Devoir d'informatique

1 Collection

1. On alloue ici un tableau de `double` de taille $p+1$ avec un appel à la fonction `malloc`, que dont on remplit les cases avec la formule fournie par le sujet :

```c
double* prob(int q, int n, int p) {
    double* pr = (double*)malloc((p+1) * sizeof(double));
    for (int k=0; k<=p; ++k) {
        pr[k] = binom(k, n-q) * binom(p-k, q) / binom(p, n);
    }
    return pr;
}
```

Notons que l'on fait ici l'hypothèse implicite que `binom(a, b)` retourne bien 0 lorsque $a<0$ ou $a>b$. Si ce n'est pas le cas, il faudrait ajouter un test supplémentaire plaçant un 0 dans la case k lorsque $k>n-q$ ou $p-k>q$.

2. Si l'on ouvre un paquet de p cartes, le nombre de nouvelles cartes que l'on peut obtenir est nécessairement dans $[\![0\,..\,p]\!]$. Comme les cases du tableau retourné correspondent précisément aux probabilités d'obtenir k nouvelles cartes pour $k\in[\![0\,..\,p]\!]$, leur somme est égale à 1 (aux soucis d'arrondis flottants près).

On peut également le vérifier en calculant $\displaystyle\sum_{k=0}^{p}\frac{\binom{n-q}{k}\times\binom{q}{p-k}}{\binom{n}{p}}$, ce qui donnera 1.

3. On peut par exemple remarquer que, compte tenu de la question précédente, si x est un réel choisi aléatoirement selon une loi uniforme sur $[0,1]$, pour un k donné, la probabilité pour x de vérifier la relation suivante :

$$\sum_{i=0}^{k-1}\texttt{pr[i]}\leqslant x<\sum_{i=0}^{k}\texttt{pr[i]}$$

est exactement `pr[k]`. On détermine donc simplement le plus petit entier k vérifiant

$$x<\sum_{i=0}^{k}\texttt{pr[i]},\quad\text{soit}\quad x-\sum_{i=0}^{k}\texttt{pr[i]}<0.$$

En l'état, cependant, on aurait un souci dans le cas (certes improbable) où $x=1$. La dernière comparaison devrait être large. Mais comme on attend un résultat entre 0 et p inclus, si le test a échoué pour les valeurs de k de 0 à $p-1$ (soit $x-\sum_{i=0}^{p-1}\texttt{pr[i]}<0$), on peut retourner la valeur p directement sans faire davantage de tests [1], soit par exemple :

1. Placer le dernier cas hors de la boucle permet en outre d'éviter que la fonction puisse ne rien retourner si la somme des probabilités n'est pas exactement égale à 1 pour des raisons d'arrondis.

```c
int tirage(double pr[], int p) {
    double x = rand_d();
    for (int k=0; k<p; ++k) {
        x = x - pr[k];
        if (x < 0) { return k; }
    }
    return p;
}
```

4. On part de 0 cartes possédées, et on utilise la fonction précédente jusqu'à parvenir à réunir les n cartes, en prenant garde à libérer les tableaux alloués lors des appels à `prob` !

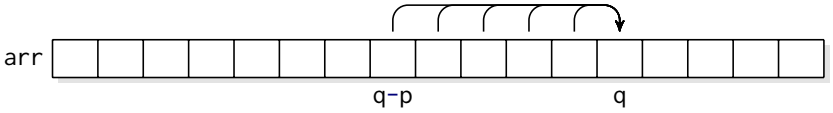
```c
int nb_paquets(int p, int n) {
    int q=0;       // Nombre de cartes différentes possédées
    int nb=0;      // Nombre de paquets ouverts
    while (q<n) {  // Tant que la collection est incomplète
        nb = nb+1;                  // On ouvre un paquet de plus
        double* pr = prob(q, n, p); // Calcul des probabilités
        q = q + tirage(pr, p);      // Tirage du nombre de nouvelles
        free(pr);                   //   cartes et libération de pr
    }
    return nb;
}
```

5. Pour avoir q cartes juste après avoir ouvert un paquet, il fallait avoir $q-k$ cartes avant l'ouverture ($k\in[\![0\,..\,p]\!]$) et que dans le paquet que l'on ouvre, parmi les p cartes, il y en ait k nouvelles. Il nous faut donc mettre à jour le tableau `arr` avec la relation suivante :

$$\texttt{arr[q]}\leftarrow\sum_{k=0}^{\min(p,q)}\texttt{arr[q-k]}\times\mathscr{P}(k,q-k,n,p)$$

On fera attention à ne pas sortir des bornes du tableau ! $q-k$ ne doit jamais être strictement négatif (on ne peut pas posséder un nombre négatif de cartes). Cela est garanti par le « min » figurant dans la formule précédente.

Une difficulté réside dans le fait que la nouvelle valeur de `arr[q]` dépend des valeurs du tableau aux positions dont les index vont de $\max(0,\ \texttt{q-p})$ à `q` :



Si l'on modifie directement une valeur dans le tableau, il faut s'assurer que cela n'aura pas de conséquence sur le calcul des autres valeurs. Par exemple, la valeur de `arr[i]` intervient dans le calcul des valeurs de `arr[i+1]`, `arr[i+2]`... Si on modifie sans précaution `arr[i]` dans le tableau avant de déterminer les nouvelles valeurs de `arr[i+1]`, `arr[i+2]`..., ces dernières seront incorrectes.

Une solution, pour éviter cette difficulté, est de mettre à jour les valeurs du tableau *de la droite vers la gauche gauche* (ainsi, lorsque l'on s'occupera de `arr[i]`, les nouvelles valeurs de `arr[i+1]`, `arr[i+2]`, ..., auront déjà été déterminées). Par exemple, en utilisant directement l'expression de $\mathscr{P}(k, q-k, n, p)$, cela donnerait :

```c
void ajoute_paquet(double arr[], int n, int p) {
    for (int q=n; q>=0; --q) {
        double sum = 0.0;  // Calcul de ∑_{k=0}^{min(p,q)} arr[q-k] × P(k, q-k, n, p)
        for (int k=0; k<=p && k<=q; ++k) {
            sum = sum + arr[q-k] * comb_d(k, n-q+k) * comb_d(p-k, q-k)
                                                    /comb_d(p, n);
        }
        arr[q] = sum;
    }
}
```

Même si dans la pratique on n'y gagne pas grand-chose, on pourrait regretter ici de ne pas utiliser la fonction `prob` précédemment définie pour déterminer d'un coup un ensemble de probabilités. C'est possible, mais l'écriture de la fonction est sensiblement plus complexe, surtout si l'on veut directement mettre à jour notre tableau, car il n'est alors pas possible de mettre à jour les valeurs une à une comme précédemment.

Cela donnerait par exemple :

```c
void ajoute_paquet(double arr[], int n, int p) {
    for (int q=n; q>=0; --q) {
        double arr_q = arr[q];
        double* pr = prob(q, n, p);  // Calcul des probabilités
        arr[q] = 0;                  // On repart de zéro pour arr[q] !
        for (int k=0; k<=p && q+k<=n; ++k) {
            arr[q+k] = arr[q+k] + arr_q * pr[k];
        }
        free(pr);
    }
}
```

Si l'on tient à utiliser `prob`, il est plus aisé de calculer les probabilités après l'ouverture du paquet dans un autre tableau, temporaire et alloué dynamiquement, et recopier les résultats dans le tableau original à la fin de la fonction :

```c
void ajoute_paquet(double arr[], int n, int p) {
    double* n_arr = (double*)malloc((n+1) * sizeof(double));
    for (int q=0; q<=n; ++q) { n_arr[q] = 0.0; }
    for (int q=0; q<=n; ++i) {
        double* pr = prob(q, n, p);    // Calcul des probabilités
        for (int k=0; k<=p && q+k<=n; ++k) {
            n_arr[q+k] = n_arr[q+k] + arr[q] * prob[k];
        }
        free(pr);
    }
    for (int q=0; q<=n; ++q) { arr[q] = n_arr[q]; }
    free(n_arr);
}
```

6. On alloue un tableau de taille $n+1$ contenant les probabilités initiales (1.0 dans la première case et 0.0 dans toutes les autres, puisque l'on a 100% de chances de posséder 0 cartes avant l'ouverture du premier paquet!) et on appelle r fois la fonction précédente :

```c
double* probabilites(int n, int p, int r) {
    double* arr = (double*)malloc((n+1) * sizeof(double));
    arr[0] = 1.0;
    for (int i=1; i<=n; ++i) { arr[i] = 0.0; }
    for (int i=0; i<r; ++i) { ajoute_paquet(arr, n, p); }
    return arr;
}
```

2 Circulation routière

2.1 Introduction

2.2 Préliminaires

1. Rien de complexe ici, on utilise un compteur :

```c
int compte(bool t[], int n) {
    int compteur = 0;
    for (int i=0; i<n; ++i) {
        if (t[i]) {
            compteur++;
        }
```

```
    }
    return compteur;
}
```

2. On verifie les éléments deux par deux. Si on trouve une différence, les files sont différentes. Si on parvient à la fin des tableaux, elles sont identiques.

```
bool egales(bool t1[], bool t2[], int n) {
    for (int i=0; i<n; ++i) {
        if (t1[i] != t2[i]) {
            return false;
        }
    }
    return true;
}
```

3. Le sujet attend en principe ici que l'on justifie qu'il s'agit d'un ordre *et* que celui-ci est total (ne pas avoir prouvé qu'il s'agit d'un ordre n'est pas une erreur bien grave, mais comme certains ont bien pris la peine de le faire, le barème refléte cet effort.)

Pour vérifier qu'il s'agit d'un ordre, on contrôle que la relation est

- réflexive (on obtient `true` avec deux files identiques) ;
- antisymétrique (si deux files `t1` et `t2` sont différentes, on peut considérer le premier booléen pour lequel elles diffèrent ; de part ce booléen, on ne peut avoir à la fois $\texttt{t1} \preceq \texttt{t2}$ et $\texttt{t2} \preceq \texttt{t1}$) ;
- transitive (supposons $\texttt{t1} \preceq \texttt{t2} \preceq \texttt{t3}$... si $\texttt{t1} = \texttt{t2}$ ou $\texttt{t2} = \texttt{t3}$, on a bien $\texttt{t1} \preceq \texttt{t3}$, et sinon on considère les indices i_a et i_b de l'algorithme lorsque l'on compare `t1` à `t2` et `t2` à `t3`, et on montre que $\min(i_a, i_b)$ permet de conclure que $\texttt{t1} \preceq \texttt{t3}$).

Le caractère total est ensuite évident : l'algorithme retourne systématiquement `true` ou `false` lorsque l'on compare deux files, donc deux files sont toujours comparables.

4. On modifie quelque peu la fonction `egales`. On notera le « `return t2[i]` » dans la boucle qui est une façon rapide d'écrire « `t1[i] < t2[i]` » puisque l'on sait, en ce point, que les deux booléens sont différents.

```
bool plus_petit(bool t1[], bool t2[], bool n) {
    for (int i=0; i<n; ++i) {
        if (t1[i] != t2[i]) {
            return t2[i];
        }
    }
    return true; // Elles sont égales
}
```

Note : il s'agissait bien de « `int n` » et non « `bool n` », comme nombre d'entre vous l'ont remarqué.

2.3 Déplacement de voitures dans la file

5. Il faut bien voir ici qu'*aucun* véhicule ne peut être bloqué, donc il n'y a aucune précaution à prendre : on ne fait qu'un décalage de tous les éléments dans le tableau d'un case vers la droite, et la case de gauche reçoit la valeur de `ajout`.

Note : en raison du caractère complètement opaque de la question concernant la valeur renvoyée par la fonction, aucun comportement particulier n'était attendu pour la fonction `avancer` de cette question et la fonction `avancer_fin` un peu plus loin.

```
bool avancer(bool t[], int n, bool ajout) {
    for (int i=n-2; i>=0; --i) {
        t[i+1] = t[i]; // On décale vers la droite le tableau
    }
    t[0] = ajout;
    return XXXX; // toutes réponses acceptées
}
```

6. On obtient pour `t`, après les deux appels :

```
{ true, false, true, false, true, true,
            false, false, false, false, false };
```

2.4 Circulation à deux files

7. On réutilise astucieusement la fonction avancer !

```
bool avancer_fin(bool t[], int n, int m) {
    return avancer(&t[m], n-m, false);
}
```

Cases situées avant la case m

8. Là aussi, il est intéressant d'utiliser la fonction avancer, en prenant bien garde à inclure la case m (soit $m+1$ cases à gérer) ! On peut ignorer la valeur retournée par la fonction (qui sera nécessairement `false` d'ailleurs).

```
void avancer_libre(bool t[], int n, int m, bool ajout) {
    avancer(t, m+1, ajout);
```

```
}
```

9. C'est la seule fonction un peu subtile : si l'on trouve une case vide entre $m-1$ et 0, en parcourant le tableau à rebours, on peut faire avancer ce qui se trouve à gauche de cette case vide. Sinon tout est bloqué. cela donne :

```
void avancer_bloque(bool t[], int n, int m, bool ajout) {
    for(int i=m-1; i>=0; --i) {
        if (!t[i]) {
            avancer(t, i+1, ajout);
            return; // On en a terminé !
        }
    }
}
```

10. Il ne reste à présent qu'à appeler intelligemment les fonctions précédentes. La file 1 avance normalement, la fin de la file 2 également, mais le début de la file 2 n'avance que si la case correspondant au croisement n'est pas bloqué (après avoir géré la file 1).

```
void avancer_files(bool t[], int n, bool aj1, bool aj2) {
    bool* t1 = t;                // Pour simplifier la suite,
    bool* t2 = &t[n];            // on définit t1 et t2
    int m = n/2;                 // Ou (n-1)/2 (n est impair ici)
    avancer(t1, n, aj1);         // La file 1 avance normalement
    avancer_fin(t2, n, m);       // La fin de la file 2 aussi
    if (t1[m]) {                 // Et on regarde le carrefour
        avancer_bloque(t2, n, m, aj2);
    } else {
        avancer_libre(t2, n, m, aj2);
    }
}
```

Il existe plein de variantes possibles, on s'efforcera de proposer des fonctions aussi claires que possibles.

11. A l'issue du programme, le tableau contient

```
{ false, true, false, true, false,
            true, false, true, false, true };
```

12. Si dans la file 1 il y a une succession ininterrompue de véhicules (ajout d'un nouveau véhicule à chaque itération), alors la seconde file est indéfiniment bloquée.

2.5 Atteignabilité

13. Il est possible de passer de la configuration (a) à la configuration (b) en 9 étapes (mais pas moins) : quatre étapes où la file 1 avance et la file 2 est bloquée, une étape où les deux files avancent, et à nouveau quatre étapes où les deux files avancent avec l'arrivée d'un nouveau véhicule à chaque itération sur la file 1 (bien évidemment, les véhicules sur la file 1 dans la configuration (b) ne peuvent être ceux apparaissant dans (a)).

14. Le passage de la configuration (a) à la configuration (c) est en revanche impossible : les règles indiquent qu'à l'étape précédant immédiatement (c) deux véhicules auraient dû se trouver sur le carrefour, ce qui est impossible. La configuration (c) ne peut être atteinte.

15. La configuration (a) ne peut être atteinte (on arrive à un paradoxe en remontant trois crans en arrière, deux véhicules simultanément sur le carrefour). En revanche, la configuration (b) peut parfaitement être atteinte.

Évolution des configurations

16. Il s'agit juste de recopier le contenu du tableau après l'appel à `malloc` :

```
bool* copie(bool t[], int n) {
    bool* res = (bool*)malloc(n * sizeof(bool));
    for (int i=0; i<n; ++i) {
        res[i] = t[i];
    }
    return res;
}
```

17. On effectue une copie (attention à la taille!), puis on appelle `avancer_files` sur la copie :

```
bool* suivant(bool t[], int n, int aj1, int aj2) {
    bool* res = copie(t, 2*n); // t est de taille 2n !
    avancer_files(res, n, aj1, aj2);
    return res;
}
```

Gestion de la table des configurations atteintes

18. Il s'agit d'une recherche dichotomique dans un tableau trié par ordre croissant. Cela permet de tester la présence de la configuration dans la liste des configurations déjà atteintes en temps logarithmique, ce qui est plus efficace qu'une simple recherche linéaire dans le tableau.

19. On peut utiliser l'invariant « si la configuration se trouve dans le tableau, alors elle se trouve dans une case d'index i vérifiant `debut` $\leqslant i <$ `fin` ». Il est *impératif* de faire très

attention au caractère strict/large des bornes, car c'est de là que viennnent les problèmes !

20. On peut sortir lorsqu'il n'y a plus d'index possible, donc lorsque `debut == fin` si l'on s'appuie sur la question précédente. On pourra donc écrire « `while (debut<fin)` ». Le caractère strict est important. Dans la plupart des cas, cela fonctionnera quand même, mais si la configuration est plus grande que toutes celles du tableau, une inégalité large conduirait à un accès au tableau en-dehors des configurations déjà enregistrées, et le résultat pourrait donc être incorrect, car on ne sait alors pas ce que contient le tableau dans cette case !

21. Toujours grâce à l'invariant, on peut écrire

```
        if (plus_petit(t, atteignables[m], n)) {
            fin = m;      // On poursuit avec [debut .. m[
        } else {
            debut = m+1;  // On poursuit avec [m+1 .. fin[
        }
```

On notera ici la dissymétrie des deux conséquences, qui découle du fait que les bornes `debut` et `fin` ne jouent pas exactement le même rôle dans l'invariant. On pourait s'en sortir avec « `debut = m` », mais il faudrait sortir de la boucle plus tôt (lorsque `fin-debut`$\leqslant 1$) et effectuer un dernier test hors de la boucle, donc en modifiant le code proposé.

22. On va procéder de la même façon que dans l'insertion d'un tri par insertion, en déplaçant les éléments d'un cran vers la fin du tableau, en commençant par le dernier, tant qu'ils sont plus grand que la configuration considérée :

```
void inserer(bool t[], int n) {
    int i = nb_atteintes;
    // On décale les éléments tant qu'ils sont plus grands que t
    while (i>0 and plus_petite(t, atteignables[i-1], n) {
        atteignables[i] = atteignables[i-1];
    }
    atteignables[i] = t; // On place t dans le trou ainsi créé
    nb_atteintes++;    // On a ajouté une configuration dans le tableau
}
```

Exploration des configurations

23. Beaucoup de possibilités d'écriture pour cette exploration des configurations atteignables. Par exemple :

```
void explorer(bool t[], int n) {
    // Quatre possibilités d'évolution à envisager
    // 0) aj1=false, aj2=false
    // 1) aj1=false, aj2=true
    // 2) aj1=true,  aj2=false
    // 3) aj1=true,  aj2=true
    for (int i=0; i<4; ++i) {
        bool aj1 = i>=2;
        bool aj2 = i%2==1;

        // Si c'est une évolution qui suit les règles
        if (!(aj1 && t[0] || aj2 && t[n])) {
            // On détermine la configuration suivante
            bool* s = suivant(t, n, aj1, aj2);

            // Et on regarde si elle est nouvelle
            if (!est_presente(s, n) {
                inserer(s, n);  // Si c'est le cas, on l'ajoute
                explorer(s, n); // et on poursuit l'exploration
            } else {
                free(s);        // Sinon, on libère la mémoire
            }
        }
    }
}
```

24. Il y a un nombre fini de configurations possibles, or la fonction n'effectue un appel récursif que si la configuration obtenue n'a pas déjà été observée. Le programme va donc nécessairement se terminer.

3 Tri RADIX

1. C'est une classique décomposition en base $b = 10$, on peut par exemple écrire :

```
int digit(int p, int k) {
    for (int i=0; i<k; ++i) {
        p = p/10;
    }
    return p%10;
}
```

On évitera, dans la mesure du possible, de calculer 10^{k+1} qui pourrait déborder de la capacité des entiers manipulés.

2. On alloue un tableau que l'on remplit de zéros, puis l'on compte :

```c
int* count(int arr[], int n, int k) {
    int* nb = (int*)malloc(10 * sizeof(int));
    for (int i=0; i<10; ++i) { nb[i] = 0; } // Compteurs à 0
    for (int i=0; i<n; ++i) {
        int d = digit(arr[i], k);   // Chiffre en pos. k de arr[i]
        nb[d] = nb[d] + 1;          // On incrémente le compteur
    }
    return nb;
}
```

3.a Lorsque l'on trie le tableau avec $\preceq_0$, les nombres sont rangés par ordre croissant de leur chiffre des unités, en conservant l'ordre original pour ceux partageant un même chiffre des unités. Cela donne donc :

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

3.b Avec $\preceq_1$, c'est le chiffre des dizaines qui est pris en compte, et on obtient :

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

4. Le tableau `positions` correspond exactement au tableau des sommes cumulées de `counts`, à un décalage d'une case vers la droite près (et un zéro dans la première case) :

counts	0	2	2	0	2	0	0	3	0	3
cumsum	0	2	4	4	6	6	6	9	9	12
↦ positions	?	0	2	?	4	?	?	6	?	9

Il devient alors simple de construire le tableau des positions :

```c
int* positions(int arr[], int n, int k) {
    int* pos = (int*)malloc(10 * sizeof(int));
    int* cnt = counts(arr, n, k);
    int sum = 0;
    for(int i=0; i<10; ++i) {
        int tmp = cnt[i];
        pos[i] = sum;
        sum = sum + tmp;
    }
    free(pos)
    return pos;
}
```

On a choisi ici de créer un tableau pos recueillant les positions (on n'oubliera pas de libérer le tableau retourné par l'appel à `counts`!) Mais on aurait très bien pu construire directement pos dans `cnt` pour éviter une allocation :

```c
int* positions(int arr[], int n, int k) {
    int* cnt = counts(arr, n, k);
    int sum = 0;
    for(int i=0; i<10; ++i) {
        int tmp = cnt[i];
        cnt[i] = sum;
        sum = sum + tmp;
    }
    return cnt;
}
```

5. On commence par appeler `positions` pour obtenir un tableau pos. On considère ensuite tous les éléments `arr[i]` du tableau, dans l'ordre. Pour un chiffre `d` donné, le premier nombre dont le chiffre en position `k` est `d` devra être placé en position `pos[d]`, le deuxième en position `pos[d]+1`, etc. On se contente donc d'incrémenter `pos[d]` à chaque occurrence du chiffre `d` en position `k`, pour qu'à tout moment il contienne la position à laquelle placer le prochain nombre dont le chiffre en position `k` est `d`.

On ne peut directement écrire le résultat directement dans le tableau `arr` car on risquerait d'écraser des nombres (et procéder par échanges n'est a priori pas adapté non plus puisque l'on souhaite un tri stable). On utilise donc un tableau temporaire pour le résultat. Cela donne :

```c
void sort_by_digits(int arr[], int n, int k) {
    int* pos = positions(arr, n, k);
    int* tmp = (int*)malloc(n * sizeof(int));
    for (int i=0; i<n; ++i) {
        int d = digit(arr[i], k);
        tmp[pos[d]] = arr[i];
        pos[d] = pos[d] + 1;
    }
    for (int i=0; i<n; ++i) { arr[i] = tmp[i]; }
    free(tmp);
}
```

6. Un nombre a doit se retrouver avant un nombre b si le premier chiffre (de la gauche vers la droite) pour lequel les deux nombres diffèrent est plus petit dans le cas de a.

Il est ainsi possible de trier le tableau en effectuant une série de tris chiffres par chiffres *de la droite vers la gauche* (donc k croissant de 0 à N − 1). En effet, le tri sur le premier chiffre pour lequel a et b diffèrent placera bien a avant b, et les tris ultérieurs, sur des

chiffres plus à gauche (et donc identiques), ne changeront plus l'ordre relatif de a et b car les tris sont stables.

7. On applique la suggestion de la fonction précédente :

```c
void sort(int arr[], int n, int N) {
    for (int k=0; k<N; ++k) {
        sort_by_digits(arr, n, k);
    }
}
```

8. `sort_by_digits` a une complexité linéaire en la taille du tableau ($\Theta(n)$). `sort` a donc une complexité en $\Theta(\mathrm{N} \times n)$. `N` étant fixé et généralement petit (dans la pratique, on ne travaillera souvent pas en base 10 mais par exemple en base 256, ce qui donne $\mathrm{N} = 4$ ou $\mathrm{N} = 8$ en général), cela revient à avoir un tri linéaire en la taille du tableau. Les performances excèdent fréquemment, pour des tableaux de grande taille, les tris par comparaison les plus efficaces, donc la complexité est quasi-linéaire.