Devoir d'informatique

Quelques remarques avant de commencer

Le sujet est constitué de trois problèmes indépendants, de difficultés comparables mais sur des thématiques différentes. Les deux premiers sont en langage C (le premier est un algorithme d'optimisation numérique, le second une étude d'un algorithme de tri de tableau), le dernier en langage OCaml (et étudie des propriétés de fonctions de $[\![0\,..\,n-1]\!]$ dans $[\![0\,..\,n-1]\!]$). N'hésitez pas à privilégier les problèmes qui vous inspirent davantage.

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

1.1 Description du problème

Ce problème s'intéresse à la régulation du trafic aérien, et plus précisément à la détermination des paramètres des plans de vol permettant de minimiser les risques de collisions entre deux appareils.

Afin d'éviter les collisions entre avions, les altitudes de vol en croisière sont normalisées. Dans la majorité des pays, les avions volent à une altitude multiple de 1000 pieds (un pied vaut 30,48 cm) au-dessus de la surface isobare à 1013,25 hPa. L'espace aérien est ainsi découpé en tranches horizontales appelées *niveaux de vol* et désignées par les lettres « FL » (*flight level*) suivies de l'altitude en centaines de pieds : « FL310 » désigne une altitude de croisière de 31000 pieds au-dessus de la surface isobare de référence.

Eurocontrol est l'organisation européenne chargée de la navigation aérienne, elle gère plusieurs dizaines de milliers de vols par jour. Toute compagnie qui souhaite faire traverser le ciel européen à un de ses avions doit soumettre à cet organisme un plan de vol comprenant un certain nombre d'informations : trajet, heure de départ, niveau de vol souhaité, etc. Muni de ces informations, Eurocontrol peut prévoir les secteurs aériens qui vont être surchargés et prendre des mesures en conséquence pour les désengorger : retard au décollage, modification de la route à suivre, etc.

Lors du dépôt d'un plan de vol, la compagnie aérienne doit préciser à quel niveau de vol elle souhaite faire évoluer son avion lors de la phase de croisière. Ce niveau de vol souhaité, le RFL pour *requested flight level*, correspond le plus souvent à l'altitude à laquelle la consommation de carburant sera minimale. Cette altitude dépend du type d'avion, de sa charge, de la distance à parcourir, des conditions météorologiques, etc.

Cependant, du fait des similitudes entre les différents avions qui équipent les compagnies aériennes, certains niveaux de vols sont très demandés ce qui engendre des conflits potentiels, deux avions risquant de se croiser à des altitudes proches. Les contrôleurs aériens de la région concernée par un conflit doivent alors gérer le croisement de ces deux avions.

Pour alléger le travail des contrôleurs et diminuer les risques, le système de régulation s'autorise à faire voler un avion à un niveau différent de son RFL. Cependant, cela engendre généralement une augmentation de la consommation de carburant. C'est pourquoi on limite le choix aux niveaux immédiatement supérieur et inférieur au RFL.

On peut modéliser ce problème de régulation est modélisé par un « graphe » dans lequel chaque vol est représenté par trois sommets. Le sommet 0 correspond à l'attribution du RFL, le sommet + au niveau supérieur et le sommet − au niveau inférieur. Chaque conflit potentiel entre deux vols sera représenté par une arête reliant les deux sommets concernés. Le coût d'un conflit potentiel (plus ou moins important en fonction de sa durée, de la distance minimale entre les avions, etc.) sera représenté par une valuation sur l'arête correspondante.

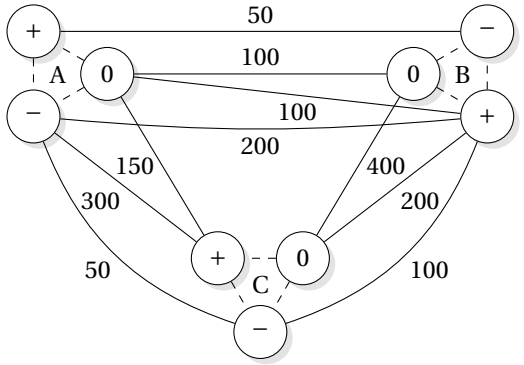


FIGURE 1 – Exemple de conflits potentiels entre trois vols.

Dans l'exemple de la figure 1, faire voler les trois avions à leur RFL engendre un cout de régulation entre A et B de 100 et un cout de régulation entre B et C de 400, soit un cout total

de la régulation de 500 (il n'y a pas de conflit entre A et C). Faire voler l'avion A à son RFL et les avions B et C au-dessus de leur RFL engendre un conflit potentiel de cout 100 entre A et B et 150 entre A et C, soit un cout total de 250 (il n'y a plus de conflit entre B et C).

On peut observer que cet exemple possède des solutions de cout nul, par exemple faire voler A et C à leur RFL et B au-dessous de son RFL. Mais en général le nombre d'avions en vol est tel que des conflits potentiels sont inévitables. Le but de la régulation est d'imposer des plans de vol qui réduisent le plus possible le cout total de la résolution des conflits.

1.2 Implémentation du problème

Chaque vol étant représenté par trois sommets, le graphe des conflits associé à n vols $v_0, v_1, \ldots, v_{n-1}$ possède $3n$ sommets que nous numéroterons de 0 à $3n-1$. Nous conviendrons que pour $0 \leqslant k < n$:

- le sommet $3k$ représente le vol v_k à son RFL ;
- le sommet $3k+1$ représente le vol v_k au-dessus de son RFL ;
- le sommet $3k+2$ représente le vol v_k au-dessous de son RFL.

Le coût de chaque conflit potentiel est stocké dans un tableau de $3n$ lignes et $3n$ colonnes, accessible grâce à nom global `conflit` : si `i` et `j` désignent deux sommets du graphe, alors `conflit[i][j]` est égal au cout du conflit potentiel (s'il existe) entre les plans de vol représentés par les sommets i et j. S'il n'y a pas de conflit entre ces deux sommets, `conflit[i][j]` vaut 0. On convient que `conflit[i][j]` vaut 0 si les sommets i et j correspondent au même vol (figure 2).

On notera que pour tout couple de sommets (i, j), `conflit[i][j]` et `conflit[j][i]`, représentent un seul et même conflit et donc `conflit[i][j] == conflit[j][i]`.

```
int conflit[9][9] = { {   0,   0,   0, 100, 100,   0,   0, 150,   0 },
                      {   0,   0,   0,   0,   0,  50,   0,   0,   0 },
                      {   0,   0,   0,   0, 200,   0,   0, 300,  50 },
                      { 100,   0,   0,   0,   0,   0, 400,   0,   0 },
                      { 100,   0, 200,   0,   0,   0, 200,   0, 100 },
                      {   0,  50,   0,   0,   0,   0,   0,   0,   0 },
                      {   0,   0,   0, 400, 200,   0,   0,   0,   0 },
                      { 150,   0, 300,   0,   0,   0,   0,   0,   0 },
                      {   0,   0,  50,   0, 100,   0,   0,   0,   0 } };
```

FIGURE 2 – Tableau des couts des conflits associé au graphe représenté figure 3

1. Écrire une fonction `int nb_conflits(int n)` prenant en argument le nombre de vols `n` et qui renvoie le nombre de conflits potentiels, c'est-à-dire le nombre d'arêtes de valuation non nulle du graphe. On rappelle que le tableau `conflits`, de taille $3n \times 3n$, est supposé global et disponible depuis n'importe quelle fonction.

2. Exprimer en fonction de n la complexité de cette fonction.

1.3 Régulation

Pour un vol v_k, on appelle *niveau relatif* l'entier r_k valant 0, 1 ou 2 tel que :

- $r_k = 0$ représente le vol v_k à son RFL ;
- $r_k = 1$ représente le vol v_k au-dessus de son RFL ;
- $r_k = 2$ représente le vol v_k au-dessous de son RFL.

On appelle *régulation* le tableau d'entiers $(r_0, r_1, \ldots, r_{n-1})$ des niveaux relatifs de chacun des vols. Par exemple, la régulation $(0, 0, \ldots, 0)$ représente la situation dans laquelle chaque avion se voit attribuer son RFL.

Il pourra être utile d'observer que les sommets du graphe des conflits choisis par la régulation r portent les numéros $3k + r_k$ pour $0 \leqslant k < n$. On remarque également qu'au sommet s du graphe correspond le niveau relatif $r_k = s \mod 3$ et le vol v_k tel que $k = \lfloor s/3 \rfloor$.

3. Écrire une fonction `int* nb_vols_par_niveau_relatif(int reg[], int n)` qui prend en paramètre une régulation (tableau de n entiers r_k) et le nombre de vols, et qui retourne un tableau de 3 entiers « `a, b et c` » dans lequel a est le nombre de vols à leurs niveaux RFL, b le nombre de vols au-dessus de leurs niveaux RFL et c le nombre de vols au-dessous de leurs niveaux RFL. La libération de la mémoire liée à l'allocation dynamique du tableau retourné sera à la charge de la fonction appelante.

On appelle *coût d'une régulation* la somme des couts des conflits potentiels que cette régulation engendre.

4. Écrire une fonction `int cout_regulation(int reg[], int n)` qui prend en paramètre un tableau de n entiers représentant une régulation et le nombre n de vols, et qui renvoie le coût de celle-ci.

5. Écrire une fonction `int* regulation_RLF(int n)` prenant en argument un nombre de vols et retournant un tableau représentant la régulation où tous les vols ont à leur niveau RFL. La libération de la mémoire liée à l'allocation dynamique du tableau retourné sera à la charge de la fonction appelante.

6. En déduire une fonction `int cout_RFL(int n)` qui renvoie le coût de la régulation pour laquelle chacun des n avions vole à son RFL.

7. Est-il envisageable de calculer les coûts de toutes les régulations possibles pour trouver celle de cout minimal (on attends un réponse justifiée) ?

1.4 L'algorithme Minimal

On définit le coût d'un sommet comme la somme des coûts des conflits potentiels dans lesquels ce sommet intervient. Par exemple, le cout du sommet correspondant au niveau RFL de l'avion A dans le graphe de la figure 1 est égal à $100 + 100 + 150 = 350$.

L'algorithme Minimal consiste à sélectionner le sommet du graphe de cout minimal ; une fois ce dernier trouvé, les deux autres niveaux possibles de ce vol sont supprimés

du graphe et on recommence avec le graphe modifié jusqu'à avoir attribué un niveau à chaque vol.

Lorsque l'on calcule la somme des coûts des conflits potentiels pour un sommet donné, on ne considère que les conflits potentiels avec des sommets qui n'ont pas encore été supprimés. Dans la pratique, plutôt que de supprimer effectivement des sommets du graphe, on utilise une tableau `etat_sommet` de $3n$ entiers tels que :

- `etat_sommet[s]` vaut 0 lorsque s désigne un sommet qui a été supprimé du graphe ;
- `etat_sommet[s]` vaut 1 lorsque s désigne un sommet choisi dans la régulation ;
- `etat_sommet[s]` vaut 2 lorsque s désigne un sommet qui n'a encore été ni choisi, ni supprimé.

8. Écrire une fonction `int cout_du_sommet(int s, int etat_sommet[], int n)` qui prend en paramètres un numéro de sommet s (n'ayant pas été supprimé) ainsi que le tableau `etat_sommet` et le nombre de vols, et qui renvoie le cout du sommet s dans le graphe défini par la variable globale `conflit` et le paramètre `etat_sommet`.

9. Écrire une fonction `int sommet_de_cout_min(int etat_sommet[], int n)` qui, parmi les sommets qui n'ont pas encore été choisis ou supprimés, renvoie le numéro du sommet de coût minimal (si plusieurs sommets sont de coût minimal, vous pouvez en choisir un librement parmi ces derniers).

10. En déduire une fonction `int* minimal(int n)` qui renvoie la régulation résultant de l'application de l'algorithme Minimal. Là encore, la libération de la mémoire allouée pour le stockage du tableau résultat est à la charge de la fonction appelante.

11. Quelle est sa complexité ? Commenter.

1.5 Recuit simulé

Une autre solution pour trouver une régulation raisonnablement bonne consiste à utiliser un algorithme dit de *recuit simulé*, s'inspirant de la façon dont les métaux, en se refroidissant, tendent à se diriger vers un état d'énergie aussi faible que possible.

Dans le cadre de notre problème, l'algorithme de *recuit simulé* part d'une régulation initiale quelconque (par exemple la régulation pour laquelle chacun des avions vole à son RFL) et d'une valeur positive flottante T choisie empiriquement.

Il réalise un nombre fini d'étapes se déroulant ainsi :

- un vol v_k est tiré au hasard ;
- on modifie r_k en tirant au hasard parmi les deux autres valeurs possibles ;
  - si cette modification diminue le cout de la régulation, cette modification est conservée ;
  - sinon, cette modification n'est conservée qu'avec une probabilité $p = \exp(-\Delta c/\mathrm{T})$ où Δc est l'augmentation de coût liée à la modification de la régulation ;
- le paramètre T est diminué d'une certaine quantité.

On fournit une fonction `int rand_i(int p)` qui retourne un entier choisi aléatoirement, avec une distribution uniforme, dans $[\![0 \,..\, p-1]\!]$, et d'une fonction `double rand_d()` retournant un flottant choisi aléatoirement, avec une distribution uniforme, dans $[0,1]$.

On dispose également d'une fonction `double exp(double x)` permettant de calculer $\exp(x)$.

12. Écrire une fonction `void etape_recuit(int reg[], int n, double T)` qui effectue les deux premières opérations d'une itération de l'algorithme de recuit simulé, telles que décrites ci-dessus. La fonction ne retourne rien, mais elle modifie éventuellement le contenu du tableau `reg` si les conditions sont atteintes.

13. En déduire une fonction `int* recuit(int n)` qui prend en argument le nombre de vols et retourne un tableau représentant une régulation obtenue avec l'algorithme de recuit simulé. On fera débuter l'algorithme avec la valeur T = 1000 et une régulation où tous les vols sont à leur RFL. À chaque étape, la valeur de T sera diminuée de 1 %. L'algorithme se terminera lorsque T < 1.0. La libération de la mémoire éllouée pour le tableau retourné est à la charge de la fonction appelante, mais on prendra garde à ce qu'il n'y ait pas d'autre fuite de mémoire.

Remarque : dans la pratique, l'algorithme de recuit simulé est fréquemment appliqué plusieurs fois de suite en partant à chaque fois de la régulation obtenue à l'étape précédente, jusqu'à ne plus trouver d'amélioration notable.

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

2.1 Implémentation

On s'intéresse, dans ce problème à un tri appelé « tri faire-valoir » (ou Stooge sort en anglais, en hommage à la troupe comique américaine *The Three Stooges*), permettant de trier en place les éléments d'un tableau par ordre croissant. Son principe est le suivant :

- si le tableau est de taille 1, il n'y a rien à faire ;
- si le tableau est de taille 2, on échange les deux éléments si le premier est plus grand que le second ;
- si le tableau est de taille $n > 2$,
  - on trie les $\lceil 2n/3 \rceil$ premières cases du tableau de manière récursive ;
  - on trie les $\lceil 2n/3 \rceil$ dernières cases du tableau de manière récursive ;
  - on trie à nouveau les $\lceil 2n/3 \rceil$ premières cases du tableau de manière récursive.

1. Proposer une fonction `void sort2(int arr[])` prenant en argument l'adresse d'un tableau à deux éléments et le triant en place par ordre croissant.

2. En déduire une fonction `void stooge_sort(int arr[], int n)` prenant en argument l'adresse d'un tableau d'entiers `arr` ainsi que sa taille `n` (supposée strictement positive) et appliquant l'algorithe de tri présenté.

2.2 Analyse

On cherche à présent à justifier le bon fonctionnement de cette approche, ainsi que sa complexité. On s'intéresse tout d'abord à sa correction.

On considère un tableau de taille $n > 2$. L'algorithme considéré effectue donc trois tris successifs sur des parties du tableau.

3. Après le premier de ces trois tris, y a-t-il des éléments bien placés, et si oui lesquels (on justifiera la réponse) ?

4. Après le second de ces trois tris, y a-t-il des éléments bien placés, et si oui lesquels (on justifiera la réponse) ?

5. En déduire la correction partielle de la méthode.

6. Justifier avec soin que l'algorithme de tri termine.

7. Le tri est-il stable ? On justifiera la réponse.

On note u_n le nombre de comparaisons effectuées lorsque l'on trie un tableau de taille n avec le tri faire-valoir.

8. Déterminer une relation de récurrence sur u_n.

9. En déduire une estimation de u_n. Que penser de l'efficacité de ce tri ?

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

3.1 Introduction

Dans ce problème, on s'intéresse aux fonctions $f : \mathcal{E}_n \mapsto \mathcal{E}_n$ où $\mathcal{E}_n$ est l'ensemble des entiers $\{0, 1, ..., n-1\}$.

On appelle *point fixe de f* tout entier i de $\mathcal{E}_n$ vérifiant $f(i) = i$.

On note f^k l'*itérée k^e* de f. Par exemple, f^3 sera la fonction $\begin{cases} \mathcal{E}_n \longmapsto \mathcal{E}_n \\ i \longmapsto f(f(f(i))) \end{cases}$

On représentera une fonction $f : \mathcal{E}_n \mapsto \mathcal{E}_n$ par une fonction OCaml de signature `int -> int`. On admettra que pour tout i entre 0 et $n-1$, `f i` est bien un entier entre 0 et $n-1$. Il n'est jamais besoin de le vérifier.

Par exemple, la fonction f_0 qui à $i \in \mathcal{E}_{10}$ associe $2i + 5[10]$, définie ci-dessous, est une fonction de $\mathcal{E}_{10} \mapsto \mathcal{E}_{10}$.

```
# let f_0 i = (2 * i + 5) mod 10;;

val f_0 : int -> int = <fun>
```

3.2 Attracteurs

1. Proposer une fonction f_1 bijective de $\mathcal{E}_4$ vers $\mathcal{E}_4$ qui ne possède aucun point fixe.

2. Combien existe-t-il de telles fonctions bijectives de $\mathcal{E}_4$ vers $\mathcal{E}_4$ sans point fixe ?

3. Écrire une fonction `has_fixed_point` de signature `(int -> int) -> int -> bool` prenant en argument une fonction f et un entier n et qui renvoie `true` si la fonction f admet un point fixe et `false` sinon. Par exemple, « `has_fixed_point f_0 10` » devra renvoyer `true`, puisque 5 est un point fixe de f_0, et « `has_fixed_point f_1 4` », `false`.

4. Écrire une fonction `iter` de signature `(int -> int) -> int -> int -> int` qui prend en premier argument une fonction f, en second un entier i de $\mathcal{E}_n$ et en troisième un entier naturel k et renvoie $f^k(i)$.

On dit que la fonction f possède un *attracteur* si, pour tout $x \in \mathcal{E}_n$, il existe un $k \geqslant 0$ tel que $f^k(x)$ est un point fixe de f. On pourra vérifier que la fonction f_2 définie sur $\mathcal{E}_7$ par les relations ci-dessous possède un attracteur :

$$f_2(0) = 5, \quad f_2(1) = 1, \quad f_2(2) = 2, \quad f_2(3) = 1, \quad f_2(4) = 0, \quad f_2(5) = 2 \quad \text{et} \quad f_2(6) = 3$$

En revanche, on notera que la fonction f_0 donnée en introduction n'admet pas d'attracteur puisque $f_0^k(1)$ n'est jamais un point fixe de f quelle que soit la valeur de k.

5. Décrire précisément une méthode permettant de vérifier qu'une fonction f admet un attracteur. On justifiera que cette méthode fonctionne.

6. Déduire de la méthode proposée une fonction `has_attractor` de signature `(int -> int) -> int -> bool` qui prend en argument une fonction f et un entier n et renvoie `true` si et seulement si la fonction f définie sur $\mathcal{E}_n$ admet un attracteur et `false` sinon. **On ne demande pas la meilleure complexité possible pour cette fonction.**

7. Quelle est la complexité de la fonction précédente ?

On suppose dans les deux questions suivantes que la fonction f admet un attracteur. Le *temps de convergence* de f en $i \in \mathcal{E}_n$ est le plus petit entier $k \geqslant 0$ tel que $f^k(i)$ soit un point fixe de f. Pour la fonction f_2, le temps de convergence en 4 est 3 car $f_2(4) = 0$, $f_2^2(4) = 5$, $f_2^3(4) = 2$ et 2 est un point fixe.

8. Écrire une fonction `time_conv` de signature `(int -> int) -> int -> int` qui prend en premier argument une fonction f et en second un entier i de $\mathcal{E}_n$ et qui renvoie, **en temps linéaire**, le temps de convergence de f en i.

3.3 Recherche efficace de points fixes.

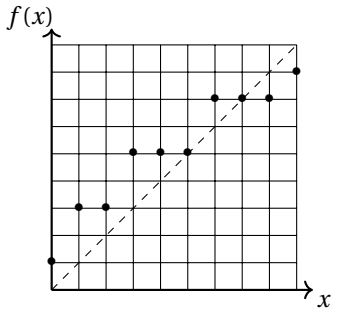
On cherche à présent à écrire une fonction `fixed_point` de signature `(int -> int) -> int -> int` prenant en argument une fonction f et n retournant *un quelconque* point fixe de la fonction f de $\mathcal{E}_n$ dans $\mathcal{E}_n$ (la fonction peut posséder plusieurs points fixes, on ne cherche ici à en exhiber qu'un seul).

Pour une fonction quelconque, la complexité de la fonction `fixed_point` sera au mieux linéaire dans le pire des cas. On s'intéresse dans la suite à des améliorations possibles de cette complexité lorsque la fonction considérée possède certaines propriétés spécifiques.

On s'intéresse dorénavant au cas d'une **fonction croissante** de $\mathcal{E}_n$ dans $\mathcal{E}_n$ pour l'ordre usuel $\leqslant$.

On rappelle qu'une fonction $f : \mathcal{E} \longmapsto \mathcal{E}$ est croissante pour l'ordre usuel si et seulement si pour tous $x, y \in \mathcal{E}^2$ tels que $x \leqslant y$, on a $f(x) \leqslant f(y)$.

À titre d'exemple, la fonction f_3 sur $\mathcal{E}_{10}$, dont le graphe est donné ci-contre, est croissante.



9. Écrire un prédicat `incr` de signature `(int -> int) -> int -> bool` prenant en argument une fonction $f : \mathcal{E}_n \longmapsto \mathcal{E}_n$ et n, et renvoyant un booléen indiquant si f est croissante sur $\mathcal{E}_n$. On impose un temps de calcul de complexité **linéaire** en n (on ne demande pas de justifier la complexité).

10. Montrer qu'une fonction croissante d'un ensemble $\mathcal{E}_n$ dans ce même ensemble $\mathcal{E}_n$ admet toujours au moins un point fixe.

11. Écrire une fonction `fixed_point` de signature `(int -> int) -> int -> int` prenant en argument une fonction croissante $f : \mathcal{E}_n \longmapsto \mathcal{E}_n$ et n. Cette fonction retourne un entier i de $\mathcal{E}_n$ tel que $f(i) = i$. On impose un temps de calcul de complexité **logarithmique** (on ne demande pas ici de justifier la complexité).

12. Exhiber une propriété qui reste vraie lors de tous les appels récursifs dans la fonction précédente permettant de justifier sa correction.

13. Démontrer soigneusement que la fonction termine en proposant un variant, et justifier brièvement que sa complexité est logarithmique.

On peut généraliser la notion de fonction croissante comme suit. On rappelle qu'une relation binaire $\preceq$ sur un ensemble $\mathcal{E}$ est une relation d'ordre si et seulement si elle est réflexive ($x \preceq x$ pour tout $x \in \mathcal{E}$), anti-symétrique (pour tous $x, y \in \mathcal{E}^2$, si $x \preceq y$ et $y \preceq x$, alors $x = y$) et transitive (pour tous $x, y, z \in \mathcal{E}^3$, si $x \preceq y$ et $y \preceq z$ alors $x \preceq z$).

Soit $\preceq$ une relation d'ordre (pas nécéssairement totale) sur un ensemble $\mathcal{E}$. Une fonction $f : \mathcal{E} \longmapsto \mathcal{E}$ est *croissante au sens de* $\preceq$ si et seulement si, pour tous $x, y \in \mathcal{E}^2$ tels que $x \preceq y$, on a $f(x) \preceq f(y)$.

On dit qu'un élément m de $\mathcal{E}$ est un *plus petit élément* de $\mathcal{E}$ au sens de $\preceq$ si et seulement si pour tout x de $\mathcal{E}$, $m \preceq x$.

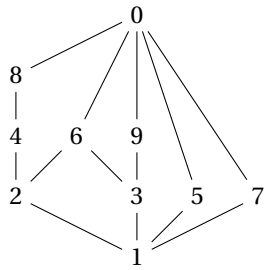
Soit une relation d'ordre $\preceq$ sur $\mathcal{E}_n$ admettant un plus petit élément m au sens de $\preceq$. Soit $f : \mathcal{E}_n \longmapsto \mathcal{E}_n$ une fonction croissante au sens de $\preceq$.

14. Montrer que la suite dans $\mathcal{E}_n$ $m, f(m), f^2(m), \ldots$ est croissante pour $\preceq$.

15. En déduire qu'il existe un entier $k > 0$ tel que $f^k(m)$ est un point fixe de f dans $\mathcal{E}_n$.

16. Démontrer que $f^k(m)$ est en fait le plus petit point fixe de f au sens de $\preceq$.

Nous nous intéressons maintenant à un choix particulier d'ordre $\preceq$ appelé *ordre de divisibilité* et noté $|$. On note $a|b$ la relation d'ordre « a divise b » sur les entiers positifs, vraie si et seulement si il existe un entier $c \geqslant 0$ tel que $ca = b$. Ainsi, l'ensemble $\mathcal{E}_{10}$ ordonné par la divisibilité peut se représenter ainsi :



D'après la définition donnée précédemment, une fonction $f : \mathcal{E}_n \longmapsto \mathcal{E}_n$ croissante au sens de l'ordre de divisibilité est une fonction telle que pour tous $x, y \in \mathcal{E}_n^2$ tels que $x|y$, on a $f(x)|f(y)$.

Par exemple, la fonction f_4 telle que définie ci-dessous est croissante au sens de l'ordre de divisibilité :

$f(0) = 0$, $f_4(1) = 2$, $f_4(2) = 4$, $f_4(3) = 6$, $f_4(4) = 4$, $f_4(5) = 8$, $f_4(6) = 0$, $f_4(7) = 2$, $f_4(8) = 0$ et $f_4(9) = 6$

17. Proposer une fonction `div_incr` de signature `(int -> int) -> int -> bool` prenant en argument une fonction f sur $\mathcal{E}_n$ et n, et renvoyant un booléen indiquant si elle est croissante au sens de la divisibilité.

18. Quelle est sa complexité ?

19. En déduire une fonction `div_fixed_point` de signature `(int -> int) -> int -> int` prenant en argument une fonction f (croissante pour $|$ sur $\mathcal{E}_n$) et n, et retournant un point fixe de f en temps logarithmique en n.

20. Démontrer que la fonction termine et justifier que sa complexité est logarithmique.