Composition d'informatique n°6

Corrigé et remarques

⋆ ⋆ ⋆

Remarques

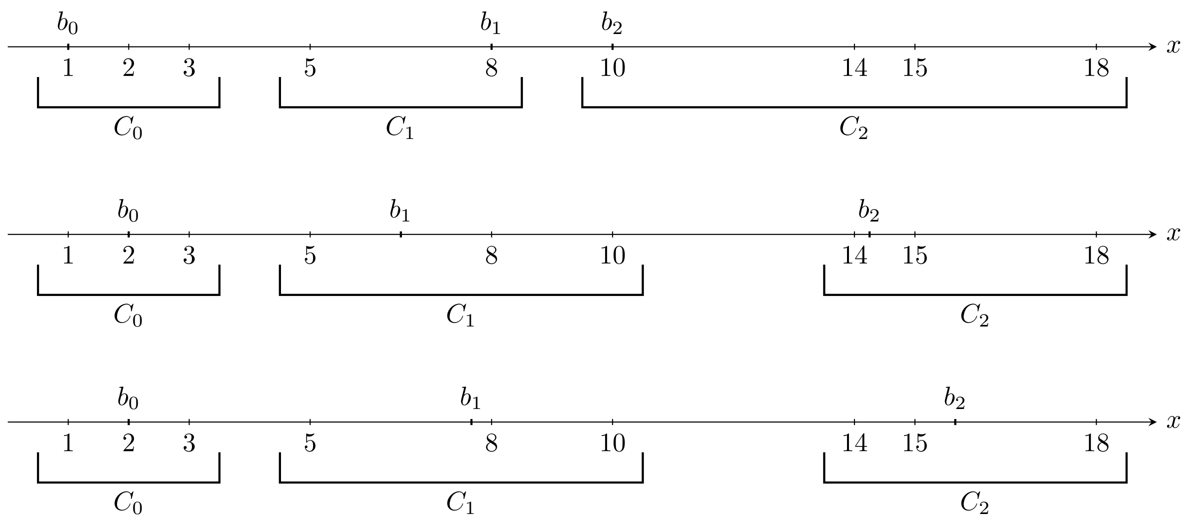
- Q1A. Une fois la solution proposée pour $K = N - 1$, il peut être bien de réécrire le score pour se convaincre qu'il est minimal.
- Q4A. Avant de diviser par `j-i` pour obtenir la moyenne, il n'est pas nécessaire de transtyper en `double`, car le numérateur est déjà un `double`.
- Q5A. Il ne faut pas faire plusieurs fois le calcul de la moyenne, sinon cela augmente inutilement la complexité.
- Q7A. Attention à ne pas oublier le dernier terme, dont le 3e argument est `N` (et pas `P[K - 1]`).
- Q8A. Un tableau initialisé sur la pile suffit pour calculer le score pour la partition courante, inutile de faire un `malloc` (surtout que le `free` est souvent oublié).
- Q8B. Deux boucles suffisent : pas besoin de la troisième car `P[0]` vaut 0 dans tous les cas.
- Q10A. La dernière classe est à traiter à part, à cause des valeurs dans la partition, qui terminent à `N`.
- Q10B. La valeur de `K` peut être égale à 2, il faut faire attention à éviter un dépassement des bornes du tableau dans ce cas.
- Q12A. Il faut faire attention aux fuites mémoires lorsqu'on calcule la fusion des classes et qu'on modifie le tableau courant.
- Q12B. Si la variable `P` a déjà été déclarée en dehors de la boucle, écrire `int* P = ...` est une erreur, car cela correspond à une nouvelle déclaration d'une variable existante.
- Q13A. La majoration ne doit pas être trop forte pour la fonction `classes_plus_proche` : la somme des tailles des tranches sur lesquelles sont calculées des moyennes est égale à N, ce qui donne une complexité totale en $\mathcal{O}(N)$ et pas $\mathcal{O}(K \times N)$.
- Q16A. Il faut justifier que l'indice qui permet d'atteindre le minimum est bien compris entre $k - 1$ et $n - 1$.
- Q17A. Attention, en programmation dynamique, il faut garder les résultats intermédiaires en mémoire ! C'est la base de cette technique de programmation ! Sans tableau ou dictionnaire, vous êtes sûr de passer à côté de ce qui est attendu...
- Q23A. Attention à respecter les règles d'inférence autorisées par le sujet. En particulier, le raisonnement par l'absurde ne faisait pas partie de la liste des règles.
- Q33A. On attend une justification que seule cette règle peut être appliquée et pas les autres règles.
- Q33B. La justification à donner doit être faite par arbre de preuve de jugement de typage, pas avec les mains.

1 Clustering en dimension 1

1.1 Préliminaires

Question 1 Si $K = N$, chaque classe d'équivalence est de cardinal 1 (sinon l'une est vide) et le score est nul, ce qui est bien minimal. Si $K = N - 1$, toutes les classes sont de cardinal 1, sauf une qui est de cardinal 2 (par le principe des tiroirs). Pour minimiser le score, il faut mettre dans la même classe les deux éléments les plus proches de E.

Question 2 On a l'exécution suivante :



Question 3 Les erreurs sont :

- ligne 5 : il faut vérifier que ni `i1`, ni `i2` n'a atteint la dernière valeur. On doit remplacer la condition booléenne par `if (i2 == n2 || (i1 < n1 && tab1[i1] <= tab2[i2])){`
- entre les lignes 9 et 10 : il faut penser à incrémenter `i2` en rajoutant `i2++;`
- ligne 15 : le cas d'arrêt doit aussi prendre en compte le cas du tableau à un seul élément, sinon l'algorithme ne termine pas. Il faut remplacer cette ligne par `if (n <= 1) return;`
- entre les lignes 26 et 27 : il faut penser à libérer la mémoire. Il faut rajouter `free(tab1); free(tab2);`.

Question 4 Il suffit juste de faire attention aux indices.

```c
double moyenne(double* E, int i, int j){
    double mu = 0;
    for (int k=i; k<j; k++){
        mu += E[k];
    }
    return mu / (j - i);
}
```

Question 5 Le code ressemble au précédent en faisant d'abord le calcul de la moyenne.

```
double somme_emc(double* E, int i, int j){
    double mu = moyenne(E, i, j);
    double S = 0;
    for (int k=i; k<j; k++){
        S += (E[k] - mu) * (E[k] - mu);
    }
    return S;
}
```

Question 6 Le tableau associé à la partition $\mathcal{P} = \{\{0, 1, 2\}, \{3, 4\}, \{5, 6, 7\}, \{8\}\}$ est `{0, 3, 5, 8}`. La partition associée au tableau `{0, 1, 4, 5}` est $\mathcal{P} = \{\{0\}, \{1, 2, 3\}, \{4\}, \{5, 6, 7, 8\}\}$.

Question 7 Il faut ici faire attention à la manière dont sont délimitées les classes d'équivalence. On traite la dernière classe à part, qui contient les valeurs jusqu'à x_{N-1}.

```
double score(double* E, int N, int* P, int K){
    double SP = somme_emc(E, P[K - 1], N);
    for (int i=0; i<K-1; i++){
        SP += somme_emc(E, P[i], P[i + 1]);
    }
    return SP;
}
```

Question 8 Il suffit d'une double boucle pour déterminer les indices des plus petits éléments de C_1 et C_2. On calcule le score à chaque étape et on garde la partition de score minimal. On fait attention à différencier l'usage d'un tableau statique et d'un tableau dynamique (pour pouvoir renvoyer le tableau, il faut l'allouer sur le tas).

```
int* clustering3(double* E, int N){
    int Pmin = malloc(3 * sizeof(int));
    Pmin[0] = 0; Pmin[1] = 1; Pmin[2] = 2;
    for (int i=1; i<N-1; i++){
        for (int j=i+1; j<N; j++){
            int P[3] = {0, i, j};
            if (score(E, N, P, 3) < score(E, N, Pmin, 3)){
                Pmin[1] = i; Pmin[2] = j;
            }
        }
    }
    return Pmin;
}
```

Question 9 On donne les complexités des fonctions précédentes :

- `moyenne(E, i, j)` est en $\mathcal{O}(j - i)$, au même titre que `somme_emc(E, i, j)` ;
- `score(E, N, P, K)` est en $\mathcal{O}(N)$, car on calcule `somme_emc` pour chaque classe, dont la somme des cardinaux est N (attention à ne pas faire une analyse trop grossière ici) ;
- on en déduit que `clustering3(E)` est en $\mathcal{O}(N^3)$, car on fait le calcul d'un score de l'ordre de $\mathcal{O}(N^2)$ fois. La création et copie des tableaux n'est pas à prendre en compte (car ce sont des tableaux de taille 3).

1.2 Clustering hiérarchique ascendant

Question 10 On garde en mémoire l'indice i_{opt} et l'écart de moyenne entre $C_{i_{\text{opt}}}$ et $C_{i_{\text{opt}}+1}$, en commençant par la fin (pour éviter à gérer le cas particulier de la dernière tranche dans la boucle). Ensuite, on parcourt tous les indices de $K-3$ à 0 et on vérifie si l'écart est inférieur pour éventuellement mettre à jour.

```
int classes_plus_proches(double* E, int N, int* P, int K){
    int iopt = K - 2;
    double moy1 = moyenne(E, P[K - 2], P[K - 1]);
    double ecartopt = moyenne(E, P[K - 1], N) - moy1;
    for (int i=K-3; i>=0; i--){
        double moy = moyenne(E, P[i], P[i + 1]);
        double ecart = moy1 - moy;
        if (ecart <= ecartopt){
            iopt = i;
            ecartopt = ecart;
        }
        moy1 = moy;
    }
    return iopt;
}
```

À noter, on aurait pu partir de la fin du tableau pour éviter les tests supplémentaires.

Question 11 On crée un nouveau tableau, qu'on remplit avec les éléments d'indice différent de `iopt + 1`.

```
int* fusion_classes(int* P, int K, int iopt){
    int* nouvP = malloc((K - 1) * sizeof(int));
    for (int i=0; i<K; i++){
        if (i <= iopt){
            nouvP[i] = P[i];
        } else if (i > iopt + 1){
            nouvP[i - 1] = P[i];
        }
    }
    return nouvP;
}
```

Question 12 On se contente d'appliquer l'algorithme décrit précédemment avec les fonctions déjà écrites. On pense à libérer la mémoire du tableau `P` lorsqu'on fusionne des classes.

```
int* CHA(double* E, int N, int K){
    int* P = malloc(N * sizeof(int));
    for (int i=0; i<N; i++) P[i] = i;
    int taille = N;
    while (taille > K){
        int iopt = classes_plus_proches(E, N, P, taille);
        int* nouvP = fusion_classes(P, taille, iopt);
        free(P);
        P = nouvP;
        taille--;
    }
    return P;
}
```

Question 13

– La fonction `classe_plus_proches` calcule de l'ordre de K moyennes, pour une somme des tranches d'indices égale à N. La complexité est donc en $\mathcal{O}(N)$.
– La fonction `fusion_classes` se contente de créer et remplir un tableau de taille $K-1$, donc en $O(K)$ (avec $K \leqslant N$).
– Finalement, la fonction `CHA` fait appel aux deux fonctions précédentes, pour $k \in [\![K, N]\!]$, soit une complexité en $O((N-K) \times N)$.

Question 14 On considère $N = 4$ et $K = 2$. On pose $E = \{0, 2, 4, 7\}$. L'algorithme de CHA donnera les classes $\mathcal{P}_1 = \{\{0, 2, 4\}, \{7\}\}$. Avec une telle partition, on aurait un score $S(\mathcal{P}_1) = (4+4)+0 = 8$. Cependant, avec la partition $\mathcal{P}_2 = \{\{0, 2\}, \{4, 7\}\}$, on obtient un score de $S(\mathcal{P}_2) = (1+1)+(2,25+2,25) = 6,5$.

1.3 Solution optimale en programmation dynamique

Question 15 Lorsque $k = 1$, $D(n, k) = S_{\text{emc}}(0, n)$ (il n'y a qu'une seule classe). Ce score peut être calculé en $\mathcal{O}(n)$.

Question 16 Une partition des n éléments de taille k consiste en une partition de $i < n$ éléments en $k-1$ classes, à laquelle on rajoute une classe formée des $n-i$ derniers éléments. Comme aucune classe ne doit être vide, on considère $i \geqslant k-1$. La partition optimale atteint le minimum parmi toutes les partitions possibles de cette forme, ce qui donne bien la formule voulue.

Question 17 On écrit une fonction auxiliaire `cluster_rec` qui prend en argument l'ensemble E, des entiers n et k et un dictionnaire (ici codé par une matrice) mémoïsant les résultats, et renvoie $D(n, k)$. L'initialisation et l'hérédité se font selon les deux questions précédentes.

```
double cluster_rec(double* E, int n, int k, double** dic){
    if (dic[n][k] < 0){
        if (k == 1){
            dic[n][k] = somme_emc(E, 0, n);
        } else {
            dic[n][k] = cluster_rec(E, k - 1, k - 1, dic) + somme_emc(E, k - 1, n);;
            for (int i=k-1; i<n; i++){
                double d = cluster_rec(E, i, k - 1, dic) + somme_emc(E, i, n);
                if (d < dic[n][k]){
                    dic[n][k] = d;
                }
            }
        }
    }
    return dic[n][k];
}
```

Une fois cette fonction écrite, il suffit de lancer un appel avec $n = N$ et $k = K$, en utilisant un dictionnaire vide. On pense à libérer la mémoire avant de renvoyer la valeur.

```c
double clustering_dynamique(double* E, int N, int K){
    double** dic = malloc((N + 1) * sizeof(double*));
    for (int n=0; n<=N; n++){
        dic[n] = malloc((K + 1) * sizeof(double));
        for (int k=0; k<=K; k++){
            dic[n][k] = -1;
        }
    }
    double d = cluster_rec(E, N, K, dic);
    for (int n=0; n<=N; n++){
        free(dic[n]);
    }
    free(dic);
    return d;
}
```

Question 18 On remarque qu'il y a de l'ordre de $N \times K$ valeurs qui sont calculées dans le dictionnaire. De plus, chaque valeur nécessaire de calculer le minimum par la boucle `for`. Cette boucle, de taille $n-k$, fait un appel à `somme_emc`, de complexité $\mathcal{O}(n-i)$. En combinant tout ça, on obtient une complexité totale en $\mathcal{O}(K \times N^3)$.

Question 19 Dans le calcul du minimum, on peut garder en mémoire l'indice `i` qui permet d'atteindre ce minimum, ce qui correspond au plus petit élément de la classe C_{k-1} dans une partition de taille k. On peut alors reconstruire une solution complète en utilisant les valeurs présentes dans le dictionnaire.

Question 20 On remarque les formules suivantes :

- $\mu(i, i+1) = x_i$;
- $\mu(i, n+1) = \dfrac{(n-i)\mu(i,n) + x_n}{n+1-i}$.

Ces formules peuvent être utilisées pour calculer tous les $\mu(i, n)$, $i < n$, en temps $\mathcal{O}(N^2)$ au total. Dès lors, on remarque, en adaptant la formule admise que :

- $S_{\text{emc}}(i, i+1) = 0$;
- $S_{\text{emc}}(i, n+1) = S_{\text{emc}}(i, n) + \frac{n-i}{n+1-i}(x_n - \mu(i,n))^2$.

À nouveau, on peut utiliser ces formules pour calculer les $S_{\text{emc}}(i, n)$ en temps $\mathcal{O}(N^2)$ au total.

2 Correspondance de Curry-Howard

On considère un ensemble fini de variables $\mathcal{V} = \{x_0, x_1, \ldots, x_{n-1}\}$, avec $n \in \mathbb{N}$. On définit l'ensemble des formules propositionnelles $\mathcal{F}$ de variables $\mathcal{V}$ par induction par :

- $\perp \in \mathcal{F}$;
- pour $x \in \mathcal{V}$, $x \in \mathcal{F}$;
- si $A, B \in \mathcal{F}$, alors $A \wedge B$, $A \vee B$ et $A \to B$ sont dans $\mathcal{F}$.

On remarque en particulier que la négation ne fait pas partie de la définition par induction des formules.

On rappelle en annexe les règles d'inférence des logiques minimale, intuitionniste et classique. On notera $\Gamma \vdash_m A$, $\Gamma \vdash_i A$ et $\Gamma \vdash_c A$ pour indiquer qu'un séquent $\Gamma \vdash A$ est prouvable en logique minimale, intuitionniste et classique respectivement.

On admet et on pourra utiliser le fait que la logique classique est correcte et complète pour la sémantique booléenne usuelle.

2.1 Premières preuves

Question 21 Il suffit de remarquer qu'elles correspondent à l'application de $\to_i$ et $\to_e$ dans le cas particulier où $B = \bot$.

$$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash A \to \bot = \neg A}\to_i \quad \text{et} \quad \dfrac{\Gamma \vdash A \to \bot \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\to_e$$

Question 22 On a la preuve suivante, en posant $\Gamma = \{A, A \to B\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash A}(\text{ax}) \qquad \dfrac{}{\Gamma \vdash A \to B}(\text{ax})}{\Gamma \vdash B}\to_e}{A \vdash (A \to B) \to B}\to_i}{\vdash A \to (A \to B) \to B}\to_i$$

Question 23 On a la preuve suivante :

$$\dfrac{\dfrac{}{\Gamma \vdash A \vee \neg A}(\text{te}) \qquad \dfrac{}{\Gamma, A \vdash A}(\text{ax}) \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, \neg A \vdash \neg A}(\text{ax}) \qquad \dfrac{\Gamma \vdash \neg\neg A}{\Gamma, \neg A \vdash \neg\neg A}(\text{aff})}{\Gamma, \neg A \vdash \bot}\neg_e}{\Gamma, \neg A \vdash A}\bot_e}{\Gamma \vdash A}\vee_e$$

2.2 Du typage pour faire des preuves

2.2.1 Vérification des règles d'inférence

Question 24 On propose simplement :

```
let et_elim (a, b) = a
```

Question 25 On a :

```
let ou_elim aob aic bic = match aob with
    | G a -> aic a
    | D b -> bic b
```

Cela correspond à la règle $\vee_e$. Dans le noms de variables, le `o` correspond à $\vee$ (ou) et `i` correspond à $\to$ (implique).

2.2.2 Quelques preuves en logique minimale

Question 26 On propose :

```
let q26 aiaib a = aiaib a a
```

Cela montre la prouvabilité de $(A \to A \to B) \to A \to B$. Avec $B = \bot$, on obtient le résultat attendu (car $\neg A = A \to \bot$).

Question 27 On reconnaît les lois de Morgan. On a :

```
let dm1 ((non_a, non_b) : ('a non, 'b non) et) = function
    | G a -> non_a a
    | D b -> non_b b
```

et

```
let dm2 (non_aob : ('a, 'b) ou non) : ('a non, 'b non) et =
    (fun a -> non_aob (G a)), fun b -> non_aob (D b)
```

L'arbre de preuve demandé est, en notant $\Gamma = \{\neg A \wedge \neg B, A \vee B\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A \vdash A}\,(\text{ax}) \quad \dfrac{\dfrac{}{\Gamma, A \vdash \neg A \wedge \neg B}\,(\text{ax})}{\Gamma, A \vdash \neg A}\,\wedge_e}{\Gamma, A \vdash \bot}\,\neg_e \quad \dfrac{\dfrac{}{\Gamma, B \vdash B}\,(\text{ax}) \quad \dfrac{\dfrac{}{\Gamma, B \vdash \neg A \wedge \neg B}\,(\text{ax})}{\Gamma, B \vdash \neg B}\,\wedge_e}{\Gamma, B \vdash \bot}\,\neg_e \quad \dfrac{}{\Gamma \vdash A \vee B}\,(\text{ax})}{\dfrac{\Gamma \vdash \bot}{\neg A \wedge \neg B \vdash \neg(A \vee B)}\,\neg_i}\,\vee_e}{\vdash (\neg A \wedge \neg B) \to \neg(A \vee B)}\,\to_i$$

2.2.3 Logique intuitionniste et logique classique

Question 28 On propose :

```
let q28 (non_aib : ('a -> 'b) non) b : 'a =
    bot_elim (non_aib (fun x -> b))
```

Question 29 On propose :

```
let q29 (non_non_a : 'a non non) : 'a = match tiers_exclu non_non_a with
    | G a -> a
    | D non_a -> bot_elim (non_non_a non_a)
```

2.3 Logique et typage

Question 30 On a le jugement de typage $\vdash$ `List.map (fun x -> x + 1) : int list -> int list`

Question 31 On a la dérivation suivante :

$$\dfrac{\dfrac{}{\Gamma, x : \beta \vdash g : \beta \to \alpha}\,(\text{var}) \quad \dfrac{}{\Gamma, x : \beta \vdash x : \beta}\,(\text{var})}{\Gamma, x : \beta \vdash g\ x : \alpha}\,(\text{app})$$

Question 32 On obtient alors :

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{Question précédente}}{\Gamma, x : \beta \vdash g\ x : \alpha} \quad \dfrac{}{\Gamma, x : \beta \vdash f : \alpha \to (\beta \to \gamma)}\,(\text{var})}{\Gamma, x : \beta \vdash f\ (g\ x) : \beta \to \gamma}\,(\text{app}) \quad \dfrac{}{\Gamma, x : \beta \vdash x : \beta}\,(\text{var})}{\Gamma, x : \beta \vdash f\ (g\ x)\ x : \gamma}\,(\text{app})}{\Gamma \vdash (\texttt{fun}\ x \to f\ (g\ x)\ x) : \beta \to \gamma}\,(\text{abs})$$

Question 33 L'expression e n'est ni une constante du langage, ni une variable, ni de la forme $\texttt{fun}\ x \to e'$. On en déduit que dans une dérivation de jugement de e, la dernière règle à utiliser serait nécessairement (app). On aurait donc un arbre de dérivation dont la racine et ses deux enfants sont de la forme :

$$\dfrac{\Gamma \vdash (\texttt{fun}\ h \to h\ 1\ 2) : \alpha \to \tau \qquad \Gamma \vdash (\texttt{fun}\ x \to 3) : \alpha}{\Gamma \vdash (\texttt{fun}\ h \to h\ 1\ 2)\ (\texttt{fun}\ x \to 3) : \tau}\ (\text{app})$$

où les prémisses peuvent être dérivées.

Question 34 Pour des raisons similaires à la question précédente, la dernière règle à utiliser serait nécessairement (abs). On aurait alors :

$$\dfrac{\Gamma, x : \sigma \vdash 3 : \beta}{\Gamma \vdash (\texttt{fun}\ x \to 3) : \sigma \to \beta}\ (\text{abs})$$

On en déduit que $\alpha = \sigma \to \beta$. De plus, sachant que la seule règle permettant de typer 3 est $\dfrac{}{\Gamma, x : \sigma \vdash 3 : \texttt{int}}\ (\text{type})$, on en déduit que $\alpha = \sigma \to \texttt{int}$.

Question 35 À nouveau, seule la règle (abs) a pu s'appliquer. On aurait alors :

$$\dfrac{\Gamma, h : \sigma \to \texttt{int} \vdash h\ 1\ 2 : \tau}{\Gamma \vdash (\texttt{fun}\ h \to h\ 1\ 2) : (\sigma \to \texttt{int}) \to \tau}\ (\text{abs})$$

En continuant, on ne peut appliquer que la règle (app), et on obtient :

$$\dfrac{\Gamma, h : \sigma \to \texttt{int} \vdash h\ 1 : \gamma \to \tau \qquad \Gamma, h : \sigma \to \texttt{int} \vdash 2 : \gamma}{\Gamma, h : \sigma \to \texttt{int} \vdash h\ 1\ 2 : \tau}\ (\text{app})$$

On en déduit que $\gamma = \texttt{int}$, ce qui permet de remonter sur la prémisse de gauche, à nouveau uniquement avec (app) :

$$\dfrac{\Gamma, h : \sigma \to \texttt{int} \vdash h : \delta \to (\texttt{int} \to \tau) \qquad \Gamma, h : \sigma \to \texttt{int} \vdash 1 : \delta}{\Gamma, h : \sigma \to \texttt{int} \vdash h\ 1 : \texttt{int} \to \tau}\ (\text{app})$$

À nouveau, $\delta = \texttt{int}$. Finalement, comme h est une variable, son seul type possible est $\sigma \to \texttt{int}$. Cela implique que $\sigma = \texttt{int}$ et que $\texttt{int} = \texttt{int} \to \tau$, ce qui est absurde d'après l'hypothèse donnée dans l'énoncé.

⋆⋆⋆