Rapport de l'épreuve orale d'informatique fondamentale

École normale supérieure
Concours MP et INFO 2019
Épreuve spécifique Ulm

Jury : Antoine Amarilli, Jacques-Henri Jourdan, Ludovic Patey

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 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 candidats en informatique pratique, même s'il est souvent demandé aux candidats de présenter certains points en pseudocode.

Cette année, comme les années précédentes, 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 74 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 20, avec une moyenne de 11.9 et un écart type de 3.4.

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 candidats de *prendre connaissance de ces programmes* et de s'assurer qu'ils maîtrisent effectivement les points qui y figurent. Les examinateurs posent parfois des questions de cours, et certains candidats ont avoué n'avoir aucune connaissance de certains points mentionnés dans le programme, ce qui est très pénalisant.

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 proposé. Dans l'ensemble, les candidats se sont bien appropriés 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 Comme les années précédentes, par souci de transparence, et pour permettre à tous les candidats de préparer cette épreuve de manière équitable, ce rapport de concours inclut en annexe l'intégralité des sujets posés aux candidats cette année [1]. Soulignons toutefois que certaines questions étaient suffisamment difficiles pour qu'il ne soit pas envisageable que les candidats puissent les traiter sans aide, même pour les meilleurs d'entre eux. De même, les sujets n'incluent pas ici certaines questions plus délicates qui étaient posées à l'oral une fois que les questions imprimées avaient été résolues.

Critères d'évaluation. Comme les années précédentes, les examinateurs ont évalué 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. 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 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 qui demandaient 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, il était parfois demandé aux candidats d'écrire le pseudocode d'algorithmes simples. 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 ; les candidats moins bons se perdent dans ces algorithmes pourtant classiques. De manière générale, il est conseillé aux candidats 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 terminer, quelles valeurs peut-il renvoyer, quelles valeurs sont possibles pour les paramètres, quel est le type d'objets manipulés, etc.

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 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 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.

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

Recommandations aux candidats. En termes de programme, il est recommandé aux candidats de s'assurer qu'ils maîtrisent les points suivants, sur lequel le jury a 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.
- 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.
- 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. Bien souvent, les candidats qui échafaudent des arguments trop savants ou trop compliqués se retrouvent en difficulté pour répondre à des questions trop simples.
- 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 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 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 candidats.

A1 – Mots sans carré et mots sans cube

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

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

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

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

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

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

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

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

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

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

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

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

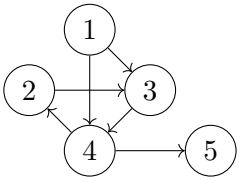
Existe-t-il des mots infinis sans cube sur un alphabet de taille 2 ? des mots infinis sans carré sur un alphabet de taille 3 ?

A2 – Familles closes de graphes

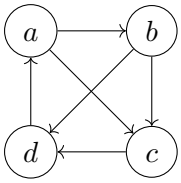
Un *graphe* $G = (V, E)$ est la donnée d'un ensemble fini non-vide de sommets V et d'un ensemble d'arêtes $E \subseteq V \times V$. On imposera toujours que $V \subseteq \mathbb{N}$, et on autorise les *boucles*, c'est-à-dire les arêtes de la forme (v, v) allant d'un sommet $v \in V$ vers lui-même. Un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

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

G



G'



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

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

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

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

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

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

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

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

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

A3 – Comptage de palindromes

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

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

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

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

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

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

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

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

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

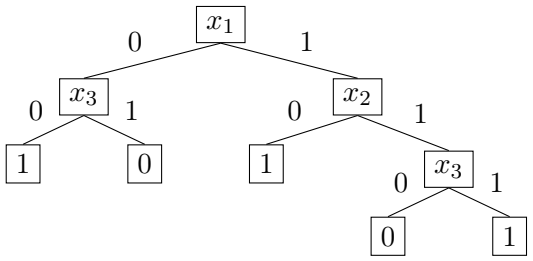
Question 6. Sur la base de cette observation, améliorer l'algorithme de la question 4 pour qu'il fonctionne en temps linéaire.

A4 – Arbres de décision

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

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

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



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

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

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

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

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

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

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

A5 – Ensembles inévitables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

J1 – Valeurs plus faibles les plus proches et arbres cartésiens

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

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

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

Question 0. Supposons, pour cette question uniquement :

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

Calculer alors le contenu du tableau a.

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

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

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

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

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

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

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

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

J2 – Preuve de programmes en logique de Hoare

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

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

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

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

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

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

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

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

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

soit valide.

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

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

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

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

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

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

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

(b) ... du triplet suivant :

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

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

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

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

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

(a) ... s'exécute sans erreur.

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

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

J3 – Cribles

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

J4 – Théorie de Sprague-Grundy

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

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

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

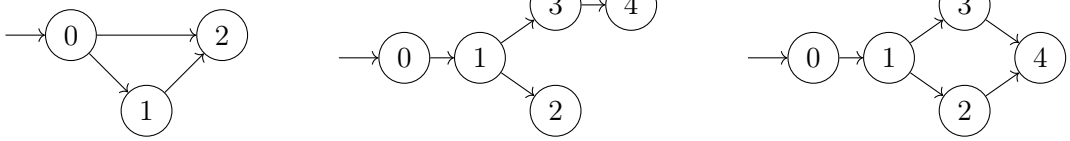
Question 0. Démontrer que la séquence (s_k) est finie, c'est-à-dire qu'il existe un entier n_{fin} tel que $\varphi_{n_{\text{fin}} \bmod 2}(s_{n_{\text{fin}}})$ n'est pas défini.

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

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

Question 1.

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



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

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

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

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

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

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

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

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

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

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

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

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

J5 – Arbres à boucs émissaires

Question 0.

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

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

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

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

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

Question 1.

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

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

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

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

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

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

Question 4.

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

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

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

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

Question 5.

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

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

L1 – May the forcing be with you

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

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

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

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

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

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

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

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

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

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

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

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

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

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

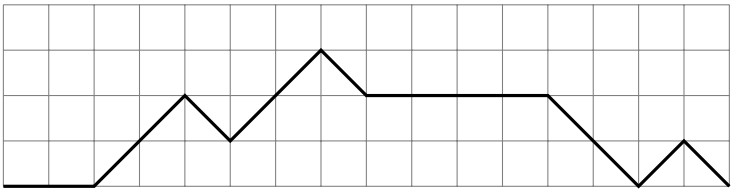
Question 4. Soit $f : \mathbb{N} \to \Sigma$ une fonction rencontrant tous les ensembles denses de la question 3. Montrer que f n'est pas calculable.

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

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

L2 – Nombres de Schröder

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



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

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

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

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

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

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

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

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

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

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

L3 – Complexité de Kolmogorov

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

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

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

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

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

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

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

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

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

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

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

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

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

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

L4 – Syndéticité par parties

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

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

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

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

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

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

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

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

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

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

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

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

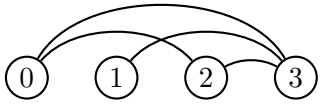
Question 6. Écrire le pseudocode d'un algorithme qui prend en entrée un ensemble fini S et un entier $d \in \mathbb{N}$, et détermine si S est localement d-syndétique. L'ensemble S est donné comme un tableau T de booléens où $T[i]$ indique si i appartient à S. Discuter de sa complexité en temps et en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

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

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

L5 – Graphes de saut

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



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

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

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

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

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

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

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

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

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

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