacées. Pouvez-vous l'aider ?

Q5(h) /3	**Complétez les ▭ dans le listing de la fonction `minFalse`.**

Vous pouvez compléter ci-dessous au brouillon. N'oubliez pas de **recopier sur la feuille de réponses**.

Input : Le tableau `D[]` de taille `n` contenant des entiers ou $+\infty$.
Le tableau `K[]` de taille `n` contenant des Booléens.
Output : Le lieu `m`, tel que `D[m]` est minimum dans `D[]` parmi tous les lieux `i` tels que `K[i]`=**`false`**.
La valeur -1 s'il n'y a pas de lieu `i` tel que `K[i]`=**`false`**.

```
m ← ▭
minD ← +∞

for (i ← 0 to n-1 step 1) {
  if ((not K[i]) and (D[i] < ▭)) {
    m ← i
    minD ← ▭
  }
}
return m
```

Le Schtroumpf informaticien se pose maintenant des questions sur l'efficacité de son algorithme de calclul des itinéraires les plus rapides et sur les réponses qu'il pourrait renvoyer...

Q5(i) /2	**Avec la carte du Schtroumpf géographe, combien d'itérations de la boucle `while` l'algorithme va-t-il exécuter ?**

Q5(j) /2	**Si la carte possède n lieux et que la plus grande durée d'un chemin direct est M, combien d'itérations de la boucle `while` l'algorithme va-t-il exécuter ?**

Ensuite, le Schtroumpf informaticien montre son algorithme au Grand Schtroumpf.
Celui-ci lui dit: « C'est très bien mon petit Schtroumpf, mais ton algorithme n'est pas très utile s'il ne calcule que les *durées* des meilleurs itinéraires ! Je veux aussi savoir quels sont ces itinéraires ! ».

Le Schtroumpf informaticien se remet alors au travail et se rend compte qu'il doit calculer, pour chaque lieu `x`, le lieu `y` qui précède directement `x` dans le plus rapide itinéraire pour arriver en `x`. Il appelle `y` le *prédécesseur* de `x`.
Par exemple, l'itinéraire le plus rapide pour arriver en 3 est 0, 4, 3.
On doit donc retenir que le prédécesseur de 3 est 4 et que le prédécesseur de 4 est 0.
Le Schtroumpf informaticien modifie son algorithme en ajoutant un tableau `P[]` qui va servir à retenir, pour chaque lieu `x`, son prédécesseur `P[x]`.
Initialement, `P[x]` doit être égal à une valeur spéciale pour indiquer que rien n'a encore été calculé.
Le Schtroumpf informaticien choisit comme valeur spéciale -1.
Ensuite, il doit modifier la boucle **`while`** pour mettre à jour `P[x]`.

Sur base de ces informations, pouvez-vous l'aider à compléter les deux lignes manquantes dans son algorithme ?

Q5(k) /4	**Complétez les _____ dans l'algorithme du Schtroumpf informaticien de façon à calculer correctement le tableau `P[]`.**

Vous pouvez compléter ci-dessous au brouillon. N'oubliez pas de **recopier sur la feuille de réponses**.

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

Q5(l) /2	**Quel est le contenu du tableau `P[]` à la fin de l'exécution de l'algorithme sur la carte du Schtroumpf géographe ?**

Malheureusement, Gargamel, l'ennemi juré des Schtroumpfs, a placé des pièges à Schtroumpfs sur certains chemins. Les Schtroumpfs ne peuvent plus les emprunter et le Schtroumpf géographe a décidé d'effacer ces chemins de la carte. De ce fait, certains lieux peuvent maintenant être complètement inaccessibles depuis le village !
Le Schtroumpf informaticien se demande alors comment son algorithme va se comporter. Le Schtroumpf à lunettes, certain de tout comprendre à l'informatique, affirme une série de propositions. Lesquelles sont vraies ?

		Propositions du Schtroumpf à lunettes.
Q5(m) /1	Vrai / Faux	Il n'existe aucune carte pour laquelle l'algorithme renvoie un tableau `D[]` contenant une valeur $+\infty$.
Q5(n) /1	Vrai / Faux	Pour certaines cartes, l'algorithme peut renvoyer un tableau `D[]` contenant une ou plusieurs valeurs $+\infty$.
Q5(o) /1	Vrai / Faux	L'algorithme n'est pas correct dans le cas où il y a des lieux inaccessibles depuis le village (lieu 0). Dans ce cas il ne calcule pas toujours la durée du plus rapide itinéraire pour tous les lieux auxquels on peut accéder depuis le village.
Q5(p) /1	Vrai / Faux	L'algorithme renvoie un tableau `D[]` avec `D[x]`$=+\infty$ pour tous les lieux `x` tels que le plus rapide itinéraire pour atteindre `x` a une durée $\geq 2^n$, où n est le nombre de lieux.
Q5(q) /1	Vrai / Faux	L'algorithme renvoie un tableau `D[]` avec `D[x]`$=+\infty$ pour tous les lieux `x` tels qu'il n'existe pas d'itinéraire pour rejoindre `x` depuis le village (lieu 0).

Donnez vos réponses ici !

Q1(a) Valeurs d'éléments du tableau t[ ][ ]. /4

```
t[1][5]=..., t[2][3]=..., t[3][0]=..., t[4][3]=...
```

Q1(b) Cochez les cases des règles qui sont respectées si col[ ]=[4, 1, 5, 2, 0, 3]. /2

☐ règle 1 ☐ règle 2 ☐ règle 3 ☐ règle 4

Q1(c) Cochez les cases des règles qui sont respectées si col[ ]=[2, 5, 4, 0, 3, 1]. /2

☐ règle 1 ☐ règle 2 ☐ règle 3 ☐ règle 4

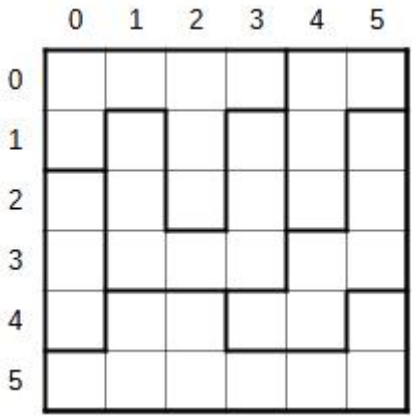
Q1(d) Cochez les cases des règles qui sont respectées si col[ ]=[5, 3, 2, 0, 4, 1]. /2

☐ règle 1 ☐ règle 2 ☐ règle 3 ☐ règle 4

Q1(e) Cochez les cases des règles qui sont respectées si col[ ]=[3, 1, 4, 0, 3, 5]. /2

☐ règle 1 ☐ règle 2 ☐ règle 3 ☐ règle 4

Q1(f) Placez 6 étoiles dans la grille en respectant les 4 règles. /2



Q1(g) Le tableau col[ ] de votre solution. /2

```
col[]=[..., ..., ..., ..., ..., ...]
```

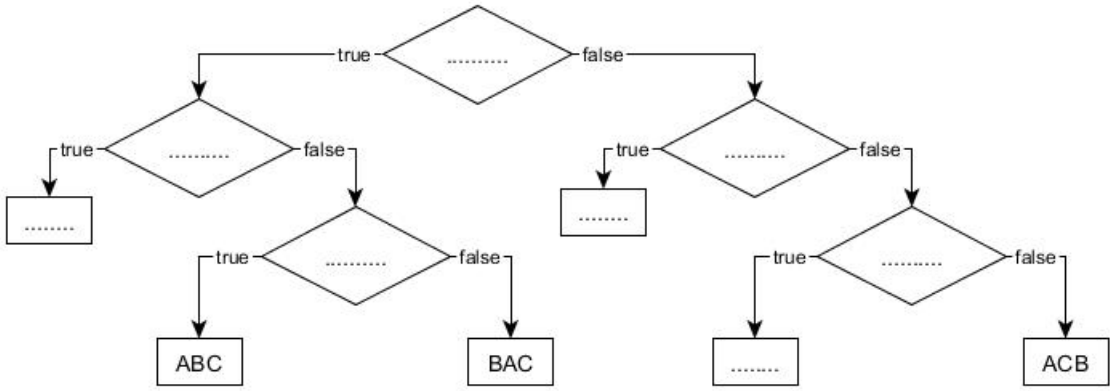
Q1(h) Complétez les ____ dans le Programme SB1. /10

```
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] ≠ ____________ ) { wall(N) }
      else {line(N)}

      if (____________) {wall(S)}

      if (_____ or ____________) {wall(W)}
      else {line(W)}

      if (____________) {wall(____)}

      if (____________) {star()}

      Go(E, 1)
   }
   Go(W, ____)
   Go(____ , ____)
}
```

Q1(i) Complétez les ____ dans le Programme SB2. /10

```
Input  : n, t[][], col[]
Output : true or false
for (i ← 0 to n-1) {
   bcol[i] ← false
   bzone[i] ← false
}
//Rule 2 - Regle 2 - Regel 2
for (i ← 0 to ____ step 1) {
   j ← col[i]
   if (bcol[____]) {return false}
   else {____________ ← true}
}
//Rule 3 - Regle 3 - Regel 3
for (i ← 0 to ____ step 1) {
   z ← ____________
   if (bzone[____________]) {return false}
   else {____________ ← true}
}
//Rule 4 - Regle 4 - Regel 4
for (i ← 0 to ____ step 1) {
   if (col[i]=____________ or col[i]=____________){return false}
}
return true
```

Q2(a) Complétez les . . . ci-dessous. Vous marquez un point par zone correctement remplie. /8



Q2(b) Quel message s'affiche si les 3 variables sont égales ? /2

. . . . . .

Q2(c) Quels messages ne s'affichent jamais si 2 variables sont égales ? /2

. . . . . .

Q3(a) Si $n = 100$, combien de fois sera exécutée la ligne (1) ? /1

. . . . . .

Q3(b) Si $n = 100$, combien de fois sera exécutée la ligne (2) ? /1

. . . . . .

Q3(c) De manière générale, pour un n donné, combien de fois sera exécutée la ligne (1) ? /1

. . . . . .

Q3(d) De manière générale, pour un n donné, combien de fois sera exécutée la ligne (2) ? /1

. . . . . .

Q3(e) Un nombre de secondes. /1

. . . . . .

Q3(f) Un nombre de minutes. /1

. . . . . .

Q3(g) Un nombre d'heures. /1

. . . . . .

Q3(h) Si $n = 10$, combien de fois au MINimum sera exécutée la ligne (3) ? /1

. . . . . .

Q3(i) Si $n = 10$, combien de fois au MAXimum sera exécutée la ligne (3) ? /1

. . . . . .

Q3(j) Si $n = 10$ et $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$, donnez une liste $L2[\,]$ de nombres entiers > 0 différents telle que la ligne (3) soit exécutée un MINimum de fois. /1

```
L2[] = [..., ..., ..., ..., ..., ..., ..., ..., ..., ...]
```

Q3(k) Si $n = 10$ et $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$, donnez une liste $L2[\,]$ de nombres entiers > 0 différents telle que la ligne (3) soit exécutée un MAXimum de fois. /1

```
L2[] = [..., ..., ..., ..., ..., ..., ..., ..., ..., ...]
```

Q3(l) Pour un n donné, combien de fois au MINimum sera exécutée la ligne (3) ? /1

. . . . . .

Q3(m) Pour un n donné, combien de fois au MAXimum sera exécutée la ligne (3) ?	/1
Q3(n) Un nombre de secondes.	/1
Q3(o) Un nombre de minutes.	/1
Q3(p) Un nombre de minutes.	/1
Q3(q) Si $n = 100$, combien P3 exécutera-t-il d'instructions environ ?	/1
Q3(r) Si $n = 1000$, combien P3 exécutera-t-il d'instructions environ ?	/1
Q3(s) Si $n = 1000000$, combien P3 exécutera-t-il d'instructions environ ?	/1
Q3(t) Si $n = 100$, quel est le rapport $\frac{\text{temps d'exécution de P3}}{\text{temps d'exécution de P1}}$?	/1
Q3(u) Si $n = 1000$, quel est le rapport $\frac{\text{temps d'exécution de P3}}{\text{temps d'exécution de P1}}$?	/1
Q3(v) Si $n = 1000000$, quel est le rapport $\frac{\text{temps d'exécution de P3}}{\text{temps d'exécution de P1}}$?	/1
Q3(w) Conclusion: si les listes sont non-triées, quel est le programme le plus rapide, P1 ou P3 ?	/1
Q4(a) Quelle est la liste associée à $[1, 2, 3, 4, 5]$?	/1
Q4(b) Quelle est la liste associée à $[3, 3, 3, 3, 3]$?	/1
Q4(c) Quelle est la longueur de $L[\,]$ si sa liste associée est $[3, 2, 2, 3]$?	/2
Q4(d) Quelle liste est égale à sa propre liste associée ?	/2
Q4(e) Que vaut $A[\,]$ si $L[\,]$ et $A[\,]$ sont de longueur 6 ? $A[\,] = [\ldots, \ldots, \ldots, \ldots, \ldots, \ldots]$	/2
Q4(f) Que vaut $L[\,]$ si $L[\,]$ et $A[\,]$ sont de longueur 6 ? $L[\,] = [1, \ldots, \ldots, \ldots, \ldots, \ldots]$	/2
Q4(g) Que vaut $L[\,]$ si elle a 7 éléments et si $Z[\,] = [2, 3]$? $L[\,] = [1, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots]$	/3

Question	Points
Q4(h) Une liste $L[\,]$. $L[\,] = [1, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots]$	/3

Q4(i) Complétez les ____ dans le Programme B1. /6

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

Question	Points
Q4(j) Quels sont les 6 premiers termes de la liste magique $M[\,]$?	/4

Q4(k) Complétez les ____ dans le Programme B2. /8

```
Input  : n
Output : M[]

M[] ← [1]
iL ← 1
iA ← 1
value ← 2

while (iL < n)
{
    M[].append(value)

    if (M[iA] = 2) {                    }

    iL ← iL + ____
    iA ← iA + ____
    value ← ____
}
if (iL = n+1) {M.pop()}
return M[]
```

Q5(a) Quelle est la durée de l'itinéraire 0, 1, 3, 2, 5 **?** /1

.....

Q5(b) Un itinéraire ou bien "non". /1

.....

Q5(c) Un itinéraire ou bien "non". /1

.....

Q5(d) Quel est l'itinéraire le plus rapide pour se rendre de 0 **à** 5 **?** /1

.....

Q5(e) Donnez les tableaux `D[]` et `K[]` à la fin de la seconde itération de la boucle `while`. /4

```
D[] = [..., ..., ..., ..., ..., ...]
K[] = [..., ..., ..., ..., ..., ...]
```

Q5(f) Donnez les tableaux `D[]` et `K[]` à la fin de la troisième itération de la boucle `while`. /4

```
D[] = [..., ..., ..., ..., ..., ...]
K[] = [..., ..., ..., ..., ..., ...]
```

Q5(g) Donnez les tableaux `D[]` et `K[]` à la fin de la quatrième itération de la boucle `while`. /4

```
D[] = [..., ..., ..., ..., ..., ...]
K[] = [..., ..., ..., ..., ..., ...]
```

Q5(h) Complétez les _____ dans le listing de la fonction `minFalse`. /3

Input : Le tableau `D[]` de taille `n` contenant des entiers ou $+\infty$.
Le tableau `K[]` de taille `n` contenant des Booléens.
Output : Le lieu `m`, tel que `D[m]` est minimum dans `D[]` parmi tous les lieux `i` tels que `K[i]=false`.
La valeur -1 s'il n'y a pas de lieu `i` tel que `K[i]=false`.

```
m ← _____
minD ← +∞

for (i ← 0 to n-1 step 1) {
  if ((not K[i]) and (D[i] < _____)) {
    m ← i
    minD ← _____
  }
}
return m
```

Q5(i) Un nombre entier. /2

.....

Q5(j) Une expression. /2

.....

Q5(k) Complétez les _____ dans l'algorithme. /4

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

Q5(l) Quel est le contenu du tableau `P[]` à la fin de l'exécution de l'algorithme sur la carte du Schtroumpf géographe ? /2

`P[] = [..., ..., ..., ..., ..., ...]`

	Vrai	Faux	Propositions du Schtroumpf à lunettes.	/5
Q5(m) /1	☐	☐	Il n'existe aucune carte pour laquelle l'algorithme renvoie un tableau `D[]` contenant une valeur $+\infty$.	
Q5(n) /1	☐	☐	Pour certaines cartes, l'algorithme peut renvoyer un tableau `D[]` contenant une ou plusieurs valeurs $+\infty$.	
Q5(o) /1	☐	☐	L'algorithme n'est pas correct dans le cas où il y a des lieux inaccessibles depuis le village (lieu 0). Dans ce cas il ne calcule pas toujours la durée du plus rapide itinéraire pour tous les lieux auxquels on peut accéder depuis le village.	
Q5(p) /1	☐	☐	L'algorithme renvoie un tableau `D[]` avec `D[x]`$=+\infty$ pour tous les lieux `x` tels que le plus rapide itinéraire pour atteindre `x` a une durée $\geq 2^n$, où n est le nombre de lieux.	
Q5(q) /1	☐	☐	L'algorithme renvoie un tableau `D[]` avec `D[x]`$=+\infty$ pour tous les lieux `x` tels qu'il n'existe pas d'itinéraire pour rejoindre `x` depuis le village (lieu 0).	