Devoir d'informatique

1 Régulation de vol (langage C, d'après CentraleSupélec)

1.2 Implémentation du problème

1. On peut par exemple compter le nombre de valeurs non-nulles strictement sous la diagonale, soit pour tous les couples (i, j) vérifiant $0 \leqslant j < i < 3n$. Attention à ne pas oublier le 3 (ni le symbole « * » entre le 3 et le n) ! Précisons qu'il est inutile de considérer le cas $i = j$, il y a de toute façon des zéros sur la diagonale.

```c
int nb_conflits(int n) {
    int nb=0;
    for (int i=1; i<3*n; ++i) {
        for (int j=0; j<i; ++j) {
            if (conflit[i][j] > 0) { nb = nb+1; }
        }
    }
    return nb;
}
```

On peut également effectuer la somme sur l'intégralité du tableau (également pour $j > i$), et diviser le résultat par deux, le tableau étant symétrique par rapport à sa diagonale.

2. Le test et l'éventuelle incrémentation ont une complexité constante, donc les deux boucles imbriquées conduisent à une complexité quadratique $\Theta(n^2)$.

1.3 Régulation

3. On alloue ici un tableau de taille 3, que l'on remplit avec des zéros. Puis on considère les vols un à un et on incrémente la case correspondant à leur régulation :

```c
int* nb_vols_par_niveau_relatif(int reg[], int n) {
    int* count = (int*)malloc(3 * sizeof(int));
    for (int i=0; i<3; ++i) { // On initialise les trois cases à 0
        count[i] = 0;
    }
    for (int i=0; i<n; ++i) { // On décompte chacun des niveaux
        count[reg[i]] = count[reg[i]] + 1;
    }
    return count;
}
```

Il faut impérativement une allocation dynamique (avec `malloc`) ici, car s'il est possible de faire le calcul dans un tableau statique, il n'est en revanche pas permis de retourner ledit tableau (qui disparaîtrait en arrivant à la limite de sa portée, à la fin de la fonction).

4. Il nous faut sommer les coûts, pour chaque paire de vols (i, j) (en considérant par exemple $0 \leqslant j < i < n$), qui se trouvent dans le tableau `conflits` à la ligne $3 \times i + \texttt{reg}[i]$ et à la colonne $3 \times j + \texttt{reg}[j]$. Cela donne :

```c
int cout_regulation(int reg[], int n) {
    int cout=0;
    for (int i=1; i<n; ++i) {
        for (int j=0; j<i; ++j) {
            cout = cout + conflit[3*i+reg[i]][3*j+reg[j]];
        }
    }
    return cout;
}
```

5. La fonction doit très simplement retourner un tableau de n entiers, rempli avec autant de zéros :

```c
int* regulation_RFL(int n) {
    int* reg = (int*)malloc(n * sizeof(int));
    for (int i=0; i<n; ++i) { reg[i] = 0; }
    return reg;
}
```

6. On utilise le résultat de la fonction précédente pour un appel à `cout_regulation`, en n'oubliant pas de libérer le tableau `reg` avant d'en finir avec notre fonction.

```c
int cout_RFL(int n) {
    int reg = regulation_RFL(n);
    int cout = cout_regulation(reg, n);
    free(reg);
    return cout;
}
```

7. Il y a 3^n régulations possibles, et le calcul du coût d'une régulation est quadratique. La fonction aurait donc une complexité $\Theta(3^n n^2)$, très rapidement trop grande pour être utilisable même pour des valeurs raisonnables de n. Même si l'on trouve un moyen de gagner du temps sur le calcul des coûts, cela resterait trop important en $\Omega(3^n)$.

1.4 L'algorithme Minimal

8. On effectue la somme des coefficients sur la ligne s qui sont sur des colonnes k telles que [1] `etat_sommet[k]` soit 1 (choisi) ou 2 (ni choisi ni supprimé), donc différent de 0 :

```c
int cout_du_sommet(int s, int etat_sommet[], int n) {
    int cout = 0;
    for (int k=0; k<3*n; ++k) {
        if (etat_sommet[k] != 0) {
            cout = cout + conflit[s][k];
        }
    }
    return cout;
}
```

9. Il s'agit d'une simple recherche de minimum, mais uniquement parmi les sommets encore à considérer, ceux dont l'état est « 2 ». La difficulté qui en découle est que l'on ne sait pas quel est le premier de ces sommets (voire même s'il en existe). On ne peut donc pas initialiser notre recherche avec le coût du sommet $s = 0$.

On utilise ici la valeur -1 pour la variable `meilleur_sommet` pour indiquer que l'on n'a encore identifié aucun sommet convenable (c'est la valeur qui sera retournée si aucun sommet n'est disponible). Si on a identifié au moins un sommet convenable, `plus_petit_cout` est le plus petit coût identifié, et `meilleur_sommet` le sommet correspondant.

```c
int sommet_de_cout_min(int etat_sommet[], int n) {
    int meilleur_sommet = -1;       // Pas encore identifié
    int plus_petit_cout;            // Pas de coût associé
    for (int s=0; s<3*n; ++s) {     // Pour tous les sommets s possibles
        if (etat_sommet[s] == 2) { // Si le sommet s est à considérer
            int cout = cout_du_sommet(s, etat_sommet, n);
            if (meilleur_sommet == -1 || cout <= plus_petit_cout) {
                meilleur_sommet = s;
                plus_petit_cout = cout;
            }
        }
    }
    return meilleur_sommet;
}
```

On pourrait également initialiser `plus_petit_cout` avec `INT_MAX` pour se dispenser de la condition `meilleur_sommet == -1` du second **if**.

1. Comme un vol n'est jamais en conflit avec lui-même (la diagonale par blocs 3×3 ne contient que des zéros), il n'est pas utile ici de s'en préoccuper.

10. Pour l'algorithme « Minimal », on crée un tableau `etat_sommet` repli de 2, puis on fait appel n fois à la fonction précédente pour choisir un sommet. Cela détermine un vol et sa régulation, et on met alors à jour l'état des sommets associé au vol sélectionné :

```c
int* minimal(int n) {
    int* reg = regulation_RFL(n);
    int* etat_sommet = (int*)malloc(3*n * sizeof(int));
    for (int i=0; i<3*n; ++i) {
        etat_sommet[i] = 2;
    }
    for (int i=0; i<n; ++i) {
        int s = sommet_de_cout_min(etat_sommet, n);
        int v = s//3;   // vol sélectionné
        int r = s%3;    // régulation sélectionnée
        reg[v] = r;     // On mémorise la régulation
        for (int k=0; k<3; ++k) { etat_sommet[3*v+k] = 0; }
        etat_sommet[s] = 1;  // On élimine/sélectionne les sommets
    }
    free(etat_sommet);  // etat_sommet n'est plus utile
    return reg;
}
```

11. La fonction `sommet_de_cout_min` a une complexité quadratique ($\Theta(n^2)$) dans le pire des cas car elle effectue jusqu'à $3n$ calculs de coûts en temps linéaire en n. C'est par exemple le cas lorsqu'il reste au moins la moitié des sommets. La fonction `Minimal`, qui l'appelle n fois (et au moins $n/2$ fois avec la moitié des sommets restants) a donc une complexité cubique ($\Theta(n^3)$).

1.5 Recuit simulé

12. Il s'agit ici d'implémenter ce qui est décrit par le sujet. Plusieurs points à noter :

- choisir un vol revient simplement à effectuer un tirage aléatoire entre 0 et $n-1$;
- pour choisir une nouvelle régulation pour ce vol, la solution la plus simple consiste à ajouter, à l'ancienne régulation r, soit 1, soit 2 (ce qui peut être obtenu avec `rand_i(2)+1`) et ramener le résultat dans $[\![0\,..\,2]\!]$ grâce à l'opérateur `%` ;
- pour déterminer la différence de coût Δ_c entre la précédente régulation et la nouvelle régulation, on pourrait faire appel à `cout_regulation` (mais la fonction a un coût quadratique) ou bien à `cout_du_sommet` (linéaire, mais cela nécessite de construire un tableau `etat_sommet`). On peut aussi plus simplement revenir directement à la matrice `conflits`, ce qui n'est pas bien compliqué et la solution choisie ici ;
- pour savoir si la nouvelle régulation vient remplacer l'ancienne, on peut remarquer que `rand_d()` $< \exp(-\Delta_C/T)$ est *toujours* vrai si Δ_c est négatif, ce qui signifie qu'il n'est pas utile de considérer les deux situations $\Delta_c \leqslant 0$ et $\Delta_c > 0$ séparément.

Ces précisions faites, la fonction ne présente guère de difficultés :

```c
void etape_recuit(int reg[], int n, double T) {
    int vol = rand_i(n);            // On choisit un vol
    int r = reg[vol];               // Sa régulation actuelle
    int n_r = (r + rand_i(2) + 1); // Une nouvelle régulation
    int diff_cout = 0;              // Calcul de Δ_c
    for (int i=0; i<n; ++i) {
        diff_cout = diff_cout + conflits[3*vol+n_r][3*i+reg[i]]
                              - conflits[3*vol+r][3*i+reg[i]]
    }
    if (rand_d() < exp(-diff_cout/T)) {
        reg[vol] = n_r;             // On adopte la nouvelle régulation
    }
}
```

13. La fonction `recuit` alloue un tableau contenant une régulation grâce à `regulation_RFL`, puis fait des appels à la fonction précédente tant que $T \geqslant 1.0$:

```c
int* recuit(int n) {
    int* reg = regulation_RFL(n);
    double T = 1000.0;
    while (T >= 1.0) {
        etape_recuit(reg, n, T);
        T = 0.99*T;
    }
    return reg;
}
```

2 Problème 2 : Tri faire-valoir (langage C)

2.1 Implémentation

1. Il suffit d'échanger les deux valeurs si nécessaire :

```c
void sort2(int arr[]) {
    if (arr2[0] > arr2[1]) {
        int tmp = arr2[0];
        arr2[0] = arr2[1];
        arr2[1] = tmp;
    }
}
```

2. On applique l'algorithme proposé :

```c
void stooge_sort(int arr[], int n) {
    if (n==0) { return; }
    if (n==1) { sort2(arr); return; }
    int s = (2*n-1)/3+1;
    stoogesort(arr, s);
    stoogesort(&arr[n-s], s);
    stoogesort(arr, s);
}
```

2.2 Analyse

3. Après le premier tri, on ne peut pas affirmer qu'un quelconque élément soit bien placé. En effet, si les éléments étaient initialement dans l'ordre inverse, le dernier tiers du tableau, non modifié, ne contient pas d'éléments bien placés. Et si les éléments étaint initialement dans l'ordre croissant, à l'exception du dernier élément plus petit que tous les autres, alors le premier tri ne modifie pas la position des éléments dans les deux premiers tiers du tableau, or ceux-ci sont mal placés.

4. Les $\lfloor n/3 \rfloor$ derniers éléments sont bien placés.

5. Les $\lceil 2n/3 \rceil$ premiers éléments ne sont pas nécessairement bien placés avant le dernier des trois tri, mais ils le deviennent après celui-ci.

6. Pour $n \geqslant 3$, on a $\lceil 2n/3 \rceil < n$.

7. Les seuls déplacements d'éléments dans le tableau sont effectués par `sort2`, qui ne fait que permuter des éléments si celui de gauche est strictement plus grand que celui de droite, donc le tri est stable.

8. On a $u_n = 3u_{\lceil 2n/3 \rceil}$.

9. Le nombre d'éléments lorsque l'on est dans le k^{e} appel récursif imbriqué est de l'ordre de $n \times \left(\frac{2}{3}\right)^k$. Il atteint 2 lorsque $k \log(2/3) \simeq \log(1/2)$.

3 Problème 3 : points fixes (langage OCaml, d'après X)

3.2 Attracteurs

1. Par exemple la fonction qui à $i \in \mathcal{E}_4$ associe $i+1[4]$.

2. Il existe neuf solutions, trois consistant en deux échanges (1032, 2301 et 3210) et six permutations circulaires (1230, 1302, 2031, 2310, 3012 et 3201).

Pour les curieux, la formule générale est $n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}$, soit l'entier le plus proche de $\frac{n!}{\text{e}}$.

3. Nous allons essayer tous les entiers de $n-1$ à 0, au moyen d'une fonction récursive, jusqu'à trouver un point fixe de la fonction, ou épuiser les entiers candidats :

```
let has_fixed_point f n =
  let rec loop = function
    | -1 -> false
    | i ->  f i = i || loop (i-1)
  in loop (n-1)
```

4. On utilise à profit la récursion ici, en remarquant que $f^k(i) = f^{k1}\ parens * f(i)$ lorsque $k > 0$:

```
let rec iter f i k =
  if k=0 then i else iter f (f i) (k-1)
```

On peut également écrire, en considérant $f^k(i) = f\left(f^{k1}(i)\right)$ lorsque $k > 0$:

```
let rec iter f i k =
  if k=0 then i else f (iter f i (k-1))
```

5. On prendra garde à toujours prêter une grande attention aux définitions de l'énoncé, ici à la notion d'attracteur. Un malentendu sur ce que cela signifie risque de conduire à un gaspillage de temps à écrire des fonctions qui ne répondent pas aux besoins exprimés par le sujet. $f^k(x)$ point fixe signifie $f\left(f^k(x)\right) = f^k(x)$ et non $f^k(x) = x$!

Pour un x donné, s'il existe k tel que $f^k(x)$ est un point fixe de f, alors nécessairement $f^{n-1}(x)$ est un point fixe de f. En effet, si l'on considère la suite x, $f(x)$, $f^2(x)$, ..., $f^n(x)$, il y a nécessairement un entier y qui apparaît au moins deux fois. Notons p et q les rangs dans la suite où y apparaît pour les deux premières fois ($y = f^p(x) = f^q(x)$, avec $0 \leqslant p < q \leqslant n$). Deux cas sont possibles :

- $q = p+1$, donc $f(y) = y$, y est un point fixe, et dans ce cas $f^{n-1}(x) = f^{n-1-p}\left(y\right) = y$ (on a $n-1-p \geqslant 0$) est un point fixe ;
- $q > p+1$ auquel cas au-delà du rang p, les valeurs $f^k(x)$ sont un cycle $f^p(x), f^{p+1}(x), \ldots, f^{q-1}(x)$ de $q-p \geqslant 2$ valeurs distinctes qui se répètent (pour tout $k \geqslant p$, $f^k(x) = f^{p+r}(x)$ où r est le reste de la division entière de $k-p$ par $q-p$), donc $f^{n-1}(x) \neq f^n(x)$.

On peut donc simplement vérifier que, pour tout $x \in \mathcal{E}_n$, $f^{n-1}(x) = f^n(x)$ (on pourrait aussi vérifier qu'il y a un point fixe parmi $x, f(x), \ldots, f^{n-1}(x)$).

Dans une telle question, l'**explication doit être très claire** (et dans la mesure du possible succinte), pour qu'il n'y ait aucune ambiguïté sur l'algorithme proposé. Au besoin, n'hésitez pas à écrire les choses en pseudo-code.

6. On applique l'idée proposée à la question précédente. Cette fois encore, pour tester, pour tout $i \in \mathcal{E}_n$ on a $f\left(f^{n-1}(i)\right) = f^{n-1}(i)$, on utilise une fonction récursive. On notera y la valeur de $f^{n-1}(i)$ pour ne pas la calculer deux fois lorsque l'on teste si $f\left(y\right) = y$. Cela donne :

```
let has_attractor f n =
  let rec loop = function
    | -1 -> true
    | k -> let y = iter f k (n-1) in
           y = f y && loop (k-1)
  in loop (n-1)
```

7. Dans le pire des cas, on doit calculer $f^n(i)$ pour tout $i \in \mathcal{E}_n$, avant d'effectuer un test en temps constant. Chaque calcul de $f^n(i)$ a un coût linéaire en n, donc la fonction a une complexité temporelle dans le pire des cas quadratique en n ($O(n)$)

8. On utilise à nouveau la récursion, en notant que le temps de convergence de i est nul si $f(i) = i$, et sinon il vaut le temps de convergence de $f(i)$ plus 1. Cela donne donc :

```
let rec time_conv f i =
  if f i = i then 0 else time_conv f (f i)
```

3.3 Recherche efficace de points fixes.

9. On teste, pour tout $i \in [\![1 \,..\, n-1]\!]$, si $f(i-1) \leqslant f(i)$. Toujours avec une fonction récursive pour implémenter l'itération :

```
let incr f n =
  let rec loop = function
    | 0-> true
    | k -> f (k-1) <= f k && loop (k-1)
  in loop (n-1)
```

10. Il existe de nombreuses façons de procéder. Considérons par exemple l'ensemble des entiers $p \in [\![0 \,..\, n-1]\!]$ tels que $f\left(p\right) > p$.

Si cet ensemble est vide, alors nécessairement $f(0) = 0$, donc f admet un point fixe.

Sinon, notons k le plus grand élément de l'ensemble. On a nécessairement $k < n-1$ puisque $f(n-1) \leqslant n-1$, donc $k+1 \in \mathcal{E}_n$.

On a donc $f(k+1) \leqslant k+1$ et $f(k+1) \geqslant f(k) > k$ dont $f(k+1) \geqslant k+1$.

Les deux inégalités imposent $f(k+1) = k+1$, donc $k+1$ est un point fixe.

On aurait également pu étudier la fonction $g(n) = f(n) - n$ et montrer l'existence d'une racine (attention, le théorème des valeurs intermédiaires concerne normalement les fonctions *continues*), ou utiliser une démonstration par récurrence sur la taille de $\mathcal{E}_n$ (c'est

nécessairement vrai pour $\mathcal{E}_1$, et si c'est vrai pour $\mathcal{E}_n$, pour le montrer pour $\mathcal{E}_{n+1}$, on distingue le cas où $f_n = n$ et celui où $f(n) < n$, auquel cas la restriction de la fonction à $\mathcal{E}_n$ est à valeur dans $\mathcal{E}_n$), ou que sans points fixes, $\forall x \in \mathcal{E}_n, f(x) > x$ donc $f(n-1) \notin \mathcal{E}_n$...

11. On va ici s'inspirer de la recherche dichotomique :

```
let fixed_point f n =
  let rec search i j =
    (* Invariant : il y a un point fixe dans [i..j] *)
    let k = (i+j)/2 in
    let image = f k in
    if image = k then k else
    if image > k then search i (k-1)
                 else search (k+1) j
  in search 0 (n-1)
```

12. La propriété est l'invariant indiqué la fonction précédente : lors de tous les appels, il existe un point fixe de f dans $[\![i \,..\, j]\!]$. On l'a montré dans la question précédente pour l'appel initial, et les conditions des appels récursifs maintiennent cet invariant.

13. Si l'on considère la quantité `j-i`, elle décroit strictement à chaque appel (en effet $k \in [\![i \,..\, j]\!]$, donc $k+1 > i$ ou $k-1 < j$, selon le cas considéré) dans l'ensemble des entiers naturels.

Pour justifier la complexité, on peut noter qu'après k itérations, on a donc `j-i`$\leqslant (n-1)/2^k$. Si l'on tente d'effectuer $\log_2(n-1)$ itérations, on parvient donc à `j == i` (et l'algorithme s'arrête, car compte tenu de l'invariant de boucle, `mini` est un point fixe).

Chaque itération de la boucle **`while`** s'effectuant **en temps constant** (ne pas oublier ce point, il n'est pas possible par exemple d'effectuer une copie d'une partie de la liste à chaque itération !), on a bien une complexité logarithmique.

Remarque : il faut s'efforcer d'être rigoureux sur ce genre de question. Il est généralement faux de dire que le nombre d'éléments restant à considérer est divisé par deux (il y a parfois un nombre impair d'éléments), et possiblement faux de dire qu'il est divisé *au moins* par deux. Cela étant dit, il suffit de majorer la suite par une suite géométrique de raison $k < 1$ pour obtenir une complexité logarithmique.

Résultats