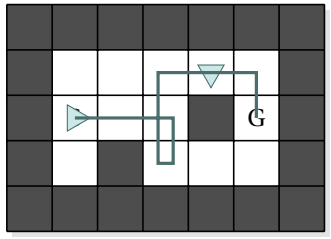
Devoir d'informatique

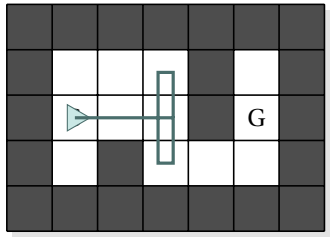
1 Labyrinthes

1. Le trajet du mobile sera le suivant :

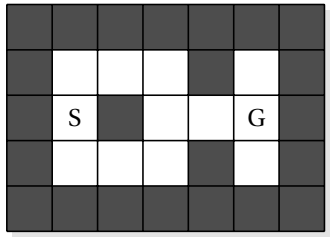


Puisque à chaque itération on effectue *soit* un déplacement d'une case, *soit* une rotation de 90° vers la droite, le nombre d'itérations de la boucle **`while`** correspond à la somme du nombre de déplacements (8) et du nombre de rotations de 90° (5), soit 13 itérations.

2. Non, le programme ne permet pas toujours d'atteindre le but, par exemple :



3. Là encore, il existe des dispositions d'obstacles pour lesquelles le mobile ne peut pas atteindre l'arrivée, quels que soient les résultats des tirages aléatoires. En particulier, il ne peut changer de direction que lorsqu'il est face à un mur, de sorte qu'il ne peut tourner à un embranchement au milieu d'un « couloir ». Par exemple, cette disposition ne lui permet pas d'atteindre le but :



4. Cette fois, aucune disposition ne permet d'empêcher le mobile d'atteindre le but. Une façon de le voir est qu'après une séquence `move()` (possiblement bloqué), `right()`, `move()` (possiblement bloqué), `right()`, `move()` (possiblement bloqué), `left()`, le mobile se trouve sur sa case initiale ou la case immédiatement en-dessous (selon la position des murs) et orienté vers le bas. À partir de là, on peut montrer qu'il peut toujours longer le mur extérieur, comme dans la question suivante.

Une autre façon de s'en persuadé est de montrer que s'il se trouve sur une case A, alors il peut toujours rejoindre une case B immédiatement voisine (disons, par exemple, à droite). En effet :

- s'il est orienté vers la droite (donc en direction de B), la séquence `move()`, `right()` l'amène en C ;
- s'il est orienté vers le haut et qu'il fait face à un mur, la séquence `move()` (bloqué), `right()`, `move()`, `right()` l'amène en B ;
- s'il est orienté vers le haut et que les cases au-dessus de A et au-dessus de B sont toutes les deux libres, la séquence `move()`, `right()`, `move()`, `right()`, `move()`, `left()` l'amène en B ;
- s'il est orienté vers le haut et que la case au-dessus de A est libre mais celle au-dessus de B est occupée (dernier cas possible pour cette orientation), la séquence `move()`, `right()`, `move()` (bloqué), `right()`, `move()`, `left()`, `move()`, `right()` l'amène en B ;
- s'il est orienté vers le bas, la sitution est symétrique d'un des trois cas précédent
- s'il est orienté vers la gauche et fait face à un mur, alors `move()` (bloqué), `right()` l'oriente vers le haut sur la même case, et on arrive à un cas déjà étudié ;
- s'il est orienté vers la gauche, qu'il ne fait pas face à un mur mais que la case au-dessus à gauche de A est occupée, `move()`, `right()`, `move()` (bloqué), `right()`, `move()`, `left()` a pour effet de l'amener sur la case A avec une orientation vers le haut, à nouveau ce même cas déjà traité ;
- s'il est orienté vers la gauche, qu'il ne fait pas face à un mur, que la case au-dessus à gauche de A est libre mais la case au-dessus de A est occupée, `move()`, `right()`, `move()`, `right()`, `move()` (bloqué), `right()`, `move()`, `left()`, `move()`, `left()` a pour effet de l'amener encore une fois sur la case A avec une orientation vers le haut ;
- enfin, s'il est orienté vers la gauche, et que les cases à gauche, au-dessus, et au-dessus à gauche de A sont toutes libres (dernier cas possible), `move()`, `right()`, `move()`, `right()`, `move()`, `right()`, `move()`, `left()`, `move()`, `right()` l'amène en B.

Donc pour n'*importe quel chemin* menant d'un point de départ à un point d'arrivée quelconque, on peut, à partir des neuf règles précédentes construire une séquence qui ménera le mobile au point d'arrivée (potentiellement en passant par des cases voisines de ce chemin). Bien entendu, comme c'est aléatoire, cela peut prendre un temps considérable. Mais on remarquera que le mobile ne peut jamais se « coincer », quelque soit l'endroit où il se trouve, il est toujours possible de rejoindre l'arrivée si c'était faisable au début.

On pouvait être plus succint, mais ce n'est pas une question triviale !

5. Par exemple (il existe de nombreuses solutions) :

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

Attention, en suivant le chemin indiqué, le mobile peut passer sur une case dont les quatre voisines sont libres. Dans cette situation, une instruction telle que

```
    while (!wall()) { right(); } // ou left()
```

causera une boucle infinie le mobile pivotant sans s'arrêter !

2 Horlogerie

1. On détermine la position après chaque déplacement (`pos` contient la position courante). Chaque dépassement de 12 compte pour une révolution complète (comptabilisés dans `nb`). Notons que cela ne peut arriver qu'une fois par déplacement, car on ne fait jamais plus de $\frac{3}{4}$ de tours par déplacement.

On peut parfaitement envisager de faire la somme des éléments du tableau et renvoyer le quotient obtenu avec une division entière par 12, même s'il est théoriquement possible que l'on ait un tableau suffisamment long pour que le programme provoque un débordement de capacité (si `INT_MAX` vaut $2^{31}-1$, il faut un tableau à $2^{31}/9$ cases, ce qui représente près d'un giga-octet de mémoire).

```
int nb_revolutions(int arr[], int n) {
    int pos = 0; // Position courante
    int nb = 0;  // Nombre de tours effectués
    for (int i=0; i<n; ++i) {
        pos = pos + arr[i];
        if (pos >= 12) {
            pos = pos - 12;
            nb++;
        }
    }
    return nb;
}
```

2. Même chose, mais on dénombre les situations où la position est 3 ou 9, ou plus simplement est congrue à 3 modulo 6 :

```
int nb_horizontal(int arr[], int n) {
    int pos = 0; // Position courante
    int nb = 0;  // Nombre d'arrêts sur 3 ou 9
    for (int i=0; i<n; ++i) {
        pos = (pos + arr[i]) % 12;
        if (pos % 6 == 3) { nb++; }
    }
    return nb;
}
```

Attention à bien lire le sujet : on parle d'*arrêts* sur une position horizontale, pas de simple *passage* sur une position horizontale (par exemple lorsque l'aiguille passe de 2 à 5 sans s'arrêter sur le 3). Quelques copies ont également confondu horizontale et verticale.

3. Encore une fonction similaire, mais on mémorise dans `last` le numéro de la case correspondant au dernier déplacement s'étant arrêté sur 2.

Il y a une ambiguïté sur le sens de « numéro de l'étape », on a choisi ici de l'interpréter par « après combien de déplacements », ce qui conduit à choisir `i+1` comme valeur à mémorisér (mais `i` aurait été accepté). Si par hasard l'aiguille ne s'arrête *jamais* sur 2, la fonction retourne -1.

Lorsque la question est ambiguë (même si en général on évite de telles questions qui rendent la correction bien plus délicate), on dispose d'un peu de latitude pour l'interpréter, mais il est toujours préférable de préciser quels choix on a fait.

```
int nb_horizontal(int arr[], int n) {
    int pos = 0;    // Position courante
    int last = -1; // Index du dernier arrêt sur 2, -1 si jamais
    for (int i=0; i<n; ++i) {
        pos = (pos + arr[i]) % 12;
        if (pos == 2) { last = i+1; }
    }
    return last;
}
```

4. On crée un tableau de taille 12, initialisé avec des 0 dans toutes les cases, afin de compter le nombre d'arrêts de l'aiguille sur chacune des positions (`count[i]` contiendra le nombre d'arrêts de l'aiguille sur la position `i`). Le décompte se fait avec une simple boucle comme précédemment, où à chaque arrêt de l'aiguille on incrémente la case adéquate de `count`.

Dans un second temps, on détermine et on retourne l'index du maximum (en prenant la

plus petite graduation en cas d'égalité, comme vu en cours :

```c
int nb_horizontal(int arr[], int n) {
    int pos = 0;
    int count[12] = {0};
    for (int i=0; i<n; ++i) {
        pos = (pos + arr[i]) % 12;
        count[pos]++
    }

    int i_best = 0;
    for (int i=1; i<12; ++i) {
        if (count[i] > count[i_best]) {
            i_best = i;
        }
    }
    return i_best;
}
```

5. La fonction peut vite être compliquée si on la prend de la mauvaise façon. On peut être malin : pour p arrêts successifs sur des graduations de même parité, cela signifie $p-1$ déplacements *de longueur paire*. Par exemple, la séquence $2 \triangleright 6 \triangleright 10$ correspond à deux déplacements de 4 et 2, et la séquence $3 \triangleright 7 \triangleright 11$ aux *mêmes* déplacements.

On adapte la fonction du cours déterminant la plus longue séquence de nombre pairs consécutifs. Notez que l'on débute avec $i = 1$ car on a choisi ici de ne pas compter la position initiale dans les positions successives, mais on pourrait faire un choix différent.

```c
int succ_same_parity(int arr[], int n) {
    int maxi = 1;
    int curr = 1;
    for (int i=1; i<n; ++i) {
        if (arr[i] % 2 == 0) {
            curr++;
            if (curr > maxi) {
                maxi = curr;
            }
        } else {
            curr = 1; // On a changé de parité
        }
    }
}
```

6. Pour cette dernière question, on peut envisager de nombreuses possibilités. Tout d'abord, remarquons que l'aiguille s'arrête deux fois sur la même position lorsque la somme des rotations entre les deux étapes est congrue à 0 modulo 12. On cherche donc les sous-tableaux contigus de longueur 5 ne contenant aucun sous-tableaux contigus dont la somme est nulle modulo 12.

Aucune case seule n'est congrue à 0 modulo 12, puisque les rotations individuelles sont entre 1 et 9, aussi il n'y a, pour chaque sous-tableau de taille 5, que 10 sommes à contrôler. Cependant, certains sous-tableaux sont considérés plusieurs fois, donc même si la complexité ne change pas, on peut faire un peu mieux.

Nous allons donc contrôler toutes les sommes de la forme

- `arr[i]+arr[i-1]`,
- `arr[i]+arr[i-1]+arr[i-2]`,
- `arr[i]+arr[i-1]+arr[i-2]+arr[i-3]`
- `arr[i]+arr[i-1]+arr[i-2]+arr[i-3]+arr[i-4]`.

et nous remarquerons que, si par exemple `arr[i]+arr[i-1]` est congrue à 0 modulo 12, les sous-tableaux se terminant en `i`, `i+1`, `i+2` et `i+3` ne conviennent tous pas.

Pour tous les `i`, on calcule donc les sommes précédentes (si elles ont du sens), et on maintient à jour une variable `min_i` qui contient le prochain index `i` pour la dernière case pour lequel on n'a pas trouvé de sommes problématiques. Ceci fait, on peut aisément compter les sous-tableaux de taille 5 qui conviennent. Cela donne :

```c
int six_succ_distinct(int arr[], int n) {
    int nb = 0, min_i = 5;
    for (int i=1; i<n, ++i) {
        int acc = arr[i];
        for (int j=1; j<5 && j<i && i+5-j>min_i; ++j) {
            acc = (acc + L[i-j]) % 12;
            if (acc == 0) { min_i = i+5-j; }
        }
        if (i >= min_i) { nb++; } // On n'a pas de problème pour
    }                             // le sous-tableau arr[i-4..i]
    return nb;
}
```

Notons que l'on a ici ignoré la position avant le premier déplacement (raison pour laquelle on part de `min_i = 5`)

Il existe de nombreuses autres possibilités : maintenir un tableau avec les six dernières positions (quitte à utiliser une fonction auxiliaire pour vérifier si elles sont différentes), ou douze compteurs dénombrant le nombre d'arrêt sur chaque position lors des six derniers arrêts... Il est important de ne pas faire d'hypothèses non présentes dans le sujet (en particulier, rien ne dit que $n \geqslant 6$) et de bien commenter la démarche et les variables introduites.

3 Partitions d'un entier

1. Pour $n > 0$, $\mathrm{P}(n,p) = 0$ lorsque $p \leqslant 0$ et $p > n$, sinon il y a toujours une solution au moins ($\{1,1,\ldots,1,n-p+1\}$)

Toujours pour $n > 0$, $\mathrm{P}(n,p) = 1$ lorsque $p = 1$ ($\{n\}$), $p = n$ ($\{1,1,\ldots,1\}$) et si $n > 1$, $p = n-1$ ($\{1,1,\ldots,1,2\}$). Pour les autres cas, il y a toujours au moins deux solutions (on peut envisager $\{2,2\}$ et $\{1,3\}$ et compléter avec ce qu'il faut).

2. $\mathrm{P}'(5,3) = 5$ (les partitions sont $\{5\}$, $\{4,1\}$, $\{3,2\}$, $\{3,1,1\}$, $\{2,2,1\}$).

3. On a, par définition de P et P′, simplement

$$\mathrm{P}'(n,p) = \mathrm{P}(n,p) + \mathrm{P}'(n,p-1)$$

4. Pour obtenir une partition de $n' = n-p$, il faut simplement retirer les zéros dans l'ensemble des $a_i - 1$ (on rappelle que les a_i sont non nuls). Il reste p' termes, avec $p' \leqslant p$.

5. On a donc $\mathrm{P}'(n-p,p) = \mathrm{P}(n,p)$

Pour se convaincre que qu'il y a bien une bijection entre les deux problèmes représentés par P et P′, on peut considérer les ensembles de a_i rangés par ordre décroissant. Une partition de n en exactement p entiers strictement positifs correspond à une partition de $n-p$ en $p-q$ entiers strictement positifs, où q correspond au nombre de 1 dans la décomposition de n. Puisque $q \leqslant p$, on a $0 \geqslant p-q \leqslant p$.

6. Il y a malheureusement eu dans le sujet une confusion entre P et P′. En combinant les relations précédentes, on obtient (lorsque $n > p$ et $p > 0$, pour $n \leqslant p$ et $p \leqslant 0$, on peut directement obtenir la valeur de $\mathrm{P}'(n,p)$)

$$\mathrm{P}'(n,p) = \mathrm{P}'(n-p,p) + \mathrm{P}'(n,p-1)$$

Cela conduit à la relation sur P (qui malheureusement ne correspond pas exactement à la forme demandée à cause du -1) :

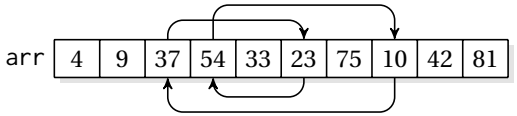
$$\mathrm{P}(n,p) = \mathrm{P}(n-p,p) + \mathrm{P}(n-1,p-1)$$

7. Cela permet de calculer $\mathrm{P}(n,p)$:

```c
int P(int n, int p) {
    if (p<=0 || p>n)  { return 0; }
    if (p==1 || p==n) { return 1; }
    return P(n-p, p) + P(n-1, p-1);
}
```

4 Tri par cyles

1. La séquence de déplacements partant de la case d'index 2 sera la suivante :



Elle conduit au tableau suivant :

arr	4	9	10	23	33	37	75	54	42	81

2. Il suffit d'utiliser un compteur et de parcourir le tableau :

```c
int nb_smaller(int arr[], int n, int elem) {
    int nb = 0;
    for (int i=0; i<n; ++i) {
        if (arr[i] < elem) {
            nb++;
        }
    }
    return nb;
}
```

3. La fonction effectue autant d'itérations qu'il y a d'éléments dans le tableau, *et le corps de la boucle a une complexité constante*[1], donc la fonction a une complexité temporelle linéaire ($O(n)$, ou même $\Theta(n)$).

4. Il suffit d'évaluer `nb_smaller(arr, n, arr[i])` pour trouver l'index de la case où devrait se trouver l'élément actuellement dans la case `arr[i]` : en effet, les éléments qui doivent se trouver avant l'élément `arr[i]` sont *exactement* les éléments plus petits que `arr[i]`, et la fonction permet de les compter, donc de dénombrer le nombre de cases qui se trouvent à gauche de la case qui doit accueillir l'élément `arr[i]`, ce qui correspond à l'index de cette même case.

5. Il y a ici une petite subtilité : l'élément `arr[i]` est présent *deux fois* dans le tableau à toute itération de la boucle! En revanche, `elem` ne s'y trouve plus. Par exemple, lorsque l'on arrive pour la première fois en ① dans le cycle proposé par l'énoncé, le tableau contient :

arr	4	33	37	54	**33**	23	75	10	42	9

Et `elem` contient 81. Si l'on compte les éléments plus petits que 81 dans le tableau, ils ne sont tous, donc on trouve 10. On ne peut pas placer `elem` dans `tab[10]`, une case qui n'existe pas! Il faut le placer dans `tab[9]`...

1. ce point est important si vous voulez être rigoureux... ici, c'est trivial, mais si on ne prend pas l'habitude de se poser la question, on l'oubliera lorsqu'une opération dans la boucle a un coût qui n'est pas constant.

Lors du second passage, `elem` vaut 9, et le tableau contient

tab	4	33	37	54	**33**	23	75	10	42	81

Cette fois-ci, compter les éléments plus petits que `elem` donne le bon index, 1 !

Par conséquent, si `elem < arr[i]`, on peut écrire `j = nb_smaller(arr, n, arr[i])` mais… si `elem > arr[i]`, il faut écrire `j = nb_plus_petits(arr, n, arr[i])-1` !

Les meilleurs programmeurs sont souvent ceux qui prêtent attention aux détails, et se méfient de leurs intuitions. Il est *toujours* intéressant d'essayer l'algorithme sur un exemple sur un morceau de papier.

Si cela peut vous consoler, *personne* n'a répondu correctement à cette question.

6. Chaque itération de la boucle **`while`** place dans la case `arr[j]` l'élément qui doit s'y trouver lorsque le tri se termine. L'élément que l'on extrait de cette même case `arr[i]` n'était donc pas bien placé puisqu'ils sont tous différents. Chaque écriture dans le tableau augmente donc le nombre d'éléments bien placés.

On dispose ici d'un *variant* de boucle : le nombre d'éléments bien placés est entier, majoré par n et augmente à chaque itération de la boucle **`while`**. Donc elle se termine nécessairement !

7. Au début et à la fin du corps de la boucle **`while`**, `j` désigne l'index de la case où doit se trouver l'élément `elem`. Lorsque l'on sort de la boucle **`while`**, on a `i=j`, donc `elem` devrait se trouver dans la case d'index `i`.

8. Au début de la boucle, en ②, on peut être certain que « les `i` premiers éléments du tableau sont bien placés » (on évitera d'utiliser le terme « triés » car le début du tableau peut être trié sans pour autant que les éléments soient bien placés… voir l'invariant du tri par insertion du cours).

9. La question n'est pas triviale. on observe une boucle **`while`** (pouvant s'exécuter n fois), dans une boucle **`for`** (qui va s'exécuter n fois), et le corps de la boucle **`while`** contient une fonction, `nb_smaller`, de complexité $\Theta(()n)$. Une analyse rapide concluera $O(n^3)$.

Mais à bien y regarder, à chaque appel à `nb_smaller` dans la boucle **`while`** correspond un élément supplémentaire bien placé dans le tableau, et la fonction ne sera donc pas appelée plus de deux fois pour le même élément (la première fois pour le placer durant l'exploration d'un cycle, la seconde éventuellement plus tard pour contrôler qu'il l'est déjà). Le nombre *total* d'itérations de la boucle **`while`** (pour l'ensemble des itérations de la boucle **`for`**) est donc $O(n)$.

Le tri a donc une complexité quadratique ($O(n^2)$).

10. Dans le meilleur des cas, tous les éléments sont bien placés, et on n'effectue aucune écriture. Dans le pire des cas, le contenu de chaque case est modifié une seule fois, pour mettre le bon élément, et on effectue n écritures.

11. On peut le montrer à partir d'un simple tableau à trois cases tel que `{ 0, 0, 42 }`. La première itération du **`for`** n'entrera pas dans le **`while`**, mais la seconde va créer une boucle infinie où j sera toujours nul, et ne vaudra donc jamais i. La présence d'une troisième valeur est ici nécessaire car le **`for`** s'arrête à la case `arr[n-2]`.

Précisons que si l'on avait utilisé `<=` dans `nb_smaller`, la boucle infinie serait survenue dès la première itération du **`for`**, avec `j` bloqué sur 1.

Résultats