A1 – Profondeur de diagrammes de décision

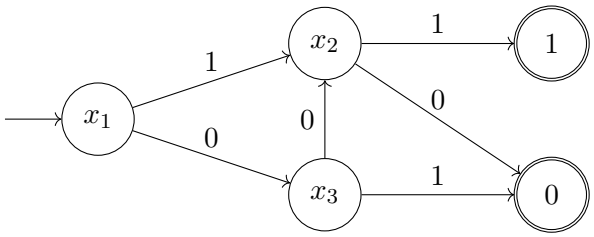
On fixe un ensemble $X = \{x_1, \dots, x_n\}$ de variables booléennes. Un *diagramme de décision* D sur X est la donnée d'un graphe orienté (V, E), supposé sans cycle, d'un nœud initial $v_{\text{init}} \in V$, d'une partition de V en $V = V_0 \sqcup V_1 \sqcup V'$, d'une partition de E en en $E = E_0 \sqcup E_1$, et d'une fonction $\mu : V' \to X$, de sorte que :

— Aucun nœud $v \in V_0$ ou $v \in V_1$ n'a d'arête sortante, i.e., il n'existe aucun $w \in V$ tel que $(v, w) \in E$;

— Tout nœud $v \in V'$ a exactement une arête sortante dans E_0 et une arête sortante dans E_1, i.e., il existe exactement un $w_0 \in V$ et exactement un $w_1 \in V$ tels que $(v, w_0) \in E_0$ et $(v, w_1) \in E_1$.

Si on se donne une *valuation* $\nu : X \to \{0, 1\}$, le diagramme de décision D associe ν à une valeur $b \in \{0, 1\}$ obtenue comme suit : on initialise le nœud courant par $v := v_{\text{init}}$, tant que le nœud courant v est dans V' alors on remplace v par $v := w_0$ ou $v := w_1$ comme défini ci-dessus selon la valeur de $\nu(\mu(v))$, et une fois que $v \in V_0$ ou $v \in V_1$ alors on renvoie 0 ou 1 suivant le cas.

Question 0. On considère $X = \{x_1, x_2, x_3, x_4\}$ et le diagramme D_0 suivant, où on indique dans chaque nœud la valeur de μ ou l'appartenance à V_0 ou V_1, et on indique le nœud initial par une flèche :



À quelle valeur est associée la valuation qui envoie x_1, x_2, x_3, x_4 respectivement vers $1, 1, 0, 1$? vers $0, 1, 0, 0$?

Question 1. Donner une formule logique décrivant la fonction booléenne représentée par D_0.

Question 2. Si on se donne une formule logique ϕ, on dit que D *représente* ϕ si ϕ est la fonction booléenne qu'il décrit. Donner un diagramme de décision D_2 représentant la fonction $\neg(x_1 \wedge x_2) \vee x_3$

Question 3. La *profondeur* d'un diagramme de décision est la plus grande longueur possible d'un chemin orienté à partir du nœud initial, en comptant le nombre de nœuds de V' traversés (y compris v_{init}). Décrire la profondeur du diagramme D_0 et celle du diagramme D_2.

Question 4. Peut-on avoir deux diagrammes de décision de profondeurs différentes qui représentent une même formule logique ?

Question 5. On appelle *profondeur minimale* d'une fonction booléenne ϕ la plus petite profondeur possible pour un diagramme de décision représentant ϕ.

Quelle est la profondeur minimale de la fonction identifiée en question 1 ?

Question 6. Une fonction booléenne ϕ sur n variables est dite *évasive* si sa profondeur minimale est de n. Donner un exemple d'une famille infinie de fonctions évasives, et justifier.

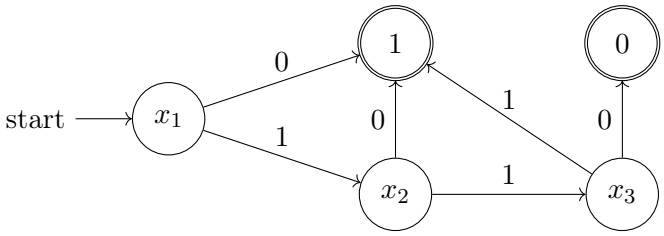
Question 7. On considère, pour tout $n \geq 1$, la fonction booléenne ψ_n définie par la formule $(x_0 \wedge x_1) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee \cdots \vee (x_{n-1} \wedge x_n)$. Ces fonctions sont-elles évasives ? Justifier.

Corrigé

Question 0. La première valuation est associée à 1 (test x_1 puis x_2), la deuxième est associée également à 1 (test x_1 puis x_3 puis x_2).

Question 1. On peut facilement obtenir une formule en forme normale disjonctive en considérant les chemins vers 1 : $(x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3 \wedge x_2)$. Ou encore, en factorisant puis en simplifiant : $x_2 \wedge (x_1 \vee \neg x_3)$.

Question 2. On peut récrire cela en utilisant la loi de de Morgan : $\neg x_1 \vee \neg x_2 \vee x_3$. Ainsi, par exemple :



Question 3. La profondeur du diagramme D_0 est de 3, celle de D_2 est également de 3 (ce sera toujours au moins 3 quelle que soit la réponse à la question 2, possiblement davantage si le diagramme teste la même variable plusieurs fois sur un même chemin).

Question 4. Manifestement oui : on peut toujours augmenter la profondeur en remplaçant le nœud initial par un test inutile dont les deux branches amènent au même nœud.

Question 5. Clairement cette profondeur est au plus de 3, vu que D_0 témoigne qu'une profondeur 3 est réalisable. Justifions qu'une profondeur de 2 est impossible. Supposons par l'absurde qu'on ait un diagramme D_5 de profondeur 2 représentant cette fonction. Distinguons suivant la variable étiquetant v_{init} :

- Si c'est x_2, alors considérons le sommet obtenu après 1. Il faut à présent vérifier que $x_1 \vee \neg x_3$ est vrai, or on n'a testé aucune de ces deux variables et on a un unique test avant de devoir atteindre une feuille, c'est clairement impossible.
- Si c'est x_1, considérons le sommet obtenu après 0. Il faut à présent vérifier que $x_2 \wedge \neg x_3$ est vrai avec une profondeur restante de 1, et comme précédemment c'est impossible.
- Si c'est x_3, considérons le sommet obtenu après 0. Il faut à présent vérifier que $x_2 \wedge x_1$ est vrai, c'est impossible pour la même raison.
- Le cas x_4 est manifestement inutile vu que la formule ne dépend pas de la valeur de x_4.

Ainsi, par l'absurde, il n'y a pas de diagramme de profondeur 2 représentant la même fonction que D_0, donc la profondeur minimale de cette fonction est bien de 3.

Question 6. *[Indication 1 : qu'est-ce qui fait qu'une fonction booléenne est évasive ?]*

[Indication 2 : en retirant x_4, est-ce que la fonction de la question 0 était évasive ? pourquoi, intuitivement ?]

[Indication 3 : donner la famille : $\phi_n : x_1 \wedge \cdots \wedge x_n$, et demander la preuve. (Cela fonctionne aussi, par dualité, pour $x_1 \vee \cdots \vee x_n$; et également pour $x_1 \otimes \cdots \otimes x_n$.)]

Démonstration du caractère évasif de $\phi_n : x_1 \wedge \cdots \wedge x_n$. Supposons par l'absurde qu'un diagramme de décision de profondeur $< n$ teste ϕ_n. Considérons la valuation où toutes les variables sont vraies, et son chemin du nœud initial vers une feuille, nécessairement étiquetée par 1. Par le principe des tiroirs, il y a une variable, mettons x_i, qui n'est pas testée sur ce chemin. Ainsi, en changeant la valeur de x_i à 0, on a une valuation qui est également envoyée vers 1 par le diagramme, or ϕ_n s'évalue alors à 0, contradiction.

Question 7. La fonction $\psi_1 : x_0 \wedge x_1$ est évasive pour la même raison qu'à la question précédente.

La fonction $\psi_2 : (x_0 \wedge x_1) \vee (x_1 \wedge x_2)$ est également évasive :

— si l'on teste d'abord x_0 et que la réponse est 0, on doit tester $x_1 \wedge x_2$ qui est évasive

— même raisonnement pour x_2

— si l'on teste d'abord x_1 et que la réponse est 1, on doit tester $x_0 \vee x_2$, qui est évasive.

En revanche, de façon surprenante, la fonction $\psi_3 : (x_0 \wedge x_1) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$ n'est *pas* évasive ! En effet :

— On teste d'abord x_1. Si la réponse est 0, alors on doit tester $x_2 \wedge x_3$ et on n'a pas besoin de tester x_0.

— Si on a $x_1 = 1$, alors il reste à tester $x_0 \vee x_2 \vee (x_2 \wedge x_3)$, ce qui se simplifie en $x_0 \vee x_2$, et on n'a pas besoin de tester x_3.

On peut facilement montrer par récurrence que, pour n multiple de 3, la fonction ψ_n n'est pas évasive. En effet, le cas de base est traité ci-dessus. Ensuite :

— On teste d'abord x_1. Si la réponse est 0, alors on doit tester le reste de la fonction mais sans dépendre de x_0.

— Si on a $x_1 = 1$, alors il reste à tester $x_0 \vee x_2 \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4) \vee \cdots$, ce qui se simplifie en $x_0 \vee x_2 \vee (x_3 \wedge x_4) \vee \cdots$, et quitte à tester x_0 et x_2, et à renommer les variables, on est ramené au cas ψ_{n-3} qui n'est pas évasive par récurrence.

On peut ensuite montrer par récurrence avec prédécesseurs que, pour tout autre n, la fonction ψ_n est évasive. En effet, pour tout x_i que l'on teste en premier :

— Si la réponse est 0, alors on doit tester $(x_0 \wedge x_1) \vee \cdots \vee (x_{i-2} \wedge x_{i-1}) \vee (x_{i+1} \wedge x_{i+2}) \vee \cdots \vee (x_{n-1} \wedge x_n)$. On est donc ramené au cas de ψ_{i-1} et ψ_{n-i-1}.

— Si la réponse est 1, alors on doit tester $(x_0 \wedge x_1) \vee \cdots \vee (x_{i-2} \wedge x_{i-1}) \vee x_{i-1} \vee x_{i+1} \vee (x_{i+1} \wedge x_{i+2}) \vee \cdots \vee (x_{n-1} \wedge x_n)$ ce qui se simplifie en $(x_0 \wedge x_1) \vee \cdots \vee (x_{i-3} \wedge x_{i-2}) \vee x_{i-1} \vee x_{i+1} \vee (x_{i+2} \wedge x_{i+3}) \vee \cdots \vee (x_{n-1} \wedge x_n)$ et on est donc ramené au cas de x_{i-1}, x_{i+1}, ψ_{i-2} et ψ_{n-i-2}.

Ainsi, si ni $i-1$ ni $n-i-1$ ne sont multiples de 3, on considère le cas d'une réponse 0, et on se ramène à $\psi_{i-1} \vee \psi_{n-i-1}$, qui sont sur des variables différentes : toutes deux sont évasives par hypothèse de récurrence avec prédécesseurs, et on montre aisément que la disjonction de deux fonctions évasives sur des variables disjointes est elle-même évasive.

Si l'un de $i-1$ ou $n-i-1$ est multiple de 3, alors on considère le cas d'une réponse 1, et on se ramène à $x_{i-1} \vee x_{i+1} \vee \psi_{i-2} \vee \psi_{n-i-2}$. On fait alors une disjonction de cas :

— Si n est congru à 1 modulo 3, alors :

  — Si $i-1$ est multiple de 3 alors $i-2$ est congru à 2 modulo 3, et $n-i-2$ est congru à $1-1-2 = 1 \bmod 3$.

  — Si $n-i-1$ est multiple de 3 alors $n-i-2$ est congru à 2 modulo 3, et $i-2$ est congru à $-(n-i-2)+n-4 \bmod 3$ c'est-à-dire $1+1-4 = 1 \bmod 3$.

— Si n est congru à 2 modulo 3, alors :

  — Si $i-1$ est multiple de 3 alors $i-2$ est congru à 2 modulo 3, et $n-i-2$ est congru à $2-1-2 = 2 \bmod 3$.

  — Si $n-i-1$ est multiple de 3 alors $n-i-2$ est congru à 2 modulo 3, et $i-2$ est congru à $1+2-4 = 2 \bmod 3$.

— Comme n n'est pas multiple de 3 par hypothèse, cette disjonction de cas est exhaustive.

On conclut donc que, dans tous les cas, soit $i-1$ et $n-i-1$ sont tous deux non-multiples de 3, soit $i-2$ et $n-i-2$ sont tous deux non-multiples de 3. Dans la première situation, on suppose que la réponse est 0, et par hypothèse de récurrence avec prédécesseurs, les deux fonctions ψ_{i-1} et ψ_{n-i-1} sont évasives et donc leur disjonction l'est aussi car elles sont sur des variables disjointes. Dans la seconde situation, on suppose que la réponse est 1, et on conclut comme précédemment car les deux fonctions ψ_{i-2} et ψ_{n-i-2} sont évasives.

Ainsi, on a montré que quelle que soit la première variable que l'on choisisse de tester, il y a une réponse pour cette variable qui nous ramène à une fonction sur les autres variables qui est évasive. Ceci justifie clairement que la fonction que nous considérons est évasive, et conclut la démonstration.

[Pour une question ouverte généralisant ces notions, voir : cf [Ama].]

Références

[Ama] Antoine Amarilli. Which monotone DNFs are evasive ? Theoretical Computer Science Stack Exchange. URL :https ://cstheory.stackexchange.com/q/44278 (version : 2019-07-17).

A2 – Langages réguliers épars

On fixe l'alphabet fini $\Sigma = \{a, b\}$. Un *mot* $w \in \Sigma^*$ est une suite finie d'éléments de Σ, et un langage $L \subseteq \Sigma^*$ est un ensemble de mots. La *densité* de L est la fonction $\delta_L : \mathbb{N} \to \mathbb{N}$ où $\delta_L(n) := |L \cap \Sigma^n|$ pour $n \in \mathbb{N}$ est le nombre de mots de longueur n dans L. On dit que L est *épars* si on a $\delta_L = O(P)$ pour un polynôme P.

Un langage *régulier* est un langage dénoté par une expression rationnelle, ou reconnu par un automate déterministe fini (on admet que ces caractérisations sont équivalentes). Tous les automates sont supposés déterministes.

Question 0. Donner un exemple d'un langage régulier épars, et d'un langage régulier non-épars.

Question 1. Est-il vrai que l'union de deux langages réguliers épars est un langage régulier épars ? Que penser de la concaténation ? Que penser de l'itération (étoile de Kleene) ?

Question 2. Est-il vrai qu'un langage est épars si et seulement si son complémentaire ne l'est pas ?

Question 3. Soient u, v, v', w des mots de Σ^*. Supposons qu'un langage L contienne tous les mots du langage dénoté par l'expression régulière $u(av|bv')^*w$. Montrer que L n'est pas épars.

Question 4. En s'inspirant de la question précédente, proposer une condition suffisante sur un automate fini pour que le langage accepté par l'automate ne soit pas épars.

Question 5. Écrire le pseudocode d'un algorithme naïf pour tester le critère de la question 4 sur un automate fourni en entrée, et discuter de sa complexité.

Question 6. On suppose qu'étant donné l'automate, on dispose d'une fonction permettant de calculer les composantes fortement connexes de son graphe en temps linéaire. En déduire un algorithme pour tester le critère de la question 4 en temps linéaire.

Question 7. Montrer que le critère identifié en question 4 est en réalité une caractérisation des automates reconnaissant des langages qui ne sont pas épars.

Question 8. Conclure à la caractérisation suivante : les langages réguliers épars sont exactement les unions finies de langages de la forme $u_0 v_1^* u_1 v_2^* u_2 \cdots v_k^* u_k$ pour des mots $u_0, \ldots, u_k, v_1, \ldots, v_k \in \Sigma^*$.

Suite des questions

Question 9. Quels sont les régimes possibles pour la fonction de densité d'un langage régulier, et quels sont les langages les réalisant ?

Question 10. Quelle est la complexité, étant donné un automate, de déterminer le régime de la fonction de densité de son langage ?

Corrigé

Question 0. Le langage vide est épars. Le langage Σ^* ne l'est pas puisque δ_{Σ^*} est la fonction qui à n associe 2^n, qui n'est pas dominée par un polynôme.

Question 1. On note d'abord que les langages réguliers sont clos par ces opérateurs, donc il suffit de vérifier si les langages épars le sont. *[On s'attend bien à ce que la clôture des langages réguliers par ces opérations soit mentionnée.]*

Les langages épars sont clos par union : pour L_1 et L_2 deux langages épars, si on a $\delta_{L_1} = O(P_1)$ et $\delta_{L_2} = O(P_2)$ pour deux polynômes P_1 et P_2, alors comme pour tout $n \in \mathbb{N}$ on a $|(L_1 \cup L_2) \cap \Sigma^n| = |L_1 \cap \Sigma^n \cup L_2 \cap \Sigma^n| \leq |L_1 \cap \Sigma^n| + |L_2 \cap \Sigma^n|$, on a $\delta_{L_1 \cup L_2} = O(P_1 + P_2)$ et $P_1 + P_2$ est également un polynôme. *[On exigera la preuve formelle.]*

Les langages épars sont clos par concaténation : soient L_1 et L_2 deux langages épars. Pour simplifier, on les suppose dominés par un polynôme commun, de sorte à ce que $\delta_{L_1} = O(P)$ et $\delta_{L_2} = O(P)$ pour un polynôme P, que pour simplifier l'on supposera également croissant. Pour tout $n \in \mathbb{N}$, on a $|L_1 L_2 \cap \Sigma^n| = \left|\cup_{0 \leq i \leq n} (L_1 \cap \Sigma^i) \times (L_2 \cap \Sigma^{n-i})\right|$. Ceci est majoré par $\sum_{0 \leq i \leq n} \left|L_1 \cap \Sigma^i\right| \times \left|L_2 \cap \Sigma^{n-i}\right|$, majorable par hypothèse par $\sum_{0 \leq i \leq n} P(i)P(n-i)$. Comme P est croissant ceci se majore grossièrement par $(n+1)P(n)^2$. C'est également un polynôme.

Les langages épars ne sont manifestement pas clos par étoile : le langage fini $a + b$ est épars mais son itération est Σ^* qui ne l'est pas.

Question 2. C'est faux, car il se peut qu'aucun des deux ne soit épars : le complémentaire de $a\Sigma^*$ est $b\Sigma^*|\epsilon$ et aucun des deux n'est épars.

En revanche, il est bien sûr impossible qu'un langage et son complémentaire soient tous deux épars, puisque leur union serait alors éparse par la question précédente, or il s'agit de Σ^* qui ne l'est pas.

Question 3. Soit L' le langage de l'expression rationnelle. Il suffit de montrer que L' n'est pas épars pour conclure. *[La formulation de la question avec L et L' est pour aider pour la question suivante.]*

Posons n_0 la somme des longueurs de u et w, posons m le PPCM des longueurs de av et bv' de sorte que $m = p\,|av| = q\,|bv'|$, et étudions $\delta_{L'}$ aux valeurs $n_0 + km$ pour $k \in \mathbb{N}$. Pour chaque "bloc" de taille m, on a au moins deux possibilités (manifestement différentes) : le remplir par des av, ou le remplir par des bv'. (Bien sûr, on a potentiellement davantage de possibilités en réalité.) Mais ceci assure que $\delta_{L'}(n_0 + km) \geq 2^k$, manifestement non dominable par un polynôme.

Question 4. *[Attention, pour le bon déroulement de la suite du sujet, il faut parvenir exactement à la formulation proposée.]*

La question 3 suggère que si l'on considère un automate déterministe alors il reconnaîtra un langage non-épars s'il y a un état q satisfaisant les conditions suivantes :

— Il y a un chemin (correspondant à u) allant de l'état initial à q ;

— Il y a une transition a allant de q à un état q', et un chemin (correspondant à v) revenant de q' à q ;

— Il y a une transition b allant de q à un état q'', et un chemin (correspondant à v') revenant de q'' à q ;

— Il y a un chemin (correspondant à w) allant de q à un état final.

On peut reformuler : il y a un état q dans l'automate qui est accessible et co-accessible (attention, ces deux termes ne sont pas au programme) avec deux boucles différentes allant de q à lui-même, l'une commençant par a et l'autre par b.

C'est une condition suffisante : si un automate présente ce motif alors il reconnaît un langage non épars par la question précédente. La question 7 montre qu'il s'agit en réalité d'une condition nécessaire et suffisante : un automate sans ce motif reconnaît un langage épars.

Question 5. Attention, le graphe correspondant à l'automate est en réalité un multi-graphe, avec potentiellement des boucles. Les détails de la représentation d'entrée ne sont pas spécifiés pour pouvoir en discuter avec le candidat ou la candidate, qui peut faire les choix qui l'arrangent. Il faut être indulgent toutefois car les multi-arêtes et les boucles ne sont explicitement pas au programme.

Pour commencer, il ne faut pas oublier d'éliminer les états inutiles. On peut le faire naïvement en temps quadratique, voire cubique (pour chaque état, pour chaque état final, faire un test d'accessibilité ; et de même avec l'état initial), ou plus intelligemment en temps linéaire : exploration BFS à partir de l'état initial pour marquer les accessibles, et une seule exploration BFS à partir de l'ensemble des états finaux (et en inversant le sens des arêtes pour marquer les co-accessibles). *[Pour cette question, ne pas exiger d'algorithme efficace pour cela : cf la question suivante.]*

Ensuite, on teste le motif à rechercher : pour chaque choix de sommet q ayant deux transitions a et b allant respectivement vers q' et q'' (noter que bien sûr q, q', q'' ne sont pas nécessairement distincts), on teste s'il y a un chemin de q' à q et de q'' à q. C'est en $O(nm)$ pour n le nombre d'états et m le nombre de transitions.

Par exemple :

```
Input: graphe G de n sommets représenté par listes d'adjacences

Fonction Accessible(u, v):
  Q := file()
  Q.insère(u)
  Visité = tableau de n cases initialisé à False
  Tant que Q n'est pas vide:
    x := Q.top()
    Q.pop()
    Si x == v:
      Renvoyer Vrai
    Fin Si
    Si Visité[x]:
      Continuer
    Fin Si
    Visité[x] := Vrai
    Pour chaque (x, y):
      Q.push(y)
    Fin Pour
  Fin Tant que
  Renvoyer Faux
Fin Fonction

Utile = tableau de n cases initialisé à Vrai
```

```
Pour chaque état q:
  Si non Accessible(qinit, q):
    Utile[q] := Faux
  Fin Si
Fin Pour

Pour chaque état q:
  OK := Faux
  Pour chaque état final qf:
    Si accessible(q, qf):
      OK := Vrai
      Break
    Fin Si
  Fin Pour
  Si non OK:
    Utile[q] := Faux
  Fin Si
Fin Pour

Pour chaque q ayant deux transitions (q, qa) et (q, qb):
  Si Utile[q] et accessible(qa, q) et accessible(qb, q):
    Renvoyer Vrai
  Fin Si
Fin Pour

Renvoyer Faux
```

Question 6. *[On rappelle que la notion de composante fortement connexe (CFC) d'un graphe orienté est exigible.]*

Il faut comme précédemment éliminer les états non-accessibles et non-coaccessibles, mais cette fois il faut impérativement le faire en temps linéaire.

On note que le motif que l'on recherche se situe forcément dans une CFC, donc il suffit de considérer chaque CFC.

À présent, notons qu'une occurrence du motif interdit dans une CFC témoigne de l'existence d'un état q avec des transitions a et b menant à des états respectifs q' et q'' qui sont dans la même CFC. Mais réciproquement, s'il y a un tel état q, alors comme q' et q'' sont dans la même CFC que q, il y a un chemin retour, et donc un motif interdit.

Pour le reformuler autrement, la condition revient à affirmer que chaque CFC est soit un cycle, soit un cycle vide (un seul état dont toutes les transitions sortantes sortent de la CFC).

Pour résumer, après élimination des états inutiles, si on suppose chaque état annoté par un identifiant de sa CFC, alors il suffit de vérifier chaque état q pour tester si on a deux transitions sortantes qui sont vers des états de la même CFC que q (potentiellement identiques, potentiellement q lui-même). C'est manifestement faisable en temps linéaire.

Question 7. Montrons que si l'automate n'a pas le motif interdit, ou autrement dit si chacune de ses CFC est un cycle (après suppression des états inutiles), alors le langage qu'il reconnaît est épars. L'automate étant déterministe, chaque mot de son langage a un unique chemin acceptant (allant d'un état initial à un état final). Notons K le nombre de CFC ; c'est une constante.

On considère l'application ϕ associant à chaque mot les informations suivantes sur son chemin

acceptant : pour chaque CFC, l'information de la première et dernière position du chemin acceptant où on est dans cette CFC (c'est clairement un segment du chemin, noter en particulier que par définition on ne peut pas revenir dans une CFC une fois qu'on en est parti), et les états à cette première et dernière position. L'application ϕ associe donc à un mot de longueur n un K-tuple d'entiers entre 0 et n et un $2K$-tuple d'états de Q. Ainsi, il y a au plus $(n+1)^K \times |Q|^{2K}$ images possibles ; c'est un polynôme P de degré K.

Or, l'application ϕ est injective. En effet, sachant qu'un chemin acceptant se trouve à l'état q en position i et r en position j, le chemin entre i et j doit rester dans la CFC commune de q et r, et comme cette CFC est un cycle il y a un seul chemin possible.

Pour L le langage accepté par l'automate, on a donc $|L \cap \Sigma^n| \leq P$ par ce qu'on vient de dire, ce qui conclut que le langage est épars.

Question 8. Un langage de la forme indiquée est manifestement régulier, et épars pour la même raison qu'en question précédente. Ainsi, des unions finies de tels langages sont des langages réguliers épars.

Pour la réciproque, on raisonne comme à la question précédente : un langage régulier épars est accepté par un automate qui satisfait donc le critère de la question 4, or le langage accepté par un tel automate s'écrit comme une union finie de langages de la forme requise suivant leur image par la fonction ϕ. Intuitivement, les u_i correspondent à des tours complets de boucle dans une CFC, et les v_i correspondent à des déplacements entre CFC ou à des tours incomplets de boucle dans une CFC (qu'on peut mettre par convention à la fin du segment où l'on reste dans une CFC).

Question 9. Il y a les langages non-épars dont la fonction de densité est $\Theta(2^{\Theta(n)})$, que nous avons caractérisés comme ceux n'ayant pas la forme de la question 8, ou acceptés par un automate comportant le motif interdit de la question 4. Ensuite, on peut montrer que pour un langage épars L, les conditions suivantes sont équivalentes :

1. L est de densité $O(n^k)$;
2. L est accepté par un automate où il n'y a pas de chemin acceptant qui passe par $k+1$ CFC cycliques (CFC qui ne sont pas un singleton sans boucle) ;
3. L est une union finie de langages de la forme $u_0 v_1^* u_1 v_2^* u_2 \cdots v_t^* u_t$ avec $t \leq k$.

En effet, (3) implique manifestement (1). Ensuite, (2) implique (3) exactement comme à la question 7. Par ailleurs, la négation de (2) implique la négation de (1) dans le même esprit qu'en questions 3 et 4 : si l'on prend un chemin témoin passant par $k+1$ CFC cycliques avec au moins un tour de boucle de chaque, que l'on ajuste les parties intermédiaires pour identifier des tours entiers de boucles dans chaque CFC, et qu'on prend le PPCM des longueurs de boucle, alors on a la possibilité de déplacer les $k+1$ points intermédiaires entre CFCs cycliques en contrôlant le nombre de tours de chaque boucle, donc $\Omega(n^{k+1})$ possibilités.

D'où la caractérisation :

1. L est de densité $\Theta(n^k)$;
2. L est accepté par un automate où il n'y a pas de chemin acceptant qui passe par $k+1$ CFC cycliques (CFC qui ne sont pas un singleton sans boucle) mais il y en a un qui passe par k telles CFC ;
3. L est une union finie de langages de la forme $u_0 v_1^* u_1 v_2^* u_2 \cdots v_t^* u_t$ avec $t \leq k$, avec $t = k$ pour au moins un terme de l'union.

Noter le cas particulier $k = 0$ où la densité est donc bornée, il n'y a aucune CFC cyclique (donc le langage est bien fini), et L est une union finie de langages singletons.

Ceci implique en particulier que, parmi les langages épars, la densité doit être équivalente à un polynôme dont le degré est déterminable par la caractérisation ci-dessus, et notamment qu'aucun régime "intermédiaire" n'est possible.

Question 10. On a vu en question 6 comment tester en temps linéaire si un automate accepte ou non un langage épars (par la caractérisation des questions 4 et 7).

Si le langage est épars, une fois calculées les CFC, et étiquetées les CFC cycliques, le plus grand nombre de CFC cycliques traversées par un chemin acceptant se calcule de bas en haut en temps linéaire, ce qui permet de savoir en temps également linéaire où on se place dans les régimes de la question 9.

[Les résultats de ce sujet ont été originellement montrés dans [SYZS92], et peuvent également être trouvés dans [Pin19], Chapitre XII, Section 4.2]

Références

[Pin19] Jean-Éric Pin. Mathematical foundations of automata theory. `https://www.irif.fr/~jep/PDF/MPRI/MPRI.pdf`, 2019.

[SYZS92] Andrew Szilard, Sheng Yu, Kaizhong Zhang, and Jeffrey Shallit. Characterizing regular languages with polynomial densities. In *MFCS*, 1992. Non disponible en libre accès.

A3 – Maintenance incrémentale de langages réguliers

On fixe un alphabet fini Σ. Un *mot* $w \in \Sigma^*$ est une suite finie d'éléments de Σ, et un langage $L \subseteq \Sigma^*$ est un ensemble de mots. Un langage est *régulier* s'il est reconnu par un automate fini, ou dénoté par une expression rationnelle (on admet que ces caractérisations sont équivalentes).

Question 0. Si l'on fixe un langage régulier $L \subseteq \Sigma^*$, le problème d'*appartenance à* L est de déterminer, étant donné en entrée un mot $w \in \Sigma^*$, si $w \in L$. Quelle est la complexité du problème d'appartenance à L en fonction de la longueur du mot d'entrée ?

Ce sujet s'intéresse à la *complexité incrémentale* du problème d'appartenance à un langage régulier L. Dans ce problème, on reçoit en entrée un mot $w \in \Sigma^*$ de longueur n. On effectue d'abord un *pré-traitement* pour déterminer si $w \in L$ et pour construire si on le souhaite une structure de données auxiliaire : cette phase de pré-traitement doit s'exécuter en $O(n)$. Ensuite, on reçoit des *mises à jour*, c'est-à-dire des paires (i, a) pour $1 \leq i \leq n$ et $a \in \Sigma$, données l'une après l'autre. À chaque mise à jour, on modifie le mot w pour que sa i-ème lettre devienne a, et on doit déterminer si $w \in L$ après cette modification. La longueur n du mot ne change jamais. La *complexité incrémentale* d'un langage est la complexité dans le pire cas pour prendre en compte une mise à jour, exprimée en fonction de n.

Question 1. Montrer que tout langage régulier a une complexité incrémentale en $O(n)$.

Question 2. Montrer que le langage régulier a^* sur l'alphabet $\Sigma = \{a, b\}$ a une complexité incrémentale en $O(1)$.

Question 3. Soit L_3 le langage des mots sur l'alphabet $\Sigma = \{a, b\}$ comportant au moins deux a, un nombre pair de a, et un nombre de b qui n'est pas divisible par 3. Ce langage est-il régulier ? Quelle est sa complexité incrémentale ?

Question 4. On dénote par $w_1, \ldots, w_n$ les lettres d'un mot $w \in \Sigma^*$ de longueur n. Pour toute permutation $\sigma \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$, on écrit par abus de notation $\sigma(w)$ pour désigner le mot $w_{\sigma(1)} \cdots w_{\sigma(n)}$. Un langage L est *commutatif* si pour tout $w \in \Sigma^*$, pour n la longueur de w, pour toute permutation $\sigma \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$, on a $w \in L$ si et seulement si $\sigma(w) \in L$.

Montrer que tout langage régulier commutatif a une complexité incrémentale en $O(1)$.

Question 5. Proposer une structure de données pour stocker un ensemble d'entiers S qui supporte les opérations suivantes :

- Ajouter un entier dans S, en $O(1)$;
- Retirer un entier de S, en $O(1)$;
- Parcourir les entiers actuellement stockés dans S, en $O(|S|)$.

Écrire le pseudocode pour ces opérations.

Question 6. On considère le langage L_6 sur l'alphabet $\Sigma = \{a, b, c\}$ dénoté par l'expression rationnelle $c^*ac^*bc^*$. Montrer que sa complexité incrémentale est $O(1)$ en utilisant la structure de données de la question précédente.

Suite des questions

Question 7. À quelle classe de langages peut-on généraliser cette technique ?

Question 8. Soient deux langages L_1 et L_2 de complexité incrémentale en $O(1)$. Quelle est la complexité incrémentale des langages $L_1 \cap L_2$ et $L_1 \cup L_2$?

Question 9. On s'intéresse aux langages réguliers admettant au moins une "lettre neutre" (notion que le candidat ou la candidate aura dû identifier à la question 7). Proposer une classe aussi générale que possible de langages de complexité incrémentale en $O(1)$.

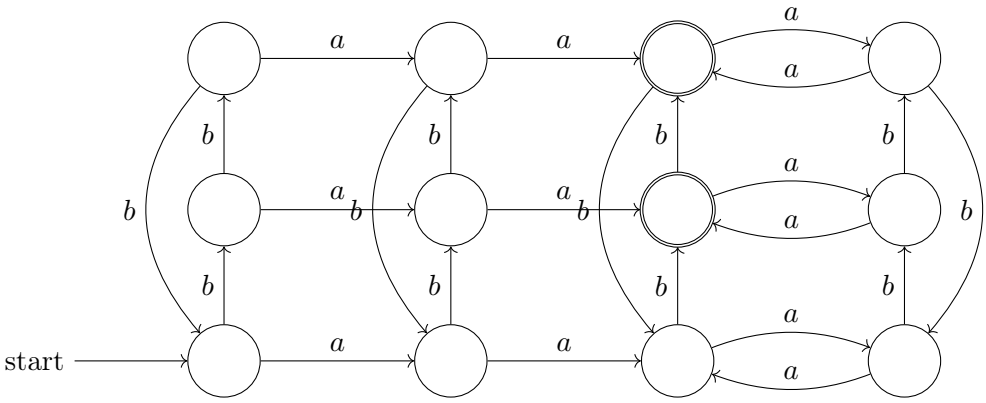
Corrigé

Question 0. On se donne un automate déterministe pour le langage L (on n'a pas besoin de se demander de la complexité de le calculer). Étant donné le mot w, on simule son exécution dans l'automate. La complexité est donc en $O(n)$. *[C'est surtout une question de cours, on attend juste $O(n)$ avec une phrase d'explication. On ne demande pas la complexité en fonction de l'automate ou autre représentation du langage.]*

Question 1. On stocke le mot w dans un tableau (ce qu'on fera toujours par la suite). Il suffit, à chaque mise à jour, de refléter la modification sur le tableau, puis de réévaluer si le mot courant appartient ou non au langage avec la question précédente.

Question 2. On maintient un compteur du nombre courant de a et du nombre courant de b, qu'on initialise au prétraitement. On maintient à jour ces compteurs à chaque mise à jour (ceci nécessite de connaître le nouveau caractère, et l'ancien caractère qui vient d'être changé, qu'on retrouve avec le tableau). Le mot courant est dans le langage si et seulement si le nombre de b est de 0.

Question 3. Le langage est régulier. On peut exhiber un automate :



On peut aussi remarquer plus intelligemment que c'est l'intersection de trois langages dont on montre facilement qu'ils sont réguliers (en exhibant un automate), or on sait d'après le cours que les langages réguliers sont clos par intersection.

Sa complexité incrémentale est en $O(1)$: on maintient un compteur du nombre de a et de b et on teste la condition.

Question 4. *Indication : considérer un automate déterministe et précalculer des chemins.*

On considère un automate déterministe pour le langage concerné. Dans le cadre du précalcul en $O(n)$, on calcule pour chaque état q de l'automate, pour chaque lettre $a \in \Sigma^*$, pour chaque entier

$0 \leq i \leq n$, la fonction $\delta^i(q, a)$ qui indique l'état auquel on aboutit après lecture de a^i depuis q. Ce calcul se fait de proche en proche : $\delta^0(q, a) := q$ et $\delta^{i+1}(q, a) := \delta(\delta^i(q, a), a)$ pour $\delta = \delta^i$ la fonction de transition de l'automate. Noter que le nombre d'états de l'automate et de lettres est une constante indépendante de la longueur n du mot, donc ce précalcul est bien en $O(n)$.

On maintient avec complexité incrémentale $O(1)$ un compteur du nombre d'occurrences de chaque lettre dans le mot, comme à la question précédente.

Pour savoir si le mot courant appartient à L, on utilise le fait que L est commutatif pour savoir que c'est le cas si et seulement si le mot $a_1^{n(a_1)} a_2^{n(a_2)} \cdots a_k^{n(a_k)}$ y appartient, où $a_1, \ldots, a_k$ désigne les lettres de Σ et $n(a)$ désigne le nombre d'occurrences dans le mot courant de la lettre a, qui est l'information maintenue par les compteurs. En effet, par définition des compteurs, le mot en question est une permutation du mot courant.

Or l'acceptation de ce mot par l'automate se détermine en $O(1)$ à l'aide des fonctions précalculées : on calcule $q_1 = \delta^{n(a_1)}(q_0, a_1)$, puis $q_2 = \delta^{n(a_2)}(q_1, a_2)$, et ainsi de suite, en k étapes où k est la taille de l'alphabet qui est une constante indépendante de n. On teste enfin si le dernier état obtenu est final ou non.

Question 5. *Indication : utiliser un tableau, mais s'inspirer de la structure de liste.*

Le problème est qu'une simple liste ne permet pas de retirer un élément identifié par sa valeur (il faut retrouver le maillon de liste qui le stocke), et qu'un tableau ne permet pas de parcourir efficacement les cases occupées. On pourrait naturellement concilier les deux, c'est-à-dire une structure de liste avec un tableau stockant des pointeurs vers les maillons. Pour que la suppression soit plus simple, on utiliserait spontanément des listes doublement chaînées.

Ceci dit, les pointeurs et les listes chaînées ne sont pas au programme, donc on attend plutôt une solution élémentaire à base de tableaux uniquement.

On stocke un tableau "suivant", un tableau "précédent", et une variable "début", avec l'invariant que les éléments actuellement stockés peuvent être parcourus comme début, suivant[début], suivant[suivant[début]], etc., jusqu'à atteindre la valeur -1.

```
début := -1
suivant := tableau d'entiers de taille n initialisé à -1
précédent := tableau d'entiers de taille n initialisé à -1

Fonction Énumérer():
  courant := début
  Tant que courant != -1:
    Produire courant
    courant := suivant[courant]
  Fin Tant que
Fin Fonction

Fonction Ajouter(x):
  // On ajoute au début
  Assertion(suivant[x] == précédent[x] == -1)
  suivant[x] := début
  // précédent[x] reste à -1
  Si début != -1:
    précédent[début] := x
  Fin Si
  début := x
Fin Fonction
```

```
Fonction Supprimer(x):
  // Si on supprime le premier élément, on change début
  Si début == x:
    début := suivant[x]
  Fin Si
  // On saute par dessus l'élément supprimé
  Si suivant[x] != -1:
    précédent[suivant[x]] := précédent[x]
  Fin Si
  Si précédent[x] != -1:
    suivant[précédent[x]] := suivant[x]
  Fin Si
  // Maintenant, on supprime
  suivant[x] := -1
  précédent[x] := -1
Fin Fonction
```

Noter que bien sûr les éléments ne sont pas parcourus dans l'ordre.

Question 6. On maintient une structure de données de la question 5 pour l'ensemble de positions contenant la lettre a, une autre pour l'ensemble des positions contenant la lettre b, en $O(1)$ par la question précédente. On maintient le nombre de a et de b comme aux questions 2–4, en $O(1)$.

Pour savoir si le mot courant est dans L, on vérifie d'abord s'il y a exactement un a et un b. Si non, alors la réponse est non. Si oui, alors on énumère le contenu des deux structures, ce qui est en $O(1)$ car elles contiennent chacune un élément. On trouve aussi la position de l'unique a et de l'unique b, et on les compare, pour savoir si on est ou non dans L.

Question 7. Intuitivement, si on peut partitionner l'alphabet Σ entre des lettres qui n'ont pas d'effet et des lettres sur lesquelles on doit réaliser un mot fini (ou un langage fini), alors la même technique s'appliquera.

Pour être précis (on attend une telle formalisation du candidat ou de la candidate) : on définit une lettre a qui est *neutre* pour L comme à la question 13. On définit le *réduit* de L pour une lettre neutre a comme le langage $L^{-a} := \{w^{-a} \mid w \in L\}$. On étend cette définition à un ensemble non-vide de lettres neutres : $L^{-\Sigma'}$ pour $\Sigma' \subseteq \Sigma$ non-vide est l'ensemble des $w^{-\Sigma'}$ pour $w \in L$ où $w^{-\Sigma'}$ s'obtient en retirant de w toutes les lettres de Σ'. Pour un langage L avec un ensemble non-vide Σ' de lettres neutres, si $L^{-\Sigma'}$ est fini, alors la complexité incrémentale est en $O(1)$ comme à la question précédente.

Pour le montrer, on crée une structure de données de la question 5 pour chaque lettre de $\Sigma \setminus \Sigma'$. On maintient ces structures, ainsi que le nombre d'occurrences de chaque lettre, en $O(1)$. Soit N la longueur maximale d'un mot de $L^{-\Sigma'}$; c'est une constante indépendante de n. Pour savoir si le mot courant appartient ou non à L, tant que le nombre d'occurrences total des lettres de $\Sigma \setminus \Sigma'$ est $> N$, on peut répondre non directement, en $O(1)$. Sinon, on utilise les structures de données de la question 5 pour identifier le sous-ensemble (de taille au plus N) de positions contenant des lettres de Σ'. On le trie en temps $O(N \log N)$, ce qui est toujours constant. On lit le mot formé et on teste (en temps constant car il est de taille $\leq N$) s'il appartient à $L^{-\Sigma'}$, ce qui conclut.

Question 8. Si on peut maintenir une structure avec complexité incrémentale en $O(1)$ pour l'appartenance à L_1, et pour l'appartenance à L_2, alors ces structures nous permettent de savoir si on appartient à L_1 et à L_2, ou à L_1 ou à L_2, toujours en $O(1)$.

Question 9. La réponse naïve (mais nécessitant un peu de recul) est : la clôture par intersections et par unions de la classe de la question 7 et des langages commutatifs couverts en question 4. Mais on s'attend à ce que le candidat ou la candidate sache se représenter la puissance d'expression de cette classe :

— L'intersection d'un langage de la question 7 avec un langage commutatif permet de poser des conditions commutatives sur les lettres neutres (qui ne sont du coup plus neutres), mais elle n'est pas intéressante sur les autres lettres (puisque le langage est fini)

— L'union de tels langages est intéressante parce qu'elle permet de changer la partition entre lettres neutres et non-neutres. En d'autres termes, on a une complexité incrémentale de $O(1)$ pour le langage "j'ai un nombre pair de a, une lettre neutre b, et exactement un c avant exactement un d, OU j'ai un nombre pair de c, une lettre neutre d, et exactement un a avant exactement un b".

— L'union permet aussi, pour la même partition, de poser des conditions commutatives différentes suivant ce qui est attendu pour les lettres non-neutres, par exemple couvrir des langages comme "j'ai une lettre neutre d, et j'ai exactement un a avant exactement un b et un nombre pair de c OU exactement un b avant exactement un a et un nombre impair de c.

Pour résumer, une forme normale serait : une union finie de langages qui sont l'entrelacement (shuffle) d'un langage commutatif sur un alphabet Σ', et d'un singleton (un seul mot) sur l'alphabet $\Sigma \setminus \Sigma'$. *[Cette caractérisation est celle des langages ZG, voir [AJP21, AP21]. On peut conjecturer qu'il s'agit des seuls langages réguliers de complexité incrémentale $O(1)$, ou en tout cas c'est le cas sous une certaine hypothèse de complexité.]*

Références

[AJP21] Antoine Amarilli, Louis Jachiet, and Charles Paperman. Dynamic Membership for Regular Languages. In *ICALP*, 2021.

[AP21] Antoine Amarilli and Charles Paperman. Locality and Centrality: The Variety ZG. Preprint : `http://arxiv.org/abs/2102.07724`, 2021.

A4 – Énumération de marquages

On fixe l'alphabet fini $\Sigma = \{a, b\}$, et on note $\hat{\Sigma} = \{\hat{a}, \hat{b}\}$ l'alphabet des *lettres marquées*. Un *mot* $w \in \Sigma^*$ est une suite finie d'éléments de Σ. Un *marquage* d'un mot w de longueur $|w|$ est un sous-ensemble m de $\{1, \ldots, |w|\}$. L'*application* du marquage m sur le mot w est un mot $\hat{w}^m$ de longueur $|w|$ sur $\Sigma \cup \hat{\Sigma}$ défini comme suit : pour tout $1 \leq i \leq |w|$, si la i-ème lettre de w est x, alors la i-ème lettre de $\hat{w}^m$ est $\hat{x}$ si $i \in m$ et x sinon.

Un *automate à marquages* est un automate sur l'alphabet $\Sigma \cup \hat{\Sigma}$. Un tel automate A définit une fonction associant, à tout mot $w \in \Sigma$, l'ensemble noté $A(w)$ des marquages m de w tels que $\hat{w}^m$ est accepté par A. Sauf mention du contraire, on supposera toujours que les automates à marquages sont déterministes.

Question 0. Donner un automate à marquages A tel que $A(w) = \emptyset$ pour tout $w \in \Sigma^*$. Donner un automate à marquages A tel que $A(w)$ contienne uniquement le marquage $\{1, 2\}$ si w est de longueur ≥ 2, et soit l'ensemble vide sinon.

Question 1. Construire un automate à marquages A tel que $A(w)$ contienne uniquement le marquage singleton $\{|w|\}$ pour tout $w \in \Sigma^*$ non-vide.

Question 2. Construire un automate à marquages A tel que $A(w)$ soit l'ensemble des marquages singleton $\{i\}$ pour chaque entier $1 \leq i \leq |w|$ tel que la i-ème lettre de w soit $a \in \Sigma$.

Question 3. Proposer un algorithme naïf pour déterminer, étant donné un automate à marquages A sur $\Sigma \cup \hat{\Sigma}$, un mot $w \in \Sigma^*$, et un marquage m de w, si $m \in A(w)$. En déduire un algorithme naïf pour calculer, étant donné A et w, l'ensemble $A(w)$. L'implémenter en pseudocode, et déterminer sa complexité.

Question 4. Étant donné un automate à marquages A (toujours supposé déterministe) et un mot $w \in \Sigma^*$, on souhaite déterminer combien $A(w)$ contient de marquages. Proposer un algorithme efficace pour ce faire, l'implémenter en pseudocode, et en décrire la complexité.

Question 5. En déduire un algorithme moins naïf pour calculer l'ensemble $A(w)$ étant donnés A et w. En exprimer la complexité en fonction de A, de w, et de $|A(w)|$.

Question 6. Quelle est la plus faible complexité envisageable pour calculer $A(w)$? Optimiser l'algorithme précédent pour s'approcher de cette meilleure complexité.

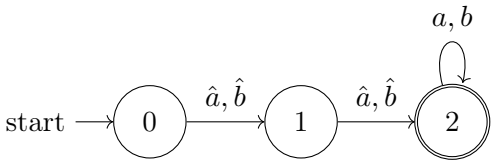
Question 7. Comment produire efficacement le i-ème marquage pour un i donné en entrée ?

Question 8. On ne suppose plus que les automates à marquages sont déterministes. Peut-on généraliser les résultats des questions précédentes ?

Corrigé

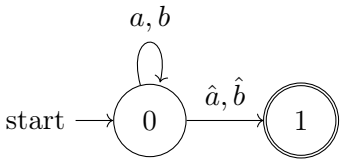
Question 0. Pour la première partie de la question, on prend simplement un automate qui n'accepte rien (sans états finaux). Ainsi l'ensemble $A(w)$ des marquages de w qui sont acceptés est vide.

Pour la seconde partie, on propose l'automate suivant :



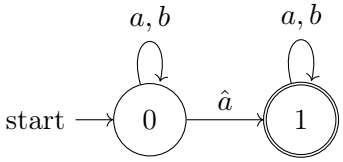
Pour un mot d'entrée w de longueur < 2, cet automate n'acceptera aucun marquage, donc $A(w) = \emptyset$. Pour un mot d'entrée w de longueur ≥ 2, cet automate lira la version marquée des deux premières lettres et la version non marquée des autres, donc $A(w)$ contient un unique marquage : $\{1, 2\}$.

Question 1. On propose :



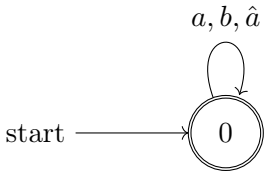
Pour un mot d'entrée w non-vide, le seul marquage que l'automate acceptera est celui où on ne marque que la dernière lettre. (Par ailleurs, on a $A(\epsilon) = \emptyset$ si le mot d'entrée est le mot vide ϵ.)

Question 2. On propose :



Étant donné un mot d'entrée w, pour qu'un marquage soit accepté, il faut et il suffit que le résultat de son application contienne un seul $\hat{a}$, donc c'est bien ce qui est demandé.

Attention, l'automate suivant ne marche pas :



Pour un mot w, l'ensemble $A(w)$ serait alors l'ensemble de tous les sous-ensembles de positions contenant un a (y compris l'ensemble vide), et non des singletons.

Question 3. On suppose que le mot d'entrée w est représenté dans un tableau, et que le marquage m est représenté dans une liste ou un tableau de booléens. L'algorithme naïf est simplement de modifier m suivant w, et de tester l'acceptation avec l'automate. La complexité est en $O(|w|)$. (Noter qu'elle ne dépend pas de l'automate, c'est normal car c'est un automate déterministe.)

On a donc l'algorithme naïf consistant à essayer tous les marquages possibles, sa complexité est en $O(2^{|w|} \times |w|)$. C'est évidemment très mauvais, et on va faire mieux.

Pseudocode (on ne demandera pas d'expliciter la représentation de l'automate, et on pourra supposer qu'on dispose d'une fonction pour tester qu'un mot est accepté par l'automate) :

```
w := mot d'entrée de n lettres
```

```
Fonction explore(i, acc):
  Si i == n:
    w2 := List.rev(acc)
    Si l'automate accepte w2:
      Produire w2
    Fin Si
  Sinon:
    x := w[i]
    explore(i+1, x::acc)
    explore(i+1, hat(x)::acc)
  Fin Si
Fin Fonction

explore(0, [])
```

Question 4. (Pour se simplifier la vie dans le corrigé, on va supposer que les automates sont complets ; on peut proposer aux candidats de faire de même si la bonne définition de « l'état obtenu après lecture de ... » devient un problème.)

Pour tout état q de l'automate, on note A_q l'automate obtenu en commençant la lecture à l'état q. On remarque d'abord une induction évidente : pour le cas de base on a $|A_q(\epsilon)| = 0$ ou $= 1$ selon que q est final ou non, et si l'on extrait la première lettre $w = xw'$, on a :

$$|A_q(w)| = \left|A_{\delta(q,x)}(w')\right| + \left|A_{\delta(q,\hat{x})}(w')\right|$$

où δ dénote la fonction de transition de l'automate.

On peut ainsi calculer les $|A_q(w[\geq i])|$ pour tous les suffixes de w de proche en proche. La complexité est en $O(|Q| \times |w|)$.

Pseudocode (on ne demandera pas d'expliciter la représentation de l'automate) :

```
w := mot d'entrée de n lettres
Count := tableau de (n+1) par |Q| cases initialisé à 0

Pour q dans Q:
  Si q est final:
    Count[n][q] := 1
  Sinon
    Count[n][q] := 0
  Fin Si
Fin Pour

Pour i de n exclu à 0 inclus:
  Pour q dans Q:
    x := w[i]
    qx := delta(q, x)
    qhatx := delta(q, hat(x))
    Count[i][q] := Count[i+1][qx] + Count[i+1][qhatx]
  Fin Pour
Fin Pour

q0 := état initial
Renvoyer Count[0][q0]
```

Question 5. On va intuitivement explorer l'arbre binaire complet de tous les marquages possibles comme en question 3. L'arbre est de hauteur $|w|$: un nœud n à profondeur i représente un marquage défini sur les i premières positions suivant le choix fait dans le chemin de la racine à n de prendre le fils gauche (ne pas marquer) ou le fils droit (marquer). Une fois parvenu à une feuille, on produira le marquage correspondant si le mot obtenu en l'appliquant est accepté.

L'idée est qu'on calcule à chaque nœud de l'arbre le nombre de marquages acceptés qui commencent par le marquage courant. Pour ce faire, on maintient au cours de notre parcours préfixe de l'arbre l'état auquel on se trouve quand on a lu le préfixe du mot d'entrée marqué comme décrit dans le préfixe de marquage de ce nœud. On utilise après la structure (précalculée) de la question précédente pour abandonner l'exploration si le sous-arbre où l'on se trouve ne contient plus aucun nœud marqué.

Quelle est la complexité de cet algorithme ? Outre le précalcul de la question précédente, appelons partie *intéressante* de l'arbre les feuilles correspondant à des marquages acceptés, ainsi que tous leurs ancêtres. Notre exploration de l'arbre ne considère que la partie intéressante, ainsi qu'éventuellement les enfants de ces nœuds (pour abandonner l'exploration à ce moment), mais ceci n'ajoute qu'un facteur constant. On effectue un traitement en temps constant à chaque nœud visité. Ainsi la complexité est proportionnelle en le nombre de nœuds de la partie intéressante. On peut la majorer par $O(|A(w)|\times|w|)$, puisque chaque feuille contribue $|w|$ nœuds au plus. (Cette majoration est grossière, mais noter que s'il y a un seul marquage dans $A(w)$ on passera bien un temps $|w|$ à le produire.) D'où une complexité totale de $O(|Q| \times |w| + |A(w)| \times |w|)$.

Question 6. On cherche une complexité en $O(\sum_{m\in A(w)} |m|)$ plus quelque chose ne dépendant que de w et A. C'est en effet la meilleure complexité « possible » puisque pour calculer tous les marquages de $A(w)$ il faut au minimum le temps de les écrire.

Pour ce faire, il faut intuitivement descendre plus vite dans l'arbre pour sauter aux feuilles, et éviter de perdre du temps à descendre étape par étape en choisissant l'enfant gauche (c'est-à-dire qu'on ne marque rien).

Formellement, on rappelle qu'un nœud est *intéressant* s'il a un descendant qui est une feuille correspondant à un marquage accepté. Un *descendant gauche* d'un nœud n est un nœud n' accessible à partir de n en descendant à gauche un certain nombre de fois. Un nœud interne est *zappable* s'il est intéressant mais que seul son enfant gauche est également intéressant.

On veut précalculer, pour chaque nœud intéressant n de l'arbre, son plus haut descendant gauche qui n'est pas zappable (possiblement une feuille). On le représente par sa profondeur et par l'état qu'on atteint alors dans l'automate dans la lecture de ce qui a été zappé.

L'algorithme sera alors le suivant. On commence à la racine, et on explore l'arbre en maintenant l'état courant (obtenu en lisant le préfixe de marquage correspondant au nœud courant). À chaque nœud, s'il est zappable, alors on saute au plus haut descendant gauche qui ne l'est pas : il n'y a rien à faire sur tous les nœuds sautés. On explore alors l'alternative correspondant à l'enfant droit (marquer la position en question, c'est-à-dire l'ajouter au marquage en cours de production dans un accumulateur) ; puis celle de l'enfant gauche (ne rien marquer) si elle est intéressante *en effectuant une récursion terminale*. Une fois parvenu à une feuille, on produit le marquage qui est actuellement dans l'accumulateur.

Il est clair que l'algorithme produit toujours le même résultat qu'avant. Justifions d'abord qu'il a la complexité attendue. Pour ce faire, le premier résultat est produit en temps constant : à chaque nœud on effectue un travail constant, et le nombre de nœuds visités est la cardinalité du marquage produit. Ensuite, l'état actuel de la pile d'appel correspond aux endroits où on a choisi d'ajouter un élément au marquage : on dépile ces appels jusqu'à trouver un endroit où le choix de ne pas mettre l'élément est intéressant, ce qui est de coût linéaire en le marquage précédent. Puis, on complète le marquage en un coût linéaire en le prochain marquage. Ainsi, au total, le nombre d'étapes est au plus deux fois la cardinalité totale des marquages.

Expliquons enfin quel précalcul on fait exactement, car on ne peut bien sûr pas faire de précalcul

réel dans l'arbre qui est de taille exponentielle. Heureusement, l'information ne dépend que la position i du nœud courant et de l'état q courant, pour nous dire qu'il faut alors zapper jusqu'à la position $i + j$ et parvenir à l'état q'. Ceci se calcule de proche en proche. Cas de base : à la fin du mot il ne faut pas zapper. Induction : à la position i et à l'état q, pour x la lettre à lire, si lire $\hat{x}$ mène à un état intéressant (c'est-à-dire $\left|A_{\delta(q,\hat{x})(w[\geq i+1])}\right| > 0$), alors quand on est à l'état q en position i on ne zappe pas. Sinon, on regarde à l'état $\delta(q, x)$ en position $i + 1$ à quelle position j et état q' on doit zapper, et c'est notre résultat. (Cette construction revient essentiellement à éliminer des ϵ-transitions dans le produit de l'automate et du mot.)

Ainsi le précalcul total est encore en $O(|Q| \times |w|)$, à quoi s'ajoute la complexité optimale recherchée.

Question 7. C'est une simple astuce : on parcourt notre arbre en considérant à chaque nœud combien de marquages sont possibles à gauche et à droite, et on récurse dans le sous-arbre gauche ou droit en fonction de comment ces nombres se comparent à l'index i désiré, qu'on ajuste quand on récurse à droite en lui soustrayant le nombre de solutions à gauche. La complexité est en $O(|w|)$ (le zapping ne nous aide pas en complexité asymptotique).

[On pourrait sans doute réaliser $O(\log |w|)$ en passant par les automates d'arbres...]

Question 8. On peut bien sûr déterminiser l'automate et appliquer les techniques des questions précédentes ; mais on paie alors une exponentielle en l'automate. La question est de savoir si on peut éviter cela, c'est-à-dire conserver une complexité polynomiale en l'automate fourni en entrée.

Le comptage, et donc la production de la i-ème solution, ne se généralisent pas. Intuitivement, le problème est que le non-déterminisme de l'automate peut nous conduire à compter plusieurs exécutions différentes mais qui reconnaîtront le même marquage. Sous une hypothèse de complexité, ce n'est pas possible.

En revanche, l'algorithme de la question 6 peut se généraliser avec une sorte de déterminisation à la volée de l'automate, et des index plus sophistiqués. Intuitivement, on n'utilise plus les comptes de la question 4 que pour savoir s'ils sont nuls ou non-nuls, ce qu'on peut effectivement calculer par une variante de la question 4 même si l'automate n'est plus déterministe. Ensuite, on maintient au cours de notre exploration l'ensemble des états où on peut se trouver sur un nœud. Les index de nœuds à zapper ne sont calculés que pour des nœuds individuels : pour chaque profondeur i et état q, on calcule la profondeur j à laquelle zapper et l'ensemble des états auquel on peut alors parvenir (ce n'est plus un seul état). Maintenant, pour savoir à un ensemble d'états Q où il faut zapper, c'est le min à prendre sur l'ensemble des nœuds individuels, donc on peut retrouver à quelle profondeur zapper. Il faut en revanche savoir à quel état on arrive à cette profondeur à partir de chaque état de Q. C'est pourquoi, à chaque profondeur i et état q, on calcule également, pour toutes les profondeurs de zapping des autres états à cette même profondeur, l'information de quels états sont accessibles.

[L'algorithme présenté ici est détaillé dans [ABMN19].]

Références

[ABMN19] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-Delay Enumeration for Nondeterministic Document Spanners. In *ICDT*, 2019.

J1 – Arbres évasés

Question 0.

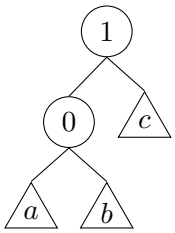
(a) Rappeler ce qu'est un arbre binaire de recherche.

(b) Quelle en est l'utilité ?

(c) Donner le pseudo-code de la fonction d'insertion dans un arbre binaire de recherche.

(d) Discuter de sa complexité en temps.

On se propose d'améliorer la complexité des requêtes sur un arbre binaire de recherche en l'équilibrant. Cet équilibrage s'appuie sur une opération locale, appelée *rotation*, dont il existe deux variantes symétriques. La *rotation à droite* agit sur un arbre de recherche de la forme suivante :



Elle le transforme en un autre arbre de recherche contenant les mêmes clefs, mais dont l'étiquette notée ici 0 est à la racine.

Question 1. Proposer une définition de l'opération de rotation à droite. En donner le pseudo-code.

Question 2.

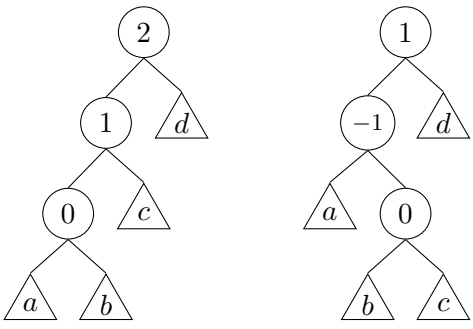
(a) En déduire un algorithme qui prend en paramètre un arbre binaire de recherche et l'étiquette de l'un de ses nœuds et transforme cet arbre binaire de recherche en déplaçant ce nœud à la racine. On supposera que toutes les étiquettes sont distinctes.

(b) Quelle est la complexité en temps de cet algorithme ?

Question 3. Proposer une amélioration de l'algorithme de recherche dans un arbre binaire de recherche, qui répond plus rapidement aux requêtes de recherche qui lui sont posées fréquemment. On ne demande pas ici de preuve de complexité.

Malheureusement, l'algorithme présenté à la question 3 n'a pas de bonnes propriétés de complexité théoriques. Nous allons étudier une autre approche.

Question 4. Considérons les arbres de recherche suivants :



(a) Comment peut-on les transformer afin de placer le nœud 0 à la racine ?

(b) En déduire une variante de l'algorithme donné en Question 3. Est-elle équivalente ?

On définit maintenant la *taille* d'un arbre a, notée taille(a), comme étant le nombre de nœuds qu'il contient. Son *rang* $\mathrm{rg}(a)$ est défini par $\mathrm{rg}(a) = \log_2(\text{taille}(a))$. Enfin, le *potentiel* $\Phi(a)$ d'un arbre a est la somme des rangs de tous ses sous arbres.

Question 5.

(a) Soit t un arbre, et soit t' l'arbre résultant d'une rotation dans t (pas nécessairement à sa racine). De plus, soit t'_0 le sous-arbre de t' sur lequel la rotation a été effectuée, et t_0 le sous-arbre de t dont la racine a la même étiquette que celle de t'_0. Montrer que :

$$\Phi(t') - \Phi(t) \leq 3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0))$$

(b) Dans cette question, considérons les transformations de la question 4.a plutôt qu'une rotation. Montrer que :

$$\Phi(t') - \Phi(t) \leq 3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0)) - k$$

Où k est une constante entière qui dépend de la transformation et qu'il faudra calculer.

On pourra utiliser, en l'admettant, l'*inégalité arithmético-géométrique* :

$$\sqrt{ab} \leq \frac{a+b}{2}$$

quels que soient a et b deux réels positifs ou nuls.

(c) En déduire la complexité en temps de la variante de l'algorithme de la question 4.b, pour une séquence de requêtes.

Suite des questions

Question 6. Supposons maintenant que chaque étiquette x a un poids $w(x)$. Soit W la somme des poids de toutes les étiquettes de l'arbre. Montrer que la complexité en temps amortie de la question 5 devient $O\left(\log \frac{W}{w(x)}\right)$ lorsque l'on cherche l'étiquette x. Commenter.

Question 7. Proposer des fonctions d'insertion et de suppression similaires à la fonction de recherche de la question 4. Quelles sont leurs complexités amorties ?

Corrigé

Question 0. Un arbre binaire de recherche est un arbre binaire étiqueté par des éléments d'un ensemble totalement ordonné. Il vérifie la propriété suivante : si l'arbre contient un nœud `Nœud`(g, x, d) (de sous-arbres gauche et droit g et d, et d'étiquette x), alors toutes les étiquettes des nœuds de g sont plus petites que x, et toutes les étiquettes des nœuds de d sont plus grandes que x. Dans le contexte de cette épreuve, on ignore le cas où plusieurs nœuds pourraient avoir la même étiquette (c'est impossible pour les dictionnaires).

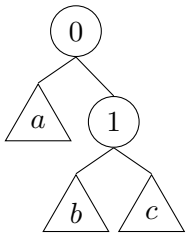
Les arbres binaires de recherche peuvent être utilisés pour implémenter une structure de donnée de dictionnaire, lorsque l'ensemble des clefs est totalement ordonné : chaque nœud de l'arbre est étiqueté par une entrée du dictionnaire, en ordonnant les entrées par l'ordre des clefs. Lorsque l'arbre n'est pas déséquilibré, les requêtes d'insertion, de suppression, et de lecture d'un nœud sont rapides, et permettent donc d'implémenter efficacement les requêtes correspondantes sur le dictionnaire.

La fonction d'insertion dans un arbre binaire de recherche peut se pseudo-coder comme suit :

```
Insérer(x, a) :=
  Si a = Vide
    Renvoyer Nœud(Vide, x, Vide)
  Si a = Nœud(g, y, d)
    Si x < y
      Renvoyer Nœud(Insérer(x, g), y, d)
    Si x > y
      Renvoyer Nœud(g, y, Insérer(x, d))
    Si x == y
        (* Dans le cas d'un dictionnaire, on peut imaginer que la fonction
           Insérer est utilisée pour modifier une entrée déjà existante. *)
      Renvoyer Nœud(g, x, d)
```

Sa complexité est asymptotiquement la profondeur du nœud inséré. Dans le pire cas, si l'arbre est complètement déséquilibré (c'est un "peigne"), la complexité est linéaire en fonction du nombre de nœuds de l'arbre. Par contre, si l'arbre est bien équilibré, la complexité est logarithmique, puisque la profondeur de l'arbre est logarithmique par rapport au nombre de nœuds.

Question 1. Après la rotation à droite, l'arbre est modifié comme suit :



Il s'agit, en fait, de la seule transformation possible qui ne modifie pas les sous-arbres a, b et c.
Cette opération est implémentée par le pseudo-code suivant :

```
Rotation_droite(n)
  Posons Nœud(Nœud(a, e0, b), e1, c) = n
  Renvoyer Nœud(a, e0, Nœud(b, e1, c))
```

On pourrait aussi donner un pseudo-code pour la rotation gauche, de manière symétrique.

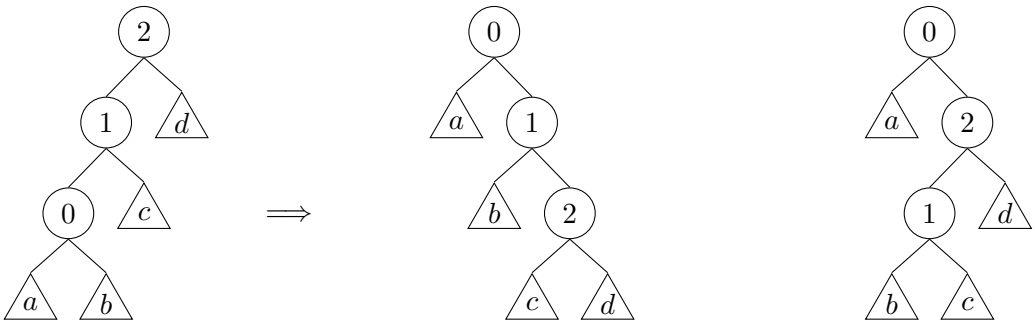
Question 2. L'algorithme procède comme pour une recherche dans un arbre binaire de recherche, mais effectue une rotation (droite ou gauche) après chaque appel récursif afin de placer à la racine le nœud demandé :

```
Recherche_et_évase(n, e)
  (* Puisqu'on suppose que l'étiquette e est dans l'arbre, n ne peut être vide *)
  Posons Nœud(g, e', d) = n
  Si e = e'
    Renvoyer n
  Si e < e'
    Renvoyer Rotation_droite(Nœud(Recherche_et_évase(g, e), e', d))
  Si e > e'
    Renvoyer Rotation_gauche(Nœud(g, e', Recherche_et_évase(d, e)))
```

Sa complexité est asymptotiquement la hauteur du nœud recherché. Dans le pire cas, si l'arbre est déséquilibré et que le nœud recherché est une feuille, la complexité est linéaire en la taille de l'arbre.

Question 3. L'idée est d'utiliser l'algorithme de la question 2 afin de répondre à la requête de recherche, mais aussi de modifier l'arbre binaire de recherche pour que le nœud qui vient d'être cherché se trouve à la racine. Ainsi, les nœuds fréquemment cherchés se trouvent proches de la racine et sont donc trouvés plus rapidement. Il s'agit là de l'idée proposée par Brian Allen et Ian Munro en 1978 [AM78].

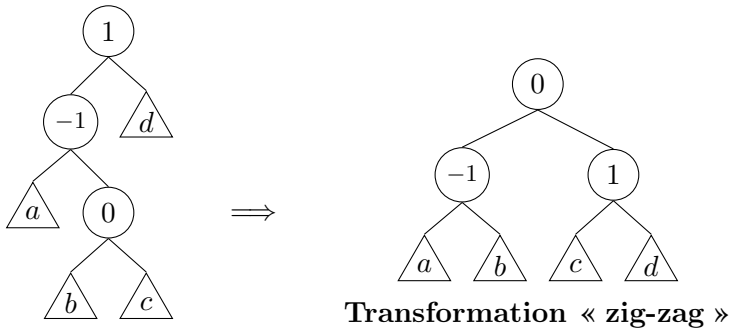
Question 4. Le premier arbre peut se transformer de deux façons différentes :



Transformation « zig-zig »

La deuxième transformation ci-dessus est celle qu'effectuerait l'algorithme de la Question 3. Par contre, la première est nouvelle : elle peut être obtenue en effectuant les deux rotations en procédant du haut vers le bas. Par la suite, on appellera cette transformation cette transformation *« zig-zig »*.

Le deuxième arbre, par contre, ne peut se transformer que de la façon suivante :



Transformation « zig-zag »

On appellera cette transformation *« zig-zag »*.

Bien sûr, 3 autres transformation symétriques existent pour des arbres dont la forme est symétrique aux deux arbres donnés dans l'énoncé. De manière similaire, on obtient alors les transformations *« zag-zig »* et *« zag-zag »*. On peut se alors se convaincre facilement qu'on a traité tous les cas possibles pour remonter un nœud de deux étages.

Il est à noter que, pour simplifier l'implémentation, toutes ces transformations peuvent s'obtenir en composant des rotations.

Pour obtenir des variantes de l'algorithme de la question 3, on peut donc utiliser une stratégie différente pour remonter le nœud en question à la racine. Plutôt que d'effectuer les rotations une par une de bas en haut, on peut les effectuer deux par deux, toujours de bas en haut, en appliquant les transformations *« zig-zig »*, *« zig-zag »*, *« zag-zig »* ou *« zag-zag »*, selon le cas pertinent. S'il y a un nombre impair de niveaux à remonter, il faut compléter ces transformations par une rotation gauche ou droite, à la racine par exemple.

On obtient alors l'opération d'évasement décrite par Sleator et al. [ST85]. Elle n'est pas équivalente à l'algorithme présenté dans la question 3, puisque, comme nous l'avons déjà évoqué, les transformations « zig-zig » et « zag-zag » n'effectuent pas les rotations de bas en haut comme dans la question 3. Comme nous allons le voir, cette opération d'évasement permet d'obtenir des garanties fortes de complexité amortie.

Question 5. Dans cette question, on utilisera les étiquettes de nœuds choisies dans les schémas des questions précédentes. Le sous-arbre t_0 correspond au sous-arbre dont la racine est le nœud 0, t_1 a le nœud 1 à la racine, etc...

Considérons d'abord le cas d'une rotation. La variation de potentiel n'est due qu'à la variation des rangs des sous-arbres enracinés en 0 et 1, puisque tous les autres sous-arbres gardent la même taille, et donc le même rang. On a :

$$\begin{aligned}
\Phi(t') - \Phi(t) &= \mathrm{rg}(t'_1) + \mathrm{rg}(t'_0) - \mathrm{rg}(t_1) - \mathrm{rg}(t_0) \\
&= \mathrm{rg}(t'_1) - \mathrm{rg}(t_0) && \text{car } \mathrm{rg}(t'_0) = \mathrm{rg}(t_1) \\
&\leq \mathrm{rg}(t'_0) - \mathrm{rg}(t_0) && \text{car } \mathrm{rg}(t'_1) \leq \mathrm{rg}(t'_0) \\
&\leq 3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0)) && \text{car } \mathrm{rg}(t'_0) = \mathrm{rg}(t_1) \geq \mathrm{rg}(t_0)
\end{aligned}$$

Considérons maintenant la transformation « zig-zig » de la question 4.

Indication : utiliser l'inégalité arithmético-géométrique pour majorer $(\mathrm{rg}(t'_2) + \mathrm{rg}(t_0))/2$.
On a :

$$\begin{aligned}
\Phi(t') - \Phi(t) &= \mathrm{rg}(t'_2) + \mathrm{rg}(t'_1) + \mathrm{rg}(t'_0) - \mathrm{rg}(t_2) - \mathrm{rg}(t_1) - \mathrm{rg}(t_0) \\
&= \mathrm{rg}(t'_2) + \mathrm{rg}(t'_1) - \mathrm{rg}(t_1) - \mathrm{rg}(t_0) && \text{car } \mathrm{rg}(t'_0) = \mathrm{rg}(t_2)
\end{aligned}$$

Que faire de ces 4 termes ?

— On peut utiliser l'inégalité arithmético-géométrique pour $\mathrm{rg}(t'_2)$:

$$\begin{aligned}
\frac{\mathrm{rg}(t'_2)+\mathrm{rg}(t_0)}{2} &= \log_2\left(\sqrt{\mathrm{taille}(t'_2)\cdot\mathrm{taille}(t_0)}\right) && \text{par définition}\\
&\leq \log_2\left(\frac{\mathrm{taille}(t'_2)+\mathrm{taille}(t_0)}{2}\right) && \text{inégalité arithmético-géométrique}\\
&\leq \log_2(\mathrm{taille}(t'_0))-1 && \text{en remarquant que } \mathrm{taille}(t'_2)+\mathrm{taille}(t_0)\leq\mathrm{taille}(t'_0)
\end{aligned}$$

En réordonnant les termes, on obtient $\mathrm{rg}(t'_2)\leq 2\mathrm{rg}(t'_0)-\mathrm{rg}(t_0)-2$.

— On a facilement $\mathrm{rg}(t'_1)\leq\mathrm{rg}(t'_0)$.

— De manière similaire, $-\mathrm{rg}(t_1)\leq-\mathrm{rg}(t_0)$.

— On conserve $-\mathrm{rg}(t_0)$.

En combinant toutes ces inégalités, on obtient $\Phi(t')-\Phi(t)\leq 3(\mathrm{rg}(t'_0)-\mathrm{rg}(t_0))-2$.

On considère maintenant le cas de la transformation « zig-zag ».

Indication : utiliser l'inégalité arithmético-géométrique pour majorer $(\mathrm{rg}(t'_{-1})+\mathrm{rg}(t'_1))/2$.

De manière similaire à la transformation « zig-zig », on a :

$$\begin{aligned}
\Phi(t')-\Phi(t) &= \mathrm{rg}(t'_1)+\mathrm{rg}(t'_{-1})+\mathrm{rg}(t'_0)-\mathrm{rg}(t_1)-\mathrm{rg}(t_{-1})-\mathrm{rg}(t_0)\\
&= \mathrm{rg}(t'_1)+\mathrm{rg}(t'_{-1})-\mathrm{rg}(t_{-1})-\mathrm{rg}(t_0) && \text{car } \mathrm{rg}(t'_0)=\mathrm{rg}(t_1)\\
&\leq 2\mathrm{rg}(t'_0)-2-\mathrm{rg}(t_{-1})-\mathrm{rg}(t_0) && \text{par inégalité arithmético-géométrique}\\
&\leq 2(\mathrm{rg}(t'_0)-\mathrm{rg}(t_0))-2 && \text{car } \mathrm{rg}(t_{-1})\geq\mathrm{rg}(t_0)\\
&\leq 3(\mathrm{rg}(t'_0)-\mathrm{rg}(t_0))-2 && \text{car } \mathrm{rg}(t'_0)=\mathrm{rg}(t_1)\geq\mathrm{rg}(t_0)
\end{aligned}$$

Pour la dernière transformation évoquée à la question 4 (à laquelle nous n'avons pas donné de nom), on peut simplement se rappeler qu'elle s'obtient comme composition de deux rotations sur le nœud 0.

Indication : réutiliser le résultat sur les rotations. Ne pas passer beaucoup de temps sur cette transformation inutile.

En utilisant deux fois l'inégalité obtenue pour les rotations, on remarque qu'elles se télescopent, et on obtient :

$$\Phi(t')-\Phi(t)\leq\mathrm{rg}(t'_0)-\mathrm{rg}(t_0)\leq 3(\mathrm{rg}(t'_0)-\mathrm{rg}(t_0))$$

La complexité d'une séquence de requêtes à l'algorithme de la question 4.a peut être évaluée en majorant la variation de potentiel d'une requête.

Pendant une telle requête, on fait des transformations dont on a déjà majoré la variation de potentiel avec les inégalités ci-dessus. Ces transformations sont de deux types :

— Les transformation « zig-zig », « zig-zag », « zag-zig » et « zag-zag » permettent de remonter le nœud d'intérêt de deux niveaux. La variation de potentiel associée est majorée par $3(\mathrm{rg}(t'_0)-\mathrm{rg}(t_0))-2$ (on a établi le résultat ci-dessus pour deux de ces transformations ; les deux autres sont symétriques). Si le nœud d'intérêt a initialement une profondeur de p, alors on utilise $\left\lfloor\frac{p}{2}\right\rfloor$ telles transformations.

— Éventuellement, si la profondeur du nœud est impaire, au plus une rotation (à gauche ou à droite) permet de finir le travail. Dans ce dernier cas, la variation de potentiel due à cette transformation est majorée par $3(\mathrm{rg}(t'_0) - \mathrm{rg}(t_0))$.

On peut alors combiner les majorations obtenues, qui se télescopent, et obtenir :

$$\Phi(t') - \Phi(t) \leq 3(\mathrm{rg}(t') - \mathrm{rg}(t_0)) - 2\left\lfloor \frac{p}{2} \right\rfloor \leq 3\log_2 N - p + 1$$

en notant :

— t et t' les arbres avant et après une requête, respectivement ;

— t_0 le sous-arbre de t dont la racine est le nœud recherché ;

— p la profondeur du nœud recherché dans t ;

— N la taille de t (ou de t').

Ainsi, pour une séquence de M requêtes de profondeurs $p_1, \ldots, p_M$, on peut utiliser l'inégalité ci-dessus et télescoper les valeurs de Φ intermédiaires. On obtient :

$$\Phi(t') - \Phi(t) \leq M(3\log_2 N + 1) - \sum_{i=1}^{M} p_i$$

D'où :

$$\sum_{i=1}^{M} p_i \leq M(3\log_2 N + 1) - \Phi(t') + \Phi(t) = O(M\log N) + O(N\log N)$$

Or, $\sum_{i=1}^{M} p_i$ est asymptotiquement le temps d'exécution des M requêtes. En particulier, si on considère un grand nombre de requêtes, le terme $O(N\log N)$ devient négligeable. Donc, en complexité amortie, le coût d'une requête est $O(\log N)$.

Question 6. On modifie les notions de rang et de potentiel pour utiliser les poids donnés aux étiquettes. En particulier, on définit le rang d'un sous-arbre comme étant le logarithme de la somme des poids des étiquettes qu'il contient :

$$\mathrm{rg}(t) = \sum_{x \in t} w(x)$$

On note W la somme de tous les poids de l'arbre.

De la même façon qu'à la question 5, on obtient l'inégalité suivante pour la variation du potentiel due à une requête de recherche de l'étiquette x :

$$\Phi(t') - \Phi(t) \leq 3(\mathrm{rg}(t') - \mathrm{rg}(t_0)) - p + 1 \leq 3(\log_2 W - \log_2 w(x)) - p + 1$$

Puis, pour une séquence de requêtes des étiquettes $x_1, \ldots, x_M$ aux profondeurs $p_1, \ldots, p_m$:

$$\Phi(t') - \Phi(t) \leq M + 3\sum_{i=1}^{M} \log_2 \frac{W}{w(x_i)} - \sum_{i=1}^{M} p_i$$

Donc le coût d'une séquence de M requêtes est asymptotiquement :

$$O\left(M + \sum_{i=1}^{M} \log_2 \frac{W}{w(x_i)}\right) + O\left(\sum_{x \in t} \log_2 \frac{W}{w(x)}\right)$$

Pour un grand nombre de requêtes, le second terme devient négligeable. Ainsi, le coût amorti d'une requête pour l'étiquette x est $O\left(\log_2 \frac{W}{w(x)}\right)$.

On peut voir ainsi voir que l'arbre s'adapte de manière automatique à la distribution des requêtes : si une requête est plus fréquente, alors son coût sera moins important. En pratique, si un nœud est accédé plus fréquemment, les transformations de l'arbre le déplaceront régulièrement en haut de l'arbre, ce qui rendra son accès plus rapide.

Question 7. Il y a plusieurs solutions pour l'insertion et la suppression. Voici une possibilité.

Pour l'insertion, on peut commencer par remonter à la racine le nœud immédiatement inférieur au nœud à insérer en coût amorti $O(\log N)$, puis il est facile d'insérer le nouveau nœud à la racine en temps constant. La variation de potentiel due à cette dernière opération est de $O(\log N)$, donc le coût total de l'opération est $O(\log N)$.

Évidemment, cette opération ne peut se faire si aucun nœud n'est inférieur au nœud à insérer. Mais dans ce cas, le problème est trivial : on peut simplement insérer le nouveau nœud à la racine.

Pour la suppression, on peut commencer par remonter à la racine le nœud à supprimer, puis le supprimer. La première opération peut s'effectuer en temps amorti $O(\log N)$. La suppression seule peut se faire en temps constant. La variation de potentiel étant négative, on peut ne pas la comptabiliser.

Il reste alors à fusionner les deux sous-arbres restants, dont on sait que les clefs de l'un sont toutes inférieures aux clefs de l'autre. Si l'un des deux arbres est vide, c'est trivial et immédiat. Sinon, un moyen simple pour effectuer cette fusion est de monter à la racine du sous-arbre inférieur son nœud minimal, en temps amorti $O(\log N)$. Après cet opération, la racine de l'arbre inférieur ne peut pas avoir de fils droit. On peut donc y placer l'autre sous-arbre. La variation de potentiel due à cette dernière opération (ajouter le sous-arbre supérieur comme fils droit de la racine de l'autre) peut se faire en temps constant, et la variation de potentiel associée n'est due qu'à la variation de rang de la racine, qui est $O(\log N)$. Au total, la suppression peut se faire en temps amorti $O(\log N)$.

Références

[AM78] Brian Allen and Ian Munro. Self-organizing binary search trees. *Journal of the ACM (JACM)*, 25(4) :526–535, 1978. Non disponible en libre accès.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3) :652–686, 1985. `https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`.

J2 – Paresse et file persistante

Question 0.

(a) Qu'est-ce qu'une file ?

(b) Rappeler la distinction entre structure de donnée persistante et impérative.

(c) Donner une implémentation *persistante* d'une file.

(d) En déduire une implémentation *impérative* d'une file.

(e) Dans le pire cas, quelle est la complexité de chacune des opérations des ces deux implémentations ?

Lorsque l'on analyse la *complexité amortie* d'une bibliothèque, on s'intéresse à la complexité d'une séquence d'opérations dans son ensemble plutôt qu'à la complexité de chaque opération fournie par la bibliothèque. Ainsi, même si une opération A est très coûteuse, son coût peut être compensé par l'exécution préalable d'un grand nombre d'opérations B, de façon à ce que la complexité globale de la séquence d'opérations soit asymptotiquement le même que si A était peu coûteuse.

Question 1.

(a) Faire l'analyse de complexité amortie de l'implémentation impérative de la question 0.

(b) Ce raisonnement peut-il s'appliquer pour l'implémentation persistante ?

On se propose d'implémenter, en OCaml, une bibliothèque de *calcul paresseux*. Celle-ci expose un type paramétré de *suspensions* `'a susp` et des fonctions de types suivants :

```
susp : (unit -> 'a) -> 'a susp
force : 'a susp -> 'a
```

Une suspension (de type `'a susp`) contient une fonction permettant de calculer une valeur de type `'a`. Elle peut être construite facilement grâce à la fonction `susp`. Le calcul n'est effectué que lorsque l'utilisateur de la bibliothèque le demande via la fonction `force`. Cette dernière fonction vérifie si le calcul a déjà été effectué : si tel est le cas, elle en renvoie le résultat pré-calculé. Sinon, elle lance le calcul, stocke le résultat pour de futurs appels, et elle le renvoie.

Question 2. Donner une implémentation possible de cette bibliothèque, dans le langage OCaml.

On définit le type des *listes paresseuses* en OCaml :

```
type 'a slist_cell =
| SNil
| SCons of 'a * 'a slist
and 'a slist = 'a slist_cell susp
```

Question 3.

(a) Comparer la notion de liste paresseuse avec la notion habituelle de liste.

(b) Écrire une fonction `scons : 'a -> 'a slist -> 'a slist` qui ajoute un élément en tête d'une liste paresseuse, ainsi qu'une valeur `snil` de liste paresseuse vide.

(c) Écrire deux fonctions `shd : 'a slist -> 'a` et `stl : 'a slist -> 'a slist` qui prennent une liste paresseuse non vide en paramètre, et qui renvoient respectivement son premier élément et la liste paresseuse des autres éléments.

(d) Écrire une fonction `sappend : 'a slist -> 'a slist -> 'a slist` qui concatène *de manière paresseuse* deux listes paresseuses. Cette fonction devra s'exécuter en temps constant.

(e) Écrire une fonction `srev : 'a slist -> 'a slist` qui renverse une liste paresseuse de manière efficace. Quelle est la complexité des accès aux différents éléments de la liste renversée ?

Grâce aux listes paresseuses, on propose ici une variante de la file proposée dans la question 0.c, qui évite le problème de complexité expliqué dans la question 0.e. Dans cette nouvelle implémentation, une file sera représenté en OCaml par le type enregistrement suivant :

```
type 'a queue = {
  rear : 'a slist;
  len_rear : int;
  front : 'a slist;
  len_front : int;
}
```

Le plus souvent, on enfilera les éléments au début de la liste `rear` et on les défilera au début de la liste `front`. De plus, on maintiendra les invariants suivants :

— les champs `len_front` et `len_rear` contiennent les longueurs des listes `front` et `rear` ;

— on a toujours `len_rear` $\leq$ `len_front`.

Question 4.

(a) Définir les fonctions d'enfilage et de défilage pour cette variante d'implémentation de file.

(b) Prouver que le temps d'exécution *amorti* de chacune de ces deux opérations est $O(1)$.

Suite des questions

Question 5. Une liste paresseuse est-elle toujours de longueur finie ? Définir en OCaml une liste paresseuse qui énumère les carrés parfaits.

Corrigé

Question 0. (a) Une file est une structure de donnée représentant une séquence d'éléments, et qui permet d'ajouter un élément au début de la séquence, et de récupérer en supprimant un élément à la fin de la séquence.

(b) Dans une structure de donnée impérative, les fonctions modifient la structure de donnée en mémoire, alors qu'une structure de donnée persistante renvoie une nouvelle version de la structure de donnée sans modifier (de manière observable) celle qui a été donnée en paramètre.

(c) On peut implémenter de manière persistante une file avec deux listes. On enfile toujours au début de la première liste, et on défile au début de la seconde. Si la seconde file est vide lorsque l'on veut défiler, alors on renverse la première liste pour lui faire prendre la place de la seconde (dans ce code, on a mis toutes les annotations de type, mais elles sont bien sûr optionnelles) :

```
type 'a queue = 'a list * 'a list

let vide : 'a queue = ([], [])

let push (q : 'a queue) (x : 'a) : 'a queue =
  (x::fst q, snd q)

let pop (q : 'a queue) : 'a * 'a queue =
  match snd q with
  | x::q2 -> (x, (fst q2, q2))
  | [] ->
    match List.rev (fst q) with
    | x::q2 -> (x, ([], q2))
    | [] -> assert false  (* La file est vide, erreur *)
```

(d) On peut simplement mettre la file persistante dans une référence. On obtient alors une file impérative :

```
type 'a queue' = 'a queue ref

let vide' () : 'a queue' =
  ref vide

let push' (q : 'a queue') (x : 'a) : unit =
  q := push !q x

let pop' (q : 'a queue') : 'a =
  let (x, qp) = pop !q in
  q := qp;
  x
```

(e) La création d'une nouvelle file est toujours en temps constant. L'opération `push` est aussi en temps constant. Par contre, l'opération `pop` est, au pire cas, en temps $O(N)$, où N est le nombre d'éléments dans la file. En effet, s'il faut renverser la première file, cela prend un temps $O(N)$.

Question 1. (a) Dans une séquence d'opérations `push` et `pop` sur une file, le retournement de liste ne traitera un élément donné qu'une seule fois. Ainsi, le coût global des retournements ne dépassera pas asymptotiquement le nombre d'éléments enfilés. Si on attribue le coût des retournements à l'enfilage plutôt qu'au défilage, la complexité amortie des opérations `push` et `pop` est donc $O(1)$, *dans l'implémentation impérative.*

(b) Dans le cas de l'implémentation persistante, il n'est plus vrai que chaque élément enfilé n'est traité qu'une seule fois par le retournement. En effet, il est très facile de copier une file persistante. Ainsi, si on commence par enfiler N éléments, puis qu'on effectue M copies, et que pour chacune de ces copies, on effectue un défilage, alors on va renverser une liste de taille N une fois pour chaque copie (soit M fois au total). La complexité globale serait alors $O(NM)$ alors qu'on a effectué que $O(N)$ enfilages et $O(M)$ défilages.

Question 2. Une suspension peut avoir deux états : en attente de calcul ou calculé. Cet état est modifié de manière impérative lors du premier appel à `force` la concernant. Il est donc naturel d'utiliser un type somme à deux alternatives pour représenter une suspension. De plus, pour permettre le changement d'état, on va utiliser une référence :

```
type 'a susp_state =
| Computed of 'a
| Suspended of (unit -> 'a)

type 'a susp = 'a susp_state ref
```

Pour la fonction `susp_state`, on utilise simplement le constructeur `Suspended` dans une référence fraîche :

```
let susp f = ref (Suspended f)
```

Enfin, la fonction `force` peut être implémentée facilement en distinguant les deux cas :

```
let force p =
  match !p with
  | Computed x -> x
  | Suspended f ->
    let x = f () in
    p := Computed x;
    x
```

Une autre possibilité est d'utiliser simplement une référence sur une fonction : après avoir évalué la suspension, on modifie la référence pour qu'elle contienne la fonction qui renvoie immédiatement la valeur déjà calculée. On obtient alors le code suivant :

```
type 'a susp = (unit -> 'a) ref

let susp f = ref f

let force p =
  let x = !p () in
  p := (fun () -> x);
  x
```

Question 3. (a) Dans la notion de liste habituelle, les éléments de la liste sont présents en mémoire dès que la liste existe. Au contraire, pour une liste paresseuse, ils sont calculés à la demande, lorsque l'on accède aux éléments de la liste. De manière intéressante, la structure de liste paresseuse peut être utilisée pour représenter des séquences infinies d'éléments, en ne stockant pas explicitement les éléments qui ne sont pas encore accédés : on stocke plutôt une fonction qui permettra, le moment venu, de calculer plus d'éléments de cette liste.

(b)

```
let snil = susp (fun () -> SNil)
let scons x l = susp (fun () -> SCons (x, l))
```

(c)

```
let shd x =
  match force x with
  | SCons (hd, _) -> hd
  | SNil -> assert false

let stl x =
  match force x with
  | SCons (_, tl) -> tl
  | SNil -> assert false
```

(d) Une possibilité serait d'utiliser une fonction récursive qui itère les éléments de la première liste afin de l'ajouter à la liste paresseuse un à un avec la fonction `scons` ci-dessus. Malheureusement, cela requiert d'accéder à tous les éléments de la première liste pendant l'appel à `sappend`. Ceci ne peut s'effectuer en temps constant.

On utilise donc une stratégie différente, qui accède à la première liste à la demande, lorsque l'accès aux éléments de la liste renvoyée est demandé :

```
let rec sappend l1 l2 =
  susp (fun () ->
    match force l1 with
    | SCons (t, q) -> SCons (t, sappend q l2)
    | SNil -> force l2)
```

Cette fonction termine bien en $O(1)$, puisque la fonction `susp` termine en $O(1)$.

(e) Comme pour une liste non paresseuse, on peut utiliser un algorithme inefficace qui ajoute, un par un, les éléments de la liste d'entrée à la fin de la liste résultat. Cependant, comme pour une liste non paresseuse, on peut faire plus efficace, en utilisant un accumulateur qui contient la liste partiellement renversée :

```
let srev l =
  let rec aux acc l =
    match force l with
    | SCons (t, q) -> aux (scons t acc) q
    | SNil -> acc
  in
  susp (fun () -> force (aux snil l))
```

Au contraire de la fonction `sappend`, la fonction `srev` a besoin de parcourir en entier la liste passée en paramètre, dès que le premier élément de la liste donnée en entrée est demandé. Donc :

— lorsque la fonction `srev` est appelée, son coût "direct" est $O(1)$, puisqu'il ne s'agit que d'un appel à `susp` ;

— lorsque le premier élément est demandé, le coût est le coût de l'évaluation de la liste d'entrée plus $O(N)$, où N est la longueur de la liste ;

— lorsque les éléments suivants sont demandés, leur coût est de $O(1)$, puisque la suspension correspondante dans le code est `scons t acc`.

Question 4. Il s'agit de la *file du banquier* proposée par Okasaki [Oka99, §3.4.2].

(a) Aussi bien lors du défilage que de l'enfilage, l'invariant `len_rear` $\leq$ `len_front` peut être violé par la modification de la seule liste concernée (`rear` pour l'enfilage, `front` pour le défilage). Pour rétablir cet invariant, il faut, comme dans le cas de la file présentée dans la question 0.c, renverser la liste `rear` pour la placer à la fin de la liste `front`. On commence par écrire une fonction qui effectue cette opération de rétablissement d'invariant, si nécessaire :

```
let refresh q =
  if q.len_rear <= q.len_front then q
  else
    { rear = snil;
      len_rear = 0;
      front = sappend q.front (srev q.rear);
      len_front = q.len_front + q.len_rear }
```

On peut alors facilement écrire les fonctions d'enfilage et de défilage :

```
let push q x =
  let q = { q with rear = scons x q.rear; len_rear = q.len_rear + 1 } in
  refresh q

let pop q =
  (shd q.front, refresh { q with front = stl q.front; len_front = q.len_front - 1 })
```

(b) Le piège dans l'analyse de complexité de cette implémentation serait d'oublier de compter le coût d'évaluation des suspensions. En effet, la fonction `refresh` termine toujours en temps $O(1)$ puisqu'elle ne fait que reporter le renversement et la concaténation des listes. Du coup, la fonction `push` termine aussi en temps constant, et, si on oubliait de compter l'évaluation de la suspension qui a lieu lors de l'appel à `shd`, la fonction `pop` terminerait aussi en temps $O(1)$ (rappelons-nous que l'appel à `stl` réutilise l'évaluation de la suspension de `shd`).

Donc, pour faire l'analyse de complexité amortie de cette implémentation, il faut garder trace des coûts des différentes suspensions présentes dans la structure de données (ou qu'il est prévu de calculer). Par exemple, le coût d'une suspension créée par `susp` est $O(1)$. Le coût de la suspension correspondant à la première cellule d'une liste de longueur N renvoyée par `srev` est $O(N)$ plus le coût de l'évaluation de toutes les cellules de la liste initiale. Le coût des suspensions suivantes est $O(1)$.

Enfin, si on concatène deux listes de tailles N et M, le coût des suspensions renvoyées par `sappend` sont :

— une constante plus le coût d'évaluation des suspensions correspondantes de la première liste pour les N premières suspensions,

— le coût d'évaluation des suspensions correspondantes de la seconde liste pour les M suivantes.

Indication 1 : Une remarque cruciale est qu'on peut payer le coût d'une suspension *à l'avance*, avant que celle-ci ne soit évaluée. Ceci repose sur le fait qu'une suspension n'est évaluée qu'une seule fois : si une suspension est copiée puis évaluée deux fois, son coût n'a besoin d'être payé qu'une seule fois.

Indication 2 : On peut aller plus loin que la remarque précédente : lorsqu'une suspension est une partie du résultat du calcul d'une autre suspension (typiquement dans le cas d'une liste paresseuse), on peut déplacer le coût de la suspension emboîtée à la suspension qui va la calculer. Plus précisément, s'il est prévu qu'une suspension A renvoie à la suite de son calcul une suspension B, alors on peut décider de payer (une partie) du coût de la suspension B en tant que coût de la suspension A. Ainsi, si A a un coût c_A et B un coût c_B, alors on peut considérer que A a un coût $c_A + x$ et B un coût $c_B - x$ pour tout $x \geq 0$. Bien sûr, cela repose aussi sur le fait qu'une suspension n'est évaluée qu'une seule fois : si A était évaluée deux fois, alors chaque calcul renverrait une suspension B différente, et on ne pourrait pas payer leur coût en avance dans A.

Ainsi, quitte à payer le coût $O(1)$ à l'avance au moment de l'appel des fonctions, on peut considérer que le coût de la suspension renvoyée par `scons` ou par `snil` est nul. Puisque `rear` n'est alimenté qu'avec `scons` et `snil`, on peut donc considérer que le coût des suspensions de la liste `rear` est nul.

Comment payer le coût des suspensions de la liste `front` ? On va prouver ici, que, quitte à déplacer les coûts des suspensions comme expliqué dans les remarques ci-dessus, le coût des `len_front - len_rear` premières suspensions de la liste `front` est $O(1)$, alors que les `len_rear` dernières suspensions sont gratuites. Il sera alors facile de conclure que `push` et `pop` sont en temps constant.

Pour prouver que cet invariant tient, on commence par remarquer qu'il est vrai trivialement lors de l'initialisation de la queue.

Lorsque l'on retire un élément avec `pop` sans rééquilibrer les listes, on paie simplement le coût $O(1)$ de la suspension que l'on évalue au début de la liste `front`. L'invariant est donc maintenu.

Lorsque l'on insère un élément avec `push` sans rééquilibrer les listes, il faut payer le coût $O(1)$ de la dernière suspension non gratuite (à la position `len_front - len_rear - 1`). Grâce aux deux remarques ci-dessus, on peut le faire par anticipation, même si cette suspension est au beau milieu de la liste `front`. On peut donc aussi maintenir l'invariant sur les coûts.

Finalement, lorsqu'un appel à `push` ou `pop` rééquilibre les deux listes, on a `len_rear = len_front` au début de l'opération. Du coup, au moment de l'appel à `refresh`, toutes les suspensions des deux listes sont gratuites. Après l'appel à `refresh`, les suspensions de la première moitié de la liste coûtent $O(1)$ (à cause de l'appel à `sappend`), la suspension suivante coûte $O($`len_front`$)$ (à cause de l'appel à `srev`), et les suspensions du reste de la liste coûtent toutes $O(1)$ (encore à cause de l'appel à `srev`). En répartissant sur les suspensions de la première moitié de la liste le coût $O($`len_front`$)$ de la suspension centrale (avec la deuxième remarque ci-dessus), on peut effectivement aboutir à un coût de $O(1)$ pour toutes les suspensions de la liste, comme le demande l'invariant.

Question 5. Puisqu'elle n'est jamais représentée directement en mémoire, une liste paresseuse n'est pas nécessairement de longueur finie. Elle peut représenter une séquence infinie d'éléments.

On peut définir la séquence des carrés parfaits de la façon suivante :

```
let carres =
  let rec aux i =
    susp (fun () -> SCons (i*i, aux (i + 1)))
  in
  aux 0
```

Références

[Oka99] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

J3 – B-arbres

Question 0.

(a) Rappeler ce qu'est un arbre binaire de recherche.

(b) Généraliser cette notion pour permettre aux nœuds de l'arbre d'avoir plus de deux fils tout en permettant des recherches d'éléments efficaces.

(c) Dans le cas d'une grande quantité d'information, il est parfois nécessaire d'utiliser des supports de stockage dont le temps de réponse pour une lecture est élevé (disque dur, ...). Expliquer alors l'intérêt de la structure de donnée décrite dans la question (b).

Soit un entier $m \geq 3$. On appelle B-arbre un arbre tel que décrit à la question 0.c, et vérifiant de plus les invariants suivants :

— toutes les feuilles ont la même profondeur ;

— le nombre d'étiquettes de tout nœud est au plus $m - 1$;

— le nombre d'étiquettes de tout nœud non racine est au moins $\lceil \frac{m}{2} \rceil - 1$;

— la racine a au moins une étiquette.

Question 1. Donner le pseudo-code d'un algorithme de recherche *efficace* dans un B-arbre et en donner la complexité en temps en fonction de m et du nombre d'étiquettes stockées dans l'arbre.

Question 2.

(a) Proposer le pseudo-code d'un algorithme d'insertion d'une nouvelle entrée dans un B-arbre. Quelle est sa complexité en temps dans le pire cas ?

(b) Si on utilise cet algorithme pour insérer des étiquettes dans l'ordre croissant, combien d'étiquettes un nœud de l'arbre résultant contiendra-t-il, typiquement ?

(c) Proposer une variante de l'algorithme d'insertion qui permet un meilleur remplissage des nœuds dans le scénario de la question précédente. Quelle est sa complexité en temps dans le pire cas ?

Question 3. Proposer un algorithme de suppression d'une entrée dans un B-arbre. Quelle est sa complexité en temps ?

Question 4. Proposer un algorithme qui prend en paramètre deux B-arbres T_1 et T_2 tels que les clefs de T_1 sont toutes inférieures à celles de T_2, et qui renvoie un B-arbre contenant les étiquettes de T_1 et de T_2. Quelle est sa complexité en temps ?

Suite des questions

Question 5. Proposer une variante des algorithmes d'insertion et de suppression dans un B-arbre tels que le nombre de fils d'un nœud non racine est compris entre $\lceil \frac{2m-1}{3} \rceil$ (inclus) et m (inclus). Quelle contrainte sur le nombre de fils de la racine doit-on maintenant avoir ?

Question 6. Donner le pseudo-code des fonctions de la question 2.c, 3 et 4.

Corrigé

Question 0. (a) Un arbre binaire de recherche est un arbre binaire étiqueté par des éléments d'un ensemble totalement ordonné. Il vérifie la propriété suivante : si l'arbre contient un nœud `Nœud`(g, x, d) (de sous-arbres gauche et droit g et d, et d'étiquette x), alors toutes les étiquettes des nœuds de g sont plus petites que x, et toutes les étiquettes des nœuds de d sont plus grandes que x. Dans le contexte de cette épreuve, on ignore le cas où plusieurs étiquettes stockées dans un même arbre pourraient être identiques (c'est impossible pour les dictionnaires).

(b) Si un nœud d'un tel arbre a plus de deux fils, alors il est naturel de demander que les étiquettes d'un fils soient toutes plus petites que celles de ses frères droits, et toutes plus grandes que celles de ses frères gauches.

Cependant, cela ne permet pas d'avoir un algorithme de recherche efficace : en effet, pour chercher une étiquette dans un arbre, il faut pouvoir déterminer facilement dans quel sous-arbre celle-ci devrait se trouver. Afin de résoudre ce problème et de permettre une recherche efficace, si un nœud a n fils, alors on stocke dans chaque nœud $n-1$ étiquettes qui permettent de « séparer » les sous-arbres. Si l'arbre est utilisé pour représenter un dictionnaire, alors on stocke aussi les valeurs associées à ces étiquettes dans les nœuds. Ainsi, si les sous-arbres sont $T_0, \ldots, T_{n-1}$ et les étiquettes stockées dans le nœud $x_0, \ldots, x_{n-2}$, alors que l'étiquette x_i soit plus grande que toutes les étiquettes du sous-arbre T_i et plus petite que toutes les étiquettes du sous-arbre T_{i+1}. On voit que, lorsque $n = 2$, cela correspond exactement au cas d'un arbre binaire de recherche. Il est alors facile, en regardant simplement le contenu d'un nœud, de savoir dans quel sous-arbre il faut continuer la recherche.

(c) Dans un arbre binaire de recherche, on doit faire un accès à la mémoire (donc au disque dur, si l'arbre est stocké dans un disque dur) pour chaque nœud menant au nœud recherché. Si l'arbre est bien équilibré, cela représente à peu près $\log_2 N$ accès mémoire.

Par contre, si le degré de branchement de l'arbre est plus élevé, on peut faire beaucoup moins d'accès. Par exemple, si chaque nœud interne a d fils, alors il faut faire environ $\log_d N$ accès mémoire. Si, par exemple, $d = 128$, cela représente environ 7 fois moins d'accès mémoire que pour un branchement binaire.

Bien sûr, cela vient avec une contrepartie : les nœuds sont plus gros, et il faut donc lire plus d'information à la fois pour chaque accès mémoire. Et, pour la même raison, la modification des nœuds est plus coûteuse. Il y a donc un compromis à chercher.

Question 1. On suppose qu'un nœud est représenté par quatre informations :

- le nombre de ses sous-arbres,
- un tableau de ses sous-arbres,
- un tableau des étiquettes qu'il contient,
- un tableau des valeurs associées à ces étiquettes.

On effectue la recherche dans un B-arbre de la façon suivante :

```
Fonction Cherche(x, nœud)
  i = Cherche_premier_supérieur(x, nœud.nombre_fils-1, nœud.étiquettes)
```

```
  Si i < nœud.nombre_fils-1 ET nœud.étiquettes[i] == x
    Renvoyer nœud.valeurs[i]
  Renvoyer Cherche(x, nœud.fils[i])
```

Où la fonction `Cherche_premier_supérieur(x, n, t)` prend en paramètre une étiquette `x`, un tableau trié `t` d'étiquettes et sa taille `n` et cherche dans `t` la première cellule dont le contenu est supérieur ou égal à `x`. Si une telle cellule est trouvée, alors cette fonction renvoie son indice ; sinon, elle renvoie `n`.

Naïvement, on pourrait implémenter la fonction `Cherche_premier_supérieur` avec une simple boucle :

```
Fonction Cherche_premier_supérieur(x, n, t)
  Pour i dans [0, n[
    Si t[i] >= x
      Renvoyer i
  Renvoyer n
```

Pour cette question, cette implémentation n'est pas suffisamment efficace. Mais on peut aussi utiliser une recherche dichotomique pour plus d'efficacité lorsque m est grand :

```
Fonction Cherche_premier_supérieur(x, n, t)
  a = 0, b = n
  Tant que b-a >= 1
    Si t[(a+b)/2] > x
      b = (a+b)/2
    Sinon
      a = (a+b)/2+1
  Renvoyer a
```

On obtient alors une complexité en temps de $O(p \log m)$, où p est la hauteur du nœud recherché. On sait que p est bornée par la profondeur (commune) des feuilles. On peut prouver facilement, par ailleurs, qu'un arbre dont les nœuds internes ont au moins $\lceil \frac{m}{2} \rceil$ fils et dont les feuilles ont toute une profondeur q a plus de $\lceil \frac{m}{2} \rceil^q$ nœuds et donc plus de $\lceil \frac{m}{2} \rceil^q$ entrées. On en déduit que la profondeur d'un arbre avec N entrées est inférieure à $\log_m N$, et que la complexité de la recherche est $O(\log_m N \log m) = O(\log N)$.

Cette complexité est apparemment indépendante du paramètre m. Ceci est en fait trompeur, parce que dans une implémentation pratique utilisant une mémoire à forte latence, l'accès à un nœud nécessite une lecture du nœud en entier, et donc un coût $O(m)$.

Question 2. (a) Tout comme dans un arbre binaire de recherche, on va essayer d'insérer la nouvelle entrée en bas de l'arbre. Cependant, au lieu de créer un nouveau nœud, on va plutôt essayer de l'ajouter à un nœud existant. On commence donc par rechercher le nœud feuille dans lequel il faut insérer la nouvelle entrée, puis on l'y insère.

Si le nœud ne contient pas trop d'étiquettes, c'est fini, l'arbre vérifie les invariants voulus.

Si le nœud contient alors trop d'étiquettes, il faut alors le diviser en deux nouveaux nœuds et remplacer donc l'ancien nœud par les deux nouveaux dans son parent, qui contiendra donc aussi une nouvelle étiquette. En pratique, dans ce cas, l'ancien nœud contient exactement m étiquettes $x_0, \ldots, x_{m-1}$. On sépare cette séquence d'étiquettes en deux moitiés et une étiquette séparatrice : on peut prendre $x_0, \ldots, x_{\lceil \frac{m}{2} \rceil - 2}$ pour la première moitié, $x_{\lceil \frac{m}{2} \rceil - 1}$ pour l'étiquette séparatrice, et $x_{\lceil \frac{m}{2} \rceil}, \ldots, x_{m-1}$ pour la seconde moitié. Les deux moitiés, de tailles $\lceil \frac{m}{2} \rceil - 1$ et $\lfloor \frac{m}{2} \rfloor$, peuvent alors être stockée dans deux nouveaux nœuds. Il faut alors remplacer l'ancien nœud par ces deux nouveaux nœuds, et utiliser le séparateur $x_{\lceil \frac{m}{2} \rceil - 1}$ comme nouvelle étiquette pour le parent. À son tour, le parent peut contenir trop d'étiquettes. Il faut alors répéter l'opération sur le parent, et ainsi de suite jusqu'à la racine. Si la racine

doit elle-même être divisée en deux, on crée une nouvelle racine qui ne contiendra qu'une étiquette (ainsi, la hauteur de l'arbre augmentera).

Pour résumer, on peut utiliser le pseudo-code suivant :

```
Fonction Insere_aux(x, v, nœud)
  Si nœud = Feuille
    Renvoyer NouveauSousArbre(x, v, Feuille)
  Sinon
    i = Fonction Cherche_premier_supérieur(x, nœud.nombre_fils, nœud.étiquette)
    r = Insere_aux(x, v, nœud.fils[i])
    Si r = NouveauSousArbre(x', v', t)
      insère x' à la position i   dans nœud.étiquettes
      insère v' à la position i   dans nœud.valeurs
      insère t  à la position i+1 dans nœud.fils
      nœud.nombre_fils += 1
      Si nœud.nombre_fils > m
        nœud' = AllouerNouveauNœud()
        x' = nœud.étiquettes[(m-1)/2]
        v' = nœud.valeurs[(m-1)/2]
        nœud'.étiquettes = nœud.étiquettes[(m+1)/2..m-1]
        nœud'.valeurs    = nœud.valeurs[(m+1)/2..m-1]
        nœud'.fils       = nœud.fils[(m+1)/2..m]
        tronquer nœud.étiquettes à (m-1)/2 éléments
        tronquer nœud.valeurs    à (m-1)/2 éléments
        tronquer nœud.fils       à (m+1)/2 éléments
        Renvoyer NouveauSousArbre(x', v', nœud')
      Sinon
        Renvoyer Ok

Fonction Insere(x, v, arbre)
  r = Insere_aux(x, v, arbre.racine)
  Si r = NouveauSousArbre(x', v', t)
    nœud' = AllouerNouveauNœud()
    nœud'.étiquettes = [x']
    nœud'.valeurs = [v']
    nœud'.fils = [arbre.racine, t]
    arbre.racine = nœud'
```

Dans le pire cas, il faut rajouter un nœud en $O(m)$ pour chaque nœud d'une feuille jusqu'à la racine. La complexité en pire cas est donc $O(m \log_m N)$. Bien sûr, en pratique, les ajouts de nouveaux nœuds sont rares. S'il n'y a pas de tel ajout, la complexité est $O(m + \log_m N)$.

(b) Si, par exemple, on insère la séquence d'entiers $1, \ldots, N$, alors on va remplir les nœuds de la frange droite de l'arbre, puis les diviser en deux. Après une telle division, le nouveau nœud de droite restera inchangé. Ainsi, à part la frange de droite, tous les nœuds de l'arbre ne seront qu'à moitié pleins, c'est-à-dire que, à une entrée près, ils seront au minimum de leur remplissage.

(c) **Pas de pseudo-code demandé pour cette question. Une simple explication détaillée suffira.**

Le problème de l'algorithme précédent, c'est qu'on ne cherche pas à remplir les nœuds voisins lorsqu'un nœud est plein. Une variante consiste, lorsqu'un nœud est trop plein, à essayer de transférer des fils et des étiquettes vers ses frères gauche et droit immédiats avant de décider de diviser le nœud en deux parties. Lors d'un tel transfert, il faut aussi modifier l'étiquette séparatrice dans le nœud parent.

La complexité en temps dans le pire cas reste la même : $O(m \log_m N)$.

Dans cette variante, lorsque l'on crée un B-arbre à partir d'une séquence croissante d'étiquettes, l'essentiel des nœuds de l'arbre est plein, sauf ceux de la frange droite, qui sont en cours de remplissage.

Question 3. Pas de pseudo-code demandé pour cette question. Une simple explication détaillée suffira.

Tout comme pour un arbre binaire de recherche, supprimer une entrée d'un B-arbre qui ne se trouve pas dans un nœud feuille est plus difficile. On commence donc par se ramener à ce cas : si une étiquette dans un arbre B ne se situe pas dans un nœud feuille, alors son prédécesseur immédiat (qui existe forcément) est nécessairement stocké dans un nœud feuille. Ainsi, si on se trouve dans cette situation, on commence par supprimer le prédécesseur immédiat de l'entrée que nous souhaitons supprimer, puis on remplace dans l'arbre l'entrée que nous voulons vraiment supprimer par celle que nous venons de supprimer dans un nœud feuille.

Il faut donc résoudre le problème de la suppression d'une entrée de l'arbre se situant dans un nœud feuille. On procède de manière duale de l'algorithme proposé dans la question 2.c : on commence par supprimer l'entrée de son nœud feuille. Si le nœud en question ne contient alors pas assez d'entrées (il en contient donc $\lceil \frac{m}{2} \rceil - 2$), alors on essaie d'en transférer depuis ses frères gauche et droits (puisque $m \geq 3$, tout nœud a nécessairement au moins un frère). Si un frère gauche ou droit est lui-même à la limite basse de son remplissage (c'est-à-dire qu'ils ont chacun $\lceil \frac{m}{2} \rceil - 1$ étiquettes), alors on peut fusionner un frère avec le nœud courant : en comptant le séparateur provenant du parent, on obtient un nouveau nœud avec $2 \lceil \frac{m}{2} \rceil - 2$ étiquettes, soit moins que la limite supérieure de $m - 1$ étiquettes. Bien sûr, cette fusion de nœuds peut poser un problème pour le parent, puisque celui-ci peut ne plus être assez rempli. Il faut donc continuer cette opération jusqu'à la racine. Arrivé à la racine, si celle-ci n'a plus qu'un fils à l'issue de l'opération, on la supprime et on prend son unique fils comme nouvelle racine.

Comme pour l'insertion, on effectue dans le pire cas $O(m)$ opérations à chaque niveau de l'arbre. La complexité dans le pire cas est donc $O(m \log_m N)$.

Tout comme pour l'insertion, dans la grande majorité des cas, il ne sera pas nécessaire de fusionner de nœuds, et la complexité sera alors $O(m + \log_m N)$.

Question 4. Tout d'abord, on se rend vite compte qu'il sera nécessaire d'extraire l'entrée la plus grande de T_1 ou la plus petite de T_2 pour servir de séparateur lors de la fusion. On peut faire cette opération grâce à l'algorithme de la question 3.

Le piège ensuite serait d'insérer l'un des deux arbres comme fils de la racine de l'autre, ou, pire, de créer un nouvelle racine comme parent des racines des deux arbres. En effet, une telle opération, dans le cas général, casserait l'invariant selon lequel toutes les feuilles ont la même hauteur.

On distingue plusieurs cas :

— Si les deux arbres ont la même hauteur, alors on essaie de fusionner leurs racines. Si la fusion des racines est trop grosse, alors on fait comme lors d'une insertion : on divise la racine en deux, et on crée une nouvelle racine.

— Si les deux arbres n'ont pas la même hauteur, supposons par exemple que T_1 soit plus haut. Dans ce cas, on cherche dans la frange droite de T_1 le nœud dont la hauteur est immédiatement supérieure à la hauteur de T_2, et on y insère T_2 comme dernier fils en utilisant le séparateur préalablement extrait. Il faut alors vérifier que ce nœud n'est pas trop plein, et réutiliser les algorithmes de la question 2 si nécessaire.

Question 5. Lors de l'insertion d'une nouvelle entrée, en utilisant la technique de la question 2.c., on peut se ramener au cas où on divise un nœud qui si ses (ou son) frères immédiats sont entièrement pleins. Dans ce cas, au lieu de diviser le nœud en deux, on peut plutôt choisir de le fusionner avec son

voisin plein et de diviser le tout en trois. Cela permet effectivement de respecter la borne inférieure $\lceil \frac{2m-1}{3} \rceil$.

Pour la suppression, on remarque la même chose : on peut se ramener au cas où il faut fusionner trois nœuds qui sont à la limite inférieure du remplissage en deux nœuds.

Néanmoins, si on utilise ces algorithmes, il faut relâcher la contrainte sur le nombre de fils de la racine, puisque celle-ci n'a pas de frère avec lequel on pourrait fusionner avant de diviser en trois. Ainsi, on doit autoriser la racine à avoir entre 2 et $2 \lfloor \frac{2m-2}{3} \rfloor + 1$ fils (inclus).

Pour plus d'informations sur les B-arbres, on pourra consulter [Knu98, §6.2.4] ou [Wik21].

Références

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3 : Sorting and Searching.* Addison Wesley, 1998.

[Wik21] Wikipedia. B-tree, 2021. `https://en.wikipedia.org/wiki/B-tree`.

P1 – Théorème général de l'analyse des programmes récursifs

Étant données deux fonctions $f, g : \mathbb{R}^+ \to \mathbb{R}^+$, on note $f(n) = \Theta(g(n))$ si $f(n) = O(g(n))$ et $g(n) = O(f(n))$.

On considère dans ce problème un algorithme récursif $\mathcal{A}$ prenant une entrée de taille $n \in \mathbb{N}^*$. On suppose que :

— si $n = 1$, $\mathcal{A}$ met un temps borné par une constante ;

— pour $n > 1$, $\mathcal{A}$ fait un nombre $a \in \mathbb{N}^*$ d'appels récursifs à $\mathcal{A}$ sur une entrée de taille $\frac{n}{b}$ (b est un rationnel strictement plus grand que 1), ainsi qu'un certain nombre d'autres opérations dont le temps est $f(n)$.

Ainsi, le temps $T(n)$ pris pour résoudre le problème au rang n est :

$$T(n) = a \times T\left(\frac{n}{b}\right) + f(n).$$

Par simplicité, on supposera dans tout le problème qu'on applique toujours l'algorithme à un n pour lequel $\frac{n}{b}$ est un entier, y compris lors des appels récursifs.

Question 0. Donner les valeurs de a, b et une estimation asymptotique de $f(n)$ sous la forme d'un $\Theta(g(n))$ pour le cas de l'algorithme de recherche par dichotomie dans un tableau de taille n.

Question 1. Donner sous la forme de pseudo-code l'algorithme de tri fusion.

En déduire les valeurs de a, b et une estimation asymptotique de $f(n)$ sous la forme d'un $\Theta(g(n))$ pour le cas de l'algorithme de tri fusion d'une liste de n éléments.

On cherche maintenant à résoudre la formule de récurrence définissant $T(n)$ dans le cas le plus général possible pour permettre de déterminer la complexité asymptotique de l'algorithme $\mathcal{A}$.

Question 2. Représenter les appels récursifs effectués par $\mathcal{A}$ sur une entrée de taille n sous la forme d'un arbre dont la racine représente l'appel principal et les enfants d'un nœud les appels récursifs directs effectués. On indiquera comme étiquette d'un nœud de l'arbre la taille de l'entrée.

Question 3. Pour un certain $k \in \mathbb{N}$ fixé et inférieur à la hauteur de l'arbre, combien de nœuds de profondeur k (c'est-à-dire, à distance k de la racine) cet arbre comporte-t-il ?

Question 4. Exprimer la hauteur de l'arbre en fonction de n et b.

Question 5. Montrer l'égalité suivante : $T(n) = \Theta(n^c) + \sum_{k=0}^{\log_b n - 1} a^k \times f\left(\frac{n}{b^k}\right)$, où $c = \log_b a$.

Question 6. Montrer que si $f(n) = O(n^{c'})$ avec $c' < c$, alors $T(n) = \Theta(n^c)$.

Question 7. Montrer que si $f(n) = \Theta(n^c)$, alors $T(n) = \Theta(n^c \log n)$.

Question 8. Montrer que si $n^{c'} = O(f(n))$ avec $c' > c$ et si $a \times f\left(\frac{n}{b}\right) \leq \alpha f(n)$ pour un certain $0 < \alpha < 1$ avec n suffisamment grand, alors $T(n) = \Theta(f(n))$.

Les résultats établis aux trois dernières questions forment les trois cas du théorème général de l'analyse des programmes récursifs.

Suite des questions

Question 9. Retrouver la complexité de la recherche par dichotomie et du tri fusion avec ce théorème.

Question 10. En utilisant ce théorème, donner la complexité d'un problème dont la complexité est décrite par la formule de récurence suivante :

$$\begin{cases} T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{pour } n \geq 2 \\ T(1) = \Theta(1) \end{cases}$$

(C'est la complexité de la multiplication de matrices par l'algorithme naïf récursif.)

Question 11. En utilisant ce théorème, donner la complexité d'un problème dont la complexité est décrite par la formule de récurence suivante :

$$\begin{cases} T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{pour } n \geq 2 \\ T(1) = \Theta(1) \end{cases}$$

(C'est la complexité de la multiplication de matrices par l'algorithme de Strassen.)

Question 12. Le théorème s'applique-t-il à toutes les récurrences de la forme $T(n) = a \times T\left(\frac{n}{b}\right) + f(n)$?

Corrigé

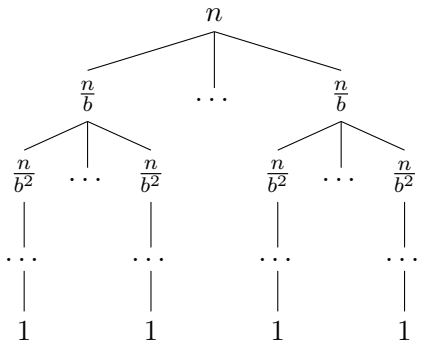
Question 0. $a = 1$, $b = 2$, $f(n) = \Theta(1)$

Question 1.

Fonction `TriFusion` *(liste)*
- **si** longueur(*liste*) ≤ 1 **alors**
  - **retourner** *liste*;
- $l_1 \leftarrow$ liste_vide();
- $l_2 \leftarrow$ liste_vide();
- **tant que** $\neg$vide(*liste*) **faire**
  - $l_1 \leftarrow$ depiler(*liste*);
  - **si** $\neg$vide(*liste*) **alors**
    - $l_2 \leftarrow$ depiler(*liste*);
- $l_1 \leftarrow$ `TriFusion`(l_1);
- $l_2 \leftarrow$ `TriFusion`(l_2);
- *result* $\leftarrow$ liste_vide();
- **tant que** $\neg$vide(l_1) $\vee$ $\neg$vide(l_2) **faire**
  - **si** vide(l_2) $\vee$ ($\neg$vide(l_1) $\wedge$ premier(l_1) $\leq$ premier(l_2)) **alors**
    - *result* $\leftarrow$ depiler(l_1);
  - **sinon**
    - *result* $\leftarrow$ depiler(l_2);
- **retourner** *result*;

(depiler(l) enlève le premier élément de l et le retourne.)
On en déduit : $a = 2$, $b = 2$, $f(n) = \Theta(n)$.

Question 2.



Question 3. a^k

Question 4. $\log_b n + 1$ (pour la définition usuelle de la hauteur d'un arbre où la hauteur d'un arbre formé d'un seul nœud est 1)

Question 5. En chaque nœud interne, si la taille de l'entrée est $\frac{n}{b^k}$, le temps de calcul propre à cet appel de fonction est $f\left(\frac{n}{b^k}\right)$. Pour chacune des feuilles, le temps de calcul est de $\Theta(1)$. On a donc :

$$T(n) = a^{\log_b n} \times \Theta(1) + \sum_{k=0}^{\log_b n - 1} a^k \times f\left(\frac{n}{b^k}\right) = \Theta(n^{\log_b a}) + \sum_{k=0}^{\log_b n - 1} a^k \times f\left(\frac{n}{b^k}\right)$$

car $a^{\log_b n} = n^{\log_b a}$ (redémontrer avec la définition de la puissance réelle si nécessaire).

Dans les trois questions suivantes, on pose

$$g(n) = \sum_{k=0}^{\log_b n - 1} a^k \times f\left(\frac{n}{b^k}\right).$$

Question 6. On pose $c = c' + \epsilon$ avec $\epsilon > 0$. Alors :

$$\begin{aligned}
g(n) &= \sum_{k=0}^{\log_b n - 1} a^k \times O\left(\left(\frac{n}{b^k}\right)^{c'}\right) \\
&= O\left(\sum_{k=0}^{\log_b n - 1} a^k \left(\frac{n}{b^k}\right)^{c-\epsilon}\right) \\
&= O\left(n^{c-\epsilon} \times \sum_{k=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^k b^{k\epsilon}\right) \\
&= O\left(n^{c-\epsilon} \times \sum_{k=0}^{\log_b n - 1} (b^\epsilon)^k\right) \\
&= O\left(n^{c-\epsilon} \times \frac{(b^\epsilon)^{\log_b n} - 1}{b^\epsilon - 1}\right) \\
&= O\left(n^{c-\epsilon} \times (n^{\epsilon \log_b b} - 1)\right) \\
&= O\left(n^c\right).
\end{aligned}$$

Ce terme est donc dominé par le $\Theta(n^c)$ dans l'expression de $T(n)$ et $T(n) = \Theta(n^c)$.

Question 7. On a :

$$\begin{aligned} g(n) &= \sum_{k=0}^{\log_b n-1} a^k \times f\left(\frac{n}{b^k}\right) \\ &= \Theta\left(\sum_{k=0}^{\log_b n-1} a^k \frac{n^c}{b^{kc}}\right) \\ &= \Theta\left(n^c \times \sum_{k=0}^{\log_b n-1} \left(\frac{a}{b^c}\right)^k\right) \\ &= \Theta\left(n^c \times \log_b n\right) \end{aligned}$$

Ce terme domine donc le $\Theta(n^c)$ dans l'expression de $T(n)$ et $T(n) = \Theta(n^c \log n)$.

Question 8. On sait qu'il existe une constante $N \in \mathbb{N}$ telle que, pour tout $k \geq 1$ avec $\frac{n}{b^{k-1}} \geq N$:

$$f(n) \geq \frac{a}{\alpha} f\left(\frac{n}{b}\right) \geq \frac{a^2}{\alpha^2} f\left(\frac{n}{b^2}\right) \geq \cdots \geq \frac{a^k}{\alpha^k} f\left(\frac{n}{b^k}\right)$$

On note k_0 le plus grand $k \geq 1$ tel que $\frac{n}{b^{k-1}} \geq N$. Sans perte de généralité, vu qu'on cherche une complexité asymptotique, on suppose $n \geq N$ et donc ce k_0 est bien défini. On note que pour $k > k_0$, $\frac{n}{b^k} < N$; N étant une constante (indépendante de n), cela signifie que pour $k > k_0$, $f\left(\frac{n}{b^k}\right) = O(1)$.

Donc :

$$\begin{aligned} g(n) = \sum_{k=0}^{\log_b n-1} a^k \times f\left(\frac{n}{b^k}\right) &\leq \sum_{k=0}^{k_0} a^k \times f\left(\frac{n}{b^k}\right) + \sum_{k=k_0+1}^{\log_b n-1} a^k \times f\left(\frac{n}{b^k}\right) \\ &\leq \sum_{k=0}^{k_0} \alpha^k \times f(n) + \sum_{k=k_0+1}^{\log_b n-1} a^k \times f\left(\frac{n}{b^k}\right) \\ &\leq \left(\sum_{k=0}^{k_0} \alpha^k\right) \times f(n) + O(\log n) \times a^{\log_b n-1} \times O(1) \\ &\leq \left(\sum_{k=0}^{\infty} \alpha^k\right) \times f(n) + O(\log n) \times (a^{\log_b n}) \\ &\frac{1}{1-\alpha} \times f(n) + O(n^c \log n) = O(f(n)) + O(n^{c'}) = O(f(n)) \end{aligned}$$

Donc $g(n) = O(f(n))$. Mais on a aussi $f(n) = O(g(n))$ car $f(n)$ est un des termes de la somme de nombres positifs définissant $g(n)$. Donc $g(n) = \Theta(f(n))$. Or $n^{c'} = O(f(n))$ d'où $n^{c'} = O(g(n))$ et $T(n) = \Theta(g(n))$, le terme $\Theta(n^c)$ étant dominé par $g(n)$. On a donc $T(n) = \Theta(g(n)) = \Theta(f(n))$.

Question 9. Pour la recherche par dichotomie, on a $c = \log_2 1 = 0$ donc $f(n) = \Theta(n^c)$ et on est dans le deuxième cas du théorème, d'où $T(n) = \Theta(n^c \log n) = \Theta(\log n)$.

Pour le tri fusion, on a $c = \log_2 2 = 1$ donc $f(n) = \Theta(n^c)$ et on est dans le deuxième cas du théorème, d'où $T(n) = \Theta(n^c \log n) = \Theta(n \log n)$.

Question 10. On a $c = \log_2 8 = 3$ et $f(n) = O(n^2)$ avec $2 < 3$ donc on est dans le premier cas du théorème, d'où $T(n) = \Theta(n^c) = \Theta(n^3)$.

Question 11. On a $c = \log_2 7$ et $f(n) = O(n^2)$ avec $2 < \log_2 7$ donc on est dans le premier cas du théorème, d'où $T(n) = \Theta(n^c) = \Theta(n^{\log_2 7})$.

Question 12. Non. Par exemple le cas $a = 1$, $b = 2$ (d'où $c = 0$) et $f(n) = \Theta(\log n)$ n'est pas couvert (en fait, le théorème peut se généraliser à ce cas précis, et donne $T(n) = \Theta((\log n)^2)$, mais d'autres cas ne sont pas couverts, par exemple les cas où la condition supplémentaire du troisième cas n'est pas vérifiée).

P2 – Tas de Fibonacci

Une *file de priorité* est une structure de données utilisée pour stocker des objets avec une *valeur de priorité* (un nombre en virgule flottante) et qui supporte les opérations suivantes :

vide() Renvoie une file de priorité vide.

insère(file, objet, priorité) Ajoute un objet à la file, avec sa valeur de priorité.

dépile(file) Supprime l'objet dont la valeur de priorité est la plus haute dans la file, et le renvoie ainsi que sa priorité.

augmente(file, objet, priorité) Mettre à jour la valeur de priorité d'un objet, avec la condition que la nouvelle valeur de priorité doit être plus grande que l'ancienne.

Question 0. Donner le pseudo-code de l'algorithme de Dijkstra pour calculer la plus courte distance d'un nœud *source* à un nœud *destination* dans un graphe avec des poids positifs ou nuls. Le pseudo-code devra utiliser une file de priorité avec les opérations ci-dessus.

Quelle est la complexité en terme du nombre n de nœuds et m d'arêtes du graphe, ainsi que de la complexité des opérations de la file de priorité utilisée ?

Question 1. Quelle est la complexité de l'algorithme de Dijkstra si on utilise un tas binaire ?

Un *tas de Fibonacci* est une structure de données complexe utilisée pour implémenter des files de priorité. Formellement, c'est une collection de t arbres (non nécessairement binaires) dont les nœuds correspondent aux n objets à stocker. Chaque arbre vérifie la *condition de tas de priorité* : la priorité d'un nœud est toujours supérieure ou égale à la priorité de ses descendants. De plus, on mémorise un certain nombre d'informations supplémentaires :

- Chaque nœud d'un arbre a une *marque* qui est une variable booléenne, initialement mise à Faux, et mise à Vrai si le nœud *a perdu un enfant depuis la dernière fois qu'il a changé de parent.*
- Une variable spéciale *max* indique quel arbre a le nœud racine avec la plus haute valeur de priorité.
- Le nombre t d'arbres et le nombre $\delta(u)$ d'enfants de chaque nœud est également gardé en mémoire.

Question 2. Proposer une manière d'implémenter l'opération vide le plus simplement possible. Proposer une implémentation la plus simple possible de l'opération insère en $O(1)$.

Question 3. On associe à chaque tas de Fibonacci H un *potentiel* $\phi(H) \coloneqq \gamma \times (t + 2m)$ où t est le nombre d'arbres dans le tas, m le nombre de nœuds marqués à Vrai et γ une constante positive que l'on définira plus loin.

Pour une opération x qui transforme un tas de Fibonacci H en un nouveau tas de Fibonacci $x(H)$ avec coût de calcul $c_x(H)$, on considère $\hat{c}_x(H) \coloneqq c_x(H) + \phi(x(H)) - \phi(H)$. Montrer que pour toute séquence d'opérations $x_1 \dots x_k$ à partir du tas de Fibonacci vide H_0, avec $H_i \coloneqq x_i(H_{i-1})$: $\frac{1}{k}\sum_{i=1}^{k} c_{x_i}(H_{i-1}) \leq \frac{1}{k}\sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1})$. On appellera $\hat{c}_x(H)$ le *coût amorti* de l'opération x.

Question 4. Dans les tas de Fibonacci, dépile fonctionne comme suit : on enlève le nœud pointé par le pointeur *max* et on fait de chacun de ses enfants une nouvelle racine d'un arbre. Ensuite, on s'assure que chaque racine r a un nombre d'enfants $\delta(r)$ distinct : pour ce faire, chaque fois que deux racines ont le même nombre d'enfants, l'une devient le fils de l'autre (en respectant la condition de tas de priorité), et on répète jusqu'à ce que toutes les racines aient des degrés distincts. Pendant ces opérations, on met à jour la marque quand c'est nécessaire.

Montrer que, en choisissant une valeur appropriée de γ, la *coût amorti* de dépile dans un tas de Fibonacci H est en : $O\left(\max\left(\max_{u \text{ nœud de } H} \delta(u), \max_{u \text{ nœud de dépile}(H)} \delta(u)\right)\right)$.

Suite des questions

Question 5. Dans les tas de Fibonacci, augmente fonctionne comme suit : si, après mise à jour des priorités sur un nœud u, la condition de tas de priorité est violée, u est détaché de son parent p et devient la racine d'un nouvel arbre. On procède ensuite en cascade sur le nœud p : si le nœud p avait déjà sa marque à Vrai, on le détache de son propre parent p' et il devient la racine d'un nouvel arbre ; on continue récursivement sur p'.

Montrer qu'en choisissant une valeur appropriée pour γ (compatible avec le choix déjà fait), le coût amorti de augmente est un $O(1)$. On pourra introduire le nombre de fois que la procédure de cascade a été appelée.

Question 6. On veut maintenant montrer que le degré maximal d'un nœud dans un tas de Fibonacci est en $O(\log n)$.

a) Montrer que pour tout nœud u avec enfants $u_1, u_2, \ldots, u_k$ ordonnés dans l'ordre chronologique de leur ajout comme enfant de u, pour tout $1 \leq i \leq k$, $\delta(u_i) \geq i - 2$.

b) Soit F_k le k-ième terme de la suite de Fibonacci : $\begin{cases} F_0 = 0 \quad F_1 = 1 \\ F_k = F_{k-1} + F_{k-2} \quad \text{pour } k \geq 2. \end{cases}$. Montrer que pour tout nœud u d'un tas de Fibonacci, la taille du sous-arbre enraciné en u est $\geq F_{\delta(u)+2}$.

c) Montrer que, pour tout k, $F_{k+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$ et conclure.

Question 7. Quelle est la complexité *amortie* de l'algorithme de Dijsktra (c.-à-d., en remplaçant les coûts réels des opérations par leurs coûts amortis) si on implémente la file de priorité avec un tas de Fibonacci ? Comparer avec le tas binaire.

Corrigé

Question 0. On suppose pour simplifier que les nœuds du graphe sont numérotés par des entiers consécutifs à partir de 0.

```
Fonction Dijkstra(G, source, destination)
    n ← nombre de nœuds de G;
    f ← vide();
    d ← tableau de n éléments;
    pour u nœud de G faire
        insère(f, u, −∞);
        d[u] ← ∞;
    augmente(source, 0);
    d[source] ← 0;
    tant que (u, p) ← dépile(f) faire
        pour (u, v, w) ∈ G faire
            si −p + w ≤ d[v] alors
                d[v] ← −p + w;
                augmente(f, v, −d[v]);
    retourner d[destination];
```

Si les complexités de insère, augmente et dépile sont respectivement de ι, α et δ, la complexité est de $O(n \times (\iota + \delta) + m \times \alpha)$.

On peut bien sûr accélérer un peu cet algorithme en arrêtant la boucle dès que la distance à *destination* a été déterminée, sans changer la complexité asymptotique.

Question 1. On a $\iota = \delta = \alpha = O(\log n)$ dans un tas binaire. En effet, insère et dépile sont des opérations classiques ; pour augmente on peut par exemple supprimer puis rajouter l'élement avec la même priorité – il est nécessaire d'avoir une structure de données auxiliaire (comme un simple tableau) pour savoir où chaque nœud se trouve dans le tas afin de trouver l'élément à supprimer en $O(1)$.

On obtient donc une complexité en $O((n+m)\log n)$ pour la complexité de l'algorithme de Dijkstra.

Question 2. Pour vide(), il suffit de retourner un tas avec $t = 0$ arbre et *max* non positionné.

Pour insère(f, o, p), afin de garantir une insertion simple en $O(1)$, on peut juste ajouter o à f comme une nouvelle racine d'un arbre formé de ce seul élément, avec la *marque* mise à Faux, et *max* pointant vers ce nouvel arbre si et seulement si la priorité p est supérieure à la priorité de l'arbre actuellement pointé par *max*.

Question 3. On a :

$$\begin{aligned}\sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1}) &= \sum_{i=1}^{k} c_{x_i}(H_{i-1}) + \sum_{i=1}^{k} \phi(H_i) - \sum_{i=0}^{k-1} \phi(H_i) \\ &= \sum_{i=1}^{k} c_{x_i}(H_{i-1}) + \phi(H_k) - \phi(H_0) \\ &\geq \sum_{i=1}^{k} c_{x_i}(H_{i-1})\end{aligned}$$

puisque $\phi(H_0) = \gamma \times (0 + 2 \times 0) = 0$ et ϕ est une fonction à valeur positive ou nulle. On obtient le résultat demandé en divisant les deux membres de l'inégalité par k.

Question 4. On pose $D = \max\left(\max_{u \text{ nœud de } H} \delta(u), \max_{u \text{ nœud de dépile}(H)} \delta(u)\right)$.

Appelons u le nœud pointé par *max*. Enlever ce nœud et faire de ses enfants de nouvelles racines se fait en $O(\delta(u))$. On énumère ensuite les degrés des racines en $O(t+\max_{u \text{ nœud de } H} \delta(u)) = O(t+D)$ en stockant par exemple dans un tableau indexé par le degré les nœuds ayant un degré donné, ce tableau ne nécessitant de considérer que $O(t+D)$ degrés différents. Chaque fois que deux racines ont le même degré, on rend l'une fils de l'autre en $O(1)$ en comparant leurs priorités. On peut devoir répéter cette opération, mais comme à chaque fois le nombre de racines décroît d'une unité, on réalise cette opération au plus $O(t + \delta(u)) = O(t + D)$ fois. On a donc un coût (non amorti) de chaque appel à dépile en $O(t + D)$. Soit c_1 une constante telle que, pour un tas de Fibonacci suffisamment grand, le coût (non amorti) est $\leq c_1(t + D)$.

Le coût amorti est $\leq c_1(t + D) + \gamma(D + 1 + 2m - t - 2m)$: on a au final au plus $D + 1$ arbres et on n'ajoute pas de nouvelle marque à Vrai (on peut éventuellement en enlever, mais l'inégalité reste correcte). On choisit $\gamma \geq c_1$ de sorte que $t(c_1 - \gamma) \leq 0$. On a alors un coût amorti $\leq c_1 D + \gamma(D + 1)$, soit en $O(D)$.

Question 5. Détacher un nœud de son parent et le faire devenir la racine d'un nouvel arbre peut se faire en $O(1)$. Cette opération peut être faite un certain nombre ℓ de fois, mais à chaque fois (sauf éventuellement la première, où le nœud n'est pas nécessairement marqué), une marque passe à Faux ; une fois qu'on atteint un parent que l'on ne détache pas, la marque de ce parent passe en revanche à Vrai. On a donc un coût amorti, pour un tas de Fibonacci suffisamment grand qui est $\leq c_2\ell + \gamma(t + \ell + 2(m - (\ell + 1) + 1) - t - 2m) = c_2\ell + \gamma(4 - \ell)$ pour une certaine constante c_2. On prend $\gamma \geq c_2$, ce qui est compatible avec le choix $\gamma \geq c_1$ fait précédemment. On obtient un coût amorti $\leq 4\gamma$, et donc en $O(1)$.

Question 6.

a) On remarque que la seule opération qui ajoute un nœud u_i comme enfant d'un autre nœud u est dépile et uniquement si l'ancien degré du nœud u avant cet ajout (qui est précisément $i-1$) était identique au degré du nœud u_i au moment où il a été ajouté. Depuis cet ajout, u_i a perdu au plus un enfant lors de l'exécution de augmente – si u_i avait perdu plus d'un enfant, augmente l'aurait détaché de u (et u_i n'a pas pu gagner d'enfant, aucune opération n'ajoute à un enfant à un nœud non-racine). On a donc $\delta(u_i) \in \{i-2, i-1\}$ et, en particulier, $\delta(u_i) \geq i-2$.

b) On pose s_k la taille minimale du sous-arbre enraciné en un nœud ayant k enfants. Clairement la fonction $k \mapsto s_k$ est croissante.
Soit u un nœud arbitraire avec comme enfants $u_1, \ldots, u_k$. On a :

$$\begin{aligned} s_{\delta(u)} &\geq 1 + \sum_{i=1}^{k} s_{\delta(u_i)} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \end{aligned}$$

On montre par récurrence sur k que $s_k \geq F_{k+2}$. Les cas de base sont clairs : $s_0 = 1 = F_2$ et $s_1 = 2 = F_3$. Pour le cas inductif, on suppose que $k \geq 2$ et que l'inégalité est vraie pour tout $i < k$. On a :

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

(Cette dernière égalité peut facilement se montrer par récurrence sur k).

c) On démontre l'inégalité par récurrence sur k.
- Pour $k = 0$, $F_{k+2} = F_2 = 1 \geq 1 = \left(\frac{1+\sqrt{5}}{2}\right)^k$.
- Pour $k = 1$, $F_{k+2} = F_3 = 2 \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$ vu que $\sqrt{5} < 3$, c.-à-d., $\frac{1+\sqrt{5}}{2} < 2$.
- Soit $k \geq 2$ et supposons l'inégalité vraie aux rangs $k-2$ et $k-1$. On a :

$$F_{k+2} = F_{k+1} + F_k \geq \left(\frac{1+\sqrt{5}}{2}\right)^{k-2} \times \left(\frac{1+\sqrt{5}}{2} + 1\right)$$

Or $\left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{6+2\sqrt{5}}{4} = \frac{1+\sqrt{5}}{2} + 1$, d'où $F_{k+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$.

En conclusion, on a montré que la taille du sous-arbre enraciné en tout nœud u est $\geq \psi^{\delta(u)}$ où $\psi = \frac{1+\sqrt{5}}{2}$. En particulier, $\psi^{\delta(u)} \leq n$ et $\delta(u) \leq \log_\psi(n)$. Cela est vrai pour n'importe quel nœud u, en particulier pour le degré maximal, qui est donc en $O(\log n)$.

Question 7. En utilisant les résultats des questions précédentes, on a pour le coût amorti de insère, augmente et dépile respectivement $O(1)$, $O(1)$ et $O(\log n)$. On obtient donc une complexité amortie de $O(m + n \log n)$ ce qui est asymptotiquement mieux que le $O((n+m)\log n)$ obtenu par une implémentation de Dijkstra avec un tas binaire comme file de priorité ; la différence ne devient cependant visible que pour les graphes ayant beaucoup d'arêtes.

P3 – Hachage universel

On considère des objets décrits par des suites finies de bits, c.-à-d., par des mots de $\{0,1\}^*$. On fixe $K \in \mathbb{N}$. Une *fonction de hachage* est une fonction $h : \{0,1\}^* \to [\![0; 2^K - 1]\!]$ associant à chaque suite finie de bits un entier entre 0 et $2^K - 1$.

Étant donnée une distribution de probabilités Pr sur l'ensemble $\{0,1\}^*$, on dit que la fonction de hachage est *uniforme pour* Pr si : $\forall 0 \leq k < 2^K, \Pr(h(w) = k) = 2^{-K}$.

Une *table de hachage* pour une fonction de hachage h est un tableau de 2^K cases contenant une liste d'éléments de Σ^*. Elle permet de stocker un ensemble fini S d'objets de Σ^* de la manière suivante : dans la case i, on stocke tous les éléments $u \in S$ tels que $h(u) = i$, dans un ordre arbitraire.

Question 0. Donner le pseudo-code des opérations de base sur les tables de hachage : rechercher si un élément est dans l'ensemble, ajouter un élément, en supprimer un.

Question 1. Donner la complexité de ces fonctions en fonction de $|S|$ et de K dans le pire des cas. Donner la complexité en moyenne de ces fonctions si on suppose que les éléments insérés dans la table suivent une loi de probabilité Pr et que h est uniforme pour Pr.

Question 2. Soit $n \in \mathbb{N}$. On fixe $\Pr_0$ comme suit : $\begin{cases} \Pr_0(w) = 0 & \text{si } |w| \neq n\,; \\ \Pr_0(w) = \frac{1}{2^{|w|}} & \text{sinon.} \end{cases}$

Montrer que $\Pr_0$ est bien une distribution de probabilités. Donner une condition nécessaire et suffisante sur K pour l'existence d'une fonction de hachage h uniforme pour $\Pr_0$ et exhiber une telle fonction.

Question 3. On fixe $\Pr_1$ comme suit : $\begin{cases} \Pr_1(\epsilon) = \frac{1}{2}; & \\ \Pr_1(w) = 0 & \text{si } w \text{ est dans le langage décrit par } (0+1)^*0\,; \\ \Pr_1(w) = \frac{1}{2^{2|w|}} & \text{sinon.} \end{cases}$

Montrer que $\Pr_1$ est bien une distribution de probabilités. Donner une condition nécessaire et suffisante sur K pour l'existence d'une fonction de hachage h uniforme pour $\Pr_1$ et exhiber une telle fonction.

Les questions précédentes montrent que le choix d'une fonction de hachage uniforme (quand c'est possible) dépend de la distribution Pr – mais celle-ci n'est pas forcément connue à l'avance. Le but du *hachage universel* est de pouvoir obtenir des garanties probabilistes d'uniformité en absence d'informations sur Pr.

Un ensemble fini H de fonctions de hachage est dit *universel* pour $L \subseteq \Sigma^*$ si pour tous $u, v \in L$ avec $u \neq v$ le nombre de fonctions $h \in H$ telles que $h(u) = h(v)$ est au plus $|H| \cdot 2^{-K}$.

Question 4. Soit H un ensemble universel de fonctions de hachage pour un ensemble L de mots. Montrer que si on tire uniformément aléatoirement une fonction dans H, alors la complexité des opérations de base sur la table de hachage est, *en espérance sur ces tirages aléatoires*, en $O(|S| \cdot 2^{-K} + 1)$.

Question 5. On fixe $n \in \mathbb{N}$ avec $n > K$ et on considère $L_n = \{u \in \Sigma^* \mid |u| = n\}$. Pour $u \in L_n$, on pose $\hat{u}$ le nombre entier dont le code en binaire est u. Soit $p > 2^n$ un nombre premier. Pour $0 < a < p$ et $0 \leq b < p$ deux entiers, on pose $h_{ab} : u \mapsto ((a\hat{u} + b) \bmod p) \bmod 2^K$. Montrer que l'ensemble $\{h_{ab} \mid 0 < a < p, 0 \leq b < p\}$ est un ensemble universel de fonctions de hachage pour L_n.

Question 6. En déduire le pseudo-code d'une structure de données permettant de gérer efficacement un ensemble d'au plus T éléments de L_n. Comparer la complexité en espace et en temps avec celle d'un arbre binaire de recherche équilibré utilisé pour le même but.

L1 – Complexité de Kolmogorov sans préfixe

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 $|w|$. La concaténation de deux mots binaires u et v est notée uv. Un ensemble $A \subseteq \Sigma^*$ est *sans préfixe* si pour tout $w \in A$ et tout préfixe u de w tel que $u \neq w$, on a $u \notin A$.

Question 0. Écrire en pseudo-code un programme qui prend en entrée un ensemble fini A de mots binaires et sa taille n, et retourne VRAI si A est sans préfixe et FAUX sinon. Discuter de sa complexité.

Question 1. Soit A un ensemble (fini ou infini) de mots binaires sans préfixe. Montrer que

$$\sum_{w \in A} 2^{-|w|} \leq 1$$

Une *machine* M est un programme qui prend en entrée un mot binaire w, et soit renvoie un mot binaire u, soit ne s'arrête pas. Une machine peut donc être vue comme une fonction partielle de Σ^* vers Σ^*. Une machine est *sans préfixe* si son domaine de définition est un ensemble sans préfixe. La *complexité de Kolmogorov* $K_M(w)$ d'un mot binaire w associé à une machine sans préfixe M est la longueur minimale d'un mot u tel que $M(u) = w$. S'il n'existe pas de tel u, alors $K_M(w) = \infty$. Il s'agit donc d'une fonction de Σ^* dans $\mathbb{N} \cup \{\infty\}$.

Question 2. Soit M une machine sans préfixe. On fixera par convention $2^{-\infty} = 0$. Montrer que :

$$\sum_{w \in \Sigma^*} 2^{-K_M(w)} \leq 1$$

Une machine sans préfixe U est *universelle* si pour toute autre machine sans préfixe M, il existe une constante $c_M \in \mathbb{N}$ telle que pour tout mot w, on a $K_U(w) \leq K_M(w) + c_M$. On notera $K_U(w) \leq^+ K_M(w)$ pour dire que l'inégalité est vraie *à constante près*. On admettra l'existence d'une machine universelle U, et on notera $K(w)$ la complexité de Kolmogorov de w associée à U.

Question 3. Montrer que pour tout mot w, $K(w) \leq^+ |w| + 2\log_2(|w|)$.

Question 4. Montrer que pour tout palindrome w, $K(w) \leq^+ |w|/2 + 2\log_2(|w|)$.

Question 5. Montrer que pour tous mots u et v, $K(uv) \leq^+ K(u) + K(v)$.

Suite des questions

Question 6 Décrire un algorithme linéaire permettant de résoudre la question 0.

On appelle *ensemble de requêtes borné* tout ensemble $A \subseteq \Sigma^* \times \mathbb{N}$ tel que

$$\sum_{(w,\ell)\in A} 2^{-\ell} \leq 1$$

Question 7. (Théorème KC)
Soit A un ensemble infini de requêtes borné (on suppose qu'il existe un programme qui liste les éléments de A). Montrer qu'il existe une machine sans préfixe M telle que pour tout couple $(w, \ell) \in A$, il existe un mot u de longueur ℓ tel que $M(u) = w$. En particulier, $K_M(w) \leq \ell$.

Corrigé

Question 0. On peut utiliser le pseudocode suivant :

```
Entrée A, n
// Pour chaque paire de mots
Pour i = 0 à n-1:
   Pour j = 0 à i-1:
     // Les ordonner par taille
      Si |A(i)| <= |A(j)|:
         mot1 := A(i);
          mot2 := A(j);
      Sinon:
          mot1 := A(j);
          mot2 := A(i);
      Fin si

      // Si mot1 est prefixe de mot2, retourner faux
      egal = VRAI
      Pour k = 0 à |mot1|-1:
          Si mot1(k) != mot2(k):
             egal = FAUX
          Fin si
      Fin pour
      Si egal == VRAI:
         Renvoyer FAUX
      Fin si
   Fin pour
Fin pour

// Si pour aucune paire de mots, l'un est préfixe de l'autre
// Alors l'ensemble est sans préfixe
Renvoyer VRAI
```

Cet algorithme est en temps $\mathcal{O}(m * n)$ où n est le nombre de mots et m la somme des longueurs des mots. En effet, chaque lettre de chaque mot est comparée une fois à la lettre en même position de chaque autre mot.

Question 1. Indice : On peut identifier le mot w avec l'intervalle $I_w =]0.w0^\infty, 0.w1^\infty[\subseteq]0,1[$, de diamètre $2^{-|w|}$. Si A est sans préfixe, alors les intervalles I_w avec $w \in A$ sont deux à deux disjoints, donc la somme de leurs diamètres est inférieure ou égale au diamètre de $]0,1[$, soit 1.

Question 2. Pour tout mot w, soit u_w un mot minimal tel que $M(u_w) = w$, s'il existe. Notons $dom(M)$ le domaine de M et $Im(M)$ l'ensemble image de la machine M. Notons que l'ensemble $A = \{u_w : w \in Im(M)\}$ est sans préfixe puisqu'il est inclus dans $dom(M)$. Alors

$$\sum_{w\in\Sigma^*} 2^{-K_M(w)} = \sum_{w\in Im(M)} 2^{-|u_w|} = \sum_{u\in A} 2^{-|u|} \leq \sum_{u\in dom(M)} 2^{-|u|} \leq 1$$

La dernière inégalité est vraie par la question 1.

Question 3. ATTENTION, il faut créer une machine sans préfixe ! Pour tout mot u, soit $\bar{u}$ le mot où toutes les lettres sont doublées. Par exemple, si $u = 010$, $\bar{u} = 001100$. Soit M la machine qui pour tout mot de la forme $\bar{u}01w$, où u est la représentation binaire de $|w|$, s'arrête et retourne w ; sur toute entrée qui n'est pas de cette forme, M ne s'arrête pas. La machine M est manifestement sans préfixe. On a $K_M(w) = |w| + 2|u| + 2$, donc $K_M(w) \leq^+ |w| + 2\log_2(|w|)$. Par universalité de U, $K(w) \leq^+ |w| + 2\log_2(|w|)$.

Question 4. Chaque étape peut être donnée comme indice pour guider le candidat. Pour tout mot u, soit u' son mot inversé.

Étape 1 : Montrons que pour tout palindrome w de la forme uu', on a $K(w) \leq^+ K(u)$. Soit M la machine qui pour un mot v, exécute $U(v)$, et si $U(v) = u$, alors M renvoie uu'. Comme U est sans préfixe, alors M est sans préfixe. Montrons que $K_M(w) \leq K(u)$. En effet, soit v un mot de taille minimale tel que $U(v) = u$. Un tel mot existe par universalité de U et par la question précédente. On a alors $M(v) = uu'$, donc $K_M(w) \leq |v| = K(u)$. Par universalité de U, on a $K(w) \leq^+ K(u)$.

Étape 2 : Montrons que pour tout palindrome w de la forme uau' avec $a \in \Sigma$, on a $K(w) \leq^+ K(u)$. Soit M la machine qui pour un mot v, exécute $U(v)$, et si $U(v) = ua$ pour un mot u et une lettre $a \in \Sigma$, alors M renvoie uau'. Montrons que $K_M(w) \leq K(u)$. En effet, soit v un mot de taille minimale tel que $U(v) = ua$. On a alors $M(v) = uau'$, donc $K_M(w) \leq |v| = K(u)$. Par universalité de U, on a $K(w) \leq^+ K(u)$.

Étape 3 : Par l'étape 1 et 2, on a pour tout palindrome w, $K(w) \leq^+ K(u)$ pour un préfixe u de w tel que $|u| \leq |w|/2$. Par la question 3, $K(u) \leq^+ |u| + 2\log_2(|u|)$. On a donc $K(w) \leq^+ |w|/2 + 2\log_2(|w|/2)$, donc $K(w) \leq^+ |w|/2 + 2\log_2(|w|)$.

Question 5. Soit U la machine universelle sans préfixe. Soit M la machine qui pour un mot w, cherche deux mots u et v tels que $w = uv$ et pour lesquels $U(u)$ et $U(v)$ s'arrêtent, et renvoie $U(u)U(v)$. Si ces mots ne sont pas trouvés, $M(w)$ ne s'arrête pas. Notons que comme U est sans préfixe, si u et v existent, ils sont uniques. Montrons que M est sans préfixe. En effet, si $M(w)$ et $M(w')$ s'arrêtent, avec w préfixe de w', alors par définition de M, il existe des mots u, v, u', v' tels que $uv = w$, $u'v' = w'$, et U est définie pour chacun de ces mots. En particulier, u est préfixe de u' ou u' est préfixe de u, ce qui contredit le fait que U est sans préfixe. Montrons maintenant que $K_M(rs) \leq K(r) + K(s)$. En effet, soient r_0 et r_1 des mots de taille minimale tels que $U(r_0) = r$ et $U(s_0) = s$. Alors $M(r_0s_0) = rs$, or $|r_0| = K(r)$ et $|s_0| = K(s)$, donc $K_M(rs) \leq K(r) + K(s)$. Par universalité de U, $K(rs) \leq^+ K(r) + K(s)$.

Question 6. Sans forcément demander le code, voir si le candidat réussit à réinventer la structure de trie :

`https://fr.wikipedia.org/wiki/Trie_(informatique)`

Question 7. On pourra s'aider de la figure 1 pour comprendre la construction qui suit. Pour un ensemble fini F de couples (w, ℓ) où w est un mot fini et ℓ est un entier, on notera $\mathsf{poids}(F) = \sum_{(w,\ell)\in F} 2^{-\ell}$. Supposons sans perte de généralité que A est un ensemble infini $\{(w_s, \ell_s) : s \in \mathbb{N}\}$. Par hypothèse, on a $\mathsf{poids}(A) \leq 1$.

Nous allons décrire une machine M à l'aide de couples (σ, τ), signifiant alors $M(\sigma) = \tau$. À chaque étape de calcul s, on ajoutera un couple (σ, τ) dans M, et on notera M_s la machine obtenue jusqu'alors. De plus, à chaque étape de calcul s, pour chaque taille $\ell \geq 1$ on définira une chaine σ_s^ℓ, soit de taille ℓ, soit égale au mot vide ϵ, et un mot binaire infini $r_s \in \{0,1\}^\omega$. Les chaînes σ_s^ℓ différentes du mot vide correspondront aux chaînes disponibles pour une association à l'étape $s+1$. Le rôle de la suite $(r_s)_{s\in\mathbb{N}}$ est double. D'abord le réel représenté par r_s en base 2 sera égal au poids de M_s. Ensuite si le $(n-1)$-ème bit de r_s est 0, cela signifie aussi que la chaîne σ_s^n est différente de ϵ et est donc disponible pour une future association. On doit s'assurer à chaque étape s que :

(1) L'ensemble des chaînes associées dans M_s forment avec l'ensemble des chaînes σ_s^ℓ différentes du mot vide, un ensemble sans préfixe.

(2) r_s vue comme la représentation binaire d'un réel est égal à $\mathsf{poids}(M_s)$

(3) Si $r_s[n-1] = 0$, alors la chaîne σ_s^n est une chaîne de longueur n. Sinon c'est le mot vide.

A l'étape 0, on définit $\sigma_0^\ell = 1^{\ell-1}0$ et r_0 comme la suite infinie de 0. Les points (1), (2) et (3) sont vérifiés à l'étape 0.

A l'étape $s+1$, considérons le couple (τ_s, ℓ_s) de A. Si $r_s(\ell_s - 1) = 0$ on énumère $(\sigma_s^{\ell_s}, \tau_s)$ dans M à l'étape $s+1$, on assigne $\sigma_{s+1}^{\ell_s}$ au mot vide et on change $r_{s+1}(\ell_s - 1)$ à 1. Pour $i \neq \ell_s$ on garde $r_{s+1}(i-1) = r_s(i-1)$ et $\sigma_{s+1}^i = \sigma_s^i$. On vérifie facilement par induction que (1), (2) et (3) sont vrais à l'étape $s+1$.

Sinon si $r_s(\ell_s - 1) = 1$, soit n le plus grand entier strictement supérieur à 0 et plus petit que ℓ_s tel que $r_s(n-1) = 0$. Notons qu'un tel entier existe forcément car sinon $r_s + 2^{-\ell_s} \geq 1$ et donc $\mathsf{poids}(A) + 2^{-\ell_s} \geq 1$ ce qui est impossible par hypothèse. On assigne σ_{s+1}^n au mot vide et on change $r_s(n-1) = 1$. Ensuite pour chaque $n < i \leq \ell_s$, on assigne σ_{s+1}^i à $\sigma_s^n 1^{i-n-1}0$ et $r_{s+1}(i-1) = 0$. Puis on énumère $(\sigma_s^n 1^{\ell_s - n-1}1, \tau_s)$ dans M à l'étape $s+1$. Pour $1 \leq i < n$ et $i > \ell_s$ on définit $r_{s+1}(i-1) = r_s(i-1)$ et $\sigma_{s+1}^i = \sigma_s^i$. On vérifie aisément par induction que (1), (2) et (3) sont vrais à l'étape $s+1$.

Cela conclut la construction. Étant donné que (1) est vrai à chaque étape s il est clair que M est une machine sans préfixe. Également par construction on a $(\sigma, \ell) \in A$ implique $M(\tau) = \sigma$ pour une chaîne τ de taille ℓ.

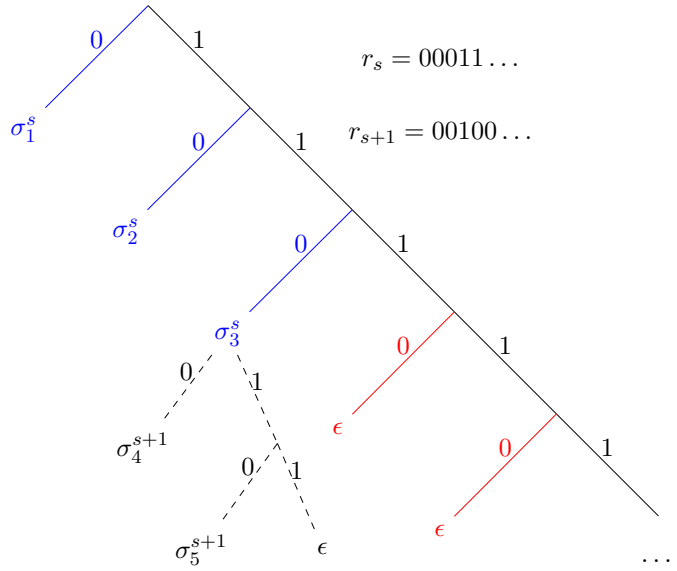


FIGURE 1 – Illustration de la preuve de la question 6. Sur la figure, à l'étape s les deux chaînes originelles de taille 4 et 5, en rouge, ont déjà été assignées. On suppose que l'on veut à présent assigner une chaîne de taille 5 à quelque chose à l'étape $s+1$. Le problème est qu'il n'y en a plus de disponible. On "remonte" alors jusqu'à la plus grande chaîne disponible, ici σ_3^s, et on "libère" dessous de nouvelles chaînes de taille 4 et 5 (en pointillé), il en reste alors une libre de taille 5, ici $\sigma_3^s 11$, pour une affectation à l'étape $s+1$. On remarque que les bits à 1 du réel r_{s+1} obtenu en rajoutant 2^{-5} à r_s correspondent exactement aux nouvelles chaînes disponibles.

L2 – Fonctions calculables

Fixons un langage de programmation raisonnable (parmi Caml, C, Java, Python). Une fonction $f : \mathbb{N} \to \mathbb{N}$ est *calculable* s'il existe un programme P qui, pour toute entrée n, s'arrête et renvoie $f(n)$. Un programme peut être représenté par un *mot binaire*, c'est-à-dire un mot fini dans l'alphabet $\{0,1\}$. On notera $\{0,1\}^*$ l'ensemble des mots binaires. On notera P_w le programme correspondant au mot w.

Question 0. Écrire en pseudocode un programme pour une fonction surjective $h : \mathbb{N} \to \{0,1\}^*$.

Question 1. Soit $f : \{0,1\}^* \to \{0,1\}$ une fonction telle que pour tout mot w, si P_w s'arrête sur l'entrée w, alors $f(w) \neq P_w(w)$. Montrer que f n'est pas calculable.

On supposera que le langage de programmation dispose d'une instruction $\texttt{exec}(w, u, t)$ qui prend en paramètre un mot binaire w, une entrée $u \in \{0,1\}^*$ et un temps $t \in \mathbb{N}$ d'exécution (ou étape de calcul). L'instruction $\texttt{exec}(w, u, t)$ exécute le programme P_w sur son entrée u pendant le temps t. Si $P_w(u)$ s'arrête avant le temps t, $\texttt{exec}(w, u, t)$ renvoie la valeur obtenue par ce programme. Si P_w ne s'arrête pas sur l'entrée u en moins de t étapes, $\texttt{exec}(w, u, t)$ renvoie un symbole spécial $\perp$.

Question 2. Imaginons que l'on dispose d'une fonction calculable $g : \{0,1\}^* \to \{0,1\}^*$ telle que :

(1) Pour tout mot w, $P_{g(w)} : \{0,1\}^* \to \{0,1\}$ s'arrête sur toutes ses entrées.

(2) Pour toute fonction calculable $f : \{0,1\}^* \to \{0,1\}$, il existe un mot w tel que $P_{g(w)}$ calcule f.

En déduire une contradiction.

Question 3. Montrer que pour toute fonction calculable $g : \{0,1\}^* \to \mathbb{N}$, il existe une fonction calculable $f : \{0,1\}^* \to \{0,1\}$ si complexe que pour tout programme P qui calcule f, il existe une entrée $u \in \{0,1\}^*$ sur laquelle $P(u)$ prend plus de $g(u)$ étapes de calcul.

Question 4. Soit $f : \{0,1\}^* \to \{0,1\}$ la fonction qui prend en entrée un mot w, et renvoie 1 s'il existe un temps t tel que $\texttt{exec}(w, w, t) \neq \perp$ et renvoie 0 sinon. Montrer que f n'est pas calculable.

Un ensemble $E \subseteq \{0,1\}^*$ est *décidable* si sa fonction caractéristique est calculable.

Question 5. Parmi ces ensembles, lesquels sont décidables ? Justifier

1. $\{w \in \{0,1\}^* :$ contient un nombre pair de 1 $\}$
2. $\{w \in \{0,1\}^* : P_w$ s'arrête sur l'entrée $w\}$
3. $\{w \in \{0,1\}^* : P_w$ s'arrête sur l'entrée w en moins de 3 étapes $\}$

Question 6. Montrer qu'il existe une fonction calculable $f : \mathbb{N} \to \{0,1\}^*$ dont l'ensemble image n'est pas décidable.

Question 7. Montrer que pour tout programme P qui s'arrête au moins sur une entrée, il existe une fonction calculable $f : \mathbb{N} \to \{0,1\}^*$ dont l'ensemble image est l'ensemble des entrées sur lesquelles P s'arrête.

Question 8. Montrer que pour toute fonction calculable $f : \mathbb{N} \to \{0,1\}^*$, il existe un programme P dont le domaine est l'ensemble image de f.

Corrigé

Question 0. L'idée est la suivante : associer n à sa représentation binaire. On peut utiliser le pseudo-code suivant :

```
binaire(n) :=

Si n = 0
 Renvoyer "";
Fin si

m = n >> 1 ; % division entiere par 2
Renvoyer binaire(m) + (n mod 2); % Le + est la concatenation
```

On a binaire(0) la chaine vide, et binaire(n) la représentation binaire de n pour n $>$ 0. Ce n'est pas suffisant, car les représentations binaires commencent toutes par un 1. On supprime le 1 du début.

```
Entree n;
Poser x::w = binaire(n+1)
Renvoyer w;
```

Question 1. *Si le candidat ne sait pas répondre à la Question 1, on pourra lui donner comme indice "Penser au paradoxe du menteur : si un homme dit que tous les Crétois sont des menteurs, cet homme peut-il être Crétois ?".*

Raisonnons par l'absurde. Supposons que f est calculable. Soit P_w un programme calculant f. Par définition, $f(w) \neq P_w(w)$, contradiction.

Question 2. *Indice : penser au paradoxe du menteur.* Soit $f : \{0,1\}^* \to \{0,1\}$ la fonction définie par $f(w) = 1 - \mathsf{exec}(g(w), w, \infty)$. La fonction serait non calculable pour les mêmes raisons que la question 1. En effet, si f est calculable, il existe un mot w tel que $P_{g(w)}$ calcule f. Alors $f(w) = 1 - \mathsf{exec}(g(w), w, \infty) = 1 - f(w)$.

Question 3. *Solution 1 : argument méta qui utilise la question 2.* Raisonnons par l'absurde. Supposons qu'il y ait une fonction calculable $g : \{0,1\}^* \to \mathbb{N}$ telle que pour toute fonction calculable $f : \{0,1\}^* \to \{0,1\}$, il existe un programme P_w qui calcule f et qui s'arrête sur toutes ses entrées u en un temps au plus $g(u)$. Alors il est possible de construire une fonction calculable $h : \{0,1\}^* \to \{0,1\}^*$ qui satisfait les propriétés de la question 2 comme suit : $h(w)$ teste si w est un programme syntaxiquement valide. Si ce n'est pas le cas, $h(w)$ renvoie le programme de la fonction qui retourne tout le temps 0. Si w est syntaxiquement valide, $h(w)$ est le code du programme $P_{h(w)}$ qui pour une entrée u, teste si $\mathsf{exec}(w, u, g(u)) \neq \perp$. Si c'est le cas, le programme renvoie $\mathsf{exec}(w, u, g(u))$, sinon, il renvoie 1. On obtient une contradiction avec la question 2.

Solution 2 : diagonalisation directe. Soit la fonction f qui, pour son entrée u, exécute $\mathsf{exec}(u, u, g(u))$. Si $\mathsf{exec}(u, u, g(u)) \notin \{0,1\}$, c'est-à-dire si $\mathsf{exec}(u, u, g(u)) \in \{\perp\} \cup \mathbb{N} \setminus \{0,1\}$, $f(u)$ peut être n'importe quelle valeur, et on fixe $f(u) = 0$. Si $\mathsf{exec}(u, u, g(u)) = i$ pour un $i \in \{0,1\}$, on fixe $f(u) = 1 - i$. Montrons que f satisfait la propriété désirée. Soit P un programme tel que $P(u)$ renvoie la valeur de $f(u)$ en au plus $g(u)$ étapes. Soit w la représentation du programme par un mot binaire. On a alors $\mathsf{exec}(w, w, g(w)) = P(w)$, et $f(w)$ est défini de telle sorte que $\mathsf{exec}(w, w, g(w)) \neq f(w)$. On a donc $P(w) \neq f(w)$.

Question 4. *Indice : penser au paradoxe du menteur.*

Raisonnons par l'absurde. Supposons que la fonction f soit calculable. Soit P un programme calculant f. Soit Q le programme qui pour un mot binaire w, exécute $P(w)$, puis récupère la valeur $i \in \{0,1\}$ renvoyée par $P(w)$, et effectue une boucle infinie si $i = 1$, et sinon s'arrête et renvoie 1 si $i = 0$. Soit w un mot binaire représentant Q. Si Q s'arrête sur l'entrée w, alors par définition de Q, $P(w) = 0$, donc par définition de P, P_w ne s'arrête pas sur w, contradiction. Si en revanche, Q ne s'arrête pas sur l'entrée w, alors par définition de Q, $P(w) = 1$, donc par définition de P, P_w s'arrête sur w. Nous avons encore une contradiction.

Question 5.

1. Oui. Il suffit de parcourir le mot, et maintenir un booléen qui change à chaque occurrence de 1 pour tester la parité du nombre de 1. En effet, si $P(0) = 1$ et Q est équivalent à P, alors $Q(0) = 1$.
2. Non. Sa fonction caractéristique est celle de la question 3. Nous avons vu qu'elle n'était pas calculable.
3. Oui. Il suffit de tester si $\mathsf{exec}(w, w, 3)$ renvoie $\perp$ ou non.

Question 6. Soit w_0 un mot binaire tel que P_{w_0} s'arrête sur w_0. Soit P le programme qui, pour une entrée n, cherche le plus petit mot binaire w de taille au plus n tel que $\mathsf{exec}(w, w, n) \neq \perp$ et qui n'est pas déjà dans l'image de P pour $0, 1, \ldots, n-1$. Si un tel mot est trouvé, $P(n) = w$. Sinon, $P(w) = w_0$. Notons que pour tout w dans l'image de P, P_w s'arrête sur w. Réciproquement, pour tout w tel que P_w s'arrête sur w, il existe un temps n_0 tel que P_w s'arrête sur w en temps n_0. Pour n suffisamment grand, tous les mots plus petits que w qui devront apparaître seront déjà apparus, et w aura la priorité. On a donc que l'image de la fonction calculée par P est $\{w : P_w \text{ s'arrête sur } w\}$, qui n'est pas décidable d'après la question 3.

Question 7. Il s'agit d'une simple variation de la question précédente.

Soit P_u un programme qui s'arrête sur une entrée w_0. Soit Q le programme qui pour une entrée n, cherche le plus petit mot binaire w de taille au plus n tel que $\mathsf{exec}(u, w, n) \neq \perp$ et qui n'est pas déjà dans l'image de Q pour $0, 1, \ldots, n-1$. Si un tel mot est trouvé, $Q(n) = w$. Sinon, $Q(w) = w_0$. Notons que pour tout w dans l'image de P, P_u s'arrête sur w. Réciproquement, pour tout w tel que P_u s'arrête sur w, il existe un temps n_0 tel que P_u s'arrête sur w en temps n_0. Pour n suffisamment grand, tous les mots plus petits que w qui devront apparaître seront déjà apparus, et w aura la priorité. On a donc que l'image de la fonction calculée par P est $\{w : P_u \text{ s'arrête sur } w\}$.

Question 8. Soit P_u un programme calculant f. Soit P le programme qui, sur l'entrée w, exécute successivement $\mathsf{exec}(u, n_0, n_1)$ pour tout $n_0, n_1 \in \mathbb{N}$, et s'arrête et renvoie 1 uniquement s'il trouve un n_0, n_1 tel que $\mathsf{exec}(u, n_0, n_1) = w$. Ainsi, si P s'arrête sur w, alors w est dans l'image de f. Réciproquement, si w est dans l'image de f, disons $f(n_0) = w$ pour un $n_0 \in \mathbb{N}$, soit n_1 le plus petit temps tel que P_u s'arrête sur n_0 en n_1 étapes. On a alors $\mathsf{exec}(u, n_0, n_1) = w$, donc P s'arrête sur w.

L3 – Fonctions primitives composables

On appelle *fonctions de base* les fonctions suivantes :

1. la fonction *constante* 0, notée $Z\colon \mathbb{N} \to \mathbb{N}$, et définie par $Z(x) = 0$
2. la fonction *successeur* $S\colon \mathbb{N} \to \mathbb{N}$, définie par $S(x) = x + 1$
3. les fonctions de *projection* $\pi_i^n : \mathbb{N}^n \to \mathbb{N}$ définies pour $0 \leq i < n \in \mathbb{N}$ par $\pi_i^n(x_0, \ldots, x_{n-1}) = x_i$

Soit $\mathcal{C}_0$ la classe des fonctions de base, et étant donné $\mathcal{C}_s$, soit $\mathcal{C}_{s+1}$ la classe qui contient les fonctions de $\mathcal{C}_s$, et telle que pour tous $n, k \in \mathbb{N}$, pour toutes fonctions $g_0, \ldots, g_{n-1} \in \mathcal{C}_s$ de type $\mathbb{N}^k \to \mathbb{N}$ et toute fonction $f \in \mathcal{C}_s$ de type $\mathbb{N}^n \to \mathbb{N}$, la fonction $h : \mathbb{N}^k \to \mathbb{N}$ définie par

$$h(x_0, \ldots, x_{k-1}) = f(g_0(x_0, \ldots, x_{k-1}), \ldots, g_{n-1}(x_0, \ldots, x_{k-1}))$$

appartient à $\mathcal{C}_{s+1}$. Soit $\mathcal{C}_\infty = \bigcup_s \mathcal{C}_s$ la plus petite classe de fonctions contenant les fonctions de base, et close par l'opération de composition définie ci-dessus.

Question 0. Montrer que pour toute fonction $f : \mathbb{N}^2 \to \mathbb{N}$ dans $\mathcal{C}_s$, la fonction $g : \mathbb{N}^2 \to \mathbb{N}$ définie par $g(x, y) = f(y, x)$ est dans $\mathcal{C}_{s+1}$.

Question 1. Montrer que pour toute fonction $f : \mathbb{N}^2 \to \mathbb{N}$ dans $\mathcal{C}_s$, la fonction $g : \mathbb{N}^2 \to \mathbb{N}$ définie par $g(x, y) = f(y, x)$ est en fait dans $\mathcal{C}_s$.

Question 2. Montrer que pour toute fonction $f \in \mathcal{C}_s$ de type $\mathbb{N}^n \to \mathbb{N}$, il existe un entier $0 \leq i < n$ et une fonction $g \in \mathcal{C}_s$ de type $\mathbb{N} \to \mathbb{N}$ telle que pour tout $x_0, \ldots, x_{n-1}$,

$$f(x_0, \ldots, x_{n-1}) = g(x_i)$$

Question 3. Montrer que pour toute fonction $g \in \mathcal{C}_\infty$ de type $\mathbb{N} \to \mathbb{N}$, il existe un $k \in \mathbb{N}$ tel que soit $g(x) = k$ pour tout $x \in \mathbb{N}$, soit $g(x) = x + k$ pour tout $x \in \mathbb{N}$.

Dans ce qui suit, nous nous intéressons à la représentation des fonctions de $\mathcal{C}_\infty$ par des mots finis. Un *codage de type* $\mathbb{N}^n \to \mathbb{N}$ est un mot fini dans l'alphabet $\{Z, S, \pi_i^n, C : i < n \in \mathbb{N}\}$ défini par récurrence comme suit : La lettre Z et la lettre S sont des codages de type $\mathbb{N} \to \mathbb{N}$. La lettre π_i^n est un codage de type $\mathbb{N}^n \to \mathbb{N}$. Si $g_0, \ldots, g_{n-1}$ sont des codages de type $\mathbb{N}^k \to \mathbb{N}$ et f est un codage de type $\mathbb{N}^n \to \mathbb{N}$, alors le mot $Cfg_0g_1 \ldots g_{n-1}$ est un codage de type $\mathbb{N}^k \to \mathbb{N}$. Un codage de type $\mathbb{N}^n \to \mathbb{N}$ correspond à une fonction de type $\mathbb{N}^n \to \mathbb{N}$ dans $\mathcal{C}_\infty$. Deux codages de type $\mathbb{N}^n \to \mathbb{N}$ sont *équivalents* s'ils définissent la même fonction.

Question 4. Donner un codage de la fonction $f : \mathbb{N}^2 \to \mathbb{N}$ définie par $f(x, y) = x + 2$.

Question 5. Écrire un pseudo-code qui prend un codage de type $\mathbb{N}^n \to \mathbb{N}$ en entrée, et renvoie n.

Question 6. Proposer une manière plus explicite de coder les fonctions de $\mathcal{C}_\infty$. Montrer qu'il existe un algorithme qui permet de traduire un codage de type $\mathbb{N}^n \to \mathbb{N}$ en sa forme plus explicite.

Question 7. Existe-t-il un algorithme qui décide si deux codages de type $\mathbb{N}^n \to \mathbb{N}$ sont équivalents ? Justifier.

Suite des questions

Soit $\mathcal{D}_0$ l'ensemble des fonctions de base, augmentée de l'addition $+ : \mathbb{N}^2 \to \mathbb{N}$. Soient $\mathcal{D}_1, \mathcal{D}_2, \ldots$ définies par récurrence comme précédemment, et $\mathcal{D}_\infty = \bigcup_s \mathcal{D}_s$.

Question 8. Formuler une hypothèse sur la forme explicite des fonctions de $\mathcal{D}_\infty$ et la prouver.

Corrigé

Question 0. $g(x,y) = f(\pi_1^2(x,y), \pi_0^2(x,y))$. Comme $\pi_1^2, \pi_0^2 \in \mathcal{C}_0$ et $f \in \mathcal{C}_s$, on a $g \in \mathcal{C}_{s+1}$.

Question 1. ATTENTION, toute la différence est que l'on veut montrer que g est dans $\mathcal{C}_s$ et non $\mathcal{C}_{s+1}$. Il faut prouver par induction sur s.

Pour $s = 0$, seules les projections π_0^2 et π_1^2 sont de type $\mathbb{N}^2 \to \mathbb{N}$, et sont mutuellement la solution l'une de l'autre.

Supposons l'hypothèse vraie pour s. Soit $h : \mathbb{N}^2 \to \mathbb{N}$ une fonction de $\mathcal{C}_{s+1}$ obtenue par composition de fonctions $g_0, \ldots, g_{n-1} \in \mathcal{C}_s$ de type $\mathbb{N}^2 \to \mathbb{N}$ et d'une fonction $f \in \mathcal{C}_s$ de type $\mathbb{N}^n \to \mathbb{N}$. Par hypothèse d'induction sur $g_0, \ldots, g_{n-1}$, pour tout $j < n$, il existe une fonction $g'_j \in \mathcal{C}_s$ telle que pour tout $g_j(x,y) = g'_j(y,x)$. En particulier, la fonction $h' : \mathbb{N}^k \to \mathbb{N}$ définie par

$$\begin{aligned} h'(x,y) &= f(g'_0(x,y), \ldots, g'_{n-1}(x,y)) \\ &= f(g_0(y,x), \ldots, g_{n-1}(y,x)) \\ &= h(y,x) \end{aligned}$$

est dans $\mathcal{C}_{s+1}$.

Question 2. Par induction sur s.

Pour $s = 0$, La propriété est trivialement satisfaite pour les fonctions Z et S en prenant pour g la même fonction. Pour les fonctions de projection π_i^n, la propriété est aussi vérifiée en prenant $g = \pi_0^1$ (la fonction identité).

Supposons l'hypothèse vraie pour s. Soit $h : \mathbb{N}^k \to \mathbb{N}$ une fonction de $\mathcal{C}_{s+1}$ obtenue par composition de fonctions $g_0, \ldots, g_{n-1} \in \mathcal{C}_s$ de type $\mathbb{N}^k \to \mathbb{N}$ et d'une fonction $f \in \mathcal{C}_s$ de type $\mathbb{N}^n \to \mathbb{N}$. Par hypothèse d'induction sur $g_0, \ldots, g_{n-1}$, pour tout $j < n$, il existe un $i_j < k$ et une fonction $h_j : \mathbb{N} \to \mathbb{N}$ dans $\mathcal{C}_s$ telle que pour tout $x_0, \ldots, x_{k-1}$,

$$g_j(x_0, \ldots, x_{k-1}) = h_j(x_{i_j})$$

et il existe un $j < n$ et une fonction $h_f : \mathbb{N} \to \mathbb{N}$ dans $\mathcal{C}_s$ telle que pour tout $x_0, \ldots, x_{n-1}$,

$$f(x_0, \ldots, x_{n-1}) = h_f(x_j)$$

Alors, soit $i = i_j$ et $h' = h_f \circ h_j$, on a pour tout $x_0, \ldots, x_{k-1}$,

$$\begin{aligned} h(x_0, \ldots, x_{k-1}) &= f(g_0(x_0, \ldots, x_{k-1}), \ldots, g_{n-1}(x_0, \ldots, x_{k-1})) \\ &= h_f(g_j(x_0, \ldots, x_{k-1})) \\ &= h_f(h_j(x_{i_j})) = h'(x_i) \end{aligned}$$

De plus $h' \in \mathcal{C}_{s+1}$.

Question 3. Par induction sur s.

Pour $s = 0$, les fonctions de base de type $\mathbb{N} \to \mathbb{N}$ sont la fonction constante, la fonction successeur, et la fonction de projection π_0^1 qui est la fonction identité. L'hypothèse est vérifiée pour chacun de ces cas.

Supposons l'hypothèse vraie pour s.

Soit $h : \mathbb{N} \to \mathbb{N}$ une fonction de $\mathcal{C}_{s+1}$ obtenue par composition de fonctions $g_0, \ldots, g_{n-1} \in \mathcal{C}_s$ de type $\mathbb{N} \to \mathbb{N}$ et d'une fonction $f \in \mathcal{C}_s$ de type $\mathbb{N}^n \to \mathbb{N}$. Par la question 2, il existe un $i < n$ et une fonction $g : \mathbb{N} \to \mathbb{N}$ dans $\mathcal{C}_s$ telle que pour tout $x_0, \ldots, x_{n-1}$,

$$f(x_0, \ldots, x_{n-1}) = g(x_i).$$

On a donc

$$h(x) = f(g_0(x), \ldots, g_{n-1}(x)) = g(g_i(x)).$$

Par hypothèse d'induction, comme $g \in \mathcal{C}_s$, il existe $k_0 \in \mathbb{N}$ tel que soit $g(x) = k_0$ pour tout x, soit $g(x) = x + k_0$ pour tout x. De même, il existe $k_1 \in \mathbb{N}$ tel que soit $g_i(x) = k_1$ pour tout x, soit $g_i(x) = x + k_1$ pour tout x.

— Si $g(x) = k_0$, alors $g(g_i(x)) = k_0$.

— Si $g(x) = x + k_0$ et $g_i(x) = k_1$, alors $g(g_i(x)) = k_0 + k_1$

— Si $g(x) = x + k_0$ et $g_i(x) = x + k_1$, alors $g(g_i(x)) = x + k_0 + k_1$

Question 4. Par exemple, $CSCS\pi_0^2$.

Question 5. Par abus de langage, on parlera d'arité d'un codage pour signifier celle de la fonction qu'il code. Montrons par récurrence sur la longueur d'un codage $|w|$ que son dernier caractère est une fonction de base, dont l'arité est celle de w.

Pour $|w| = 1$, soit $w \in \{Z, S\}$ (auquel cas $n = 1$), soit $w = \pi_i^m$, auquel cas $n = m$. Pour $|w| > 1$, w est de la forme $Cfg_0g_1 \ldots g_{n-1}$, auquel cas par hypothèse de récurrence, le dernier caractère de g_{n-1} est une fonction de base dont l'arité est celle de g_{n-1}, or l'arité de g_{n-1} est celle de w.

```
string arité(string codage) {
  Renvoyer codage[ longueur(codage) - 1]
}
```

Question 6. Par la question 2 et la question 3, toute fonction de type $\mathbb{N}^n \to \mathbb{N}$ dans $\mathcal{C}_\infty$ peut être représentée par un couple (n, k) ou par un triplet (n, i, k) avec $n, k \in \mathbb{N}$ et $i < k$. Ici, (n, i, k) code pour la fonction $f : \mathbb{N}^n \to \mathbb{N}$ telle que pour tout $x_0, \ldots, x_{n-1}$,

$$f(x_0, \ldots, x_{n-1}) = x_i + k$$

Et (n, k) code pour la fonction $f : \mathbb{N}^n \to \mathbb{N}$ telle que pour tout $x_0, \ldots, x_{n-1}$,

$$f(x_0, \ldots, x_{n-1}) = k$$

Il est possible de transformer algorithmiquement un codage de type $\mathbb{N}^n \to \mathbb{N}$ en notre représentation compacte. Pour cela, nous avons besoin d'une fonction qui, à un mot, retourne le plus petit codage qui est préfixe de ce mot. Dans ce qui suit, si m est un mot, $longueur(m)$ est la longueur de la chaine et $m[i..]$ est le mot tronqué des i premiers caractères.

```
string prefixe(string mot) {
  Si mot[0] \neq C alors
   Renvoyer mot[0];
```

```
  Fin si
  Si mot[0] = C alors
      mot = mot[1..] % Supprimer le C
      f = prefixe(mot);
     int res = "C" + f; % Concaténer C et f.
      int n = arité(f)
     Pour i = 0 à n-1
       g = prefixe(mot)
       mot = mot[longueur(g)..] % retirer le préfixe g du mot.
      res = res + g % Concaténer g au résultat
     Fin pour
     Renvoyer res; % Renvoyer C f g_0 g_1... g_{n-1}
  Fin si

  % Nous ne sommes pas censés arriver jusqu'ici
}
```

Nous pouvons maintenant définir la fonction de transformation en la forme compacte :

```
string transformer(string codage) {
  Si codage[0] = Z alors
   renvoyer (1, 0);
  Fin si
  Si codage[0] = S alors
   renvoyer (1, 0, 1)
  Fin si
  Si codage[0] = \pi^n_i alors
   renvoyer (n, i, 0)
  Fin si
  Si codage[0] = C alors
      f = prefixe(codage[1..]);
      codage = codage[longueur(f)+1 ..]; % Supprimer Cf du codage
      Si f = (n, k) alors
         Renvoyer (n, k); % S'il s'agit d'une constante, inutile d'evaluer
      Fin Si
      Si f = (n, i, k) alors
          Pour j = 0 à i faire
             g = prefixe(codage);
             codage = codage[longueur(g) .. ]; % Supprimer g du codage
          Fin pour;
          g = transformer(g);
          Si g = (n1, k1) alors
            renvoyer (n1, k+k1);
          Fin si
          Si g = (n1, i1, k1) alors
            renvoyer (n1, i1, k+k1);
          FinS Si
      Fin Si
  Fin si

  % Nous ne sommes pas censés arriver jusqu'ici
```

```
}
```

Question 7. Il existe un algorithme qui décide si deux codages de type $\mathbb{N}^n \to \mathbb{N}$ sont équivalents. En effet, il suffit de transformer chacun des codages en leur représentation compacte, puis tester si les représentations sont égales.

Question 8. Toute fonction $f : \mathbb{N}^n \to \mathbb{N}$ peut s'exprimer sous la forme

$$f(x_0, \ldots, x_{n-1}) = k + \sum_{i=0}^{n-1} k_i x_i$$

pour $k, k_0, \ldots, k_{n-1} \in \mathbb{N}$.

C'est le cas des fonctions de base et de l'addition. Il reste à la prouver pour la composition. Si

$$h(x_0, \ldots, x_{k-1}) = f(g_0(x_0, \ldots, x_{\ell-1}), \ldots, g_{n-1}(x_0, \ldots, x_{\ell-1}))$$

avec

$$f(x_0, \ldots, x_{n-1}) = k_f + \sum_{i=0}^{n-1} k_{f,i} x_i$$

et pour tout $i < n$,

$$g_0(x_0, \ldots, x_{\ell-1}) = k_i + \sum_{j=0}^{\ell-1} k_{i,j} x_j$$

alors

$$h(x_0, \ldots, x_{\ell-1}) = k_f + \sum_{i=0}^{n-1} k_{f,i} k_i + \sum_{i=0}^{n-1} \sum_{j=0}^{\ell-1} k_{f,i} k_{i,j} x_j = k_f + \sum_{i=0}^{n-1} k_{f,i} k_i + \sum_{j=0}^{\ell-1} \sum_{i=0}^{n-1} k_{f,i} k_{i,j} x_j$$

L4 – Fonctions primitives récursives

La classe $\mathcal{F}$ des *fonctions primitives récursives* est la plus petite classe contenant les *fonctions de base* :

1. la fonction *constante* 0 notée $Z : \mathbb{N} \to \mathbb{N}$ et définie par $Z(x) = 0$;
2. la fonction *successeur* notée $S : \mathbb{N} \to \mathbb{N}$ et définie par $S(x) = x + 1$;
3. les fonctions de *projection* $\pi_i^n : \mathbb{N}^n \to \mathbb{N}$ définies pour $0 \leq i < n$ par $\pi_i^n(x_0, \ldots, x_{n-1}) = x_i$;

et telle que

4. pour tous $n, k \in \mathbb{N}$, pour toutes fonctions $g_0, \ldots, g_{n-1} \in \mathcal{F}$ de type $\mathbb{N}^k \to \mathbb{N}$ et toute fonction $f \in \mathcal{F}$ de type $\mathbb{N}^n \to \mathbb{N}$, la fonction $h : \mathbb{N}^k \to \mathbb{N}$ de *composition* suivante appartient à $\mathcal{F}$:

$$h(x_0, \ldots, x_{k-1}) = f(g_0(x_0, \ldots, x_{k-1}), \ldots, g_{n-1}(x_0, \ldots, x_{k-1}))\,;$$

5. pour tout $n \in \mathbb{N}$, pour toute fonction $f \in \mathcal{F}$ de type $\mathbb{N}^n \to \mathbb{N}$, et toute fonction $g \in \mathcal{F}$ de type $\mathbb{N}^{n+2} \to \mathbb{N}$, la fonction $h : \mathbb{N}^{n+1} \to \mathbb{N}$ de *récursion primitive* suivante appartient à $\mathcal{F}$:

$$\begin{aligned} h(x_0, \ldots, x_{n-1}, 0) &= f(x_0, \ldots, x_{n-1}) \\ h(x_0, \ldots, x_{n-1}, k+1) &= g(x_0, \ldots, x_{n-1}, k, h(x_0, \ldots, x_{n-1}, k)). \end{aligned}$$

Question 0. Montrer en détail que l'addition et la multiplication sont dans $\mathcal{F}$.

Question 1. Montrer (sans détailler) que les fonctions suivantes sont dans $\mathcal{F}$:

(a) x^y ;
(b) $x \dot{-} 1$ (où $x \dot{-} 1 = 0$ si $x = 0$, $x - 1$ sinon) ;
(c) $x \dot{-} y$ (où $x \dot{-} y = 0$ si $y \geq x$, $x - y$ sinon) ;
(d) $\mathrm{sg}(x)$ qui vaut 0 si $x = 0$ et 1 si $x \neq 0$;
(e) $\overline{\mathrm{sg}}(x)$ qui vaut 0 si $x \neq 0$ et 1 sinon ;
(f) $|x - y|$;
(g) $\min(x, y)$;
(h) $\mathrm{rm}(x, y)$ (le reste de la division de x par y, avec $\mathrm{rm}(x, 0) = x$) ;
(i) $\mathrm{qt}(x, y)$ (la division entière de x par y, avec $\mathrm{qt}(x, 0) = 0$) ;
(j) $\mathrm{div}(x, y)$ qui vaut 1 si x divise y et 0 sinon.

Question 2. Montrer que $\mathcal{F}$ est close par *minimisation bornée* : Si $f : \mathbb{N}^{n+1} \to \mathbb{N}$ est dans $\mathcal{F}$, alors on peut aussi trouver dans $\mathcal{F}$ la fonction $h : \mathbb{N}^{n+1} \to \mathbb{N}$ telle que $h(x_0, \ldots, x_{n-1}, y)$ est le plus petit $0 \leq z < y$ tel que $f(x_0, \ldots, x_{n-1}, z) = 0$ si un tel z existe, et sinon $h(x_0, \ldots, x_{n-1}, y)$ vaut y.

Question 3. Donner le pseudo-code d'une fonction qui prend en paramètre la description d'une fonction primitive récursive $f : \mathbb{N}^k \to \mathbb{N}$ ainsi que k entiers $a_0, \ldots, a_{k-1}$, et retourne $f(a_0, \ldots, a_{k-1})$.

Question 4. Donner un argument justifiant qu'il existe une fonction de type $\mathbb{N} \to \mathbb{N}$ que l'on peut programmer dans un langage de programmation raisonnable (C, Java, OCaml, ...) qui n'est pas dans $\mathcal{F}$.

Soit $\mathcal{A} : \mathbb{N}^2 \to \mathbb{N}$ la *fonction d'Ackermann* définie par

$$\mathcal{A}(0, y) = y + 1 \qquad \mathcal{A}(x + 1, 0) = \mathcal{A}(x, 1) \qquad \mathcal{A}(x + 1, y + 1) = \mathcal{A}(x, \mathcal{A}(x + 1, y))$$

Pour tout $x \in \mathbb{N}$, on note $\mathcal{A}_x : \mathbb{N} \to \mathbb{N}$ la fonction $y \mapsto \mathcal{A}(x, y)$.

Question 5. Donner une forme explicite aux fonctions $\mathcal{A}_0$, $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ et montrer que pour tout x, $\mathcal{A}_x$ est dans $\mathcal{F}$.

Question 6. Montrer que pour tous x, y, $\mathcal{A}_x(y) > y$, $\mathcal{A}_x(y+1) > \mathcal{A}_x(y)$ et $\mathcal{A}_{x+1}(y) > \mathcal{A}_x(y)$.

Suite des questions

Pour tous x et k, soit $\mathcal{A}_x^k$ la k-ième composée de $\mathcal{A}_x$, où $\mathcal{A}_x^0$ est la fonction identité, et $\mathcal{A}_x^{k+1} = \mathcal{A}_x \circ \mathcal{A}_x^k$.

Question 7. Montrer que pour tous x, y et k, $\mathcal{A}_x^{k+1}(y) > \mathcal{A}_x^k(y)$, $\mathcal{A}_x^k(y) \geq y$, et si $x' \leq x$, alors $\mathcal{A}_{x'}^k(y) \leq \mathcal{A}_x^k(y)$.

Question 8. Montrer que pour tous x, y et k, on a $\mathcal{A}_x^k(y) \leq \mathcal{A}_{x+1}(y+k)$.

Soit $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N}^n \to \mathbb{N}$. La fonction f *domine* g, noté $g \prec f$ s'il existe une constante C telle que $\forall (x_0, \ldots, x_{n-1}) \in \mathbb{N}^n$,

$$g(x_0, \ldots, x_{n-1}) \leq f(\sup(x_0, \ldots, x_{n-1}, C))$$

Question 9. Pour tout $x \geq 0$, soit $\mathcal{C}_x = \{g \in \mathcal{F} : \exists k \in \mathbb{N}, g \prec \mathcal{A}_x^k\}$. Montrer que $\mathcal{F} \subseteq \bigcup_x \mathcal{C}_x$.

Corrigé

Question 0. ATTENTION, on demande d'être très formel et de tout détailler pour cette question.

L'addition est définie par

$$x + 0 = x$$
$$x + (y+1) = S(x+y)$$

La fonction $f : \mathbb{N} \to \mathbb{N}$ définie par $f(x) = x$ est la fonction de projection π_0^1, donc est dans $\mathcal{F}$. Soit $g : \mathbb{N}^3 \to \mathbb{N}$ la fonction définie par $g(x, y, z) = S(z)$. Cette fonction est dans $\mathcal{F}$ car $g = S \circ \pi_2^3$. Par récursion primitive avec f et g, la fonction d'addition est dans $\mathcal{F}$.

La multiplication est définie par

$$x \times 0 = 0$$
$$x \times (y+1) = x + (x \times y)$$

La fonction $f : \mathbb{N} \to \mathbb{N}$ définie par $f(x) = 0$ est la fonction constante Z, donc est dans $\mathcal{F}$. Soit $g : \mathbb{N}^3 \to \mathbb{N}$ la fonction définie par $g(x, y, z) = x + z$. Elle est obtenue en composant des projections avec l'addition, donc est dans $\mathcal{F}$. Par récursion primitive avec f et g, la fonction de multiplication est dans $\mathcal{F}$.

Question 1.

(a) x^y est définie par $x^0 = 1$ et $x^{y+1} = x^y \times x$.

(b) $x \dot{-} 1$ est définie par $0 \dot{-} 1 = 0$ et $(x+1) \dot{-} 1 = x$.

(c) $x \dot{-} y$ est définie par $x \dot{-} 0 = x$ et $x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1$.

(d) $\mathrm{sg}(x)$ est définie par $\mathrm{sg}(0) = 0$ et $\mathrm{sg}(x+1) = 1$.

(e) $\overline{\mathrm{sg}}(x)$ est définie par $1 \dot{-} \mathrm{sg}(x)$.

(f) $|x - y|$ est définie par $(x \dot{-} y) + (y \dot{-} x)$.

(g) $\min(x, y)$ est définie par $x \dot{-} (x \dot{-} y)$.

(h) $\mathrm{rm}(x, y)$ satisfait $\mathrm{rm}(x+1, y) = \begin{cases} \mathrm{rm}(x,y)+1 & \text{si } \mathrm{rm}(x,y)+1 \neq y \\ 0 & \text{sinon} \end{cases}$.

On a donc $\mathrm{rm}(0, x) = 0$ et $\mathrm{rm}(x+1, y) = (\mathrm{rm}(x, y) + 1) \times \mathrm{sg}(y \dot{-} (\mathrm{rm}(x, y) + 1))$.

(i) $\mathrm{qt}(x, y)$ satisfait $\mathrm{qt}(x+1, y) = \begin{cases} \mathrm{qt}(x,y)+1 & \text{si } \mathrm{rm}(x,y)+1 = y \\ \mathrm{qt}(x,y) & \text{sinon} \end{cases}$.

On a donc $\mathrm{qt}(0, x) = 0$ et $\mathrm{qt}(x+1, y) = \mathrm{qt}(x, y) + \overline{\mathrm{sg}}(y \dot{-} (\mathrm{rm}(x, y) + 1))$.

(j) On a $\mathrm{div}(x, y) = \overline{\mathrm{sg}}(\mathrm{rm}(y, x))$.

Question 2. $h(x_0, \ldots, x_{n-1}, 0) = 0$ et

$$h(x_0, \ldots, x_{n-1}, y+1) = \begin{cases} h(x_0, \ldots, x_{n-1}, y) & \text{si } f(x_0, \ldots, x_{n-1}, h(x_0, \ldots, x_{n-1}, y)) = 0 \\ y+1 & \text{sinon} \end{cases}$$

On a donc $h(x_0, \ldots, x_{n-1}, y+1) = h(x_0, \ldots, x_{n-1}, y) \times \overline{\text{sg}}(f(x_0, \ldots, x_{n-1}, h(x_0, \ldots, x_{n-1}, y))) + (y+1) \times \text{sg}(f(x_0, \ldots, x_{n-1}, h(x_0, \ldots, x_{n-1}, y)))$. La fonction h est dans $\mathcal{F}$ par récursion primitive.

Question 3.

```
% Évalue la fonction code sur une liste d'entiers args
int eval(code, args) {
  match code with
  | Zero -> 0
  | Succ -> args[0]+1
  | Proj n i -> args[i]
  | Comp n f gs -> eval(f, eval_args(gs, args))
  | Rec n f g ->
      match args[n-1] with
      | 0 -> eval(f, args[0..n-2])
      | k+1 -> eval(g, args[0..n-2] @ [k] @ eval(Rec n f g, args[0..n-2] @ [k]))
      end
}

% Évalue une liste de codes en leur passant à tous les mêmes arguments
list(int) eval_args(codes, args) {
  match codes with
  | [] -> []
  | f :: codes -> eval(f, args) :: eval_args(codes, args)
}
```

Question 4. Il est possible de lister calculatoirement toutes les fonctions $f_0, f_1, \ldots$ de type $\mathbb{N} \to \mathbb{N}$ dans $\mathcal{F}$. Mais alors il est possible de programmer la fonction $h : \mathbb{N} \to \mathbb{N}$ qui satisfait $h(x) = f_x(x) + 1$. La fonction h est différente de toutes les fonctions de type $\mathbb{N} \to \mathbb{N}$ de $\mathcal{F}$. En effet, si h est dans $\mathcal{F}$, il existe un n tel que $h = f_n$. Mais alors $h(n) = f_n(n) + 1 \neq f_n(n) = h(n)$. Contradiction.

Question 5. On a $\mathcal{A}_0(y) = y + 1$, $\mathcal{A}_1(y) = y + 2$, $\mathcal{A}_2(y) = 2y + 3$, $\mathcal{A}_3(y) = 2^{n+3} - 3$. En effet :

- $\mathcal{A}_0(y) = y + 1$ par définition.
- $\mathcal{A}_1(0) = \mathcal{A}_0(1) = 2$ et $\mathcal{A}_1(y+1) = \mathcal{A}_0(\mathcal{A}_1(y)) = \mathcal{A}_1(y) + 1$
- $\mathcal{A}_2(0) = \mathcal{A}_1(1) = 3$ et $\mathcal{A}_2(y+1) = \mathcal{A}_1(\mathcal{A}_2(y)) = \mathcal{A}_2(y) + 2$
- $\mathcal{A}_3(0) = \mathcal{A}_2(1) = 5$ et $\mathcal{A}_3(y+1) = \mathcal{A}_2(\mathcal{A}_3(y)) = 2\mathcal{A}_2(y) + 3$ C'est une suite arithmético-géométrique. On rappelle que si $u_{n+1} = au_n + b$ avec $a \neq 1$ et ℓ est la solution de l'équation $a\ell + b = \ell$, alors $u_n = \ell + (u_0 - \ell)a^n$. Ici, $a = 2$, $b = 3$, $\ell = -3$ et $u_n = -3 + (5+3)2^n = -3 + 8 \times 2^n = -3 + 2^{n+3}$.

On prouve par récurrence sur n que $\mathcal{A}_n \in \mathcal{F}$. $\mathcal{A}_0$ est la fonction successeur, et est donc dans $\mathcal{F}$. Supposons que $\mathcal{A}_n \in \mathcal{F}$. $\mathcal{A}_{n+1}(0) = \mathcal{A}_n(1)$ et $\mathcal{A}_{n+1}(y+1) = \mathcal{A}_n(\mathcal{A}_{n+1}(y))$. Par le schéma de récursion primitive, $\mathcal{A}_{n+1} \in \mathcal{F}$.

Question 6. Montrons par récurrence sur x que $\mathcal{A}_x(y) > y$. Pour $x = 0$, $\mathcal{A}_0(y) = y + 1 > y$, donc le cas de base est vrai. Supposons que $\mathcal{A}_x(y) > y$ pour tout y, et montrons par récurrence sur y que $\mathcal{A}_{x+1}(y) > y$. Pour $y = 0$, $\mathcal{A}_{x+1}(0) = \mathcal{A}_x(1)$, or $\mathcal{A}_x(1) > 1$ par hypothèse de récurrence, donc $\mathcal{A}_{x+1}(0) > 0$. Supposons que $\mathcal{A}_{x+1}(y) > y$. Alors $\mathcal{A}_{x+1}(y + 1) = \mathcal{A}_x(\mathcal{A}_{x+1}(y))$, or $\mathcal{A}_x(\mathcal{A}_{x+1}(y)) > \mathcal{A}_{x+1}(y)$ par première hypothèse de récurrence, et $\mathcal{A}_{x+1}(y) > y$ par seconde hypothèse de récurrence. Ainsi, $\mathcal{A}_{x+1}(y + 1) > y + 1$ (deux inégalités strictes consécutives sur des entiers).

Montrons que pour tous x et y, $\mathcal{A}_x(y+1) > \mathcal{A}_x(y)$. Pour $x = 0$, $\mathcal{A}_0(y+1) = y+2 > y+1 = \mathcal{A}_0(y)$. Pour $x > 0$, $\mathcal{A}_x(y + 1) = \mathcal{A}_{x-1}(\mathcal{A}_x(y))$, or par l'inégalité précédente, $\mathcal{A}_{x-1}(\mathcal{A}_x(y)) > \mathcal{A}_x(y)$, donc $\mathcal{A}_x(y + 1) > \mathcal{A}_x(y)$.

Montrons que pour tous x et y, $\mathcal{A}_{x+1}(y) > \mathcal{A}_x(y)$. Pour $x = 0$, $\mathcal{A}_1(y) = y+2 > y+1 = \mathcal{A}_0(y)$. Pour $x > 0$, $\mathcal{A}_{x+1}(y) = \mathcal{A}_x(\mathcal{A}_{x+1}(y))$, or $\mathcal{A}_{x+1}(y) > y$ par la première inégalité prouvée, donc $\mathcal{A}_x(\mathcal{A}_{x+1}(y)) > \mathcal{A}_x(y)$ par la seconde inégalité prouvée. Donc $\mathcal{A}_{x+1}(y) > \mathcal{A}_x(y)$.

Question 7. Montrons que $\mathcal{A}_x^{k+1}(y) > \mathcal{A}_x^k(y)$. En effet, $\mathcal{A}_x^{k+1}(y) = \mathcal{A}_x(\mathcal{A}_x^k(y)) > \mathcal{A}_x^k(y)$ par la question 6.

Montrons par récurrence sur k que $\mathcal{A}_x^k(y) \geq y$. Pour $k = 0$, $\mathcal{A}_x^0$ est la fonction identité, donc $\mathcal{A}_x^0(y) = y$. Supposons que $\mathcal{A}_x^k(y) \geq y$. Alors $\mathcal{A}_x^{k+1}(y) = \mathcal{A}_x(\mathcal{A}_x^k(y)) \geq \mathcal{A}_x^k(y) \geq y$ par la question 6.

Montrons par récurrence sur k que pour tous x, x', y, si $x' \leq x$, alors $\mathcal{A}_{x'}^k(y) \leq \mathcal{A}_x^k(y)$. Pour $k = 0$, $\mathcal{A}_{x'}^0$ et $\mathcal{A}_x^0$ sont toutes les deux la fonction identité, donc $\mathcal{A}_{x'}^0(y) \leq \mathcal{A}_x^0(y)$. Supposons que $\mathcal{A}_{x'}^k(y) \leq \mathcal{A}_x^k(y)$. Alors par la question 6 (croissance de $\mathcal{A}_{x'}$),

$$\mathcal{A}_{x'}(\mathcal{A}_{x'}^k(y)) \leq \mathcal{A}_{x'}(\mathcal{A}_x^k(y)) \leq \mathcal{A}_x(\mathcal{A}_x^k(y)).$$

Donc

$$\mathcal{A}_{x'}^{k+1}(y) = \mathcal{A}_{x'}(\mathcal{A}_{x'}^k(y)) \leq \mathcal{A}_x(\mathcal{A}_x^k(y)) = \mathcal{A}_x^{k+1}(y).$$

Question 8. Par récurrence sur k. Pour $k = 0$, $\mathcal{A}_x^0$ est la fonction identité, et par la question 6, $\mathcal{A}_{x+1}(y + 0) > y = \mathcal{A}_x^0(y)$. Supposons que $\mathcal{A}_x^k(y) \leq \mathcal{A}_{x+1}(y + k)$. Par la question 6 (croissance de $\mathcal{A}_x$), on a $\mathcal{A}_x^{k+1}(y) = \mathcal{A}_x(\mathcal{A}_x^k(y)) \leq \mathcal{A}_x(\mathcal{A}_{x+1}(y + k))$. Par la définition de la fonction d'Ackermann, $\mathcal{A}_{x+1}(y + k + 1) = \mathcal{A}_x(\mathcal{A}_{x+1}(y + k))$. Ainsi, $\mathcal{A}_x^{k+1}(y) \leq \mathcal{A}_{x+1}(y + k + 1)$.

Question 9 La preuve est disponible ici :

`http://www.discmath.ulg.ac.be/cours/slides01-cc.pdf`