Devoir d'informatique

Quelques remarques avant de commencer

Le sujet est constitué de quatre exercices indépendants. Les fonctions demandées doivent être écrites dans le langage C.

On prendra bien soin à veiller à la lisibilité du code proposé, en choisissant judicieusement les noms de variables utilisés, et en assortissant les fonctions de commentaires ou d'explications brèves mais pertinentes permettant de comprendre les choix effectués. Ces commentaires n'ont pas à être insérés dans le code, il est généralement préférable de décrire la fonction avant ou après le code pour des raisons de lisibilité.

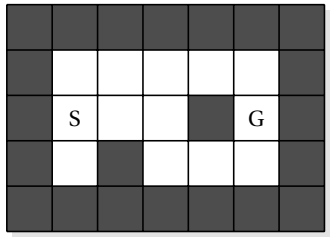
Si une question impose une complexité, votre proposition doit respecter cette complexité, ne perdez pas de temps à proposer une solution moins efficace, cela ne vous rapportera pas de points.

Vous pouvez introduire toutes les fonctions auxiliaires dont vous avez besoin. En langage C, la fonction `abs` est disponible (mais il n'en est pas de même pour `min` et `max`). Vous pouvez également utiliser dans une question toutes les fonctions décrites dans les questions précédentes du même problème, et ce *même si vous n'avez pas réussi à en proposer une implémentation.*

1 Labyrinthes

On s'intéresse à une grille, constituée de cases dites « accessibles » (représentées en blanc dans la suite) et « bloquées » (en noir), dans laquelle évolue un mobile. On peut imaginer le problème comme celui d'un robot évoluant dans une salle avec des murs et des obstacles, la grille représentant le problème vu de dessus, comme illustré ci-dessous. On suppose la grille entièrement close (la salle est entourée de murs).

Le mobile possède une orientation (initialement vers la droite). Il débute sur une case indiquée « S » et le but est de l'amener, par une succession de déplacements, sur une case marquée d'un « G ». **On suppose toujours la case « S » adjacente au mur de gauche, et la case marquée « G » adjacente au mur de droite, et qu'il existe toujours un chemin menant de l'une à l'autre.**



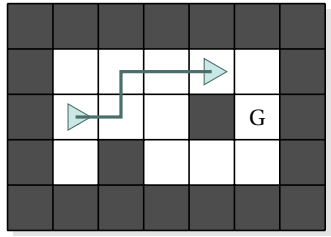
Le mouvement du mobile est gouverné par un programme C. Il est possible de lui donner des ordres grâce aux trois fonctions suivantes (fournies) :

- `void move(void)`, qui le fait avancer d'une case dans la direction dans laquelle il est orienté ;
- `void right(void)`, qui le fait pivoter *sur place* de 90° dans le sens horaire ;
- `void left(void)`, qui le fait pivoter *sur place* de 90° dans le sens direct.

Par exemple, la séquence d'appels

```
move();
left();
move();
right();
move();
move();
```

conduira aux déplacements suivants pour le mobile (on rappelle que le mobile part de la case marquée « S » avec une orientation vers la droite) :



Le mobile dispose par ailleurs de deux capteurs, dont l'état peut être lu grâce à deux fonctions booléennes (fournies également) :

- `bool goal(void)`, qui renvoie `true` si et seulement si le mobile se trouve sur la case marquée d'un « G » ;
- `bool wall(void)`, qui renvoie `true` si et seulement si la case qui se trouve immédiatement *devant* le mobile (en tenant compte de son orientation) est un obstacle (case noire).

Un exemple de programme contrôlant le mobile est proposé ci-dessous :

```
while (!goal()) {
    if (!wall()) {
        move();
    } else {
        right();
    }
}
```

1. On suppose que la grille est dans l'état représenté sur le document joint. Tracer sur cette grille le trajet du mobile exécutant le programme précédent depuis la case marquée « S » (on rappelle qu'il débute avec une orientation vers la droite) jusqu'à la case marquée « G », et indiquer en-dessous le nombre exact d'itérations effectuées par la boucle `while` (lequel revient ici au nombre d'appels à `wall`).

2. Le programme précédent termine-t-il toujours, quelle que soit la position des obstacles, pour une grille de cette taille et cette position des cases « S » et « G » ? Le justifier ou proposer une disposition

des obstacles pour laquelle le mobile n'atteint jamais son but avec le programme précédent (on noircira, sur le document joint, les cases nécessaires pour illustrer le placement des obstacles, et on tracera le comportement problématique du mobile). Le placement des obstacles est libre, mais on rappelle qu'il doit au moins exister un chemin menant au but.

Pour éviter certains problèmes, le choix est fréquemment fait, lorsque l'on a un changement de direction, de choisir le sens de la rotation au hasard. On fournit donc une fonction :

```c
void turn(void) {
    if (random() % 4 != 0) {
        right();
    } else {
        left()
    }
}
```

Lors d'un appel à `turn`, une rotation sur place de 90° est effectuée, trois fois sur quatre dans le sens horaire, et une fois sur quatre dans le sens direct. Bien évidemment, les programmes faisant appel à `turn` ne sont pas déterministes.

On propose le programme suivant :

```c
while (!goal()) {
    if (!wall()) {
        move();
    } else {
        turn();
    }
}
```

3. Pour les mêmes dimensions de la grille et les mêmes positions des cases « S » et « G », existe-t-il une disposition des obstacles telle que le mobile n'atteindra jamais la case marquée « G » avec le programme précédent? Proposer une telle disposition des obstacles sur le document joint (pas besoin, cette fois, de tracer le déplacement du mobile puisqu'il n'est pas déterministe), ou justifier qu'il n'en existe aucune.

Note : le document joint est à rendre avec la copie. Pour chaque question, vous disposez de deux représentations de la grille à compléter, afin de pouvoir recommencer en cas d'erreur (auquel cas, barrer clairement le premier essai infructueux).

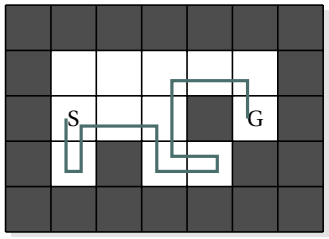
On modifie légèrement le programme :

```c
while (!goal()) {
    if (!wall()) {
        move();
    }
    turn();
}
```

4. Toujours pour les mêmes dimensions de la grille et les mêmes positions des cases « S » et « G », existe-t-il une disposition des obstacles telle que le mobile n'atteindra jamais la case marquée « G » avec le programme précédent? Proposer une telle disposition des obstacles sur le document joint ou justifier qu'il n'en existe aucune.

On souhaiterait disposer d'un programme qui garantisse que l'on atteigne l'objectif. Compte tenu du fait que les cases marquées « S » et « G » sont toutes les deux adjacentes à l'enceinte extérieure, il existe un algorithme qui le garantit (sous réserve qu'un chemin existe), bien connu des amateurs de labyrinthes (et, de ce qu'il m'a été donné d'entendre hier, d'au moins un étudiant de MP2I), qui consiste à toujours longer le mur à sa droite. Initialement, le mobile étant dirigé vers la droite, il peut ne pas avoir de mur à sa droite... Si c'est le cas, il devra pivoter sur sa droite (donc s'orienter vers le bas).

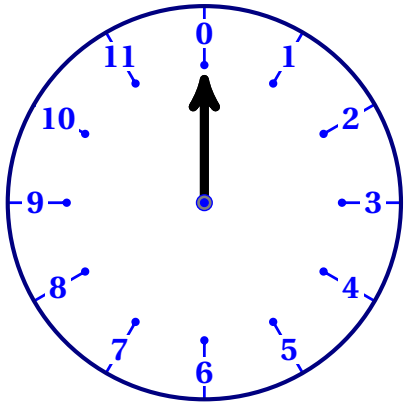
Ainsi, pour la disposition de grille ci-dessous, on cherche à obtenir le trajet suivant :



5. Proposer un programme C implémentant cette solution. Indication : le programme demandé peut s'écrire de manière à peine plus élaborée que les programmes précédents (typiquement deux fois plus long, avec deux tests et quatre ordres, et cela inclue la problématique de l'orientation initiale).

2 Horlogerie

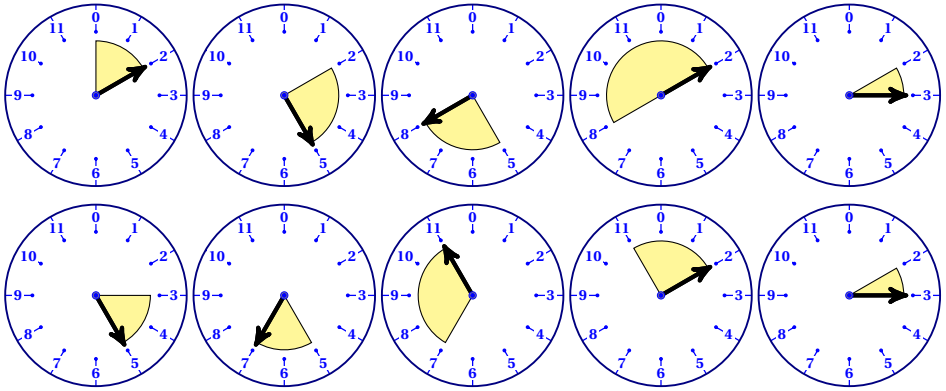
On s'intéresse à une « horloge », avec un cadran muni de douze graduations (numérotées de 0 à 11) régulièrement disposées, et d'une aiguille initialement en position verticale, dirigée vers la graduation 0, telle que celle représentée ci-dessous :



On dispose d'un tableau `arr` contenant des entiers entre 1 et 9 (inclus), et d'un entier `n` contenant le nombre de cases du tableau, par exemple :

arr	2	3	3	6	1	2	2	4	3	1

On considère les éléments du tableau comme des rotations, dans le sens indirect, de l'aiguille, du nombre de graduations correspondant. Par exemple, pour le tableau ci-dessus, les dix positions prises successivement par l'aiguille sont les suivantes :



Toutes les fonctions proposées doivent avoir une **complexité linéaire** en la taille du tableau. Il est permis d'utiliser, dans les fonctions, des tableaux à condition qu'ils soient de taille fixe (indépendante de n). On rappelle qu'il est attendu une explication (qui peut être très succinte si la fonction est simple), des fonctions proposées.

1. Proposer une fonction `int nb_revolutions(int arr[], int n)` renvoyant le nombre de tours *complets* que l'aiguille a effectué au total.

2. Proposer une fonction `int nb_horizontal(int arr[], int n)` renvoyant le nombre de fois où l'aiguille s'est arrêtée en position horizontale (y compris, éventuellement, la position finale).

3. Proposer une fonction `int last_stop_two(int arr[], int n)` renvoyant le numéro de l'étape à l'issue de laquelle l'aiguille s'est arrêtée sur le 2 pour la dernière fois.

4. Proposer une fonction `int most_common(int arr[], int n)` renvoyant la position sur laquelle l'aiguille s'est arrêtée le plus souvent (on ne prend pas en compte la position initiale, mais on prend en compte la position finale). En cas d'égalité, vous pouvez choisir la position que vous retournez parmi les plus fréquentes.

5. Proposer une fonction `int succ_same_parity(int arr[], int n)` renvoyant le plus grand nombre d'arrêts consécutifs sur des positions de même parité.

6. Proposer une fonction `int six_succ_distinct(int arr[], int n)` renvoyant le nombre de fois où l'aiguille s'est arrêté six fois de suite sur des positions différentes, ces séquences de six arrêts pouvant se recouvrir.

3 Partitions d'un entier

Soit un entier $n \in \mathbb{N}^*$ strictement positif. Une *partition* de n en p entiers est un ensemble (non ordonné) $\{a_i \in \mathbb{N}^*, i \in [\![1\,..\,p]\!]\}$ d'entiers strictement positifs tels que

$$\sum_{i=1}^{p} a_i = n$$

Par exemple, $\{5\}$, $\{4,1\}$, $\{2,2,1\}$ sont trois partitions de $n=5$ en respectivement $p=1$, $p=2$ et $p=3$ entiers. p est appelé *taille* de la partition.

On souhaite déterminer le nombre $\mathrm{P}(n,p)$ de partitions *distinctes* de n en p entiers. L'ordre des a_i n'ayant pas d'importance, $\{2,2,1\}$, $\{2,1,2\}$ et $\{1,2,2\}$ représentent une *même* partition de 5. Ainsi, $\mathrm{P}(5,3)=2$ car il n'existe que deux partitions de 5 en 3 entiers : $\{2,2,1\}$ et $\{3,1,1\}$.

1. Pour quelle(s) valeur(s) de n et p la valeur de $\mathrm{P}(n,p)=1$? Pour quelle(s) valeur(s) de n et p la valeur de $\mathrm{P}(n,p)=0$?

Pour déterminer $\mathrm{P}(n,p)$, on introduit $\mathrm{P}'(n,p)$, le nombre de partitions de n en *au plus* p entiers, soit

$$\mathrm{P}'(n,p) = \sum_{k=1}^{p} \mathrm{P}(n,k)$$

2. Que vaut $\mathrm{P}'(5,3)$?

3. Proposer une relation simple liant $\mathrm{P}(n,p)$ et $\mathrm{P}'(n,p)$ et un autre terme exprimé avec P.

Pour déterminer $\mathrm{P}(n,p)$, il nous faut une seconde relation. Pour ce faire, prenons une partition $\{a_i, i \in [\![1\,..\,p]\!]\}$ de n de taille p et considérons l'ensemble d'entiers $\{a_i - 1, i \in [\![1\,..\,p]\!]\}$.

4. Justifier que cet ensemble n'est pas nécessairement une partition d'un entier, mais que l'on peut aisément construire à partir de cet ensemble une partition d'un entier n' de taille p', en précisant ce que l'on sait des valeurs n' et p'.

5. En déduire une autre relation entre P et P′.

6. À partir de ces deux relations, trouver une relation de récurrence sur P de la forme

$$\mathrm{P}(n,p) = \mathrm{P}(\alpha,p) + \mathrm{P}(n,\beta)$$

où α est un entier strictement inférieur à n et β un entier strictement inférieur à p.

7. En déduire une fonction C `int P(int n, int p)` déterminant et renvoyant $\mathrm{P}(n,p)$.

Note : cette fonction peut nécessiter un temps de calcul considérable car elle effectue de nombreuses fois le même calcul. Nous verrons ultérieurement comment la programmation dynamique nous fournit des moyens efficaces de réécrire cette fonction.

4 Tri par cyles

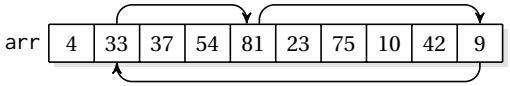
Il existe de nombreuses méthodes pour trier les éléments d'une liste. La plupart du temps, la principale préoccupation est de réduire le temps d'exécution du tri. Parfois, on souhaite un tri n'utilisant pas trop de mémoire supplémentaire. Il est plus rare que l'on cherche à réduire les écritures mémoires, mais cela peut arriver. Par exemple si l'on travaille avec des mémoires « solides » qui ont une durée de vie qui s'expriment en terme de nombre de modifications (comme les cartes mémoires par exemple, ou les disques durs SSD).

Nous allons voir dans ce sujet une méthode de tri intéressante du point de vue de ce dernier critère. On souhaite trier par ordre croissant les entiers contenus dans un tableau `tab` de taille `n`. **On suppose par ailleurs les entiers deux à deux distincts**.

Pour trier le tableau, lorsqu'un élément `arr[i]` n'est pas à sa place, on souhaite le déplacer à sa place définitive. L'ennui, c'est qu'un autre élément se trouve à la place en question (qui lui non plus n'est pas à sa place), et que l'on voudra déplacer pour le mettre, à son tour, à la bonne place.. Il s'ensuit un jeu de chaises musicales. Considérons par exemple le tableau suivant :

arr	4	33	37	54	81	23	75	10	42	9

Le second élément, 33, dans la case d'index 1, n'est pas à sa place. Il devrait se trouver dans la case d'index 4, où il chasse un élément (81) qui devrait se trouver dans la dernière case, dans laquelle se trouve l'élément 9 qui, lui, devrait se trouver dans la case d'index 1.



La remise en place de ces éléments conduit au tableau suivant :

arr	4	9	37	54	33	23	75	10	42	81

1. L'élément 37 dans la case d'index 2 n'est pas non plus à sa place. Déterminer, comme ci-dessus, la séquence de déplacements à effectuer pour le replacer, et représenter l'état du tableau après cette séquence de déplacements.

2. Proposer une fonction « int `nb_smaller(int arr[], int n, int elem)` » prenant en argument un tableau d'entiers `arr`, sa taille `n` et un entier `elem` et déterminant le nombre d'éléments dans le tableau `arr` strictement plus petits que `elem`.

3. Quelle est la complexité temporelle de la fonction `nb_smaller` ?

4. Justifier que la fonction précédente permet de déterminer dans quelle case devrait se trouver un élément `tab[i]` du tableau si le tableau était correctement trié.

On propose une ébauche de programme C effectuant un tri sur ce principe :

```c
for (int i=...; i<...; ++i) {
    // Invariant = ②
    int j = nb_smaller(arr, n, arr[i]);
    if (j != i) {
        int elem = arr[i];
        while (j != i) {
            int tmp = arr[j];
            arr[j] = elem;
            elem = tmp;
            // ① Mise à jour de j
        }
        arr[i] = elem;
    }
}
```

5. Déterminer comment doit être mis à jour l'index j à l'endroit identifié par ① dans le programme précédent.

6. Prouver (soigneusement) que la boucle while **`while`** termine.

7. Justifier qu'après l'instruction `arr[i] = elem`, l'élément `arr[i]` est bien placé dans le tableau.

8. En déduire un invariant de boucle à placer en ②, et en déduire ce qu'il convient de mettre comme paramètres dans la boucle **`for`**.

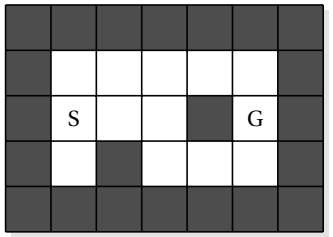
9. Quelle est la complexité temporelle de ce tri ? On justifiera soigneusement la réponse.

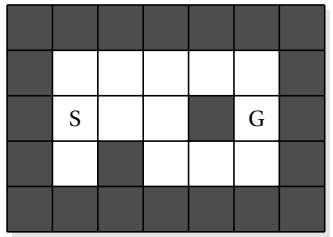
10. Combien effectue-t-on d'écritures dans le tableau dans le pire et dans le meilleur des cas ?

11. Expliquer, à l'aide d'un exemple, que le tri peut ne pas terminer si les éléments ne sont pas deux à deux distincts.

Document-réponse pour l'exercice 1

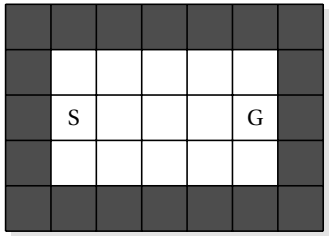
question 1

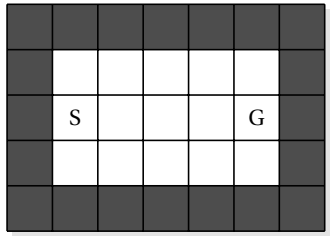




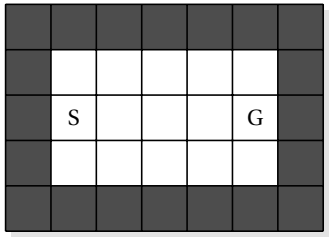
Nombre d'itérations :

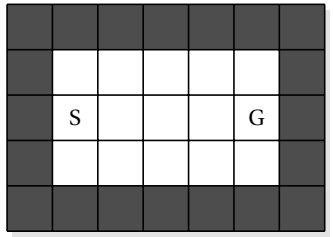
question 2 (s'il existe un cas problématique)





question 3 (s'il existe un cas problématique)





question 4 (s'il existe un cas problématique)