Rapport de l'épreuve orale d'informatique fondamentale

École normale supérieure
Concours MP et INFO 2018
Épreuve spécifique Ulm

Jury : Antoine Amarilli et Jacques-Henri Jourdan

Coefficients (en pourcentage du total d'admission) : concours MP (option MPI) 23,1% - concours Info 13,3%

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 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 la vaste majorité des sujets conduit à l'étude d'un algorithme et de sa complexité. Cette épreuve ne mesure pas la compétence des candidats en informatique pratique, même si nous avons demandé aux candidats de présenter certains points en pseudocode.

Cette année, comme l'an dernier, les sujets étaient présentés au candidat sous forme imprimée, et quelques minutes étaient laissées au candidat pour prendre connaissance des définitions préliminaires et des premières questions. Les examinateurs ont généralement laissé les candidats traiter le début des sujets par eux-mêmes, 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 les candidats qui parvenaient à la fin des questions imprimées, les questions suivantes étaient posées directement à l'oral au tableau.

L'épreuve est publique, mais il est demandé aux spectateurs de solliciter l'accord du candidat afin de ne pas le gêner. Les spectateurs doivent rester silencieux pendant l'épreuve et ne peuvent pas interférer avec son déroulement.

Résultats. Cette année, le jury a examiné 101 candidats admissibles aux concours MP et INFO. Le jury n'a pas connaissance de quel concours est présenté par les candidats qu'il auditionne, ainsi les candidats des deux concours sont-ils évalués de la même manière. Conformément aux instructions fournies pour l'harmonisation, les notes se sont réparties de 5 à 18.5, avec une moyenne de 12.02 et un écart type de 3.41.

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. Nous recommandons fortement aux candidats de prendre connaissance de ces programmes et de s'assurer qu'ils maîtrisent effectivement les points qui y figurent. Bien entendu, les sujets proposés aux candidats leur demandaient d'explorer des notions nouvelles, qui allaient au-delà du programme, et qui étaient donc définies rigoureusement dans le sujet que nous leur proposions. Dans l'ensemble, les candidats se sont bien approprié ces notions. Les doutes des candidats à leur sujet, lorsqu'ils étaient explicitement formulés et relevaient d'une méprise compréhensible, n'étaient généralement pas pénalisés.

Sujets Les sujets proposés cette année se sont concentrés plus particulièrement sur certains thèmes : graphes orientés et non-orientés, arbres, automates finis et langages réguliers, programmation dynamique, algorithmes de tri, probabilités et comptage.

Comme l'an dernier, par souci de transparence, et pour permettre à tous les candidats de préparer cette épreuve de manière équitable, nous avons inclus, en annexe de ce rapport de concours, l'intégralité des sujets que nous avons posés aux candidats cette année [1]. Soulignons toutefois que certaines questions étaient suffisamment difficiles pour que nous ne nous attendions pas à ce que les candidats puissent les traiter sans aide, même pour les meilleurs d'entre eux ; à l'inverse, nous n'incluons pas ici certaines questions plus délicates que nous avons posées à l'oral une fois que les questions imprimées avaient été résolues.

Critères d'évaluation. Comme l'an dernier, nous avons mesuré la capacité des 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. Nous avons 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. Nous avons évalué à quel point les candidats maîtrisaient les algorithmes et structures de données au programme, ainsi que leur capacité à écrire au tableau certaines routines simples en pseudocode (exploration d'arbre, etc.).

La performance des candidats s'est distinguée suivant certaines dimensions indépendantes. Tous les candidats ont réussi à atteindre une compréhension satisfaisante des définitions préalables du problème étudié, mais certains y sont parvenus seuls, et d'autres ont eu besoin d'être davantage guidés. Pour des questions où nous demandions au candidat de formaliser une preuve, par exemple par récurrence, certains candidats savent articuler la preuve soigneusement à l'oral comme à l'écrit, mais d'autres, alors même qu'ils ont compris l'intuition de la question, ne parviennent pas à la formaliser de façon convaincante. Les meilleurs candidats sont ceux qui parviennent à convaincre à l'oral, de façon synthétique, qu'ils ont compris les arguments clés et qu'ils 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, nous avons parfois demandé aux candidats d'écrire le pseudocode d'algorithmes simples, ce qui les a parfois désarçonnés. 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 proposent (ils savent notamment reconnaître un parcours en largeur, en profondeur) ; les candidats moins bons se perdent dans ces algorithmes pourtant classiques.

Face à des questions plus ardues, on observe également une grande diversité de réactions. Bien sûr, les meilleurs candidats sont ceux qui proposent d'emblée la bonne approche, ou qui savent interpréter rapidement les indices que nous leur donnons, et parviennent ainsi à régler de telles questions avec très peu d'aide. De manière plus générale, nous avons apprécié que le candidat propose des pistes, avec un recul critique toutefois (c'est-à-dire, 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 candidats qui restent mutiques, et qui bloquent sans communiquer avec l'examinateur sur les difficultés qu'ils rencontrent, ont souvent reçu des notes plus basses.

Un facteur important de la variation entre les performances des candidats était lié à l'intuition que ceux-ci s'étaient formés des notions présentées dans le sujet. Souvent, ces notions étaient définies formellement, mais sans intuition : il était important de s'en former une, mais surtout de savoir en changer si elle s'avérait inadaptée pour les questions qui se posaient ensuite. Certains candidats, après un début prometteur, sont restés bloqués, apparemment parce que leur compréhension intuitive du sujet obscurcissait certains points nécessaires pour traiter les questions posées par la suite.

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

Un autre point important pour l'évaluation est la connaissance des notions au programme et la réaction lorsque nous avons posé des questions de cours : les meilleurs candidats savent donner une présentation correcte et synthétique des points exigibles, les bons candidats parviennent à rassembler leurs idées avec quelques hésitations, et les mauvais candidats montrent qu'ils ont mal compris certains points, ou parfois sèchent carrément. Les candidats qui ont reçu les plus mauvaises notes sont ceux qui n'ont pas su traiter le problème posé et qui ne connaissaient pas non plus leur cours, mais même de bons candidats se sont pénalisés par des erreurs. De manière générale, on peut regretter que plusieurs candidats aient avoué n'avoir absolument aucune connaissance de certains points du programme.

Recommandations aux candidats. En termes de programme, nous conseillons aux candidats de s'assurer qu'ils maîtrisent les points suivants, sur lequel nous avons identifié certaines lacunes :

- 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, 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).
- Tas : il faut connaître l'invariant des tas, 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 : construction pour l'intersection (automate produit), pour la complémentation, pour la déterminisation, pour la transformation d'une expression rationnelle en automate.
- Il faut savoir déterminer si une structure de données est persistante ou impérative.
- Il faut savoir expliquer comment un automate fini lit un mot fourni en entrée ; on s'attend notamment à ce que le candidat sache décrire un algorithme qui détermine, étant donné un mot et un automate fini, si le mot est accepté ou rejeté par l'automate. Il faut pouvoir préciser la complexité de cette tâche, dans le cas où l'automate est déterministe comme dans le cas où il ne l'est pas : cette complexité s'exprime en général en fonction du mot *et* de l'automate. Il faut savoir comment on représente les transitions d'un automate déterministe *et* non-déterministe (représentations par une fonction, ou par un ensemble de transitions).

En termes de méthode, nous formulons 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. Bien souvent, les candidats qui échafaudent des arguments trop savants ou trop compliqués se retrouvent en difficulté pour répondre à des questions trop simples.
- Ne pas se bloquer sur une unique façon de voir les notions proposées dans le sujet ; essayer d'adopter une autre vision si nécessaire.

— 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 des affaiblissements de la question, etc. Trop de 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, extraction de sous-chaînes ou *slicing* en Python, définition d'ensembles), vérifier les indices, l'initialisation des structures de données, les bornes des boucles, les valeurs de retour possibles.
— Ne pas ignorer les indications de l'examinateur.
— 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. À défaut, il vaut mieux communiquer ce que l'on sait (ou croit savoir) ; garder le silence est extrêmement pénalisant.

Annexe. Dans la suite de ce document, nous incluons l'intégralité des sujets qui ont été posés, sous la forme des feuilles distribuées aux candidats.

A1 – Évaluation accélérée d'automates

On suppose donnés un alphabet Σ, un automate fini déterministe complet A sur Σ, et un mot $w = a_0 \cdots a_{n-1}$ de Σ^*. Pour $0 \leq i < j \leq n$, on notera $w[i, j]$ le facteur $a_i a_{i+1} \cdots a_{j-1}$ de w.

L'objet du problème est de répondre efficacement à des *requêtes* de la forme suivante : étant donnés deux indices i et j avec $0 \leq i < j \leq n$, déterminer si $w[i, j]$ est accepté par A.

Question 0. Donner le pseudo-code d'un algorithme qui, étant donné une requête, y répond. Préciser sa complexité en temps et en espace.

Dans la suite de ce problème, on va chercher à concevoir, étant donnés A et w, une structure de données appelée *index* qui permette de répondre plus efficacement aux requêtes.

Question 1. Expliquer comment calculer à partir de A et w un index de taille $O(n^2)$ qui permette de répondre à toute requête en temps $O(1)$.

Question 2. Quelle est la complexité en temps de calculer l'index de la question 1 ?

Question 3. Notons Q l'ensemble d'états de l'automate A, et $\delta : Q \times \Sigma \to Q$ sa fonction de transition. On étend δ à une fonction $\delta^* : Q \times \Sigma^* \to Q$ par récurrence en posant $\delta^*(q, \epsilon) := q$ et $\delta^*(q, au) := \delta^*(\delta(q, a), u)$. On appelle *effet de transition* du mot $u \in \Sigma^*$ la fonction $f_u : Q \to Q$ définie par $f_u(q) := \delta^*(q, u)$ pour tout $q \in Q$.

Décrire l'effet de transition du mot vide ϵ. Pour tous mots $u, v \in \Sigma^*$, exprimer f_{uv} en fonction de f_u et de f_v.

Question 4. On supposera pour simplifier que la longueur $|w|$ de w est une puissance de 2. On dit que deux nombres $i < j$ sont des *multiples consécutifs d'une même puissance de 2* si $j - i$ est une puissance de 2 qui divise i et j.

En s'inspirant de la question 3, proposer un algorithme de type "diviser pour régner" qui calcule f_u pour tous les facteurs $u = w[i, j]$ de w où i et j sont des multiples consécutifs d'une même puissance de 2. Analyser la complexité en temps de ce calcul.

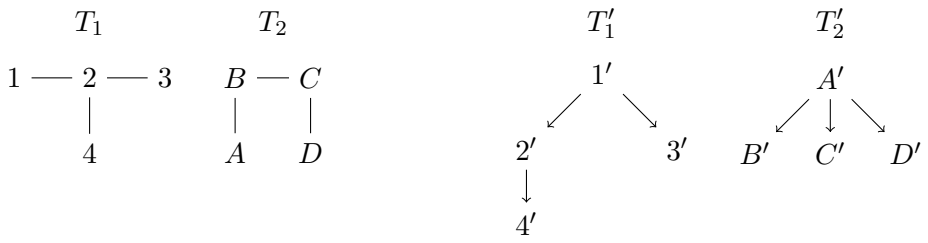
Question 5. Présenter un algorithme qui utilise l'index de la question 4 pour répondre à des requêtes, et préciser sa complexité en temps. Conclure.

A2 – Homomorphismes d'arbres

On rappelle qu'un *arbre* est un graphe non-orienté connexe acyclique. Un *arbre enraciné* T est obtenu en prenant un arbre, en distinguant un sommet de l'arbre appelé *racine*, et en orientant les arêtes de sorte que la racine n'ait aucune arête entrante et chaque sommet ait au plus une arête entrante.

Un *homomorphisme d'arbres* d'un arbre T_1 vers un arbre T_2 est une application h de l'ensemble des nœuds de T_1 vers celui des nœuds de T_2 qui satisfait la condition suivante : pour toute arête $\{u, v\}$ de T_1, la paire $\{h(u), h(v)\}$ est une arête de T_2. Un *homomorphisme d'arbres enracinés* h d'un arbre enraciné T_1 vers un arbre enraciné T_2 est une application h de l'ensemble des nœuds de T_1 vers celui des nœuds de T_2 telle que l'image par h de la racine de T_1 est la racine de T_2, et telle que, pour toute arête (u, v) de T_1, le couple $(h(u), h(v))$ est une arête de T_2.

Question 0. Construire un homomorphisme de l'arbre T_1 vers l'arbre T_2. Y a-t-il un homomorphisme de l'arbre enraciné T_1' vers l'arbre enraciné T_2' ?



Question 1. Proposer un algorithme qui, étant donné un arbre T_1, construit un homomorphisme de T_1 vers l'arbre comportant deux sommets et une unique arête joignant ces deux sommets. En déduire un algorithme qui, étant donné deux arbres T_1 et T_2, détermine s'il existe un homomorphisme de T_1 vers T_2, et en calcule un le cas échéant.

Question 2. Proposer un algorithme qui, étant donné deux arbres enracinés T_1 et T_2, détermine s'il existe un homomorphisme de T_1 vers T_2, et en calcule un le cas échéant.

Question 3. On fixe un ensemble de couleurs C. Un *graphe C-colorié* est un graphe G et une application qui associe à chaque arête $\{u, v\}$ de G une couleur de C. Un *graphe C-colorié orienté* est défini de la même manière mais les arêtes sont des couples (u, v). On étend ces définitions pour parler d'arbres et arbres enracinés C-coloriés. On définit un homomorphisme h d'arbres C-coloriés d'un arbre T_1 à un arbre T_2 en imposant également que la couleur de toute arête $\{u, v\}$ de T_1 soit la même que celle de son image par h dans T_2, et on définit de même la notion d'homomorphisme d'arbres enracinés C-coloriés.

Proposer un algorithme qui, étant donné deux arbres enracinés C-coloriés T_1 et T_2, détermine s'il existe un homomorphisme de T_1 vers T_2, et en calcule un le cas échéant.

Question 4. Proposer un algorithme qui, étant donné deux arbres C-coloriés T_1 et T_2, détermine s'il existe un homomorphisme de T_1 vers T_2, et en calcule un le cas échéant.

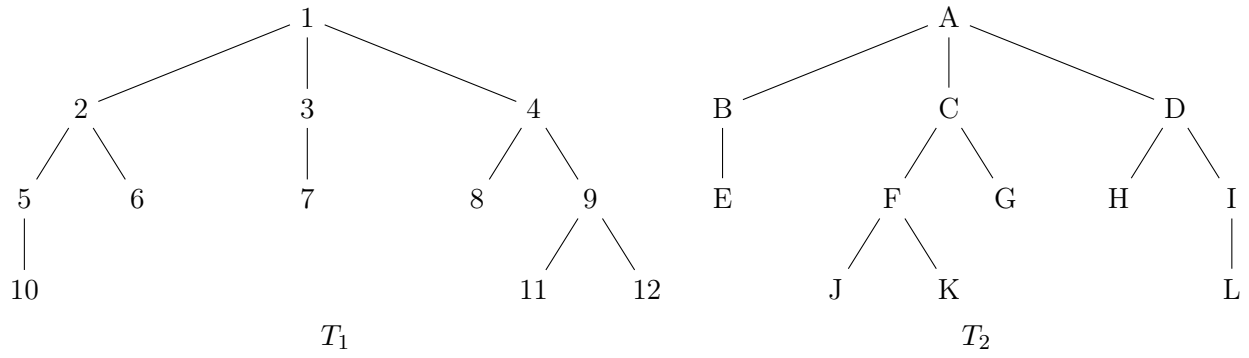
A3 – Isomorphisme d'arbres

Un *isomorphisme* d'un arbre T_1 vers un arbre T_2 est une bijection ι des nœuds de T_1 vers ceux de T_2 qui satisfait les conditions suivantes :

— On a $\iota(r_1) = r_2$, où r_1 et r_2 sont les racines respectives de T_1 et de T_2

— Pour tout nœud n_1 de T_1 qui n'est pas la racine, pour p_1 le parent de n_1, on a que $\iota(n_1)$ est un enfant de $\iota(p_1)$.

Question 0. Donner un isomorphisme de T_1 vers T_2 pour l'exemple suivant. Cet isomorphisme est-il unique ?



T_1 T_2

Question 1. On définit une relation sur les arbres en disant que T_1 et T_2 sont *isomorphes* s'il existe un isomorphisme de T_1 à T_2. Montrer qu'il s'agit d'une relation d'équivalence.

Question 2. On suppose que les arbres T_1 et T_2 sont *binaires*, c'est-à-dire que chaque nœud interne a exactement deux enfants. Proposer un algorithme qui détermine si T_1 et T_2 sont isomorphes, et calcule un isomorphisme le cas échéant. Préciser sa complexité en temps et en espace.

Question 3. Généraliser cet algorithme au cas où T_1 et T_2 ne sont plus binaires, et discuter de la complexité en temps et en espace.

Question 4. On suppose à nouveau que T_1 et T_2 sont binaires, et on cherche un algorithme plus efficace. Étant donné deux séquences d'entiers naturels s_1 et s_2, on note $\min(s_1, s_2)$ la plus petite de ces séquences dans l'ordre lexicographique, on note $\max(s_1, s_2)$ la plus grande, et on note $s_1 \odot s_2$ la séquence obtenue en concaténant s_1 et s_2. Pour $i \in \mathbb{N}$, on note (i) la séquence d'entiers singleton comportant un seul terme égal à i.

Le *code* $\chi(T)$ d'un arbre binaire T est la séquence d'entiers définie inductivement comme suit :

— Si T est une feuille, alors on a $\chi(T) := (1)$

— Si T n'est pas une feuille, alors soit n la racine de T, soit T' le sous-arbre enraciné au premier enfant de n, soit T'' le sous-arbre enraciné au second enfant de n, soit v le nombre de sommets de T, soit $\kappa' := \chi(T')$, et soit $\kappa'' := \chi(T'')$, alors on a $\chi(T) := (v) \odot \min(\kappa', \kappa'') \odot \max(\kappa', \kappa'')$.

Montrer qu'il existe un isomorphisme de T_1 vers T_2 si et seulement si $\chi(T_1) = \chi(T_2)$.

A4 – Automates et palindromes

On fixe un alphabet Σ avec $|\Sigma| > 1$. Un mot $w \in \Sigma^*$ est un *palindrome* s'il s'écrit $w = a_1 \cdots a_n$ et qu'on a $a_i = a_{n-i+1}$ pour tout $1 \leq i \leq n$. On note $\Pi \subseteq \Sigma^*$ le langage des palindromes. Pour un automate fini A sur Σ, on note $L(A)$ le langage reconnu par A.

Question 0. Soit $\Pi_n := \Pi \cap \Sigma^n$. Montrer que pour tout automate fini déterministe complet A, pour tout $n \in \mathbb{N}$, si $L(A) \cap \Sigma^{2n} = \Pi_{2n}$, alors A a au moins $|\Sigma|^n$ états.

Question 1. En déduire que le langage Π n'est pas régulier.

Question 2. Étant donné un automate fini A sur Σ, peut-on calculer un automate A_Π qui reconnaisse $L(A) \cap \Pi$?

Question 3. Pour tout mot $u = b_1 \cdots b_m$ de Σ^*, on note $\overline{u} := b_m \cdots b_1$ son miroir. Étant donné A, peut-on calculer un automate A'_Π qui reconnaisse $\{u \in \Sigma^* \mid u\overline{u} \in L(A)\}$?

Question 4. On appelle Π_{pair} l'ensemble des palindromes de longueur paire, i.e., $\Pi_{\text{pair}} := \bigcup_{n\in\mathbb{N}} \Pi_{2n}$. Proposer un algorithme qui, étant donné un automate fini A sur Σ, détermine si $L(A) \cap \Pi_{\text{pair}}$ est vide, fini, ou infini. Discuter de sa complexité en temps et en espace.

Question 5. Modifier l'algorithme de la question 4 pour calculer la cardinalité de $L(A) \cap \Pi_{\text{pair}}$ quand cet ensemble est fini, en faisant l'hypothèse que l'automate d'entrée A est déterministe. Comment la complexité est-elle affectée ?

Question 6. Modifier l'algorithme des questions 4 et 5 pour qu'il s'applique à $L(A) \cap \Pi$.

A5 – Réparation de mots pour un langage régulier

On fixe un langage régulier L sur un alphabet Σ. Étant donné un mot $w \in \Sigma^*$, une *réparation par insertion* du mot w pour le langage L est une décomposition $w = uv$ de w en deux mots, et un mot $z \in \Sigma^*$, de sorte que le mot uzv appartienne à L.

Question 0. On pose L_0 le langage régulier défini par l'expression rationnelle $(ab)^*$. Proposer une réparation par insertion du mot $w_0 = aab$ pour L_0. Proposer une réparation par insertion du mot $w_0' = abab$ pour L_0. Le mot $w_0'' = aa$ admet-il une réparation par insertion pour L_0 ?

Question 1. Une *réparation par insertion finale* d'un mot $w \in \Sigma^*$ pour un langage régulier L est un mot $z \in \Sigma^*$ tel que wz appartienne à L. Proposer un algorithme qui, étant donné w et L, détermine s'il existe une réparation par insertion finale de w pour L. Préciser sa complexité en temps et en espace.

Question 2. On souhaite modifier l'algorithme de la question 1 pour que, lorsqu'une réparation par insertion finale existe, l'algorithme calcule un z correspondant qui soit de longueur minimale. Expliquer comment procéder, et préciser la complexité en temps et en espace.

Question 3. Proposer un algorithme qui, étant donné un mot $w \in \Sigma^*$ et un langage régulier L, détermine s'il existe une réparation par insertion de w pour L. Préciser sa complexité en temps et en espace.

Question 4. Modifier l'algorithme de la question 3 pour que, lorsqu'une réparation par insertion existe, l'algorithme calcule une décomposition $w = uv$ et un z correspondant qui soit de longueur minimale. Préciser la complexité en temps et en espace.

Question 5. Une *réparation par suppression* d'un mot $w \in \Sigma^*$ pour un langage L est une décomposition $w = uzv$ de w de sorte que $uv \in \Sigma^*$. Proposer un algorithme naïf qui, étant donné w et L, détermine s'il existe une réparation par suppression de w pour L, et si oui, calcule une telle suppression telle que z soit de longueur minimale. Préciser sa complexité en temps et en espace.

Question 6. Proposer un algorithme plus efficace pour déterminer, étant donné w et L, s'il existe une réparation par suppression de w pour L. La complexité de l'algorithme doit être linéaire en w.

Question 7. Modifier l'algorithme de la question 6 pour que, quand une réparation par suppression existe, l'algorithme calcule une décomposition $w = uzv$ et un z correspondant qui soit de longueur minimale. On demande toujours une complexité linéaire en w.

A6 – Clôture par sur-mots et sous-mots

On fixe un alphabet Σ. Étant donné deux mots $w, w' \in \Sigma^*$, on dit que w' est un *sur-mot* de w, noté $w \preccurlyeq w'$, s'il existe une fonction strictement croissante ϕ de $\{1, \ldots, |w|\}$ dans $\{1, \ldots, |w'|\}$ telle que $w_i = w'_{\phi(i)}$ pour tout $1 \leq i \leq |w|$, où $|w|$ dénote la longueur de w et w_i dénote la i-ème lettre de w. Étant donné un langage L, on note $\overline{L}$ le langage des sur-mots de mots de L, c'est-à-dire $\overline{L} := \{w' \in \Sigma^* \mid \exists w \in L, w \preccurlyeq w'\}$.

Question 0. On pose L_0 le langage défini par l'expression rationnelle ab^*a, et L_1 le langage défini par l'expression rationnelle $(ab)^*$. Donner une expression rationnelle pour $\overline{L_0}$ et pour $\overline{L_1}$.

Question 1. Montrer que, pour tout langage L, on a $\overline{\overline{L}} = \overline{L}$.

Question 2. Existe-t-il des langages L' pour lesquels il n'existe aucun langage L tel que $\overline{L} = L'$?

Question 3. Montrer que, pour tout langage régulier L, le langage $\overline{L}$ est également régulier.

Question 4. On admettra pour cette question le résultat suivant : pour toute suite $(w_n)_{n\in\mathbb{N}}$ de mots de Σ^*, il existe $i < j$ tels que $w_i \preccurlyeq w_j$.

Montrer que, pour tout langage L (non nécessairement régulier), il existe un langage fini $F \subseteq L$ tel que $\overline{F} = \overline{L}$.

Question 5. Un langage L est *clos par sur-mots* si, pour tout $u \in L$ et $v \in \Sigma^*$ tel que $u \preccurlyeq v$, on a $v \in L$. Déduire de la question précédente que tout langage clos par sur-mots est régulier.

A7 – Langages continuables et mots primitifs

On fixe un alphabet fini Σ et on suppose $|\Sigma| > 1$. Dans ce sujet, on considérera des automates sur l'alphabet Σ qui seront toujours supposés déterministes complets.

Un mot non-vide $w \in \Sigma^*$ est dit *primitif* s'il n'existe pas de mot $u \in \Sigma^*$ et d'entier $p > 1$ tel que $w = u^p$.

Question 0. Le mot *abaaabaa* est-il primitif ? Le mot *ababbaabbbababbab* (de longueur 17) est-il primitif ?

Question 1. Proposer un algorithme naïf qui, étant donné un mot, détermine s'il est primitif, et discuter de sa complexité en temps et en espace.

Question 2. Donner un exemple d'un langage régulier infini qui ne contienne aucun mot primitif.

Question 3. Donner un exemple d'un langage régulier infini qui ne contient que des mots primitifs.

Question 4. Un langage régulier L est dit *continuable* s'il a la propriété suivante : pour tout $u \in \Sigma^*$, il existe $v \in \Sigma^*$ tel que $uv \in L$. Donner un exemple de langage régulier infini non continuable. Existe-t-il des langages réguliers continuables dont le complémentaire soit infini ?

Question 5. Étant donné un automate A, proposer un algorithme pour déterminer si le langage $L(A)$ qu'il reconnaît est continuable. Justifier sa correction et discuter de sa complexité en temps et en espace.

Question 6.

(a) Montrer que tout langage régulier continuable contient une infinité de mots primitifs.

(b) Étant donné un langage régulier continuable L, donner une borne supérieure sur la taille du plus petit mot primitif de L.

(c) La réciproque de la question (a) est-elle vraie ?

A8 – Graphes avec un nombre prescrit de chemins

Un *graphe orienté* est une paire $G = (V, E)$ où $E \subseteq V \times V$ est l'ensemble des arêtes. Un *chemin* de $u \in V$ à $v \in V$ est une séquence $u_1, \ldots, u_n$ avec $n \geq 1$ telle que $u_1 = u$, $u_n = v$, et pour tout $1 \leq i < n$, le couple (u_i, u_{i+1}) est dans E. Un *graphe pointé* est un triplet (G, s, t) où s et t sont deux sommets de G. On dit qu'un graphe pointé *réalise* un entier $n \in \mathbb{N}$ s'il existe exactement n chemins de s à t dans G.

L'objet de ce sujet est de construire des petits graphes pointés réalisant n'importe quel entier.

Question 0. Construire un graphe pointé qui réalise 3.

Question 1. Pour tout $n \in \mathbb{N}$, proposer une construction naïve d'un graphe pointé qui réalise n et expliciter son nombre de sommets.

Question 2. Quelles hypothèses simplificatrices peut-on faire sur un graphe pointé qui réalise un entier fini et non-nul ?

Question 3. Pour tout $k \in \mathbb{N}$, construire un graphe pointé à $k + 2$ sommets qui réalise 2^k.

Question 4. Pour tout $n > 0$, construire un graphe pointé à $\lceil \log_2 n \rceil + 2$ sommets qui réalise n.

Question 5. Cette construction est-elle optimale ?

A9 – Notions d'acyclicité pour les hypergraphes

Dans ce sujet, on représentera des *graphes* non-orientés comme un ensemble G d'éléments appelés *arêtes* qui sont soit des paires, soit des singletons, et on impose que si $\{x, y\}$ est une paire de G alors G contient aussi les singletons $\{x\}$ et $\{y\}$. Intuitivement, une paire $\{x, y\}$ représente une arête non-orientée entre x et y au sens usuel, et chaque singleton représente un sommet.

Un *hypergraphe* H est un ensemble d'ensembles non-vides appelés *hyperarêtes*. L'*ensemble de sommets* de H est $\mathrm{V}(H) := \bigcup_{E \in H} E$. Étant donné un hypergraphe H et un sous-ensemble $X \subseteq \mathrm{V}(H)$ de ses sommets, le *sous-hypergraphe induit par* X est l'hypergraphe $H[X] := \{E \cap X \mid E \in H\}$.

Question 0. On considère le graphe G_0 (avec 11 arêtes dont 6 singletons) et l'hypergraphe H_0 (avec 3 hyperarêtes dont 1 singleton) suivants. Décrire $G_0[\{2, 3, 4, 5\}]$ et $H_0[\{1, 5\}]$.

Question 1. Pour tout hypergraphe H, on note $\mathrm{M}(H)$ le sous-ensemble de H obtenu en retirant les arêtes qui ne sont pas maximales au sens de l'inclusion ; par exemple, $\mathrm{M}(H_0)$ est identique à H_0 sauf que l'on retire l'hyperarête $\{5\}$.

Un *cycle induit* dans un hypergraphe H est un n-uplet $(x_1, \ldots, x_n)$ d'éléments distincts de $\mathrm{V}(H)$ avec $n \geq 3$ tel que, pour $X := \{x_1, \ldots, x_n\}$, l'hypergraphe $\mathrm{M}(H[X])$ est exactement $\{\{x_i, x_{i+1}\} \mid 1 \leq i < n\} \cup \{\{x_n, x_1\}\}$.

G_0 contient-il un cycle induit ? H_0 contient-il un cycle induit ?

Question 2. Montrer qu'un graphe a un cycle induit si et seulement s'il a un cycle au sens usuel, c'est-à-dire une séquence de sommets $(v_1, \ldots, v_n)$ pour $n \geq 3$ tels que $\{\{v_i, v_{i+1}\} \mid 1 \leq i < n\}$ et $\{v_n, v_1\}$ sont des arêtes du graphe qui ne sont pas des singletons.

Question 3. Donner un exemple d'un hypergraphe H avec une hyperarête $E \in H$ tels que H ne contienne pas de cycle induit mais $H \setminus \{E\}$ en contienne un. Commenter.

Question 4. Le *graphe d'incidence* $\mathrm{I}(H)$ d'un hypergraphe H est le graphe qui a comme sommets les sommets de $\mathrm{V}(H)$ et les arêtes de H, et qui a comme arêtes non-singletons l'ensemble suivant : $\bigcup_{E \in H} \{\{x, E\} \mid x \in E\}$. On dit que H est *Berge-cyclique* si le graphe $\mathrm{I}(H)$ est cyclique (au sens usuel).

Montrer qu'un graphe est Berge-cyclique si et seulement s'il est cyclique au sens usuel.

Est-il possible qu'un hypergraphe ait un cycle induit mais ne soit pas Berge-cyclique ? Est-il possible qu'un hypergraphe soit Berge-cyclique mais n'ait pas de cycle induit ?

Question 5. Étant donné un hypergraphe H, on dit que $x \in \mathrm{V}(H)$ est une *feuille* si, en notant H_x l'hypergraphe $\{E \in H \mid x \in E\}$, on a $|\mathrm{M}(H_x)| = 1$. On note $H[-x] := H[\mathrm{V}(H) \setminus \{x\}]$.

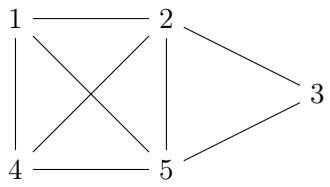
Montrer que, pour tout hypergraphe H et feuille x de H, l'hypergraphe H a un cycle induit si et seulement si $H[-x]$ en a un.

Question 6. Montrer qu'il existe un hypergraphe H et une feuille x de H de sorte que H soit Berge-cyclique mais $H[-x]$ ne le soit pas. Commenter.

A10 – Calcul de triangles

On considère un graphe non-orienté $G = (V, E)$ où $V = \{1, \ldots, n\}$ est l'ensemble des sommets et E est un ensemble d'arêtes qui sont des paires de sommets de V. Un *triangle* dans G est un sous-ensemble $\{x, y, z\} \in V$ tel que $\{x, y\}$, $\{y, z\}$, et $\{x, z\}$ appartiennent à E. Dans ce problème, on veut concevoir des algorithmes qui prennent en entrée un graphe et calculent (sans doublons) l'ensemble des triangles du graphe.

Question 0. Déterminer les triangles du graphe suivant :



Question 1. Étant donné un graphe G représenté comme une matrice d'adjacence, proposer un algorithme naïf en $O(|V|^3)$ pour déterminer l'ensemble des triangles de G.

Dans la suite du sujet, on supposera toujours que le graphe d'entrée est fourni sous forme de listes d'adjacence, et on supposera toujours que chacune de ces listes est triée.

Question 2. Étant donné un graphe G, proposer un algorithme en $O(|E| \times |V|)$ pour déterminer l'ensemble des triangles de G.

Question 3. Si l'on compare l'algorithme de la question 1 et celui de la question 2, lequel parmi ces algorithmes a la meilleure complexité ? Le choix de la représentation du graphe d'entrée était-il important ?

Question 4. Étant donné un graphe G, si l'on note Δ le degré maximal d'un sommet de G, proposer un algorithme en temps $O(|E| \times \Delta)$ pour déterminer l'ensemble des triangles de G. Commenter la performance de cet algorithme.

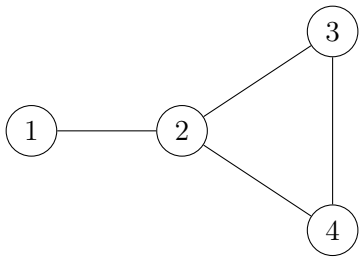
Question 5. Étant donné un graphe G, proposer un algorithme astucieux en $O\big(|E|^{3/2} + |V|\big)$ pour déterminer l'ensemble des triangles de G.

A11 – Dégénérescence de graphes

On considère un graphe non-orienté $G = (V, E)$ où V est l'ensemble des sommets et E est un ensemble d'arêtes qui sont des paires de sommets de V.

Un *sous-graphe* H de G est un graphe (V', E') où $V' \subseteq V$ et $E' \subseteq E$. Pour $k \in \mathbb{N}$, on dit qu'un graphe non-vide G est *k-dégénéré* si tout sous-graphe non-vide de G contient un sommet de degré au plus k dans H. La *dégénérescence* de G est le plus petit $k \in \mathbb{N}$ tel que G est k-dégénéré.

Question 0. Montrer que le graphe G_0 suivant est 2-dégénéré. Quelle est sa dégénérescence ?



Question 1. Pour tout graphe G, on note Δ_G le degré maximal de G. Montrer que tout graphe G est Δ_G-dégénéré. Est il vrai que tout graphe G est de dégénérescence Δ_G ?

Question 2. Un sous-graphe $H = (V', E')$ de G est appelé *sous-graphe induit* si E' est la restriction de E à V', i.e., $E' = \{\{u, v\} \in E \mid u, v \in V'\}$. Montrer que, dans la définition d'un graphe k-dégénéré et de la dégénérescence, il suffit de considérer les sous-graphes induits.

Question 3. Donner le pseudo-code d'un algorithme naïf pour calculer la dégénérescence d'un graphe, et en préciser la complexité en temps.

Question 4. Caractériser les graphes de dégénérescence 1.

Question 5. Un graphe G est dit *régulier* si tous ses sommets ont le même degré. Montrer que tout graphe régulier G est de dégénérescence Δ_G. La réciproque est-elle vraie ?

Question 6. Pour $k \in \mathbb{N}$, un *k-arrangement* d'un graphe G est un ordre total sur ses sommets $v_1 < \cdots < v_n$ tel que, pour tout $1 \leq i \leq n$, on ait $|\{v_j \in V \mid j < i \wedge \{v_i, v_j\} \in E\}| \leq k$. Montrer qu'un graphe a un k-arrangement si et seulement s'il est k-dégénéré.

Question 7. Proposer un algorithme en temps linéaire pour calculer la dégénérescence.

A12 – Plus grands facteurs

On fixe l'alphabet $\Sigma := \{a, b\}$. Pour un langage L sur Σ, étant donné un mot $w \in \Sigma^*$, on veut calculer la longueur du plus grand facteur de w dans le langage L, notée $\mathrm{lpgf}(L, w)$; on convient que cette longueur est de -1 si aucun facteur de w n'est dans L. On a donc $-1 \leq \mathrm{lpgf}(L, w) \leq |w|$ où $|w|$ dénote la longueur du mot w.

Question 0. On considère le langage L_0 défini par l'expression rationnelle $(ab)^*$, et le mot $w_0 := baababababa$. Calculer $\mathrm{lpgf}(L_0, w_0)$.

Question 1. Étant donné un langage L, représenté comme un automate fini déterministe complet, et un mot $w \in \Sigma^*$, proposer un algorithme naïf pour calculer $\mathrm{lpgf}(L, w)$. Donner un pseudo-code pour cet algorithme, et déterminer sa complexité en temps et en espace.

Question 2. Proposer un algorithme plus efficace pour ce problème qui s'exécute en temps et en espace $O(|Q| \times |w|)$. Est-il important que l'automate soit déterministe ?

Question 3. On dit que $w' \in \Sigma^*$ est un *sous-mot* de w s'il existe une fonction strictement croissante ϕ de $\{1, \ldots, |w'|\}$ dans $\{1, \ldots, |w|\}$ telle que $w'_i = w_{\phi(i)}$ pour tout $1 \leq i \leq |w'|$, où w'_i dénote la i-ème lettre de w' et de même pour w.

Étant donné un langage L représenté comme un automate et un mot w, comment calculer la longueur du plus grand *sous-mot* de w dans le langage L ?

Question 4. On pose le langage $L_4 := \{a^n b^n \mid n \in \mathbb{N}\}$. Proposer un algorithme efficace qui calcule $\mathrm{lpgf}(L_4, w)$ pour un mot d'entrée w.

Question 5. On pose le langage $L_5 := \{uu \mid u \in \Sigma^*\}$. Proposer un algorithme qui calcule $\mathrm{lpgf}(L_5, w)$ pour un mot d'entrée w en temps $O(|w|^2)$.

Question 6. On pose le langage L_6 des mots bien parenthésés sur Σ défini inductivement comme suit : le mot vide ϵ est bien parenthésé, la concaténation de deux mots bien parenthésés est bien parenthésée, et si $w \in \Sigma^*$ est bien parenthésé alors awb l'est aussi. Proposer un algorithme efficace qui calcule $\mathrm{lpgf}(L_6, w)$ pour un mot d'entrée w.

A13 – Entrelacements et langages réguliers

On fixe un alphabet Σ. Pour tout mot $z \in \Sigma^*$, on note $|z|$ la longueur de z, et on note z_i la i-ème lettre de z pour $1 \leq i \leq |z|$. Un *entrelacement* de deux mots $u, v \in \Sigma^*$ est un mot $w \in \Sigma^*$ tel qu'il existe deux fonctions $f_u : \{1, \ldots, |u|\} \to \{1, \ldots, |w|\}$ et $f_v : \{1, \ldots, |v|\} \to \{1, \ldots, |w|\}$ satisfaisant les propriétés suivantes :

— f_u et f_v sont strictement croissantes ;

— L'image de f_u et l'image de f_v sont des ensembles disjoints et leur union vaut $\{1, \ldots, |w|\}$;

— Pour tout $1 \leq i \leq u$, on a $u_i = w_{f_u(i)}$;

— Pour tout $1 \leq j \leq v$, on a $v_j = w_{f_v(j)}$.

Pour deux mots $u, v \in \Sigma^*$, on note $u \sqcup\!\sqcup v$ le langage des mots qui sont un entrelacement de u et de v.

Question 0. Le mot *abacab* est-il un entrelacement de *baa* et *acb* ? Le mot *bacb* est-il un entrelacement de *ab* et de *cb* ?

Question 1. Donner le pseudo-code d'un algorithme qui, étant donné $u, v, w \in \Sigma^*$, détermine si $w \in u \sqcup\!\sqcup v$. Analyser sa complexité en temps et en espace.

Question 2. Proposer un algorithme qui, étant donné $u, v \in \Sigma^*$ et un automate fini A qui reconnaît un langage L, détermine si $(u \sqcup\!\sqcup v) \cap L$ est non-vide. En donner un pseudo-code, et analyser sa complexité en temps et en espace.

Question 3. Pour deux langages $L, L' \subseteq \Sigma^*$, on note $L \sqcup\!\sqcup L' := \bigcup_{(u,v) \in L \times L'} u \sqcup\!\sqcup v$.
Montrer que si L_1 et L_2 sont deux langages réguliers alors L l'est aussi.

Question 4. On pose $(\Sigma^*)_2 := \bigcup_{i \in \mathbb{N}} \Sigma^i \times \Sigma^i$, où Σ^i dénote les mots de longueur i sur l'alphabet Σ. Étant donné un langage L, on définit $\psi(L) := \{(u, v) \in (\Sigma^*)_2 \mid (u \sqcup\!\sqcup v) \cap L \neq \emptyset\}$. Décrire $\psi(a^*b^*)$ en français et le caractériser.

Question 5. On considère l'alphabet $\Sigma_2 := \Sigma \times \Sigma$. On définit une fonction $\zeta : (\Sigma^*)_2 \to (\Sigma_2)^*$ inductivement par $\zeta((\epsilon, \epsilon)) := (\epsilon, \epsilon)$ et, pour tout $(x, y) \in \Sigma_2$ et $(u, v) \in \Sigma^*$, on a $\zeta((xu, yv)) := (x, y)\zeta(u, v)$. Décrire le langage $\zeta(\psi(a^*b^*))$.

Question 6. Dans cette question, on fixe $\Sigma_2 := \{a, b\}$, et on considère le langage $L_6 := \{a^n b^n \mid n \in \mathbb{N}\}$. Soit A un automate fini déterministe complet sur Σ_2, soit Q son ensemble d'états, et soit $\delta^* : \Sigma_2^* \to Q$ la fonction telle que, pour tout $w \in \Sigma_2^*$, l'état $\delta^*(w)$ soit l'état auquel on aboutit dans A après avoir lu w. Montrer que, s'il existe $i \neq j$ tels que $\delta^*(a^i) = \delta^*(a^j)$, alors le langage reconnu par A est différent de L_6. Conclure que L_6 n'est pas régulier.

Question 7. Montrer qu'il existe un langage régulier L_7 tel que $\zeta(\psi(L_7))$ ne soit pas régulier.

J1 – Arbre de suffixes

On fixe Σ un alphabet fini et $\$ \notin \Sigma$ un caractère spécial. Si u et u' sont deux mots, on dit que u est un *facteur* de u' s'il existe deux mots v et v' tels que $u' = vuv'$. Si v est le mot vide ε, on dit que u est un *préfixe* de u', et si $v' = \varepsilon$, on dit que u est un *suffixe* de u'. On note $\mathrm{Pref}(u')$ l'ensemble des préfixes de u', et $\mathrm{Suff}(u')$ l'ensemble des suffixes de u'.

Soit $D \subseteq \Sigma^*\$$ un ensemble fini de mots sur Σ terminés par \$, c'est-à-dire que les mots de D sont de la forme $u\$$ où $u \in \Sigma^*$ est un mot sur Σ. On note $\mathrm{Pref}(D) := \bigcup_{u \in D} \mathrm{Pref}(u)$, et $\mathrm{Suff}(D) := \bigcup_{u \in D} \mathrm{Suff}(u)$. L'*arbre lexicographique de* D est un arbre aux arêtes étiquetées par des lettres de $\Sigma \cup \{\$\}$, muni d'une bijection ϕ de l'ensemble de ses nœuds vers $\mathrm{Pref}(D)$, qui satisfait les conditions suivantes :

— la racine de l'arbre est envoyée vers le mot vide par ϕ ;

— pour $u \in \Sigma^*$ et $\alpha \in \Sigma \cup \{\$\}$, si on a $\phi(x) = u\alpha$ pour un nœud x, alors x n'est pas la racine, on a $\phi(x') = u$ pour le parent x' de x, et l'arête de x' à x est étiquetée par le caractère α.

On note en particulier que, pour chaque nœud x de l'arbre, le préfixe $\phi(x)$ est égal à la concaténation des étiquettes des arêtes le long du chemin qui va de la racine à x.

Question 1

(a) Donner l'arbre lexicographique de $D = \{aab\$; abb\$; ac\$; acbaa\$; b\$\}$.

(b) Donner un algorithme permettant de construire l'arbre lexicographique d'un ensemble de mots $D \subseteq \Sigma^*\$$. Quelle est sa complexité en temps ?

(c) Donner un algorithme qui, étant donné l'arbre lexicographique d'un ensemble de mots $D \subseteq \Sigma^*\$$ et un mot $u \in \Sigma^*$, détermine si $u\$ \in D$ et si u est un préfixe d'un mot de D. Quelle est sa complexité en temps ?

Question 2 Soit $t \in \Sigma^*\$$ un mot terminé par \$. On appelle *arbre inefficace des suffixes* de t l'arbre lexicographique de $\mathrm{Suff}(t) \setminus \{\epsilon\}$.

(a) Donner l'arbre inefficace des suffixes du mot $abbabab\$$.

(b) Asymptotiquement, quelle est la taille de l'arbre inefficace des suffixes de t, dans le pire cas ? Donner un exemple de famille de mots pour lequel le pire cas est atteint.

(c) En gardant une structure arborescente, proposer une amélioration de l'efficacité de la représentation en mémoire de cet arbre. Montrer que la structure de données ainsi obtenue nécessite un espace $O(|t|)$.

La structure de données de la question (c) est appelée *l'arbre des suffixes* de t. Dans les questions suivantes, **on admettra qu'il existe un algorithme pour calculer l'arbre des suffixes de t en temps $O(|t|)$.**

Question 3

(a) Soit $M \subseteq \Sigma^*$ un ensemble fini de mots. Donner un algorithme efficace pour déterminer pour chaque mot de $m \in M$ s'il est un facteur de t. Quelle est sa complexité en temps ?

(b) Donner un algorithme efficace pour déterminer un plus long facteur commun de deux mots t et t'. Quelle est sa complexité en temps et en espace ?

J2 – Domaines abstraits linéaires

Le but de ce problème est d'étudier la structure de données d'*octogones*, qui permet d'approximer des parties de $\mathbb{R}^n$, et d'effectuer certaines opérations sur celles-ci.

Pour toutes les questions de ce problème, on suppose que l'on peut effectuer constant les opérations arithmétiques usuelles sur les nombres réels en temps constant. (C'est bien sûr faux dans la réalité, mais les nombres à virgule flottante donnent une bonne approximation.)

Soit $n \in \mathbb{N}^*$. Pour $A \in M_n(\mathbb{R} \cup \{+\infty\})$ une matrice carrée de taille n dont chaque case contient une valeur de $\mathbb{R} \cup \{+\infty\}$, on définit $\gamma_Z(A) \subseteq \mathbb{R}^n$ par :

$$\gamma_Z(A) := \{x \in \mathbb{R}^n \mid \forall\, 1 \leq i, j \leq n,\ x_i - x_j \leq A_{ij}\}$$

Intuitivement, on pensera à A comme un ensemble de contraintes d'inégalité de la forme $x_i - x_j \leq K$ sur $\mathbb{R}^n$.

Question 0. Représenter graphiquement $\gamma_Z\left(\begin{bmatrix} 0 & 2 \\ -1 & 0 \end{bmatrix}\right)$.

Si $U \subseteq \mathbb{R}^n$ est une partie non vide de $\mathbb{R}^n$, on définit $\alpha_Z(U) \in M_n(\mathbb{R} \cup \{+\infty\})$ comme suit pour tous $1 \leq i, j \leq n$:

$$\alpha_Z(U)_{ij} := \sup_{x \in U} x_i - x_j$$

Les contraintes d'inégalité imposées par A sont donc les plus fortes qui sont respectées par U.

On dit que A est *close* si $\gamma_Z(A) \neq \emptyset$ et $\alpha_Z(\gamma_Z(A)) = A$.

On dit qu'une matrice $A \in M_n(\mathbb{R} \cup \{+\infty\})$ vérifie *l'inégalité triangulaire* lorsque :

$$\forall ijk,\ A_{ij} \leq A_{ik} + A_{kj}$$

Question 1.

(a) Soit U une partie non vide de $\mathbb{R}^n$. Montrer que $\alpha_Z(U)$ vérifie l'inégalité triangulaire et est à diagonale nulle. En déduire qu'une matrice close vérifie l'inégalité triangulaire et est à diagonale nulle.

(b) Soit $A \in M_n(\mathbb{R} \cup \{+\infty\})$. On suppose que $\gamma_Z(A) \neq \emptyset$. Prouver que pour tout i et j, on a $\alpha_Z(\gamma_Z(A))_{ij} \leq A_{ij}$.

(c) Prouver qu'une matrice A est close si et seulement si elle vérifie l'inégalité triangulaire et est à diagonale nulle. On se limitera au cas où A est à coefficients finis, et on admettra le résultat général pour les questions suivantes.

(d) Pour $U \neq \emptyset$, prouver que $\alpha_Z(U)$ est close.

(e) Donner un algorithme qui, étant donné A, détecte si $\gamma_Z(A) \neq \emptyset$, et qui, le cas échéant, calcule $\alpha_Z(\gamma_Z(A))$ en temps $O(n^3)$.

J3 – Arbres de Braun

On prend la définition suivante pour la notion d'arbre binaire : il s'agit d'une structure de données finie, consistant soit en un arbre vide $\langle\rangle$ (*i.e.,* une feuille), soit en un nœud. Un nœud $\langle g, d, c\rangle$ contient un sous-arbre binaire gauche g, un sous-arbre binaire droit d et une *charge utile* c. On ne décrit pas précisément ce que contient la charge utile : cela dépend de l'utilisation qui est faite de l'arbre. La *taille* d'un arbre binaire a, notée $T(a)$, est le nombre de nœuds qu'il contient. La *hauteur* d'un arbre binaire est 0 si l'arbre est vide, et sinon c'est le nombre maximal de nœuds qu'il faut traverser pour atteindre une feuille depuis la racine (feuille exclue et racine incluse).

Un *arbre de Braun* est un arbre binaire dont tous les nœuds $\langle g, d, c\rangle$ vérifient :

$$0 \;\leq\; T(g) \;-\; T(d) \;\leq\; 1$$

Question 1

(a) Montrer que, si on ignore les charges utiles, alors pour chaque $n \in \mathbb{N}$ il existe un unique arbre de Braun de taille n. Dessiner les 6 premiers arbres de Braun non vides.

(b) Proposer un algorithme naïf pour calculer la taille d'un arbre de Braun.

(c) Donner le comportement asymptotique de la hauteur d'un arbre de Braun en fonction de sa taille.

Question 2 Expliquer comment on peut utiliser la structure d'arbre de Braun pour implémenter une file de priorité persistante. On demande de donner un algorithme pour les opérations d'insertion (en $O(\log n)$), de consultation du minimum (en $O(1)$) et de suppression du minimum (en $O(\log n)$).

Question 3 Donner un algorithme pour calculer la taille d'un arbre de Braun ayant une complexité sous-linéaire. Quelle est sa complexité ?

J4 – Problèmes d'ordonnancement

On s'intéresse à des problèmes d'*ordonnancement* : on a n *tâches* à réaliser (des programmes à exécuter sur un ordinateur, par exemple), et on souhaite déterminer quand les réaliser en respectant certaines contraintes pour optimiser un objectif. La définition exacte des contraintes et de l'objectif varieront selon la question.

Chaque tâche aura toujours une durée d'exécution $p_i \in \mathbb{N}^*$, et on supposera qu'une tâche ne peut pas être interrompue une fois commencée. On cherche donc à associer à chaque tâche i un instant de début $t_i \in \mathbb{N}$, en s'assurant qu'une seule tâche est exécutée à la fois à chaque instant, c'est-à-dire que :

$$\forall 1 \leq i, j \leq n,\ t_i - t_j \geq p_j \text{ ou } t_j - t_i \geq p_i$$

Question 1 Dans cette question, on ne considère aucune contrainte supplémentaire.

(a) Proposer un algorithme pour calculer un ordonnancement qui minimise la date de fin de la dernière tâche. Quelle est sa complexité en temps ?

(b) Proposer un algorithme pour calculer un ordonnancement qui minimise la moyenne des dates de fin. Quelle est sa complexité en temps ?

(c) À chaque tâche i on associe un poids $w_i \in \mathbb{N}^*$. Proposer un algorithme pour calculer un ordonnancement qui minimise la moyenne des dates de fin, pondérée par les poids. Quelle est sa complexité en temps ?

Question 2 On suppose, **dans cette question uniquement**, qu'en plus d'une durée d'exécution p_i, chaque tâche a une *date de disponibilité* r_i. Une tâche ne peut jamais commencer avant cette date. Un ordonnancement valide $(t_i)_{1 \leq i \leq n}$ vérifiera donc toujours :

$$\forall 1 \leq i \leq n,\ r_i \leq t_i$$

(a) Adapter naïvement l'algorithme proposé en 1a. L'algorithme obtenu est-il correct ? S'il l'est, quelle est sa complexité en temps ?

(b) Adapter naïvement l'algorithme proposé en 1b. L'algorithme obtenu est-il correct ? S'il l'est, quelle est sa complexité en temps ?

J5 – Constitution d'échantillons aléatoires

On se propose d'étudier plusieurs algorithmes permettant de tirer aléatoirement un élément dans un ensemble, selon une distribution donnée. Comme source de hasard, on suppose que l'on dispose de deux primitives dans notre langage de programmation :

— `unif(x)` prend un entier $x \in \mathbb{N}^*$, et renvoie un entier naturel tiré selon la distribution uniforme sur $\{0, \dots, x-1\}$;

— `bern(p)` prend un réel $p \in [0, 1]$, et renvoie un booléen tiré selon une distribution de Bernoulli de paramètre p.

On suppose que ces primitives s'exécutent en temps $O(1)$. Par ailleurs, on suppose que les opérations arithmétiques usuelles sur les réels peuvent être calculées de manière exacte en temps constant. (Ces hypothèses sont fausses en pratique, mais de bonnes approximations sont possibles.)

Question 1 Proposer un algorithme permettant de mélanger uniformément un tableau. Plus précisément, l'algorithme doit choisir uniformément aléatoirement une permutation des indices du tableau, et l'appliquer au tableau. Quelle est sa complexité en temps ?

Question 2

(a) Proposer un algorithme permettant de choisir uniformément k éléments distincts dans un ensemble fini E. Quelle est sa complexité en temps et en mémoire ?

On se place dans le cas où l'ensemble E est trop gros pour tenir dans la mémoire : le seul moyen de lire ses éléments est d'appeler une fonction `suivant_E()`, qui renverra successivement tous ses éléments. Par ailleurs, on ne connaît pas à l'avance le nombre d'éléments de E : une autre fonction `terminé_E()` renvoie un booléen indiquant si tous les éléments ont été renvoyés par `suivant_E()`. (On garantit cependant bien entendu que E contient au moins k éléments.)

(b) Proposer un algorithme permettant de choisir uniformément k éléments distincts dans un ensemble fini E. Quelle est sa complexité en temps et en mémoire ?

Question 3. On s'intéresse maintenant à tirer aléatoirement dans $\{0, \dots, n-1\}$, selon une distribution fixée. Plus précisément, on prend en entrée un tableau de poids entiers W de somme S et on veut que la probabilité de tirer l'entier i soit $\frac{W[i]}{S}$.

(a) Proposer un algorithme pour résoudre ce problème, avec une complexité $O(n)$ en temps pour chaque tirage.

On se place maintenant dans le cas où de nombreux tirages aléatoires seront effectués successivement sur cette même distribution. On veut donc que le tirage aléatoire soit rapide, quitte à passer un peu de temps à faire des *pré-calculs* sur P : ils ne seront effectués qu'une fois, et pourront être réutilisés pour faire des tirages plus rapides.

(b) Proposer un algorithme pour résoudre ce problème, qui nécessite un temps de pré-calcul $O(n)$, et qui a une complexité en temps de $O(\log n)$ pour chaque tirage. Donner un pseudo-code **détaillé** de cet algorithme.

J6 – Arithmétique de Presburger

Soit $\mathcal{X} = \{x_1, \ldots, x_n\}$ un ensemble fini de variables. Une *valuation entière* (respectivement *valuation booléenne*) sur $\mathcal{X}$ est une fonction de $\mathcal{X}$ dans $\mathbb{N}$ (respectivement dans $\{0,1\}$). On note $\mathbb{N}^{\mathcal{X}}$ (respectivement $2^{\mathcal{X}}$) l'ensemble des valuations entières (respectivement booléennes).

On considère l'alphabet $\Sigma = 2^{\mathcal{X}}$, et on associe à chaque mot $a = a_0 \cdots a_{m-1}$ de Σ^* la valuation entière ϕ_a définie par :

$$\phi_a(x_i) := \sum_{j=0}^{m} 2^j a_j(x_i)$$

On dit alors qu'un ensemble E de valuations entières est *régulier* si le langage L_E défini par :

$$L_E := \{a \in \Sigma^* \mid \phi_a \in E\}$$

est régulier (c'est à dire s'il correspond exactement aux mots reconnus par un automate fini).

Question 1

(a) Soit $x_i \in \mathcal{X}$ une variable et $v \in \mathbb{N}$ un entier. Montrer que l'ensemble de valuations entières suivant est régulier :

$$\{\phi \in \mathbb{N}^{\mathcal{X}} \mid \phi(x_i) = v\}$$

(b) Montrer que si E et F sont deux ensembles réguliers de valuations entières, alors $E \cup F$, $E \cap F$ et $\mathbb{N}^{\mathcal{X}} \setminus E$ sont réguliers.

Question 2 Soit $x_i \in \mathcal{X}$ une variable. Si $\phi \in \mathbb{N}^{\mathcal{X}\setminus\{x_i\}}$, et $u \in \mathbb{N}$, on note $\phi[x_i \leftarrow u]$ la valuation dans $\mathbb{N}^{\mathcal{X}}$ définie par :

$$\phi[x_i \leftarrow u](x_j) := \begin{cases} u & \text{si } x_i = x_j \\ \phi(x_i) & \text{sinon} \end{cases}$$

Soit $E \subseteq \mathbb{N}^{\mathcal{X}}$ un ensemble régulier de valuations entières. Montrer que les ensembles de valuations entières suivants sont réguliers :

$$E_{\exists x_i} := \{\phi \in \mathbb{N}^{\mathcal{X}\setminus\{x_i\}} \mid \exists u \in \mathbb{N},\ \phi[x_i \leftarrow u] \in E\}$$
$$E_{\forall x_i} := \{\phi \in \mathbb{N}^{\mathcal{X}\setminus\{x_i\}} \mid \forall u \in \mathbb{N},\ \phi[x_i \leftarrow u] \in E\}$$

Question 3 On dit qu'une fonction $f : \mathbb{N}^{\mathcal{X}} \to \mathbb{N}$ est *affine* s'il existe des entiers naturels $u_0, u_1, \ldots, u_n$ tels que :

$$\forall \phi \in \mathbb{N}^{\mathcal{X}},\ f(\phi) = u_0 + u_1\phi(x_1) + \cdots + u_n\phi(x_n)$$

Soient f et g deux fonctions affines de $\mathbb{N}^{\mathcal{X}}$ dans $\mathbb{N}$.

(a) Montrer que l'ensemble $\{\phi \in \mathbb{N}^{\mathcal{X}} \mid f(\phi) = g(\phi)\}$ est régulier.

(b) Montrer que l'ensemble $\{\phi \in \mathbb{N}^{\mathcal{X}} \mid f(\phi) \leq g(\phi)\}$ est régulier.

Question 4 Déduire des questions précédentes que l'ensemble suivant est régulier :

$$\{\phi \in \mathcal{X} \mid \phi(x_1) = \max(\phi(x_2), 3\phi(x_3)) \wedge (\exists u \in \mathbb{N},\ (u \leq 18 \wedge \phi(x_2) \bmod 42 = u) \vee \phi(x_3) = \phi(x_1)+1)\}$$

J7 – Détection de cycle

Dans ce problème, on admettra qu'il est possible d'implémenter la structure de donnée de dictionnaire à clés entières de façon à ce que les accès en lecture et en écriture se fassent en temps constant, et en n'occupant qu'un espace mémoire linéaire en le nombre d'entrées dans le dictionnaire.

Soit $N > 2$ un entier, f une fonction de $[\![0, N-1]\!]$ dans lui-même, et $x_0 \in [\![0, N-1]\!]$. On définit la suite $(x_i)_{i\in\mathbb{N}}$ par :

$$x_{i+1} = f(x_i)$$

Question 0 Montrer qu'il existe i et j deux entiers naturels tels que $x_i = x_j$ et $i < j$.

On pose :

$$\mu = \min\{i \in \mathbb{N} \mid \exists j \in \mathbb{N},\, j > i \wedge x_i = x_j\}$$
$$\lambda = \min\{i \in \mathbb{N}^* \mid x_\mu = x_{\mu+i}\}$$

Le but de ce problème est de trouver des algorithmes *efficaces* pour calculer μ et λ, étant donnée une implémentation de f.

Question 1

(a) Donner un algorithme simple pour calculer λ et μ, qui est efficace en temps. Quel est le nombre (exact) d'appels à f effectués ? Quelle est la complexité en mémoire, en fonction de λ, μ et N ? Le nombre d'appels à f est-il optimal ?

(b) Donner un algorithme simple pour calculer λ et μ, qui est efficace en mémoire. Quelle est sa complexité en temps et en mémoire, en fonction de λ, μ et N ?

On cherche maintenant un algorithme efficace en temps *et* en mémoire.

Question 2

(a) Quels sont les éléments de l'ensemble $\{n \in \mathbb{N}^* \mid x_n = x_{2n}\}$, en fonction de λ et μ ?

(b) En déduire un algorithme pour calculer λ et μ en temps $O(\lambda + \mu)$ et en espace mémoire $O(1)$.

(c) Combien d'appels à f sont effectués au total par cet algorithme ?

J8 – Sommes d'intervalles dans un tableau

Soit T un tableau d'entiers relatifs, de taille $N > 0$.

Question 1 Donner un algorithme pour calculer des indices i et j tels que $0 \leq i \leq j < N$ et tels que la somme $\sum_{k=i}^{j} T[k]$ soit maximale. Quelle est sa complexité en temps ?

Question 2 Si, pour une valeur de j donnée, on connaît la valeur de $\max_{i \in [\![0,j]\!]} \sum_{k=i}^{j} T[k]$, comment calculer efficacement $\max_{i \in [\![0,j+1]\!]} \sum_{k=i}^{j+1} T[k]$? En déduire un algorithme en temps linéaire pour répondre à la question 1.

Question 3 Soit $m \in [\![1, N]\!]$. Donner un algorithme pour calculer en temps linéaire un indice $i \in [\![0, N-m]\!]$ tel que la somme $\sum_{k=i}^{i+m-1} T[k]$ soit maximale.

Question 4 Donner un algorithme efficace pour calculer des indices i et j optimaux comme aux questions 1 et 2, mais avec la contrainte supplémentaire $j - i < m$. Quelle est sa complexité en temps et en espace ?

J9 – Tas et compagnie

Dans ce sujet, les éléments de tout tableau T de taille n sont numérotés de 0 à $n-1$.

Question 1.

(a) Qu'est-ce qu'un tas ? Comment peut-on utiliser un tableau pour implémenter un tas ?

(b) Proposer un algorithme pour transformer, en temps linéaire, un tableau quelconque d'entiers en un tas tel que décrit dans la question précédente, en conservant ses éléments.

Question 2. On dit qu'un tableau T d'entiers de taille $2n$ est un *tas d'intervalles* s'il vérifie les propriétés suivantes :

$$T\left[2\left\lfloor\tfrac{i-1}{2}\right\rfloor\right] \leq T[2i] \qquad i \in [\![1, n-1]\!] \tag{1}$$

$$T\left[2\left\lfloor\tfrac{i-1}{2}\right\rfloor+1\right] \geq T[2i+1] \qquad i \in [\![1, n-1]\!] \tag{2}$$

$$T[2i] \leq T[2i+1] \qquad i \in [\![0, n-1]\!] \tag{3}$$

(a) Dans un tas d'intervalles, où sont les entiers le plus petit et le plus grand ?

(b) Proposer un algorithme pour extraire simultanément le minimum et le maximum d'un tas d'intervalles. Quelle est sa complexité ?

(c) Prouver que, pour un tableau d'entiers T quelconque, si T vérifie (1) et (2), alors (3) est vraie pour tout $i \in [\![0, n-1]\!]$ si et seulement si elle est vraie pour tout $i \in \left[\!\left[\left\lfloor\tfrac{n}{2}\right\rfloor, n-1\right]\!\right]$.

(d) Proposer un algorithme pour insérer deux entiers dans un tas d'intervalles. Quelle est sa complexité ?

(e) Montrer qu'un tas d'intervalles permet d'implémenter une structure de données représentant un ensemble d'entiers distincts et répondant efficacement aux requêtes suivantes :

- lecture du plus petit entier ;
- lecture du plus grand entier ;
- extraction du plus petit entier ;
- extraction du plus grand entier ;
- insertion d'un entier.

Quelle est la complexité de chacune de ces opérations ?

J10 – Arithmétique d'Avizienis

Soient a et b deux entiers positifs vérifiant $\frac{b}{2} < a < b$. On introduit le système de numération d'Avizienis : celui-ci s'apparente au système de numération en base b, mais il utilise $2a+1$ chiffres qui vont de $-a$ à a. En pratique, on note $\bar{1}, \bar{2}, \ldots$ pour les chiffres négatifs.

Plus précisément, si $x_n, \ldots, x_0$ sont de tels chiffres (des éléments de $[\![-a, a]\!]$, donc), alors l'interprétation de $x_n \cdots x_0$ est :

$$(x_n \cdots x_0)_{\text{avi}} = \sum_{k=0}^{n} x_k b^k$$

Question 1.

(a) La représentation d'un entier dans le système de numération d'Avizienis est-elle unique ?

(b) Quels nombres peut-on représenter avec n chiffres ?

Question 2.

(a) Quelles sont les représentations de 0 ?

(b) Proposer un algorithme qui détermine le signe d'un nombre non nul représenté avec le système de numération d'Avizienis. Dans quel cas cet algorithme est-il particulièrement efficace ?

Un circuit logique combinatoire est un graphe orienté acyclique tel que :

— Les nœuds avec seulement des arêtes sortantes sont appelés entrées du circuit, on les note $e_1 \ldots e_n$.

— Les nœuds avec seulement des arêtes entrantes sont appelés des sorties du circuit, notés $s_1 \ldots s_m$. Ces nœuds de sortie ont obligatoirement exactement une arête entrante.

— Les nœuds internes (i.e., ayant des arêtes entrantes et sortantes) sont également appelées portes. On en utilise ici de quatre types, étiquetés respectivement NON, ET, OU ou XOU. Les nœuds NON ont une arête entrante exactement, tandis qu'un nœud ET, OU ou XOU a deux arêtes entrantes exactement.

Un calcul du circuit consiste à affecter des 0 ou 1 à chaque nœud d'entrée, puis à propager ces booléens le long des arêtes, en effectuant au passage les calculs indiqués par les nœuds internes. De la sorte, un circuit de n entrées et m sorties peut être associé à une fonction de $\{0,1\}^n$ dans $\{0,1\}^m$.

La profondeur d'un circuit est la longueur du plus long chemin entre une entrée et une sortie.

Question 3.

(a) Donner un circuit prenant 3 booléens (vus comme des éléments de $\{0,1\}$) en entrée et calculant leur somme représentée en base 2 sur 2 bits

(b) En déduire un circuit qui prend en entrée deux entiers positifs ou nuls représentés en base 2 sur n bits et qui calcule la somme représentée en base 2 sur $n+1$ bits. Quelle est sa profondeur, asymptotiquement ?

(c) Proposer un circuit permettant de comparer deux entiers positifs ou nuls représentés en base 2 sur n bits. Quelle est sa profondeur, asymptotiquement ?

Question 4. Montrer qu'il existe un circuit logique combinatoire prenant en entrée deux nombres représentés dans le système d'Avizienis et qui calcule une représentation de leur somme dans le système d'Avizienis. Le circuit aura une profondeur bornée (indépendamment de n), et on choisira une représentation appropriée pour les entrées et sorties sous forme de booléens.

J11 – Constitution d'échantillons aléatoires

On suppose que notre langage de programmation fournit une primitive `pièce`, qui renvoie un booléen tiré selon la distribution de Bernoulli de paramètre $\frac{1}{2}$.

On définit la *complexité moyenne en temps* d'un programme utilisant `pièce` comme étant l'espérance, sur tous les résultats des tirages de `pièce`, de la complexité en temps de l'exécution du programme au sens habituel.

Par ailleurs, on suppose que les opérations arithmétiques usuelles sur les réels peuvent être calculées de manière exacte en temps constant. (Ces hypothèses sont fausses en pratique, mais de bonnes approximations sont possibles.)

Question 1 Proposer un algorithme qui prend en paramètre un réel $p \in [0, 1]$, et qui renvoie un booléen tiré selon la distribution de Bernoulli de paramètre p. Quelle est sa complexité moyenne en temps ? Quelle est sa complexité en espace ?

Question 2 Soit L un langage régulier sur l'alphabet $\{a, b\}$ décrit par un automate déterministe $\mathcal{A}$ et soit $n \in \mathbb{N}$. On note L_n l'ensemble des mots de L de longueur n, et on pose $\gamma_n = \frac{|L_n|}{2^n}$. On suppose $\frac{1}{\gamma_n}$ borné par une constante. Donner un algorithme qui tire aléatoirement uniformément un mot dans L_n, avec une complexité moyenne en temps $O(n)$. Quelle est sa complexité en espace ?

Question 3

(a) Déduire de la question 2 un algorithme qui prend en paramètre un entier $x \in \mathbb{N}^*$ et qui renvoie un entier naturel tiré selon la distribution uniforme sur $\{0, \ldots, x-1\}$, avec une complexité moyenne en temps $O(\log x)$.

(b) En espérance et à une constante *additive* près, combien d'appels à `pièce` sont effectués par cet algorithme ?

(c) Proposer un algorithme qui effectue moins d'appels à `pièce` en moyenne.

J12 – Énumérations

On fixe $\mathcal{A}$ un automate déterministe sur un alphabet Σ, et un entier $m \in \mathbb{N}^*$.

Question 1

(a) Proposer un algorithme qui calcule la longueur minimale des mots acceptés par $\mathcal{A}$, si cette quantité est bien définie, et qui détecte si elle ne l'est pas. Quelle est sa complexité ?

(b) On suppose dans cette question que $\mathcal{A}$ n'a pas de cycle. Proposer un algorithme qui calcule la longueur maximale des mots acceptés par $\mathcal{A}$, si cette quantité est bien définie, et qui détecte si elle ne l'est pas. Quelle est sa complexité ?

(c) Que se passe-t-il lorsque $\mathcal{A}$ a un cycle ?

Question 2

(a) Proposer un algorithme permettant de calculer le nombre modulo m de mots de longueur n dans $L(\mathcal{A})$. Quelle est sa complexité en temps et en espace ?

(b) Le problème devient-il plus difficile si on demande de calculer le résultat exact (sans modulo) ?

(c) L'hypothèse que $\mathcal{A}$ est déterministe est-elle importante ?

J13 – Réseaux de tri

Dans ce sujet, on se propose d'étudier les *réseaux de tri*. Il s'agit d'une certaine catégorie d'algorithmes de tri, pour lesquels on fixe à l'avance une série de comparaisons et d'échanges constituant ledit réseau.

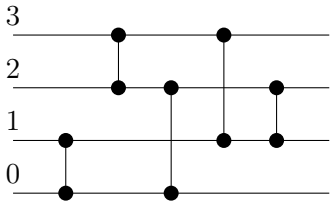
Plus précisément, dans ce sujet, un réseau de tri agit sur un tableau T de n entiers, et est décrit par une séquence de *modules de comparaison*. Un module de comparaison est décrit par la donnée d'une paire (i, j) de deux indices tels que $0 \leq i < j < n$. Un tel module de comparaison a pour effet d'échanger les contenus des cellules $T[i]$ et $T[j]$ si et seulement si $T[j] < T[i]$.

On représentera un réseau de tri de la façon suivante :

— Chaque cellule du tableau est représenté par une ligne horizontale allant de l'extrême gauche à l'extrême droite du schéma.

— Les modules de comparaison sont alors représentés de gauche à droite. Chacun d'entre eux est représenté par une ligne verticale reliant les lignes correspondant aux deux cellules concernées.

Par exemple, le réseau de tri $(0, 1)(2, 3)(0, 2)(1, 3)(1, 2)$ est représenté par le schéma suivant :



Question 1 Exécuter le réseau de tri donné en exemple sur le tableau $[2, 4, 5, 1]$. Ce réseau permet-il de trier n'importe quel tableau de taille 4 ?

Question 2 Donner un réseau de tri qui trie tout tableau d'entiers de taille n. On ne demande pas que ce réseau de tri soit optimal selon quelque critère que ce soit. Combien de modules de comparaison contient-il, asymptotiquement ?

Question 3 Montrer qu'un réseau de tri permet de trier tout tableau si et seulement si il permet de trier tout tableau de booléens.

Question 4 Dans cette question, on suppose que $n = 2^k$ est une puissance de 2 avec $k > 1$. Par ailleurs, on ne considère que les tableaux T tels que les sous-tableaux $T[0] \dots T[\frac{n}{2} - 1]$ et $T[\frac{n}{2}] \dots T[n-1]$ sont déjà triés. Construire un réseau de tri qui trie ces tableaux, en utilisant $O(n \log n)$ modules de comparaison.

Indication : on pourra utiliser une approche diviser-pour-régner, en commençant par trier les positions paires et les positions impaires indépendamment.

Question 5 En déduire un réseau de tri efficace pour trier tout tableau de taille quelconque. Asymptotiquement, combien de modules de comparaison contient-il ?

On définit la profondeur d'un réseau de tri par le nombre maximal de modules de comparaison qu'une valeur doit traverser avant d'arriver à la fin du réseau.

Question 6 Quelle est, asymptotiquement, la profondeur du réseau de tri de la question 5 ?