ÉCOLE NORMALE SUPÉRIEURE DE LYON

Concours d'admission session 2017

Filière universitaire : Second concours

COMPOSITION D'INFORMATIQUE

Durée : 3 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve.

⋆ ⋆ ⋆

1 Préliminaires

On s'intéresse à la représentation d'ensembles disjoints, c'est-à-dire de collections $\mathcal{S} = \{S_1, \ldots, S_k\}$, où $S_1, \ldots, S_k$ sont des ensembles disjoints finis. Ce sujet comporte trois parties qui ne sont pas indépendantes. Il est toutefois possible d'admettre le résultat de toute question pour répondre aux questions suivantes.

L'énoncé de ce sujet est écrit en utilisant PYTHON comme langage de référence. Cependant, lorsque du code est demandé, ce code peut être écrit en PYTHON, en pseudo-code ou dans un langage au choix du candidat, en utilisant les structures de contrôle habituelles. Toutes les réponses devront être justifiées.

Étant données deux fonctions $f, g : \mathbb{N}^k \to \mathbb{N}$, nous disons que f est en $O(g(n_1, \ldots, n_k))$ lorsqu'il existe des constantes $M, N_0, \ldots, N_k \in \mathbb{N}$ telles que $f(n_1, \ldots, n_k) \leq M \cdot g(n_1, \ldots, n_k)$ pour tous $n_1, \ldots, n_k$ avec $n_i \geq N_i$ pour tout $i = 1, \ldots, k$.

La complexité, ou le temps d'exécution, d'un programme `P` ou d'une séquence d'instructions $o_1, \ldots, o_m$, est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc.) nécessaires à l'exécution de `P` (resp. à l'exécution de la séquence d'instructions $o_1, \ldots, o_m$). Cette complexité peut dépendre de paramètres $n_1, \ldots, n_k$ et donc être vue comme une fonction $f : \mathbb{N}^k \to \mathbb{N}$. Dans ce cas nous dirons que `P` (resp. $o_1, \ldots, o_m$) a une complexité en $O(g(n_1, \ldots, n_k))$ lorsque f est en $O(g(n_1, \ldots, n_k))$.

2 Ensembles disjoints par forêts

Un **noeud** est une liste PYTHON `x` à au moins deux éléments. L'élément `x`[0] contient la valeur du noeud, et l'élément `x`[1] désigne la liste représentant le parent du noeud `x`. On dit que `x` est une **racine** si `x`[1] désigne `x`. La fonction suivante renvoie `True` si le noeud `x` est une racine et `False` sinon.

```
def est_racine(x) : return (x[1] == x)
```

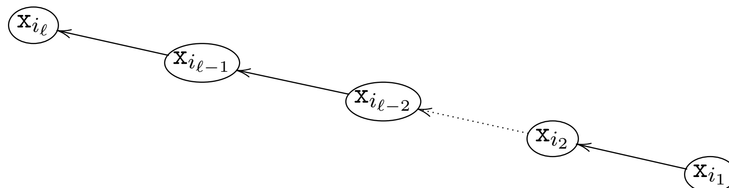
D'autre part, la fonction suivante crée et renvoie un nouveau noeud racine.

```
def cree_racine () :
   x = [None,None]
   x[1] = x
   return x
```

Considérons n noeuds $\mathtt{x}_1, \ldots, \mathtt{x}_n$ tels que chaque $\mathtt{x}_i$ a pour parent un $\mathtt{x}_j$, c'est-à-dire tels que pour tout $1 \leq i \leq n$, il existe un $1 \leq j \leq n$ avec $\mathtt{x}_i[1] = \mathtt{x}_j$. On dit qu'un noeud $\mathtt{x}_j$ est un **ascendant** d'un noeud $\mathtt{x}_i$ (et reciproquement que $\mathtt{x}_i$ est un **descendant** d'un noeud $\mathtt{x}_j$), notation $\mathtt{x}_i \preceq \mathtt{x}_j$, si on peut atteindre $\mathtt{x}_j$ à partir de $\mathtt{x}_i$ en suivant les parents, c'est-à-dire s'il existe une suite de noeuds $\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_\ell}$ telle que

$$\mathtt{x}_i = \mathtt{x}_{i_1} \quad \text{et} \quad \mathtt{x}_{i_1}[1] = \mathtt{x}_{i_2} \quad \text{et} \quad \ldots \quad \text{et} \quad \mathtt{x}_{i_{\ell-2}}[1] = \mathtt{x}_{i_{\ell-1}} \quad \text{et} \quad \mathtt{x}_{i_{\ell-1}}[1] = \mathtt{x}_{i_\ell} = \mathtt{x}_j \qquad (1)$$

ce que l'on représente de la manière suivante :



Remarquons tout d'abord que l'ensemble des ascendants d'un noeud donné est linéairement ordoné par $\preceq$.

Question 2.1 *Justifier brièvement l'assertion suivante :*

— *Si* $\mathtt{x}_i \preceq \mathtt{x}_j$ *et* $\mathtt{x}_i \preceq \mathtt{x}_\ell$, *alors soit* $\mathtt{x}_j \preceq \mathtt{x}_\ell$, *soit* $\mathtt{x}_\ell \preceq \mathtt{x}_j$.

On suppose maintenant que la relation $\preceq$ est **acyclique**, au sens où $\mathtt{x}_i = \mathtt{x}_j$ dès lors que $\mathtt{x}_i \preceq \mathtt{x}_j \preceq \mathtt{x}_i$. Dans ce cas, la relation $\preceq$ définit une **forêt**, c'est-à-dire un ensemble d'arbres.

Question 2.2 *Justifier brièvement l'assertion suivante :*

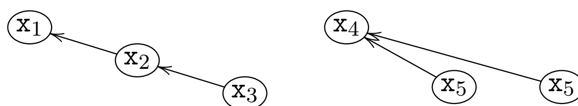
— *En supposant que* $\preceq$ *est acyclique, tout noeud a un ascendant qui est une racine.*

Un **arbre** de $\preceq$ est par définition l'ensemble des descendants d'un noeud racine. Ainsi, chaque noeud $\mathtt{x}_i$ appartient à un unique arbre de $\preceq$, et les noeuds $\mathtt{x}_1, \ldots, \mathtt{x}_n$ décrivent une collection d'arbres disjoints $\mathcal{S} = \{S_1, \ldots, S_k\}$, c'est-à-dire une collection d'ensemble disjoints.

Exemple. Considérons des noeuds $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_6$ avec

$$\mathtt{x}_1[1] = \mathtt{x}_1, \quad \mathtt{x}_2[1] = \mathtt{x}_1, \quad \mathtt{x}_3[1] = \mathtt{x}_2 \qquad \text{et} \qquad \mathtt{x}_4[1] = \mathtt{x}_5[1] = \mathtt{x}_6[1] = \mathtt{x}_4$$

Ces noeuds représentent la forêt



(dont les racines sont $\mathtt{x}_1$ et $\mathtt{x}_4$) et la collection $\{\{\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3\}, \{\mathtt{x}_4, \mathtt{x}_5, \mathtt{x}_6\}\}$.

Recherche de représentant. Supposons donnés n noeuds $\mathtt{x}_1, \ldots, \mathtt{x}_n$, engendrant une relation $\preceq$ acyclique, et décrivant la collection d'ensembles disjoints $\mathcal{S} = \{S_1, \ldots, S_k\}$. En d'autres termes, $\mathcal{S}$ est la collection des arbres de $\preceq$. Par définition, le **représentant** de l'ensemble S_i est le noeud racine de l'arbre correspondant. Le **représentant** d'un noeud `x` est le représentant de l'unique S_i auquel `x` appartient.

Question 2.3 *Écrire une fonction* `repr` *qui prend en argument un noeud et qui renvoie son représentant. Donner une borne de complexité de cette fonction en fonction de* n.

Il est très simple de savoir si deux noeuds quelquonques appartiennent au même ensemble.

Question 2.4 *Écrire une fonction qui prend arguments deux noeuds, et qui renvoie* `True` *si ces deux noeuds appartiennent au même ensemble et* `False` *sinon.*

Union d'ensembles. Nous allons maintenant implémenter une fonction `union` qui permet de réaliser l'union de deux ensembles **disjoints** $S, T \in \mathcal{S}$. Cette opération va en fait être implémentée directement au niveau des noeuds et de leurs représentants. La fonction `union` prend en arguments deux noeuds `x` et `y` **dont les représentants sont supposés distincts**. Si $\mathtt{x} \in S$ et $\mathtt{y} \in T$, alors l'exécution de `union(x, y)` modifie les représentants respectifs `px` et `py` de `x` et `y` de telle sorte que `px` devienne le parent de `py`, ou bien que `py` devienne le parent de `px`. Dans le contexte de ce sujet, il est pratique d'implémenter la fonction `union` en utilisant une fonction auxillière `relie`, et d'écrire `union` de la manière suivante :

```
def union(x,y) : relie(repr(x), repr(y))
```

Question 2.5 *Écrire une fonction* `relie`*, qui prend en arguments deux noeuds* `x` *et* `y` *et qui modifie le parent de* `y` *pour qu'il devienne* `x`*. La complexité de cette fonction doit être en* $O(1)$*.*

Coût d'une séquence d'appels à `cree_racine`, `union` et `repr`. Considérons une séquence de m instructions $o_1, \ldots, o_m$, dans laquelle chaque o_i consiste en l'appel à une des fonctions `cree_racine`, `repr` ou `union`. On suppose que cette séquence débute par $n \leq m$ appels à `cree_racine`, de sorte à créer n noeuds racines $\mathtt{x}_1, \ldots, \mathtt{x}_n$, et que les instructions $o_{n+1}, \ldots, o_m$ consistent uniquement en des appels à `repr` ou `union` sur des noeuds parmi $\mathtt{x}_1, \ldots, \mathtt{x}_n$. On suppose en outre que tous les appels à `union` sont faits sur des noeuds de représentants distincts.

Question 2.6 *Montrer que dans une séquence d'instructions* $o_1, \ldots, o_m$ *comme ci-dessus, il y a au plus* $n - 1$ *appels à* `union`*.*

Question 2.7 *Montrer que la complexité d'une séquence d'instructions* $o_1, \ldots, o_m$ *comme ci-dessus est en* $O(m \cdot n)$*.*

L'objet de la question suivante est de montrer que la borne asymptotique de la Question 2.7 peut être atteinte.

Question 2.8 *Donner une suite de séquences d'instructions* $(o_{k,1}, \ldots, o_{k,m_k})_{k \geq 0}$ *comme ci-dessus, et qui satisfait les deux propriétés suivantes :*

- *La suite* $(m_k)_{k \geq 0}$ *est croissante.*
- *Soit* n_k *le nombre d'appels à la fonction* `cree_racine` *dans la séquence d'instructions* $o_{k,1}, \ldots, o_{k,m_k}$*. Alors il existe des constantes* M, K_0 *avec* $K_0 \in \mathbb{N}$ *et* $M \in \mathbb{Q}$, $M > 0$, *et telles que la complexité de* $o_{k,1}, \ldots, o_{k,m_k}$ *est supérieure ou égale à* $M \cdot n_k \cdot m_k$ *pour tout* $k \geq K_0$*.*

Compression de chemins. Nous nous intéressons maintenant à une heuristique pour la recherche de représentant, appellée **compression de chemins**, et qui permet, comme nous allons le voir par la suite, d'améliorer la complexité « amortie » de la recherche de représentant dans une suite de m opérations $o_1, \ldots, o_m$ comme ci-dessus.

La compression de chemin est une opération réalisée par une fonction de recherche de représentant, et qui permet, pour deux appels successifs à cette fonction sur le même noeud, d'effectuer le second appel en temps constant (par rapport à m). Le principe est qu'un appel à `repr` sur un noeud `x` de représentant `y` modifie les parents de tous les acendants de `x` (y compris le parent de `x` lui-même) de telle sorte que chacun de ces parents soit `y`.

Question 2.9 *Donner une nouvelle implémentation de la fonction* `repr`*, qui prend en argument un noeud, qui renvoie le représentant de ce noeud, et qui implémente l'heuristique de compression de chemins. La complexité de* `repr(x)` *doit être en* $O(s)$ *où* s *est tel que si* $\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_\ell}$ *satisfont (1), sont deux à deux distincts, et sont tels que* $\mathtt{x} = \mathtt{x}_{i_1}$ *et* $\mathtt{x}_{i_\ell}$ *est le représentant de* `x`*, alors* $s = \ell$*.*

3 Arbres avec noeuds classés

Nous avons au §2 proposé une structure de données pour représenter des ensembles disjoints sous forme d'arbres. Cette structure de donnée était en particulier équippée d'une fonction `union` réalisant l'union de deux ensembles, implémentée à l'aide de la fonction auxillière `relie` de la Question 2.5. Un appel à `relie(x, y)` modifie `y` de sorte que `x` devienne le représentant de `y`. Nous allons maintenant voir une heuristique qui permet d'améliorer la compléxité d'une séquence d'instructions $o_1, \ldots, o_m$, et selon laquelle `relie(x, y)` choisi pour représentant parmi `x` et `y` celui qui est la racine de l'arbre le plus haut.

Noeuds classés. Un **noeud classé** est une liste PYTHON `x` avec au moins **trois** éléments. L'élément `x[0]` contient la valeur du noeud, l'élément `x[1]` désigne le parent du noeud `x`, et l'élément `x[2]` est entier postif ou nul, le **rang** de `x`. Les noeuds classés sont donc en particulier des noeuds au sens du §2. De ce fait les définitions de noeud racine et de $\preceq$ ne sont pas modifées. De plus `x[1][1]` désigne le parent du parent de `x` et `x[1][2]` désigne le rang du parent de `x`. La fonction `cree_racine` peut être adaptée aux noeuds classés de manière à créer un noeud de rang nul :

```
def cree_racine () :
   x = [None,None,0]
   x[1] = x
   return x
```

Question 3.1 *Réécrire la fonction* `relie` *de la manière suivante. Si* `x` *et* `y` *sont des racines de rangs strictement supérieurs aux rangs de leurs descendants, alors* `relie(x, y)` *choisi pour racine parmi* `x` *et* `y` *celui de plus grand rang, et met à jour le rang du noeud racine de sorte qu'il soit le plus petit entier strictement supérieur aux rangs de ses descendants dans le nouvel arbre, et supérieur ou égal aux rangs de* `x` *et* `y` *avant l'appel à* `relie(x, y)`. *La complexité de cette fonction doit être en* $O(1)$.

Coût d'une suite d'opérations. On suppose à partir de maintenant et pour toute la suite de cette partie que la fonction `cree_racine` est celle définie juste avant la Question 3.1, que la fonction `union` utilise la fonction `relie` de la Question 3.1 et que la fonction `repr` est celle de la Question 2.9.

Considérons une séquence d'instructions $o_1, \ldots, o_m$, dans laquelle chaque o_i consiste en l'appel à une des fonctions `cree_racine`, `repr` ou `union` sur des noeuds classés. On suppose en outre que la séquence $o_1, \ldots, o_n$ consiste en n appels à `cree_racine` et que tous les appels à `union` sont faits sur des noeuds de représentants distincts. L'objet de la suite de cette partie est de montrer que la complexité de $o_1, \ldots, o_m$ est presque linéaire en n et m. Nous allons voir qu'elle est en $O(m \cdot \alpha(n))$, où α est une fonction à croissance très lente (avec $\alpha(n) \leq 4$ pour toute application pratiquement concevable).

La fonction α. Les fonctions $(A_k)_{k\in\mathbb{N}}$, avec $A_k : \mathbb{N} \to \mathbb{N}$ sont définies de la manière suivante :

$$\begin{array}{rcl} A_0(j) & := & j+1 \\ A_{k+1}(j) & := & A_k^{(j+1)}(j) \end{array}$$

où, pour $f : \mathbb{N} \to \mathbb{N}$, on définit $f^{(i)} : \mathbb{N} \to \mathbb{N}$ par $f^{(0)}(j) := j$ et $f^{(i+1)}(j) := f(f^{(i)}(j))$.

Question 3.2

(i) *Montrer que pour tout $k \in \mathbb{N}$, la fonction A_k est strictement croissante, et que pour tout $j \in \mathbb{N}$, on a $j < A_k(j)$.*

(ii) *Montrer que si $k < \ell$, alors pour tout $j > 0$ on a $A_k(j) < A_\ell(j)$.*

(iii) *Montrer que pour tout $j \in \mathbb{N}$, il existe $k \geq 0$ tel que $j \leq A_k(1)$.*

La fonction $\alpha : \mathbb{N} \to \mathbb{N}$ est définie par

$$\alpha(n) \quad := \quad \min\{k \mid A_k(1) \geq n\}$$

On a donc $n \leq A_{\alpha(n)}(1)$.

Une borne temporelle amortie. Fixons maintenant et pour la suite de cette partie une séquence de m' instructions $o'_1, \ldots, o'_{m'}$, dans laquelle chaque o'_i consiste en l'appel à une des fonctions `cree_racine`, `repr` ou `union`. On suppose que cette séquence débute par $n \leq m$ appels à `cree_racine`, de sorte à créer n noeuds classés racines $\mathtt{x}_1, \ldots, \mathtt{x}_n$, que les instructions $o'_{n+1}, \ldots, o'_{m'}$ consistent uniquement en des appels à `repr` ou `union` sur des noeuds parmi $\mathtt{x}_1, \ldots, \mathtt{x}_n$, et que tous les appels à `union` sont faits sur des noeuds de représentants distincts. Nous allons voir que la complexité de $o'_{n+1}, \ldots, o'_{m'}$ est en $O(m' \cdot \alpha(n))$.

Afin de montrer ce résultat, nous décomposons chaque appel à `union` en deux appels à `repr` suivis d'un appel à `relie`. Ceci génère une séquence de m instructions $o_1, \ldots, o_m$, avec $m \leq 3m'$. Notons que les n-premières instructions $o_1, \ldots, o_n$ sont des appels à `cree_racine`, créant les noeuds racines $\mathtt{x}_1, \ldots, \mathtt{x}_n$, et que les instructions $o_{n+1}, \ldots, o_m$ consistent uniquement en des appels à `repr` et `relie`. De plus, si la complexité de $o_1, \ldots, o_m$ est en $O(m \cdot \alpha(n))$, alors la complexité de $o'_1, \ldots, o'_{m'}$ est en $O(m' \cdot \alpha(n))$.

On va maintenant s'intéresser uniquement à la séquence $o_{n+1}, \ldots, o_m$. Notons qu'aucun noeud n'est créé (ni détruit) dans cette séquence. De plus, avant l'instruction o_{n+1} le rang de chaque noeud est nul.

Question 3.3 *Soit $k \geq n$ et considérons le noeud $\mathtt{x}_i$ (avec $1 \leq i \leq n$) après l'instruction o_k.*

(i) *Montrer que $\mathtt{x}_i[2] \leq \mathtt{x}_i[1][2]$, et que $\mathtt{x}_i[2] = \mathtt{x}_i[1][2]$ si et seulement si $\mathtt{x}_i$ est une racine.*

(ii) *Soit r' le rang de $\mathtt{x}_i$ après l'instruction o_{k+1}. Montrer que $r' \geq \mathtt{x}_i[2]$, et que $r' = \mathtt{x}_i[2]$ si $\mathtt{x}_i$ n'est pas une racine.*

(iii) *Montrer que $\mathtt{x}_i[2] \leq n - 1$.*

Méthode des potentiels. À chaque noeud $\mathtt{x}_i$ pour $1 \leq i \leq n$, nous allons associer des **potentiels** $\Phi_n(\mathtt{x}_i), \Phi_{n+1}(\mathtt{x}_i), \ldots, \Phi_m(\mathtt{x}_i) \in \mathbb{N}$, avec $\Phi_n(\mathtt{x}_i) = 0$, et associer à chaque instruction o_k (pour $k = n+1, \ldots, m$) sa **complexité amortie**

$$\hat{c}_k \quad := \quad c_k + \sum_{i=1}^{n} (\Phi_k(\mathtt{x}_i) - \Phi_{k-1}(\mathtt{x}_i)) \qquad (2)$$

où c_k est la complexité de l'instruction o_k. Dans toute la suite de cette partie, c_k et $\hat{c}_k$ sont supposées être des fonctions de n et m. Les fonctions potentielles Φ_k vont être définies de sorte que le coût amorti $\hat{c}_k$ de l'instruction o_k soit en $O(\alpha(n))$.

Question 3.4 *Supposons que $\hat{c}_k$ est en $O(\alpha(n))$ et satisfait (2) pour tout $k = n+1, \ldots, m$, et que $\Phi_n(\mathtt{x}_i) = 0$ pour tout $i = 1, \ldots, n$. Montrer que la complexité de $o_{n+1}, \ldots, o_m$ est en $O(m \cdot \alpha(n))$.*

Comme d'autre part la complexité de o_k pour $k \leq n$ est en $O(1)$, il s'en suivra que la complexité de $o_1, \ldots, o_m$ est en $O(m \cdot \alpha(n))$.

Potentiel d'un noeud. Considérons un noeud $\mathtt{x}$ parmi $\mathtt{x}_1, \ldots, \mathtt{x}_n$, qui n'est pas une racine et qui a un rang supérieur ou égal à 1. On pose

$$\begin{array}{rcl} \mathrm{niv}(\mathtt{x}) & := & \max\{k \mid \mathtt{x}[1][2] \geq A_k(\mathtt{x}[2])\} \\ \mathrm{iter}(\mathtt{x}) & := & \max\{i \mid \mathtt{x}[1][2] \geq A^{(i)}_{\mathrm{niv}(\mathtt{x})}(\mathtt{x}[2])\} \end{array}$$

Pour chaque $k \geq n$, la fonction Φ_k est définie par

$$\Phi_k(\mathtt{x}) := \begin{cases} \alpha(n) \cdot \mathtt{x}[2] & \text{si } \mathtt{x} \text{ est un racine ou si } \mathtt{x}[2] = 0 \\ (\alpha(n) - \mathrm{niv}(\mathtt{x})) \cdot \mathtt{x}[2] - \mathrm{iter}(\mathtt{x}) & \text{sinon} \end{cases}$$

où $\mathtt{x}[2]$ désigne le rang de $\mathtt{x}$ juste après l'instruction o_k.

Question 3.5 *Montrer que* $\Phi_n(\mathtt{x}) = 0$.

Quelques propriétés sur les potentiels. Soit $k \geq n+1$ et considérons le noeud $\mathtt{x}_i$ (avec $1 \leq i \leq n$) après l'instruction o_k.

Question 3.6 *Supposons que, après l'instruction* o_k, $\mathtt{x}_i$ *n'est pas une racine et que* $\mathtt{x}_i[2] \geq 1$.

(i) Montrer que $\mathtt{x}_i[1][2] < A_{\alpha(n)}(\mathtt{x}_i[2])$.

(ii) Montrer que $\mathrm{niv}(\mathtt{x}_i) < \alpha(n)$.

(iii) Montrer que $\mathrm{iter}(\mathtt{x}_i) \geq 1$.

(iv) Montrer que $\mathrm{iter}(\mathtt{x}_i) \leq \mathtt{x}_i[2]$.

(v) Montrer que $\Phi_k(\mathtt{x}_i) < \alpha(n) \cdot \mathtt{x}_i[2]$.

(vi) Montrer que $\Phi_k(\mathtt{x}_i) \geq 0$.

Question 3.7 *Montrer que* $0 \leq \Phi_k(\mathtt{x}_i) \leq \alpha(n) \cdot \mathtt{x}_i[2]$.

Coût amortit des opérations. Soit $k \geq n+1$. Nous allons montrer que $\hat{c}_k$ est en $O(\alpha(n))$.

Question 3.8 *Supposons que, avant l'instruction* o_k, *le noeud* $\mathtt{x}_i$ *n'est pas une racine. Montrer que* $\Phi_k(\mathtt{x}_i) \leq \Phi_{k-1}(\mathtt{x}_i)$.

Question 3.9 *Supposons que* o_k *est un appel à* `relie`. *Montrer que* $\hat{c}_k$ *est en* $O(\alpha(n))$.

Soit $k \geq n+1$ et supposons que o_k est un appel à `repr`. Nous allons montrer que $\hat{c}_k$ est en $O(\alpha(n))$.

Question 3.10 *Montrer que pour tout* $i = 1, \ldots, n$, *on a* $\Phi_k(\mathtt{x}_i) \leq \Phi_{k-1}(\mathtt{x}_i)$.

Supposons que o_k est un appel à `repr(z)` et supposons qu'il y a s noeuds entre `z` et son représentant, c'est-à-dire que si $\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_\ell}$ satisfont (1), sont deux à deux distincts, et sont tels que $\mathtt{z} = \mathtt{x}_{i_1}$ et $\mathtt{x}_{i_\ell}$ est le représentant de `z`, alors $s = \ell$.

Question 3.11 *Soit* $\mathtt{x}_{i_{j_0}}$ *de rang* ≥ 1 *avant* o_k *et tel qu'il existe* $\mathtt{x}_{i_{j_1}}$ *avec* $j_0 < j_1 < \ell$ *et tel que* $\mathrm{niv}(\mathtt{x}_{i_{j_0}}) = \mathrm{niv}(\mathtt{x}_{i_{j_1}})$ *avant* o_k. *Notons* $\mathtt{x} = \mathtt{x}_{i_{j_0}}$ *et* $\mathtt{y} = \mathtt{x}_{i_{j_1}}$.

(i) Soit i *la valeur de* $\mathrm{iter}(\mathtt{x})$ *avant* o_k *et* q *la valeur de* $\mathrm{niv}(\mathtt{x}) = \mathrm{niv}(\mathtt{y})$ *avant* o_k. *Montrer que avant* o_k, *on a* $\mathtt{y}[1][2] \geq A_q^{(i+1)}(\mathtt{x}[2])$.

(ii) Soit i *la valeur de* $\mathrm{iter}(\mathtt{x})$ *avant* o_k. *Montrer que après* o_k, *on a* $\mathtt{x}[1][2] \geq A_q^{(i+1)}(\mathtt{x}[2])$.

(iii) Montrer que soit $\mathrm{niv}(\mathtt{x})$, *soit* $\mathrm{iter}(\mathtt{x})$ *croit après* o_k.

(iv) Montrer que $\Phi_k(\mathtt{x}) < \Phi_{k-1}(\mathtt{x})$.

Question 3.12 *Supposons que* o_k *est un appel à* `repr`. *Montrer que* $\hat{c}_k$ *est en* $O(\alpha(n))$.

4 La fonction d'Ackermann

Les fonctions $(A_k)_{k\in\mathbb{N}}$ définies au §3 correspondent essentiellement à la fonction dite d'Ackermann. Cette fonction est connue pour avoir une croissance très rapide. Dans cette partie, on s'intéresse aux fonctions A_k pour $k \leq 4$, en particulier afin de justifier le fait que dans le contexte du §3, on peut dans les applications concrètes toujours supposer $\alpha(n) \leq 4$.

Question 4.1 *Montrer que pour tous $i, j \in \mathbb{N}$, on a $A_1^{(i)}(j) = 2^i(j+1) - 1$.*

Question 4.2 *Soit $t : \mathbb{N} \to \mathbb{N}$ la fonction définie par $t(0) = 2$ et $t(i+1) = 2^{t(i)}$. Montrer que $t(i) < A_3(i)$ pour tout $i > 0$.*

Question 4.3 *Rappelons que le nombre d'atomes dans l'univers est de l'ordre de 10^{80}. Montrer que $A_4(1) > 10^{80}$ et justifier pourquoi, pour toute séquence d'instructions $o_1, \ldots, o_m$ pratiquement concevable, si n est le nombre d'appels à* `cree_racine` *dans $o_1, \ldots, o_m$, alors on a $\alpha(n) \leq 4$.*

* *
*