A1 – Mots sans carré et mots sans cube

Dans ce sujet, les lettres d'un mot w de longueur n sur un alphabet Σ seront notées $w[0], \ldots, w[n-1]$. Un mot non-vide u est un *carré* s'il existe $v \in \Sigma^*$ tel que $u = v \cdot v$, et c'est un *cube* s'il existe $v \in \Sigma^*$ tel que $u = v \cdot v \cdot v$, où $\cdot$ dénote la concaténation.

Un mot $w \in \Sigma^*$ est *sans carré* si aucun de ses facteurs n'est un carré, et les mots sans cube sont définis de manière analogue. L'objet de ce sujet est de construire des mots sans carré et sans cube de longueur arbitrairement grande.

Question 0. Si l'alphabet Σ a une seule lettre, quelle est la plus grande longueur possible pour un mot sans carré ? pour un mot sans cube ?

Question 1. Si l'alphabet Σ a deux lettres, quelle est la plus grande longueur possible pour un mot sans carré ?

Jusqu'à la question 5, on considère l'alphabet $\Sigma = \{a, b\}$. On définit une fonction $h : \Sigma \to \Sigma^*$ par $h(a) = ab$ et $h(b) = ba$, et on l'étend à une fonction de Σ^* dans Σ^* de la façon suivante : pour $w = w[0] \cdots w[n-1]$ un mot de Σ^*, on note $h(w) := h(w[0]) \cdots h(w[n-1])$. En particulier, pour ϵ le mot vide, on a $h(\epsilon) := \epsilon$

Question 2. Démontrer que, pour tout mot $w \in \Sigma^*$, le mot $h(w)$ ne contient pas de facteur qui soit un cube et soit de longueur impaire.

Question 3. Démontrer que, pour tout mot $w \in \Sigma^*$, si $h(w)$ contient un facteur de longueur paire qui est un cube alors w contient un facteur qui est un cube.

Question 4. Conclure qu'il existe des mots sans cube arbitrairement longs sur l'alphabet $\Sigma = \{a, b\}$.

Question 5. Un *pré-cube* est un mot de la forme $v \cdot v \cdot v[0]$ où $v \in \Sigma^*$ est non-vide. Un mot est *sans pré-cube* si aucun de ses facteurs n'est un pré-cube. Expliquer pourquoi il existe des mots sans pré-cube arbitrairement longs sur l'alphabet $\Sigma = \{a, b\}$.

Question 6. Montrer qu'il existe des mots sans carré arbitrairement longs sur un alphabet de taille 4.

Question 7. Montrer qu'il existe des mots sans carré arbitrairement longs sur un alphabet de taille 3.

Question 8. Un *mot infini* sur l'alphabet Σ est une séquence infinie $w[0], \ldots, w[n], \ldots$ d'éléments de Σ. On dit qu'il est *sans carré* si aucun de ses facteurs finis n'est un carré, c'est-à-dire que pour tous $i < j$ dans $\mathbb{N}$ le mot $w[i], \ldots, w[j-1]$ n'est pas un carré.

Existe-t-il des mots infinis sans cube sur un alphabet de taille 2 ? des mots infinis sans carré sur un alphabet de taille 3 ?

Suite des questions

Question 9. On construit une suite de mots en posant $w_0 := a$ et $w_{i+1} = h(w_i)$ pour tout $i \in \mathbb{N}$, où h est défini juste avant la question 2. Écrire le pseudocode d'un algorithme qui, étant donné $n \in \mathbb{N}$, calcule w_n. Quelle est sa complexité en temps et en espace ?

Question 10. Proposer une autre définition par récurrence de la suite (w_i) et écrire le pseudocode d'un algorithme plus efficace pour calculer w_i étant donné i.

Corrigé

Question 0. Pour $\Sigma = \{a\}$, le mot a est sans carré, le mot aa est sans cube, et ce sont les plus longs possibles car aa est un carré et aaa est un cube.

Question 1. Pour $\Sigma = \{a, b\}$, les mots sans carré les plus longs sont de longueur 3. En effet, si l'on tente de construire un mot w sans carré le plus long possible, la première lettre $w[0]$ est a sans perte de généralité. Il faut que $w[0] \neq w[1]$ car sinon $w[0]w[1]$ est un carré, ainsi la deuxième lettre $w[1]$ est b. Pour la même raison, la troisième lettre $w[2]$ est a, et le mot aba est encore sans carré. Mais, pour la même raison, il faudrait que la quatrième lettre $w[3]$ soit b, mais alors on obtient $abab$ qui est un carré et donc n'est pas sans carré. On a donc un mot sans carré de longueur 3 (à savoir aba) et on a montré qu'il ne peut y avoir de mot sans carré de longueur strictement supérieure.

Question 2. *[L'application h est en fait un morphisme (notion hors-programme), plus spécifiquement celui qui définit la séquence de Thue-Morse [Wik19].]*

Indication : que dire des occurrences de facteurs aa ou bb dans $h(w)$? Réponse : ils commencent forcément à des positions impaires, puisque les facteurs de taille 2 commençant à des positions paires sont des images de h.

Indication : considérer le cas d'un facteur v^3 de longueur impaire où v contient deux caractères identiques successifs. Réponse : si $h(w)$ a un facteur v^3 où v est de taille impaire et contient un facteur aa ou bb, car alors l'occurrence suivante de v commence alors à une parité différente et contient donc le même facteur aa ou bb commençant à une parité différente, ce qui n'est pas possible d'après ce qui précède.

Maintenant, la seule possibilité restante pour un facteur impair est d'alterner entre a et b, et donc commencer et finir par le même caractère (en particulier pour les facteurs aaa et bbb). En considérant le facteur vvv on a donc deux successions du même caractère à la frontière entre les deux premiers vv, et entre les deux derniers vv, et ces occurrences démarrent encore à des parités différentes, ce qui est encore exclu comme au paragraphe précédent.

Autre argument combinatoire plus synthétique : supposons que $h(w)$ contienne un facteur de longueur impaire qui soit un cube. Soit n_a et n_b le nombre d'utilisations de a et de b dans ce facteur. Nécessairement, par définition de h, on a $\mid n_a - n_b \mid = 1$ Mais, si $h(w)$ est un cube, alors n_a et n_b sont forcément tous deux des multiples de 3, ce qui est impossible.

Question 3. *Indication : considérer le cas d'un tel facteur commençant à une position paire.* Si c'est le cas alors chaque occurrence de v dans le facteur v^3 de $h(w)$ correspond effectivement à un facteur v' dans w tel que $h(v') = v$, et comme h est clairement injectif cela témoigne que w contient également un facteur cube $(v')^3$.

Si le facteur cube commence à une position impaire alors on utilise le fait que chaque position impaire de $h(w)$ détermine la position précédente, par définition du fait d'être une image de h. Ainsi, le caractère précédent chaque occurrence de v dans w_{i+1} est le même, et on peut décaler d'un cran à

gauche pour avoir un facteur cube de longueur paire et commençant à une position paire, et ainsi se ramener au cas précédent.

Question 4. On définit simplement une suite en posant $w_0 = a$, qui est sans cube, et en appliquant itérativement h, qui fait strictement croître la longueur ; par les deux questions précédentes, chaque terme de la suite est sans cube, et les termes de la suite comportent des mots arbitrairement longs.

Question 5. Pour des facteurs de la forme $v \cdot v \cdot v[0]$ où v est de longueur impaire, on raisonne exactement comme à la question 2 (pour la version non combinatoire de l'argument), en notant que dans le raisonnement on a seulement utilisé le fait qu'on avait un facteur vv avec v de longueur impaire (si v contenait un facteur aa ou bb) ou l'interface entre les deux premiers facteurs v et les deux derniers (ainsi on aboutit déjà à la contradiction avec $v \cdot v[0]$). Pour les facteurs pairs, on raisonne exactement comme à la question 3 : s'il existe un facteur $v \cdot v \cdot v[0]$ dans $h(w)$ alors on peut en trouver un qui commence à une position impaire, on argumente alors qu'on a également un facteur de la forme $v \cdot v \cdot v[0]v[1]$ car les lettres en position paire d'un mot image de h déterminent la lettre suivante, et en appliquant l'inverse de h on obtient un facteur pré-cube dans w.

Question 6. *Indication : regrouper les lettres du mot deux par deux.*

Formellement, on considère l'alphabet $\Sigma' = \{\mathrm{l}_{aa}, \mathrm{l}_{ab}, \mathrm{l}_{ba}, \mathrm{l}_{bb}\}$ et l'opération ℓ qui à un mot w sur $\Sigma = \{a, b\}$ associe un mot $w' := \ell(w)$ sur Σ' défini par $w'[j] = \mathrm{l}_{w[j]w[j+1]}$ pour chaque $0 \leq j < |w|$. On peut montrer que si $w' := \ell(w)$ a un carré alors w a un pré-cube, en effet si $w'[p + j] = w'[p + j + d]$ pour un certain $p, d \in \mathbb{N}^*$ et pour tout $0 \leq j < d$, alors on en déduit que $w[p + j] = w[p + j + d]$ pour les mêmes $p, d \in \mathbb{N}^*$ et tout $0 \leq j < d$, c'est-à-dire qu'on a trouvé un facteur carré, et de plus le caractère $w[p + (d - 1) + d + 1]$ qui suit ce facteur carré dans w, c'est-à-dire le caractère $w[p + 2d]$, est le deuxième caractère de $w'[p + 2d - 1]$ et donc le deuxième caractère de $w'[p + d - 1]$. Par définition de w', ce deuxième caractère est égal au premier caractère de $w'[p + d]$ et donc au premier caractère de $w'[p]$ c'est-à-dire à $w[p]$, ainsi le facteur dans w est bien un pré-cube.

Ainsi, comme on a des mots arbitrairement grands sur $\Sigma = \{a, b\}$ sans pré-cube par la question précédente, alors on a des mots arbitrairement grands sans carré sur l'alphabet Σ'.

Question 7. On considère l'image du mot précédent dans l'alphabet $\Sigma'' = \{\mathrm{l}_{ab}, \mathrm{l}_{ba}, \mathrm{l}_{=}\}$ en substituant l_{aa} et l_{bb} par $\mathrm{l}_{=}$. Montrons que le résultat n'a pas de carré. Pour un facteur carré vv avec $|v| = 1$, d'après la question précédente la seule possibilité serait que $v = \mathrm{l}_{=}$, mais c'est impossible car cela impliquerait que le mot original w contient trois caractères consécutifs égaux, or on sait que ça ne peut être le cas d'une image de h d'après la question 2.

Pour un facteur carré vv plus grand, on remarque d'abord qu'on ne peut pas avoir deux fois de suite $\mathrm{l}_{=}$ dans vv, pour la même raison qu'au paragraphe précédent. Ensuite, on observe que pour chaque occurrence de $\mathrm{l}_{=}$ dans le facteur vv, sa préimage dans Σ' est déterminée par le caractère précédent et le caractère suivant (en restant dans le facteur vv). Spécifiquement, quand un caractère $\mathrm{l}_{=}$ est précédé par l_{xy} (resp, suivi par l_{st}) alors il s'agissait avant substitution de l_{yy} (resp., de l_{ss}). Ainsi, l'égalité des caractères autres que $\mathrm{l}_{=}$ entre les deux occurrences de v suffit à garantir que les occurrences de $\mathrm{l}_{=}$ ont les mêmes préimages dans le facteur correspondant pour le mot sur Σ' (car chacune d'entre elle est précédée ou suivie par une lettre qui n'est pas $\mathrm{l}_{=}$ =, vu que deux telles lettres ne peuvent se suivre). Ainsi, le mot sur Σ' contiendrait aussi un carré, ce qui est exclu.

[Les questions de 2 à 7 sont tirées de [Sal81], chapitre 1.]

Question 8 *Indication : Démontrer que pour tout $n \in \mathbb{N}^*$, on a $h(n) = w_{n-1}\overline{w_{n-1}}$, où la fonction $\overline{\cdot} : \Sigma \to \Sigma^*$ est définie par $\overline{a} := b$, $\overline{b} := a$, et $\overline{w} := \overline{w[0]} \cdots \overline{w[|w| - 1]}$ pour tout mot w.*

Sous-indication : montrer que pour tout mot $w \in \Sigma^*$ on a $\overline{h(w)} = h(\overline{w})$.

Démonstration de la sous-indication : par récurrence sur la longueur du mot :

— Initialisation : le cas du mot vide est trivial, et pour un mot de longueur 1 on a bien $\overline{h(a)} = \overline{ab} = ba = h(b) = h(\overline{a})$ et $\overline{h(b)} = \overline{ba} = ab = h(a) = h(\overline{b})$,

— Hérédité : pour un mot $w \in \Sigma^*$ écrit $w = w'x$ où $w' \in \Sigma^*$ et $x \in \Sigma$, on a $\overline{h(w)} = \overline{h(w')h(x)}$, ce qui par application de l'hypothèse de récurrence et de l'initialisation respectivement vaut $h(\overline{w'})h(\overline{x})$, soit $h(\overline{w'}\overline{x})$ soit $h(\overline{w'x})$ soit $h(\overline{w})$ ce qui conclut.

L'affirmation de l'indication se démontre ensuite immédiatement par récurrence :

— Initialisation : on a bien $w_1 = ab = w_0\overline{w_0}$.

— Hérédité : soit $n \in \mathbb{N}^*$ et supposons qu'on a $w_n = w_{n-1}\overline{w_{n-1}}$. On a alors $w_{n+1} = h(w_n) = h(w_{n-1}\overline{w_{n-1}}) = h(w_{n-1})h(\overline{w_{n-1}})$. Le premier facteur vaut w_n par définition, et en appliquant l'indication précédente au deuxième facteur on obtient : $h(\overline{w_{n-1}}) = \overline{h(w_{n-1})} = \overline{w_n}$

Ainsi, on peut voir qu'en définissant une suite de mots en commençant par a et en appliquant itérativement la fonction h, on définit bien un mot infini, puisque chaque terme de la suite est un préfixe du terme suivant. Ceci permet donc bien de définir un mot infini qui est sans cube (puisque tous ses préfixes finis le sont). L'argument pour les mots infinis sans carré sur un alphabet de taille 3 est ensuite simplement par application des questions 6 et 7 au mot infini résultant.

Autre argument, générique, par compacité (sans considération de la nature du morphisme h). On considère une suite de mots sans cube sur l'alphabet $\{a, b\}$, dont les longueurs tendent vers $+\infty$, ce qui existe par la question 4. À chaque mot, on fait correspondre un nombre réel dans $[0, 1]$ en l'interprétant comme une représentation en base 5 (on met les lettres 0 et 4 à part pour s'assurer de l'unicité des représentations). Cela donne une suite de réels dans $[0, 1]$, dont on peut extraire une sous-suite convergente. La représentation en base 5 de la limite est alors sans carré. En effet, chacun de ses préfixes est égal à partir d'un certain rang à un préfixe de chaque élément de la sous-suite choisie.

Question 9. On s'intéresse donc à la suite traitée dans les questions précédentes. On a notamment :

— $w_0 = a$ (donné)

— $w_1 = h(a) = a\ b$

— $w_2 = h(a)h(b) = ab\ ba$

— $w_3 = h(a)h(b)h(b)h(a) = abba\ baab$

[Point potentiellement à discuter : on convient de représenter les chaînes de caractères comme des tableaux dont on suppose qu'ils sont suffisamment grands.]

Simplement en suivant la définition de la suite, on peut alors écrire :

```
T[0][0] = "a"
Pour i de 0 à n exclu:
  Pour j de 0 à longueur(T[i]) exclu:
    Si T[i][j] = "a":
      T[i+1][2*j] = "a"
      T[i+1][2*j+1] = "b"
    Sinon:
      T[i+1][2*j] = "b"
      T[i+1][2*j+1] = "a"
    Fin si
  Fin pour
Fin pour
Renvoie T[n]
```

Il est évident par récurrence que $|w_n| = 2^n$, ainsi la complexité en temps de l'algorithme est en $O(2^n)$ et la complexité en espace est également en $O(2^n)$ puisque $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

On peut améliorer un peu la complexité en espace (mais seulement d'un facteur constant) en ne gardant à chaque étape que le tableau courant et le tableau précédent, ou même en ne gardant qu'un seul tableau que l'on lit à partir de la fin pour construire l'itération suivante (ainsi les deux itérations ne se marchent pas sur les pieds) :

```
T[0] = "a"
Pour i de 0 à n exclu:
  Pour j de longueur(T[i])-1 inclus à 0 inclus:
    Si T[j] = "a":
      T[2*j] = "a"
      T[2*j+1] = "b"
    Sinon:
      T[2*j] = "b"
      T[2*j+1] = "a"
    Fin si
  Fin pour
Fin pour
Renvoie T
```

Question 10. *Indication possible : construire des termes de la suite pour observer la récurrence (qui est celle de la question 8).*

Indication possible : on ne peut bien sûr pas espérer mieux que $O(2^n)$, mais on veut un algorithme dont le temps d'exécution soit de l'ordre de $O(|w_n|)$ et non de $O(|w_n|^2)$.

Indication possible : utiliser la caractérisation de la question 8.

Sur la base de la caractérisation de la suite prouvée à la question 8, on peut maintenant calculer w_n en étendant la longueur de la suite à chaque fois, ce qui donne :

```
T[0] = "a"
iterations = 0
depart = 1
Tant que iterations < n
  Pour j de 0 inclus à depart exclu:
    Si T[depart] = "a":
      T[depart+j] = "b"
    Sinon:
      T[depart+j] = "a'
    Fin si
  Fin pour
  iterations += 1
  depart *= 2
Fin tant que
```

La complexité en temps et en espace est à présent linéaire en la taille de w_n.

Références

[Sal81] Arto Salomaa. *Jewels of formal language theory.* 1981.

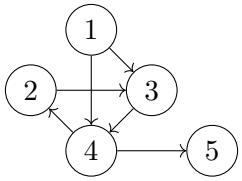
[Wik19] Wikipedia. Thue-morse sequence, 2019. https://en.wikipedia.org/wiki/Thue%E2%80%93Morse_sequence.

A2 – Familles closes de graphes

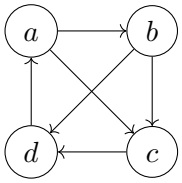
Un *graphe* $G = (V, E)$ est la donnée d'un ensemble fini non-vide de sommets V et d'un ensemble d'arêtes $E \subseteq V \times V$. On imposera toujours que $V \subseteq \mathbb{N}$, et on autorise les *boucles*, c'est-à-dire les arêtes de la forme (v, v) allant d'un sommet $v \in V$ vers lui-même. Un *homomorphisme* d'un graphe $G = (V, E)$ vers un graphe $G' = (V', E')$ est une fonction $\phi : V \to V'$ telle que, pour tout $(x, y) \in E$, on ait $(\phi(x), \phi(y)) \in E'$.

Question 0. Décrire un homomorphisme de G vers G'. Y a-t-il un homomorphisme de G' vers G ?

G



G'



Question 1. Caractériser en termes d'homomorphismes les graphes comportant des boucles.

Question 2. Écrire le pseudocode d'un algorithme naïf pour déterminer, étant donné la matrice d'adjacence de deux graphes G et G', s'il y a un homomorphisme de G vers G', et en calculer un le cas échéant. Analyser sa complexité en temps et en espace.

Question 3. Soient G et G' deux graphes orientés et soient $G_1, \ldots, G_n$ et $G'_1, \ldots, G'_m$ les composantes connexes de G et de G' respectivement. Supposons que l'on ait déterminé, pour chaque $1 \leq i \leq n$ et $1 \leq j \leq m$, s'il y a un homomorphisme de G_i vers G'_j. Peut-on s'en servir pour déterminer s'il y a un homomorphisme de G vers G' ?

Une *famille* de graphes est un ensemble de graphes (possiblement infini). Une famille $\mathcal{F}$ est dite *close* si pour tout graphe $G \in \mathcal{F}$, si G a un homomorphisme vers un graphe G', alors on a $G' \in \mathcal{F}$.

Question 4. Un graphe $G = (V, E)$ est dit *cyclique* s'il existe une séquence $v_1, \ldots, v_n$ de sommets de V avec $n > 1$ telle que $v_n = v_1$ et pour tout $1 \leq i < n$, on a $(v_i, v_{i+1}) \in E$. Montrer que la famille des graphes cycliques est close.

Question 5. Proposer un graphe $G_\perp$ tel que si $G_\perp$ appartient à une famille close $\mathcal{F}$ alors $\mathcal{F}$ est la famille de tous les graphes.

Question 6. Expliquer pourquoi une famille close non-vide est nécessairement infinie.

Question 7. Un graphe $G = (V, E)$ est dit *minimal* pour une famille close $\mathcal{F}$ si $G \in \mathcal{F}$ mais pour tout $e \in E$, si l'on note $G_{-e} = (V, E \setminus \{e\})$, alors $G_{-e} \notin \mathcal{F}$. Montrer que si $\mathcal{F}$ n'est pas la famille vide alors $\mathcal{F}$ contient un graphe minimal.

Question 8. Une famille close $\mathcal{F}$ est dite *finiment engendrée* s'il existe une famille finie $\mathcal{F}'$ telle que $\mathcal{F}$ est exactement l'ensemble des graphes G pour lesquels il existe un homomorphisme d'un graphe de $\mathcal{F}'$ vers G. Donner un exemple de famille close finiment engendrée et de famille close qui ne l'est pas.

Suite des questions

Question 9. Montrer que pour toute famille close $\mathcal{F}$ qui n'est pas finiment engendrée, pour tout $n \in \mathbb{N}$, il existe un graphe minimal pour $\mathcal{F}$ ayant au moins $n \in \mathbb{N}$ arêtes.

Corrigé

Question 0. On peut prendre la fonction définie comme suit (ce n'est pas la seule possibilité) :

— $\phi(1) := b$

— $\phi(2) := a$

— $\phi(3) := c$

— $\phi(4) := d$

— $\phi(5) := a$

Il n'y a pas d'homomorphisme dans l'autre sens : tous les sommets de G' ont une arête entrante et une arête sortante donc on ne pourrait les envoyer que vers $2, 3, 4$. Si on envoie a vers un de ces sommets, alors on envoie forcément b vers le suivant, c vers le suivant, et l'arête de a vers c ne peut pas être représentée, c'est donc impossible.

Question 1. Un graphe G est dit *à boucle* s'il possède une boucle, et on voit que :

— Si G est à boucle, alors tout graphe G' a un homomorphisme vers G obtenu en envoyant tous les sommets de G' vers un sommet de G ayant une boucle.

— À l'inverse, si G est à boucle et a un homomorphisme vers un graphe G', alors G' est nécessairement à boucle : il suffit pour cela de considérer l'image de la boucle.

On peut donc caractériser les graphes à boucles en termes d'homomorphisme : un graphe est à boucle si et seulement si tout graphe a un homomorphisme vers lui. En effet, par ce qui précède, si un graphe est à boucle alors tout graphe a un homomorphisme vers lui, et si un graphe n'est pas à boucle alors certains graphes n'ont pas d'homomorphisme vers lui, à savoir les graphes à boucles.

Question 2. On fait simplement un test par force brute :

```
Entrée : matrices carrées d'adjacence M1 et M2 de G et G'
Def Verifie(Tableau T):
  Pour i de 0 à taille(M1) exclu:
    Pour j de 0 à taille(M1) exclu:
      Si M1[i][j] et non M2[T[i]][T[j]]:
        Renvoyer faux
      Fin si
    Fin pour
  Fin pour
  Renvoyer vrai
Fin def

Def Genere(Tableau T, Entier i):
  Si (i == taille(M1)):
    Si Verifie(T):
      Renvoyer T
    Sinon:
      Renvoyer NULL
```

```
    Fin si
  Fin si
  Pour j de 0 à taille(M2) exclu:
    T[i] := j
    T2 := Genere(T, i+1)
    Si T2 != NULL:
      Renvoyer T2
    Fin si
  Fin pour
Fin def

T <- tableau de taille taille(M1)
Renvoyer Genere(T, 0)
```

Si on note n et m les tailles respectives de G et G', la complexité en temps est en $O(m^n \times n^2)$: pour chacune des m^n possibilités on passe un temps total de $O(n^2)$ à recopier le tableau (par valeur) dans les appels récursifs (c'est-à-dire un temps de $O(n)$ pour chacun des n appels récursifs pour cette possibilité), et un temps $O(n^2)$ pour vérifier chaque possibilité. La complexité en espace est de $O(n^2)$ car on a n copies d'un tableau de taille n sur la pile d'appel à tout moment.

Si on remplace le tableau par une variable globale, c'est un peu meilleur en pratique au niveau de la complexité en temps mais la complexité asymptotique reste la même. En revanche, la complexité en mémoire devient alors de $O(n)$ car on ne stocke à tout instant qu'un tableau de taille n et une pile de récursion de hauteur n où chaque appel récursif occupe une quantité constante de mémoire.

[NB : les notions d'occupation mémoire pour les fonctions récursives, piles d'exécution, etc., sont au programme de l'option informatique.]

Question 3. *Indication : montrer que, pour tout graphe connexe G, l'image de G dans un graphe G' par un homomorphisme doit elle aussi être connexe.* Démonstration : pour tous sommets u' et v' de l'image de l'homomorphisme, en prenant u et v deux préimages quelconques de u' et v' respectivement, comme G est connexe il doit y avoir un chemin non-orienté entre u et v, dont l'image par l'homomorphisme est aussi un chemin non-orienté entre u' et v'. Ainsi, toute paire de sommets de l'image est connectée par un chemin non-orienté, donc elle est bel est bien connexe.

Ceci suggère la caractérisation suivante : il y a un homomorphisme de G vers G' si et seulement si, pour chaque composante connexe G_i de G, il existe une composante connexe G'_j de G' telle qu'il y ait un homomorphisme de G_i dans G'_j. En effet, pour l'implication directe, l'existence d'un homomorphisme de G vers G' assure (en considérant ses restrictions aux composantes connexes) que pour toute composante connexe G_i de G il existe un homomorphisme de G_i dans G'. Or, par le résultat de l'indication, l'image de cet homomorphisme est connexe, donc il est inclus dans une composante connexe de G', mettons G'_j, ce qui établit le résultat du sens direct.

Pour le sens réciproque, supposons que pour chaque composante connexe G_i de G il existe un homomorphisme h_i de G_i dans G'_j ; c'est en particulier un homomorphisme de G_i dans G'. Définissons une fonction h sur les sommets de G par $h(v) := h_i(v)$ pour i l'indice tel que $v \in G_i$: par définition d'une composante connexe, cet indice est unique donc h est bien défini. Il est clair que h est bien un homomorphisme, car pour chaque arête (x, y) de G, les sommets x et y sont dans la même composante connexe par définition des composantes connexes ; notons-la G_i. Ainsi, on a $h(x) = h_i(x)$ et $h(y) = h_i(y)$, dont comme h_i est un homomorphisme on sait que $(h_i(x), h_i(y))$ est une arête de G', ainsi $(h(x), h(y))$ est bien une arête de G'. Donc h est bien un homomorphisme.

[NB : la notion de composante connexe d'un graphe orienté est bien au programme, et ne doit pas être confondue avec la notion de composante fortement connexe.]

Question 4. En d'autres termes on veut montrer que si G est cyclique et que G a un homomorphisme vers un graphe G' alors G' est également cyclique. Pour ce faire, soit ϕ un homomorphisme, et considérons la séquence $\phi(v_1), \ldots, \phi(v_n)$ de sommets de G'. La définition d'un homomorphisme assure qu'il s'agit également d'un cycle.

Question 5. Soit $G_\perp = (\{a\}, \emptyset)$. Tout graphe $G = (V, E)$ admet un homomorphisme depuis $G_\perp$ défini par $\phi(a) := v$ pour un $v \in V$ quelconque (rappelons qu'on a interdit qu'un graphe ait un ensemble de sommets vide). Ainsi, si $G_\perp$ appartient à une famille close $\mathcal{F}$, alors tout graphe G appartient également à $\mathcal{F}$, donc c'est la famille de tous les graphes.

Question 6. Une famille close est aussi close par surensembles : on peut rajouter ce qu'on veut à un graphe et on reste dans la famille. Ainsi, une famille est clairement infinie.

Question 7. Comme $\mathcal{F}$ n'est pas la famille vide, soit G un graphe de $\mathcal{F}$. Montrons la contraposée : si $\mathcal{F}$ ne contient pas de graphe minimal alors elle contient un graphe n'ayant aucune arête, ce qui implique (comme à la question 5) qu'elle contient tous les graphes.

Pour ce faire, on procède par récurrence descendante sur les sous-graphes de G (c'est-à-dire les graphes ayant le même ensemble de sommets et un sous-ensemble d'arêtes), suivant la cardinalité de leur nombre d'arêtes, pour montrer que tous ces sous-graphes sont dans $\mathcal{F}$:

— Initialisation : G est dans $\mathcal{F}$ par hypothèse.

— Récurrence : soit $n \in \mathbb{N}^*$ tel que tout sous-graphe de G comportant n arêtes soit dans $\mathcal{F}$. Soit G' un sous-graphe de G avec $n-1$ arêtes : on peut l'écrire comme G''_{-e} pour G'' un sous-graphe de G avec n arêtes et e une arête de ce graphe. Par hypothèse de récurrence, G'' est dans $\mathcal{F}$. Comme $\mathcal{F}$ ne contient pas de graphe minimal, on sait que G''_{-e} est également dans $\mathcal{F}$, ce qui prouve le résultat.

Ainsi, en retirant toutes les arêtes, on aboutit à un graphe sans arêtes dans $\mathcal{F}$, ce qui conclut.

Question 8. La famille comprenant tous les graphes est finiment engendrée : c'est précisément la famille des graphes ayant un homomorphisme depuis G_0 pour G_0 un choix quelconque de graphe sans arêtes.

Montrons que la famille $\mathcal{F}$ des graphes cycliques n'est pas finiment engendrée en procédant par l'absurde. Supposons qu'il existe une famille $\mathcal{F}'$ satisfaisant les conditions données, et soit $n \in \mathbb{N}$ la plus grande taille d'un graphe de $\mathcal{F}'$: comme la famille est finie, ce nombre est fini. Considérons à présent le graphe $C_{n+1} = (\{1, \ldots, n+1\}, \{(i, i+1) \mid 1 \leq i \leq n\} \cup \{(n+1, 1)\})$. Ce graphe est cyclique, et ainsi il doit exister un graphe G' de $\mathcal{F}'$ ayant un homomorphisme h vers C_{n+1}. Maintenant, G' a au plus n sommets, c'est également le cas de l'image de h dans C_{n+1}, plus précisément du sous-graphe induit G'' de C_{n+1} obtenu en conservant les sommets de l'image de h et les arêtes qui joignent ces sommets. Il est alors clair que G' a un homomorphisme vers G''. Mais G'' est un sous-graphe induit de n sommets de C_{n+1}, il est clair qu'il n'a pas de cycle, puisque retirer un sommet quelconque de C_{n+1} et ses deux arêtes incidentes donne un graphe qui n'a pas de cycle. Ainsi, G' a un homomorphisme vers un graphe acyclique : par la contraposée de la question 4, il est donc acyclique. Pourtant, comme G' admet un homomorphisme depuis un graphe de $\mathcal{F}'$, à savoir lui-même, la définition de $\mathcal{F}'$ nous assure alors qu'il est dans $\mathcal{F}$. Comme il est acyclique, c'est une contradiction.

Question 9. On montre la contraposée de cette affirmation : si $\mathcal{F}$ est une famille close et qu'il existe $n \in \mathbb{N}$ tel que tout graphe minimal pour $\mathcal{F}$ ait au plus n arêtes, alors la famille est finiment engendrée.

Pour ce faire, considérons $\mathcal{F}''$ l'ensemble de tous les graphes minimaux pour $\mathcal{F}$. À noter que cet ensemble n'est pas vide, car $\mathcal{F}$ n'est pas la famille de tous les graphes, vu qu'elle n'est pas finiment engendrée. Cet ensemble est infini, mais on peut le rendre fini en quotientant par isomorphisme. Plus

précisément, on considère chaque graphe de $\mathcal{F}''$, on supprime les sommets isolés (sans arête incidente) sauf éventuellement le dernier si c'est le seul sommet, puis on renomme bijectivement les sommets de chaque graphe de $\mathcal{F}''$ pour que l'ensemble de sommets soit un sous-ensemble de $\{1, \ldots, 2n\}$: le nombre de graphes possibles sur cet ensemble de sommets est alors clairement fini, donc l'ensemble $\mathcal{F}'$ de graphes obtenus après renommage est fini. Ces graphes sont toujours des graphes de la famille $\mathcal{F}$, car ils sont le résultat d'un renommage bijectif d'un graphe de $\mathcal{F}$, donc ils ont un homomorphisme (et en fait même un isomorphisme) depuis le graphe dont ils sont issus.

Montrons donc que $\mathcal{F}'$ permet d'engendrer la famille $\mathcal{F}$. Pour une direction de l'inclusion, si un graphe G est tel qu'il y ait un homomorphisme d'un graphe de $\mathcal{F}'$ vers G, alors comme les graphes de $\mathcal{F}'$ sont des graphes de $\mathcal{F}$ et que $\mathcal{F}$ est close, on a bien $G \in \mathcal{F}$. Pour la direction réciproque, soit G un graphe quelconque de $\mathcal{F}$, et démontrons que G admet un homomorphisme depuis un graphe de $\mathcal{F}'$.

Pour ce faire, nous aurons besoin du lemme suivant : pour tout graphe G d'une famille close $\mathcal{F}$ qui n'est pas la famille de tous les graphes, il existe un sous-graphe G' de G qui soit minimal pour $\mathcal{F}$. La preuve est comme à la question 7 : on retire itérativement des arêtes de G tant que c'est possible et que le graphe obtenu est toujours dans $\mathcal{F}$. Comme $\mathcal{F}$ n'est pas la famille de tous les graphes, on obtient forcément à la fin un graphe minimal G' dans $\mathcal{F}$. Ce sous-graphe minimal G' a un homomorphisme vers G vu qu'il est un sous-graphe de G. Ainsi, il y a bien un graphe de $\mathcal{F}'$, à savoir G', qui a un homomorphisme vers G, ce qui montre la deuxième direction de l'implication.

A3 – Comptage de palindromes

On fixe un alphabet Σ. Étant donné un mot $w \in \Sigma^*$, on écrit $|w|$ sa longueur, on écrit ses lettres $w = w_0 \cdots w_{|w|-1}$, et on note $w[i..j]$ le facteur $w_i, \ldots, w_{j-1}$ de w, de sorte que $w[0..|w|] = w$ et que pour tout $0 \leq i < |w|$ le mot $w[i..i]$ est le mot vide ϵ

Le *miroir* d'un mot $w = w_0 \cdots w_{|w|-1}$ de Σ^* est le mot $\overline{w} = w_{|w|-1} \cdots w_0$. Le mot w est un *palindrome* si $w = \overline{w}$. L'objet de ce sujet est de compter le *nombre de palindromes* d'une chaîne $w \in \Sigma^*$ donnée en entrée, c'est-à-dire son nombre de facteurs qui sont des palindromes, formellement $\left|\{(i,j) \mid 0 \leq i \leq j \leq |w|\,, w[i..j] = \overline{w[i..j]}\}\right|$.

Question 0. Quel est le nombre de palindromes de la chaîne $w = abaa$?

Question 1. Écrire le pseudocode d'un algorithme naïf pour compter le nombre de palindromes d'une chaîne donnée en entrée. Décrire sa complexité en temps et en espace.

Question 2. Écrire le pseudocode d'un algorithme moins naïf à base de programmation dynamique pour cette tâche. Décrire sa complexité en temps et en espace.

Question 3. Dans la suite du sujet, on étudiera un algorithme pour compter les palindromes *de longueur impaire*. Décrire une transformation en temps linéaire du mot d'entrée qui permette de se ramener à ce problème.

Question 4. Proposer un algorithme astucieux pour compter les palindromes de longueur impaire avec une meilleure complexité en espace que l'algorithme de la question 2.

Question 5. Pour un mot $w \in \Sigma^*$ et une position $0 \leq i < |w|$, on dit qu'il y a un palindrome centré en i de rayon $\rho'_i \geq 0$ si on a que $i - \rho'_i \geq 0$, que $i + \rho'_i + 1 \leq |w|$, et que $w[(i - \rho'_i)..(i + \rho'_i + 1)]$ est un palindrome. Le *rayon maximal d'un palindrome centré en* i, noté ρ_i, est alors le plus grand rayon d'un palindrome centré en i.

Soit $0 \leq i < |w|$ et $0 < d \leq \rho_i$. Exprimer une inégalité sur ρ_{i+d} qui fasse intervenir ρ_{i-d}. Donner une condition suffisante pour qu'il y ait égalité.

Question 6. Sur la base de cette observation, améliorer l'algorithme de la question 4 pour qu'il fonctionne en temps linéaire.

Corrigé

Question 0. Il y a 5 facteurs de longueur 0 et 4 facteurs de longueur 1 qui sont des palindromes, ainsi qu'un facteur de longueur 2 (aa) et un facteur de longueur 3 (aba) soit onze facteurs en tout qui sont des palindromes.

Question 1. On peut proposer :

```
Entrée: mot w de longueur k
compte := 0
Pour i de 0 à k inclus faire:
  Pour j de i à k inclus faire:
    # Est-ce que w[i..j] est un palindrome ?
    ok = True
    Pour k de i à j exclu faire:
      Si w[i] != w[j-k-1]:
        ok = False
        Sortir de pour
      Fin si
    Fin pour
    Si ok:
      compte += 1
    Fin si
  Fin pour
Fin pour
Renvoyer compte
```

La complexité en temps est en $O(|w|^3)$ et la complexité en espace est constante.

On peut éventuellement gagner un facteur 2 en ne testant que la moitié des positions (arrondies à l'inférieur) dans la boucle `Pour` la plus profonde.

Question 2. *Indication : On cherchera à déterminer quels facteurs sont des palindromes.*

```
Entrée: mot w de longueur k
T[][] = tableau à double entrée de false
Pour i de 0 à k inclus faire:
  # longueur 0
  T[i][i] = true
Fin pour
Pour i de 0 à k-1 inclus faire:
  # longueur 1
  T[i][i+1] = true
Fin pour
Pour d de 2 à k inclus:
  Pour i de 0 à k-d inclus:
    T[i][i+d] := w[i] == w[i+d-1] ET T[i+1][i+d-1]:
  Fin pour
Fin pour
compte := 0
Pour i de 0 à k inclus faire:
  Pour d de 0 à k-i inclus faire:
```

```
    Si T[i][i+d]:
      compte += 1
    Fin si
  Fin pour
Fin pour
Renvoyer compte
```

Cet algorithme est correct car on stocke dans le tableau $T[i][j]$ l'information de si $w[i..j]$ est un palindrome : si $i = j$ ou $i = j + 1$ alors c'est toujours vrai, sinon c'est le cas si les deux caractères extrêmes sont égaux et le facteur qui les exclut (qui est moins long donc déjà traité) est également un palindrome.

La complexité en mémoire est clairement quadratique, ainsi que la complexité en temps.

On peut passer à une complexité en mémoire linéaire si on se rend compte qu'on n'a besoin de stocker que deux tranches du tableau, correspondant à la longueur en cours de calcul et à la longueur précédente.

En pratique on peut aussi vouloir interrompre l'algorithme si on ne trouve plus de palindromes d'une certaine longueur c'est-à-dire que pour un tour de la boucle `Pour` sur d on n'assigne plus aucun Vrai dans le tableau.

Question 3. Intuitivement, on insère un symbole spécial '#' entre chaque paire de lettres (y compris au début et à la fin) pour n'avoir besoin de considérer que des palindromes de longueur impaire (centrés sur une lettre). Il est alors clair (par parité de la position des #) que tous les facteurs palindromiques du mot récrit sont de longueur impaire, et que :

— à chaque palindrome de longueur impaire $2k+1$ du mot original correspondent deux palindromes de longueur impaire dans le mot récrit centrés sur le même caractère, dont la longueur est soit $4k + 1$ soit $4k + 3$ selon qu'on prend les # environnants ou non

— à chaque palindrome de longueur paire non nulle $2k$ du mot original correspondent deux palindromes de longueur impaire dans le mot récrit centrés sur le # entre les deux caractères, et là encore de longueur $4k + 1$ ou $4k + 3$ selon qu'on prend ou non des # environnants.

— pour les palindromes de longueur nulle il n'y a qu'un seul choix possible, à savoir un caractère # isolé

Ainsi pour résoudre notre problème original il suffit de récrire le mot d'entrée w en un mot w' en ajoutant des # en temps linéaire, compter le nombre de palindromes de longueur impaire pour w', puis retrancher $|w| + 1$ (correspondant aux palindromes de longueur nulle qui sont les seuls à ne pas être en double), diviser par deux, puis ajouter $|w| + 1$ (pour retrouver les palindromes de longueur nulle).

Question 4. L'idée est qu'on va considérer chaque centre possible de palindrome de longueur impaire et chercher la longueur du plus long palindrome centré en cette position.

```
compte := 0
Pour i de 0 à k exclu faire:
  compte += 1
  d := 1
  Tant que Vrai faire:
    Si i-d >= 0 ET i+d < k ET w[i-d] == w[i+d]:
      compte += 1
    Sinon:
      Sortir de Tant que
    Fin si
```

```
    d := d + 1
  Fin tant que
Fin pour
```

L'algorithme est toujours en temps quadratique mais l'occupation mémoire est à présent constante et non plus linéaire.

Question 5. Si $\rho_i \geq d$ alors on sait que la position $i-d$ intervient dans un palindrome w' centré en d. On peut alors distinguer trois cas :

— Si $\rho_{i-d} < \rho_i - d$, alors le plus grand palindrome centré en $i-d$ est entièrement dans w', donc on peut trouver un palindrome de la même taille centré en $i+d$. On sait également qu'on ne peut pas en trouver de plus grand, sinon, symétriquement, on aurait un plus grand palindrome centré en $i-d$.

— Si $\rho_{i-d} = \rho_i - d$, alors le même raisonnement s'applique : on peut trouver un palindrome de la même taille centré en $i+d$. Mais, cette fois, on ne peut garantir qu'on ne peut pas en trouver de plus grand.

— Si $\rho_{i-d} > \rho_i - d$, alors on a un palindrome de rayon $\rho_i - d$ centré en $i+d$, et on sait qu'il est maximal, puisque sinon, celui centré en i ne serait pas maximal.

Formellement : on a $\rho_{i+d} \geq \min(\rho_i - d, \rho_{i-d})$, et une condition suffisante pour avoir égalité est que $\rho_{i-d} \neq \rho_i - d$. Notons par ailleurs que cette inégalité est bien entendu aussi correcte lorsque $\rho_i < d$.

Question 6. *Indication possible : éviter des itérations de la boucle interne de la question 4 en se servant de la question 5.*

Grâce à l'inégalité donnée par la question 5, on peut éviter de faire une partie des itérations de la boucle interne de la question 4. Cela donne le pseudo-code suivant :

```
Entrée: mot W de longueur k
T <- tableau de k cases initialisé à 0

i := 0
Pour j de 1 à k-1
  d := j-i
  Si T[i] >= d
    // On évite des itérations de la boucle ci-dessous grâce à la question 5
    T[j] := min(T[i-d], T[i]-d)
  Fin si
  // Boucle comme dans la question 4.
  Tant que j-(T[j]+1) >= 0 ET j+(T[j]+1) < k ET W[j-(T[j]+1)] == W[j+(T[j]+1)]
    T[j] += 1
    i := j
  Fin tant que
Fin pour
```

On trouve facilement à partir de Tle nombre de palindromes en sommant toutes les cases.

La complexité est linéaire parce que, pour une case i, si on passe plus de 0 itérations dans la boucle interne, c'est soit qu'on n'est pas dans le cas d'égalité de la question 5, soit que $T[i] < d$. Par conséquent, à l'entrée de la boucle, on aura $T[j] = T[i] - d$ ou $T[i] < d \wedge T[j] = 0$, c'est-à-dire, dans tous les cas $T[j] + j > T[i] + i$. Donc, pour chaque tour de la boucle interne, $T[i] + i$ augmentera de 1 au moins, et cette quantité est bornée par k.

[C'est l'algorithme de Manacher [Man75] ; voir par exemple [Lee11].]

Références

[Lee11] LeetCode. Longest palindromic substring part II, 2011. https://web.archive.org/web/20140625144935/http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html .

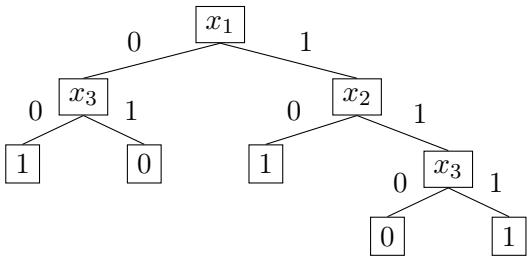
[Man75] Glenn Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *JACM*, 22(3), 1975.

A4 – Arbres de décision

On considère un ensemble de variables propositionnelles $\mathcal{X} = \{x_1, \ldots, x_n\}$ muni de l'ordre total où $x_i < x_j$ si et seulement si $i < j$. Un *arbre de décision* sur $\mathcal{X}$ est un arbre binaire dont les feuilles sont étiquetées par 0 ou par 1, et dont les nœuds internes sont étiquetés par une variable de $\mathcal{X}$ et ont deux enfants appelés *enfant* 0 et *enfant* 1. On impose que si un nœud interne n étiqueté par x_i a un descendant n' qui est un nœud interne étiqueté par x_j alors $x_i < x_j$.

Un arbre de décision T sur $\mathcal{X}$ décrit une fonction booléenne Φ_T qui à toute *valuation* $\nu : \mathcal{X} \to \{0, 1\}$ associe une valeur de vérité calculée comme suit : si T consiste exclusivement d'une feuille étiquetée $b \in \{0, 1\}$ alors la fonction Φ_T s'évalue à b quelle que soit ν. Sinon, on considère le nœud racine n de T et la variable x_i qui l'étiquette, on regarde la valeur $\nu(x_i) \in \{0, 1\}$ que ν donne à x_i, et le résultat de l'évaluation de Φ_T sous ν est celui de l'évaluation de $\Phi_{T'}$ sous ν, où T' est le sous-arbre de T enraciné en l'enfant b de n.

Question 0. On considère l'arbre de décision T_0 suivant et la fonction Φ_{T_0} qu'il définit. Évaluer cette fonction pour la valuation donnant à x_1, x_2, x_3 respectivement les valeurs $0, 1, 0$. Donner un exemple de valuation sous laquelle cette formule s'évalue en 0.



Question 1. On considère la formule de la logique propositionnelle $(x_1 \wedge x_2) \vee \neg(x_1 \wedge \neg x_3)$. Construire un arbre de décision sur les variables $x_1 < x_2 < x_3$ qui représente la même fonction.

Question 2. Quels arbres de décision représentent des tautologies ? des fonctions satisfiables ?

Question 3. Étant donné un arbre de décision représentant une fonction Φ, expliquer comment construire un arbre de décision représentant sa négation $\neg\Phi$.

Question 4. Étant donné un arbre de décision représentant une fonction Φ, donner le pseudocode d'un algorithme qui calcule une représentation de Φ sous la forme d'une formule de la logique propositionnelle. Analyser sa complexité en temps et en espace.

Question 5. Étant donné deux arbres de décision représentant des formules Φ_1 et Φ_2 sur le même ensemble de variables et avec le même ordre, expliquer comment construire un arbre de décision représentant $\Phi_1 \wedge \Phi_2$. Quelle est sa complexité en temps et la taille de l'arbre ainsi obtenu ?

Question 6. Étant donné un arbre de décision et la séquence de variables $x_1, \ldots, x_n$, donner le pseudocode d'un algorithme qui calcule combien de valuations satisfont la fonction booléenne qu'il capture. Quelle est sa complexité en temps ?

Question 7. Peut-on efficacement récrire une formule quelconque de la logique propositionnelle en arbre de décision ?

Suite des questions

On se propose dans les quelques prochaines questions de démontrer formellement l'intuition de la question 7.

Question 7a. Une formule booléenne en *forme normale disjonctive*, également appelée DNF, est une formule qui est une disjonction de conjonctions de littéraux (c'est-à-dire une variable ou sa négation).

Montrer qu'étant donné un arbre de décision on peut efficacement calculer une formule en DNF qui représente la même fonction.

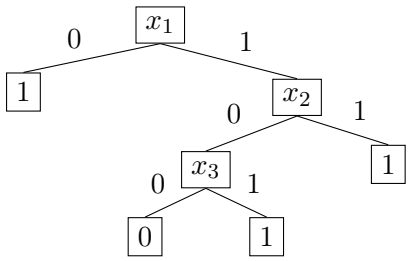
Question 7b. Montrer que, pour tout n, la formule $\bigwedge_{1 \leq i \leq n}(x_i \vee y_i)$ n'admet pas de représentation concise sous la forme d'une DNF.

Question 7c. Conclure.

Corrigé

Question 0. L'évaluation renvoie 1 (pour la feuille tout à gauche). Pour une valuation qui renvoie 0, on peut par exemple envoyer x_1, x_2, x_3 à 1, 1, 0 respectivement.

Question 1.



Question 2. Un arbre de décision représente une tautologie si et seulement si toutes ses feuilles sont étiquetées par 1. En effet, dans ce cas il est clair que la fonction représentée renvoie toujours 1. À l'inverse, par contraposition, s'il y a une feuille étiquetée par 0 alors en suivant un chemin de la racine à cette feuille on obtient une valuation partielle qu'on peut compléter en une valuation totale justifiant que la fonction n'est pas une tautologie. Noter que dans ce cas on pourrait représenter la même fonction de manière plus concise par un arbre n'ayant qu'une feuille.

De la même manière, un arbre de décision représente une fonction satisfiable si et seulement s'il a une feuille étiquetée par 1.

Question 3. Il suffit d'inverser l'étiquette de chaque feuille (c'est-à-dire remplacer 0 par 1 et 1 par 0). Pour montrer que c'est correct, si l'on considère une valuation ν, alors l'évaluation de l'arbre original T et de l'arbre récrit T' sous cette valuation mène dans T' à la feuille correspondant à la feuille atteinte dans T, et ainsi on renvoie 0 (resp., 1) dans T' si et seulement si on renvoie 1 (resp., 0) dans T. Ainsi, T' capture bien $\neg\Phi$.

Question 4. *[On peut discuter comment on représente concrètement la formule propositionnelle ? La représentation comme un arbre est mentionnée dans le programme, sinon on peut aussi la représenter comme une chaîne de caractères avec des parenthèses.]*

Possibilité 1 : récriture vers des décisions. On récrit chaque feuille en vrai ou faux, et chaque nœud interne en une disjonction qui choisit quel sous-arbre utiliser suivant la valeur de la variable testée.

```
Entrée : arbre T

Def Récrire(n):
  Si n est une feuille:
    Renvoyer son étiquette
  Fin si
  x <- Variable testée en n
  n0, n1 <- Enfants de n
  F0 <- Récrire(n0)
  F1 <- Récrire(n1)
  Renvoyer Or(And(x, F0), And(Not(x), F1))
Fin def

Renvoyer Récrire(racine de T)
```

La complexité de cet algorithme est manifestement linéaire en temps et en espace (à condition que les arbres soient passés par référence).

Possibilité 2 : récriture vers une forme normale disjonctive (DNF). On récrit en une disjonction de clauses conjonctives correspondant à l'étiquette des chemins vers des feuilles 1.

```
Entrée : arbre T

Def Récrire(nœud n, liste de littéraux L)
  Si n est une feuille:
    Si elle est étiquetée 1:
      Renvoyer la liste singleton ["AND".join(L)]
    Sinon:
      Renvoyer la liste vide
    Fin si
  Fin si
  x <- Variable testée en n
  n0, n1 <- Enfants de n
  Renvoyer (Récrire(n0, (-x)::L) union Récrire(n1, (+x)::L))
Fin def

Renvoyer "OR".join(Récrire(racine de T, []))
```

La complexité de cet algorithme est quadratique en temps et en espace (la borne supérieure est claire, la borne inférieure s'obtient en considérant un long chemin $x_1, \ldots, x_n$ de taille n suivi d'un arbre binaire complet à n feuilles et $n-1$ nœuds internes sur les variables suivantes : on a un arbre de taille $O(n)$ mais la DNF a taille $O(n^2)$ vu que chaque feuille lui fait répéter le préfixe $x_1, \ldots, x_n$.

Question 5. On définit la transformation récursivement comme suit (un peu comme le produit de deux automates) :

```
Entrée : deux arbres de décision Ta et Tb

infty := |Ta| + |Tb| + 1

Def Conjonction(nœud na de Ta, nœud nb de Tb):
  Si na et nb sont des feuilles:
```

```
    ba <- étiquette de na
    bb <- étiquette de nb
    Renvoyer une feuille étiquetée par (ba ET bb)
  Fin si
  xa <- variable de na (ou infty si na est une feuille)
  xb <- variable de nb (ou infty si nb est une feuille)
  Si xa == xb:
    na0, na1 <- enfants de na
    nb0, nb1 <- enfants de nb
    T0 <- Conjonction(na0, nb0)
    T1 <- Conjonction(na1, nb1)
    Renvoyer un nœud interne étiqueté par x et ayant T0 et T1 comme enfants
  Fin si
  // na et nb ont des variables différentes
  Si xa < xb:
    na0, na1 <- enfants de na
    T0 <- Conjonction(na0, nb)
    T1 <- Conjonction(na1, nb)
    Renvoyer un nœud interne étiqueté par xa et ayant T0 et T1 comme enfants
  Sinon:
    nb0, nb1 <- enfants de nb
    T0 <- Conjonction(na, nb0)
    T1 <- Conjonction(na, nb1)
    Renvoyer un nœud interne étiqueté par xb et ayant T0 et T1 comme enfants
  Fin si
Fin def

Renvoyer Conjonction(racine de Ta, racine de Tb)
```

Justification de la correction : pour toute valuation, le chemin de la racine du nouvel arbre à une feuille franchit une succession de nœuds internes correspondant à des paires (na, nb) dans la construction de l'algorithme, et la suite des premières et secondes valeurs donnent les chemins compatibles avec cette valuation dans Ta et Tb. Or, la feuille à laquelle on arrive dans l'arbre récrit porte pour étiquette le ET des étiquettes des deux feuilles auxquelles on arrive dans les arbres originaux, donc on obtient bien le bon résultat.

Pour prouver la terminaison de cet algorithme, il faut noter l'invariant que Conjonction est toujours appelé avec une paire de nœuds qui sont descendants de chacun des deux nœuds précédents, avec au moins une des deux relations qui est stricte. Ainsi, l'algorithme termine, et le nombre d'appels est au plus en taille de Ta fois taille de Tb. Cette borne inférieure ne peut pas être améliorée en général : si on considère un arbre sur des variables $x_1, \ldots, x_k$ et un autre sur des variables $x_{k+1}, \ldots, x_{k'}$, l'effet de l'algorithme sera de créer une copie du deuxième arbre pour remplacer chacune des feuilles étiquetées 1 du premier arbre, or le nombre de feuilles d'un arbre binaire complet (c'est-à-dire où chaque nœud interne a deux enfants) est linéaire en sa taille.

Question 6. *Réponse naïve possible en premier lieu : on peut tester toutes les valuations. Dans ce cas, on s'attend à ce que le candidat remarque que la complexité est mauvaise.*

Il n'est pas compliqué de compter efficacement les valuations acceptantes, mais il faut faire attention aux variables manquantes dans l'arbre.

```
Entrée : un arbre de décision T, le nombre k de variables
  (on identifie les variables avec x1... xk)
```

```
Sortie : nombre de valuations satisfaisantes de T

Def Compte(nœud n, indice de la variable précédente xprev):
  Si n est une feuille étiquetée 0:
    Renvoyer 0
  Fin si
  Si n est une feuille étiquetée 1:
    Renvoyer 2^{k-iprev)
  Fin si
  // n est un nœud interne
  i <- indice de la variable testée en n
  n0, n1 <- enfants de n
  factor <- 2^{i-iprev-1} // valuations de variables précédemment sautées
  v0 <- Compte(n0, i)
  v1 <- Compte(n1, i)
  Renvoyer factor*(v0 + v1)
Fin def

Renvoyer Compte(racine de T, 0)
```

La complexité de l'algorithme est manifestement linéaire (pour des opérations arithmétiques en temps constant ; elle est polynomiale sinon).

Question 7. On ne s'attend pas à ce qu'une récriture efficace soit possible, parce qu'étant donné un arbre de décision, on peut facilement déterminer si la fonction correspondante est satisfiable (par la question 2, ou la question 6) ; or on sait qu'il est "difficile" de déterminer si une formule booléenne est satisfiable. (Cette notion est au programme de l'option informatique, même si la notion de NP-complétude n'est bien sûr pas exigible.)

Question 7a. C'est la possibilité 2 de la question 4.

Question 7b. On montre qu'une représentation de cette fonction sous forme de DNF doit avoir au moins 2^n clauses. Pour une valuation qui satisfait la fonction, on appelle *clause témoin* une clause qui est rendue vraie par la valuation : chaque valuation satisfaisante a au moins une clause témoin.

À présent, considérons les valuations satisfaisantes $\nu : \{x_1, \ldots, x_n, y_1, \ldots, y_n\} \to \{0, 1\}$ de la formule qui sont minimales, c'est-à-dire celles où, pour chaque $1 \leq i \leq n$, précisément l'un de x_i, y_i est assigné à vrai. Il y a clairement 2^n telles valuations.

À présent, supposons par l'absurde que la DNF a strictement moins de 2^n clauses. Par le principe des tiroirs, ceci implique que deux valuations minimales ν et ν' ont la même clause témoin. Soit z_i une variable telle que $\nu(z_i) \neq \nu'(z_i)$, ce qui existe forcément car les valuations minimales sont différentes. Sans perte de généralité, on va supposer que ν rend x_i vrai (et donc y_i faux par minimalité) et que ν' rend x_i faux (et donc y_i vrai car c'est une valuation satisfaisante). La clause comporte forcément l'un des littéraux x_i, y_i, puisque sinon elle est satisfaite par une valuation non satisfaisante qui rend x_i et y_i tous deux faux. Mais ce littéral ne peut être ni x_i (sinon ν' ne satisfait pas la clause), ni y_i (sinon ν ne la satisfait pas), on a donc une contradiction.

On a donc démontré par l'absurde que toute représentation de la fonction comme DNF doit avoir au moins 2^n clauses.

Question 7c. Posons $n \in \mathbb{N}$ et considérons la fonction définie à la question 7b sur $2n$ variables. Si cette fonction a une représentation sous la forme d'un arbre de décision de taille k, alors on sait par la

question 7a qu'elle peut être représentée comme une DNF de taille $O(k^2)$, or on sait par la question 7b que toute DNF pour cette fonction a une taille $\geq 2^k$. Ainsi on en déduit qu'il existe des familles de fonctions (à savoir celles de la question 7b) pour lesquelles toute représentation sous la forme d'un arbre de décision (indépendamment de l'ordre) est de taille super-polynomiale.

A5 – Ensembles inévitables

On fixe un alphabet fini $\Sigma = \{a, b\}$. Pour $w \in \Sigma^*$, on écrit $w = w_1 \cdots w_n$ où $n := |w|$ est la *longueur* de w. On dit qu'un mot $w \in \Sigma^*$ *évite* un mot $s \in \Sigma^*$ si s n'apparaît *pas* comme facteur de w. On dit que w *évite* un ensemble de mots $S \subseteq \Sigma^*$ s'il évite chaque mot de S.

Question 0. Donner un mot de longueur au moins 12 qui évite l'ensemble $S = \{aaaa, aaab, aba, baaa, bab, bbbb\}$.

Question 1. Donner le pseudocode d'un algorithme simple qui détermine, étant donné un mot $w \in \Sigma^*$ et un ensemble fini $S \subseteq \Sigma^*$, si w évite S. Analyser sa complexité en temps et en espace quand tous les mots de S font la même longueur.

On dit qu'un ensemble $S \subseteq \Sigma^*$ est *inévitable* s'il n'existe qu'un nombre fini de mots $w \in \Sigma^*$ qui évitent S. Sinon, il est dit *évitable*.

Question 2. L'ensemble de la question 0 est-il inévitable ? L'ensemble $\{aaa, abb, baa, abab\}$ est-il inévitable ?

Question 3. Montrer que l'ensemble $\{aaa, bab, baab, bbb\}$ est inévitable.

Question 4. Montrer le lemme suivant : pour tout alphabet fini Σ fixé et $k \in \mathbb{N}$, il existe $n \in \mathbb{N}$ tel que pour tout mot $w \in \Sigma^*$ tel que $|w| > n$, il existe deux entiers $p < q$ tels que pour tout $0 \leq l < k$, on ait $w_{p+l} = w_{q+l}$.

Question 5. Utiliser cette observation pour proposer un algorithme (pas nécessairement efficace) qui, étant donné un ensemble fini $S \subseteq \Sigma^*$, détermine si S est inévitable. Décrire sa complexité en temps et en espace.

Question 6. On dit qu'une expression rationnelle e est *inévitable* si l'ensemble (généralement infini) des mots du langage de e est inévitable. Donner un exemple d'expression rationnelle inévitable.

Question 7. L'expression rationnelle e est *bornée* s'il existe $n \in \mathbb{N}$ tel que tout mot satisfaisant e soit de longueur $\leq n$. Si e est bornée, comment peut-on déterminer si elle est inévitable ?

Question 8. Expliquer comment déterminer si une expression rationnelle e quelconque est inévitable.

Question 9. Pour e une expression rationnelle, on dit qu'un ensemble $S \subseteq \Sigma^*$ est *inévitable pour* e s'il existe un ensemble infini de mots du langage de e qui évitent S. Étant donné une expression rationnelle e et un ensemble fini S, expliquer comment déterminer si S est inévitable pour e.

Question 10. Montrer que, pour tout ensemble inévitable $S \subseteq \Sigma^*$, il existe un sous-ensemble $S' \subseteq S$ fini tel que S' soit inévitable.

Question 11. Étant donné une expression rationnelle e inévitable, expliquer comment calculer un sous-ensemble inévitable fini du langage de e.

Corrigé

Question 0. On peut prendre par exemple *aabbaabbaabb*.

Question 1. La principale difficulté est d'être rigoureux avec les indices.

```
Entrée:
  - mot w représenté comme un tableau de longueur n,
  - ensemble S représenté comme un tableau de longueur m de tableaux de
  longueurs nS[i] pour chaque i

Pour j de 0 à m exclu faire:
  Pour i de 0 à n-nS[j] exclu faire:
    ok := True
    Pour d de 0 à nS[j] exclu faire:
      Si w[i+d] != S[i][d]:
        ok := False
        Sortir de Pour
      Fin si
    Fin pour
    Si ok:
      // on a trouvé une occurrence de S[j] dans w
      Renvoyer faux
    Fin si
  Fin pour
Fin pour
Renvoyer vrai
```

Si les m mots de S font la même longueur l, la complexité en temps est en $O(n \times m \times l)$ et la complexité en espace est constante.

[On peut faire mieux en temps en utilisant l'algorithme de Knuth-Morris-Pratt (prétraitement en $O(m \times l)$ puis vérification en $O(n \times m)$) ou l'algorithme d'Aho-Corasick (prétraitement en $O(m \times l)$ puis vérification en $O(n)$), mais cela n'est pas demandé.]

Question 2. L'ensemble de la question 0 est évitable puisque, pour tout $i \in \mathbb{N}$, le mot $(aabb)^i$ l'évite.
L'ensemble $\{aaa, abb, baa, abab\}$ est évitable aussi : prendre b^i pour tout $i \in \mathbb{N}$.

Question 3. Procédons par l'absurde. Comme l'ensemble contient *aaa* et *bbb*, un mot suffisamment long évitant l'ensemble doit nécessairement comporter un b suivi d'un a, c'est-à-dire un facteur ba ; et si le mot est suffisamment long ce facteur est suivi d'autres caractères. Mais alors ce facteur ne peut être suivi de b (sinon on a un facteur *bab*), donc il est suivi de a, c'est-à-dire un facteur *baa*. Mais ce facteur ne peut être suivi de a (ce qui donnerait *aaa*), ni de b (ce qui donnerait *baab*), contradiction. Ainsi l'ensemble est-il bien inévitable.

Question 4. C'est une application immédiate du principe des tiroirs. On prend $n = |\Sigma|^k + k$. Ce mot a strictement plus de $|\Sigma|^k$ positions où commence un facteur de taille k, donc par application du principe des tiroirs il y a deux positions différentes où commence un facteur identique, ce qui est le résultat voulu.

Question 5. Soit k la longueur maximale d'un mot de S. Montrons tout d'abord que S n'est pas inévitable si et seulement s'il existe un mot évitant S avec deux occurrences distinctes d'un même facteur de longueur k. *(C'est une indication possible.)*

Dans le sens direct, si S est évitable alors il existe une infinité de mots évitant S, donc il existe des mots arbitrairement longs évitant S, en particulier des mots de longueur strictement supérieure au n de la question précédente. Ainsi, si on prend un tel mot, il a deux occurrences distinctes du même facteur de taille k.

Réciproquement, s'il existe un mot évitant S avec deux occurrences du même facteur de longueur k, alors on enlève un suffixe de ce mot pour que le mot se termine à la fin de la deuxième occurrence du facteur : le mot w_0 résultant évite toujours S. Soit $d > 0$ le décalage entre les deux occurrences. On peut ensuite construire des mots de plus en plus longs évitant S en ajoutant itérativement une copie de la lettre à d positions en partant de la fin : les mots résultants évitent tous S car ils ont un ensemble de facteurs de longueur k qui est le même que celui de w_0, qui évitait S, or la propriété d'éviter S est entièrement déterminée par les facteurs de longueur k par définition de k. Ainsi, S est évitable.

La caractérisation que nous avons démontrée implique que S est inévitable si et seulement si tous les mots de longueur $> n$ ont un facteur dans S, pour n la valeur de la question précédente. En effet, le sens réciproque est trivial (car il y a alors un nombre fini de mots évitant S), et pour le sens direct on vient de montrer que si S est inévitable alors tout mot ayant deux occurrences distinctes d'un même facteur de longueur k n'évite pas S, donc en particulier tous les mots de longueur $> n$.

Ceci suggère l'algorithme suivant : on énumère tous les mots de longueur $\leq n$, et on teste s'ils ont un facteur dans S. La complexité de cet algorithme est en $\left(\sum_{w \in S} |w|\right) \times |\Sigma|^k + k$ où k est la longueur maximale d'un mot de S, c'est-à-dire exponentiel en S. La complexité en espace demande de stocker le mot en cours, c'est-à-dire $|\Sigma|^k + k$.

En pratique il vaudrait mieux construire incrémentalement un mot par backtracking, en abandonnant si on contient un facteur dans S, et en réussissant si on a répété un facteur de longueur k déjà atteint.

Un bien meilleur algorithme (en $O(|\Sigma|^{k+1})$ pour k la longueur maximale d'un mot de S) est de considérer chaque mot de Σ^k et de construire un graphe orienté sur cet ensemble (avec $|\Sigma|^{k+1}$ arêtes) : on relie w à w' si et seulement s'il existe $a \in \Sigma$ tel que, si on écrit $w = a'w''$, on a $w' = w''a$. On élimine ensuite tous les sommets qui contiennent un mot de S. Il est clair que si ce graphe a un cycle alors on construit (comme dans la caractérisation précédente) un mot infini évitant S ; à l'inverse si on peut construire un tel mot alors comme à la caractérisation précédente un mot suffisamment long témoigne de l'existence d'un cycle dans ce graphe.

Question 6. Tout simplement $(aaa|bab|baab|bbb)^*$ d'après la question 3.

Question 7. Si une expression rationnelle est bornée, alors son langage est fini, et on peut simplement appliquer la question 5.

Question 8. On construit à partir de e l'expression rationnelle $\Sigma^* e \Sigma^*$ des mots qui ont un facteur dans e, i.e., ceux qui n'évitent pas e. On souhaite savoir si le complémentaire de ce langage est fini. Pour ce faire, on peut transformer e en un automate fini qui reconnaît le même langage avec les outils du cours, et complémenter cet automate (la clôture par complémentaire est également au programme). Il s'agit alors de déterminer si cet automate reconnaît un langage fini : après suppression des états inutiles, c'est le cas si et seulement s'il n'a pas de cycle.

On peut tester algorithmiquement tout cela, mais l'étape coûteuse est la complémentation : l'automate obtenu pour $\Sigma^* e \Sigma^*$ n'est généralement pas déterministe, et la clôture d'un automate implique de le complémenter d'abord, ce qui prend un temps exponentiel en général.

(Noter que cette construction généralise celle de la question 5.)

Question 9. On construit aisément l'expression rationnelle e_S des mots qui ont un facteur dans S, i.e., n'évitent pas S. On souhaite savoir si l'intersection de e avec le complémentaire de ce langage est finie. Pour ce faire, comme à la question 8, on construit à partit de e_S un automate A_S qu'on complémente en un automate A'_S (en le déterminisant au préalable), et on construit à partir de e un automate A. On teste alors si les langages de A et A'_S ont une intersection infinie en construisant un automate pour l'intersection (clôture par intersection au programme) et en testant comme à la question précédente.

Question 10. Soit $S \subseteq \Sigma^*$ inévitable. Soit n la longueur maximale d'un mot qui évite S. Ainsi, tout mot de longueur $n+1$ a un facteur dans S : soit S' l'ensemble de ces facteurs. C'est un ensemble fini, et il est inévitable puisque par définition tout mot de longueur $\geq n+1$ a un préfixe de longueur $n+1$ qui a un facteur dans S.

Question 11. En suivant la question précédente : à partir de l'automate construit à la question 8, on construit l'ensemble fini des mots dans le complémentaire L' du langage $\Sigma^* e \Sigma^*$ directement à partir de l'automate, on regarde leur longueur maximale n, puis on prend tous les mots de longueur $n+1$ et on leur trouve un facteur qui satisfait e. (En pratique il suffit de se limiter à un ensemble clos par préfixe des mots de longueur $n+1$, donc on peut notamment s'arrêter à tous les mots qui n'ont pas de continuation dans L', qu'on peut voir sur l'automate.)

J1 – Valeurs plus faibles les plus proches et arbres cartésiens

On fixe u un tableau de nombres entiers relatifs de taille n. On définit les séquences d'ensembles d'entiers $(A_i)_{0\leq i<n}$ et $(B_i)_{0\leq i<n}$ et les tableaux d'entiers a et b de taille n par :

$$A_i = \{j \mid 0 \leq j < i \wedge u[j] \leq u[i]\} \qquad B_i = \{j \mid i < j < n \wedge u[j] < u[i]\}$$

$$a[i] = \begin{cases} \max A_i & \text{si } A_i \neq \emptyset \\ -1 & \text{sinon} \end{cases} \qquad b[i] = \begin{cases} \min B_i & \text{si } B_i \neq \emptyset \\ n & \text{sinon} \end{cases}$$

Question 0. Supposons, pour cette question uniquement :

$$u = [0, 8, 4, 12, 2, 10, 6, 14, 0, -1]$$

Calculer alors le contenu du tableau a.

Question 1. Donner le pseudo-code et la complexité en temps et en mémoire d'un algorithme naïf permettant de calculer a à partir de u.

Question 2. Donner le pseudo-code et la complexité en temps et en mémoire d'un algorithme plus efficace pour effectuer ce calcul. Adapter cet algorithme pour calculer b.

Un arbre binaire étiqueté par des entiers est soit une feuille (notée $\perp$), soit de la forme $N(x, T_1, T_2)$, où x est un entier et T_1 et T_2 sont des arbres binaires étiquetés pat des entiers. Si T est un tel arbre, on note $I(T)$ la séquence d'entiers définie inductivement comme suit, où $\cdot$ dénote la concaténation de séquences :

$$I(\perp) = [] \qquad I(N(x, T_1, T_2)) = I(T_1) \cdot [x] \cdot I(T_2)$$

Un *arbre cartésien* associé à u est un arbre binaire T_c étiqueté par des entiers tel que $I(T_c) = u$ et T_c est un tas où la racine est étiquetée par un élément de valeur minimale du tableau.

Question 3. Calculer l'ensemble des arbres cartésiens associés à l'exemple de la question 0.

Donner une condition nécessaire et suffisante sur u pour qu'il existe exactement un arbre cartésien associé à u.

Question 4. On suppose dans cette question que tous les éléments de u sont distincts. Établir un lien entre la structure de T_c et les tableaux a et b. En déduire un algorithme efficace pour calculer T_c à partir de u. Quelle est sa complexité en temps et en mémoire ?

Corrigé

Question 0. $a = [-1, 0, 0, 2, 0, 4, 4, 6, 0, -1]$

Question 1.

```
Pour i = 0 à n-1
  j <- i-1
  Tant que j >= 0 et u[j] > u[i]
    j <- j-1
  a[i] <- j
```

Cet algorithme a une complexité en temps de $O(n^2)$ et $O(1)$ en espace.

Question 2. Pour une position i donnée, on considère l'ensemble S des positions j qui ne sont pas "cachées" par des positions intermédiaires. C'est à dire qu'une position j est dans S s'il n'existe pas de position $k \in [\![j+1, i-1]\!]$ telle que $u[k] < u[j]$. On peut utiliser une pile `s` pour maintenir cet ensemble de positions :

```
s <- [-1]
Pour i = 0 à n-1
  Tant que s.top() >= 0 Et u[s.top()] > u[i]
    s.pop()
  a[i] <- s.top()
  s.push(i)
```

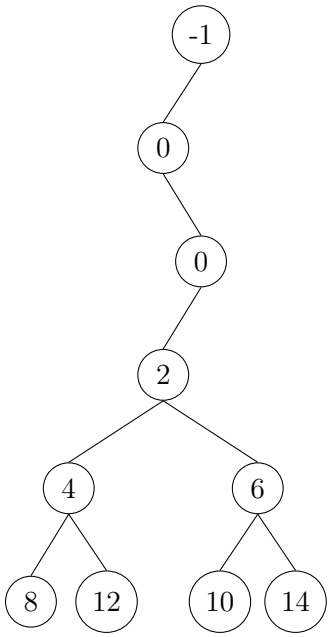
Sa complexité en temps est $O(n)$. En effet, le nombre d'itérations de la boucle extérieure est n et le nombre d'itérations de la boucle intérieure est bornée par le nombre d'appels à `push`, qui est lui-même borné par n.

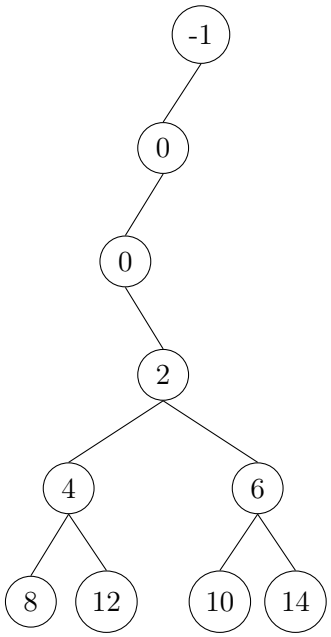
Sa complexité en mémoire est $O(n)$. En effet, la taille de la pile est bornée par $n + 1$.

Pour calculer b, on utilise un algorithme très similaire :

```
s <- [n]
Pour i = n-1 à 0
  Tant que s.top() < n Et u[s.top()] >= u[i]
    s.pop()
  b[i] <- s.top()
  s.push(i)
```

Question 3. Les deux arbres cartésiens correspondant à l'exemple de la question 0 sont :





Pour la seconde partie de la question, on commence par prouver le lemme suivant : il existe toujours au moins un arbre cartésien. On procède par récurrence forte sur n, la longueur de u. En effet, si $n = 0$, alors une feuille est un arbre cartésien. Sinon, on place à la racine une occurrence quelconque du minimum de u, on utilise comme sous-arbre gauche un arbre cartésien correspondant au préfixe de u s'arrêtant juste avant cette occurrence, et comme sous-arbre droit un arbre cartésien correspondant au suffixe de u commençant juste après cette occurrence. Ces deux arbres cartésiens existent selon l'hypothèse de récurrence.

Maintenant, on peut prouver le résultat final : l'arbre cartésien est unique ssi toute paire de valeurs identiques dans u est séparée par une valeur inférieure (propriété (A)). On procède encore par récurrence forte sur n. Le cas $n = 0$ est tout aussi élémentaire que précédemment. Sinon :

- Si la propriété (A) est vraie, alors le minimum n'apparaît qu'une seule fois, et il est nécessairement à la racine (par la propriété de tas). Il n'y a donc qu'un seul choix possible pour partitionner u. Par ailleurs, le préfixe et le suffixe concernés vérifient aussi la propriété (A), et donc, selon l'hypothèse de récurrence, il n'y a pas de choix possible pour les sous-arbres. L'arbre cartésien est donc bien unique.
- Si l'arbre cartésien est unique, alors :
  - Le minimum de u ne peut apparaître qu'une seule fois. En effet, dans le cas contraire, on pourrait partitionner u de plusieurs manières différentes. Pour chacune de ces partitions, on peut utiliser le lemme prouvé plus haut et obtenir des sous-arbres droit et gauche et donc des arbres cartésiens différents.
  - Par hypothèse de récurrence, le préfixe et le suffixe correspondant aux sous-arbres droit et gauche vérifient la propriété (A).
  - Mais alors, la propriété (A) est vérifiée pour u tout entier, puisque le préfixe et le suffixe sont séparés par le minimum de u.

Question 4. Le propriété demandée est la suivante. Soit $i \in [\![0, n-1]\!]$:

- Si $u[i]$ est à la racine de l'arbre, alors $a[i] = -1$ et $b[i] = n$.
- Si $u[i]$ étiquette un fils droit dans l'arbre, alors $a[i] \neq -1$, son parent est étiqueté par $u[a[i]]$, et, si $b[i] \neq n$, alors $u[a[i]] > u[b[i]]$.
- Symétriquement, si $u[i]$ étiquette un fils gauche dans l'arbre, alors $b[i] \neq n$, son parent est étiqueté par $u[b[i]]$, et, si $a[i] \neq -1$, alors $u[a[i]] < u[b[i]]$.

Pour prouver cette propriété, on peut tout d'abord traiter facilement le cas de la racine : il est clair que $u[i]$ est le minimum de u ssi $a[i] = -1$ et $b[i] = n$. Donc $u[i]$ étiquette la racine ssi $a[i] = -1$ et $b[i] = n$.

Supposons que $u[i]$ étiquette un fils droit. Alors le parent est étiqueté par une valeur plus petite, et toutes les valeurs situées entre la position i et la position correspondant au parent sont plus grandes (puisque situées dans l'arbre sous le nœud étiqueté par $u[i]$). Selon la définition de a, on a $a[i] \neq -1$ et ce parent est étiqueté par $u[a[i]]$.

Par ailleurs, on remarque que si $b[i] \neq n$, le nœud étiqueté par $u[b[i]]$ est nécessairement un ancêtre du nœud étiqueté par $u[i]$. En effet, le plus proche ancêtre commun de ces deux nœuds correspond nécessairement à une position située entre i et $b[i]$. Et, par la propriété de tas, ce plus proche ancêtre commun est nécessairement étiqueté par une valeur plus faible que $u[i]$. Par définition de $b[i]$, il ne peut s'agir que du nœud étiqueté par $u[b[i]]$. En conclusion, le plus proche ancêtre commun des nœuds étiquetés par $u[i]$ et $u[b[i]]$ est ce dernier, donc il est l'ancêtre du premier.

Finalement, si $u[i]$ étiquette un fils droit, alors $a[i] \neq -1$ et son parent est étiqueté par $u[a[i]]$. Et, si $b[i] \neq n$, alors $u[b[i]]$ étiquette un ancêtre, qui ne peut pas être le parent. Donc $u[b[i]]$ étiquette un ancêtre du nœud étiqueté par $u[a[i]]$. Donc $u[b[i]] < u[a[i]]$.

La propriété symétrique s'obtient avec le même raisonnement.

Après avoir calculé a et b en temps linéaire, ces trois propriétés permettent de calculer le parent de chaque nœud de l'arbre. On peut alors inverser cette fonction "parent" à l'aide, par exemple de deux tableaux, pour déterminer les deux fils de chaque nœud et donc reconstituer l'arbre sous une représentation plus habituelle. Ce calcul se fait en temps $O(n)$ et en espace $O(n)$.

J2 – Preuve de programmes en logique de Hoare

On considère un petit language de programmation imaginaire comprenant des expressions arithmétiques et booléennes, les structures de contrôle "`si ... alors ... sinon ...`" et "`tant que ... faire ...`", des variables entières et des tableaux d'entiers.

Écrire dans un tableau en dehors des bornes déclenche une erreur à l'exécution, mais, pour simplifier, on suppose qu'il est possible de *lire* un tableau en dehors de ses bornes sans provoquer d'erreur. Le résultat ainsi obtenu est l'entier 0.

Soit le programme P_0 suivant, écrit dans ce petit langage de programmation :

```
i := 0
tant que i < longueur(t) faire
  si i == 0 ou t[i] >= t[i-1] alors
    i := i+1
  sinon
    tmp := t[i];
    t[i] := t[i-1];
    t[i-1] := tmp;
    i := i-1
```

Question 0. Que fait ce programme ? Donner une justification *informelle*. Quelle est sa complexité en temps et en espace ?

Dans ce sujet, on se propose de s'approcher d'une preuve *très rigoureuse* de cette spécification. Pour cela, on introduit la notion de *triplet de Hoare* : il s'agit de la donnée d'un fragment de programme C et de deux assertions P et Q qui dépendent du contenu des variables du programme. Un tel triplet de Hoare est noté $\{P\}C\{Q\}$. On dit que P est la *précondition* du triplet et que Q est sa *postcondition*.

Un triplet $\{P\}C\{Q\}$ est dit *valide* lorsque, quel que soit le contenu initial des variables du programme, si P est vrai avant l'exécution de C, alors celle-ci ne provoque pas d'erreur, et, si elle termine, alors Q est vraie après l'exécution de C. On note parfois "$\{P\}C\{Q\}$" pour signifier "$\{P\}C\{Q\}$ est valide".

Question 1. Trouver une assertion Q (la plus forte possible) telle que le triplet :

$$\{\mathtt{y} = 3\} \quad \mathtt{x := 1; \ y := y + 1} \quad \{Q\}$$

soit valide.

Afin de prouver la validité de triplets de Hoare, on dispose de règles de raisonnement. Pour alléger les notations, on énonce chaque règle sous la forme d'une règle d'inférence : ses hypothèses sont écrites au dessus d'une ligne horizontale et sa conclusion en dessous. On donne ici certaines de ces règles[1] :

$$\frac{P \Rightarrow P' \qquad Q' \Rightarrow Q \qquad \{P'\}C\{Q'\}}{\{P\}C\{Q\}} \qquad \frac{\{P\}C_1\{Q\} \qquad \{Q\}C_2\{R\}}{\{P\}C_1\,;C_2\{R\}} \qquad \frac{}{\{P[x/e]\}x \ \mathtt{:=}\ e\{P\}}$$

$$\frac{\{I \wedge b\}C\{I\}}{\{I\}\mathtt{tant\ que}\ b\ \mathtt{faire}\ C\{I \wedge \neg b\}}$$

Note : $P[x/e]$ désigne l'assertion P dans laquelle on a remplacé toutes les occurrences de x par e.

1. Il est possible d'en donner une preuve rigoureuse, mais on se contentera ici de les admettre.

Question 2. À l'aide de ces règles, prouver la validité...

(a) ... du triplet de la question 1.

(b) ... du triplet suivant :

$\{\texttt{n} \geq 0\}$
```
i := 0;
f := 1;
tant que i < n faire
  i := i + 1;
  f := i * f
```
$\{\texttt{f} = \texttt{n}!\}$

Question 3. Proposer, sans la prouver, une règle pour un fragment de programme de la forme ...

(a) ... `si` b `alors` C_1 `sinon` C_2

(b) ... `t[`e_1`] :=` e_2 pour écrire dans un tableau (on pourra s'inspirer de la règle pour l'affectation ci-dessus).

Question 4. Proposer et prouver des triplets de Hoare permettant de prouver que le programme P_0...

(a) ... s'exécute sans erreur.

(b) ... ne change pas le multi-ensemble des éléments du tableau `t`.

(c) ... a la spécification proposée à la question 0 (on admettra la terminaison de l'algorithme).

Corrigé

Question 0. Ce programme permet de trier le tableau `t`. Il s'agit d'une implémentation du "gnome sort", qui est en fait une sorte de tri par insertion. En effet, l'unique boucle du programme permet à la fois d'itérer sur les éléments à insérer et d'effectuer l'insertion proprement dite. Plus précisément, l'algorithme recherche, de gauche à droite, un élément mal positionné (c'est-à-dire plus petit que son prédécesseur). Lorsqu'un tel élément est trouvé, il est échangé avec ses prédécesseurs jusqu'à ce qu'il trouve sa place.

La complexité en temps du programme est $O(n^2)$

Question 1. L'assertion $Q \Leftrightarrow \mathtt{y} = 4 \wedge \mathtt{x} = 1$ est la plus forte qui convient ici.

Question 2.

(a) On a la dérivation suivante :

$$\dfrac{\dfrac{}{\{\mathtt{y}+1=4 \wedge 1=1\}\mathtt{x := 1}\{\mathtt{y}+1=4 \wedge \mathtt{x}=1\}} \quad \dfrac{}{\{\mathtt{y}+1=4 \wedge \mathtt{x}=1\}\mathtt{y := y}+1\{\mathtt{y}=4 \wedge \mathtt{x}=1\}}}{\{\mathtt{y}+1=4 \wedge 1=1\}\mathtt{x := 1; y := y}+1\{\mathtt{y}=4 \wedge \mathtt{x}=1\}}$$

Il suffit alors d'appliquer une fois la règle de conséquence, pour obtenir le triplet demandé, puisque $\mathtt{y} = 3 \Rightarrow \mathtt{y} + 1 = 4 \wedge 1 = 1$.

(b) *On ne demande pas le détail de l'application de chaque règle et de l'énoncé de chaque assertion intermédiaire. On s'assure surtout du bon choix de l'invariant et on vérifie que le candidat comprend quelles règles sont utilisées à chaque point de la démonstration.*

Avant de commencer à prouver des triplets de Hoare, il faut chercher l'invariant de boucle qui permettra d'appliquer la règle de la construction `tant que`. Cet invariant doit contenir deux informations :

- $0 \leq \mathtt{i} \leq n$, ce qui permettra de donner du sens à l'expression `i!`, et de garantir que $i = n$ à la fin de l'exécution de la boucle.
- $\mathtt{f} = \mathtt{i}!$, ce qui permettra d'obtenir le résultat final.

On choisit donc $I \Leftrightarrow 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{f} = \mathtt{i}!$ comme invariant.

On peut maintenant faire la preuve proprement dite. On ne détaille pas ici l'application de chaque règle, on donne simplement les préconditions et postcondition à chaque position dans le programme :

$\{\texttt{n} \geq 0\}$
```
i := 0;
```
$\{\texttt{n} \geq 0 \wedge \texttt{i} = 0\}$
```
f := 1;
```
$\{\texttt{n} \geq 0 \wedge \texttt{i} = 0 \wedge \texttt{f} = 1\}$
$\Downarrow$
$\{0 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{f} = \texttt{i}!\}$
```
tant que i < n faire
```
$\{0 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{f} = \texttt{i}! \wedge \texttt{i} < \texttt{n}\}$
$\Downarrow$
$\{0 \leq \texttt{i} + 1 \leq \texttt{n} \wedge (\texttt{i} + 1) * \texttt{f} = (\texttt{i} + 1)!\}$
```
    i := i + 1;
```
$\{0 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{i} * \texttt{f} = \texttt{i}!\}$
```
    f := i * f
```
$\{0 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{f} = \texttt{i}!\}$
$\{0 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{f} = \texttt{i}! \wedge \neg(\texttt{i} < \texttt{n})\}$
$\Downarrow$
$\{\texttt{f} = \texttt{n}!\}$

Question 3.

(a) La règle pour la construction `si .. alors ... sinon` est la suivante :

$$\frac{\{P \wedge b\}C_1\{Q\} \qquad \{P \wedge \neg b\}C_2\{Q\}}{\{P\}\texttt{si } b \texttt{ alors } C_1 \texttt{ sinon } C_2\{Q\}}$$

(b) La règle pour la construction $t\texttt{[}e_1\texttt{]} \texttt{ := } e_2$ est la suivante :

$$\frac{}{\{0 \leq e_1 < \texttt{longueur}(t) \wedge P[t \,/\, t(e_1 \mapsto e_2)]\} \;\; t\texttt{[}e_1\texttt{]} \texttt{ := } e_2 \;\; \{P\}}$$

Où la notation $t(i \mapsto n)$ désigne le tableau t dans lequel on a modifié la position i pour y mettre l'entier n.

Note : La règle $\{0 \leq e_1 < \texttt{longueur}(t) \wedge P[t[e_1] \,/\, e_2]\} \;\; t\texttt{[}e_1\texttt{]} \texttt{ := } e_2 \;\; \{P\}$ est fausse. En effet, il est possible de parler de P de la cellule $t[e_1]$ sans que l'expression $t[e_1]$ apparaisse syntaxiquement.

Question 4. *Ici encore, on ne cherche pas à expliciter tous les détails de l'application de chaque règle. On s'assure, à chaque fois, que l'invariant de boucle est suffisant, et que le candidat comprend*

(a) On cherche donc à prouver le triplet $\{\top\}P_0\{\top\}$.

La seule cause d'erreur dans ce langage de programmation est l'écriture en dehors des bornes d'un tableau. Pour prouver le résultat demandé ici, il est donc uniquement nécessaire de s'assurer que la variable `i` est toujours dans l'intervalle voulu. La condition de la boucle permet de s'assurer que `i` n'est pas trop grand. Il reste donc à s'assurer qu'il n'est pas trop faible : on va donc utiliser comme invariant de boucle $0 \leq \texttt{i}$.

Cela donne la preuve suivante (on n'explicite plus les utilisations de la règle de conséquence) :

```
{⊤}
i := 0
{0 ≤ i}
tant que i < longueur(t) faire
    {0 ≤ i < longueur(t)}
    si i == 0 ou t[i] >= t[i-1]
        i := i+1
        {0 ≤ i}
    sinon
        {0 < i < longueur(t)}
        tmp := t[i];
        t[i] := t[i-1];
        t[i-1] := tmp;
        {0 < i}
        i := i-1
        {0 ≤ i}
    {0 ≤ i}
{⊤}
```

(b) On fixe maintenant un multi-ensemble T_0 et on cherche à prouver le triplet :

$$\{\texttt{contenu(t)} = T_0\}P_0\{\texttt{contenu(t)} = T_0\}$$

La raison pour laquelle le contenu du tableau ne change pas est que la seule opération effectuée par le programme sur le tableau est l'échange de deux valeurs consécutives. Il est donc naturel d'utiliser $\{\texttt{contenu(t)} = T_0\}$ comme invariant. Naturellement, il faut aussi conserver l'invariant de la question précédente ($0 \leq \texttt{i}$), ainsi que les assertions intermédiaires nécessaires à la preuve du (a). On ne fait pas réapparaître ces assertions ci-dessus : dans une réelle preuve de ce triplet, elles devront aussi apparaître.

```
{contenu(t) = T_0}
i := 0
tant que i < longueur(t) faire
    {contenu(t) = T_0}
    si i == 0 ou t[i] >= t[i-1]
        i := i+1
    sinon
        {contenu(t) = T_0 ∧ 0 < i < longueur(t)}
            ⇓
        {contenu(t(i ↦ t[i − 1])((i − 1) ↦ t[i])) = T_0}
        tmp := t[i];
        {contenu(t(i ↦ t[i − 1])((i − 1) ↦ tmp)) = T_0}
        t[i] := t[i-1];
        {contenu(t((i − 1) ↦ tmp)) = T_0}
        t[i-1] := tmp;
        {contenu(t) = T_0}
        i := i-1
{contenu(t) = T_0}
```

(c) En plus de la spécification prouvée en (b), il faut de plus prouver ici que le tableau est bien trié à la fin de l'exécution de l'algorithme. En d'autres termes, on cherche à prouver le triplet de Hoare :

$$\{\texttt{contenu(t)} = T_0\} P_0 \{\texttt{contenu(t)} = T_0 \wedge \texttt{t est trié}\}$$

Pour prouver ce dernier triplet, il faut utiliser l'invariant suivant : "`t` est trié jusqu'à la position `i-1`". Il est bien entendu toujours nécessaire de conserver les invariants trouvés en (a) et (b). Encore une fois, pour alléger les notations, nous ne répéterons pas les parties de la preuve déjà décrites dans les parties précédentes. Il est cependant nécessaire de les faire apparaître dans une preuve en logique de Hoare.
On obtient alors la preuve suivante :

```
i := 0
tant que i < longueur(t) faire
    {t[0] ≤ ... ≤ t[i-1]}
    si i == 0 ou t[i] >= t[i-1]
        {t[0] ≤ ... ≤ t[i]}
        i := i+1
        {t[0] ≤ ... ≤ t[i-1]}
    sinon
        {t[0] ≤ ... ≤ t[i-2]}
        tmp := t[i];
        t[i] := t[i-1];
        t[i-1] := tmp;
        {t[0] ≤ ... ≤ t[i-2]}
        i := i-1
        {t[0] ≤ ... ≤ t[i-1]}
    {t[0] ≤ ... ≤ t[i-1]}
{t[0] ≤ ... ≤ t[i-1] ∧ i ≥ longueur(t)}
    ⇓
{t[0] ≤ ... ≤ t[longueur(t)-1]}
    ⇓
{t est trié}
```

J3 – Cribles

On dit qu'un entier naturel est *sans carré* s'il n'est pas divisible par le carré d'un entier supérieur ou égal à 2.

Question 0. Donner le pseudo-code d'un algorithme permettant de calculer le nombre d'entiers naturels sans carré inférieurs ou égaux à n. Quelles sont ses complexités en temps et en espace ?

On note $\pi(n)$ le nombre de nombres premiers inférieurs ou égaux à n.

On admet le *théorème des nombres premiers* : quand n tend vers l'infini, $\pi(n)$ est équivalent à $\frac{n}{\ln n}$. Ceci implique en particulier que la valeur du k-ème nombre premier p_k est équivalente à $k \ln k$.

Question 1. Donner le pseudo-code d'un algorithme permettant de calculer $\pi(n)$. Quelles sont ses complexités en temps et en espace ?

Question 2. Décrire une structure de données permettant de représenter un sous-ensemble E de $[\![1, n]\!]$ et supportant les opérations suivantes :

— initialisation à $[\![1, n]\!]$ en temps $O(n)$,

— suppression d'un élément en temps $O(1)$,

— énumération dans l'ordre croissant de tous les éléments de l'ensemble $E \subseteq [\![1, n]\!]$ en temps $O(|E|)$.

La structure de données occupera un espace $O(n)$.

Question 3. En déduire le pseudo-code d'un algorithme qui permet de résoudre la question 1 en temps $O(n)$.

Question 4. Donner le pseudo-code d'un algorithme permettant de calculer tous les $\text{pgcd}(p, q)$ pour $p, q \in [\![1, n]\!]$. Quelle est sa complexité en temps ?

On admet qu'il est possible d'implémenter une structure de donnée de dictionnaire dont les clés sont des entiers (non nécessairement contigus) de telle façon que tous les accès soient en temps $O(1)$. Autrement dit, on peut se donner une structure T de sorte que, pour tout entier $i \in \mathbb{N}$, on puisse lire la valeur $T[i]$ ou écrire la valeur $T[i]$ en temps $O(1)$; la structure utilise un espace mémoire $O(n)$ où n est le nombre d'entiers i différents pour lesquels on a écrit $T[i]$.

Question 5. On note $\Phi(n)$ le nombre de paires d'entiers (p, q), avec $1 \leq p \leq n$, $1 \leq q \leq n$, et p et q premiers entre eux. Démontrer la formule de récurrence :

$$\Phi(n) = n^2 - \sum_{d=2}^{n} \Phi\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$$

En déduire le pseudo-code d'un algorithme permettant de calculer $\Phi(n)$ en temps sous-linéaire par rapport à n. Quelles sont ses complexités en temps et en espace ?

Corrigé

Question 0. On adapte le crible d'Ératosthène pour éliminer les facteurs carrés au lieu d'éliminer les facteurs premiers :

```
sanscarre <- tableau de taille n+1 contenant Vrai
pour x de 2 à sqrt(n)
  pour xx de x*x à n, en incrémentant par x*x à chaque itération
    sanscarre[xx] <- Faux
pour x de 2 à n
  si sanscarre[x]
    afficher x
```

La complexité en espace est $O(n)$.

Le nombre d'itérations de la boucle interne pour une valeur de x est (à quelques unités près) $\frac{n}{x^2}$. Par conséquent, la complexité en temps est :

$$O\left(\sum_{2\leq x\leq\sqrt{n}}\frac{n}{x^2}\right) = O\left(n\sum_{x=1}^{+\infty}\frac{1}{x^2}\right) = O(n)$$

Question 1. On peut utiliser un *crible d'Ératosthène*, dont le pseudo-code est le suivant :

```
premier <- tableau de taille n+1 contenant Vrai
cnt <- 0
pour p de 2 à n
  si premier[p]
    cnt <- cnt+1
    pour pp de 2*p à n, en incrémentant par p à chaque itération
      premier[pp] <- Faux
renvoyer cnt
```

La complexité en espace est $O(n)$.

Le nombre d'itérations de la boucle interne, pour le nombre premier p, est (à quelques unités près) $\frac{n}{p}$. Par conséquent, la complexité en temps est :

$$O\left(\sum_{\substack{p \text{ premier}\\ p\leq n}}\frac{n}{p}\right) = O\left(n\sum_{k=1}^{\pi(n)}\frac{1}{k\ln k}\right) = O\left(n\times\ln\ln\pi(n)\right) = O(n\ln\ln n)$$

Question 2. *Note : on ne demande pas que le candidat donne un pseudo-code pour cette question. Par contre, on demande à ce que le pseudo-code en question soit détaillé dans la question 3.*

Pour chaque élément $x \in E$, on stocke, à la position x de deux tableaux de taille n, l'élément suivant et précédent de l'ensemble. S'il n'y a pas un tel élément suivant ou précédent, on stocke une valeur spéciale (0 et $n+1$ dans ce pseudo-code). On stocke aussi dans une variable le premier élément. On obtient le pseudo-code suivant :

```
Fonction init(n)
  suivants := tableau de taille n+1
  précédents := tableau de taille n+1
  pour i de 1 à n
    suivants[i] := i+1
```

```
    précédents[i] := i-1
    premier := 1

Fonction supprimer(i)
  Si suivants[i] != n+1
     précédents[suivants[i]] := précédents[i]
  Si précédents[i] != 0
     suivants[précédents[i]] := suivants[i]
  Sinon premier := suivants[i]

Fonction énumérer()
  x := premier
  Tant que x != n+1
    Afficher x
    x := suivants[x]
```

Question 3. Il s'agit du *crible d'Euler* : l'idée est de n'éliminer chaque nombre composite qu'une seule fois, lorsque l'on considère son plus petit facteur premier. Lorsque, dans le crible, on considère un nombre premier p, il faut donc un moyen d'obtenir l'ensemble des entiers de $[\![p+1, n]\!]$ dont le plus petit facteur premier est p. Quitte à multiplier par p, il faut donc un moyen d'obtenir l'ensemble des entiers de $\left[\!\left[2, \left\lfloor \frac{n}{p} \right\rfloor\right]\!\right]$ qui ne sont pas divisibles par un autre nombre premier plus petit. Il se trouve que cet ensemble est exactement l'ensemble des entiers supérieurs à p qui restent non marqués au moment de considérer les multiples de p dans le crible.

On utilise donc la structure de donnée de la question 2 pour maintenir cet ensemble et l'énumérer.

Attention : il est tentant d'énumérer l'ensemble en même temps que l'on élimine les multiples de p. La conséquence, c'est que, pendant l'énumération, on risque de ne pas énumérer les multiples de p, et donc de ne pas éliminer les multiples de p^2. Il faut donc soit énumérer l'ensemble à l'avance, soit éliminer les multiples des puissances de p à la main.

On obtient alors le pseudo-code suivant :

```
suivants := tableau de taille n+1
précédents := tableau de taille n+1
cnt <- 0
Pour i de 2 à n
  suivants[i] := i+1
  précédents[i] := i-1
p := 2
Tant que p <= n
  x := p
  E := []
  cnt <- cnt+1
  Tant que x * p <= n
    E.push(x)
    x := suivants[x]
  Pour chaque x dans E
    suivants[précédents[x * p]] := suivants[x * p]
    Si suivants[x * p] != n + 1
      précédents[suivants[x * p]] := précédents[x * p]
  p := suivants[p]
```

Note : en pratique, cet algorithme n'est pas très utile. En effet, la liste doublement chaînée prend

beaucoup d'espace et est assez lente à maintenir, alors que le crible d'Ératosthène peut être implémenté très efficacement à l'aide de bitfields. Par ailleurs, le terme $\ln \ln n$ *n'est de toute façon jamais plus grand que 6. Ceci est largement compensé par l'efficacité des bitfields.*

Question 4. L'algorithme classique pour le calcul du PGCD est l'algorithme d'Euclide. Sa complexité en temps est logarithmique, mais ce n'est pas ce qui est demandé ici. En effet, on peut utiliser la mémoïsation pour s'assurer que la complexité globale est $O(n^2)$. D'ailleurs, on peut alors éviter la division, et la remplacer par une simple soustraction. Cela donne le pseudo-code suivant :

```
res <- tableau de taille [1..n]*[1..n] initialisé à -1
Fonction pgcd(p, q)
  Si p == 0 ou q == 0
    Renvoyer p+q
  Si res[p][q] == -1
    Si p < q
      res[p][q] <- pgcd(p, q-p)
    Sinon
      res[p][q] <- pgcd(p-q, q)
  Renvoyer res[p][q]
Pour p dans 1..n
  Pour q dans 1..n
    pgcd(p, q)
Renvoyer res
```

Sa complexité en temps est $O(n^2)$.

Question 5. Notons $\Phi_d(n) = \#\{(p,q) \mid \text{pgcd}(p,q) = d \wedge 1 \leq p,q \leq n\}$. On a donc $\Phi(n) = \Phi_1(n)$. On remarque tout d'abord que :

$$\begin{aligned}\Phi(n) &= \#\{(p,q) \mid \text{pgcd}(p,q) = 1 \wedge 1 \leq p,q \leq n\} \\ &= \#[\![1,n]\!]^2 - \sum_{d=2}^{n} \Phi_d(n)\end{aligned}$$

Mais deux entiers ont d pour PGCD ss'ils s'écrivent sous la forme $p_0 d$ et $q_0 d$ avec p_0 et q_0 premiers entre eux. On en déduit :

$$\Phi_d(n) = \Phi\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$$

Donc :

$$\Phi(n) = n^2 - \sum_{d=2}^{n} \Phi\left(\left\lfloor \frac{n}{d} \right\rfloor\right) \tag{1}$$

On peut implémenter cette formule directement. Si on utilise de la programmation dynamique, alors la complexité en espace sera $O(n)$ et la complexité en temps semble être $O(n^2)$. Ce n'est pas très efficace : en effet, l'algorithme de la question 4 permet de résoudre ce problème avec la même complexité.

Il est possible, cependant, d'implémenter cette formule plus efficacement.

Tout d'abord, on se rend compte que la version récursive de l'algorithme ne calcule pas toutes les valeurs de $\Phi(x)$ pour $x \in [\![1,n]\!]$. En effet, les seules valeurs calculées sont les $\Phi\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$, pour $d \in [\![1,n]\!]$. C'est un invariant de la formule de récurrence, dont la validité découle de l'égalité suivante :

$$\left\lfloor \frac{\left\lfloor \frac{n}{d} \right\rfloor}{d'} \right\rfloor = \left\lfloor \frac{n}{dd'} \right\rfloor$$

Or, il n'y a que $O(\sqrt{n})$ entiers qui peuvent s'écrire sous la forme $\left\lfloor \frac{n}{d} \right\rfloor$. En effet, ceux qui ne sont pas dans $[\![1, \lfloor\sqrt{n}\rfloor]\!]$ sont nécessairement de la forme $\left\lfloor \frac{n}{d} \right\rfloor$, avec $d \leq \sqrt{n}$.

Par ailleurs, dans la formule (1), de nombreux termes de la somme sont dupliqués : lorsque d est grand (i.e., $d \gg \sqrt{n}$), des valeurs voisines pour d donnent la même valeur pour $\lfloor n/d \rfloor$. Au lieu de faire ces appels récursifs identiques plusieurs fois un par un, on peut déterminer les premières et dernières valeurs de d pour lesquelles $\lfloor n/d \rfloor$ est identique, ne faire qu'un seul appel récursif, et multiplier le résultat par le nombre d'occurence ainsi calculé.

Plus précisément, on obtient la formule :

$$\Phi(n) = n^2 - \sum_{d=2}^{\lfloor\sqrt{n}\rfloor} \Phi\left(\left\lfloor \frac{n}{d} \right\rfloor\right) - \sum_{n'=1}^{\lceil\sqrt{n}\rceil-1} \left(\left\lfloor \frac{n}{n'} \right\rfloor - \left\lfloor \frac{n}{n'+1} \right\rfloor\right) \Phi(n') \tag{2}$$

Pour implémenter cette nouvelle formule, et ne calculer que les valeurs de Φ qui sont effectivement nécessaires tout en mémoïsant, le plus simple est d'utiliser un dictionnaire faisant correspondre à n l'entier $\Phi(n)$, lorsque celui-ci est connu. Ce dictionnaire peut par exemple être implémenté avec la table de hachage (dont le sujet nous permet de supposer l'existence), afin de permettre des accès en lecture et en écriture en $O(1)$.

On obtient le pseudo-code suivant :

```
dico <- Dictionnaire (int -> int) vide
Fonction Phi(n)
  Si dico[n] est vide
    res <- n*n
    Pour d = 2 à floor(sqrt(n))
      res <- res - Phi(floor(n/d))
    Pour n' = 1 à ceil(sqrt(n))-1
      res <- res - Phi(n') * (floor(n/n') - floor(n/(n'+1)))
    dico[n] <- res
  Renvoyer res

Renvoyer Phi(n)
```

Ainsi, le temps passé pour le calcul de $\Phi(n)$ sans compter les appels récursifs est $O(\sqrt{n})$. Le temps total est donc le suivant, où l'on comptabilise la complexité locale de chaque appel, et on scinde la somme entre les $\lfloor n/d \rfloor$ pour $d \leq \sqrt{n}$ (seconde somme), et les $\lfloor n/d \rfloor$ pour $d > \sqrt{n}$ où le résultat est donc $\leq \sqrt{n}$ (première somme) :

$$O\left(\sum_{i=1}^{\lfloor\sqrt{n}\rfloor} \sqrt{i} + \sum_{d=1}^{\lfloor\sqrt{n}\rfloor} \sqrt{\frac{n}{d}}\right) = O\left(\left(\sqrt{n}\right)^{3/2} + \sqrt{n} \sum_{d=1}^{\lfloor\sqrt{n}\rfloor} \frac{1}{\sqrt{d}}\right) = O\left(n^{3/4} + \sqrt{n}\sqrt{\sqrt{n}}\right) = O\left(n^{3/4}\right)$$

J4 – Théorie de Sprague-Grundy

Un *graphe* est une paire (V, E) d'un ensemble de sommets V et d'arêtes $E \subseteq V^2$. Un graphe est *acyclique* s'il existe une relation d'ordre total $\preccurlyeq$ sur V telle que $\forall (u, v) \in E,\ u \preccurlyeq v \wedge u \neq v$.

Un *jeu* est un triplet (S, T, ι), où S est un ensemble fini d'*états*, $T \subseteq S^2$ un ensemble de *transitions* et $\iota \in S$ un *état initial*, tels que le graphe (S, T) soit acyclique.

Une *stratégie* est une fonction partielle $\varphi : S \rightharpoonup S$ telle que, pour tout état s où elle est définie, $(s, \varphi(s)) \in T$. On peut faire jouer deux stratégies φ_0 et φ_1 l'une contre l'autre, en les faisant jouer alternativement. On définit ainsi la séquence d'états $s_0 = \iota, s_1 = \varphi_0(s_0), s_2 = \varphi_1(s_1), s_3 = \varphi_0(s_2), \ldots, s_k = \varphi_{(k-1) \bmod 2}(s_{k-1})$.

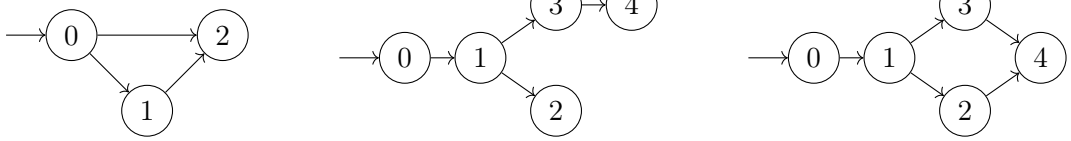
Question 0. Démontrer que la séquence (s_k) est finie, c'est-à-dire qu'il existe un entier n_{fin} tel que $\varphi_{n_{\text{fin}} \bmod 2}(s_{n_{\text{fin}}})$ n'est pas défini.

Pour deux stratégies φ_0 et φ_1 jouées par les joueurs 0 et 1, le joueur perdant est le premier joueur pour lequel la stratégie n'est plus définie. Formellement, si n_{fin} est pair, alors le joueur 0 perd, et le joueur 1 gagne. Inversement, si n_{fin} est impair, alors le joueur 1 perd et le joueur 0 gagne.

Une *stratégie gagnante* pour le joueur $i \in \{0, 1\}$ est une stratégie qui garantit que le joueur i gagne quand il joue suivant cette stratégie, quelle que soit la stratégie jouée par l'autre joueur.

Question 1.

(a) Parmi les jeux suivants, quels sont ceux qui ont une stratégie gagnante pour le joueur 0 ? Pour le joueur 1 ?



(b) On définit le jeu suivant : il y a un tas de n jetons entre les deux joueurs. Chacun son tour, un joueur peut prendre 1, 2, 3 ou 4 jetons. Le joueur qui prend le dernier jeton gagne. Expliquer comment cette description informelle du jeu peut se formaliser avec la notion formelle de jeu définie plus haut. Sous quelle condition sur n existe-t-il une stratégie gagnante pour le joueur 0 ? Pour le joueur 1 ?

Question 2. Soit un jeu (S, T, ι). Pour chacun de ses états $s \in S$, on définit sa *valeur de Grundy* $G(s) \in \mathbb{N}$ par :

$$G(s) = \min(\mathbb{N} \setminus \{G(s') \mid (s, s') \in T\})$$

(a) Expliquer pourquoi cette relation définit G de manière unique.

(b) Donner une condition nécessaire et suffisante sur G pour que le joueur 0 ait une stratégie gagnante. Qu'en est-il du joueur 1 ?

(c) Donner le pseudo-code d'un algorithme permettant de déterminer, étant donné un jeu, quel(s) joueur(s) a(ont) une stratégie gagnante. Quelle est sa complexité en temps et en espace ?

Si $\alpha, \beta \in \{0, 1\}$, on note $\alpha \oplus \beta = (\alpha + \beta) \bmod 2$ le *ou exclusif* de α et de β. Si $a = a_k \dots a_0$ et $b = b_k \dots b_0$ sont deux entiers positifs ou nuls écrits en base 2 (en ajoutant des 0 au début, le cas échéant), on note $a \oplus b$ l'entier positif ou nul dont la représentation en base 2 est $(a_k \oplus b_k) \dots (a_0 \oplus b_0)$.

Question 3. Dans le *jeu de Nim*, il y a n tas de $t_0, \dots, t_{n-1}$ jetons chacun entre les deux joueurs. Chacun son tour, un joueur choisit un tas, et en retire autant de jetons qu'il le souhaite. Le joueur qui prend le dernier jeton gagne. Démontrer qu'il existe une stratégie gagnante si et seulement si $t_0 \oplus \dots \oplus t_{n-1} \neq 0$.

Question 4. Si $J_1 = (S_1, T_1, \iota_1)$ et $J_2 = (S_2, T_2, \iota_2)$ sont deux jeux, alors leur *somme* $J_1 + J_2 = (S_{1+2}, T_{1+2}, \iota_{1+2})$ est définie par :

$$
\begin{aligned}
S_{1+2} &= S_1 \times S_2 \\
T_{1+2} &= \{((s_1, s_2), (s_1, s_2')) \mid s_1 \in S_1,\ (s_2, s_2') \in T_2\} \cup \{((s_1, s_2), (s_1', s_2)) \mid (s_1, s_1') \in T_1,\ s_2 \in S_2\} \\
\iota_{1+2} &= (\iota_1, \iota_2)
\end{aligned}
$$

(a) Soit (s_1, s_2) un état de $J_1 + J_2$. Démontrer que $G_{1+2}(s_1, s_2) = G_1(s_1) \oplus G_2(s_2)$, où $G_{1+2}(\cdot, \cdot)$, $G_1(\cdot)$ et $G_2(\cdot)$ désignent respectivement les valeurs de Grundy dans $J_1 + J_2$, J_1 et J_2.

(b) Quelle est la valeur de Grundy des états du jeu de Nim ? Commenter.

Corrigé

Question 0. Supposons qu'elle ne soit pas finie : puisque le graphe est fini, il existe deux indices distincts $i < j$ tels que $s_i = s_j$. Mais alors, si l'on considère la séquence $s_i, \dots, s_j$, on a que (s_k, s_{k+1}) est une arête du graphe pour tout $i \leq k < j$. Ainsi, on a $s_i \neq s_{i+1}$, on a $s_i \preccurlyeq s_{i+1}$, et on a par transitivité $s_{i+1} \preccurlyeq s_j$, ce qui contredit le caractère antisymétrique de $\preccurlyeq$.

Question 1. Tout d'abord, il est évident qu'un jeu ne peut pas être gagnant pour les deux joueurs. En effet, dans le cas contraire, on obtiendrait une contradiction en faisant jouer les stratégies gagnantes des joueurs 0 et 1 l'une contre l'autre.

(a) Tout état final (sans transition sortante) ne peut pas avoir d'image par une stratégie. Par conséquent, s'il existe une transition de l'état initial à un état final, alors la stratégie qui joue vers l'état final à partir de l'état initial est gagnante. Donc le premier jeu est gagnant pour le joueur 0.

Pour le deuxième jeu, la stratégie qui joue vers le bas dans l'état "fourche" est gagnante pour le joueur 1. En effet, le joueur 0 n'a de toute façon pas le choix pour son premier coup, et il se retrouve ensuite dans un état final, où il ne peut plus jouer.

Pour le troisième jeu, n'importe quelle stratégie maximale est gagnante pour le joueur 0.

(b) Ceci est bien un jeu au sens formel. En effet, on peut par exemple prendre :

$$\begin{aligned} S &= [\![1, n]\!] \\ T &= \{(s_1, s_2) \in S^2 \mid 0 < s_1 - s_2 \leq 4\} \\ \iota &= n \end{aligned}$$

Maintenant, considérons la stratégie φ définie par :

$$\varphi(x) = \begin{cases} x - (x \bmod 5) & \text{si } 5 \nmid x \\ \text{indéfini} & \text{sinon} \end{cases}$$

Il est facile de vérifier qu'il s'agit effectivement d'une stratégie pour ce jeu. Par ailleurs, si un joueur commence en jouant cette stratégie, alors soit $5 \mid n$, et il abandonne immédiatement, soit, par une récurrence immédiate, il ne jouera que des états x tels que $5 \nmid x$, et, par conséquent, il ne perdra pas. Par conséquent, φ est une stratégie gagnante pour le joueur 0 dès lors que $5 \nmid n$.

Mais, si $5 \mid n$, alors c'est une stratégie gagnante pour le joueur 1, puisque le joueur 0 est forcé de lui fournir un état x tel que $5 \nmid x$.

Pour conclure, si $5 \mid n$, alors le joueur 1 a une stratégie gagnante (et donc pas le joueur 0), et sinon c'est le contraire : le joueur 0 a une stratégie gagnante, mais pas le joueur 1.

Question 2.

(a) En reproduisant le raisonnement de la question 0, on prouve facilement que la longueur des chemins qui part de chaque état d'un jeu est borné. Pour un état s, soit $l(s)$ le maximum de cette longueur.

Notons que tous les successeurs s' d'un état s (c'est-à-dire les états s' tels que $(s, s') \in T$) sont tels que $l(s') < l(s)$. Par conséquent, la définition de $G(s)$ ne fait appel qu'à des valeurs de $G(s')$ telles que $l(s') < l(s)$: on peut par conséquent dire que G est définie de manière unique par la relation, inductivement sur $l(s)$.

Alternativement, on peut simplement faire appel à l'ordre $\preccurlyeq$ donné par le sujet, sans parler de la longueur des chemins.

(b) On va prouver que le jeu est gagnant pour le joueur 0 ssi $G(\iota) \neq 0$. Pour cela, on commence par remarquer que, si $G(s) \neq 0$, alors il existe $s' \in S$ tel que $G(s') = 0$. On peut donc définir une stratégie φ telle que, pour tout état s, si $G(s) \neq 0$, on a $G(\varphi(s)) = 0$. On va montrer que dès qu'un joueur est dans un état s tel que $G(s) \neq 0$, alors, s'il joue cette stratégie, il va gagner. En effet, après un coup, le joueur adverse est alors dans un état s' avec $G(s') = 0$. Selon la définition de G, le joueur adverse, s'il n'abandonne pas, est alors forcé de jouer vers un état s'' tel que $G(s'') \neq 0$. On prouve donc par récurrence que si le joueur joue la stratégie φ en commençant dans un état s tel que $G(s) \neq 0$, alors il ne jouera que dans de tels états, où φ est bien définie. C'est donc l'autre joueur qui va perdre.

Donc, lorsque $G(\iota) \neq 0$, le jeu est gagnant pour le joueur 0.

Inversement, si $G(\iota) = 0$, le joueur 0 est forcé de jouer dans un état s tel que $G(s) \neq 0$, et le joueur 1 pourra alors jouer la stratégie φ et gagner. Donc, dans cette situation, le jeu est gagnant pour le joueur 1.

Mais un jeu ne peut être gagnant pour les deux joueurs simultanément. Donc :

— Le jeu est gagnant pour le joueur 0 ssi $G(\iota) \neq 0$.

— Le jeu est gagnant pour le joueur 1 ssi $G(\iota) = 0$.

(c) Il suffit de calculer G pour tous les états, à l'aide de la programmation dynamique, puis d'utiliser le critère de la question précédente. Un et un seul joueur a une stratégie gagnante.

```
dyn <- Tableau de taille |S|, initialisé à -1
Fonction Grundy(s)
  Si dyn[s] = -1
    tabVu <- Tableau de taille |successeurs(s)| + 1 initialisé à Faux
    Pour chaque s' dans successeurs(s)
      soit g = Grundy(s')
      Si g <= |successeurs(s)|
        tabVu[g] = Vrai
    Pour i à partir de 0
      Si Non tabVu[i]
        dyn[s] = i
        Stop
  Renvoyer dyn[s]
Si Grundy(init) == 0 Renvoyer {1} Sinon Renvoyer {0}
```

La complexité en espace est $O(|T|)$, et, puisqu'on ne considère chaque transition qu'une fois au plus, la complexité en temps est $O(|T|)$.

Note : Au lieu de calculer la valeur exacte de $G(s)$, on peut simplement calculer un booléen correspondant à sa nullité. En pratique, on obtient un algorithme plus simple, la complexité en temps reste la même, et la complexité en espace devient $O(|S|)$.

Question 3. L'énoncé suggère qu'un état $(t_0, \ldots, t_n)$ du jeu de Nim est gagnant si et seulement si $\sigma := t_0 \oplus \ldots \oplus t_{n-1} \neq 0$. Inversement, tous les états tels que $\sigma = 0$ seraient perdants.

Pour démontrer ce résultat, il faut trouver une stratégie gagnante : en pratique, cela revient, pour chaque état :

— s'il vérifie $\sigma \neq 0$, à lui trouver un successeur vérifiant $\sigma = 0$,

— sinon, à prouver que tous ces successeurs vérifient $\sigma \neq 0$.

Si $(t_0, \ldots, t_n)$ est tel que $\sigma := t_0 \oplus \ldots \oplus t_{n-1} \neq 0$, il faut donc trouver un indice i et un entier $t'_i < t_i$ tel que $t_0 \oplus \ldots \oplus t'_i \oplus \ldots \oplus t_{n-1} = 0$. Pour cela, il faut et il suffit que $t'_i < t_i$ et que $t'_i = t_i \oplus \sigma$. Il suffit donc de trouver un i tel que $t_i \oplus \sigma < t_i$. Ceci se produit dès que le bit de poids le plus fort de σ (qui existe puisque $\sigma \neq 0$) est à 1 dans t_i. Une telle valeur de i existe nécessairement, puisque ce bit est à 1 dans $\sigma = t_0 \oplus \ldots \oplus t_{n-1}$.

Pour résumer, lorsqu'un joueur doit jouer dans un état $(t_0, \ldots, t_n)$ avec $\sigma := t_0 \oplus \ldots \oplus t_{n-1} \neq 0$, il trouve un indice i tel que le bit de poids le plus fort de σ soit allumé dans σ, et il retire suffisamment de jetons au tas i pour qu'il ne contienne plus que $t_i \oplus \sigma$ jetons dans ce tas. Le nouvel état vérifie alors $\sigma = 0$.

Maintenant, si on est dans un état tel que $t_0 \oplus \ldots \oplus t_{n-1} = 0$, alors n'importe quel coup transformera une valeur t_i en t'_i, et on aura alors $t_0 \oplus \ldots \oplus t'_i \oplus \ldots \oplus t_{n-1} = t_i \oplus t'_i$. Mais, puisque $t_i \neq t'_i$, on a $t_i \oplus t'_i \neq 0$, et le nouvel état vérifie bien $t_0 \oplus \ldots \oplus t'_i \oplus \ldots \oplus t_{n-1} \neq 0$

Question 4.

(a) On prouve le résultat par récurrence sur le graphe acyclique du jeu $J_1 + J_2$ (ou, plus précisément, par récurrence sur $l_1(s_1) + l_2(s_2)$ comme dans la question 2, ou sur les relations d'ordre pour G_1 et G_2). Pour cela, il faut, selon la définition des valeurs de Grundy, prouver deux assertions :

— Pour tout $g < G_1(s_1) \oplus G_2(s_2)$, il existe un successeur (s'_1, s'_2) de (s_1, s_2) tel que $g = G_{1+2}(s'_1, s'_2)$.

— Pour tout successeur (s'_1, s'_2) de (s_1, s_2), on a $G_{1+2}(s'_1, s'_2) \neq G_1(s_1) \oplus G_2(s_2)$.

Tout en sachant, par hypothèse de récurrence, que tout successeur (s'_1, s'_2) vérifie $G_{1+2}(s'_1, s'_2) = G_1(s'_1) \oplus G_2(s'_2)$.

Soit donc $g < G_1(s_1) \oplus G_2(s_2)$. On pose $\sigma = G_1(s_1) \oplus G_2(s_2) \oplus g$. Puisque $g \neq G_1(s_1) \oplus G_2(s_2)$, on a $\sigma \neq 0$. Le bit de poids le plus fort de σ est nécessairement à 1 dans $G_1(s_1) \oplus G_2(s_2)$ ou dans g. Tous les bits de poids supérieurs sont identiques dans $G_1(s_1) \oplus G_2(s_2)$ et dans g, puisqu'ils sont à 0 dans σ. Par conséquent, en utilisant l'inégalité $g < G_1(s_1) \oplus G_2(s_2)$, le bit de poids le plus fort de σ est nécessairement à 1 dans $G_1(s_1) \oplus G_2(s_2)$. Il existe alors $i \in \{1, 2\}$ tel que le bit de poids le plus fort de σ est à 1 dans $G_i(s_i)$. On a alors $G_i(s_i) \oplus \sigma < G_i(s_i)$, et il existe donc, par définition de G_i, un successeur s'_i de s_i tel que $G_i(s'_i) = G_i(s_i) \oplus \sigma$. En posant $s'_{3-i} = s_{3-i}$, l'état (s'_1, s'_2) est un successeur de l'état (s_1, s_2) dans le jeu $J_1 + J_2$, et $G_{1+2}(s'_1, s'_2) = G_1(s'_1) \oplus G_2(s'_2) = G_1(s_1) \oplus G_2(s_2) \oplus \sigma = g$.

Soit (s'_1, s'_2) un successeur de (s_1, s_2), et soit i l'indice tel que $s'_i \neq s_i$. On a $s'_{3-i} = s_{3-i}$ et, par définition de G_i, on a $G_i(s'_i) \neq G_i(s_i)$. Donc $G_1(s'_1) \oplus G_2(s'_2) \neq G_1(s_1) \oplus G_2(s_2)$.

(b) On prouve facilement (récurrence immédiate) que la valeur de Grundy d'un état d'un jeu de Nim à un tas est le nombre de jetons de ce tas. On remarque que le jeu de Nim général peut être obtenu en sommant des tas uniques. Donc en utilisant la question 4a, la valeur de Grundy d'un état $(t_0, \ldots, t_{n-1})$ du jeu de Nim est $t_0 \oplus \ldots \oplus t_{n-1}$.

Cela est cohérent avec les questions 2b et 3.

J5 – Arbres à boucs émissaires

Question 0.

(a) Rappeler ce qu'est un arbre binaire de recherche et comment un tel arbre peut être utilisé pour implémenter la structure de données de dictionnaire. On supposera que chaque clé est unique.

(b) Donner le pseudo-code de la fonction de recherche dans un arbre binaire de recherche. Quelle est sa complexité en pire cas ?

La *taille* d'un arbre binaire de recherche t est le nombre de nœuds qu'il contient. On la note $|t|$. Sa *hauteur* est le nombre maximal de nœuds sur un chemin allant directement de sa racine à l'une de ses feuilles. On la note $h(t)$.

Soit $\alpha \in \left[\frac{1}{2}, 1\right]$. On dit qu'un arbre est *α-équilibré en taille* si, pour chacun de ses sous-arbres t de fils gauche et droit g et d, on a :

$$|g| \leq \alpha\,|t| \qquad\qquad |d| \leq \alpha\,|t| \tag{1}$$

Question 1.

(a) Quels sont les arbres 1-équilibrés en taille ?

(b) Écrire le pseudo-code d'un algorithme permettant de construire un arbre de recherche $\frac{1}{2}$-équilibré en taille dont le contenu est donné par un tableau trié donné en entrée. Quelle est sa complexité en temps et en espace ?

On suppose maintenant que $\alpha \in \left]\frac{1}{2}, 1\right[$. On dit qu'un arbre t est *α-équilibré en hauteur* lorsque $h(t) \leq 1 + \frac{\ln|t|}{-\ln\alpha}$.

Question 2. Démontrer qu'un arbre non vide α-équilibré en taille est α-équilibré en hauteur.

Question 3. Donner la complexité de la recherche d'un élément dans un arbre α-équilibré en hauteur.

Pour insérer une nouvelle paire clef-valeur dans un arbre α-équilibré en hauteur, on commence par utiliser l'algorithme habituel. Si l'arbre n'est plus α-équilibré en hauteur, on cherche un *bouc émissaire* : il s'agit du plus proche ancêtre du nouveau nœud qui ne vérifie pas les inégalités (1). Puis, on remplace le sous-arbre correspondant au bouc émissaire par un arbre du même contenu construit avec l'algorithme de la question 1b.

Question 4.

(a) Un tel bouc émissaire existe-t-il toujours ?

(b) L'arbre ainsi obtenu est-il toujours α-équilibré en hauteur ?

(c) Quelle est sa complexité en temps et en espace, dans le pire et le meilleur cas ?

Pour faire une analyse plus fine de la complexité, on associe à chaque sous-arbre $t = N(g, k, v, d)$ la quantité $\Delta(t) = \max\{0, \mathrm{abs}(|g| - |d|) - 1\}$. De plus, on note $\Phi(t)$ la somme de tous les $\Delta(t')$ pour t' un sous-arbre (non nécessairement direct) de t.

Question 5.

(a) Donner une borne supérieure sur la variation de $\Phi(t)$ lors d'une insertion. Celle-ci dépendra, le cas échéant, de la taille du bouc émissaire.

(b) Quelle est la complexité globale en temps de l'insertion de N paires clef-valeur en partant d'un arbre vide ?

Corrigé

Question 0.

(a) Lorsqu'il est utilisé pour implémenter un dictionnaire, un arbre binaire de recherche est un arbre binaire dont les nœuds sont étiquetés par une clef et la valeur correspondante. Pour tout nœud interne $N(g, k, v, d)$ d'un arbre binaire de recherche, les nœuds du sous-arbre gauche g sont étiquetés par des clefs plus petites que k, et les nœuds du sous-arbre droit d sont étiquetés par des clefs plus grandes que k. Lorsque l'on veut lire ou modifier la valeur associée à une clef, on recherche le nœud correspondant en comparant les clefs avec la clef recherchée, et on y fait l'opération voulue. Lorsque l'on veut insérer une nouvelle valeur, on crée un nouveau nœud en bas de l'arbre, à la seule position possible.

Note : dans le pire cas (si les entrées sont insérées dans l'ordre), l'arbre devient déséquilibré, et les accès au dictionnaire sont en temps linéaire. Pour éviter ce problème et obtenir des temps d'accès logarithmiques, il faut équilibrer l'arbre. L'objectif de ce sujet est d'expliquer une méthode d'équilibrage.

(b)
```
Fonction cherche(t, k)
  Si t = N(g, k', v, d)
    Si k == k'
      Renvoyer v
    Si k < k'
      Renvoyer cherche(g, k')
    Si k > k'
      Renvoyer cherche(d, k')
  Sinon
    Renvoyer INTROUVABLE
```

Sa complexité est proportionnelle à la hauteur de l'arbre. Si l'arbre est déséquilibré, cela peut être linéaire en la taille de l'arbre.

Question 1.

(a) Bien sûr, on a toujours $|d| \leq |t|$ et $|g| \leq |t|$. Donc tout arbre binaire de recherche est 1-équilibré.

(b) Pour qu'un arbre soit 1/2-équilibré, il faut que :

$$|g| \leq \frac{1}{2}|t| = \frac{1}{2}(1 + |g| + |d|)$$

C'est-à-dire :

$$|g| \leq 1 + |d|$$

Et symétriquement :

$$|d| \leq 1 + |g|$$

On démontre facilement que ces deux conditions constituent une condition nécessaire et suffisante. Pour construire l'arbre à partir du tableau, il faut donc mettre une clef médiane à la racine de l'arbre, et recommencer récursivement pour les sous-arbres gauche et droit. On obtient donc le pseudo-code suivant :

```
Fonction construitArbre(tab, deb, fin)
  Si fin == deb
    Renvoyer Feuille
  Sinon
    Soit mid = (deb+fin)/2
    Renvoyer N(construitArbre(tab, deb, mid), tab[mid],
               construitArbre(tab, mid+1, fin))
```

Chaque appel récursif correspond exactement à une feuille ou un nœud. Puisqu'il en existe un nombre linéaire, la complexité en temps est linéaire. La complexité en mémoire, si on ne compte pas l'entrée et la sortie, est logarithmique (il s'agit de la taille de la pile, qui correspond à la hauteur de l'arbre).

Note : Il est important de ne pas copier le tableau, mais de réutiliser le même en passant des indices de début et fin. Sinon, la complexité devient $O(N \log N)$.

Question 2. On prouve cela par induction sur l'arbre. Si l'arbre est réduit à un nœud, alors $h(t) = 1$ et $|t| = 1$, et l'inégalité est vérifiée. Sinon, l'arbre est de la forme $N(g, k, v, d)$, avec :

$$|g| \leq \alpha\,|t| \qquad |d| \leq \alpha\,|t|$$
$$h(g) \leq 1 + \frac{\ln|g|}{-\ln\alpha} \qquad h(d) \leq 1 + \frac{\ln|d|}{-\ln\alpha}$$

Or :

$$\begin{aligned} h(t) &= 1 + \max\{h(g), h(d)\} \\ &\leq 2 + \frac{\ln\max\{|g|\,,|d|\}}{-\ln\alpha} \\ &\leq 2 + \frac{\ln(\alpha\,|t|)}{-\ln\alpha} \\ &= 1 + \frac{\ln|t|}{-\ln\alpha} \end{aligned}$$

Question 3. La profondeur d'un tel arbre étant logarithmique, la recherche d'un élément peut s'effectuer, par la fonction de la question 0, en temps logarithmique par rapport à sa taille.

Question 4. *Ce n'est pas demandé dans le sujet, mais voici le pseudo-code de l'algorithme en question. Note : il faut stocker la taille de l'arbre dans une variable globale* `t.taille` *pour ne pas avoir à la recalculer à chaque insertion.*

```
Fonction contenu(t, tab, deb)
  Si t = N(g, k, v, d)
    Soit fin = contenu(g, deb)
    t[fin] = (k, v)
    Renvoyer contenu(d, fin+1)
  Sinon // Feuille
    Renvoyer deb

Fonction taille(t)
  Si t = N(g, _, _, d)
    Renvoyer taille(g) + taille(d) + 1
  Sinon
    Renvoyer 0

Fonction insèreAux(t, k, v, h)
  Si t = F
     Si h < 0
       Renvoyer Déséquilibré(N(F, k, v, F), 1)
     Sinon
       Renvoyer Équilibré(N(F, k, v, F))
```

```
  Si t = N(g, k', v', d)
    Si k < k'
      resi = insèreAux(g, k, v, h-1)
      Si resi = Équilibré(g')
        Renvoyer Équilibré(N(g', k', v', d))
      Si resi = Déséquilibré(g', ng')
        d' = d
        nd' = taille(d')
    Sinon
      resi = insèreAux(d, k, v, h-1)
      Si resi = Équilibré(d')
        Renvoyer Équilibré(N(g, k', v', d'))
      Si resi = Déséquilibré(d', nd')
        g' = g
        ng' = taille(g')
    n = ng' + nd' + 1
    Si ng' > alpha * n Ou nd' > alpha * n
      tab = tableau de taille n
      contenu(N(g', k, v, d'), tab, 0)
      Renvoyer(Équilibré(construitArbre(tab, 0, n)))
    Sinon
      Renvoyer(Déséquilibré(N(g', k, v, d'), n))

Fonction insère(t, k, v)
  t.taille += 1
  t.arbre := insèreAux(t.arbre, k, v, 1-ln(t.taille)/ln(alpha))
```

(a) *Indication : remarquer que si l'insertion a cassé l'α-équilibre en hauteur, alors le nouveau nœud est forcément à la profondeur maximale, puisque la hauteur de l'arbre vient nécessairement de changer.*

Un tel bouc émissaire existe toujours. En effet, dans le cas contraire, en itérant la définition de l'α-équilibre en taille, on obtiendrait :

$$1 \leq \alpha^{h(t)-1} |t| < \alpha^{1+\frac{\ln|t|}{-\ln\alpha}-1} |t| = 1$$

Ce qui est contradictoire.

(b) S'il n'y a pas besoin de bouc émissaire, il n'y a rien à prouver. Sinon, soit t le bouc émissaire, avant qu'il ne soit rééquilibré. Puisque l'arbre était globalement α-équilibré en hauteur, le bouc émissaire contient un unique nœud de hauteur maximale : celui que nous venons d'insérer, et qui a cassé l'α-équilibre.

Par ailleurs, le bouc émissaire n'est pas α-équilibré en taille. On peut en déduire (comme à la question 1) que la différence de taille de ses deux enfants est plus grande ou égale à 2. Et avant l'insertion, cette différence était au moins égale à 1. Ceci prouve que, si l'on avait ignoré la valeur des clefs, le nouveau nœud aurait pu être placé à une position ne modifiant pas la hauteur de l'arbre bouc émissaire. Par conséquent, puisque le rééquilibrage remplace l'arbre bouc émissaire par un arbre de hauteur minimale pour sa taille, la hauteur du nouvel arbre est au plus égale à la hauteur du bouc émissaire *avant l'insertion.*

Puisque l'insertion n'augmente pas la hauteur de l'arbre, et que celui-ci était α-équilibré en hauteur avant l'insertion, il est bien α-équilibré en hauteur après l'insertion.

(c) S'il n'y a pas de bouc émissaire, alors la complexité est proportionnelle à la hauteur de l'arbre, c'est-à-dire logarithmique en sa taille. Si, au contraire, il y a eu rééquilibrage, alors il faut rajouter au terme logarithmique la taille du bouc émissaire, ce qui correspond au coût de son rééquilibrage.

Question 5.

(a) S'il n'y a pas de rééquilibrage, alors chaque $\Delta(t)$ ne peut varier que d'une unité au plus, et cette variation n'a lieu que pour les nœuds se trouvant sur le chemin de la racine au nouveau nœud. Étant donné que ce chemin est de longueur au plus $1 + \frac{\log|t|}{-\log\alpha}$, le potentiel $\Phi(t)$ n'augmentera pas plus que cette quantité.

Lors d'un rééquilibrage, on rajoute à cette quantité la variation de $\Phi(t)$ due à ce rééquilibrage. Après le rééquilibrage, tous les $\Delta(t')$ concernant des sous-arbres du sous-arbre reconstruit sont nuls.

Par ailleurs, si $t' = N(g, k, v, d)$ est un bouc émissaire, supposons, par exemple, $|g| > \alpha\,|t'|$. On a alors $|g| - |d| = 2\,|g| - |t'| + 1 > (2\alpha - 1)\,|t'| + 1$. Donc $\Delta(t') > (2\alpha - 1)\,|t'|$ (ceci étant vrai aussi lorsque $|d| > \alpha\,|t'|$).

Donc la variation de $\Phi(t)$ due au rééquilibrage est au plus $-(2\alpha - 1)\,|t'|$ si t' est le bouc émissaire.

Ainsi, lors d'une insertion faisant intervenir un rééquilibrage, la variation de $\Phi(t)$ est majorée par $1 + \frac{\log|t|}{-\log\alpha} - (2\alpha - 1)\,|t'|$, où t' est le bouc émissaire utilisé.

(b) On peut se servir de Φ comme une "réserve de temps" que l'on peut utiliser pour les insertions coûteuses. Ainsi, le coût d'une insertion est la somme du coût d'une insertion décrit dans la question 4c et de la variation de $\Phi(t)$, éventuellement multipliée par une constante. Si l'on choisit cette constante convenablement, le terme $(2\alpha - 1)\,|t'|$ de la variation de $\Phi(t)$ va compenser la reconstruction du bouc émissaire. Le coût résiduel de l'insertion d'un nouveau nœud est donc logarithmique, et la complexité totale associée à la création d'un arbre de taille N est donc $O(N \log N)$.

L1 – May the forcing be with you

On considère l'alphabet $\Sigma = \{0, 1\}$. Un *mot* est une suite finie dans l'alphabet Σ. On note Σ^* l'ensemble des mots. Pour $\sigma, \tau \in \Sigma^*$, on note $\sigma \preceq \tau$ si σ est un préfixe de τ. Par exemple, $010010 \preceq 010010110$. L'ensemble Σ^*, muni de la relation de préfixe $\preceq$, forme un ordre partiel.

Un ensemble $D \subseteq \Sigma^*$ est *dense* dans $(\Sigma^*, \preceq)$ si tout mot $\sigma \in \Sigma^*$ est préfixe d'un mot dans D.

Question 0. Parmi les ensembles suivants, lesquels sont denses ?

(1) $D_1 = \{\sigma \in \Sigma^* : (\exists i)\sigma(i) = 1\}$
(2) $D_2 = \{\sigma \in \Sigma^* : (\forall i)\sigma(i) = 1\}$
(3) L'ensemble des mots qui sont la représentation binaire d'un nombre premier, avec les bits de poids fort à droite.

Question 1. Montrer que pour tout ensemble $D \subseteq \Sigma^*$, l'ensemble suivant est dense :

$$D' = D \cup \{\sigma \in \Sigma^* : (\forall \tau \succeq \sigma)\tau \notin D\}$$

On identifie une fonction $f : \mathbb{N} \to \Sigma$ avec une suite infinie $f(0), f(1), \ldots$

Un mot $\sigma = b_0 b_1 \ldots b_{n-1}$ est *préfixe* d'une fonction $f : \mathbb{N} \to \Sigma$ (noté $\sigma \preceq f$) si $b_i = f(i)$ pour tout $i < n$. Un ensemble $D \subseteq \Sigma^*$ *rencontre* une fonction $f : \mathbb{N} \to \Sigma$ s'il existe un préfixe fini $\sigma \in D$ tel que $\sigma \preceq f$.

Question 2. Montrer que pour toute énumération d'ensembles denses $D_0, D_1, \cdots \subseteq \Sigma^*$, il existe une fonction $f : \mathbb{N} \to \Sigma$ qui les rencontre tous.

On considère un *programme informatique* de manière abstraite comme une fonction partielle de type $\mathbb{N} \to \Sigma$. En pratique, on peut choisir son langage de programmation préféré (C, C++, Java, Python, ...) et considérer un programme concret dans ce langage. Étant donné un programme informatique P et une entrée $n \in \mathbb{N}$, on écrit $P(n) \downarrow$ si le programme P s'arrête en un temps fini sur l'entrée n, et $P(n) \uparrow$ s'il tourne à l'infini. Autrement dit, la notation $P(n) \uparrow$ signifie que le programme P vu comme une fonction partielle n'est pas défini en n. Quand $P(n) \downarrow$, on note $P(n)$ la valeur retournée par le programme. Il est possible de fixer une énumération de tous les programmes informatiques $P_0, P_1, P_2, \ldots$

Question 3. Montrer que pour tout $e \in \mathbb{N}$, l'ensemble suivant est dense :

$$D_e = \{\sigma \in \Sigma^* : \exists i[P_e(i) \uparrow \text{ ou } \sigma(i) \neq P_e(i)]\}$$

où $\sigma(i)$ est la valeur de σ à la position i en commençant à 0.

Une fonction $f : \mathbb{N} \to \Sigma$ est *calculable* s'il existe un programme informatique P_e tel que pour tout $i \in \mathbb{N}$, $P_e(i) \downarrow$ et $P_e(i) = f(i)$. Autrement dit, la fonction f est calculable s'il existe un programme en C, C++, ... qui, pour toute entrée $i \in \mathbb{N}$, va s'arrêter après un nombre fini d'étapes, et retourner la valeur de $f(i)$.

Question 4. Soit $f : \mathbb{N} \to \Sigma$ une fonction rencontrant tous les ensembles denses de la question 3. Montrer que f n'est pas calculable.

Une fonction $f : \mathbb{N} \to \Sigma$ est *univers* si n'importe quel mot fini sur Σ est facteur du mot infini f.

Question 5. Adapter la question 4 pour montrer qu'il existe une fonction univers qui n'est pas calculable.

Suite des questions

On peut augmenter les langages de programmation en ajoutant des fonctions arbitraires qui ne sont pas forcément calculables. Étant donnée une fonction $f : \mathbb{N} \to \Sigma$, on note $P_0^f, P_1^f, \dots$ la liste de tous les programmes informatiques codés dans le langage de programmation augmenté de f (sans forcément faire appel à f). La fonction f peut être vue comme un *oracle* que l'on interroge pour prendre des décisions dans le programme. Concrètement, le programme informatique peut avoir des instructions du type if$(f(x) == y)\{...\}$ où x et y sont des variables d'entiers.

Question 6. Expliquer pourquoi si $P_e^f(n) \downarrow$, alors seulement un nombre fini de valeurs de f seront utilisées dans l'exécution.

On étend donc la notation P_e^σ aux mots finis $\sigma = b_0 b_1 \dots b_{n-1}$. Ainsi, $P_e^\sigma(i) \downarrow$ si le programme P_e s'arrête sur l'entrée i et n'interroge la fonction f que sur des valeurs inférieures à n (autrement dit dans le domaine de définition de σ). On note $P_e^\sigma(i) \uparrow$ dans le cas contraire, c'est-à-dire si le programme P_e ne s'arrête pas sur l'entrée i, ou bien s'il interroge f sur une valeur supérieure à n.

Toute séquence de mots finis, strictement croissante par la relation de préfixe, induit une fonction unique $f : \mathbb{N} \to \Sigma$. En ce sens, un mot fini $\sigma \in \Sigma^*$ de cette séquence peut être considéré comme une approximation finie de la fonction $f : \mathbb{N} \to \Sigma$ que l'on construit. Étant donnée une approximation finie $\sigma \in \Sigma^*$, les fonctions *candidates* sont celles ayant σ pour préfixe.

Certaines propriétés de la fonction résultante f sont déjà déterminées par l'approximation σ. Par exemple, si $\sigma(3) = 1$, toute fonction candidate ayant σ pour préfixe satisfera $f(3) = 1$. D'autres propriétés ne sont pas encore déterminées. Par exemple, si σ a pour longueur 10, la propriété $f(42) = 0$ n'est pas déterminée. En particulier, il existe une extension τ de σ satisfaisant $\tau(42) = 0$ et une autre extension ρ de σ satisfaisant $\rho(42) = 1$.

On dit que σ *force* $P_e^{\mathtt{f}}(n) \downarrow$ si $P_e^\sigma(n) \downarrow$ et σ *force* $P_e^{\mathtt{f}}(n) \uparrow$ si pour tout suffixe $\tau \succeq \sigma$, $P_e^\tau(n) \uparrow$.

Question 7. Montrer que pour tout $e, n \in \mathbb{N}$ et pour tout mot fini σ, il existe une extension τ forçant soit $P_e^{\mathtt{f}}(n) \downarrow$, soit $P_e^{\mathtt{f}}(n) \uparrow$. Autrement dit, montrer pour $e, n \in \mathbb{N}$ que l'ensemble $D_0 \cup D_1$ est dense, où

$$D_0 = \{\sigma \in \Sigma^* : \sigma \text{ force } P_e^{\mathtt{f}}(n) \downarrow\} \text{ et } D_1 = \{\sigma \in \Sigma^* : \sigma \text{ force } P_e^{\mathtt{f}}(n) \uparrow\}$$

Question 8. Montrer que si $f : \mathbb{N} \to \Sigma$ rencontre D_0, alors $P_e^f(n) \downarrow$ et si f rencontre D_1, alors $P_e^f(n) \uparrow$.

Question 9. Soit $g : \mathbb{N} \to \Sigma$ une fonction non calculable. Montrer pour $e \in \mathbb{N}$ que l'ensemble $D_0 \cup D_1$ est dense, où

$$D_0 = \{\sigma \in \Sigma^* : (\exists n)\sigma \text{ force } P_e^{\mathtt{f}}(n) \neq g(n)\} \text{ et } D_1 = \{\sigma \in \Sigma^* : (\exists n)\sigma \text{ force } P_e^{\mathtt{f}}(n) \uparrow\}$$

Question 10. Soit $g : \mathbb{N} \to \Sigma$ une fonction non calculable. Adapter la question 5 et la question 9 pour montrer qu'il existe une fonction $f : \mathbb{N} \to \Sigma$ univers qui n'est pas calculable et telle que $g \neq P_e^f$ pour tout $e \in \mathbb{N}$.

Corrigé

Question 0. (1) D_1 est dense, car tout mot fini peut être étendu en un mot contenant au moins un bit de valeur 1. (2) D_2 n'est pas dense. En effet, le mot $\sigma = 0$ de longueur 1 n'est pas extensible en un élément de D_2. (3) Cet ensemble n'est pas dense, car toute extension du mot $\sigma = 011$ sera la représentation binaire d'un nombre pair supérieur à 2 et donc ne représentera pas un nombre premier.

Question 1. Soit $\sigma \in \Sigma^*$. Soit il existe une extension de σ dans D, donc dans D', soit aucune extension de σ n'appartient à D, auquel cas $\sigma \in \{\sigma \in \Sigma^* : (\forall \tau \succeq \sigma)\tau \notin D\}$ et donc $\sigma \in D'$. Dans les deux cas, σ est le préfixe d'un mot dans D'.

Question 2. On définit par induction une séquence de mots strictement croissante par la relation de préfixe ($\sigma_0 \prec \sigma_1 \prec \dots$) telle que pour tout $i \in \mathbb{N}$, $\sigma_i \in D_i$. Commençons avec le mot vide σ_{-1}. Ayant défini σ_{i-1}, comme l'ensemble D_i est dense, il existe une extension de σ_{i-1} dans D_i. Soit σ_i cette extension. On continue ainsi de suite. La fonction $f : \mathbb{N} \to \Sigma$ est définie comme l'unique fonction telle que $\sigma_i \preceq f$ pour tout $i \in \mathbb{N}$. Note : pour obtenir une séquence strictement croissante et ainsi garantir l'unicité de la fonction f, au lieu de prendre une extension de σ_{i-1} dans D_i, on peut prendre une extension de $\sigma_{i-1}0$ dans D_i, où $\sigma_{i-1}0$ est le mot σ_{i-1} concaténé avec un 0.

Question 3. Si P_e est partiel, c'est-à-dire s'il existe un $i \in \mathbb{N}$ tel que $P_e(i) \uparrow$, alors $D_e = \Sigma^*$ et est trivialement dense. En effet, tout $\sigma \in \Sigma^*$ a une extension dans D_e, à savoir σ lui-même. Si P_e est total, alors pour tout $\sigma = b_0 b_1 \dots b_{n-1}$, soit $b_n \in \{0, 1\}$ tel que $P_e(n) \neq b_n$. Alors $b_0 b_1 \dots b_n$ est une extension de σ dans D_e.

Question 4. Raisonnons pas l'absurde, et supposons que f est calculable. Alors il existe un $e \in \mathbb{N}$ tel que pour tout $i \in \mathbb{N}$, $P_e(i) \downarrow$ et $P_e(i) = f(i)$. Comme f rencontre D_e, il existe un préfixe fini $\sigma \preceq f$ dans D_e. En particulier, il existe un $i \in \mathbb{N}$ tel que $\sigma(i) \neq P_e(i)$. Comme $\sigma(i) = f(i)$, on obtient une contradiction.

Question 5. Pour tout mot fini $\tau \in \Sigma^*$, l'ensemble D_τ de tous les mots finis ayant τ comme suffixe est dense. En effet, pour tout $\sigma \in \Sigma^*$, il suffit de concaténer τ à σ pour obtenir une extension dans D_τ. Soit $f : \mathbb{N} \to \Sigma$ une fonction rencontrant tous les ensembles D_e de la question 3, et tous les ensembles D_τ définis précédemment. Par la question 4, f est une fonction non calculable, et comme elle rencontre tous les ensembles D_τ, c'est une fonction univers.

Question 6. Si $P_e^f(n) \downarrow$ s'arrête, c'est en un temps fini, donc après avoir exécuté un nombre fini d'instructions et donc avoir évalué f sur un nombre fini de valeurs.

Question 7. Si on déroule les définitions de D_0 et D_1, on s'aperçoit qu'il s'agit d'un cas particulier de la question 1, avec $D = D_0$ et $D' = D_0 \cup D_1$.

Question 8. (1) Supposons que f rencontre D_0. Autrement dit, il existe un préfixe fini $\sigma \preceq f$ tel que σ force $P_e^{\mathtt{f}}(n) \downarrow$. En déroulant la définition, on obtient $P_e^\sigma(n) \downarrow$, donc que le programme P_e s'arrête sur l'entrée n en n'appelant que des valeurs dans le domaine de σ. En remplaçant σ par f, on augmente le domaine de l'oracle P_e^f, mais cela n'aura pas d'impact sur l'exécution de $P_e^f(n)$ car f n'est évaluée que sur le domaine de σ.

(2) Supposons que f rencontre D_1. Il existe donc un préfixe $\sigma \preceq f$ tel que pour tout suffixe $\tau \succeq \sigma$, $P_e^\tau(n) \uparrow$. Raisonnons par l'absurde. Si $P_e^f(n) \downarrow$, alors par la question 6, un nombre fini de valeurs de l'oracle f sont interrogées par l'algorithme. Soit $\tau \preceq f$ un préfixe plus grand que σ tel que l'exécution

de $P_e^f(n)$ ne fasse appel qu'à des valeurs dans le domaine de τ. Ainsi τ est une extension de σ telle que $P_e^\tau(n)\downarrow$, contradiction.

Question 9. Soit σ un mot. Prouvons qu'il existe une extension de σ dans $D_0 \cup D_1$. Définissons le programme h qui en entrée $n \in \mathbb{N}$ cherche une extension $\tau \succeq \sigma$ telle que $P_e^\tau(n)\downarrow$. Si une telle extension est trouvés, alors $h(n) = P_e^\tau(n)$. Sinon, $h(n)$ ne s'arrête pas. La fonction h est partielle calculable. Deux cas sont possibles :

— Cas 1 : Il existe un n tel que $h(n)\uparrow$. Alors la condition σ force $P_e^{\mathtt{f}}(n)\uparrow$, donc $\sigma \in D_1$.

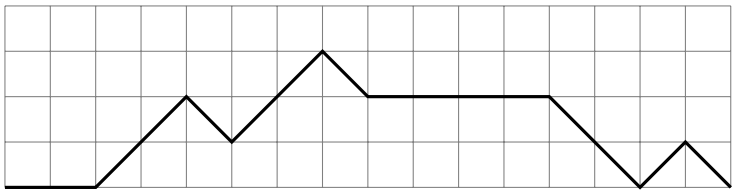
— Case 2 : La fonction h est totale calculable. Comme g n'est pas calculable, il existe un n tel que $h(n) \neq g(n)$. Par définition de h, il existe une extension τ de σ telle que $P_e^\tau(n) \neq h(n)$. En particulier, τ est une extension de σ dans D_0.

Tout mot σ ayant une extension dans D_0 ou D_1, l'ensemble $D_0 \cup D_1$ est dense.

Question 10. Soit f une fonction rencontrant les ensembles denses de la question 5 et de la question 9. Par la question 5, f est univers et non calculable. Par la question 9, $g \neq P_e^f$ pour tout $e \in \mathbb{N}$. =

L2 – Nombres de Schröder

Un *chemin de Schröder de longueur* $2n$ est un chemin de $(0,0)$ à $(2n,0)$ formés de pas unitaires nord-est et sud-est (pas $(1,1)$ ou $(1,-1)$) ou de pas horizontaux doubles (pas $(2,0)$), et qui de plus sont toujours au-dessus de l'axe des x. Voici un exemple de chemin de Schröder :



Question 0. Dessiner les chemins de Schröder de longueur 2 et de longueur 4.

Question 1. Le n-ième *nombre de Schröder* S_n est défini comme le nombre de chemins de Schröder de longueur $2n$. Par convention, il existe un unique chemin de Schröder de longueur 0. Déterminer la formule de récurrence de S_{n+1} en fonction de $S_0, \ldots, S_n$.

Question 2. Écrire un algorithme qui prend en entrée une liste de coordonnées (x, y) triée par abscisses, et détermine s'il existe un chemin de Schröder passant par tous ces points. On considère qu'un pas horizontal passe également par son centre. Quelle est la complexité en temps ? en mémoire ?

Un chemin de Schröder est *aérien* s'il ne comporte aucun pas horizontal au niveau du sol. Sinon, le chemin est *terrestre*. Soit A_n le nombre de chemins aériens de Schröder de longueur 2n.

Question 3. Construire une bijection entre les chemins de Schröder terrestres de longueur $2n$ et les chemins de Schröder aériens de longueur $2n$. Que peut-on en déduire du lien entre S_n et A_n ?

Un arbre est *planaire* si l'ensemble des enfants d'un nœud est ordonné. Ainsi, deux arbres pouvant être rendus identiques en changeant l'ordre des enfants d'un nœud sont considérés comme différents. Un *bosquet* est un arbre planaire à branchement quelconque, possédant une racine, et au moins une arête, tel que tout sommet qui n'est ni une feuille, ni la racine possède au moins deux enfants.

Question 4. Dessiner les bosquets possédant 1, 2 et 3 feuilles.

Question 5. Soit B_n le nombre de bosquets à n feuilles. Montrer pour tout $n \geq 2$ qu'il existe $B_n/2$ bosquets dont la racine possède au moins deux enfants.

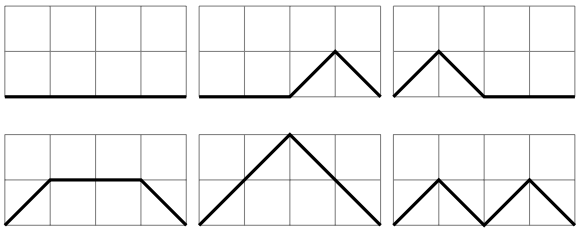
Question 6. Soit T un bosquet possédant n feuilles et p noeuds internes. Prouver que T possède $n + p - 1$ arêtes.

Question 7. Construire une bijection des bosquets à $n + 1$ feuilles vers les chemins de Schröder de longueur $2n$.

Corrigé

Question 0. Les chemins de Schröder de longueur 2 sont

Ceux de longueur 4 sont :



Question 1. Un chemin de Schröder de taille $2(n+1)$ commence soit par un pas horizontal $(2,0)$, auquel cas il est suivi d'un chemin de Schröder de taille $2n$, ou bien il commence par un pas ascendant $(0,1)$. Dans ce cas, soit $i > 0$ la première abscisse sur le chemin de type $(i,0)$. Les coordonnées $(0,1)$ et $(i,0)$ délimitent un chemin de Schröder de taille $2i$, et la coordonnée $(i,0)$ est suivie d'un chemin de Schröder de taille $2(n-i)$. Ainsi, $S_0 = 1$ et

$$S_{n+1} = S_n + \sum_{i=0}^{n} S_i S_{n-i}$$

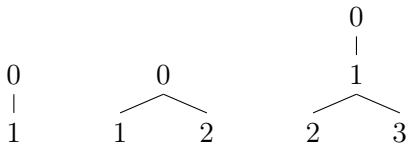
Question 2. On peut utiliser le pseudocode suivant :

```
Entrée L
(posX, posY) := (0, 0)
Tant que L <> []:
  (nextX, nextY) := tete(L)
  L = queue(L)
  // S'il faut monter ou descendre trop vite, impossible
  Si |nextY - posY| > nextX - posX:
    Renvoyer Non
  // Mise à jour des coordonnes après avoir atteint la bonne hauteur
  posY := nextY
  posX := posX + abs(nextY - posY)
  // Il reste à aller horizontalement pour rejoindre (nextX,nextY)
  // Attention, tout pas horizontal est de 2
  // Donc on atteint soit (nextX,nextY), soit (nextX+1,nextY)
  Si nextX - posX = 0 mod 2:
    posX := nextX
  Sinon
    posX := nextX+1
  Fin Si
Fin Tant que
Renvoyer Oui
```

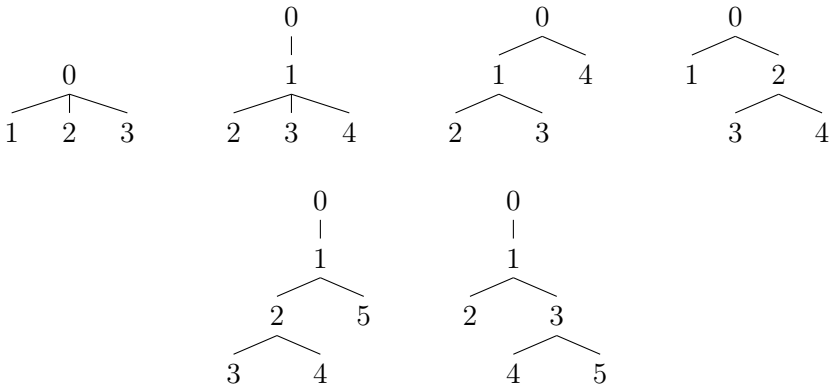
Question 3. Soit un chemin terrestre de Schröder. Considérons la dernière occurrence d'un pas horizontal au niveau du sol. Supprimer ce pas, et entourer la partie qui précède d'un pas montant et d'un pas descendant, afin de réhausser le préfixe. Par exemple :

Il s'agit clairement d'une bijection. On en déduit que $S_n = 2A_n$ pour tout $n \geq 1$.

Question 4. L'unique bosquet à une feuille et les bosquets à 2 feuilles sont



Les bosquets à 3 feuilles sont



Question 5. Les bosquets à n feuilles pour $n \geq 2$ se séparent en deux catégories disjointes :

— ceux dont la racine possède exactement un enfant

— ceux dont la racine possède au moins deux enfants

La première catégorie est en bijection avec la seconde, en supprimant la racine et l'unique arête partant de la racine. Les deux catégories ayant la même cardinalité, chacun a une cardinalité de $B_n/2$.

Question 6. Ce n'est pas spécifique aux bosquets, mais aux arbres. Soit T un arbre de n feuilles et p noeuds internes. Chaque feuille et chaque noeud interne qui n'est pas la racine est relié à son parent par une arête. Il existe donc $n + p - 1$ arêtes.

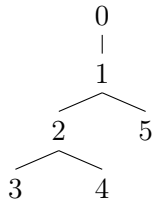
Question 7.

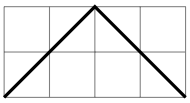
— *Indice 1 :* Faire un parcours en profondeur des arêtes.

— *Indice 2 :* Séparer le cas des branches au niveau de la racine, et des autres niveaux.

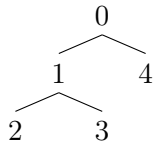
À partir d'un bosquet, on va suivre les arêtes par un parcours en profondeur pour construire un chemin de Schröder. On ne lit chaque arête qu'une seule fois. Une arête liée à la racine est transformée en un pas horizontal $(0, 2)$ *à part son tout premier nœud.* Le premier enfant d'un nœud qui n'est pas la racine est transformé en un pas montant $(1, 1)$, et le dernier enfant en un pas descendant $(1, -1)$. Tout autre enfant est transformé en un pas horizontal $(0, 2)$.

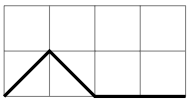
Par exemple,



devient


En effet, on considère l'arête (0,1) que l'on ignore puisqu'elle est la première arête liée à la racine. Ensuite l'arête (1,2) lie le premier enfant de 1, dont est un pas montant, puis l'arête (2,3) lie le premier enfant de 2, donc est un pas montant. Ensuite, l'arête (2,4) lie le dernier enfant de 2, donc un pas descendant, puis (1,5) lie le dernier enfant de 1, donc un pas descendant.



devient


En effet, on considère l'arête (0,1) que l'on ignore puisqu'elle est la première arête liée à la racine. Ensuite l'arête (1,2) lie le premier enfant de 1, dont est un pas montant, puis l'arête (1,3) lie le dernier enfant de 1, donc est un pas descendant. Enfin, (0,4) est une arête liée à la racine et est interprétée comme un pas horizontal.

Lemme 1. Le chemin obtenu est un chemin de Schröder. Il s'agit de montrer que pour tout pas montant, il existe un pas descendant, et que l'on maintient un nombre de pas montant supérieur au nombre de pas descendants tout au long de la construction. Les arêtes liées à la racine ne sont ni montantes ni descendantes. Chaque noeud autre que la racine a au moins deux enfants, donc sa première et sa dernière arête sont différentes. La première arête étant visitée avant sa dernière arête, le nombre de pas montants est à tout instant supérieur au nombre de pas descendants.

Lemme 2. Le chemin de Schröder est de taille $2n$. A chaque noeud qui n'est pas une feuille (contenant $x > 0$ enfants), on va ajouter un pas montant, un pas descendant, chacun de longueur 1, et $x - 2$ pas horizontaux de longueur 2. On va donc ajouter une longueur totale de $2(x - 1)$. Si m est le nombre d'arêtes du bosquet et p le nombre de noeuds qui ne sont pas des feuilles, le chemin va être de longueur $2(m - p)$. Par la question précédente, $m - p = n$, donc le chemin est de longueur $2n$.

Lemme 3. La transformation est injective. Immédiat. Si deux bosquets sont différents, il existe une étape dans le parcours en profondeur où les instructions seront différentes.

Lemme 4. La transformation est surjective. Étant donné un chemin de Schröder, il est facile de construire un bosquet pré-image.

L3 – Complexité de Kolmogorov

Un *mot binaire* est un mot fini dans l'alphabet $\Sigma = \{0, 1\}$. On note Σ^* l'ensemble des mots binaires. La longueur d'un mot $w \in \Sigma^*$ est notée $|w|$.

Fixons un langage de programmation L raisonnable (parmi Caml, C, Java, Python). Étant donné un mot binaire w, une *description de w dans le langage de programmation L* est un mot binaire de la forme $0^{|p|}1pu$ où p et u sont deux mots binaires tels que p est la représentation binaire d'un programme P écrit dans le langage de programmation L, et qui sur l'entrée u affiche le mot w. La *complexité de Kolmogorov* $K_L(w)$ de w dans L est définie par

$$K_L(w) = \min\{|d| : d \text{ est une description de } w \text{ dans } L\}$$

Question 0. Montrer que pour tout mot w, $K_L(w) \leq |w| + \mathcal{O}(1)$, où $\mathcal{O}(1)$ signifie "à constante près".

Question 1. Montrer que pour tout $n \in \mathbb{N}$, il existe un mot w de longueur $n+1$ tel que $K_L(w) > n$.

Question 2. Montrer que pour tous mots w et u, $K_L(wu) \leq 2K_L(w) + K_L(u) + \mathcal{O}(1)$. Peut-on améliorer cette borne ?

Question 3. Soit K_U la complexité de Kolmogorov définie pour un autre langage de programmation U. Montrer qu'il existe une constante $c \in \mathbb{N}$ telle que pour tout mot fini w, $K_L(w) \leq K_U(w) + c$.

Une fonction partielle $f : \Sigma^* \to \Sigma^*$ est *calculable* s'il existe un programme P qui lorsqu'il prend en entrée un mot binaire w, affiche $f(w)$ s'il est défini et n'affiche rien sinon.

Question 4. Montrer que pour toute fonction partielle calculable $f : \Sigma^* \to \Sigma^*$ et tout mot fini w, $K_L(f(w)) \leq K_L(w) + \mathcal{O}(1)$.

On note $\bar{n} \in \Sigma^*$ la représentation en base 2 d'un entier $n \in \mathbb{N}$. La complexité de Kolmogorov d'un entier $n \in \mathbb{N}$ est celle de $\bar{n}$. De même, on dit qu'une fonction $f : \mathbb{N} \to \Sigma^*$ est calculable s'il existe un programme P qui, lorsqu'il prend en entrée un mot binaire $w \in \Sigma^*$, affiche $f(n)$ s'il existe n tel que $\bar{n} = w$, et n'affiche rien sinon.

Question 5. Montrer que pour toute fonction calculable $f : \mathbb{N} \to \Sigma^*$, on a $K_L(f(n)) \leq \log_2(n) + \mathcal{O}(1)$.

Question 6. Donner un algorithme en pseudo-code calculant une fonction surjective de $\mathbb{N}$ dans Σ^*.

Question 7. Montrer que la fonction K_L n'est pas calculable.

Question 8. Montrer, en utilisant la complexité de Kolmogorov et l'unicité de la décomposition en facteurs premiers, qu'il existe une infinité de nombres premiers.

Corrigé

Question 0. On peut définir un programme P dans le langage de programme L qui affiche son entrée. Soit p le mot binaire représentant le programme P. Le mot $0^{|p|}1pw$ est une description de w dans le langage de programmation L. Donc $K_L(w) \leq |w| + 2|p|$. La longueur $2|p|$ est une constante ne dépendant pas du mot w.

Question 1. Il existe au plus $2^{n+1}-1$ mots binaires de longueur $\leq n$, donc il existe au plus $2^{n+1}-1$ mots w tels que $K_L(w) \leq n$. Comme il existe 2^{n+1} mots binaires de longueur $n+1$, il existe au moins un mot w tel que $K_L(w) > n$.

Question 2. Soient $d_0 = 0^{|p_0|}1p_0u_0$ et $d_1 = 0^{|p_1|}1p_1u_1$ des descriptions des mots w et u dans le langage de programmation L, tels que $|d_0| = K_L(w)$ et $|d_1| = K_L(u)$.

Soit P le programme qui prend en entrée des chaînes de la forme $0^{|d_0|}1d_0d_1$, extrait les mots d_0 et d_1, et les décode pour obtenir p_0, u_0, p_1 et u_1. Ensuite, le programme exécute p_0 sur l'entrée u_0 et p_1 sur l'entrée u_1, ce qui affiche wu. Soit p le mot binaire représentant le programme P. Le mot $d = 0^{|p|}1p0^{|d_0|}1d_0d_1$ est une description de w dans le langage L. Donc $K_L(wu) \leq |d| \leq 2|p| + 2|d_0| + |d_1| + 2 \leq 2K_L(w) + K_L(u) + 2|p| + 2$. La valeur $2|p| + 2$ est une constante ne dépendant pas des mots w et u.

On peut améliorer cette borne en $K_L(wx) \leq K_L(w) + K_L(x) + 2\log_2(K_L(w)) + \mathcal{O}(1)$. En effet, le codage des paires (d_0, d_1) est fait de manière non-optimale sous la forme $0^{|d_0|}1d_0d_1$. On pourrait stocker en binaire le nombre $n = |d_0|$ en doublant tous les digits, suivi de 01, puis de d_0 et d_1. Pour décoder cette paire, il suffit de trouver la première occurrence de 01, en déduire $|d_0|$ puis extraire d_0 et d_1.

Question 3. Le langage L a été choisi parmi Caml, C, Java, Python, qui sont tous des langages *universels*, c'est à dire que pour tout autre langage U, on peut interpréter le langage U dans L. On définit donc un *interpréteur* I dans le langage L comme suit : Le programme I prend en entrée un mot binaire de la forme $0^{|p|}1pu$ où p et u sont des mots binaires, calcule le nombre $|p|$, utilise cette longueur pour savoir où couper l'entrée et extraire les mots p et u, puis décode le programme P correspondant au mot p, et l'exécute sur u. Ce programme I admet une représentation sous forme d'un mot binaire $v \in \Sigma^*$.

Montrons que pour tout mot w, $K_L(w) \leq K_U(w) + 2|v|$. Soit d une description de w dans le langage U telle que $K_U(w) = |d|$. Le mot $0^{|v|}1vd$ est une description de w dans le langage L, donc $K_L(w) \leq 2|v| + |d| + 1 \leq K_U(w) + 2|v| + 1$. La constante $2|v| + 1$ ne dépend pas de w.

Question 4. Soit $d = 0^{|p|}1pu$ une description de w dans le langage L, telle que $|d| = K_L(w)$. Soit Q le programme qui prend en entrée d, qui extrait les mots p et u, puis exécute le programme P correspondant à p sur l'entrée u, récupère le mot w, et applique la fonction f sur w et affiche $f(w)$. Soit v un mot correspondant au programme Q. Le mot $0^{|v|}vd$ est une description du mot $f(w)$ dans le langage L. Ainsi, $K_L(f(w)) \leq 2|v| + d + 1 = K_L(w) + 2|v| + 1$. Le nombre $2|v| + 1$ est une constante qui ne dépend pas de w.

Question 5. Notons $\bar{n}$ la représentation binaire de n. Par la question 4, $K_L(w_n) \leq K_L(\bar{n}) + \mathcal{O}(1)$. Par la question 0, $K_L(\bar{n}) \leq |\bar{n}| + \mathcal{O}(1)$. Comme $|\bar{n}| \leq \log_2(n) + 1$, on obtient $K_L(w_n) \leq \log_2(n) + \mathcal{O}(1)$.

Question 6. On peut utiliser le pseudocode suivant :

```
Entrée n
n := n+1
mot := ""
Tant que n >= 0:
 n >> 2
 Si n mod 2 = 1:
        Si n != 1:
     mot := "1" + mot
        Fin si
 Sinon:
```

```
    mot := "0" + mot
       Fin si
  Fin Tant que
  Renvoyer mot
```

Question 7. Raisonnons par l'absurde. Supposons que la fonction K est calculable. Alors la fonction f qui pour une entrée $n \in \mathbb{N}$, cherche un mot w_n tel que $K_L(w_n) > n$ est aussi une fonction calculable. En effet, par la question 6, on peut énumérer tous les mots binaires, et tester si $K_L(w) > n$. Un tel mot w_n existe par la question 1, donc le test sera vrai. Par la question 4, $K_L(f(n)) \leq K_L(n) + \mathcal{O}(1)$, et $K_L(n) \leq \log_2(n) + \mathcal{O}(1)$. Ainsi $K_L(f(n)) \leq \log_2(n) + \mathcal{O}(1)$ pour tout $n \in \mathbb{N}$. Cependant, par définition de f, $K_L(f(n)) > n$, contradiction.

Question 8. Raisonnons par l'absurde. Supposons qu'il existe un nombre fini de nombres premiers $p_0, \ldots, p_{k-1}$. Par l'unicité de la décomposition en nombre premiers, tout nombre n peut être écrit de la forme $p_0^{x_0} \cdot p_1^{x_1} \cdots p_{k-1}^{x_{k-1}}$, avec $x_i < \log_2(n)$ pour tout $i < k$. Ainsi, le nombre n peut être décrit avec k nombres inférieurs à $\log_2(n)$, chacun écrit en binaire prenant la place $\log_2(\log_2(n))$, donc on obtient $K_L(n) \leq k \log_2(\log_2(n)) + \mathcal{O}(1)$. Les mots binaires étant en bijection avec les entiers (par leur représentation binaire), en particulier pour tout mot binaire w, $K_L(w) \leq k \log_2(w) + \mathcal{O}(1)$. Cela contredit l'existence (question 2) pour tout n d'un mot w de longueur n tel que $K_L(w) > n + \mathcal{O}(1)$.

L4 – Syndéticité par parties

Soit $F \subseteq \mathbb{N}$ et $b \in \mathbb{N}$. On note $F+b$ l'ensemble $\{a+b : a \in F\}$ et $F-b$ l'ensemble $\{a \in \mathbb{N} : a+b \in F\}$. Un ensemble $S \subseteq \mathbb{N}$ est *syndétique* s'il existe un ensemble fini $F \subseteq \mathbb{N}$ tel que $\mathbb{N} = \bigcup_{a \in F} S - a$.

Question 0. Montrer qu'un ensemble S est syndétique si et seulement s'il existe un entier $d \in \mathbb{N}$ tel que pour tout $k \in \mathbb{N}$, $\{k, k+1, \ldots, k+d-1\} \cap S \neq \emptyset$. On dit aussi que S est *d-syndétique*.

Question 1. Parmi les ensembles suivants, lesquels sont syndétiques ?

(1) L'ensemble des entiers naturels
(2) L'ensemble des nombres pairs
(3) L'ensemble des nombres premiers
(4) $\{3n+5 : n \in \mathbb{N}\}$

Un ensemble $T \subseteq \mathbb{N}$ est *épais* si pour tout ensemble fini $F \subseteq \mathbb{N}$, il existe un $n \in \mathbb{N}$ tel que $F + n \subseteq T$.

Question 2. Montrer qu'un ensemble T est épais si et seulement si pour tout $k \in \mathbb{N}$, il existe un $n \in \mathbb{N}$ tel que $\{n, n+1, n+2, \ldots, n+k-1\} \subseteq T$.

Question 3. Parmi les ensembles suivants, lesquels sont épais ?

(1) L'ensemble des entiers naturels
(2) L'ensemble des nombres pairs
(3) L'ensemble des nombres premiers
(4) $\{2^n + m : n \in \mathbb{N}, m \in \{0, \ldots, n\}\}$

Question 4. Pour toute 2-partition $A_0 \sqcup A_1 = \mathbb{N}$, existe-t-il toujours une partie épaisse ? syndétique ?

Un ensemble $S \subseteq \mathbb{N}$ est *syndétique par parties* s'il est l'intersection d'un ensemble syndétique U et d'un ensemble épais T. Si U est d-syndétique, alors S est dit *d-syndétique par parties*.

Question 5. Montrer que pour toute 2-partition $A_0 \sqcup A_1 = \mathbb{N}$, l'un au moins de A_0 et A_1 est syndétique par parties.

Soit $d \in \mathbb{N}$. Un ensemble fini $F = \{n_0 < n_1 < \cdots < n_{k-1}\}$ est *localement d-syndétique* si pour tout $0 \leq i < k-1$, on a $n_{i+1} - n_i < d$.

Question 6. Écrire le pseudocode d'un algorithme qui prend en entrée un ensemble fini S et un entier $d \in \mathbb{N}$, et détermine si S est localement d-syndétique. L'ensemble S est donné comme un tableau T de booléens où $T[i]$ indique si i appartient à S. Discuter de sa complexité en temps et en espace.

Question 7. Soit $d \in \mathbb{N}$. Montrer que si un ensemble S est d-syndétique par parties alors pour tout $k \in \mathbb{N}$, il existe un ensemble localement d-syndétique F de cardinalité k tel que $F \subseteq S$.

Question 8. Soit $d \in \mathbb{N}$ et $S \subseteq \mathbb{N}$. Montrer que, si pour tout $k \in \mathbb{N}$, il existe un ensemble localement d-syndétique F de cardinalité k tel que $F \subseteq S$, alors S est d-syndétique par parties. Commenter.

Suite des questions

Soit une famille $\mathscr{S}$ d'ensembles d'entiers naturels close par le haut (si $S \in \mathscr{S}$ et $T \supseteq S$, alors $T \in \mathscr{S}$). La famille *duale* de $\mathscr{S}$ est la famille $\mathscr{T} = \{T \subseteq \mathbb{N} : \forall S \in \mathscr{S}[S \cap T \neq \emptyset]\}$.

Question 9. Montrer que la famille des ensembles épais est la famille duale des ensembles syndétiques.

Question 10. Soit une famille $\mathscr{S}$ d'ensembles d'entiers naturels close par le haut. Soit $\mathscr{T}$ sa famille duale, et soit $\mathscr{A} = \{S \cap T : S \in \mathscr{S}, T \in \mathscr{T}\}$ la famille intersection. Montrer que si $A \in \mathscr{A}$, et $X_0 \sqcup X_1 = A$, alors soit $X_0 \in \mathscr{A}$, soit $X_1 \in \mathscr{A}$.

Question 11. En déduire que pour toute k-partition $A_0 \sqcup \cdots \sqcup A_{k-1} = \mathbb{N}$, l'un au moins de $A_0, \ldots, A_{k-1}$ est syndétique par parties.

Corrigé

Question 0. Supposons que S est syndétique. Soit $F \subseteq \mathbb{N}$ un ensemble fini tel que $\mathbb{N} = \bigcup_{a \in F} S - a$, et soit $d = \max F + 1$. Nous allons montrer que S est d-syndétique. Soit $k \in \mathbb{N}$. Comme $\mathbb{N} = \bigcup_{a \in F} S - a$, il existe un $a \in F$ tel que $k + a \in S$. Comme $a < d$, $\{k, k+1, \ldots, k+d-1\} \cap S \neq \emptyset$.

Supposons maintenant que S est d-syndétique. Soit $F = \{0, \ldots, d-1\}$. Montrons que $\mathbb{N} = \bigcup_{a \in F} S - a$. Soit $n \in \mathbb{N}$. Comme S est d-syndétique, il existe un $a \leq d$ tel que $n + a \in S$. Ainsi, il existe un $a \in F$ tel que $n = S - a$, donc $n \in \bigcup_{a \in F} S - a$.

Question 1. On utilisera la caractérisation de la question 0 qui est plus intuitive. (1) L'ensemble des entiers naturels est 0-syndétique, donc est syndétique. (2) L'ensemble des nombres pairs est 1-syndétique, donc syndétique. (3) L'ensemble des nombres premiers n'est pas syndétique. En effet, pour tout $d \in \mathbb{N}$, la séquence $(d+1)! + 2, (d+1)! + 3, \ldots, (d+1)! + d + 1$ ne contient aucun nombre premier, montrant ainsi que l'ensemble des nombres premiers n'est pas d-syndétique. (4) L'ensemble $S = \{3n + 5 : n \in \mathbb{N}\}$ est 5-syndétique (il est presque 3-syndétique, mais le $+5$ oblige à agrandir le coefficient de syndéticité.

Question 2. Soit T un ensemble épais et $k \in \mathbb{N}$. En prenant $F = \{0, 1, \ldots, k-1\}$, il existe un $n \in \mathbb{N}$ tel que $F + n \subseteq T$. Comme $F + n = \{n, n+1, \ldots, n+k-1\}$, la propriété est prouvée.

Soit T un ensemble tel que pour tout $k \in \mathbb{N}$, il existe un $n \in \mathbb{N}$ tel que $\{n, n+1, n+2, \ldots, n+k-1\} \subseteq T$. Soit $F \subseteq \mathbb{N}$ un ensemble fini. En prenant $k = \max F + 1$, il existe un n tel que $F + n \subseteq \{n, n+1, n+2, \ldots, n+k-1\} \subseteq T$, donc T est épais.

Question 3. (1) L'ensemble des entiers naturels est épais car pour tout ensemble fini $F \subseteq \mathbb{N}$, $F+0 \subseteq \mathbb{N}$. (2) L'ensemble des nombres pairs n'est pas épais, car pour l'ensemble $F = \{0, 1\}$, il n'existe pas de n tel que $F + n$ ne contient que des nombres pairs. (3) L'ensemble des nombres premiers n'est pas épais, car pour $F = \{3, 4\}$, il n'existe pas de n tel que $F + n$ ne contient que des nombres premiers. (4) L'ensemble $S = \{2^n + m : n \in \mathbb{N}, m \in \{0, \ldots, n\}\}$ est épais, car pour tout ensemble fini $F \subseteq \mathbb{N}$, soit $n = \max F$. Alors $F + 2^n \subseteq \{2^n, 2^n + 1, 2^n + 2, \ldots, 2^n + n\} \subseteq S$.

Question 4. Soit $A_0 = \{0\} \cup \{n : (\exists s \in \mathbb{N}) 2^{2s} \leq n < 2^{2s+1}\}$ et $A_1 = \{n : (\exists s \in \mathbb{N}) 2^{2s+1} \leq n < 2^{2s+2}\}$. Alors $A_0 \sqcup A_1 = \mathbb{N}$, mais ni A_0 ni A_1 ne sont syndétiques. En effet, A_0 et A_1 contiennent des "trous" arbitrairement grands.

Soit A_0 l'ensemble des nombres pairs et A_1 l'ensemble des nombres impairs. Alors $A_0 \sqcup A_1 = \mathbb{N}$, mais ni A_0, ni A_1 ne sont épais, comme montré par $F = \{0, 1\}$.

***Poser à l'oral :* Y a-t-il toujours une partie épaisse ou syndétique ?** Réponse : oui. Indication : si S n'est pas syndétique alors $\mathbb{N} \setminus S$ est épaisse. Preuve intuitive : S a des trous arbitrairement grands donc $\mathbb{N} \setminus S$ contient des segments arbitrairement longs.

Preuve formelle. Supposons que S n'est pas syndétique. Ainsi, par la caractérisation de la question 0, pour tout entier $d \in \mathbb{N}$, il existe $k \in \mathbb{N}$ tel que $\{k, \dots, k+d-1\} \cap S = \emptyset$. De façon équivalente, $\{k, \dots, k+d-1\} \subseteq \mathbb{N} \setminus S$. D'après la caractérisation de la question 2, ceci montre que $\mathbb{N} \setminus S$ est syndétique.

Question 5. Il faut d'abord remarquer que, comme $\mathbb{N}$ est épais est syndétique, alors tout ensemble épais ou syndétique est syndétique par parties.

Ensuite, c'est juste une application directe de la partie orale de la question 4 : l'un de A_0, A_1 est épaisse ou syndétique, donc syndétique par parties.

Question 6. On peut utiliser le pseudocode suivant :

```
Entrée T, d
entered := false
Pour i de 0 à |T| exclu:
  Si T[i] and not entered:
    entered := true
    precedent := i
  Sinon:
    Si T[i]:
      Si i - precedent > d:
        Renvoyer Faux
      Fin Si
      precedent := i
    Fin Si
  Fin Si
Fin Pour
Renvoyer Vrai
```

La complexité en temps est manifestement linéaire en le tableau d'entrée, et la complexité en mémoire est constante.

Autre algorithme possible : construire une liste ordonnée puis faire le max des différences.

Question 7. Soit S un ensemble d-syndétique par parties. Il existe donc un ensemble épais T et un ensemble d-syndétique U tel que $S = T \cap U$. Soit $k \in \mathbb{N}$. Comme T est épais, il existe un $n \in \mathbb{N}$ tel que $\{n, n+1, \dots, n+dk\} \subseteq T$. Comme U est d-syndétique, il existe un ensemble localement d-syndétique $F \subseteq U \cap \{n, n+1, \dots, n+dk\}$ de cardinalité k. Comme $U \cap \{n, n+1, \dots, n+dk\} \subseteq U \cap T$, on a bien $F \subseteq S$.

Question 8. Soit S un ensemble tel que pour tout $k \in \mathbb{N}$, il existe un ensemble localement d-syndétique de k éléments $F \subseteq S$. Montrons que $S = T \cap U$ pour un ensemble épais T et un ensemble d-syndétique U. Soit $F_0, F_1, \dots$ une séquence infinie de sous-ensembles localement d-syndétiques de S qui ne contiennent pas de trous plus grands que d et tels que $\max F_i < \min F_{i+1}$ et $|F_i| = i+1$: ceci s'obtient par application itérée de la condition avec des k suffisamment grands relativement à l'ensemble précédent (et en retirant un nombre petit de valeurs trop petites si nécessaire). Soit $U = \bigcup_i F_i \bigcup_i \{n : \max F_i \leq n \leq \min F_{i+1}\} \cup \{n : n \leq F_0\}$. L'ensemble U est d-syndétique, car les seuls espaces sont au sein d'un F_i. Soit $T_0 = \bigcup_i \{n : \min F_i \leq n \leq \max F_i\}$ et $T = T_0 \cup \bigcup_i \{n \in S : \max F_i \leq n \leq \min F_{i+1}\}$. L'ensemble T_0 est épais, et donc l'ensemble T l'est également. En outre, $T \cap U = S$.

Commentaire attendu : c'est la réciproque du résultat de la question 7, qui est donc une caractérisation.

Question 9. Soit T un ensemble épais. Montrons que pour tout ensemble syndétique S, $S \cap T \neq \emptyset$. Soit $d \in \mathbb{N}$ tel que S est d-syndétique et soit $F = \{0, 1, \ldots, d\}$. Comme T est épais, il existe un $n \in \mathbb{N}$ tel que $F + n \subseteq T$. Comme S est d-syndétique, $S \cap (F + n) \neq \emptyset$, donc $S \cap T \neq \emptyset$.

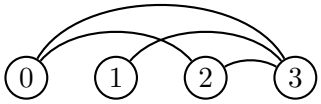
Soit T un ensemble tel que pour tout ensemble syndétique S, $S \cap T \neq \emptyset$. Supposons que T n'est pas épais, c'est-à-dire qu'il existe un $F \subseteq \mathbb{N}$ tel que pour tout $n \in \mathbb{N}$, $F + n \not\subseteq T$. Il s'ensuit que $\overline{T}$ est syndétique, mais $T \cap \overline{T} = \emptyset$. Contradiction.

Question 10. Soit $A = S \cap T$ pour un ensemble $S \in \mathscr{S}$ et $T \in \mathscr{T}$. Soit $X_0 \sqcup X_1 = A$. Soit $S' = X_0 \cup (S \setminus A)$. On peut montrer que $X_0 = S' \cap T$. Ainsi, si $S' \in \mathscr{S}$, alors $X_0 \in \mathscr{A}$ et la preuve est terminée. Si $S' \notin \mathscr{S}$, alors soit $T' = \mathbb{N} \setminus S'$. On a $T' \in \mathscr{T}$, en effet par l'absurde si on avait un $S'' \in \mathscr{S}$ qui n'intersecte pas T' alors S'' serait inclus dans le complémentaire de T' c'est-à-dire dans S', contradiction car $S' \notin \mathscr{S}$ et $\mathscr{S}$ est close vers le haut. Comme $X_1 = T' \cap S$, on a $X_1 \in \mathscr{A}$.

Question 11. Soit $\mathscr{S}$ la famille des ensembles syndétiques. Par la question 9, la famille $\mathscr{T}$ des ensembles épais est la famille duale de $\mathscr{S}$. Il s'ensuit que la famille intersection $\mathscr{A} = \{S \cap T : S \in \mathscr{S}, T \in \mathscr{T}\}$ est la famille des ensembles syndétiques par partie. La question 10 montre que si S est syndétique par parties, et $S_0 \sqcup S_1 = S$, alors soit S_0 soit S_1 est syndétique par parties. Sachant que $\mathbb{N}$ est syndétique par parties, par une induction immédiate, Par une induction immédiate, toute k-partition $A_0 \sqcup \cdots \sqcup A_{k-1} = \mathbb{N}$ admet une partie syndétique par parties.

L5 – Graphes de saut

Un *graphe de saut* de longueur $n \geq 1$ est un graphe non-orienté dont les sommets sont $0, 1, \ldots, n-1$, tel que pour tout $a \leq b < c \leq d < n$, s'il existe une arête entre b et c, alors il en existe une entre a et d. On peut interpréter l'existence d'une arête entre a et b comme disant "le saut de a à b est grand". Voici un exemple de graphe de saut de longueur 4 :



Question 0. Dessiner les graphes de saut de longueur 1, 2 et 3.

Question 1. Écrire un algorithme prenant en entrée la matrice d'adjacence d'un graphe et détermine si ce graphe est un graphe de saut. Quelle est sa complexité en temps ? en espace ?

Un graphe de saut de longueur n est *comprimé* si pour tout $i < n-1$, les sommets i et $i+1$ ne sont pas reliés.

Question 2. Montrer que les graphes de saut comprimés de longueur $n+1$ sont en bijection avec les graphes de saut de longueur n.

Question 3. Soit P_n le nombre de graphes de saut de longueur n. On pose par convention $P_0 = 1$. Déterminer la formule de récurrence de P_{n+1} en fonction de $P_0, \ldots, P_n$.

Le but des questions suivantes est de tester s'il existe une permutation des sommets transformant un graphe quelconque en un graphe de saut.

Question 4. Écrire un programme prenant en entrée une liste L de longueur n représentant une permutation des entiers de 0 à $n-1$ inclus, et retourne une liste L' représentant la prochaine permutation suivant un certain ordre total $\prec$ sur les permutations. Si L est la dernière permutation de cet ordre, alors le programme retournera la liste vide.

Question 5. Écrire un algorithme naïf prenant en entrée un entier $n \geq 0$ et une matrice d'adjacence M de taille $n \times n$, et décide s'il existe une permutation sur $\{0, \ldots, n-1\}$ tel que le graphe résultant est un graphe de saut. Quelle est sa complexité en temps ? en mémoire ?

Question 6. Montrer que pour tout graphe $G = (\{0, 1, 2\}, E)$, il existe une permutation de $\{0, 1, 2\}$ tel que le graphe résultant est un graphe de saut.

Question 7. Donner un graphe $G = (\{0, 1, 2, 3\}, E)$ dont aucune permutation ne résulte en un graphe de saut.

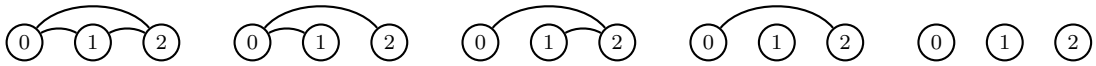
Suite des questions

Question 8. Que peut-on dire sur l'existence de graphes non connexes dont les sommets sont permutables pour obtenir un graphe de saut ?

Corrigé

Question 0. Les graphes de saut de longueur 1 et 2 :

Les graphes de saut de longueur 3 sont



Question 1.

```
Entrée n, M
Pour x de 1 à n exclus
  Pour y de x à n exclus
    Si M[x][y] == 1:
      Si (x > 0 ET M[x-1][y] == 0) OU (y < n-1 ET M[x][y+1] == 0):
        Renvoyer Non
      Fin Si
    Fin Si
  Fin Pour
Fin Pour
Renvoyer Oui
```

La complexité en temps est $\mathcal{O}(n^2)$ et en espace est de $\mathcal{O}(1)$.

Question 2. Soit h la fonction qui transforme un graphe de saut comprimé $G_0 = (\{0, 1, \ldots, n\}, E_0)$ de longueur $n+1$ en un graphe de saut $G_1 = (\{0, 1, \ldots, n-1\}, E_1)$ de longueur n comme suit :

$$E_1 = \{\{a, b-1\} : \{a, b\} \in E_0, a < b\}$$

Noter que $\{a, b-1\}$ n'est jamais un singleton car G_0 est un graphe de saut comprimé. Montrons que G_1 est un graphe de saut. Si $\{b, c-1\} \in E_1$ et $a \leq b$ et $c-1 \leq d-1$, alors par définition de E_1, $\{b, c\} \in E_0$, et comme G_0 est un graphe de saut, $\{a, d\} \in G_0$, donc $\{a, d-1\} \in E_1$.

La fonction h est clairement injective. Montrons que h est surjective. Soit $G_1 = (\{0, 1, \ldots, n-1\}, E_1)$ un graphe de saut de longueur n. Soit $G_0 = (\{0, 1, \ldots, n\}, E_0)$ le graphe de saut comprimé défini par

$$E_0 = \{\{a, b\} : \{a, b-1\} \in E_1, a < b\}$$

Comme G_1 est un graphe de saut, si $\{a, b-1\} \in E_1$ avec $a < b$, comme $\{a, b-1\}$ n'est pas un singleton, $a < b-1$, donc E_0 est un graphe de saut comprimé. De plus, $h(G_0) = G_1$.

Question 3. Il existe un seul graphe de saut de longueur 0, à savoir le graphe vide. Ainsi $P_0 = 1$. Soit $G = (\{0, 1, \ldots, n\}, E)$ un graphe de saut de taille $n+1$. Soit $i \leq n$ le plus petit sommet tel que $\{i, i+1\} \in E$, ou bien $i = n$ si un tel sommet n'existe pas. Le graphe restreint aux sommets $\{0, \ldots, i\}$ est un graphe de saut comprimé de taille $i+1$, et le graphe restreint aux sommets $\{i+1, \ldots, n\}$ est un graphe de saut de taille $n-i$. Les arêtes entre des sommets de $\{0, \ldots, i\}$ et $\{i+1, \ldots, n\}$ existent toutes car $\{i, i+1\} \in E$. Par la question 2, il existe autant de graphes de saut comprimés de taille $i+1$ que de graphes de saut de taille i, donc par hypothèse d'induction, il existe $P_i P_{n-i}$ graphes de saut pour i fixé. La formule est donc $P_0 = 1$ et

$$P_{n+1} = \sum_{i=0}^{n} P_i P_{n-i}$$

Question 4. On définit l'ordre lexicographique sur les permutations de $\{0, \ldots, n-1\}$ comme suit : Soient $L \neq L'$ deux permutations sur $\{0, \ldots, n-1\}$. Soit k le plus petit indice tel que $L[k] \neq L'[k]$. $L \prec L'$ si $L[k] < L'[k]$. Par exemple :

0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
1 0 2 3
...

Il existe un excellent post MathOverflow expliquant l'ordre lexicographique sur les permutations [1].
L'algorithme informel pour trouver la prochaine permutation de L est ainsi :

1. Si pour tout $k < n-1$, $L[k] > L[k+1]$, L est le dernier élément de l'ordre lexicographique, et l'on retourne la liste vide.
2. Soit k le plus grand élément tel que $L[k] < L[k+1]$. Ainsi, la liste de $L[k+1]$ à $L[n-1]$ est décroissante.
3. Soit $\ell > k$ le plus grand indice tel que $L[k] < L[\ell]$. Un tel indice existe car l'inégalité est satisfaite en prenant $\ell = k+1$.
4. Échanger $L[k]$ et $L[\ell]$.
5. Ordonner de manière décroissante la séquence de $L[k+1]$ à $L[n-1]$.

Question 5.

```
Entrée n, M
L := [0, ..., n-1]
Tant que L <> []
  Si c'est un graphe de saut en permutant les indices de M par L (question 1):
    Retourner oui
  Fin Si
  L := prochaine_permutation(L)
Fin Tant Que
Renvoyer Non
```

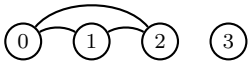
La complexité en temps est $\mathcal{O}(n! * n^2)$ et en espace est de $\mathcal{O}(n)$.

1. https://stackoverflow.com/questions/11483060/stdnext-permutation-implementation-explanation

Question 6. Soit un graphe $G = (\{0, 1, 2\}, E)$. Si G est déjà un graphe de saut, alors il suffit de prendre la permutation identité. Les graphes restants sont

Dans les deux premiers cas, intervertir 1 et 2 suffit. Dans le dernier cas, il suffit d'intervertir 0 et 1.

Question 7. Le seul graphe de taille 4 ne pouvant pas être transformé en un graphe de saut par une permutation est



Remarque : Si v_0 et v_1 sont liés par une arête, tandis que v_2 n'est lié ni à v_0 ni à v_1, alors tout permutation des sommets pour former un graphe de saut placera v_2 entre v_0 et v_1. Autrement dit, on aura $v_0 < v_2 < v_1$ ou $v_1 < v_2 < v_0$.

Preuve pour le graphe : Par la remarque précédente, 3 n'étant lié ni à 0, ni à 1 qui sont liés entre eux, il doit être placé entre 0 et 1. Par le même raisonnement, 3 doit être placé entre 1 et 2, et 3 doit être placé entre 0 et 2. Impossible.

Question 8. *Observation 1 :* Toutes les composantes connexes sauf au plus une une doivent être singleton. Par la remarque de la question 7, si v_0, v_1 sont deux sommets d'une composante connexe reliés par une arête, et v_2, v_3 sont deux autres sommets d'une seconde composante connexe reliés par une arête, alors v_0 et v_1 doivent tous les deux se situer entre v_2 et v_3, et v_2 et v_3 doivent tous les deux se situer entre v_0 et v_1. Impossible, donc il ne peut pas y avoir deux composantes connexes non triviales.

Observation 2 : Le graphe doit être biparti. Comme le graphe n'est pas connexe, par l'observation 1, il existe une composante connexe triviale, v. Par la remarque de la question 7, toute arête entre deux sommets v_0 et v_1, v doit se situer entre v_0 et v_1. Il s'ensuit que le graphe est biparti.