Banque MP inter-ENS – Session 2021

Rapport relatif à l'épreuve orale d'informatique fondamentale spécifique Ulm

— **Écoles partageant cette épreuve :** ENS Ulm

— **Coefficients** (en pourcentage du total des points de chaque concours) :
  — Concours MP option MPI : 23,15%
  — Concours INFO : 13,33%

— **Membres du jury :** A. Amarilli, J.-H. Jourdan, L. Patey, P. Senellart

Modalités de l'épreuve. L'épreuve orale d'informatique fondamentale décrite dans ce rapport est spécifique au concours d'entrée de l'École normale supérieure de Paris, et est entièrement indépendante de l'épreuve analogue qui figure au concours des autres Écoles normales supérieures. L'épreuve dure une heure, sans préparation, et vise à interroger les candidates et candidats sur des questions d'informatique fondamentale au tableau. Elle couvre des notions d'informatique principalement théoriques, mais diffère d'une épreuve de mathématiques en cela que les sujets conduisent en l'étude de notions propres à l'informatique, tels que des algorithmes ou programmes, des langages formels, des graphes, etc. Cette épreuve ne mesure pas la compétence des candidates et candidats en informatique pratique, même s'il leur est souvent demandé de présenter certains points en pseudocode.

Les épreuves orales de l'année 2020 n'ont pas eu lieu en raison de la pandémie de COVID-19, mais les épreuves de cette année se sont tenues, dans le respect d'un protocole sanitaire.

Les sujets sont présentés aux candidates et candidats sous forme imprimée, et quelques minutes leur sont laissées pour prendre connaissance des définitions préliminaires et des premières questions. Les examinateurs ont généralement laissé les candidates et candidats traiter le début des sujets en autonomie, en progressant naturellement vers un dialogue interactif pour des questions plus délicates, ou quand il s'avérait nécessaire de préciser certains points des réponses proposées. Les sujets étaient toujours composés d'un unique problème formé de plusieurs questions successives ; pour celles et ceux qui parvenaient à la fin des questions imprimées, les questions suivantes étaient posées directement à l'oral au tableau.

Cette année, exceptionnellement, le protocole sanitaire ne permettait pas d'accueillir du public. En temps normal, l'épreuve est publique, mais il est demandé aux spectateurs et spectatrices de solliciter l'accord du candidat ou de la candidate pour éviter toute gêne. Les membres du public doivent rester silencieux pendant l'épreuve et ne peuvent pas interférer avec son déroulement.

Résultats. Cette année, le jury a examiné 73 candidates et candidats admissibles aux concours MP et INFO. Le jury n'a pas connaissance de quel concours est présenté par les candidates et candidats qu'il auditionne, et les évalue donc de la même manière. Conformément aux instructions fournies pour l'harmonisation, les notes se sont réparties de 1,9 à 19,0, avec une moyenne de $11,08$ et un écart type de 3,95.

Programme. L'épreuve porte sur le programme de l'option informatique des *deux* années de classes préparatoires (MPSI et MP), ainsi que sur le programme informatique commun, et peut également faire appel à des compétences mathématiques exigibles suivant les programmes de cette discipline. Il est fortement recommandé aux candidates et candidats de *prendre connaissance de ces programmes* et de s'assurer qu'ils et elles maîtrisent effectivement les points qui y figurent. Comme les années précédentes,

nous avons posé des questions de cours, et trop de candidates et de candidats se trompent sur les notions du programme, voire avouent ne pas les maîtriser, ce qui est fortement pénalisé.

Bien entendu, les sujets proposés aux candidates et aux candidats leur demandaient d'explorer des notions nouvelles, qui allaient au-delà du programme, et qui étaient donc définies rigoureusement dans le sujet proposé. Dans l'ensemble, les candidates et candidats se sont bien appropriés ces notions. Les doutes qu'ils et elles ont pu formuler à leur sujet, lorsqu'ils étaient explicitement formulés et relevaient d'une méprise compréhensible, n'étaient généralement pas pénalisés.

Sujets Comme les années précédentes, par souci de transparence, et pour permettre à toutes les candidates et à tous les candidats de préparer cette épreuve de manière équitable, ce rapport de concours inclut en annexe l'intégralité des sujets posés cette année [1]. Soulignons toutefois que certaines questions étaient suffisamment difficiles pour qu'il ne soit pas envisageable de les résoudre sans aide. De même, les sujets n'incluent pas ici certaines questions plus délicates qui étaient posées à l'oral une fois que les questions du sujet imprimé avaient été résolues.

Critères d'évaluation. Comme les années précédentes, les examinateurs ont évalué la capacité des candidates et candidats à comprendre le sujet correctement, si possible sans aide, et à se forger une intuition des notions abstraites qui y étaient présentées de manière formelle. Ils ont évalué leur réponse aux questions, notamment leur aptitude à proposer des idées de résolution, à explorer des directions prometteuses, et à réagir aux indications du jury ; mais aussi à exposer leur raisonnement de façon synthétique et compréhensible à l'oral, ou de manière plus rigoureuse à l'écrit au tableau si cela était demandé. Ils ont également évalué la maîtrise des algorithmes et structures de données au programme, ainsi que la capacité à écrire au tableau certaines routines simples en pseudocode (exploration d'arbre, etc.).

À nouveau, la performance des candidates et candidats s'est distinguée suivant certaines dimensions indépendantes. Toutes les candidates et tous les candidats ont réussi à atteindre une compréhension satisfaisante des définitions préalables des problèmes étudiés, mais parfois cela s'est fait sans aide, et parfois il a fallu guider davantage. Pour des questions qui demandaient de formaliser une preuve, par exemple par récurrence, certaines candidates et certains candidats savent articuler la preuve soigneusement à l'oral comme à l'écrit, mais d'autres, même en ayant compris l'intuition de la question, ne parviennent pas à la formaliser de façon convaincante. Les meilleures candidates et les meilleurs candidats sont celles et ceux qui parviennent à convaincre à l'oral, de façon synthétique, qu'ils ou elles ont compris les arguments clés et savent structurer la preuve ; et qui parviennent également à écrire rapidement et rigoureusement une preuve formelle au tableau lorsque l'examinateur en fait la demande.

Pour des questions algorithmiques, il était parfois demandé d'écrire le pseudocode d'algorithmes simples. Les meilleures candidates et les meilleurs candidats n'hésitent pas à poser des questions pertinentes (par exemple, comment représente-t-on les arbres, les graphes ?), et adoptent du recul sur le code qu'ils ou elles proposent ; d'autres se perdent dans ces algorithmes pourtant classiques. De manière générale, il est conseillé de connaître à l'avance la notion de pseudocode et d'avoir un peu d'entraînement dans la programmation au tableau, et notamment pour vérifier les indices et des points de bon sens : l'algorithme peut-il effectivement terminer, quelles valeurs peut-il renvoyer, quelles valeurs sont possibles pour les paramètres, quel est le type d'objets manipulés, les noms sont-ils cohérents, les cas de base sont-ils traités, le comportement sur les petites valeurs est-il cohérent, etc.

Face à des questions plus ardues, on observe également une grande diversité de réactions. Bien sûr, les meilleures candidates et les meilleurs candidats sont celles et ceux qui proposent d'emblée la bonne approche, ou qui savent interpréter rapidement les indices donnés par l'examinateur, et parviennent ainsi à régler de telles questions avec très peu d'aide. De manière plus générale, les examinateurs ont apprécié que le candidat ou la candidate propose des pistes, avec un recul critique toutefois (c'est-à-dire,

1. Des propositions de corrections de ces sujets sont susceptibles d'être mises en ligne par les membres du jury sur leurs sites Web respectifs, indépendamment et à titre personnel.

en sachant estimer si l'approche a des chances d'aboutir), et si possible avec vivacité et enthousiasme. À défaut d'inspiration, il est toujours préférable d'engager le dialogue avec l'examinateur, ou de proposer des exemples simples à étudier. Les candidates et candidats qui restent mutiques, et qui bloquent sans communiquer avec l'examinateur sur les difficultés qu'ils ou elles rencontrent, ont souvent reçu des notes plus basses.

Recommandations aux candidates et candidats. En termes de programme, il est recommandé aux candidates et candidats de s'assurer qu'ils ou elles maîtrisent les points suivants, sur lequel le jury a identifié certaines lacunes, comme les années précédentes :

- Algorithmes dynamiques : il faut savoir présenter l'algorithme informellement, en expliquant quelles sont les valeurs calculées, dans quel ordre elles sont calculées, et quel espace mémoire elles occupent ; il faut aussi savoir formaliser ces intuitions en pseudocode.
- Fonctions récursives, mémoïsation : Il faut savoir repérer quand on veut calculer une quantité qui peut être définie récursivement, traduire cela en fonction récursive (sans oublier le cas de base), mémoïser la fonction de façon autonome (sans se tromper sur l'utilisation du tableau), déterminer la complexité de la fonction mémoïsée, comprendre le lien avec un algorithme dynamique pour effectuer le même calcul.
- Parcours d'arbres, de graphes. Une fois fixée une représentation des arbres ou des graphes, il faut savoir écrire sans erreur un pseudocode pour les explorer et pour calculer des quantités simples (par exemple la hauteur d'un arbre). Il faut savoir distinguer parcours en profondeur et parcours en largeur (et savoir quelle structure de données est utilisée par chaque), savoir écrire leur pseudocode avec la bonne structure de données, connaître leur complexité, et savoir quand employer quel parcours.
- Plus courts chemins dans un graphe : algorithmes de Dijkstra et de Floyd–Warshall. Il faut connaître ces deux algorithmes, savoir les distinguer, et savoir quand employer quel algorithme. Il faut savoir écrire un pseudocode pour l'algorithme de Dijkstra avec une file de priorités implémentée à l'aide d'un tas stocké dans un tableau. Il faut savoir quand ces algorithmes sont applicables (poids négatifs, cycles de poids négatif).
- Arbres binaires de recherche : il faut connaître leur invariant, savoir qu'ils peuvent être utilisés pour implémenter un dictionnaire, savoir écrire le pseudocode des fonctions d'insertion, de suppression et de lecture dans un arbre binaire de recherche, et connaître leur complexité. En particulier, il faut savoir que, sans l'utilisation d'une méthode d'équilibrage (non exigible), ces opérations n'ont pas une complexité logarithmique dans le pire cas.
- Tas : il faut connaître l'invariant des tas (et ne pas le confondre avec celui des arbres binaires de recherche...), savoir écrire un pseudocode pour les opérations d'insertion et d'extraction du minimum dans un tas réalisé à l'aide d'un tableau, et connaître leur complexité.
- Automates : constructions pour l'intersection (automate produit), pour la complémentation, pour la déterminisation, pour la transformation d'une expression rationnelle en automate.

En termes de méthode, le jury formule les recommandations suivantes :

- Savoir adopter le bon niveau de détail : présenter informellement à l'oral les grandes lignes d'une solution, mais ne pas rechigner à écrire le détail au tableau sur demande de l'examinateur.
- Ne pas se laisser désorienter par des questions sur des points de cours ou sur des points apparemment faciles ou purement mécaniques dans le traitement d'une question. L'examinateur est susceptible de demander des détails sur des points « évidents » ou des définitions du cours juste pour vérifier que le candidat ou la candidate a bien compris, et ceci perturbe parfois des candidates et candidats rapides qui ne s'imaginent pas qu'on puisse douter de leur compréhension de ces points de base.
- Plutôt que de rester silencieusement bloqué sur une question, réfléchir à voix haute, et communiquer avec l'examinateur sur les difficultés rencontrées. À défaut, étudier des exemples, proposer d'étudier des versions plus faibles de la question, etc. Trop de candidates et candidats ne réfléchissent pas à voix haute et restent silencieux, même quand on leur demande de communiquer sur leurs idées.

— Savoir écrire du pseudocode, distinguer quelles opérations ont un coût élémentaire et lesquelles sont plus coûteuses (comparaison de chaîne, définition d'ensembles), vérifier les indices, l'initialisation des structures de données, les bornes des boucles, les valeurs de retour possibles. En particulier, pour les candidates et candidats qui écrivent du pseudocode en Python :
  — l'opération de *slicing*, permettant de récupérer une partie contiguë d'un tableau, n'est *pas* en temps constant ;
  — il ne faut pas invoquer les « listes à tout faire » de Python, mais présenter son code avec des structures de données dont on comprend bien le fonctionnement et la complexité.
— Travailler à présenter ses idées à l'oral de façon claire et synthétique, et également au tableau de manière soignée et organisée.
— Connaître son cours, et être prêt à exposer de façon synthétique les points au programme.

Annexe. La suite de ce document présente l'intégralité des sujets qui ont été posés, sous la forme des feuilles distribuées aux candidates et candidats.

A1 – Profondeur de diagrammes de décision

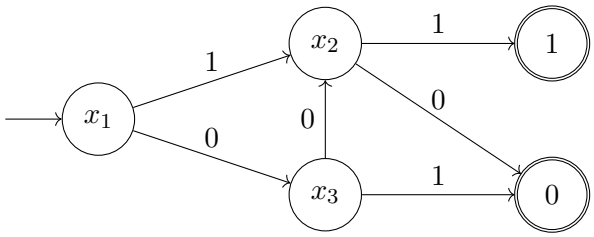
On fixe un ensemble $X = \{x_1, \dots, x_n\}$ de variables booléennes. Un *diagramme de décision* D sur X est la donnée d'un graphe orienté (V, E), supposé sans cycle, d'un nœud initial $v_{\text{init}} \in V$, d'une partition de V en $V = V_0 \sqcup V_1 \sqcup V'$, d'une partition de E en en $E = E_0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

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 $u_0, \ldots, u_k, v_1, \ldots, v_k \in \Sigma^*$.

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.

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 ?

J1 – Arbres évasés

Question 0.

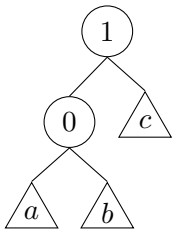
(a) Rappeler ce qu'est un arbre binaire de recherche.

(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.

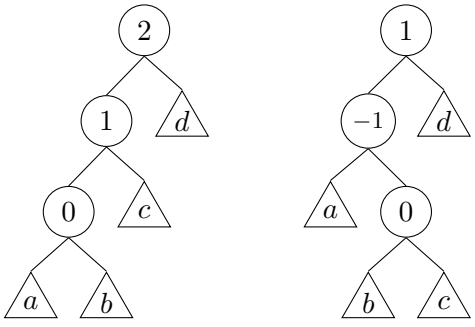
(a) En déduire un algorithme qui prend en paramètre un arbre binaire de recherche et l'étiquette de l'un de ses nœuds et transforme cet arbre binaire de recherche en déplaçant ce nœud à la racine. On supposera que toutes les étiquettes sont distinctes.

(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* rg(a) est défini par $\mathrm{rg}(a) = \log_2(\mathrm{taille}(a))$. 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.

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 $O(1)$.

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 ?

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, $K(uv) \leq^+ K(u) + K(v)$.

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.

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.

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 $\mathcal{A}_{x+1}(y) > \mathcal{A}_x(y)$.

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.

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 : $O\left(\max\left(\max_{u \text{ nœud de } H} \delta(u), \max_{u \text{ nœud de dépile}(H)} \delta(u)\right)\right)$.

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.