Devoir d'informatique

Quelques remarques avant de commencer

Le sujet est constitué de trois problèmes 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.

Vous pouvez introduire toutes les fonctions auxiliaires dont vous avez besoin. 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.*

Si d'aventure vous trouvez ce que vous pensez être une erreur dans le sujet, indiquez-le sur votre copie, en précisant les choix que vous avez fait pour la contourner.

1 Collection

À l'occasion du dernier événement sportif en date, l'entreprise Fajitas met en vente un ensemble de cartes de collection. Il y a, au total, $n = 500$ cartes différentes à collectionner, et celles-ci sont vendues dans des paquets « aveugles » contenant chacun $p = 20$ cartes. Dans chaque paquet, l'assortiment des cartes est aléatoire, mais toutes les cartes d'un même paquet sont différentes. On suppose qu'il n'y a aucune carte plus rare qu'une autre, de sorte qu'une carte donnée a une probabilité p/n de se retrouver dans un paquet donné.

Un acheteur potentiel souhaite réunir une collection complète des n cartes, et pour ce faire, on envisage d'acquérir successivement des paquets, un par un, jusqu'à y parvenir. Pour estimer le coût de l'opération, il souhaite effectuer une simulation.

La probabilité d'obtenir k nouvelles cartes, en supposant que l'on en possède déjà q, dans les conditions décrites précédemment, s'écrit :

$$\mathscr{P}(k,q,n,p) = \frac{\binom{n-q}{k} \times \binom{q}{p-k}}{\binom{n}{p}}$$

On fournit une fonction `double comb_d(int a, int b)` qui retourne une approximation flottante de $\binom{b}{a}$ (compte tenu des valeurs de `a` et `b` qui seront manipulées, $\binom{b}{a}$ risque de dépasser les capacités des entiers, on admettra que les calculs sur des flottants n'influent pas notablement les résultats que l'on obtiendra). On pourra supposer que la fonction retourne 0 si $a < 0$ ou $a > b$.

1. Proposer une fonction `double* prob(int q, int n, int p)` retournant un tableau `pr` de $p+1$ flottants (`double`) tel que `pr[k]` corresponde à la probabilité d'obtenir `k` nouvelles cartes en achetant un paquet de `p` cartes alors que l'on possède déjà `q` cartes différentes parmi les `n` possibles. Le tableau retourné par la fonction sera alloué dynamiquement, et la libération de la mémoire sera à la charge de la fonction appelante.

2. Justifier que la somme des termes du tableau retourné est égal à 1.0 (on négligera les éventuelles erreurs d'arrondis dues aux flottants).

On fournit une fonction `double rand_d()` retournant un flottant dans l'intervalle $[0,1]$ avec une distribution uniforme.

3. Proposer une fonction `int tirage(double pr[], int p)` qui prend en argument un tableau de $p+1$ probabilités dont la somme vaut 1.0 et retourne un entier k tiré au sort dans $[\![0\,..\,p]\!]$ de façon à ce que l'entier k soit retourné avec une probabilité `pr[k]`. Par exemple, pour $p = 4$ et le tableau `pr` suivant :

pr	0.2	0.1	0.5	0.15	0.05

la fonction devra retourner, statistiquement, « 0 » 20% du temps, « 1 » 10% du temps, « 2 » 50% du temps, « 3 » 15% du temps et « 4 » 5% du temps.

4. Proposer une fonction `int nb_paquets(int p, int n)`, prenant en argument le nombre de cartes par paquet et le nombre de cartes constituant une collection complète, et qui, à l'aide des fonctions précédentes, effectue une simulation (on parlera de méthode de *Monte-Carlo*) d'un client qui achète des paquets tant qu'il n'a pas obtenu une collection complète, et retourne le nombre de paquets qui ont été achetés.

Pour avoir une idée plus précise de l'investissement nécessaire, on pourra effectuer plusieurs simulations avec la fonction précédente et regarder la distribution des résultats.

Alternativement, on peut chercher à déterminer la probabilité de posséder q cartes différentes après l'ouverture d'un nombre r donné de paquets. Pour ce faire, on propose l'approche suivante : on suppose disposer d'un tableau `arr` de $n+1$ flottants tel que, après l'ouverture d'un nombre donné r de paquets, `arr[q]` contient la probabilité (dans $[0,1]$) de posséder q cartes différentes.

5. Proposer une fonction `void ajoute_paquet(double arr[], int n, int p)` qui prend en argument un tel tableau de taille `n+1` contenant les probabilités après l'ouverture de r paquets, le nombre total `n` de cartes à collectionner, et le nombre `p` de cartes par paquets, ne retourne rien, mais mets à jour le contenu du tableau `arr` de sorte qu'il contienne, dans chaque case `arr[q]`, la probabilité de posséder q cartes différentes après l'achat de $r+1$ paquets (soit un paquet supplémentaire).

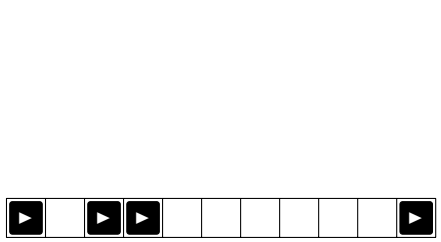
6. En déduire une fonction `double* probabilites(int n, int p, int r)` retournant un tableau de taille $n+1$ contenant, dans chaque case `arr[q]`, la probabilité de posséder `q` cartes différentes après l'ouverture de `r` paquets contenant chacun `p` cartes

différentes.

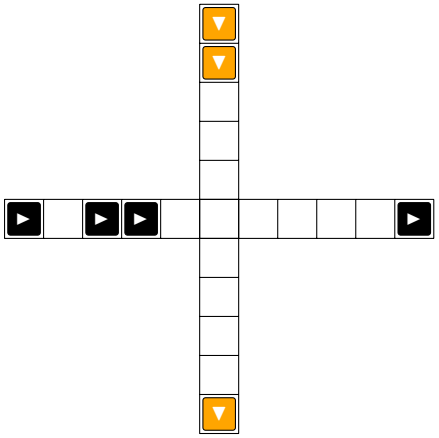
2 Circulation routière

2.1 Introduction

Ce problème étudie la conception d'un logiciel d'étude de trafic routier. On modélise le déplacement d'un ensemble de voitures sur des files à sens unique (voir figure ci-dessous). C'est un schéma simple qui peut permettre de comprendre l'apparition d'embouteillages et de concevoir des solutions pour fluidifier le trafic.



(a) Représentation d'une file de longueur onze comprenant quatre voitures, situées respectivement sur les cases d'indices 0, 2, 3 et 10.



(b) Configuration représentant deux files de circulation à sens unique se croisant en une case.

2.2 Préliminaires

Dans un premier temps, on considère une seule file, comme sur la figure ci-dessous.

Une file de longueur n est assimilée à un tableau de n cases. Une case peut contenir au plus une voiture. Les voitures présentes dans une file circulent toutes dans la même direction (sens des indices croissants, désigné par les flèches sur la figure) et sont indifférenciées.

On représente en langage C une file de voitures à l'aide d'un tableau de booléens. Par exemple, la file précédente sera décrite par le tableau ci-dessous :

```
bool t[] = { true, false, true, true, false, false,
                                false, false, false, false, true };
```

1. Proposer une fonction `int compte(bool t[], int n)` prenant en argument un tableau de booléens représentant une file et sa taille, et retournant le nombre de voitures que cette file contient.

2. Proposer une fonction `bool egales(bool t1[], bool t2[], bool n)` prenant en argument deux tableaux de même taille et leur taille (commune) et retournant un booléen indiquant si les files qu'ils représentent sont identiques.

Dans la suite, on aura besoin d'une relation d'ordre sur les tableaux représentant des files de même taille afin d'écrire des algorithmes efficaces. Pour ce faire, on utilisera l'ordre *lexicographique*. Pour deux tableaux `t1` et `t2` de taille n, on considérera que `t1`$\preceq$`t2` si et seulement si les tableaux représentent des files identiques ou bien s'il existe $i \in [\![0\,..\,n-1]\!]$ tel que `t1[i]` contient `false`, `t2[i]` contient `true`, et pour tout $j < i$, on a `t1[j]==t2[j]`.

3. Justifier qu'il s'agit d'un ordre total sur les tableaux booléens de taille n.

4. Proposer une fonction `bool plus_petit(bool t1[], bool t2[], int n)` prenant en argument deux tableaux de même taille et leur taille commune, et retournant un booléen indiquant si `t1`$\preceq$`t2`.

2.3 Déplacement de voitures dans la file

On identifie désormais une file de voitures à un tableau de booléens. On considère les schémas de la figure ci-après, représentant des exemples de files.

Une étape de simulation pour une file consiste à déplacer les voitures de la file, à tour de rôle, en commençant par la voiture la plus à droite, d'après les règles suivantes :

- une voiture se trouvant sur la case la plus à droite de la file sort de la file ;
- une voiture avance d'une case vers la droite si elle arrive sur une case inoccupée ;
- une case libérée par une voiture devient inoccupée ;
- la case la plus à gauche peut devenir occupée ou non, selon le cas considéré.

Pour cette étape de la simulation, on souhaite disposer d'une fonction `void avancer(bool t[], int n, bool ajout)` prenant en argument un tableau `t`, sa taille `n` et un booléen `ajout` indiquant si une nouvelle voiture arrive sur la file, modifiant le contenu du tableau de façon à appliquer l'étape de simulation décrite précédemment. Les schémas suivants illustrent les deux évolutions possibles de la file, correspondant aux tableaux après un appel à la fonction `avancer` :

(a) Tableau initial `t`

(b) `t` après `avancer(t, n, false)`

(c) `t` après `avancer(t, n, true)`

5. Proposer une fonction `void avancer(bool t[], int n, bool ajout)` répondant aux critères précédents.

6. Quel est le contenu de `t` après le programme suivant ?

```
bool t[] = { true, false, true, true, false, false,
                              false, false, false, false, true };

avancer(t, 11, false);
avancer(t, 11, true);
```

2.4 Circulation à deux files

On suppose à présent que l'accès à une des cases de la file, d'index m dans le tableau, peut ne pas être accessible, de sorte que les véhicules peuvent ne pas être en mesure, à un instant donné, de passer de la case d'index $m-1$ à la case m. On se propose ici de créer des fonctions permettant de gérer la circulation dans ce cas de figure.

Bien évidemment, la fonction `avancer` définie précédemment peut être utilisée, avec des arguments bien choisis, pour écrire les fonctions de cette partie du problème !

Cases situées après la case m

On considère un tableau `t` de taille n et `m` l'indice d'une case de ce tableau ($0 \leqslant$ `m` $< n$). On s'intéresse à une étape partielle où seules les voitures situées sur la case d'indice `m` ou à droite de cette case avancent normalement, les autres voitures ne se déplaçant pas.

Par exemple, la file

m

devient

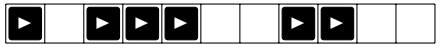
m

7. Proposer une fonction `void avancer_fin(bool t[], int n, int m)` prenant en argument un tableau `t`, sa taille `n` et un entier `m` et modifiant le contenu du tableau `t` pour réaliser cette étape partielle de déplacement.

Cases situées avant la case m

Soient un tableau `t` de taille n, un booléen `ajout` et `m` l'indice d'une case *inoccupée* de ce tableau. On considère une étape partielle où seules les voitures situées à gauche de la case d'indice `m` se déplacent, les autres voitures ne se déplacent pas. Le booléen `ajout` indique si une nouvelle voiture est introduite sur la case la plus à gauche.

Par exemple, la file



m

devient

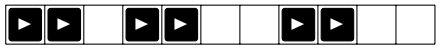
m

lorsque aucune nouvelle voiture n'est introduite.

8. Définir une fonction `void avancer_libre(bool t[], int n, int m, bool ajout)` qui modifie le tableau `t` pour réaliser cette étape partielle de déplacement.

On considère un tableau `t` de taille n dont la case d'indice `m` > 0 est temporairement inaccessible (attention, la case d'index `m` du tableau ne contient pas nécessairement `true`) et bloque l'avancée des voitures. Une voiture située immédiatement à gauche de la case d'indice `m` ne peut pas avancer. Les voitures situées sur les cases plus à gauche peuvent avancer, à moins d'être bloquées par une case occupée, les autres voitures ne se déplacent pas. Un booléen `ajout` indique si une nouvelle voiture est introduite lorsque cela est possible.

Par exemple, la file



m

devient

m

lorsque aucune nouvelle voiture n'est introduite.

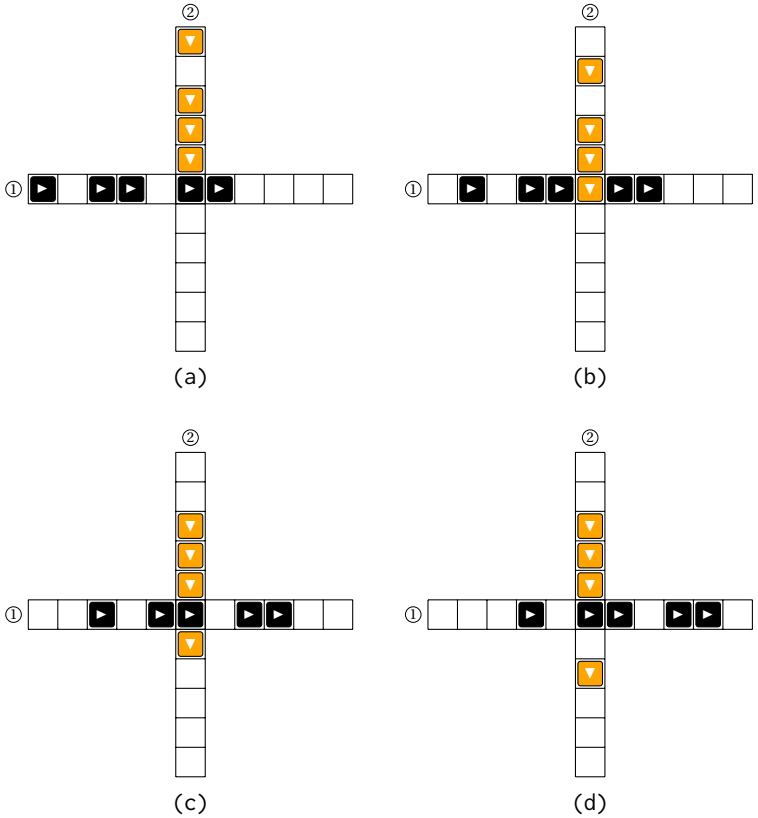
9. Définir une fonction `void avancer_bloque(bool t[], int n, int m, bool ajout)` qui modifie le tableau `t` pour réaliser cette étape partielle de déplacement.

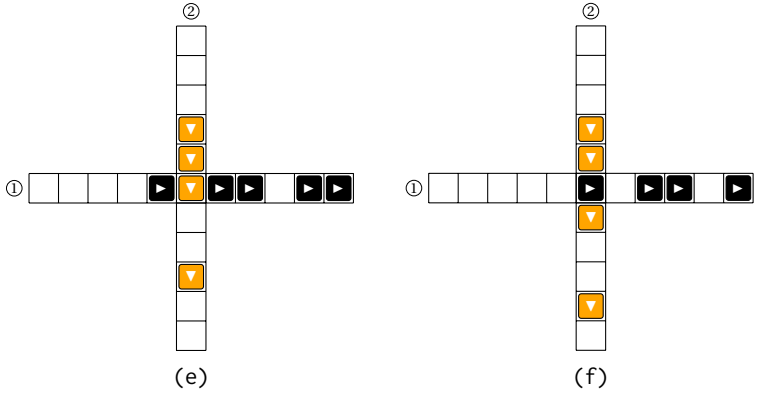
On considère dorénavant deux files ① et ② de même longueur n *impaire* se croisant en leur milieu ; on note `m` l'indice de la case du milieu. La file ① est toujours prioritaire sur la file ②. Les voitures ne peuvent pas quitter leur file et la case de croisement ne peut être occupée que par une seule voiture. Les voitures de la file ② ne peuvent accéder au croisement que si aucune voiture de la file ① ne s'apprête à y accéder.

L'ensemble des deux files est représenté en C par un *unique* tableau de taille $2n$. Les n premières cases correspondent à la file ①, les n suivantes à la file ② (les cases m et $n+m$ ne peuvent donc pas contenir toutes les deux `true`).

Une étape de simulation à deux files se déroule en deux temps. Dans un premier temps,

on déplace toutes les voitures situées sur le croisement ou après. Dans un second temps, les voitures situées avant le croisement sont déplacées en respectant la priorité. Par exemple, partant d'une configuration donnée par la première figure ci-dessous, les configurations successives sont données par les figures suivantes en considérant qu'aucune nouvelle voiture n'est introduite





10. Définir la fonction `void avancer_files(bool t[], int n, bool aj1, bool aj2)` qui prend en argument un tableau de taille $2n$ représentant deux files, la taille n d'une file, et deux booléens indiquant l'arrivée ou non de véhicules sur respectivement la file ① et la file ② et modifiant le contenu du tableau en respectant les règles énoncées précédemment.

Attention, on rappelle qu'ici `n` n'est pas la taille du tableau `t` puisque `t` contient les informations des *deux* files.

11. Que contient `t` à l'issue du programme ci-dessous ?

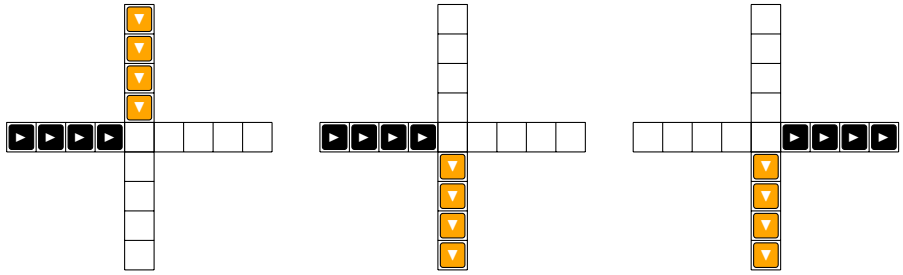
```
bool t[] = { false, true, false, true, false,
             false, true, true, false, false };

avancer_files(t, 5, true, false);
avancer_files(t, 5, false, true);
```

12. En considérant que de nouvelles voitures peuvent être introduites sur les premières cases des files lors d'une étape de simulation, décrire une situation où une voiture de la file ② serait indéfiniment bloquée.

2.5 Atteignabilité

On considère les trois configurations illustrées par la figure ci-dessus :



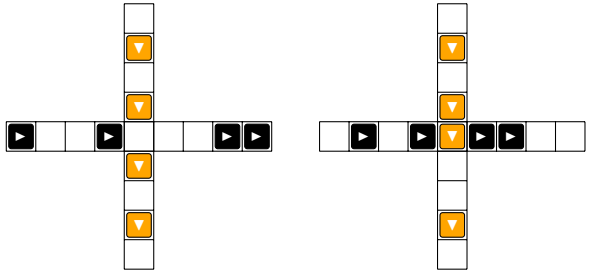
13. Est-il possible de passer de la configuration (a) à la configuration (b) ? Si oui, en combien d'étapes *au minimum* est-ce possible ?

14. Même question pour le passage de la configuration (a) à la configuration (c).

Certaines configurations peuvent être néfastes pour la fluidité du trafic. Une fois ces configurations identifiées, il est intéressant de savoir si elles peuvent apparaître en partant d'une configuration donnée. Lorsque c'est le cas, on dit qu'une telle configuration est *atteignable*.

On s'intéressera en particulier à la possibilité d'atteindre une situation donnée en partant de deux files vides (c'est-à-dire ne contenant aucun véhicule).

15. Les configurations suivantes sont-elles atteignables à partir de deux files vides ?



On souhaite à présent déterminer toutes les configurations qui peuvent être atteintes à partir d'une situation donnée, en suivant des règles spécifiques. Pour ce faire, on va construire un tableau `atteignables` de pointeurs `int*` qui accueillera les adresses de tableaux d'entiers de taille $2n$ correspondant à des configurations atteignables.

Un entier `nb_atteintes` contient le nombre de configurations atteintes. Les `nb_atteintes` premières cases du tableau `atteignables` contiendront les adresses d'autant de tableaux correspondant à des configurations *deux à deux distinctes* atteignables, rangés par ordre lexicographique croissant. Le contenu des autres cases du tableau est indéterminé.

Initialement, ces deux structures sont déclarées comme des variables globales de la façon suivante (on choisira une taille pour le tableau `atteignables` suffisamment grande pour qu'on ne risque pas de le remplir intégralement, on supposera dans la suite que cela n'arrivera pas [1]) :

```c
int nb_atteintes = 0;

int* atteignables[10000];
```

Évolution des configurations

Puisque l'on veut pouvoir mémoriser toutes les configurations rencontrées, on ne peut pas directement modifier les tableaux que l'on manipule : il nous faut créer de *nouveaux* tableaux qui correspondent à la configuration à l'étape suivante.

16. Proposer une fonction `bool* copie(bool t[], int n)` qui prend en argument un tableau et sa taille, alloue un tableau de même taille avec un appel à `malloc`, copie le contenu du tableau passé en argument dans le nouveau tableau, et retourne ce dernier.

17. En déduire une fonction `bool* suivant(bool t[], int n, int aj1, int aj2)` qui prend en argument un tableau `t` de taille $2n$ représentant une configuration à deux files de longueur n, un entier `n`, et deux booléens `aj1` et `aj2` correspondant à l'arrivée de véhicules respectivement sur les files ① et ②, et retournant un *nouveau* tableau de taille $2n$ correspondant à la situation après une étape de simulation. Le tableau `t` ne doit pas être modifié.

Gestion de la table des configurations atteintes

Avant d'ajouter un pointeur désignant une configuration à la table des configurations atteignables, il convient de vérifier que la configuration n'est pas déjà présente dans le tableau. On propose la fonction suivante :

```c
bool est_presente(bool t[], int n) {
    int debut = 0;
    int fin = nb_atteintes;

    while ( ... ) {
        int m = (debut+fin)/2;
        if egales(t, atteignables[m], n) {
            return true;
        }
        if (plus_petit(t, atteignables[m], n)) {
            ...
```

1. Bien évidemment, en pratique, il faudrait contrôler que `nb_atteintes` n'atteind pas la taille du tableau !

```
        } else {
            ...
        }
    }
    return false;
}
```

18. Quel algorithme est proposé ici ? Quel est son intérêt ?

19. Proposer un invariant de boucle pour l'algorithme proposé.

20. Proposer une condition booléenne à placer dans le while.

21. Completer les deux conséquences du second `if`

Si la configuration n'est pas déjà présente dans le tableau, on va l'ajouter. Pour se faire, on place le pointeur vers le tableau décrivant cette configuration dans la case d'index[2] `nb_atteintes` du tableau `atteignables`, puis on utilise un mécanisme similaire à celui du tri par insertion pour rétablir l'ordre lexicographique croissant pour les configurations, en replaçant l'élément nouvellement ajouté à la bonne place par « insertion ». On termine en incrémentant `nb_atteintes`

22. Proposer une fonction `void inserer(bool t[], int n)` qui effectue l'insertion de la configuration décrite par `t` dans le tableau `atteintes` en préservant l'ordre lexicographique, et mets à jour `nb_atteintes`.

Exploration des configurations

Pour explorer toutes les configurations, on propose la démarche suivante :

- On ajoute le tableau `t` correspondant à la configuration initale à la table avec la fonction `inserer`.
- On appelle une fonction `void explorer(bool t[], int n)` avec le tableau `t` correspondant à la configuration initiale et la taille `n` des files. Cette fonction est récursive et effectue les opérations suivantes :
  - Elle détermine chacun des tableaux correspondant aux différentes évolutions possibles, grâce à la fonction `suivant`, selon les valeurs possibles de `aj1` et `aj2`.
  - Pour chaque évolution possible, elle teste si elle est déjà dans le tableau `atteignables`. Si la configuration n'est pas encore présente dans le tableau, elle l'y ajoute avec la fonction `inserer`, puis s'appelle récursivement avec cette configuration.

On propose la règle suivante pour l'arrivée des voitures : **une voiture ne peut se présenter à l'entrée d'une file que si la première case de la file est libre.** En d'autres termes, la seule valeur possible pour `aj1` est `false` si la première case de la file ① est occupée. Si en revanche la première case de la file ① est libre, les deux valeurs booléennes doivent être considérées pour `aj1`. Il en va de même pour la file ②.

23. Proposer une implémentation de la fonction `void explorer(bool t[], int n)`. **On réfléchira à la façon d'éviter de possibles fuites de mémoire**. Pour n=5, le programme appellera donc la fonction `explorer` de la sorte :

```
int nb_atteintes = 0;

int* atteignables[10000];

int main(void) {
    bool initiale[] = { false, false, false, false, false,
                        false, false, false, false, false };

    inserer(initiale, 5);

    explorer(initiale, 5);

    // Analyse des résultats
}
```

24. Justifier que la fonction termine.

3 Tri RADIX

Dans ce problème, on cherche à implémenter un tri très efficace sur des tableaux d'**entiers naturels** dont le nombre maximal de chiffres N est supposé connu à l'avance (ce qui est le cas des entiers (`int`) C qui, lorsqu'ils sont représentés sur des mots de 64 bits, ont au plus N = 19 chiffres. Il est possible d'étendre ce principe aux entiers relatifs et même aux nombres flottants, ce qui rend ce tri très utile à certaines applications (tri de polygones en synthèse d'image par exemple). **Lorsqu'il sera question de tri dans ce problème, on sous-entendra qu'il s'agit de tri par ordre croissant pour la relation d'ordre considérée.**

Dans la suite, on travaille dans la base décimale usuelle, et on identifie les chiffres d'un nombre p par leur position k, de $k = 0$ à $k = N - 1$, considérée de la droite vers la gauche. Ainsi, pour l'entier $p = 1492$, le chiffre en position $k = 0$ est 2, celui en position $k = 2$ est 4 et ainsi de suite. On complétera avec des zéros au besoin, de sorte que pour ce même entier $p = 1492$, le chiffre en position $k = 5$ par exemple sera un 0.

1. Proposer une fonction `int digit(int p, int k)` prenant en argument un entier naturel p et une position (entier positif ou nul) et retournant son chiffre, dans l'écriture décimale de p, en position k. On ne cherchera pas à trouver une solution particulièrement efficace, on privilégiera ici la simplicité et une solution en $O(k)$.

Bien évidemment, cette opération devra être efficace dans une utilisation réelle du tri,

2. On rappelle que l'on suppose que cette case existe bien.

mais il existe des solutions pratiques pour obtenir une complexité $O(1)$, surtout si l'on travaille avec des bases non décimales. **Dans la suite, pour tous les calculs de complexité, on supposera disposer d'une fonction `digit` de complexité constante ($O(1)$).**

Afin de trier par ordre croissant un tableau `arr` contenant `n` entiers positifs, le tri RADIX va procéder par étapes. À chaque étape du tri, il commencera par compter le nombre d'occurrences de chacun des dix chiffres 0 à 9 pour une position k donnée dans l'ensemble des n entiers.

Ces résultats sont regroupés dans un tableau `counts` de taille 10 tel que, dans la case `counts[i]`, on trouve le nombre d'occurrences du chiffre i à la place k dans les n entiers de `arr`. Par exemple, pour le tableau

arr	49	42	37	1	29	22	11	17	9	4	27	14

on obtiendra, pour $k = 0$, le tableau suivant :

counts	0	2	2	0	2	0	0	3	0	3

Pour $k = 1$, on obtiendrait :

counts	3	3	3	1	2	0	0	0	0	0

2. Proposer une fonction `int* count(int arr[], int n, int k)` allouant, remplissant et retournant un tableau `counts` correspondant à la description précédente. On attend une complexité en $O(n)$.

Ces décomptes nous seront utiles pour implémenter chaque étape de tri.

Pour un nombre entier positif p, on note p_k son chiffre, en représentation digitale, à la position k. On définit une famille de N relations d'ordre $\preceq_k$ (ordres totaux) sur les entiers positifs de la façon suivante :

$$p \preceq_k q \qquad \equiv \qquad p_k \leqslant q_k$$

En d'autres termes, p est plus petit que q pour $\preceq_k$ si et seulement si le chiffre en position k dans la représentation décimale de p est plus petit que le chiffre en position k dans la représentation de q. Ainsi, $1492 \preceq_0 1789$ car $2 \leqslant 9$, et $1492 \succeq_1 1789$ car $9 \geqslant 8$.

On rappelle que le tri d'un tableau, pour une relation d'ordre $\preceq$, est qualifié de *stable* si, lorsqu'un élément p apparaît avant un élément q dans le tableau initial et que $p \preceq q$, alors p apparaît nécessairement avant q dans le tableau trié.

Dans la suite, on souhaite implémenter un tri **stable** triant les éléments d'un tableau selon la relation d'ordre $\preceq_k$, pour un k donné (dans $[\![0\,..\,N-1]\!]$).

3.a Indiquer le contenu du tableau suivant après un tel tri pour la relation d'ordre $\preceq_0$

arr	49	42	37	1	29	22	11	17	9	4	27	14

3.b Indiquer, de même, le contenu du tableau suivant après un tel tri pour la relation d'ordre $\preceq_1$

arr	49	42	37	1	29	22	11	17	9	4	27	14

Pour implémenter ce tri, on souhaite dans un premier temps construire un tableau `positions` indiquant, pour chaque chiffre d de 0 à 9, la position de la case occupée par le premier nombre du tableau `arr` dont le k^e chiffre est d. Pour les chiffres d n'apparaissant pas, le contenu de la case peut être une valeur arbitraire.

En d'autres termes, pour notre tableau

arr	49	42	37	1	29	22	11	17	9	4	27	14

on cherche à construire, pour $k = 0$ par exemple, le tableau suivant (où les ? sont des valeurs arbitraires, que l'on pourra choisir librement)

positions	?	0	2	?	4	?	?	6	?	9

Le tableau se lit de la façon suivante : puisque `positions[7]` contient la valeur 6, cela signifie que le premier nombre apparaissant tableau `arr` dont le chiffre des unités ($k = 0$) est 7 (soit l'entier 37) devra se retrouver dans la case d'index 6 du tableau trié.

4. Proposer une fonction `int* positions(int arr[], int n, int k)` allouant, remplissant et retournant un tableau de taille 10 respectant les règles précédentes (on ne demande pas de valeur particulière dans les cases correspondant aux chiffres n'apparaissant pas dans les écritures décimales des entiers du tableau à la position k, vous pouvez y placer ce qui vous arrange). On attend une complexité en $O(n)$. Vous pouvez évidemment vous servir de fonctions déjà écrites.

5. En s'aidant de cette fonction `positions`, proposer une fonction `void sort_by_digits(int arr[], int n, int k)` effectuant un tri *stable et en place* des éléments du tableau `arr` pour la relation d'ordre $\preceq_k$. Ce tri doit avoir une complexité linéaire ($O(n)$) en la taille du tableau, et ne pas causer de fuite de mémoire.

6. Justifier que l'on peut trier, pour l'ordre usuel $\leqslant$, un tableau d'entiers positifs ayant au plus N chiffres en faisant un certain nombre d'appels à la fonction précédente (on précisera les appels effectués et l'ordre dans lequel ils sont effectués).

7. En déduire une fonction `sort(int arr[], int n, int N)` triant un tableau d'entiers ayant au plus N chiffres par ordre croissant pour $\leqslant$.

8. Quelle est, en fonction de n et N, la complexité de ce tri? Pourquoi ce tri est-il particulièrement intéressant en pratique?