be-OI 2024 **Finale - CADET** samedi 23 mars 2024	**Remplissez ce cadre en MAJUSCULES et LISIBLEMENT svp** PRÉNOM : ..... NOM : ..... ÉCOLE :	O **Réservé**

Finale de l'Olympiade belge d'Informatique 2024 (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 ← 2007
age ← 2024 − 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.

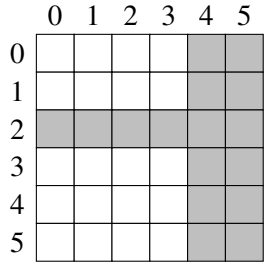
```
Input  : tab[ ], un tableau de n nombres.
         n, le nombre d'éléments du tableau.
Output : sum, la somme de tous les nombres contenus dans le tableau.

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 – Tests

Vous devez noircir les cases d'un tableau en évaluant un test logique portant sur les numéros de lignes et de colonnes des cases. La variable `i` contient un numéro de ligne et la variable `j` un numéro de colonne. Si le test est vrai pour un couple de valeurs `i` et `j`, alors il faut noircir la case à l'intersection de la ligne numéro `i` et de la colonne numéro `j`.

Voici un exemple de condition: `(i==2) or (j>3)`
La condition est vraie si le numéro de ligne vaut 2 ou si le numéro de colonne est supérieur à 3.
Les numéros des lignes sont notés à gauche et les numéros des colonnes sont notés au-dessus du tableau.
Il faut donc noircir les cases comme ci-dessous.



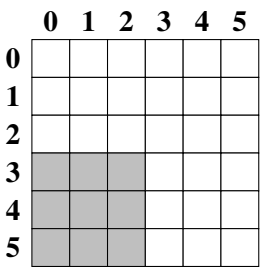
Les tests logiques utilisent les opérateurs de comparaison des exemples ci-dessous.

`i==1`	Vrai si `i` est égal à 1.
`j>3`	Vrai si `j` est plus grand que 3.
`j>=4`	Vrai si `j` est plus grand ou égal à 4.
`i+j<5`	Vrai si `i+j` est plus petit que 5.
`i<=j+2`	Vrai si `i` est plus petit ou égal à `j+2`.
`(i==3) or (j==2)`	Vrai si au moins une des 2 conditions est vraie.
`(i<2) and (j>3)`	Vrai si les 2 conditions sont vraies.

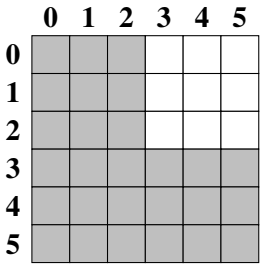
Certaines questions utilisent aussi les notions des exemples ci-dessous.

`max(14,18)`	Fonction maximum. Retourne la plus grande valeur, donc 18.
`min(i,j)`	Fonction minimum. Retourne la plus petite valeur.

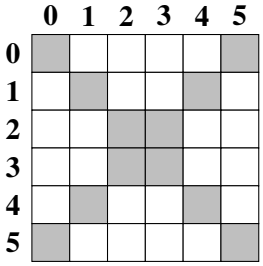
Les solutions sont affichées ci-dessous.

Q1(a) /2	**Noircir les cases qui vérifient la condition `(i>=3) and (j<3)`** 

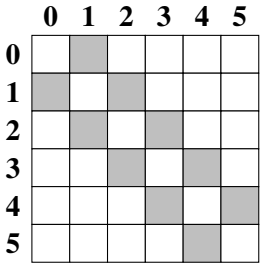
Q1(b) /2	**Noircir les cases qui vérifient la condition `(i>=3) or (j<3)`**



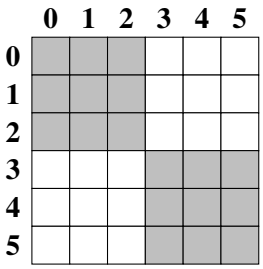
Q1(c) /4	**Noircir les cases qui vérifient la condition `(i==j) or (i+j==5)`**



Q1(d) /4	**Noircir les cases qui vérifient la condition `(i==j+1) or (i==j-1)`**



Q1(e) /4	**Noircir les cases qui vérifient la condition `min(i,j)>=3 or max(i,j)<=2`**

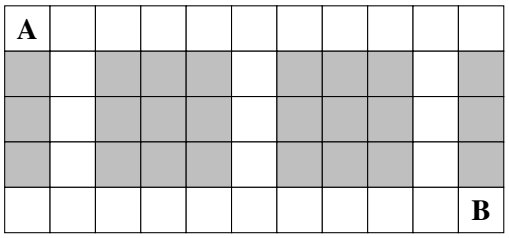


Question 2 – Chemins

Combien de chemins ?

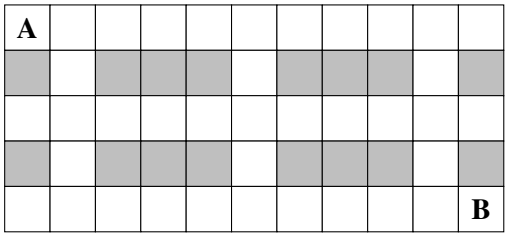
Les dessins ci-dessous montrent des salles avec des dalles carrées (cases blanches) et des obstacles (cases grises). Un robot part de **A** en haut à gauche et doit se déplacer jusqu'à **B** en bas à droite. À chaque étape, il peut seulement avancer d'une case vers le bas ou vers la droite. Il ne peut jamais aller vers la gauche, ni vers le haut. Le robot doit rester sur les cases blanches, il ne peut jamais passer sur une case grise. Dans chaque cas, combien de chemins différents le robot peut-il emprunter pour aller de **A** à **B** ?

Q2(a) /1 — **Combien de chemins différents le robot peut-il emprunter de A à B ?**



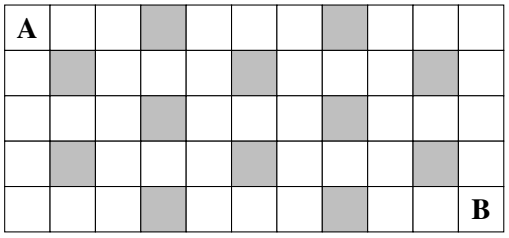
Solution: 3

Q2(b) /1 — **Combien de chemins différents le robot peut-il emprunter de A à B ?**



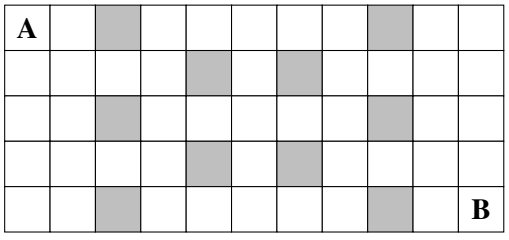
Solution: 6

Q2(c) /2 — **Combien de chemins différents le robot peut-il emprunter de A à B ?**



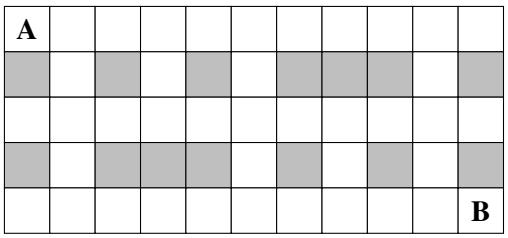
Solution: 1

Q2(d) /2 — **Combien de chemins différents le robot peut-il emprunter de A à B ?**



Solution: 4

Q2(e) /3	**Combien de chemins différents le robot peut-il emprunter de A à B ?**

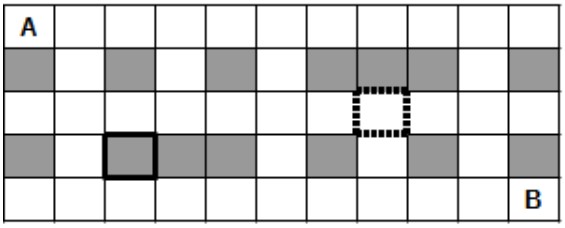


Solution: 11

Un programme pour compter les chemins

Il y a plusieurs façons de compter les chemins menant de **A** à **B**. Un algorithme DP (programmation dynamique) permet de le faire dans tous les cas et devient indispensable dans les cas difficiles.

Le listing incomplet d'un algorithme DP est donné plus loin. La salle y est réprésentée par un tableau `t[][]` contenant des **0** pour chaque case blanche et des **1** pour chaque case grise.



Salle

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

Tableau `t[][]` correspondant

Dans l'exemple ci-dessus, le tableau `t[][]` a 5 lignes et 11 colonnes. Les numéros des lignes sont notés à gauche du tableau et les numéros des colonnes sont notés au-dessus du tableau. Chaque case est repérée par son numéro de ligne et son numéro de colonne. La case entourée d'une bordure épaisse est grise, l'élément correspondant `t[3][2]` est donc égal à **1**. La case entourée de pointillés est blanche, l'élément correspondant `t[2][7]` est donc égal à **0**. Les cases **A** et **B** sont bien sûr blanches et représentées par des **0** dans `t[][]`.

L'algorithme DP calcule, pour chaque case, combien de chemins différents relient **A** à cette case. Le nombre de chemins pour atteindre la case **A** vaut **1**, puisqu'on s'y trouve au départ et qu'il est impossible d'y revenir par la suite. Le nombre de chemins vaut **0** pour les cases grises (puisqu'on ne peut jamais les atteindre). Le nombre de chemins pour atteindre une case blanche peut-être calculé à partir des nombres de chemins pour atteindre ses voisines de gauche et du dessus (c'est la propriété essentielle qui en fait un programme DP). Attention aux cases de la première colonne (qui n'ont pas de voisine de gauche) et de la première ligne (qui n'ont pas de voisine du dessus) !

Les nombres de chemins sont mémorisés dans un tableau `dp[][]` ayant les mêmes nombres de lignes et de colonnes que le tableau `t[][]`. Au début de l'algorithme, le tableau `dp[][]` est rempli de **0**. Le programme part de la case **A** et progresse dans le sens de la lecture (ligne par ligne, vers la droite dans chaque ligne). Avec la salle de l'exemple, après exécution de l'algorithme, `dp[3][2]` vaudra **0** car aucun chemin ne peut atteindre une case grise et `dp[2][7]` contiendra le nombre de chemins différents menant de la case **A** à la case entourée de pointillés.

Le programme démarre en disposant du nombre de lignes (il n'y en a pas toujours 5 !) dans la variable `ni`, du nombre de colonnes dans la variable `nj`, du tableau `t[][]` correctement rempli pour représenter la salle à analyser et du tableau

`dp[][]` rempli avec des **0**.
Le programme doit retourner le nombre de chemins différents menant de **A** à **B**.

Q2(f) /6	**Complétez les _____ dans le programme DP.** **Note entre 0 et 6. Vous perdez 1 point par faute ou absence de réponse.**
Solution: Les solutions sont affichées sur fond gris ci-dessous.	

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

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

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

Cases inaccessibles

Certaines cases blanches sont inaccessibles. Cela veut dire que le robot ne peut pas les atteindre en partant de **A** à cause des obstacles (cases grises). On suppose que le programme DP a été exécuté et on dispose donc du nombre de lignes dans la variable `ni`, du nombre de colonnes dans la variable `nj`, du tableau `t[][]` correctement rempli pour représenter la salle et du tableau `dp[][]` rempli par le programme DP.
Complétez le programme NoPath ci-dessous qui doit retourner le nombre de cases blanches inaccessibles.

Q2(g) /5	**Complétez les _____ dans le programme NoPath qui compte les cases blanches inaccessibles.** **Note entre 0 et 5. Vous perdez 1 point par faute ou absence de réponse.**
Solution: Les solutions sont affichées sur fond gris ci-dessous.	

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

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

Cases intermédiaires

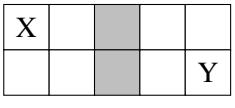
On veut compter le nombre de chemins passant par une case intermédiaire **X**. Le robot démarre toujours de **A** en haut à gauche et doit se déplacer jusqu'à **B** en bas à droite, mais il doit obligatoirement passer par **X**.

Salle 1: On sait qu'il y a 12 chemins différents entre **A** et **X** et 7 chemins différents entre **X** et **B**.

Q2(h) /1	**Dans la salle 1, combien de chemins relient A à B en passant par X ?**
Solution: $12 \cdot 7 = 84$	

Dans les questions suivantes, il y a 2 cases intermédiaires **X** et **Y**.
À chaque fois, on dispose d'un plan incomplet de la salle où figurent les cases **X** et **Y** et on connaît
N(A,X) le nombre de chemins différents entre **A** et **X**, **N(X,B)** le nombre de chemins différents entre **X** et **B**,
N(A,Y) le nombre de chemins différents entre **A** et **Y**, **N(Y,B)** le nombre de chemins différents entre **Y** et **B**.

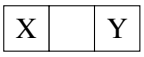
Salle 2: On sait que N(A,X)=5, N(X,B)=12, N(A,Y)=8, N(Y,B)=10 et on a le plan partiel



.

Q2(i) /2	**Dans la salle 2, combien de chemins relient A à B en passant par X ou par Y ?**
Solution: $(5 \cdot 12) + (8 \cdot 10) = 140$	

Salle 3: On sait que N(A,X)=6, N(X,B)=16, N(A,Y)=10, N(Y,B)=8 et on a le plan partiel



.

Q2(j) /1	**Dans la salle 3, combien de chemins relient A à B en passant par les 2 cases X et Y ?**
Solution: $6 \cdot 8 = 48$	

Q2(k) /2	**Dans la salle 3, combien de chemins relient A à B en passant par X ou Y (ou par les deux) ?**
Solution: $(6 \cdot 16) + (10 \cdot 8) - (6 \cdot 8) = 128$	

Question 3 – Loops

Considérons le programme **LoopA** ci-dessous. Il est constitué de 3 boucles **`for`** imbriquées et d'un test avec 3 conditions qui doivent être vraies pour exécuter une instruction d'affichage.

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

LoopA affiche tous les triplets de nombres entiers supérieurs à zéro dont la somme vaut 100.
Dans chaque triplet, les 3 nombres sont affichés en ordre croissant.
Ainsi **LoopA** affiche le triplet `(20,30,50)` mais pas le triplet `(50,20,30)`.

Q3(a) /1	**Quel est le premier triplet affiché ?**
Solution: $(1, 2, 97)$	

Q3(b) /1	**Quel est le dixième triplet affiché ?**
Solution: $(1, 11, 88)$	

Q3(c) /1	**Quel est le dernier triplet affiché ?**
Solution: $(32, 33, 35)$	

Q3(d) /1	**Combien de fois le test est-il évalué dans LoopA ?**
Solution: $100 \cdot 100 \cdot 100 = 1000000$ de fois	

Le programme **LoopB** affiche exactement la même chose que **LoopA**.
Mais il utilise un test plus simple: les conditions `i<j` et `j<k` ne sont plus nécessaires.

Q3(e) /3	**Complétez les ____ dans LoopB pour qu'il affiche la même chose que LoopA.** **Note entre 0 et 3. Vous perdez 1 point par faute ou absence de réponse.**
Solution: Les solutions sont affichées sur fond gris ci-dessous.	

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

Q3(f) /1	**Le test dans LoopB est-il évalué plus, moins ou autant de fois que celui de LoopA ?** **☐ Plus ☒ Moins ☐ Autant** **Le test est évalué 161700 fois dans LoopB.**

Q3(g) /1	**Le `print` dans LoopB est-il exécuté plus, moins ou autant de fois que celui de LoopA ?** **☐ Plus ☐ Moins ☒ Autant** **Les 2 programmes affichent exactement la même chose !**

Le programme **LoopC** affiche exactement la même chose que **LoopA** et **LoopB**.
Le test a encore changé et il n'y a plus que 2 boucles **`for`**.

Q3(h) /3	**Complétez les ____ dans LoopC pour qu'il affiche la même chose que LoopA et LoopB.** **Note entre 0 et 3. Vous perdez 1 point par faute ou absence de réponse.**
Solution: Les solutions sont affichées sur fond gris ci-dessous.	

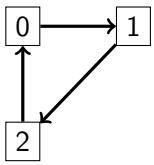
```
for (i ← 1 to 100 step 1) {
   for (j ← i+1 to 100 step 1) {
      k = 100-i-j
      if (j<k) {
         print(i,j,k)
      }
   }
}
```

Q3(i) /1	**Le test dans LoopC est-il évalué plus, moins ou autant de fois que celui de LoopB ?** **☐ Plus ☒ Moins ☐ Autant** **Le test est évalué 4950 fois dans LoopC.**

Question 4 – Teleportations

Le professeur Spock a mis au point plusieurs *réseaux téléporteurs*. Un tel réseau est formé de n cabines de téléportation, numérotées de 0 à $n-1$. Chaque cabine permet de se téléporter vers exactement une **autre** cabine (il s'agit encore de prototypes, chaque cabine ne permet pour l'instant d'accéder qu'à une destination fixée). Afin de procéder à des contrôles de sécurité, Spock souhaite vérifier que l'utilisation répétée des téléporteurs est sans danger. Pouvez-vous l'aider dans cette tâche ?

Le **premier réseau** mis au point par le professeur comporte seulement 3 cabines.



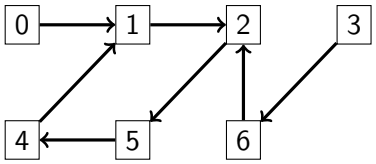
Le professeur teste son réseau en téléportant plusieurs fois de suite un objet placé au départ dans la cabine numéro 0. L'objet arrive à la cabine numéro 1 après une téléportation, à la cabine numéro 2 après deux téléportations, revient à la cabine numéro 0 après trois téléportations et ainsi de suite.

Q4(a) /1	**En partant de la cabine 0, dans quelle cabine arrive l'objet après 10 téléportations?**
Solution: 1	

Q4(b) /2	**En partant de la cabine 0, dans quelle cabine arrive l'objet après 50 téléportations?**
Solution: 2	

Q4(c) /2	**En partant de la cabine 0, dans quelle cabine arrive l'objet après 1000 téléportations?**
Solution: 1	

Les tests étant concluants, on passe au **second réseau**, comportant 7 cabines.

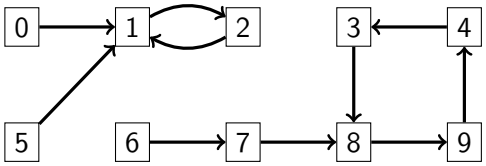


Q4(d) /2	**En partant de la cabine 4, dans quelle cabine arrive-t-on après 50 téléportations?**
Solution: 2	

Q4(e) /2	**En partant de la cabine 3, dans quelle cabine arrive-t-on après 1000 téléportations?**
Solution: 4	

Vous remarquez qu'en appliquant un grand nombre de téléportations, on finit toujours par revenir dans les mêmes cabines. Ainsi, dans le réseau précédent, en partant de la cabine numéro 0, on visite successivement les cabines 0, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4,$\cdots$. On dit qu'une cabine qui permet de revenir à elle-même après un certain nombre de téléportations *appartient à un cycle*. Un *cycle* est une liste de cabines où chaque cabine mène à la suivante et la dernière ramène à la première. La longueur d'un cycle est le nombre de cabines dans celui-ci. Le **second réseau** possède un cycle de longueur 4 formé des cabines $1, 2, 5, 4$.

Ci-dessous, le **troisième réseau** comporte 10 cabines et possède 2 cycles, un de longueur 2, l'autre de longueur 4.



Le professeur a prouvé que dans tous ses *réseaux téléporteurs*, où chaque cabine permet de se téléporter vers exactement une autre cabine, on arrive toujours dans un cycle après un certain nombre de téléportations, quelle que soit la cabine de départ.

Q4(f) /3	**Dans un réseau à `n` cabines, en partant de n'importe quelle cabine, après combien de téléportations est-on certain d'être arrivé dans un cycle ?**
Solution: `n-2`	

Il est utile de déterminer informatiquement le cycle dans lequel on arrive à partir d'une cabine donnée.
Un réseau de téléportation peut être représenté par un tableau `T[]` indiquant, pour chaque cabine, quelle est la cabine vers laquelle la téléportation s'opère. Ainsi, le tableau correspondant au **troisième réseau** formé de `n=10` cabines est `T=[1, 2, 1, 8, 3, 1, 7, 8, 9, 4]`.
Le cycle de longueur 2 se traduit par les relations `T[1]=2` et `T[2]=1`.
Le cycle de longueur 4 se traduit par les relations `T[3]=8`, `T[8]=9`, `T[9]=4` et `T[4]=3`.
Les cabines 0, 1, 2 et 5 permettent d'arriver au cycle de longueur 2.
Les autres cabines permettent d'arriver au cycle de longueur 4 (après 2 téléportations si on part de la cabine 6).

Vous devez compléter le **Programme A** ci-dessous. Il prend comme entrées un tableau `T[]` représentant le réseau à étudier, le nombre `n` de cabines et le numéro `c` de la cabine de départ.
Il génère en sortie un tableau de valeurs logiques (booléens) `cyc[]` de longueur `n`, tel que `cyc[i]` vaut **`true`** si la cabine numéro `i` appartient au cycle auquel on arrive en partant de la cabine `c`, **`false`** sinon.
Le programme utilise en interne un tableau `check[]` de `n` valeurs logiques.
Vous pouvez utiliser l'opérateur logique **`not`** (**`not true`** est égal à **`false`** et **`not false`** est égal à **`true`**).

Q4(g) /5	**Complétez les ____ dans le Programme A.** **Note entre 0 et 5. Vous perdez 1 point par faute ou absence de réponse.**
Solution: Les solutions sont affichées sur fond gris ci-dessous.	

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

Il est également important de connaître quelles cabines forment les cycles du réseau. Le **Programme B** doit remplir le tableau de booléens `cy[]` afin que `cy[i]` soit égal à **`true`** si la cabine numéro `i` appartient à un cycle et à **`false`** sinon. Le programme utilise en interne un tableau `pos[]` de `n` valeurs entières.

Q4(h) /4	**Complétez les ____ dans le Programme B.** **Note entre 0 et 4. Vous perdez 2 points par faute ou absence de réponse.**
Solution: Les solutions sont affichées sur fond gris ci-dessous.	

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

Les longueurs des cycles sont également importantes. Le **Programme C** doit remplir le tableau `lency[]` afin que `lency[i]` soit égal à 0 si la cabine numéro `i` n'appartient pas à un cycle et à la longueur de ce cycle sinon. Le **Programme C** utilise le tableau `cy[]` préalablement calculé par le **Programme B**.

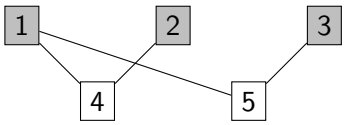
Q4(i) /5	**Complétez les ____ dans le Programme C.** **Note entre 0 et 5. Vous perdez 1 point par faute ou absence de réponse.**
Solution: Les solutions sont affichées sur fond gris ci-dessous.	

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

Question 5 – Quidditch

Le professeur Dumbledore est bien embêté: il doit organiser un match de Quidditch (le sport préféré des sorciers) et doit séparer les élèves en deux groupes qui lui permettront de constituer des équipes. Le problème est que de nombreux élèves ne s'entendent pas bien. Par exemple, Ron ne veut plus parler à Drago depuis que celui-ci a fait apparaître des limaces baveuses dans son lit.

Le professeur Dumbledore a décidé d'enregistrer les inimitiés des élèves dans un graphe comme ci-dessous. Il est composé de "nœuds" portant les noms des élèves (ici remplacés par des nombres pour plus de lisibilité) et de liens (appelés "arêtes") entre ces nœuds indiquant quand deux élèves ne s'entendent pas.



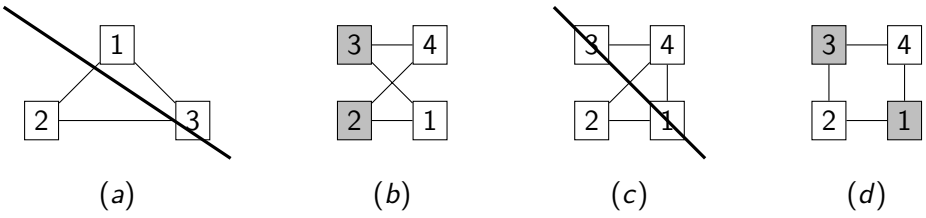
Premier exemple : un graphe à 5 nœuds et 4 arêtes.

On voit ainsi sur cet exemple que 1 et 4 ne s'entendent pas et ne peuvent donc pas être dans le même groupe. Par contre, 1 s'entend bien avec 2 et 3.

Dumbledore va donc séparer les élèves en deux groupes pour constituer deux équipes. Ces deux groupes ne doivent pas forcément être de même taille, mais **on ne peut pas avoir deux élèves qui ne s'entendent pas dans le même groupe**.

Sur l'exemple ci-dessus, il est facile de voir qu'on a deux groupes: $\{1, 2, 3\}$ et $\{4, 5\}$. Pour rendre cela plus visible, Dumbledore colorie les élèves du premier groupe en gris et laisse les autres en blanc.

Voici d'autres graphes. Pour chacun d'entre eux, on demande si on peut séparer leurs nœuds en deux groupes comme expliqué ci-dessus. Si c'est possible, coloriez les nœuds d'un des deux groupes. Si c'est impossible, barrez le graphe. Vous pouvez travailler ci-dessous au brouillon. N'oubliez pas de **recopier sur la feuille de réponses**.



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

Q5(a) /2	**Coloriez les nœuds d'un groupe dans le graphe (a) ou barrez le graphe si c'est impossible.**
Solution: Voir ci-dessus.	

Q5(b) /2	**Coloriez les nœuds d'un groupe dans le graphe (b) ou barrez le graphe si c'est impossible.**
Solution: Voir ci-dessus.	

Q5(c) /2	**Coloriez les nœuds d'un groupe dans le graphe (c) ou barrez le graphe si c'est impossible.**
Solution: Voir ci-dessus.	

Q5(d) /2	**Coloriez les nœuds d'un groupe dans le graphe (d) ou barrez le graphe si c'est impossible.**
Solution: Voir ci-dessus.	

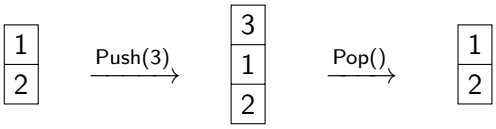
Le professeur Dumbledore veut maintenant trouver un algorithme pour faire cette répartition en groupes, autrement dit, colorier ses graphes de façon automatique.

Pour ce faire, il consulte son grimoire préféré: “De Algorithmica” écrit par le mage Cormenius (et ses collègues) au treizième siècle. Malheureusement, de l’encre indélébile magique a été renversée sur la page qui intéresse Dumbledore et des morceaux de l’algorithme sont manquants.

Dumbledore comprend qu’il aura besoin d’une structure de données appelée “pile”. Une pile permet de retenir des nœuds quand on parcourt un graphe, comme s’il s’agissait d’une pile d’assiettes, avec un nœud en bas de la pile, un autre au-dessus, *etc* jusqu’à atteindre le sommet de la pile. On peut réaliser les opérations suivantes sur une pile (voir exemple plus bas):

1. Push(x) ajoute un nœud x au sommet de la pile;
2. Pop() renvoie le nœud du sommet de la pile et l’enlève de la pile;
3. IsEmpty() renvoie **`true`** si la pile est vide et renvoie **`false`** sinon.

Voici un exemple où on part d’une pile avec deux éléments, on ajoute 3, puis on le retire (la dernière opération Pop() renvoie donc 3).



Q5(e) /1	**Supposons qu’on parte d’une pile vide. Donnez une séquence de** Push() **qui permet d’obtenir la pile:** 
Solution: Push(87), Push(11), Push(42)	

Q5(f) /1	**Supposons qu’on ait la pile**  **et qu’on applique la séquence d’opérations suivantes:** Pop(), Pop(), Push(24), Push(37), Push(71), Pop(), Pop()**. Quelle est la pile résultante ?**

Solution: 

Q5(g) /1	**Quelle est la valeur renvoyée par le dernier** Pop() **de la séquence de la question précédente ?**
Solution: le dernier Pop() renvoie 37.	

Le professeur Dumbledore peut maintenant étudier un premier algorithme de *parcours de graphe*.
Le graphe à parcourir est constitué de `n` nœuds et est décrit par le tableau de valeurs logiques (booléens) `Edge[][]`. `Edge[x][y]` et `Edge[y][x]` valent **`true`** s'il y a une arête entre les nœuds `x` et `y` et valent **`false`** sinon.
Avec le graphe du premier exemple, `Edge[1][4]` et `Edge[4][1]`, `Edge[1][5]` et `Edge[5][1]`, `Edge[2][4]` et `Edge[4][2]`, `Edge[3][5]` et `Edge[5][3]` valent **`true`**. Toutes les autres valeurs dans le tableau `Edge[][]` pour ce graphe sont égales à **`false`**.

L'algorithme reçoit le nombre de nœuds `n`, le tableau `Edge[][]` et un nœud initial `s` par où le parcours commence.
Il utilise un tableau `color[]` qui permet de *colorier* les nœuds au fur et à mesure du parcours.
Au début du programme on initialise `color[i]` à -1 pour tous les nœuds `i`.
Pour colorier un nœud `i`, on met `color[i]` à 0 et cette valeur ne changera plus par la suite.
L'algorithme utilise aussi une pile qui est initialement vide pour retenir des nœuds en attente de visite.
Remarque: contrairement à l'habitude en informatique, les indices commencent à 1 dans ce programme.

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

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

Push(s)
color[s] ← 0

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

Q5(h) /3	**Dans quel ordre l'algorithme colorie-t-il les nœuds du graphe du premier exemple (à 5 nœuds et 4 arêtes) si le nœud initial est `s=4` ?**
Solution: 4, 1, 2, 5, 3	

Le professeur Dumbledore essaie finalement de retrouver l'algorithme qui permet de séparer les nœuds du graphe en deux groupes.
Il comprend qu'il doit utiliser deux couleurs différentes, qui seront 0 et 1, correspondant aux deux groupes.
Il comprend bien que si un nœud a la couleur 0, tous les nœuds qui lui sont connectés par une arête doivent avoir la couleur 1 et vice-versa.
Pour l'aider dans cette tâche, le professeur Dumbledore utilise une fonction `invert` telle que `invert(0)=1` et `invert(1)=0`. Avec ces idées, pouvez-vous compléter l'algorithme ci-dessous ?
Il doit renvoyer un tableau `color[]` avec les couleurs de chaque nœud si la séparation est possible et renvoyer **`false`** si elle est impossible.

Q5(i) /6	**Complétez les ____ dans l'algorithme.** **Note entre 0 et 6. Vous perdez 2 points par faute ou absence de réponse.**
Solution: Les solutions sont affichées sur fond gris ci-dessous.	

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

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