BANQUE MP INTER-ENS – SESSION 2021
RAPPORT SUR L'ÉPREUVE PRATIQUE D'ALGORITHMIQUE ET DE PROGRAMMATION DU CONCOURS COMMUN DES ÉCOLES NORMALES SUPÉRIEURES

Écoles concernées : Lyon, Paris-Saclay, Rennes, Ulm

Coefficients (en pourcentage du total d'admission)
- Lyon (toutes options) : 16,9 %
- Paris-Saclay : 13,2 %
- Rennes : 17,1 %
- Ulm : 13,3 %

Jury : Louis Jachiet, Samuel Thibault, Jill-Jênn Vie, Yannick Zakowski

Contenu de ce document

Nous rappelons l'organisation de l'épreuve, annonçons une nouveauté des éditions futures, puis faisons des remarques générales sur son déroulement cette année. Nous faisons ensuite un court compte-rendu, essentiellement statistique, pour chacun des quatre sujets. Nous proposons ensuite nos corrigés : certains comportent des éléments de réponse pour la partie orale, un autre une correction en OCaml, un autre une correction en Python.

Organisation de l'épreuve

L'objectif de cette épreuve est d'évaluer la capacité de mettre en œuvre une chaîne complète de résolution d'un problème informatique, à savoir la construction d'algorithmes, le choix de structures de données, leurs implémentations, et l'élaboration d'arguments mathématiques pour justifier ces décisions. Le déroulement de l'épreuve est le suivant : un travail sur machine d'une durée de 3 h 30, immédiatement suivi d'une présentation orale pendant 20 minutes.

Juste avant la distribution des sujets, les candidat·es disposent d'une période de 10 minutes pour se familiariser avec l'environnement informatique et poser des questions aux surveillants si elles ou ils rencontrent des difficultés d'ordre pratique.

Un sujet contient typiquement une dizaine de questions écrites et une dizaine de questions orales. Tous les sujets commencent par la génération pseudo-aléatoire d'entrées pour le problème étudié. Nous invitons fortement les candidat·es à se familiariser à l'avance avec la manière dont ces suites pseudo-aléatoires sont générées et utilisées dans les sujets précédents afin de gagner du temps le jour de l'épreuve. En particulier, les suites pseudo-aléatoires dépendent d'un u_0 qui est donné individuellement à chaque candidat·e au début de l'épreuve.

Les questions écrites attendent des réponses purement numériques. Chaque question requiert l'implémentation d'un algorithme et son utilisation sur les entrées générées au début du sujet. Une question est typiquement divisée en sous-questions pour des entrées de plus en plus grandes, ce qui permet de tester l'efficacité de l'algorithme mis en œuvre. À la fin de la partie pratique, les candidat·es remettent au jury une fiche-réponse contenant les réponses numériques aux questions écrites.

Une aide précieuse est donnée aux candidat·es sous la forme d'une fiche-réponse type pour $\widetilde{u_0}$. Cette fiche permet de vérifier l'exactitude des réponses pour une graine différente du u_0 de la candidate ou du candidat. Il est très fortement recommandé, comme indiqué dans l'introduction des sujets, de vérifier que le générateur aléatoire se comporte comme

attendu avec la graine $\widetilde{u_0}$, pour chaque question. Les examinateurs ont encore eu quelques (rares, heureusement) cas de candidat·es traitant le sujet avec un générateur faux et donc sans possibilité de diagnostiquer efficacement leurs erreurs.

Les questions orales sont de nature plus théorique et sont destinées à être présentées pendant l'oral. Le déroulement de l'oral est le suivant : nous commençons par demander de présenter le plus efficacement possible les questions orales préparées pendant la première phase et, s'il reste du temps, s'ensuit une discussion avec le jury sur les questions non traitées. La présentation orale vise à évaluer la bonne compréhension du sujet et le recul des candidat·es. Les examinateurs s'efforcent d'aborder toutes les questions préparées pendant la première étape, et, suivant le temps disponible, des extensions de ces questions ou des questions qui n'ont pas été traitées par manque de temps. Pour réaliser un bon oral, il est important de prendre le temps de réfléchir aux questions à préparer mentionnées dans le sujet et de préparer suffisamment de notes au brouillon pour être capable d'exposer clairement les solutions au tableau.

La partie écrite de l'épreuve représentait cette année de l'ordre de 60 % de la note finale, le coefficient exact variant suivant le sujet. On observe dans l'ensemble, mais pas systématiquement, une bonne corrélation entre les résultats obtenus aux deux parties.

Nouveauté du concours 2022 : lecture de fichiers ou de l'entrée standard

Pour les éditions futures du concours, nous souhaitons diversifier l'évaluation de cette épreuve. Ainsi les candidat·es pourront être amené·es à devoir lire l'entrée standard, ou bien des fichiers contenant des instances d'objets. Le cas échéant, nous fournirons des bouts de code sur la clé USB fournie. En voici quelques exemples.

Lecture d'un fichier. On suppose que les données se trouvent dans `fichier.txt` et que l'on souhaite récupérer une liste de chaînes de caractères contenant pour chaque ligne un élément dans la liste :

fichier.py — Code Python 3

```python
with open('fichier.txt') as f:
    lignes = []
    for ligne in f:
        lignes.append(ligne.strip())  # Enlever le \n final
    # Alternativement aux trois lignes qui précédent on peut utiliser
    # la ligne suivante :
    lignes = f.read().splitlines()
```

fichier.ml — Code OCaml 4.04

```
let lignes =
  let fichier = open_in "fichier.txt" in
  let str = really_input_string f (in_channel_length f) in
  close_in f;
  String.split_on_char '\n' str
```

Pour les versions précédentes d'OCaml, il est possible d'obtenir la même chose avec d'autres fonctions de la bibliothèque `Str`, à condition de l'importer avec `open Str`.

Lecture de l'entrée standard. Supposons que l'entrée standard contienne la hauteur, la largeur, et le contenu d'une grille au format ASCII.

Entrée standard

```
3
5
#####
#...#
#####
```

Note. Le contenu d'un fichier `entree.txt` peut aussi être lu via l'entrée standard en lançant le programme d'une des façons suivantes :

```
python entree.py < entree.txt
ocaml entree.ml < entree.txt
```

où `entree.py` et `entree.ml` sont les programmes lisant sur l'entrée standard suivants :

`entree.py` Code Python 3

```python
hauteur = int(input())
largeur = int(input())
grille = []
for _ in range(hauteur):
  grille.append(input())
```

`entree.ml` Code OCaml

```
let hauteur = read_int ()
let largeur = read_int ()
let grille = List.init hauteur (fun _ -> read_line ())
```

Remarques générales

Écriture du programme. Dans certains cas, les examinateurs ont inspecté le code des candidat·es afin de lever certaines ambiguïtés lors de leur présentation de leurs algorithmes. Cela était possible uniquement pour les candidat·es qui avaient soigné la lisibilité de leur code, et seulement si les réponses aux questions étaient facilement identifiables et exécutables. Nous conseillons ainsi aux candidat·es de soigner la lisibilité de leur code. Les candidat·es peuvent s'inspirer des propositions de corrigés fournies en annexe de ce rapport.

Par ailleurs, plusieurs candidat·es mélangent le u_0 commun à tous les sujets pour tester leur code et le u_0 de leur code. Pour éviter cela, nous recommandons fortement d'éviter les copier-coller avec une version du code pour le u_0 et une pour le $\widetilde{u_0}$, et plutôt de lire le u_0 dans une variable et d'exécuter tout le code avec, cf. les différents corrigés proposés.

La durée de l'oral étant courte relativement au nombre de questions pouvant être traitées, nous conseillons aux candidat·es de préparer une réponse précise mais intuitive plutôt que de se perdre dans une preuve laborieuse au tableau. Si le jury n'est pas convaincu par un argument simple, il sera toujours possible de le convaincre par une preuve détaillée sans que cela impacte la note finale. Inversement, si le jury est convaincu par un raisonnement intuitif, la candidate ou le candidat dispose alors de plus de temps pour aborder des questions globalement peu traitées et donc susceptibles de rapporter beaucoup de points. Par exemple, nombre de candidat·es se lancent dans d'interminables preuves par induction alors qu'il existe parfois une explication intuitive immédiate : nous encourageons les candidat·es à favoriser la seconde. La capacité à exposer un argument formel

pour répondre à une question est évaluée dans le cadre de l'épreuve d'informatique fondamentale, tandis que l'objet de l'oral ici est de s'assurer que les candidat·es font le lien entre la résolution d'un problème informatique dans un cadre formel inédit et sa mise en pratique. Le jury saura donc apprécier le recul que démontre un argument simple et intuitif par rapport à une suite d'arguments formels désincarnés.

Sur la gestion du tableau, nous invitons les candidat·es à éviter l'écueil suivant : écrire tout leur raisonnement au tableau et ainsi perdre beaucoup de temps. L'écueil inverse, de ne pas utiliser du tout le tableau est plus rare, mais il rend parfois le raisonnement difficile à suivre. Il est difficile de donner une règle générale sur l'utilisation du tableau mais quand la preuve s'explique bien par un exemple ou un dessin, alors il ne faut pas hésiter à faire le dessin au tableau (par exemple de donner un petit graphe d'exemple) et ensuite de faire la preuve dessus à l'oral. Si le ou la candidate veut se lancer dans une preuve par induction, il peut aussi être intéressant d'écrire l'hypothèse mais pas tout le raisonnement.

Certaines questions orales demandent aux candidat·es de présenter leur algorithme et d'analyser leur complexité. Nous encourageons vivement les candidat·es à présenter leurs algorithmes de façon claire et concise. Contrairement à ce que nous avons parfois pu constater, il ne s'agit pas de recopier un programme en pseudo-Python ou pseudo-Caml au tableau. Il faut s'efforcer de présenter (uniquement) les étapes clés de l'algorithme, en langage naturel si possible, afin de supporter efficacement l'analyse de complexité par la suite. En particulier, il est essentiel d'identifier clairement la structure itérative ou récursive d'un algorithme. Trop souvent, des candidat·es se sont trompé·es entre une complexité en $O(m+n)$ et une complexité en $O(m \times n)$ à cause d'une présentation de l'algorithme trop confuse. On visera donc à être à la fois concis·e sans pour autant sacrifier la précision de la présentation. Enfin, ce type de question ne doit pas empêcher un·e candidat·e de proposer un algorithme simple (accompagné d'une analyse de complexité correcte) même si l'implémentation de l'algorithme en question n'a pas été achevée durant la partie pratique de l'épreuve. Si l'algorithme proposé est en réalité trop naïf pour traiter les instances proposées dans le sujet, le jury saura apprécier un regard critique sur l'algorithme qui exploiterait l'analyse de complexité.

Nous conseillons très fortement aux futur·es candidat·es de s'entraîner à faire un ou deux sujets et à lire des corrections pour avoir une idée des subtilités algorithmiques qui les attendent et maîtriser les premières questions et choix de structures algorithmiques qui sont similaires d'un sujet à l'autre.

Langages de programmation. Les sujets sont de difficulté équivalente dans les divers langages Python et OCaml. Nous notons une tendance générale à utiliser plutôt Python que OCaml. Certains élèves hésitent et passent du temps à chercher le *meilleur* langage pour le sujet. Ceci est une perte de temps, les sujets sont calibrés pour être implémentables avec le même niveau de difficulté en OCaml et en Python. On conseille donc aux candidat·es de choisir à l'avance un langage qu'ils connaissent le mieux. Cela permet de s'entraîner pour bien connaître et éviter les problèmes et limitations liées au langage. Souvent, des candidat·es se lancent en Python sans se rappeler, par exemple, que dans ce langage les appels récursifs sont limités par défaut à 1000 (cf. `sys.setrecursionlimit`) et que les listes sont représentées par des tableaux.

Pour finir, nous voulons aussi conseiller aux candidat·es d'étudier les structures de données basiques pour chaque langage. La différence en efficacité d'un programme qui utilise une liste au lieu d'un tableau est très visible dans ce type de sujet. Cela ne veut pas dire que pendant l'oral nous espérons avoir tous les détails de l'implémentation. Au contraire, les meilleur·es candidat·es se focalisent peu sur l'utilisation ou non d'une liste ou d'une table dans leur propre implémentation.

Exemple. Nous terminons par un exemple, la présentation d'un algorithme de parcours de graphe. Voici les quatre phrases que l'on attend pour une telle question.

- Un algorithme de parcours de graphe part d'un sommet et suit les arêtes pour visiter les sommets du graphe connectés au sommet original.
- L'ordre de traitement des arêtes est déterminé par le choix du parcours : en largeur, on traite en priorité les arêtes par distance croissante au sommet original, et en profondeur, on traite en priorité les arêtes sortant du dernier sommet visité. Ceci induit la structure de donnée utilisée : une file (FIFO) pour un parcours en largeur, une pile (LIFO) pour un parcours en profondeur.
- Dans les deux cas, chaque arête est mise dans la structure de données exactement une fois, ce qui est assuré par un tableau de booléens déterminant si un sommet a déjà été visité ou non.
- La complexité du parcours est ainsi $O(n + m)$, où n est le nombre de sommets et m le nombre d'arêtes.

Sujet 1 : Comptage de mots reconnus par un automate.

Ce sujet abordait différentes stratégies pour compter le nombre de mots ou de facteurs d'un mots acceptés par un automate donné. Le sujet commence de façon classique par la génération de mots et d'automates avant de passer au décompte des mots ou facteurs acceptés par l'automate et enfin de passer au cas d'un mot régulièrement mis à jour.

Pour la partie écrite, les questions Q1 à Q4 étaient de purs exercices de programmation et ont été correctement traitées par presque tou·te·s les candidat·es. Les questions Q5 à Q7 nécessitaient une légère réflexion algorithmique pour obtenir les bonnes complexités et ont été bien réussies par une très large majorité des élèves. Les questions Q8 et Q9 ont été déterminantes dans ce sujet. Une moitié des candidat·es a réussi à obtenir tous les points sur Q8. Beaucoup de candidat·es ayant réussi Q8 ont abordé Q9 mais tou·tes n'ont pas vu la subtilité qui consistait à choisir le meilleur des deux algorithmes entre Q8 et Q7.

Le sujet était probablement plus court que d'ordinaire mais le jury a été agréablement surpris par le fait que plusieurs élèves ont attaqué et réussi les dernières questions : Q10 a été réussie par 7 élèves et Q11 par 3. Le jury est impressionné de ces 3 élèves qui ont fini le sujet écrit et ces élèves ont obtenu de très bonnes notes mais aucun·e n'a obtenu la note de 20/20, peut-être aurait-il fallu mieux préparer la partie orale ?

Comme à l'ordinaire, il y a une très forte corrélation entre la réussite à l'écrit et à l'oral, même si certain·es candidat·es ont pu gagner des points grâce à des idées non transformées en code faute de temps, ou de perdre des points, n'ayant pas pu justifier certaines propriétés ou complexités correctement. L'oral est aussi l'occasion pour le jury de constater que certain·es ont des choix très fantaisistes pour les structures de données ce qui généralement se traduit par peu de questions écrites réussies. Ce problème est probablement lié au manque d'entraînement.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Tous les points	95	100	100	97	89	84	76	53	37	18	8
Réponses partielles	100	100	100	100	95	92	87	71	53	18	8

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9
Tous les points	87	76	89	87	37	32	45	16	5
Réponses partielles	100	100	100	100	92	79	63	32	18

Table 1. Taux de candidat·es ayant répondu à chaque question du sujet 1

Sujet 2 : Comment couper la poire en deux ?

Ce sujet abordait la question du partitionnement de graphe en deux parties de poids équilibrés tout en modérant le nombre d'arêtes à cheval entre les deux parties. Le sujet commençait ainsi par la génération pseudo-aléatoire de graphes simples. Une deuxième partie s'intéressait alors au cas où l'on partitionne le graphe en deux parties ayant le même nombre de sommets, plusieurs heuristiques étant étudiées. La troisième partie s'intéressait au cas des graphes planaires, qu'il fallait donc générer, avant de calculer des arbres couvrants, permettant ainsi une heuristique de faible complexité.

La question écrite Q1 n'a posé aucun problème. Pour la question écrite Q2 il était important d'éviter une complexité quadratique qui a empêché quelques candidat·es de pouvoir obtenir les résultats les plus lourds pour Q2 et Q3. La question orale QO1 a permis d'expliquer cette difficulté. La QO2 a permis de discuter de la structure de données à utiliser. Puisque par construction les graphes comportaient $O(n \cdot \sqrt{n})$ arêtes, l'utilisation d'une matrice d'adjacence ou de listes d'adjacence pouvaient toutes deux se justifier. La plupart des candidat·es ont pu motiver leur choix.

La question écrite Q4 a commencé à être déterminante. La plupart des candidat·es a pu obtenir les premiers résultats, mais les résultats sur de grands graphes nécessitaient de soigneusement mémoriser le poids de la coupe en cours de considération pour éviter d'avoir à le recalculer à chaque étape. Seul un quart des candidat·es a pu obtenir tous les résultats pour Q4. La Q5 accentuait cette difficulté, et seul un sixième des candidat·es a pu obtenir tous les résultats pour Q5. Cette difficulté se ressent également sur les questions orales QO3 et QO4. Diverses optimisations étaient possibles. Seul·es quelques candidat·es ont su en tirer entièrement parti, mais la moyenne à la question QO3 est de 54/100. La question orale QO5 était l'occasion de donner une intuition sur le comportement concret de l'heuristique proposée. Elle a en général été bien abordée. La QO6 précisait la question ce qui exigeait un raisonnement vraiment correct. Ce sont essentiellement les quelques candidat·es qui ont pu obtenir tous les résultats pour Q4 et Q5 qui ont pu aborder Q6.

Pour la question écrite Q7, les candidat·es n'osent pas implémenter un algorithme naïf sous prétexte qu'il a une complexité exponentielle. n était explicitement petit (au plus 24), c'était donc effectivement ce qui était attendu, et même un algorithme très naïf pouvait fournir les réponses pour les graphes de petite taille. Certain·es candidat·es sont effectivement allés y chercher quelques points. Pour obtenir tous les résultats il était nécessaire d'éviter les recalculs de poids et éviter d'énumérer toutes les partitions possibles pour réduire la complexité (qui bien sûr reste exponentielle). La question orale QO7 était l'occasion d'expliquer au moins l'approche simpliste même si elle n'avait pas été implémentée. Peu de candidat·es savent faire une exploration exhaustive correctement.

La génération de graphes planaires et leur représentation a été abordée à l'oral pour QO8 par un quart des candidat·es, mais la réalisation écrite pour Q8 n'a été abordée que par un candidat. Seul ce même candidat a abordé la question orale QO9 plutôt intuitive sur le partitionnement de ces graphes planaires, mais aucune réalisation écrite n'a pu être faite. Les questions Q9 et Q10 (et donc QO10) nécessitaient un très bon recul sur l'algorithmique des graphes, elles n'ont pas été abordées.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Tous les points	100	97	85	24	15	15	12	0	0	0
Réponses partielles	100	100	100	64	52	15	21	3	0	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10
Tous les points	94	73	9	85	70	39	15	15	0	0
Réponses partielles	100	97	100	85	82	48	39	24	3	0

Table 2. Taux de candidat·es ayant répondu à chaque question du sujet 2

Sujet 3 : Jeux sur des arbres (*tries on tries*)

Ce sujet portait sur la construction d'un arbre lexicographique représentant les coups possibles d'un jeu à deux joueurs dont il faut identifier une stratégie gagnante. La troisième partie, conceptuellement plus difficile, considérait un cas particulier d'adversaire et faisait implémenter un algorithme de Q-learning (apprentissage par renforcement). Ce jeu existe dans la vraie vie sous diverses variantes de la condition de terminaison : quart de singe en français, *Ghost* en anglais.

Les questions Q1 à Q3 permettaient de faire connaissance avec les objets du sujet, et c'est à partir de Q5 qu'il était nécessaire d'avoir une construction de l'arbre lexicographique pour avancer et nécessaire de prendre du recul sur cette structure de données pour avoir les complexités attendues.

Nous sommes surpris que si peu de candidat·es aient correctement traité la Q4 alors qu'une simple observation, déjà identifiée à la QO1 (les nœuds terminaux correspondent aux mots distincts) permet de la résoudre sans construire l'arbre lexicographique, à condition de savoir retirer les doublons d'une liste de mots. Sans doute se jeter dans la construction de l'arbre lexicographique, hors programme, a donné du fil à retordre à la plupart des candidat·es. Dans l'ensemble, la plupart des candidat·es ont eu les bonnes intuitions derrière l'algorithme récursif pour déterminer la stratégie gagnante de QO3.

Pour le nombre de mots inaccessibles Q5, il était plus simple et économique de raisonner par complémentaire, en termes de nœuds accessibles, plutôt que de tester pour chaque préfixe de chaque mot s'il est dans le corpus. Ainsi, un parcours de l'arbre suffit à répondre à la question, qu'il soit en largeur ou profondeur.

Pour la Q6, certain·e·s candidat·es ont mal compris la définition de nœud gagnant, dont il est clairement précisé dans le sujet qu'elle se réfère au joueur 1. Cela pouvait leur coûter en termes de points à l'écrit, mais malgré cette maladresse, leur raisonnement correct pouvait donner des points à l'oral.

Beaucoup de candidat·es ne songent pas à faire des dessins, ce qui était essentiel pour la question difficile Q7 et qui pouvait aider à résoudre la question Q6.

La troisième partie, de Q8 à Q10, a été peu traitée. Elle introduisait de nombreuses notations, mais n'était pas difficile d'un point de vue algorithmique. Comme indiqué au début de chaque sujet, on recommande de lire l'entièreté du sujet avant de commencer à programmer ; par exemple, la Q8 voire la Q06 étaient à la portée de tout candidat·e.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Tous les points	95	95	85	75	62	38	12	38	2	0
Réponses partielles	100	95	92	90	75	50	22	55	2	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9
Tous les points	50	60	52	65	30	15	2	0	0
Réponses partielles	100	100	92	80	55	32	12	0	5

Table 3. Taux de candidat·es ayant répondu à chaque question du sujet 3

Sujet 4 : Coloriages de graphes.

Ce sujet abordait le problème du coloriage de graphes. Ce problème étant NP-complet il a été principalement traité à l'aide de diverses heuristiques ou cas particuliers

Après la traditionnelle question sur la génération de nombres pseudo-aléatoires, le sujet commence vraiment avec l'étude d'une fonction appelée `ÉlimineDoublon` qui, comme

son nom l'indique, élimine les doublons d'une liste. Bien que relativement élémentaire dans son code et facilement remplaçable par des fonctions de la bibliothèque (par exemple, `list(set(l))` en Python), cette fonction est introduite pour aider les candidat·es à générer efficacement les graphes.

Les trois premières questions de programmation ont été quasiment unanimement réussies. Les candidat·es ont tou·tes obtenu des points sur la première question orale qui concernait la fonction `ÉlimineDoublon`. En revanche, presque la moitié n'a pas obtenu tous les points car l'analyse de complexité n'était pas la bonne ou la présentation de la correction de cette fonction n'était pas claire. C'est dommage car il semble que la fonction a bien été comprise mais les candidat·es n'ont que peu préparé la question.

Pour vérifier la génération de graphe, nous avons utilisé la seconde question orale ainsi que la quatrième question écrite qui demande le nombre total d'arêtes. Plusieurs candidat·es n'ont pas réussi à générer les graphes les plus grands, souvent par manque de bonne méthode pour construire le graphe. C'est dommage car une méthode est pourtant donnée explicitement dans le sujet.

Après la génération de graphe, le sujet s'intéresse à une première heuristique de coloration où l'on traite les nœuds dans l'ordre, remplaçant la couleur de chaque nœud par la plus petite couleur non utilisée par ses voisins. La principale difficulté de cette question consiste à trouver le plus petit entier positif qui n'est pas dans une liste de valeurs. Cette question écrite n'a pas posé de problèmes aux candidat·es (à part la génération du grand graphe), en revanche les questions orales ont posé des soucis. La question orale 4 était une véritable question d'informatique, et trouver un exemple où l'heuristique ne reste pas proche du nombre chromatique était difficile. Cette question a beaucoup pénalisé les candidat·es qui ne l'avaient pas préparée. La question orale 3 était une question assez classique d'analyse de complexité et de correction. La principale difficulté étant de remarquer que la somme du nombre de voisins est égale au nombre d'arêtes.

Le sujet passe ensuite au cas de la bicoloration où il faut remarquer que cette question se traite avec un simple parcours de graphe (le sujet fournit une indication à ce propos) où les nœuds ont un de ces trois états : visité de profondeur paire, visité de profondeur impaire, et non visité. Cette question ainsi que la question orale correspondante ont été bien traitées ce qui montre que les candidat·es maîtrisent assez bien le parcours de graphe.

Pour les deuxième et troisième heuristiques étudiées, bien que la solution attendue ne soit pas forcément très longue (en nombre de lignes de code) il fallait avoir une véritable réflexion algorithmique pour obtenir la complexité attendue et donc tous les points. Il existe différentes manières d'aborder ces questions mais celle utilisée par le corrigé repose sur la technique « paresseuse ».

Enfin la dernière question du sujet était une question assez ouverte. Il était facile avec un algorithme « brute-force » d'obtenir la réponse aux premières sous-questions que l'on devait progressivement optimiser pour obtenir les réponses aux sous-questions suivantes.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Tous les points	100	92	90	80	75	42	12	5	0
Réponses partielles	100	100	100	92	98	72	40	52	12

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9
Tous les points	98	55	35	22	38	52	2	10	2
Réponses partielles	100	100	100	95	90	88	52	52	10

TABLE 4. Taux de candidat·es ayant répondu à chaque question du sujet 4

Comptage de mots reconnus par un automate.

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2021

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !

Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...

Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***. Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

1 Préliminaires

Notations et rappels sur les automates et les mots

On rappelle que pour deux entiers naturels a et b, $a \mod b$ désigne le reste de la division entière de a par b, c'est à dire l'unique entier r avec $0 \leq r < b$ tel que $a = k \times b + r$ pour $k \in \mathbb{N}$.
Un mot w de longueur n sur un alphabet Σ est constitué de n lettres $w_0 \dots w_{n-1}$ (toutes dans Σ).
Étant donné un mot $w = w_0 \dots w_{n-1}$ de longueur n et deux entiers $0 \leq i \leq j \leq n$, on note $w[i:j]$ le mot $w_i \dots w_{j-1}$, c'est à dire le facteur du mot w entre les positions i (incluse) et j (exclue).
Un automate fini déterministe est usuellement décrit par un quintuplet $(\Sigma, \mathcal{Q}, q_0, \delta, \mathcal{F})$ où :

- Σ représente l'alphabet sur lequel l'automate travaille,
- $\mathcal{Q}$ représente l'ensemble des états,
- q_0 est l'unique état initial,
- δ est la fonction de transition de l'automate, telle que $\delta(q, c)$ donne l'état dans lequel l'automate se trouve après avoir lu la lettre c dans l'état q,
- $\mathcal{F}$ est l'ensemble des états finaux.

Dans ce sujet, Σ et $\mathcal{Q}$ seront toujours des ensembles de nombres entiers de la forme $0, 1, 2 \dots, n-1$ (où $n > 0$ est la taille de l'ensemble) tandis que l'état initial sera toujours 0. Ainsi Σ et $\mathcal{Q}$ seront entièrement définis par leur taille et donc, pour ce sujet, l'automate est entièrement décrit par le quadruplet $(L, Q, \delta, \mathcal{F})$ où L est la taille de l'alphabet, Q est le nombre d'états de l'automate, δ est la fonction de transition et $\mathcal{F}$ est l'ensemble des états finaux.
De façon usuelle, on étend la fonction de transition δ aux mots de la façon suivante $\delta(q, \epsilon) = q$ (ici ϵ représente le mot vide) et $\delta(q, uv) = \delta(\delta(q, u), v)$ (ici uv représente la concaténation de u et v).
Quand un automate lit un mot w, il arrive donc dans l'état $\delta(0, w)$. Si $\delta(0, w) \in \mathcal{F}$ le mot est alors accepté par l'automate sinon il est refusé.

Générateur de nombres pseudo-aléatoires

Étant donné u_0 on définit la récurrence suivante :

$$v(0) = u_0$$
$$\forall t \in \mathbb{N},\ v(t+1) = 101833 \times v(t) \mod 1\,000\,000\,007$$

Et ensuite on définit u de la façon suivante :

$$\forall t \in \mathbb{N},\ u(t) = v(t \mod 1\,000\,003)$$

L'entier u_0 vous est donné, et doit être recopié sur votre fiche réponse avec vos résultats. Une fiche réponse type vous est donnée en exemple, et contient tous les résultats attendus pour une valeur de u_0 différente de la vôtre (notée $\widetilde{u_0}$). Il vous est conseillé de tester vos algorithmes avec cet $\widetilde{u_0}$. Pour chaque calcul demandé, avec le bon choix d'algorithme le calcul ne devrait demander qu'au plus quelques secondes, jamais plus d'une minute.

Question 1 *Calculer les valeurs suivantes :*

a) $u(1) \mod 1\,000$, ***b)*** $u(42) \mod 1\,000$, ***c)*** $u(10^9) \mod 1\,000$.

```
let u0 = read_int ()
let umax = 1000003
let u =
  let v = Array.make umax u0 in
  for i = 0 to umax-2 do
    v.(i+1) <- 101833 * v.(i) mod 1000000007 ;
  done ;
  fun n -> v.(n mod umax)
```

Génération d'automates

On définit l'automate fini déterministe $\mathcal{E}(L, Q)$ par le quadruplet $(L, Q, \delta, \mathcal{F})$ avec $\delta(q, c) = (u(q \times L + c) \mod Q)$ et l'on a $i \in \mathcal{F}$ si et seulement si $u(Q \times L + i)$ est impair.

Question 2 *On note* TransitionsAcceptantes($\mathcal{A}$) *le nombre de paires* (q, c) *où* q *est un état et* c *une lettre, telles que* $\delta(q, c) \in \mathcal{F}$. *Calculer* TransitionsAcceptantes($\mathcal{A}$) *pour les automates* $\mathcal{E}(L, Q)$ *avec les valeurs de* L *et* Q *suivantes :*

a) $L = 2$, $Q = 100$ ***b)*** $L = 10$, $Q = 10\,000$, ***c)*** $L = 3$, $Q = 30\,000$.

```
let e q l =
  let delta q' c = u(q'*l+c) mod q in
  let f i = u(q*l+i) mod 2 = 1 in
  (q,l,delta,f)
  (* La fonction e renvoie deux entiers q et l et deux fonctions delta
  et f. Ces fonctions prennent un temps constant à s'exécuter.  *)

let list_init n f =
(* Existe déjà sous le nom List.init dans les versions récentes d'OCaml.
  Initialise une liste l_0 ... l_{n-1} où l_i = f (i). *)
  let rec foo l = function
    | 0 -> l
    | i ->  foo (f (i-1)::l) (i-1)
  in
  foo [] n

(* Compte le nb de transitions acceptantes *)
let transitionsAcceptantes (q,l,d,f) =
  list_init (q*l) (fun i -> (i mod q),(i/q)) |> (* toutes les paires (q,c)*)
    List.map (fun (q',c) -> (d q' c)) |> (* les delta(q,c) *)
    List.filter f |> (* ceux qui sont finaux *)
    List.length (* calcule leur nombre *)
```

Génération de mots

On définit le mot $\mathcal{M}(L,T)$ de longueur T sur l'alphabet avec L lettres comme le mot $w_0 \dots w_{T-1}$ avec, pour $0 \leq i < T$, $w_i = (u(i) \mod L)$.

Question 3 *Calculer les mots suivants :*

a) $\mathcal{M}(4,3)$ ***b)*** $\mathcal{M}(8,4)$ ***c)*** $\mathcal{M}(10,5)$

```
let m l t = list_init t (fun i -> u(i) mod l)
```

On note $\textsc{Etat}(\mathcal{A}, w)$ l'état dans lequel se trouve l'automate après avoir lu le mot w, c'est à dire $\textsc{Etat}(\mathcal{A}, w) = \delta(0, w)$.

Question 4 *Déterminer* $\textsc{Etat}(\mathcal{E}(L,Q), \mathcal{M}(L,T))$ *pour les valeurs de* L, Q *et* T *suivantes :*

a) $L = 2$, $Q = 100$, $T = 100$,

b) $L = 10$, $Q = 10\,000$, $T = 50\,000$,

c) $L = 3$, $Q = 30\,000$, *et* $T = 100\,000$.

```
let etat (q,l,d,f) mot = List.fold_left d 0 mot
  (* fold_left d 0 [w_0; ... ; w_{n-1}] = d(... d(0,w_0) ...,w_{n_1})  *)
```

Question à développer pendant l'oral 1 *Décrire la structure de donnée utilisée pour représenter les automates et la complexité de l'algorithme pour calculer* $\textsc{Etat}(\mathcal{E}(L,Q), \mathcal{M}(L,T))$ *en fonction de* L, Q *et* T. *Quel algorithme proposez-vous pour savoir si* $\mathcal{M}(L,T)$ *est reconnu par* $\mathcal{E}(L,Q)$, *avec quelle complexité ?*
Votre calcul de complexité ne doit pas prendre en compte le temps de construire $\mathcal{E}(L,Q)$ *et* $\mathcal{M}(L,T)$ *mais seulement le temps de calculer* $\textsc{Etat}(\mathcal{E}(L,Q), \mathcal{M}(L,T))$ *une fois que ceux-ci sont construits.*

L'automate est représenté par un quadruplet de deux entiers (`q` le nombre d'états, `l` le nombre de lettres de l'alphabet) et de deux fonctions (`d` la fonction de transition, `f` la fonction déterminant si un état est final). Ces deux fonctions s'exécutent en temps constant.
La calcul de $\textsc{Etat}(\mathcal{E}(L,Q), \mathcal{M}(L,T))$ est linéaire en T. En effet on ne fait que, pour chaque lettre c_i, calculer $q_{i+1} = \delta(q_i, c_i)$.
Pour tester si un mot w est reconnu par un automate $\mathcal{A}$ on peut calculer $\textsc{Etat}(\mathcal{A}, w)$ et regarder si l'état obtenu est un état final. La complexité est linéaire en la taille du mot.

Accessibilité dans l'automate

Un état q de l'automate est dit accessible s'il existe un mot w tel que $q = \delta(0, w)$.

Question à développer pendant l'oral 2 *Pour un automate $\mathcal{A}$, on note* GrapheAccessibilité($\mathcal{A}$) *le graphe dont les nœuds sont les états de $\mathcal{A}$ et il y a un arc de q vers q' quand il existe une lettre c telle que $\delta(q,c)=q'$. Justifier qu'un état q est accessible dans l'automate $\mathcal{A}$ si et seulement si il existe un chemin de l'état initial de $\mathcal{A}$ à q dans le graphe* GrapheAccessibilité($\mathcal{A}$).

On peut ajouter des étiquettes sur le graphe GrapheAccessibilité($\mathcal{A}$). L'arc $q \to q'$ est étiqueté par la lettre c telle que $\delta(q,c)=q'$. Si plusieurs c existent on en choisit une, peu importe laquelle. S'il y a un chemin de 0 à q dans GrapheAccessibilité($\mathcal{A}$), on peut regarder les étiquettes des arcs de ce chemin et on obtient un mot w tel que $\delta(0,w)=q$.
Inversement si un état q de l'automate est accessible c'est qu'il existe un mot $w = w_0 \dots w_k$ tel que $q = \delta(q,w)$. On peut alors en déduire un chemin $n_0 \dots n_{k+1}$ dans le graphe GrapheAccessibilité($\mathcal{A}$) avec $n_0 = 0$ et $n_{i+1} = \delta(n_i, w_i)$. C'est bien un chemin de GrapheAccessibilité($\mathcal{A}$) par construction.

Question 5 *Notons* NbEtatsAccessibles($\mathcal{A}$) *le nombre d'états accessibles dans l'automate $\mathcal{A}$. Calculer* NbEtatsAccessibles($\mathcal{E}(L,Q)$) *pour les valeurs de L et Q suivantes :*

a) $L=2$, $Q=100$, ***b)*** $L=10$, $Q=10\,000$, ***c)*** $L=3$, $Q=30\,000$.

```
let nbAccessibles (q,l,d,f) =

  let vu = Array.make q false in

  let voisins x = list_init l (fun c->d x c) in

  let rec explore = function
    | [] ->  ()
    | x::t ->
       if not vu.(x)
       then
         begin
           vu.(x) <- true ;
           explore ((voisins x)@t)
         end
       else
         explore t
  in
  explore [0] ;
  Array.to_list vu |> List.filter (fun x->x) |> List.length
```

Question à développer pendant l'oral 3 *Présenter votre algorithme et sa complexité en fonction de L et Q.*

D'après la question orale précédente, le nombre d'états accessibles correspond au nombre de nœuds accessibles depuis 0 dans le graphe GrapheAccessibilité($\mathcal{A}$). Nous faisons ici un parcours du graphe GrapheAccessibilité($\mathcal{A}$) (ici le parcours est en profondeur mais peu importe l'ordre de parcours) pour trouver le nombre de nœuds accessibles et le nombre de nœuds visités correspond au nombre d'états accessibles.
La complexité d'un parcours en profondeur est, dans le pire cas, linéaire en le nombre d'arcs (ici $L \times Q$) plus le nombre de nœuds (ici Q). La complexité totale est donc $O(L \times Q)$.

2 Compter les facteurs d'un mot acceptés par un automate

Étant donné $\mathcal{A}$ un automate et w un mot sur le même langage, on définit FacteursAcceptés($\mathcal{A}, w$) comme le nombre de facteurs non-vides de w acceptés par $\mathcal{A}$. Noter que l'on peut compter plusieurs fois un même mot qui apparaît plusieurs fois : un automate qui accepterait uniquement le mot "0" accepterait deux facteurs de "010" (parce que le facteur "0" y apparaît deux fois). Formellement, FacteursAcceptés($\mathcal{A}, w$) correspond au nombre de paires (i, j) avec $0 \leq i < j \leq |w|$ telles que $w[i : j]$ est un mot accepté par $\mathcal{A}$.
Dans cette section nous allons étudier deux algorithmes pour compter le nombre de facteurs d'un mot acceptés par un automate. Dans un premier temps, les automates considérés auront un grand nombre d'états tandis que dans un second temps nous regarderons le cas d'automates très petits.

2.1 Cas d'un automate avec beaucoup d'états

Question 6 *Calculer le nombre de préfixes non-vides d'un mot $\mathcal{M}(L, T)$ qui sont acceptés par un automate $\mathcal{E}(L, Q)$ pour les valeurs de L, Q et T suivantes :*

a) $L = 2$, $Q = 100$, $T = 100$

b) $L = 10$, $Q = 10\,000$, $T = 50\,000$

c) $L = 3$, $Q = 30\,000$, $T = 100\,000$.

```
let prefixesAcceptes (q,l,d,f) mot =
  (* la fonction compte renvoie le nombre d'états acceptants
  rencontrés dans le reste du mot à partir de l'état etat *)
  let rec compte etat = function
    | [] -> 0
    | c::q ->
       let netat = d etat c in
       (if f netat then 1 else 0)+compte netat q
  in
  compte 0 mot
```

Question 7 *En s'inspirant de l'algorithme développé à la question précédente, calculer* FacteursAcceptés($\mathcal{E}(L, Q), \mathcal{M}(L, T)$) *pour les valeurs de L, Q et T suivantes :*

a) $L = 2, Q = 100, T = 100$,

b) $L = 10, Q = 10\,000, T = 500$,

c) $L = 3, Q = 30\,000$, *et* $T = 3\,000$.

```
(* cette fonction prend un mot et somme prefixesAcceptes
     appelé sur chacun des suffixes  *)
  let rec facteursAcceptes auto = function
  | [] -> 0
  | c::q ->
     prefixesAcceptes auto (c::q) + facteursAcceptes auto q
```

Question à développer pendant l'oral 4 *Présenter votre algorithme et sa complexité en fonction de L, Q et T.*

Notre algorithme pour calculer le nombre de préfixes acceptés simule l'évaluation de l'automate lisant le mot et compte le nombre d'états acceptants par lesquels l'automate passe.
Pour calculer le nombre de facteurs acceptés d'un mot w, on somme le résultat de `prefixesAcceptes` pour chaque suffixe de w. Comme `prefixesAcceptes` tourne en $O(T)$ et qu'il y a $O(T)$ suffixes on obtient un algorithme en $O(T^2)$.

2.2 Cas d'un automate avec très peu d'états

Étant donné un automate $\mathcal{A} = (L, Q, \delta, \mathcal{F})$ et un mot $w = w_0 \dots, w_{T-1}$, on définit $\mathcal{G}(\mathcal{A}, w)$ comme le graphe orienté dont les nœuds sont $\bot$, $\top$ ainsi que les paires (q, i) avec $0 \leq q < Q$ et $0 < i \leq T$. Le graphe contient les arcs suivants :

— $(q, i) \to (q', i+1)$ pour $0 \leq q < Q$, $q' = \delta(q, w_i)$, et $0 < i < T$,
— $\top \to (\delta(0, w_i), i+1)$ pour chaque $0 \leq i < T$,
— $(q, i) \to \bot$ pour chaque q état final de $\mathcal{A}$ et $0 < i \leq T$.

Question à développer pendant l'oral 5 *Montrer que* FacteursAcceptés$(\mathcal{A}, w)$ *est égal au nombre de chemins différents de $\top$ à $\bot$ dans $\mathcal{G}(\mathcal{A}, w)$. Montrer que le nombre de chemins de $\top$ à $(q, i+1)$ dépend uniquement de δ, w_i et des nombres de chemins de $\top$ à (q', i) pour $0 \leq q' < Q$.*

Considérons la fonction qui associe à $w[i:j]$ le chemin $\top \to (\delta(0, w[i:i+1]), i+1) \to (\delta(0, w[i:i+2]), i+2) \to \dots \to (\delta(0, w[i:j]), j)$ et montrons que c'est une bijection entre d'un côté les chemins qui partent de $\top$ et ne passent pas par $\bot$ et de l'autre les paires $(i < j)$ correspondant à des facteurs $w[i:j]$ du mot.
Il est assez clair que des (i, j) différents seront envoyés sur des chemins différents mais inversement un chemin partant de $\top$ et ne passant pas par $\bot$ est forcément de la forme $\top \to (q_i, i) \to (q_{i+1}, i+1) \to (q_j, j)$ pour certains i, j et une certaine suite $q_i \dots q_j$. En effet, les seuls nœuds accessibles depuis

(q, i) sont des nœuds de la forme $(q', i+1)$. On en déduit aussi que $q_i = 0$ et que $q_{i+1} = \delta(q_i, w_i)$ et donc que $q_j = \delta(0, w[i:j])$ ce qui indique que le chemin arrive sur un nœud dont l'état est final si et seulement si $w[i:j]$ est accepté.
Maintenant il y a aussi une bijection entre les chemins de $\top$ à $\bot$ et les chemins de $\top$ à (q, j) avec q final. En combinant ces deux bijections on obtient une bijection entre les facteurs $w[i:j]$ qui appartiennent au langage et les chemins de $\top$ à $\bot$.
Les chemins de $\top$ à $(q, i+1)$ sont un chemin de longueur 1 quand $q = \delta(0, w_i)$ plus les chemins de $\top$ à (q', i) pour q' tel que $\delta(q', w_i) = q$.

Question 8 *Déterminer* FacteursAcceptés$(\mathcal{E}(L, Q), \mathcal{M}(L, T))$ *pour les valeurs de* L, Q *et* T *suivantes :*

a) $L = 2$, $Q = 10$, $T = 1\,000$

b) $L = 10$, $Q = 30$, $T = 5\,000$

c) $L = 3$, $Q = 17$, *et* $T = 50\,000$.

```
let facteursAcceptes2 (q,l,d,f) mot =
  let mottab = Array.of_list mot in
  let t = Array.length mottab in
  let nbChemins = Array.make_matrix (t+1) q 0 in
  let nbFacteurs = ref 0 in

  for pos = 0 to t-1 do
    nbChemins.(pos).(0) <-nbChemins.(pos).(0)+1 ;
    for e = 0 to q-1 do
      let n_etat =  d e mottab.(pos) in
      nbChemins.(pos+1).(n_etat) <-
      ↪  nbChemins.(pos+1).(n_etat)+nbChemins.(pos).(e) ;
      if f n_etat then nbFacteurs := !nbFacteurs + nbChemins.(pos).(e)
    done
  done ;
  !res
```

Question à développer pendant l'oral 6 *Présenter votre algorithme et sa complexité en fonction de* L, Q *et* T.

Notre algorithme remplit itérativement le contenu de `nbChemins` en utilisant la réponse à la question orale précédente. Notre algorithme fait donc deux boucles imbriquées, une tournant Q fois, une tournant T fois, la complexité globale est donc $O(Q \times T)$.

3 Compter avec mises à jour

Dans cette section, on cherche un algorithme capable de résoudre le problème suivant : le mot qui nous intéresse est régulièrement modifié et l'on veut savoir rapidement après chaque modification, si le mot est accepté ou le nombre de ses facteurs qui sont acceptés par un petit automate.
Une "mise à jour" d'un mot est une paire (p, c) indiquant qu'il faut remplacer la lettre à la position p par c. Ainsi après avoir appliqué la mise à jour (p, c) le mot $w_0 \dots w_{n-1}$ devient $w_0 \dots w_{p-1}\ c\ w_{p+1} \dots w_{n-1}$.
On définit $\mathcal{L}(i, L, T)$, la i-ème mise à jour sur un mot de taille T avec un alphabet de L lettres de la façon suivante. La position de la mise à jour $\mathcal{L}(i, L, T)$ est $u(T \times L + 2i + 1) \mod T$ tandis que la nouvelle lettre est $u(T \times L + 2i) \mod L$.
Étant donné un mot initial w (sur un alphabet à L lettres et de longueur T), l'application des mises à jour de $\mathcal{L}(0, L, T)$ puis $\mathcal{L}(1, L, T)$, etc. produit une séquence de mots. On note $\mathcal{U}(i, L, T)$ le résultat des i premières mises à jour sur $\mathcal{M}(L, T)$. On a donc $\mathcal{U}(0, L, T) = \mathcal{M}(L, T)$ puis $\mathcal{U}(i + 1, L, T)$ qui est le résultat de l'application de $\mathcal{L}(i, L, T)$ sur $\mathcal{U}(i, L, T)$.

Cas avec peu de mises à jour

Question 9 *En utilisant les réponses aux questions précédentes, calculer* $\sum_{0 \le i < N} \textsc{FacteursAcceptés}(\mathcal{E}(L, Q), \mathcal{U}(i, L, T))$ *pour les valeurs de* L, Q, T *et* N *suivantes :*

a) $L = 3$, $Q = 100$, $T = 100$, $N = 100$

b) $L = 5$, $Q = 30\,000$, $T = 100$, $N = 1\,000$

c) $L = 7$, $Q = 10$, $T = 3\,000$, $N = 100$

```
let facteursAcceptesOptim q l t mot  =
  if t < q (* Choisit le meilleur algorithme en fct des paramètres *)
  then facteursAcceptes (e q l) mot
  else facteursAcceptes2 (e q l) mot

let compteAvecMaj q l t n =
  let mot = Array.of_list (m l t) in
  let rec faireMaj i = (* compte les facteurs acceptés pour *)
    if i >= n (* les mises à jour après la i-ème*)
    then 0
    else
      begin
        (* compte pour le mot courant *)
        let sub = facteursAcceptesOptim q l t (Array.to_list mot) in
        (* fait la mise à jour *)
        mot.((u (t*l+2*i+1)) mod t) <- (u (t*l+2*i)) mod l ;
        (* somme avec la mise à jour *)
        sub + faireMaj (i+1)
      end
  in
  faireMaj 0
```

Question à développer pendant l'oral 7 *Expliquer l'algorithme utilisé et donner sa complexité en fonction de Q, L, T, et N.*

Les mises à jour vont être traitées en générant le mot initial puis en mettant à jour le mot. À chaque étape on calcule le nombre de facteurs acceptés.
La fonction `facteursAcceptesOptim` a la complexité $\min(T^2, Q \times T)$, la complexité totale est $N \times T \times \min(T, Q)$

Cas d'un petit automate

Étant donnés un automate $\mathcal{A}$ et un mot w, l'effet de w sur $\mathcal{A}$, noté $\textsc{Effet}(w, \mathcal{A})$, est une fonction de l'ensemble des états de $\mathcal{A}$ vers l'ensemble des états de $\mathcal{A}$ telle que chaque état q est envoyé sur q' si q' est l'état dans lequel $\mathcal{A}$ arrive en lisant w depuis l'état q.
Étant donnés un automate $\mathcal{A}$ et un mot w non vide, $\textsc{Arb}(\mathcal{A}, w)$ est un arbre binaire. Dans cet arbre, chaque nœud correspond à un facteur v du mot w et stocke l'effet de v sur $\mathcal{A}$. En particulier la racine de $\textsc{Arb}(\mathcal{A}, w)$ correspond à w et stocke l'effet de w sur $\mathcal{A}$. $\textsc{Arb}(\mathcal{A}, w)$ peut être construit récursivement de la façon suivante :

- si $|w| = 1$ alors $\textsc{Arb}(\mathcal{A}, w)$ est un nœud sans fils,
- sinon on calcule w_1 et w_2 avec $w = w_1 w_2$ et $|w_1| \leq |w_2| \leq |w_1| + 1$. $\textsc{Arb}(\mathcal{A}, w)$ est alors un nœud qui a pour fils gauche $\textsc{Arb}(\mathcal{A}, w_1)$ et pour fils droit $\textsc{Arb}(\mathcal{A}, w_2)$.

Dans le calcul de l'arbre, pour chaque nœud $\textsc{Arb}(\mathcal{A}, u)$, on stocke $\textsc{Effet}(u, \mathcal{A})$. On note $\textsc{Accepte}(\mathcal{A}, w)$ l'entier qui vaut 1 si $\mathcal{A}$ accepte w et 0 sinon.

Question 10 *Écrire un algorithme capable de calculer $\textsc{Arb}(\mathcal{A}, w)$ et capable de calculer efficacement $\textsc{Arb}(\mathcal{A}, w')$ à partir de $\textsc{Arb}(\mathcal{A}, w)$ où w' est une mise à jour de w. L'utiliser pour calculer $\sum_{0 \leq i < N} \textsc{Accepte}(\mathcal{E}(L, Q), \mathcal{U}(i, L, T))$ pour $T = 50\,000$, $N = 25\,000$, et les valeurs de L et Q suivantes :*

a) $L = 10$, $Q = 30$ ***b)*** $L = 12$, $Q = 24$ ***c)*** $L = 5$, $Q = 25$

La fonction `compte` présentée ci-dessous est une fonction générique qui va servir pour Q10 et Q11. Elle utilise un arbre binaire comme décrit dans l'énoncé pour créer l'arbre et ensuite elle utilise cet arbre pour faire des mises à jour efficacement.
Ces mises à jour dépendent de 3 fonctions selon les « valeurs » que l'on stocke dans les nœuds des arbres. Pour Q10, la valeur stockée dans un nœud sera l'effet du mot correspondant à un nœud (sous la forme d'un tableau contenant Q cases) mais pour Q11 ce sera un effet différent.
Ainsi pour la Q10, la valeur que l'on stockera ce sera l'effet et donc dans un nœud correspondant à un mot w ce sera la fonction $q \mapsto \delta(q, w)$:

- pour une feuille et donc une lettre c c'est simplement $q \mapsto \delta(q, c)$ et donc `initEffet`$(c) = q \mapsto \delta(q, c)$;

— pour un nœud interne correspondant à $w = w_1 w_2$ dont les sous-nœuds correspondent aux mots w_1 et w_2 et donc dont les valeurs sont $f_1 = q \mapsto \delta(q, w_1)$ et $f_2 = q \mapsto \delta(q, w_2)$ alors on va calculer $f_1 \circ f_2$ car $(f_1 \circ f_2)(q) = f_2(f_1(q)) = q \mapsto \delta(\delta(q, w_1), w_2) = q \mapsto \delta(q, w_1 w_2)$. Donc `combineEffet` est une composition de fonction.

Enfin, l'algorithme dépend d'une troisième fonction, la fonction `compteEffet` qui sert à maintenir un compteur C . Initialement $C = 0$ et avant chaque mise à jour sur l'arbre le compteur C est mis à jour à la valeur (C + `compteEffet racine`).

Pour la Q10, la valeur à la racine correspond à la fonction $q \mapsto \delta(q, w)$ où w est le mot entier. Ce qui nous intéresse c'est de savoir si le mot est accepté et donc `compteEffet` va valoir 1 si le mot est accepté c'est à dire si $\delta(0, w) \in \mathcal{F}$ et 0 sinon. Ainsi après avoir appliqué toutes les modifications notre compteur C nous donnera la réponse attendue, à savoir le nombre de mots intermédiaires acceptés.

```
type 'a arb =
   | Noeud of int*'a*'a arb*'a arb
   | Leaf of 'a

 let rec compte (initEffet,combineEffet,compteEffet) q l t n =

  let effet = function | Noeud(_,e,_,_) | Leaf e -> e in

  let rec maj pos nouvelleValeur = function
    | Noeud(milieu, _, gauche,droite) ->
       let nouvG,nouvD =
         if pos < milieu
         then (maj  pos nouvelleValeur gauche,droite)
         else (gauche,maj (pos-milieu) nouvelleValeur droite)
       in
       Noeud(milieu, combineEffet (effet nouvG) (effet nouvD), nouvG, nouvD)
    | Leaf _ -> Leaf nouvelleValeur
  in

  let auto = e q l in
  let mot = Array.of_list (m l t) in


  let rec createArbre pos longueur =
    if longueur = 1
    then  Leaf (initEffet auto mot.(pos))
    else
      let mil = (1+long)/2 in
      let arbreG = createArbre pos mil in
      let arbreD = createArbre (pos+mil) (long-mil) in
      let effetNoeud = combineEffet (effet arbreG) (effet arbreD) in
      Noeud(mil,effetNoeud, arbreG, arbreD)
  in

  let compteArbre arbre = compteEffet auto (effet arbre) in
  let rec faitLesMAJ i arbre =
    if i = n then 0
    else
      begin
        let pos = (u (t*l+2*i+1)) mod t
        and lettre = (u (t*l+2*i)) mod l in
        compteArbre arbre +
          faitLesMAJ (i+1) (maj  pos (initEffet auto lettre) arbre)
      end
  in
  faitLesMAJ 0 (createArbre 0 t)
```

```
let initES (q,l,d,f) c = Array.init q (fun etat -> d etat c)

let compteES (q,l,d,f) tr = if f(tr.(0)) then 1 else 0

let combineES g d = Array.map (fun v -> d.(v)) g
```

Question à développer pendant l'oral 8 *Expliquer l'algorithme utilisé et donner sa complexité en fonction de Q, L, T, et N.*

Voir ci-haut pour la présentation de l'algorithme utilisé. Pour la complexité, au moment de l'initialisation on fait T appels à `initEffet` et à `combineEffet` puis à chaque mise à jour on fait $\log_2(T)$ appels à `combineEffet` et un appel à `initEffet` ainsi qu'un appel à `compteEffet`.
Les fonctions `combineEffet` et `initEffet` prennent un temps Q tandis que `compteEffet` a un temps constant donc au total on a un temps $O((T + N \times \log_2(T)) \times Q)$.

Question à développer pendant l'oral 9 *Expliquer quelles informations (en plus de l'effet) doivent être stockées dans chaque nœud de* $\textsc{Arb}(\mathcal{A}, w)$ *pour pouvoir répondre à* $\textsc{FacteursAcceptés}(\mathcal{A}, w)$ *et supporter des mises à jour efficaces.*

Pour pouvoir répondre efficacement à $\textsc{FacteursAcceptés}(\mathcal{A}, w)$ il faut stocker dans chaque nœud correspondant au mot w et pour chaque état q :
- le nombre de suffixes de w qui emmènent l'automate dans l'état q,
- le nombre de préfixes de w qui partant de l'état q arrivent dans un état final,
- l'effet du mot w,
- le nombre de facteurs de w acceptés.

Calculer cela pour une lettre seule ne pose pas de vraie difficulté mais pour combiner deux sous-mots w_1 et w_2 il faut remarquer qu'un suffixe de w_1w_2 est soit un suffixe de w_2 soit un suffixe de w_1 concaténé avec w_2 à droite. Et donc les suffixes qui emmènent dans l'état q sont tous les suffixes de w_2 qui arrivent dans q plus tous les suffixes de w_1 qui arrivent dans un q' avec $v_2(q') = q$ pour v_2 l'effet de w_2.
On peut faire le même raisonnement pour les préfixes et on calcule l'effet comme dans la Q10.

Question 11 *En déduire un algorithme capable de calculer efficacement* $\left(\sum_{0\leq i<N} \textsc{FacteursAcceptés}(\mathcal{E}(L,Q),\mathcal{U}(i,L,T))\right) \mod 1\,000\,000$ *pour* $T = 5\,000$, $N = 20\,000$, *et les valeurs de L et Q suivantes :*

a) $L = 10$, $Q = 30$ ***b)*** $L = 12$, $Q = 24$ ***c)*** $L = 5$, $Q = 25$

```
let combineEC (geff,gdeb,gfin,gnb) (deff,ddeb,dfin,dnb) =
  let q = Array.length geff in
  let eff = Array.map (fun v -> deff.(v)) geff in
  let deb = Array.copy ddeb in
  let fin = Array.copy gfin in
  for i = 0 to q-1 do
    deb.(deff.(i)) <- deb.(deff.(i)) + gdeb.(i) ;
    fin.(i) <- fin.(i) + dfin.(geff.(i)) ;
    done ;
  let nb =  (list_init q (fun i->gdeb.(i)*dfin.(i)) |>
               List.fold_left (+) (gnb+dnb)) mod 1000000 in
  (eff,deb,fin,nb)

let initEC (q,l,d,f) c =

  let tr = Array.make q 0 in
  let deb = Array.make q 0 in
  let fin = Array.make q 0 in
  for q2 = 0 to q-1 do
    tr.(q2) <- d q2 c ;
      if f (d q2 c) then
        fin.(q2) <- 1 ;
    done ;
    deb.(d 0 c) <- 1;
  (tr,deb,fin,if f(d 0 c) then 1 else 0)

let compteEC auto (e,d,f,n) = n
```

Fiche réponse type : Comptage de mots reconnus par un automate.

$\widetilde{u_0}$: 42

Question 1

a) 986

b) 700

c) 631

Question 2

a) 113

b) 51133

c) 45445

Question 3

a) 2,2,1

b) 2,2,5,6

c) 2,6,3,0,8

Question 4

a) 12

b) 2222

c) 24124

Question 5

a) 80

b) 9999

c) 28250

Question 6

a) 56

b) 25692

c) 50493

Question 7

a) 2835

b) 65060

c) 2269352

Question 8

a) 448058

b) 5001486

c) 812170725

Question 9

a) 195813

b) 2483989

c) 246686965

Question 10

a) 18508

b) 9459

c) 12542

Question 11

a) 627390

b) 934507

c) 889835

Comment couper la poire en deux ?

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2021

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !

Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...

Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***.

Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

L'objectif général de ce sujet est, pour un graphe donné, de le partitionner en deux parties de poids équilibrés tout en modérant le nombre d'arêtes à cheval entre les deux parties.

1 Préliminaires

On rappelle que pour deux entiers naturels a et b, $a \bmod b$ désigne le reste de la division entière de a par b.
Dans tout le sujet les listes sont indexées à partir de 0.

1.1 Suite de nombres pseudo-aléatoires

On fixe $M = 2^{31} - 1 = 2\,147\,483\,647$.
On définit la suite $\overline{u}(n)$ par récurrence :

$$\overline{u}(0) = u_0, \quad \forall n \in \mathbb{N}, \overline{u}(n+1) = (16\,807 \times \overline{u}(n) + 17) \bmod M$$

u_0 vous ayant été donné, et doit être reporté sur votre fiche réponse.
On définit alors la suite

$$u(n) = \overline{u}(n \bmod 999\,983)$$

Question 1 *Calculer* $u(n) \bmod 997$ *pour*
a) $n = 16$, ***b)*** $n = 1024$, ***c)*** $n = 1\,000\,000$.

1.2 Somme de contrôle de listes de nombres

On définit l'opérateur de somme de contrôle :

$$S(x, y) = (2 \times x + y) \bmod 997$$

On définit la *somme de contrôle* d'une liste de nombres par récurrence :

$$C([]) = 0, \quad \forall n \in \mathbb{N}\ C([x_1, x_2, ..., x_{n+1}]) = S(C([x_1, x_2, ..., x_n]), x_{n+1})$$

Question 2 *Calculer*
a) $S(u(16), u(17))$, ***b)*** $C([u(100), u(101), ..., u(1\,000)])$,
c) $C([u(999\,900), u(999\,901), ..., u(1\,000\,000)])$.

Question à développer pendant l'oral 1 *Expliquez votre implémentation et sa complexité.*

On peut utiliser une version itérative sur la liste, de gauche à droite, ou bien une version récursive, mais il vaut mieux éviter d'obtenir une complexité en $O(n^2)$.

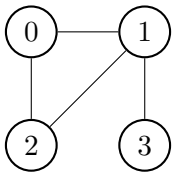
1.3 Génération de graphe aléatoire

On souhaite ici générer des graphes pseudo-aléatoirement. Tous les graphes manipulés seront simples, i.e. sans arête d'un sommet à lui-même (boucle) ni arêtes multiples pour une même paire de sommets.
Pour un n et un a donné, on note G_n^a le graphe non orienté ayant n sommets $\{s_0, s_1, ..., s_{n-1}\}$ et dont l'ensemble des arêtes est

$$\{\ (s_{(u(a+2i)) \bmod n}, s_{(u(a+2i+1)) \bmod n})\ |\ i \in [\![0, n \cdot \lfloor\sqrt{n}\rfloor - 1]\!] \text{ et } (u(a+2i)) \bmod n \neq (u(a+2i+1)) \bmod n\ \}$$

Il n'y a donc pas de boucle, et les doublons étant ignorés, il n'y a pas d'arêtes multiples non plus.

Par exemple, pour $\widetilde{u_0}$, $\widetilde{G_4^{16}}$ a donc pour ensemble d'arêtes $\{(0,2),(1,0),(1,3),(2,1)\}$, car la boucle $(0,0)$ est exclue, et l'arête $(0,1)$ est confondue avec l'arête $(1,0)$. $\widetilde{G_4^{16}}$ est alors ainsi :



Question 3 *Calculer*

a) *pour* G_4^{16}, $C([deg(s_0), deg(s_1), ..., deg(s_3)])$,

b) *pour* G_{1000}^{16}, $C([deg(s_0), deg(s_1), ..., deg(s_{999})])$,

c) *pour* $G_{10\,000}^{16}$, $C([deg(s_0), deg(s_1), ..., deg(s_{9\,999})])$

Il suffit de partir du graphe n'ayant que les sommets et aucune arête, et ajouter les arêtes en itérant sur i.

Question à développer pendant l'oral 2 *Discutez le choix de structure de données pour représenter ces graphes en mémoire.*

Avec $n \cdot \sqrt{n}$ arêtes, il vaut quand même mieux utiliser des listes d'adjacences plutôt qu'une matrice d'adjacence, pour obtenir un stockage en $n \cdot \sqrt{n}$ et des parcours sur les arêtes en $n \cdot \sqrt{n}$ aussi, plutôt que n^2.

Note : pour le u_0 et le $\widetilde{u_0}$ fournis, tous les graphes G_n^a étudiés sont connexes, on pourra donc utiliser cette propriété pour simplifier les algorithmes.

2 Partitionnement : Cas général

Dans cette partie, on suppose que le nombre de sommets du graphe manipulé est pair. On souhaite alors partitionner ce graphe en deux parties A et B ayant exactement le même nombre de sommets. On appelle coupe l'ensemble des arêtes dont les extrêmités sont l'une dans A, l'autre dans B. On cherche à minimiser le nombre d'arêtes de la coupe, appelé poids de la coupe.

2.1 Heuristique itérative

On utilise d'abord une approche heuristique qui raffine progressivement la coupe obtenue.

- On part du partitionnement $A = \{s_0, s_1, ..., s_{\frac{n}{2}-1}\}$ $B = \{s_{\frac{n}{2}}, s_{\frac{n}{2}+1}, ..., s_{n-1}\}$
- À chaque étape, on cherche une paire de sommets $(s_i, s_j) \in A \times B$, telle qu'échanger les deux sommets s_i et s_j entre A et B diminue le plus possible le poids de la coupe (la différence étant appelée gain). En cas d'égalité, on minimise i, puis j.
- On répète la deuxième étape tant que l'on parvient à diminuer le poids de la coupe.

Question 4 *Calculer le poids de la coupe obtenu par cette heuristique pour les graphes*
a) G_8^{16} ***b)*** G_{100}^{16} ***c)*** G_{300}^{16}

Question à développer pendant l'oral 3 *Expliquez l'algorithme utilisé pour implémenter cette heuristique, les structures de données qu'il utilise, et donnez la complexité de chaque étape.*

Une boucle principale infinie itère sur le nombre d'étapes. On utilise (par exemple) un tableau de booléens pour savoir quels sommets sont dans B, les autres étant dans A. Pour éviter de recalculer le poids de la coupe à chaque fois, on peut se souvenir du poids actuel, et le mettre à jour à chaque étape avec l'échange de sommets. On commence par calculer les gains de passage d'un sommet de A à B ou inversement. On utilise alors deux boucles `for` imbriquées pour les combiner et trouver le meilleur choix de paire. Si l'on a utilisé une matrice d'adjacence, savoir s'ils sont voisins coûte $O(1)$, sinon $O(\sqrt{n})$. Chaque étape nécessite ainsi $O(n^2)$ ou $O(n^2 \cdot \sqrt{n})$ selon leur représentation.

L'heuristique étant coûteuse, on essaie d'en réduire le coût. On choisit d'abord i tel que passer s_i de A à B produit une coupe de poids minimum. On choisit ensuite j tel que échanger s_i et s_j entre A et B produit une coupe de poids minimum.

Question 5 *Calculer le poids de la coupe obtenu par cette heuristique pour les graphes*
a) G_8^{16} ***b)*** G_{100}^{16} ***c)*** G_{300}^{16} ***d)*** G_{1000}^{16}

Question à développer pendant l'oral 4 *Donnez la nouvelle complexité de chaque étape.*

Le principe est le même mais l'on ne combine plus les deux boucles, ce qui pourrait faire penser (dans le cas matrice d'adjacence) que l'on passe chaque étape à $O(n)$. Mais il faut tout de même calculer les gains, ce qui coûte $O(n^2)$. On peut éventuellement stocker les gains et les mettre à jour à chaque étape, pour obtenir vraiment $O(n)$ pour chaque étape. Avec une liste d'adjacence, on reste de toute façon à $O(n \cdot \sqrt{n})$.

Question à développer pendant l'oral 5 *Il serait naturel de s'attendre à ce que le poids obtenu par cette seconde heuristique soit en général plus grand que celui obtenu par la première : pouvez-vous justifier cette intuition ? De fait, on constate que le poids n'est pas plus grand dans le cas des graphes générés par* $\widetilde{u_0}$ *notamment : comment peut-on expliquer ce phénomène ? (il est même souvent moindre : on ne cherchera pas à expliquer pourquoi)*

Quand on choisit séparément le sommet de A et le sommet de B, la combinaison des deux choisis indépendemment pourrait être moins intéressante que de choisir ensemble. Il y a un cas particulier, c'est quand $s_i \in A$ et $s_j \in B$ choisis sont voisins, on pense à tort que passer s_i dans B économise l'arête de s_i à s_j, mais puisqu'inversement on passe s_j dans A, on la re-perd, et donc l'estimation est fausse de 1 arête pour le choix de s_i et de même pour s_j. Si on ne se contentait pas de compter les arêtes mais qu'on prenait en compte un poids d'arête, cela pourrait être arbitrairement significatif. Ici, c'est toujours 1, et donc en général très faible par rapport aux gains obtenus par l'échange.

On revient sur la première approche (qui en général obtient effectivement un poids moindre), et l'on essaie de réduire son coût. Avant de choisir la paire (i, j), on élimine les candidats inintéressants de A et de B. On calcule d'abord séparément pour les sommets de A le meilleur gain que l'on peut obtenir en passant un sommet dans B, et pour les sommets de B le meilleur gain que l'on peut obtenir en passant un sommet dans A. On ne conserve alors comme candidats parmi A que les sommets dont le gain est au moins aussi grand que le meilleur gain que l'on peut obtenir en passant un sommet de A dans B, moins 2. De même, on ne conserve comme candidats parmi B que les sommets dont le gain est au moins aussi grand que le meilleur gain que l'on peut obtenir en passant un sommet de B dans A, moins 2. On peut alors chercher une paire (i, j) parmi ces candidats seulement. En cas d'égalité, on minimise i, puis j.

Question à développer pendant l'oral 6 *Pourquoi peut-on se contenter de ces candidats, et pourquoi « moins 2 » ?*

On discutait plus haut que l'erreur d'estimation commise est au plus 1 des deux côtés, et donc choisir un sommet n'apportant qu'un gain moindre ne peut pas être intéressant. Au pire on se trompe de 1 pour l'un et pour l'autre, donc 2 en tout.

Question 6 *Vérifiez que cette heuristique obtient les mêmes poids de coupe que l'heuristique de la question 4.*
Calculer le poids de la coupe obtenu par cette heuristique pour les graphes
a) G^{16}_{500} ***b)*** G^{16}_{1000} ***c)*** G^{16}_{2000} *(ce dernier peut prendre plusieurs secondes à calculer)*

2.2 Algorithme exact

On cherche désormais à obtenir une coupe de poids minimal. On note $min(n, a)$ le poids minimal de la coupe du graphe G^a_n en deux parties.

Question 7 *Calculer*

a) $min(8, 16)$,

b) $C([min(16, 0), min(16, 1), ..., min(16, 9)])$,

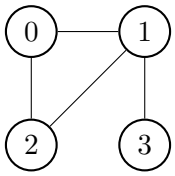
c) $min(24, 16)$

Question à développer pendant l'oral 7 *Expliquez l'algorithme utilisé et les structures de données qu'il utilise, estimez sa complexité.*

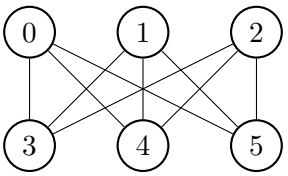
On peut utiliser une approche très simpliste qui énumère tous les entiers de 0 à $2^n - 1$ et utiliser leur représentation binaire pour énumérer toutes les partitions possibles (y compris de taille inégale), et l'on recalcule à chaque fois le poids de la coupe et conserve le minimum. Chaque calcul de poids de coupe est faisable en $O(n \cdot \sqrt{n})$, on obtient donc $O(2^n \cdot n \cdot \sqrt{n})$, ce qui est horriblement coûteux.
On peut éviter de recalculer à chaque fois le poids, en partant de la situation où tous les sommets sont dans A et aucun dans B, donc coût nul. On passe les sommets entre A et B en mettant à jour le poids. À chaque étape on doit parcourir les $\simeq \sqrt{n}$ voisins des sommets échangés. En moyenne on échange un nombre constant de sommets à chaque étape, et il y a 2^n étapes à énumérer, donc $O(2^n \cdot \sqrt{n})$ en tout.
Le problème principal est d'énumérer toutes les partitions possibles alors que seules celles pour lesquelles $|A| = |B|$ sont utiles. Pour cela on peut utiliser une approche récursive qui évite de parcourir tous les 2^n cas, en excluant divers choix, notamment il faudra de toute façon remplir à la fois A et B avec $n/2$ sommets. Ainsi au final on énumère les choix de $n/2$ sommets parmi n, qui se trouve constituer $O((2e)^{n/2})$ étapes. À chaque étape la mise à jour coûte $\simeq \sqrt{n}$.

3 Cas des graphes planaires

Un graphe planaire est un graphe tel que l'on peut le dessiner dans le plan sans que des arêtes se croisent. Par exemple, $\widetilde{G_4^{16}}$ vu précédemment est planaire :



Alors que ce graphe-ci :



est bien connu [1] pour n'être pas planaire : on ne peut le dessiner dans le plan sans qu'au moins deux arêtes se croisent.

3.1 Génération de graphes planaires

On souhaite générer des graphes qui sont planaires par construction. Pour cela on garde note, au cours de la génération, d'un ensemble de sommets appelé contour, autour duquel on ajoute les nouveaux sommets.

On construit le graphe P_n^a à n sommets $\{s_0, s_1, ..., s_{n-1}\}$ progressivement ainsi :
- Initialement, le graphe ne contient que trois sommets s_0, s_1, s_2, les arêtes (s_0, s_1), (s_0, s_2), (s_1, s_2), et le contour est composé des trois sommets $C_0 = \{s_0, s_1, s_2\}$

1. Il s'agit de l'énigme des trois maisons.

— À l'étape i ($i \geq 0$), on considère le sommet du contour C_i d'indice minimal, qui est s_i. On considère également

$$d_i = \begin{cases} 5 \text{ quand } u(a+i) \bmod 100 \leq 30 \\ 7 \text{ quand } u(a+i) \bmod 100 = 31 \\ 6 \text{ quand } 31 < u(a+i) \bmod 100 \end{cases}$$

On pose alors

$$d'_i = min\left(max(d_i - deg(s_i), 1), n - n_i\right)$$

qui est le nombre de sommets que l'on va ajouter au graphe, où n_i est le nombre de sommets actuels (qui est égal à $i + |C_i|$). On s'assure ainsi d'ajouter au moins un sommet, tout en s'assurant qu'on ne dépasse jamais le nombre n de sommets ciblé.

— On ajoute donc d'_i sommets $\{s_{n_i}, s_{n_i+1}, ..., s_{n_i+d'_i-1}\}$.

— On ajoute également les arêtes allant du sommet s_i à ces sommets : (s_i, s_{n_i}), (s_i, s_{n_i+1}), ... $(s_i, s_{n_i+d'_i-1})$

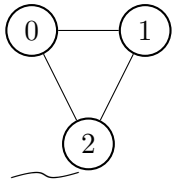
— On considère également les deux sommets s_x et s_y ($x < y$) qui sont voisins de s_i dans le graphe et appartiennent au contour C_i (par construction, s_i a toujours exactement deux voisins dans le contour). On ajoute les arêtes pour former un chemin de s_x à s_y passant par les nouveaux sommets : (s_x, s_{n_i}), (s_{n_i}, s_{n_i+1}), ..., $(s_{n_i+d'_i-1}, s_y)$.

— Enfin, on retire le sommet s_i du contour, et l'on y ajoute les nouveaux sommets à la place : $C_{i+1} = (C_i \setminus \{s_i\}) \cup \{s_{n_i}, s_{n_i+1}, ..., s_{n_i+d'_i-1}\}$

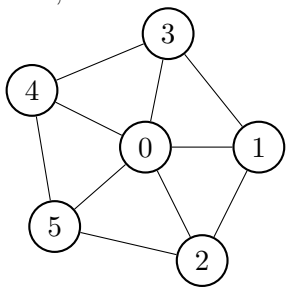
— Si le nombre de sommets a atteint n, l'algorithme est terminé, sinon on passe à l'étape $i+1$.

Par exemple, pour pour $u_0 = \widetilde{u_0}$, la construction de $\widetilde{P_8^{32}}$ s'effectue ainsi :

— Initialement on a

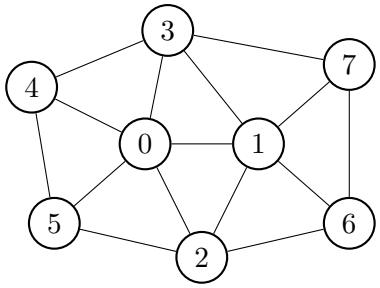


— On effectue l'étape $i = 0$. Puisque $\widetilde{u(32+0)} \bmod 100 = 14$, $d_0 = 5$. s_0 étant de degré 2 pour l'instant, on obtient $d'_0 = 3$. Les voisins de s_0 (dans le graphe) qui sont dans le contour $C_0 = \{s_0, s_1, s_2\}$ sont (dans l'ordre) s_1 et s_2. On ajoute ainsi trois sommets s_3, s_4 et s_5 et les arêtes correspondantes, obtenant :



Et le nouveau contour est $C_1 = \{s_1, s_2, s_3, s_4, s_5\}$.

— On effectue l'étape $i = 1$. Puisque $\widetilde{u(32+1)} \bmod 100 = 44$, $d_1 = 6$. s_1 est de degré 3 pour l'instant, mais l'on a déjà $n_1 = 6$ sommets or on cible n sommets, on obtient donc $d'_1 = 2$. Les voisins de s_1 (dans le graphe) qui sont dans le contour C_1 sont (dans l'ordre) s_2 et s_3. On ajoute ainsi les deux derniers sommets s_6 et s_7 et les arêtes correspondantes, obtenant :



Et le dernier contour est $C_2 = \{s_2, s_3, s_4, s_5, s_6, s_7\}$.

Question à développer pendant l'oral 8 *Quel est à peu près le degré moyen des sommets des graphes obtenus ? Quel type de structure de données vaut-il mieux utiliser pour représenter ces graphes en mémoire ?*

Le degré visé est en général 6, parfois 5 et très rarement 7. Au final le degré moyen est a priori presque 6. Il vaut donc clairement mieux utiliser des listes d'adjacences plutôt qu'une matrice d'adjacence.

Question 8 *Calculer*

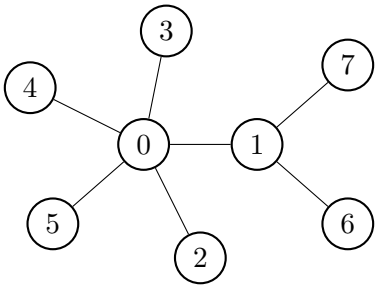
a) *pour* P_{16}^{32}, $C([deg(s_0), deg(s_1), ..., deg(s_{15})])$,

b) *pour* P_{1000}^{32}, $C([deg(s_0), deg(s_1), ..., deg(s_{999})])$,

c) *pour* $P_{50\,000}^{32}$, $C([deg(s_0), deg(s_1), ..., deg(s_{49\,999})])$

3.2 Arbre couvrant

Pour les graphes planaires générés ci-dessus, on souhaite extraire un arbre couvrant, c'est-à-dire un graphe qui comprend les mêmes sommets, mais un sous-ensemble des arêtes de façon à produire un arbre. Pour cela, on effectue un parcours en largeur du graphe, et l'on ne conserve dans l'arbre couvrant que les arêtes utilisées lors du parcours en largeur. On parcourt à chaque fois les sommets et les voisins par ordre croissant d'indice dans le graphe.
On note $T_n^a(i)$ l'arbre couvrant ainsi obtenu pour le graphe P_n^a en partant du sommet s_i.
Par exemple, pour $u_0 = \widetilde{u_0}$, pour le graphe planaire $\widetilde{P_8^{32}}$, en partant du sommet s_0, on parcourt ses voisins et ajoute ainsi à l'arbre couvrant dans l'ordre les arêtes (s_0, s_1), (s_0, s_2), (s_0, s_3), (s_0, s_4), et (s_0, s_5). On repart alors successivement de ces différents sommets par ordre croissant d'indice, d'abord avec s_1, pour lequel on ajoute les arêtes (s_1, s_6) et (s_1, s_7). Puis pour s_2, s_3, s_4, s_5 il n'y a pas d'arête à ajouter. On repart alors successivement des nouveaux sommets s_6 et s_7, pour lesquels il n'y a pas d'arête à ajouter. On obtient ainsi l'arbre couvrant $\widetilde{T_8^{32}(0)}$:



Question 9 *Calculer*

a) *dans* $T_{16}^{32}(0)$, $C([deg(s_0), deg(s_1), ..., deg(s_{15})])$,

b) *dans* $T_{1000}^{32}(0)$, $C([deg(s_0), deg(s_1), ..., deg(s_{999})])$,

c) *dans* $T_{50\,000}^{32}(0)$, $C([deg(s_0), deg(s_1), ..., deg(s_{49\,999})])$

3.3 Partitionnement de graphes planaires

Dans le cas des graphes planaires, on change légèrement le problème de partitionnement : pour un graphe G ayant n sommets on cherche un sous-ensemble S des sommets (appelé séparateur) tel que si l'on retire ces sommets du graphe G et les arêtes correspondantes, on obtient deux ensembles de sommets A et B pour lesquels il n'y a pas dans G d'arête dont une extrémité est dans A et l'autre dans B. S « sépare » donc effectivement le graphe en deux parties. On cherche à réduire le nombre de sommets du séparateur, $|S|$, tout en gardant A et B de tailles similaires.

Question à développer pendant l'oral 9 *En pratique, on constate que l'on parvient en général à obtenir* $|S| = O(\sqrt{n})$*, pourquoi ?*

Si l'on représente le graphe dans le plan (puisqu'il est planaire !), intuitivement on le coupe facilement en deux en le tranchant en deux morceaux. La surface utilisée par la représentation du graphe est de l'ordre de n, et la ligne utilisée pour le trancher en deux est donc de l'ordre de $\sqrt{n}$.

Nos graphes planaires étant très réguliers, nous allons pouvoir utiliser une heuristique un peu simpliste pour déterminer S, mais qui se révèlera obtenir de bons résultats.

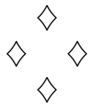
- On commence par calculer un arbre couvrant tel que réalisé à la section 3.2, en partant du sommet 0.
- On calcule également les profondeurs des différents sommets dans cet arbre, on note $maxp$ la profondeur maximum obtenue.
- On note H l'ensemble des sommets de profondeur au moins $maxp - 1$.
- On pioche s_i le sommet de H avec i maximal.
- On choisit $s_j \in H$ tel que le chemin de s_i à s_j dans l'arbre couvrant forme un séparateur S pour lequel $||A| - |B||$ est minimal, en choisissant j minimal en cas d'égalité.

Note : même si ce n'est pas vrai dans le cas général, dans le cas des graphes planaires générés mentionnés dans cet exercice, il se trouve que cette heuristique les partitionne toujours en au plus 2 composantes connexes.
On note S_n^a le séparateur obtenu ainsi pour P_n^a.
Par exemple, pour $u_0 = \widetilde{u_0}$, dans le cas du graphe $\widetilde{P_8^{32}}$, H comprend tous les sommets sauf s_0, on part du sommet s_7, et l'on obtient $\widetilde{S_8^{32}} = \{s_0, s_1, s_5, s_7\}$ pour lequel $||A| - |B|| = |2 - 2| = 0$

Question 10 *On ordonne les sommets de* S_n^a *par indice croissant. Calculez*
a) $C(S_{16}^{32})$ ***b)*** $C(S_{1000}^{32})$ ***c)*** $C(S_{10\,000}^{32})$

Question à développer pendant l'oral 10 *Expliquez votre implémentation et estimez sa complexité.*

Le calcul d'arbre couvrant est un parcours en largeur classique, il faut simplement se souvenir des arêtes parcourues.
On peut calculer les profondeurs en même temps, ou bien dans un second parcours en largeur.
Le chemin de s_i à s_j dans l'arbre couvrant peut se faire avec un parcours simpliste, qui se trouvera en $O(n)$ de toute façon puisque c'est un arbre. On peut alors calculer $|A|$ avec un calcul de connexité classique, en $O(n)$ aussi. s_i étant fixé, on ne parcourt que pour s_j, et il ne parcourt que le contour du graphe, donc de l'ordre de $\sqrt{n}$ sommets. On a ainsi une complexité estimable à $n \cdot \sqrt{n}$.



Fiche réponse type : Comment couper la poire en deux ?

$\widetilde{u_0}$: 42

Question 1

a) 781

b) 374

c) 805

Question 2

a) 373

b) 167

c) 943

Question 3

a) 33

b) 233

c) 506

Question 4

a) 2

b) 341

c) 1915

Question 5

a) 2

b) 327

c) 1900

d) 12511

Question 6

a) 4314

b) 12572

c) 36952

Question 7

a) 2

b) 386

c) 20

Question 8

a) 870

b) 348

c) 582

Question 9

a) 687

b) 235

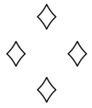
c) 377

Question 10

a) 111

b) 375

c) 923



Jeux sur des arbres (*tries on tries*).

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2021

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !

Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...

Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***. Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

1 Préliminaires

Étant donné un corpus, c'est-à-dire une liste de mots valides, on considère un jeu à deux joueurs où, à tour de rôle, chaque joueur énonce une lettre. À eux deux, ils constituent ainsi un mot commun en ajoutant la lettre qu'ils jouent à la fin du mot en cours de construction. Le mot que les deux joueurs construisent doit toujours être un préfixe d'un des mots du corpus. Toutefois, le premier joueur qui forme un mot du corpus en ajoutant sa lettre a perdu. C'est le joueur 1 qui commence. Par exemple, supposons que le corpus est réduit à un seul mot, {ARBORE}. Une partie peut alors se dérouler comme suit : le joueur 1 dit la lettre A, le joueur 2 la lettre R, puis ils jouent alternativement B, O, R, et E. À ce point, le joueur 2 perd à l'instant où il joue la lettre E, puisqu'il a formé le mot ARBORE. Si par contre le corpus contenait également le mot A, alors le joueur 1 perdrait dès son premier coup en prononçant la lettre A.

Notations

On rappelle que pour deux entiers naturels a et b, $a \mod b$ désigne le reste de la division entière de a par b, c'est à dire l'unique entier r avec $0 \leq r < b$ tel que $a = k \times b + r$ pour $k \in \mathbb{N}$.

Rappels sur les arbres lexicographiques et les jeux

Un mot w de taille n sur un alphabet Σ est constitué de n lettres $w[0] \cdots w[n-1]$, toutes dans Σ. ε désigne le mot vide. $p \subseteq w$ indique que p est un préfixe de w. Si de plus $p \neq w$ on note $p \subset w$ et on dit que p est un préfixe strict de w.

Un arbre lexicographique, ou arbre de préfixes, est un arbre de recherche qui permet de représenter un ensemble de mots, et supporte des opérations de recherche et d'insertion. On décrit un tel arbre par le quintuplet $A = (\Sigma, V, \alpha, \delta, T)$ où :

- Σ est un alphabet, ici représenté par les entiers de 0 à $M-1$;
- V représente les nœuds de l'arbre, ici des entiers dont 0 est la racine ;
- $\alpha : V \to \mathcal{P}(\Sigma)$ représente la fonction de voisinage et $\delta : V \times \Sigma^* \to V$ la fonction de transition dans l'arbre : s'il y a une arête étiquetée $a \in \Sigma$ du nœud u au nœud v, alors $a \in \alpha(u)$ et $v = \delta(u, a)$. On étend la fonction de transition δ aux mots de la façon suivante : $\delta(u, \varepsilon) = u$ et $\delta(u, ww') = \delta(\delta(u, w), w')$ où ww' désigne la concaténation des mots w et w'.
- $T \subseteq V$ représente les nœuds dits terminaux de l'arbre.

On peut associer un mot à chaque nœud de l'arbre : la racine de l'arbre correspond au mot vide ε, et on obtient le mot d'un nœud en lisant les lettres sur la branche reliant la racine à ce nœud, cf. Figure 1. L'ensemble des mots de l'arbre correspond aux mots des nœuds terminaux. Et les mots des nœuds de l'arbre correspondent aux préfixes des mots de l'arbre. Pour savoir si un mot w est dans l'arbre, on part de la racine et on emprunte les arêtes étiquetées par les lettres de w si elles existent. S'il ne manque aucune arête et qu'on aboutit à un nœud $v \in T$, alors le mot est dans l'arbre. Dans le cas contraire, le mot n'est pas dans l'arbre.

Un jeu à deux joueurs sur un arbre $A = (\Sigma, V, \alpha, \delta, T)$ se déroule de la façon suivante :

- les nœuds V sont les configurations possibles du jeu ; la racine est contrôlée par le joueur 1, et une arête relie toujours deux nœuds contrôlés par des joueurs différents ;
- les arêtes représentent les coups possibles depuis le nœud u : si l'on joue la lettre $a \in \alpha(u)$ depuis le nœud u, on emprunte l'arête et on atterrit dans le nœud $v = \delta(u, a)$;

— le jeu s'arrête lorsqu'on atteint un nœud de T, dans ce cas l'un des joueurs gagne.

Générateur de nombres pseudo-aléatoires

Étant donné u_0 on définit la récurrence suivante :

$$u(0) = u_0$$
$$\forall t \in \mathbb{N},\ u(t+1) = (28 \times u(t)) \mod 336\,529$$

L'entier u_0 vous est donné, et doit être recopié sur votre fiche réponse avec vos résultats. Une fiche réponse type vous est donnée en exemple, et contient tous les résultats attendus pour une valeur de u_0 différente de la vôtre (notée $\widetilde{u_0}$). Il vous est conseillé de tester vos algorithmes avec cet $\widetilde{u_0}$. Pour chaque calcul demandé, avec le bon choix d'algorithme le calcul ne devrait demander qu'au plus quelques secondes, jamais plus d'une minute.

Question 1 *Calculer les valeurs suivantes :*

a) $u(1) \mod 1\,000$, ***b)*** $u(42) \mod 1\,000$, ***c)*** $u(10^5) \mod 1\,000$.

Génération de mots

On définit le corpus $L(N)$ par la liste des mots w_i pour $i = 0, \ldots, N-1$ comme suit : w_i est de taille T_i et sa t-ième lettre est $w_i[t]$ pour $t = 0, \ldots, T_i - 1$ où :

$$T_i = T_{\min} + (u(i) \mod (T_{\max} - T_{\min} + 1)) \quad w_i[t] = u(\lfloor i/7 \rfloor + t) \mod M$$

Les mots ont au moins $T_{\min} = 4$ lettres, au plus $T_{\max} = 24$ lettres et l'alphabet comporte $M = 17$ lettres de 0 à 16. En particulier, $\varepsilon \notin L(N)$.

Question 2 *Afficher les 4 dernières lettres du dernier mot w_{N-1} du corpus $L(N)$ pour les valeurs de N suivantes :*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

On note $\textsc{NbOcc}(a, L)$ le nombre d'occurrences d'une lettre $a \in \Sigma$ dans le corpus L, dont on se servira dans la dernière section 3. Il se peut qu'un même mot apparaisse plusieurs fois dans le corpus, auquel cas les lettres d'un mot apparaissant plusieurs fois doivent être comptées plusieurs fois.

Question 3 *Calculer $\textsc{NbOcc}(w_{N-1}[T_{N-1}-1], L(N))$, le nombre d'occurrences de la dernière lettre du dernier mot du corpus $L(N)$, pour les valeurs de N suivantes :*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

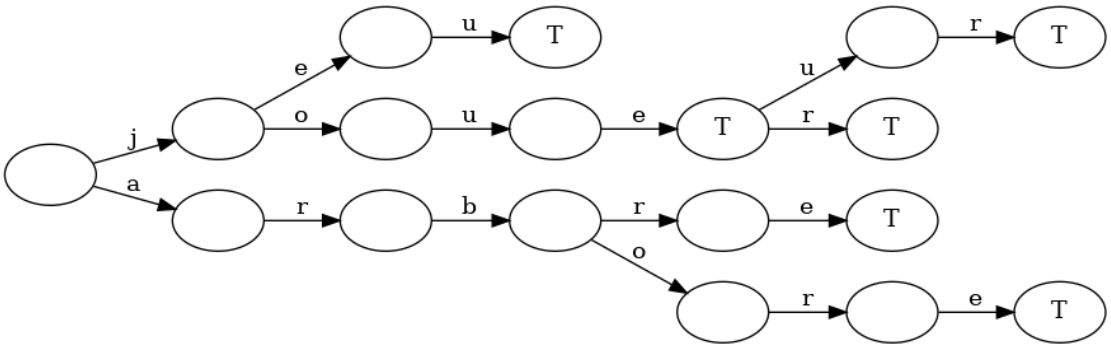


FIGURE 1 – Un arbre lexicographique pour le corpus {arbore, arbre, jeu, joue, jouer, joueur}. La racine est le nœud tout à gauche et les nœuds terminaux sont indiqués par T.

Construction de l'arbre lexicographique

On construit un arbre lexicographique associé à un corpus en partant d'un arbre vide et en insérant successivement les mots du corpus dans l'arbre, comme représenté sur la Figure 1. On note $A(N)$ l'arbre lexicographique associé au corpus $L(N)$.

Question 4 *Pour les valeurs de N suivantes, déterminer le nombre de nœuds terminaux présents dans $A(N)$.*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

Question à développer pendant l'oral 1 *Décrire la structure de donnée utilisée pour représenter un arbre lexicographique et la complexité de l'algorithme utilisé pour le construire, en fonction de N, $T_{\max}$ et M. Quel est l'intérêt d'une telle structure de données, et quelle autre structure pourrait-on utiliser pour stocker le corpus ? À quoi correspond le nombre de nœuds terminaux ?*

On peut avoir une structure récursive : un arbre dont chaque nœud contient un booléen qui indique s'il est terminal ou pas, et dont les enfants sont représentés par une liste de paires (lettre, arbre) (I). Les enfants peuvent aussi être représentés par un tableau d'arbres, comme le degré d'un nœud est borné par $M = 17$ (II). Une autre solution est d'implémenter l'arbre lexicographique comme un graphe par liste d'adjacence : les voisins d'un nœud sont encodés par des paires lettre et identifiant du nœud d'arrivée (III). En Python on peut utiliser un dictionnaire (table de hachage) pour accéder à un voisin en $O(1)$ plutôt que $O(M)$: les clés sont les lettres et les valeurs les identifiants de nœuds (IV). Ajouter un mot w a une complexité $O(|w| \times M)$ dans les cas (I-III) et $O(|w|)$ dans le cas (IV). Ainsi la complexité totale est $O(NT_{\max}M)$ ou $(NT_{\max})$. Par ailleurs il s'agit d'un arbre M-aire enraciné et tous les nœuds ont exactement un père sauf la racine, donc il y a autant de nœuds que d'arêtes plus 1. Il est également possible de représenter le corpus par un automate plutôt qu'un arbre lexicographique, ce qui peut réduire encore le nombre de nœuds. Dans ce cas on choisirait une

implémentation comme (III) ou (IV). Enfin, le nombre de nœuds terminaux de $A(N)$ correspond au nombre de mots distincts du corpus $L(N)$.

2 Stratégie gagnante contre un adversaire omniscient

Pour commencer, on suppose dans cette section que vous et votre adversaire connaissez tous les mots du corpus, que c'est vous qui commencez (vous êtes le joueur 1) et que l'adversaire a la meilleure stratégie possible.
On note MotsInaccessibles(N) la liste des mots distincts du corpus $L(N)$ qu'on ne pourra jamais jouer, car le jeu s'interrompt forcément avant d'avoir pu les prononcer.

Question 5 *Compter la taille de* MotsInaccessibles(N) *pour les valeurs de N suivantes :*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

Question à développer pendant l'oral 2 *Présenter votre algorithme et sa complexité en fonction de N, $T_{\max}$ et M.*

Un parcours en profondeur depuis la racine qui s'arrête aux nœuds terminaux suffit pour compter les nœuds accessibles en $O(NT_{\max})$ car le nombre d'arêtes est presque égal au nombre de nœuds. Le nombre de nœuds terminaux total est le nombre de mots distincts du corpus, ainsi on peut en déduire le nombre de mots inaccessibles.

Dans l'arbre, un nœud est dit gagnant si et seulement si le joueur 1 (vous, dans cette section) a une stratégie pour gagner à partir de ce nœud, quoi que fasse le joueur 2. Notez que "gagnant" se réfère toujours au joueur 1 qui commence, même si c'est au joueur 2 de jouer à partir de ce nœud. On dit que le joueur 1 (vous) a une stratégie gagnante si la racine est gagnante. On vous garantit que la racine est gagnante pour le corpus généré.

Question à développer pendant l'oral 3 *Formellement, dans quels cas un nœud est-il gagnant ?*

Un nœud est gagnant si et seulement si :
- il est contrôlé par nous (donc de profondeur paire si on commence la numérotation à 0) et [terminal ou a au moins un fils gagnant]
- ou bien il n'est pas contrôlé par nous, n'est pas terminal et tous ses fils sont gagnants.

Question 6 *Calculer le nombre de nœuds gagnants qui sont des voisins directs de la racine dans le corpus $L(N)$ pour les valeurs de N suivantes :*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

Question à développer pendant l'oral 4 *Présenter votre algorithme et sa complexité en fonction de N, $T_{\max}$ et M.*

La règle ci-dessus nous donne un algorithme récursif : la règle est simple à calculer aux feuilles, et par induction. Donc on ne fait qu'un passage de chaque nœud et la complexité est $O(NT_{\max})$.

À présent que vous avez vérifié que le joueur 1 a une stratégie gagnante, on s'intéresse à la stratégie la plus concise, i.e. retenir un nombre minimum de mots pour pouvoir gagner au jeu.
Un ensemble de mots est stable si, en ne jouant que des lettres correspondant à des préfixes de ces mots, on s'assure que l'adversaire ne peut que jouer des préfixes de ces mots. D'une certaine manière, un tel ensemble piège l'adversaire. Formellement, un ensemble L' est stable si et seulement si pour tout mot w de L' et tout préfixe strict $p \subset w$ de taille impaire, si $v = \delta(0, p)$, alors on a la relation suivante : $\forall b \in \alpha(v), \exists w' \in L', pb \subseteq w'$.
Par exemple, pour le corpus $[\mathrm{ET}, \mathrm{EN}, \mathrm{BAS}]$, $\{\mathrm{ET}, \mathrm{EN}\}$ et $\{\mathrm{BAS}\}$ sont stables mais $\{\mathrm{ET}\}$ n'est pas stable.
On cherche alors un ensemble stable de taille minimale qui ne contient que des mots qui nous font gagner : un ensemble de mots distincts Antisèche(N) du corpus $L(N)$ de cardinal minimal tel que jouer seulement avec les mots de Antisèche(N) nous permette de gagner.

Question 7 *Calculer la taille de* Antisèche(N) *pour les valeurs de N suivantes :*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

Question à développer pendant l'oral 5 *Présenter votre algorithme et sa complexité en fonction de N, $T_{\max}$ et M.*

Un ensemble de mots distincts du corpus (resp. nous faisant gagner) correspond à un ensemble de nœuds terminaux de l'arbre (resp. gagnants). On cherche un sous-arbre qui contient la racine, dont les nœuds terminaux sont tous gagnants et en nombre minimum. Comment trouver un ensemble stable ? On part de la racine gagnante et on élague tous les nœuds non gagnants de l'arbre. Le sous-arbre obtenu n'a que des nœuds gagnants, contient la racine, et on peut montrer que l'ensemble des mots correspondant à ses nœuds terminaux est non vide et stable, par définition d'un nœud gagnant. Pour respecter la propriété de stabilité, il faut conserver tous les fils des nœuds contrôlés par l'adversaire, tandis que pour les nœuds qu'on contrôle, on peut se contenter de conserver un seul fils. Le nombre minimal de nœuds terminaux pour chaque sous-arbre enraciné peut être calculé par induction sur l'arbre : 1 pour les nœuds terminaux, le minimum des valeurs des fils gagnants lorsqu'on contrôle le nœud, la somme des valeurs des fils lorsqu'on ne contrôle pas le nœud. Complexité identique à la question précédente $O(NT_{\max})$.

3 Pas de stratégie gagnante contre un adversaire limité

Dans cette section, c'est l'adversaire qui commence : c'est lui le joueur 1. Heureusement, il n'est pas omniscient et même assez limité, ainsi vous vous intéressez au meilleur coup à jouer à partir d'un nœud donné. Ainsi dans cette section, on ne s'intéresse plus à gagner à coup sûr, mais à quantifier la valeur d'un nœud, d'un coup (i.e. une paire nœud-arête), d'une stratégie.
On introduit un nœud supplémentaire $\perp$ qui n'a pas de coup possible et on note $V^{\perp} = V \cup \{\perp\}$.

Déroulement d'une partie

On appelle politique une manière de jouer. Formellement, une politique $\pi(a|u)$ est la probabilité que l'on a de jouer la lettre $a \in \Sigma$ en étant dans le nœud u. Dans cette section nous allons étudier comment créer une séquence de politiques $\pi_0, \ldots, \pi_\ell$ où π_{i+1} est une amélioration de π_i. Au départ, vous ne savez pas quel coup jouer et jouez de façon uniformément aléatoire $\pi_0 = \pi_{\text{unif}}$ parmi les coups possibles depuis chaque nœud. Au fur et à mesure de l'apprentissage vous améliorerez votre politique.
Les sessions $t = 1, \ldots, T$ correspondent à votre tour de jeu. Une session contient deux tours, joueur 2 puis joueur 1, et se déroule de la façon suivante : à partir d'un nœud u, on peut jouer une lettre $a \in \alpha(u)$ qui aboutit au nœud $v = \delta(u, a)$ de l'adversaire. Si l'on perd immédiatement on atterrit dans $\perp$. Sinon, l'adversaire peut jouer une lettre supplémentaire b, nous fait atterrir dans le nœud $u' = \delta(v, b) = \delta(u, ab)$, et on passe à la session suivante. Dans toute cette section, lorsque l'adversaire doit jouer, il emprunte l'arête proportionnellement au nombre d'occurrences de la lettre dans le corpus :

$$p(b|v) = \frac{\textsc{NbOcc}(b, L)}{\sum_{b' \in \alpha(v)} \textsc{NbOcc}(b', L)}$$

où $p(b|v)$ est la probabilité que l'adversaire joue la lettre b en partant du nœud v.

Question 8 *Pour les valeurs de N suivantes, calculer la lettre ℓ_0 du premier coup le plus probable joué par l'adversaire ; en cas d'égalité, donner la plus petite lettre.*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

Mécanisme de récompense

Au début de chaque session, le jeu est sur un nœud u où l'on doit jouer, et il faut choisir une lettre $a \in \alpha(u)$ qui amène le jeu dans le nœud $v = \delta(u,a)$. Si l'on forme un mot du corpus, alors, pour cette section seulement, on dit que l'adversaire amène le jeu dans l'état spécial $u' = \perp$ avec une récompense $r = -1$. Sinon la partie continue et l'adversaire choisit une lettre $b \in \alpha(v)$ selon la loi de probabilité $p(b|v)$ et amène le jeu dans l'état $u' = \delta(v,b)$ avec une récompense qui vaut $r = 1$ s'il forme un mot du corpus et $r = 0$ sinon, auquel cas la partie continue et on passe à la session suivante.
On note $p(u',r|v)$ la probabilité d'atterrir dans le nœud u' avec une récompense r en partant de $v = \delta(u,a)$ où a est la lettre qu'on joue depuis le nœud u.

Question à développer pendant l'oral 6 *Pour quelles valeurs de* $(u',r,v) \in V^{\perp} \times \{-1,0,1\} \times V$ *est-ce que* $p(u',r|v)$ *est non nul ?*

Seulement pour les nœuds v contrôlés par l'adversaire :
si $v \in T$, $p(\perp,-1|v) = 1$
si $v \notin T$, $\begin{cases} \forall b \in \alpha(v), p(u',1|v) = p(b|v) \text{ si } u' = \delta(v,b) \in T \\ \forall b \in \alpha(v), p(u',0|v) = p(b|v) \text{ si } u' = \delta(v,b) \notin T. \end{cases}$

Détaillons à présent ce que l'on cherche à optimiser :

- U_t est une variable aléatoire correspondant au nœud à partir duquel on joue à l'instant t, A_t la lettre jouée à l'instant t qui aboutit au nœud U_{t+1} (à partir duquel on joue à l'instant $t+1$) avec une récompense de R_{t+1} ;
- γ est un facteur dit d'actualisation pour les récompenses futures, dans tout ce sujet $\gamma = 0{,}9$;
- G_t est la récompense amortie : $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. Ainsi cette valeur objectif accorde plus d'importance aux récompenses proches qu'aux récompenses lointaines dans le temps. Intuitivement, on souhaite maximiser nos chances de gagner dans un futur proche.

De plus pour une politique π on note :

- la fonction de valeur $\mathcal{V}_\pi$ sur les nœuds, qui représente la récompense amortie moyenne si l'on part du nœud u et que l'on suit la politique π : $\mathcal{V}_\pi(u) = \mathbb{E}_\pi[G_t|U_t = u]$;
- la fonction de valeur $\mathcal{Q}_\pi$ sur les coups (nœuds-arêtes), qui représente la récompense amortie moyenne si l'on part du nœud u, que l'on joue la lettre a et que l'on suit la politique π : $\mathcal{Q}_\pi(u,a) = \mathbb{E}_\pi[G_t|U_t = u, A_t = a]$.

Évaluation de politique

On définit l'opérateur de Bellman $\mathcal{K}_\pi$:

$$\mathcal{K}_\pi(V)(u) = \sum_{a \in \alpha(u)} \left(\pi(a|u) \sum_{(u',r) \in V^{\perp} \times \{-1,0,1\}} p(u',r|\delta(u,a))[r + \gamma V(u')] \right).$$

L'opérateur $\mathcal{K}_\pi$ est contractant et admet un point fixe qui est $\mathcal{V}_\pi$ (on ne vous demande pas de le prouver). Ainsi on définit la suite $V_\pi^{(0)}(u) = 0$ pour tout u et $V_\pi^{(i+1)} = \mathcal{K}_\pi(V_\pi^{(i)})$, cette suite tend vers $\mathcal{V}_\pi$ et on supposera que $V_\pi^{(100)} = \mathcal{V}_\pi$.

Question 9 *Pour les valeurs de* N *suivantes, calculer* $V_{\pi_{\text{unif}}}^{(100)}(u'_0)$ *où* π_{unif} *est la politique initiale uniformément aléatoire et* u'_0 *est le nœud à partir duquel on joue si l'adversaire commence en jouant* ℓ_0 *:*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

Question à développer pendant l'oral 7 *Donner une borne supérieure sur le nombre d'itérations nécessaires à la convergence. En déduire un autre algorithme pour calculer* $\mathcal{V}_\pi$ *et sa complexité.*

Le nombre d'itérations est au plus $T_{\max}+1$ car la structure est un DAG de profondeur $T_{\max}+2$ dont les feuilles ont déjà la valeur correcte 0. Comme il s'agit de ce cas particulier, on peut calculer par induction la valeur de $\mathcal{V}_\pi$ en complexité $O(NT_{\max})$.

Amélioration de politique

À présent qu'on a calculé la valeur de chaque nœud, on peut améliorer notre politique :

$$\pi_{i+1}(a|u) = \operatorname*{argmax}_a \mathcal{Q}_{\pi_i}(u,a) \triangleq \operatorname*{argmax}_a \sum_{(u',r)\in V^\perp\times\{-1,0,1\}} p(u',r|\delta(u,a))[r+\gamma\mathcal{V}_{\pi_i}(u')]$$

À nouveau, en cas d'égalité, on préférera la plus petite lettre. Ce qui signifie que $\pi_{i+1}(a|u)$ vaut 1 ssi a est la plus petite lettre vérifiant $\mathcal{Q}_{\pi_i}(u,a) = \max_{a'} \mathcal{Q}_{\pi_i}(u,a')$, et 0 partout ailleurs.

Question 10 *Alterner entre évaluation de $\mathcal{V}_\pi$ puis amélioration de π jusqu'à convergence. Si l'adversaire joue ℓ_0 et qu'on atterrit en u'_0, quel est le meilleur coup à jouer pour les valeurs suivantes de N ?*

a) $N = 10$ ***b)*** $N = 1\,000$, ***c)*** $N = 100\,000$.

Question à développer pendant l'oral 8 *Expliquer l'algorithme utilisé et donner sa complexité par itération en fonction de N, $T_{\max}$, M.*

L'évaluation de $\mathcal{V}_\pi$ et l'amélioration de π se font en un parcours du DAG et ont chacune un coût $O(NT_{\max})$ car il y a autant de coups possibles que de nœuds.

Question à développer pendant l'oral 9 *Si le nombre de nœuds est fini et petit, et qu'on considère un graphe quelconque de jeu, quel algorithme proposez-vous pour déterminer la fonction de valeur sur les nœuds $\mathcal{V}_\pi$?*

L'équation $\mathcal{K}_\pi(V) = V$ peut être écrite sous forme matricielle, et cela devient un système linéaire à résoudre. La complexité est $O(N^3)$ par l'algorithme du pivot de Gauss.

Fiche réponse type : Jeux sur des arbres (*tries on tries*).

$\widetilde{u_0}$: 42

Question 1

a) 176

b) 338

c) 645

Question 2

a) (3, 16, 12, 5)

b) (1, 11, 15, 12)

c) (13, 6, 14, 13)

Question 3

a) 10

b) 1395

c) 79680

Question 4

a) 8

b) 883

c) 86502

Question 5

a) 6

b) 740

c) 72795

Question 6

a) 2

b) 3

c) 16

Question 7

a) 1

b) 1

c) 17

Question 8

a) 3

b) 0

c) 5

Question 9

a) -0.9

b) 0.10435984226959871

c) -0.12291525213759016

Question 10

a) 16

b) 5

c) 8

Coloriages de graphes.

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2021

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !

Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...

Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***. Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

1 Préliminaires

Notations et rappels

On rappelle que pour deux entiers naturels a et b, $a \mod b$ désigne le reste de la division entière de a par b, c'est à dire l'unique entier r avec $0 \leq r < b$ tel que $a = k \times b + r$ pour $k \in \mathbb{N}$.
Ce sujet ne portant que sur les graphes non orientés, chaque utilisation du mot graphe doit être entendue comme graphe non orienté. Un graphe est donc, pour ce sujet, un couple $G = \langle V, E \rangle$ où V est l'ensemble fini des sommets ou nœud du graphe et E est l'ensemble des arêtes, chaque arête étant un ensemble de sommets de cardinal 2. Notez que cette définition interdit les boucles (un sommet n'est jamais relié à lui-même par une arête).
Un coloriage d'un graphe $G = \langle V, E \rangle$ est une fonction qui associe une couleur à chaque sommet du graphe. Dans ce sujet les couleurs seront codées par des entiers et un coloriage sera donc une fonction de type $V \to \mathbb{N}$. Un coloriage d'un graphe $G = \langle V, E \rangle$ est dit valide s'il n'existe pas deux sommets de même couleur reliés par une arête. Formellement, un coloriage c est valide quand pour tout $\{a, b\} \in E \Rightarrow c(a) \neq c(b)$.
Le nombre de couleurs d'un coloriage c pour un graphe $\langle V, E \rangle$ est le nombre de couleurs différentes utilisées par c, c'est-à-dire le cardinal de l'ensemble $c(V)$. Le nombre chromatique d'un graphe est le nombre minimum de couleurs d'un coloriage valide de ce graphe.

Générateur de nombres pseudo-aléatoires

Étant donné u_0 on définit la récurrence :

$$\forall t \in \mathbb{N},\ u_{t+1} = (900\,007 \times u_t) \mod 1\,000\,000\,007$$

L'entier u_0 vous est donné, et doit être recopié sur votre fiche réponse avec vos résultats. Une fiche réponse type vous est donnée en exemple, et contient tous les résultats attendus pour une valeur de u_0 différente de la vôtre (notée $\widetilde{u_0}$). Il vous est conseillé de tester vos algorithmes avec cet $\widetilde{u_0}$ et de comparer avec la fiche de résultat fournie. Pour chaque calcul demandé, avec le bon choix d'algorithme le calcul ne devrait demander qu'au plus de l'ordre de la seconde, jamais plus d'une minute.

Question 1 *Calculer les valeurs suivantes :*

a) $u_1 \mod 1\,000$ ***b)*** $u_{42} \mod 1\,000$ ***c)*** $u_{1\,000} \mod 1\,000$ ***d)*** $u_{10^6} \mod 1\,000$

```
u0 = int(input())
u = [u0]
while len(u)<1000003:
    u.append( (u[-1]*900007) % (10**9+7))
```

Élimination de doublons dans une liste d'entiers

Il existe différentes méthodes pour éliminer les doublons d'une liste. Dans ce sujet nous allons présenter une méthode qui peut être efficace sur les listes composées d'entiers naturels où l'on

connaît une borne sur l'entier le plus grand présent dans la liste.

```
Le tableau estDoublon est supposé déjà alloué à la taille N_max = 100 000
La liste l ne contient que des entiers dans l'intervalle [0, N − 1] avec N_max ≥ N
fonction ÉlimineDoublon(l) :
    pour i ∈ l faire
        estDoublon[i] ← Faux
    fin
    listeSansDoublon ← liste vide
    pour i ∈ l faire
        si non estDoublon[i] alors
            Ajoute i à listeSansDoublon
            estDoublon[i] ← Vrai
        fin
    fin
    retourner listeSansDoublon
```

Question à développer pendant l'oral 1 *Expliquer le fonctionnement de la fonction* ÉlimineDoublon, *justifier de sa correction et donner sa complexité en fonction de la taille de la liste* l *(on ne prendra pas en compte le temps d'allocation du tableau* estDoublon*).*

La fonction ÉlimineDoublon prend en paramètre une liste d'entiers l et travaille en deux temps. Dans une première boucle elle s'assure que estDoublon$[i]$ est à Faux pour chaque élément de la liste. Dans une deuxième boucle on itère à nouveau sur les éléments de la liste, si estDoublon$[i]$ est à Faux c'est que c'est la première fois que l'on voit i donc il n'est pas encore dans listeSansDoublon et on l'y ajoute sinon on ignore i. Au total listeSansDoublon contient bien tous les éléments de l mais aucun doublon.
La complexité est celle de faire deux itérations sur l donc $O(|l|)$.

Question 2 *On note* $L(M) = \{u_i \mod M \mid 0 \leq i < M\}$. *Calculer le cardinal de l'ensemble* $L(M)$ *pour les valeurs de* M *suivantes :*

a) $M = 10$ ***b)*** $M = 5\,432$ ***c)*** $M = 98\,765$

```python
estDoublon = [False]* 100000
def sansDoublons(l):
    for i in l:
        estDoublon[i] = False
    r = []
    for i in l:
        if not estDoublon[i]:
            r.append(i)
        estDoublon[i]=True
    return r

def l(m):
    return len(sansDoublons([u[i] % m for i in range(m)]))
```

Générateur de graphes pseudo-aléatoires

Étant donnés deux nombres N et M on définit le graphe $\mathcal{G}(N, M)$ comme le graphe qui a pour sommets l'ensemble $V = \{0 \dots N-1\}$ et pour arêtes l'ensemble $E = \{\{l_i, g_i\} \mid 0 \leq i < M\}$ où l_i et g_i sont définis de la façon suivante :

$$\begin{aligned} l_i &= u_{2i} \mod N \\ g_i &= (l_i + 1 + (u_{2i+1} \mod (N-1))) \mod N \end{aligned}$$

Comme il est possible d'avoir une paire d'entiers (i, j) (avec $i \neq j$) telle que $l_i = l_j$ et $g_i = g_j$ ou alors $l_i = g_j$ et $g_i = l_j$, il est possible que $\mathcal{G}(N, M)$ contienne un nombre d'arêtes inférieur strictement à M. Pour construire le graphe, il est recommandé de construire dans un premier temps la liste des voisins de chaque sommet avec éventuellement des doublons, puis d'utiliser la fonction ÉlimineDoublon pour les éliminer.

Question 3 *Calculer le nombre de voisins du sommet 0 dans les graphes $\mathcal{G}(N, M)$ pour les valeurs de N et M suivantes :*

a) $N = 5$, $M = 10$

b) $N = 100$, $M = 300$

c) $N = 1\,234$, $M = 123\,456$

d) $N = 98\,765$, $M = 234\,567$

```python
def g(n, m):
    graphe = [[] for _ in range(n)]
    for i in range(m):
        l = u[2*i] % n
        g = (l+ (u[2*i+1]%(n-1)) + 1) % n
        graphe[l].append(g)
        graphe[g].append(l)
    return [sansDoublons(s) for s in graphe]
```

Question 4 *Calculer le nombre total d'arêtes des graphes $\mathcal{G}(N, M)$ pour les valeurs de N et M suivantes :*

a) $N = 5$, $M = 10$ ***b)*** $N = 100$, $M = 300$

c) $N = 1\,234$, $M = 123\,456$ ***d)*** $N = 98\,765$, $M = 234\,567$

```
def nbArr(g):
    return sum(len(l) for l in g)/2
```

Question à développer pendant l'oral 2 *Expliquer comment le graphe $\mathcal{G}(N, M)$ est généré et quelle est la complexité de cette construction en fonction de N et M. Expliquer aussi la structure de données choisie pour stocker le graphe.*

Le graphe est d'abord généré avec doublons et stocké sous forme de liste d'adjacence. Une fois que l'on a ajouté toutes les arêtes on passe chaque liste de voisins d'un nœud dans la fonction qui élimine les doublons.
La complexité est linéaire en M pour produire toutes les arêtes puis à nouveau linéaire sur chaque liste d'adjacence donc linéaire au total en M.

2 Une première heuristique pour approximer le nombre chromatique

Un « recoloriage » d'un sommet s dans un coloriage consiste à remplacer la couleur du sommet s par le plus petit entier naturel qui n'est pas utilisé par un des voisins de s.
Un premier algorithme, que l'on nomme Colorier1 pour colorier les graphes consiste simplement à partir du coloriage où tous les sommets sont coloriés à 0 puis à recolorier chacun des sommets, en commençant par le sommet 0, puis le sommet 1, puis le sommet 2, etc., jusqu'au sommet $N - 1$.

Question 5 *Dans le coloriage renvoyé par Colorier1 sur les graphes $\mathcal{G}(N, M)$ quelle est la couleur du sommet $\lfloor N/2 \rfloor$ et quel est le nombre de couleurs utilisées pour pour les valeurs de N et M suivantes :*

a) $N = 5$, $M = 10$ ***b)*** $N = 100$, $M = 300$

c) $N = 1\,234$, $M = 123\,456$ ***d)*** $N = 98\,765$, $M = 234\,567$

```python
def recolorie(gr,col,i):
    couleursPrises = [col[j] for j in gr[i]]
    for c in range(len(couleursPrises)+1):
        estDoublon[c]=False
    for c in couleursPrises:
        estDoublon[c]=True
    for c in range(len(couleursPrises)+1):
        if not estDoublon[c]:
            col[i] = c
            return None

def colorier1(n,m):
    gr=g(n,m)
    c = [0]*n
    for i in range(n):
        recolorie(gr,c,i)
    return
```

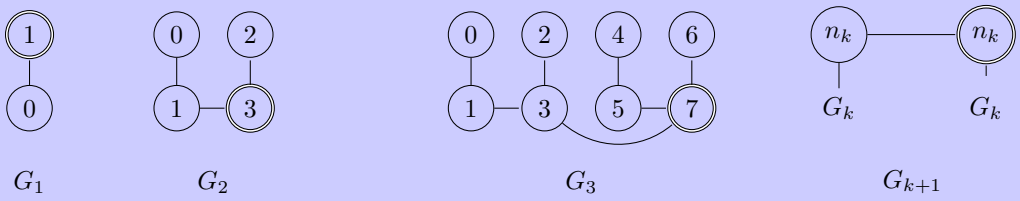
Question à développer pendant l'oral 3 *Justifier que* Colorier1 *renvoie toujours un coloriage valide. Quelle est la complexité de votre programme pour calculer* Colorier1 *(on ne prendra pas en compte le temps de construction du graphe) ?*

On utilise l'invariant suivant : après la k-ième itération de boucle les nœuds 0 à k n'ont aucun voisin de la même couleur qu'eux. En effet recolorier choisit toujours une couleur non utilisée et donc quand on recolorie un nœud il n'a pas la même couleur que ses voisins. De plus cela ne va pas créer de conflit avec les nœuds déjà recoloriés.
L'algorithme ci-haut appelle simplement n fois la fonction `recolorie`. Cette fonction parcourt tous les voisins pour obtenir leur couleur puis cherche une couleur non utilisée, pour cela on regarde parmi les « nombre de voisins » plus une premières couleurs laquelle est libre. Cette fonction a donc trois boucles non imbriquées qui prennent chaque un temps proportionnel au nombre de voisins. Le temps total de `colorier1` est donc proportionnel à la somme du nombre de voisins de chaque nœud c'est à dire proportionnel au nombre d'arêtes.

Question à développer pendant l'oral 4 *Exhiber un exemple de graphe où le nombre de couleurs du coloriage renvoyé par* Colorier1 *est supérieur strictement au nombre chromatique. Le nombre de couleurs utilisées par* Colorier1 *reste-t-il, même dans le pire cas, proche du nombre chromatique ?*

Nous allons directement exhiber une suite de graphes $G_1 \dots$ où chaque G_k est bicoloriable mais où le nœud n_k serait colorié avec la k-ième couleur (et donc le graphe serait colorié avec au moins k couleurs). On pose G_1 comme le graphe avec deux nœuds $n_1 = 0$ et 1 tous deux reliés par une

arête. Ensuite on construit G_{k+1} en utilisant deux copies de G_k et une arête entre les deux nœuds n_k. L'ordre des sommets est important pour le coloriage : on met donc d'abord les sommets de la première copie puis les sommets de la seconde copie et on fixe n_{k+1} comme étant le n_k de cette seconde copie.
On remarque facilement que le sommet n_{k+1} est bien colorié avec la couleur $k+1$ car il a des voisins de chacune des couleurs 0 à $k-1$ dans la deuxième copie de G_k et il a comme voisin un sommet de couleur k.
On remarque aussi que ce graphe est bicoloriable car on peut colorier G_{k+1} en coloriant G_k et mettant une moitié de G_{k+1} avec ce premier coloriage et on colorie l'autre moitié avec les couleurs inversées. Note : on pouvait aussi remarquer que c'est un arbre (il n'y a pas de cycle) et que tous les arbres sont bicoloriables.



G_1 G_2 G_3 G_{k+1}

3 Colorier avec deux couleurs

Tester si un graphe est coloriable avec deux couleurs est un problème bien plus simple que de tester si un graphe est coloriable avec K couleurs pour $K \geq 3$. Nous allons donc commencer par nous intéresser au cas bicolore.
Pour le cas bicolore on remarquera que dans un coloriage, la validité d'un coloriage correspond au fait qu'un nœud est toujours de la couleur qui n'est pas celle de ses voisins. On peut donc générer un premier coloriage en appliquant cette propriété récursivement puis vérifier que le coloriage obtenu est bien correct.
On note $P(N, M, K)$ comme la chaîne de caractère $b_0 b_1 b_2 \dots b_9$ où $b_i = 1$ quand $\mathcal{G}(N+i, M+i)$ est coloriable avec K couleurs et $b_i = 0$ sinon.

Question 6 *Calculer $P(N, M, 2)$ pour les valeurs de N et M suivantes :*

a) $N = 10$, $M = 6$ ***b)*** $N = 1234$, $M = 543$ ***c)*** $N = 98\,765$, $M = 45\,678$

```python
def bicoloriable(n,m):
    couleur = [None] * n
    gr = g(n,m)
    for i in range(n):
        if couleur[i] == None:
            todo = [i]
            couleur[i] = 0
            while len(todo):
                t = todo.pop()
                for v in gr[t]:
                    if couleur[v]==couleur[t]:
                        return '0'
                    if couleur[v] == None:
                        couleur[v] = 1-couleur[t]
                        todo.append(v)
    return '1'

def p2(n,m):
    return ''.join([bicoloriable(n+k,m+k) for k in range(10)])
```

Question à développer pendant l'oral 5 *Décrire l'algorithme utilisé pour tester si un graphe est coloriable à deux couleurs. Donner et justifier sa complexité (qui ne prend pas en compte le temps de construire le graphe).*

Comme proposé dans le sujet nous allons générer un coloriage et vérifier qu'il est correct. Ces deux opérations auront lieu dans un même parcours de graphe (ici en profondeur).
Il est important de remarquer que l'on lance un parcours depuis chaque nœud non colorié mais il se trouve que le parcours reste linéaire en le nombre d'arêtes plus le nombre de nœuds en constatant que chaque ajout dans `todo` force une couleur et donc il y a au plus N ajouts dans `todo` et M itérations de la boucle sur les voisins d'un nœud (chaque nœud et chaque arête étant traité une unique fois).

4 Une seconde heuristique pour approximer le nombre chromatique

Étant donné un graphe G, on dit qu'il est naïvement coloriable avec K couleurs si G est vide ou s'il existe un sommet n de degré strictement inférieur à K tel que G' soit naïvement coloriable avec K couleurs où G' est le graphe G où l'on a retiré le sommet n et ses arêtes adjacentes.

Question à développer pendant l'oral 6 *Justifier qu'un graphe naïvement coloriable avec K couleurs est un graphe qui est coloriable avec K couleurs. Montrer que si G est un graphe naïvement coloriable alors n'importe quel G' obtenu en enlevant des arêtes et des sommets de G est naïvement coloriable avec K couleurs.*

Tout d'abord on remarque que la définition de naïvement coloriable avec K couleurs revient à l'existence d'une suite $u_1 \dots u_n$ de sommets à enlever avec la condition que quand on retire le sommet u_i celui-ci a un degré au plus $K-1$.
Nous allons maintenant justifier qu'un graphe naïvement coloriable avec K couleurs est un graphe qui est coloriable avec K couleurs. Pour cela on part d'une séquence $u_1 \dots u_n$ pour retirer les sommets de G et d'un coloriage où tous les nœuds ont une couleur en dehors de $[\![0, K-1]\!]$ (par exemple la couleur K). On va ensuite recolorier les sommets en suivant l'ordre $u_n \dots u_1$. Par définition de la séquence à chaque fois qu'on recolorie un sommet u_i il a au plus $K-1$ sommets voisins coloriés avec une couleur dans l'intervalle $[\![0, K-1]\!]$ et donc il reste au moins une couleur de disponible. Au final tous les sommets vont donc être recoloriés avec une couleur qui sera entre 0 et $K-1$ et donc le graphe est K-coloriable.
Par ailleurs on note qu'en retirant un nœud et ses arêtes adjacentes, le degré des nœuds du graphe ne fait que diminuer. Donc s'il existe une séquence pour retirer tous les sommets du graphe G, il existera toujours une séquence pour retirer les sommets d'un graphe G' où l'on a retiré des sommets de G. En effet une séquence u pour G peut être adaptée en une séquence pour G' en retirant de u les sommets déjà enlevés.

Question 7 *Calculer le plus petit K tel que $\mathcal{G}(N, M)$ est naïvement coloriable avec K couleurs pour les valeurs de N et M suivantes :*

a) $N = 10,\ M = 32$

b) $N = 123,\ M = 3\,456$

c) $N = 4\,321,\ M = 100\,000$

d) $N = 54\,321,\ M = 423\,000$

```python
def naivementColoriable(n,m):
    gr=g(n,m)
    deg = [len(l) for l in gr]
    todo = [[] for _ in range(n)]
    estFait = [False] * n
    faits = 0
    for v in range(n):
        todo[len(gr[v])].append(v)
    for k in range(n):
        while len(todo[k]):
            v=todo[k].pop()
            if not estFait[v]:
                estFait[v] = True
                faits+=1
                for vois in gr[v]:
                    deg[vois]-=1
                    todo[deg[vois]].append(vois)
        if faits == n:
            return k+1
```

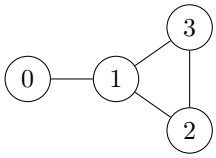
Question à développer pendant l'oral 7 *Décrire l'implémentation de l'algorithme utilisé. Donner*

et justifier sa complexité.

Tester pour un K donné si le graphe est K - est assez simple : on liste tous les nœuds de degré inférieur à K et on les met dans une liste à traiter. Pour traiter un nœud on parcourt ses voisins en décrémentant leur degré. Si le degré d'un nœud passe à $K-1$ on l'ajoute à la liste des nœuds à faire. Ce test est linéaire en la taille du graphe $O(N+M)$ ici.
Ceci nous donne une première idée d'algorithme : on teste tous les K un par un jusqu'à trouver le bon ce qui donne un algorithme trop lent en $O(K \times (N+M))$ mais qui pouvait obtenir des points en laissant tourner l'algorithme.
Une deuxième idée consiste à remarquer que la fonction « ce graphe est-il naïvement coloriable avec K couleurs ? » est une fonction monotone et donc on peut utiliser une dichotomie et obtenir une complexité de $O(\log(K) \times (M+N))$ ce qui est très rapide.
Une meilleure idée consiste à tester de nouveau tous les K par ordre croissant mais en gardant la trace des nœuds supprimés à l'étape K pour l'étape $K+1$. Pour cela on garde une liste `todo`$[k]$ des nœuds dont le degré courant est k ou dont le degré a été k. Pour tester un k on supprime tous les nœuds non déjà supprimés dans `todo`$[k]$ (à chaque fois que l'on touche à un nœud il faut penser à mettre à jour `todo` pour ses voisins). Au total on va traiter chaque nœud une seule fois et donc l'algorithme est en $O(N+M)$. Attention on peut avoir l'impression que `todo` est très gros, car un nœud peut être dans plusieurs `todo` mais il faut remarquer qu'un nœud ne sera au plus que dans les `todo` 0 à d où d est son degré et donc la complexité spatiale est $O(M)$.

5 Une troisième heuristique pour approximer le nombre chromatique

Nous introduisons maintenant un algorithme, Colorier2, pour colorier les graphes. L'idée de cet algorithme est de partir d'un coloriage invalide et de corriger progressivement ses erreurs.
Plus précisément, l'algorithme Colorier2 travaille récursivement, prenant en paramètre un graphe G et un coloriage C. Si C est valide, l'algorithme s'arrête sinon l'algorithme recolorie le sommet qui a le plus de voisins de la même couleur que lui (rappel : on recolorie toujours avec la plus petite couleur non utilisée par un voisin). En cas d'égalité entre plusieurs sommets qui ont le même nombre de voisins de même couleur, l'algorithme recolorie le plus petit sommet (c'est-à-dire le sommet codé par le plus petit nombre).
Par exemple sur le graphe ci-dessous, si tous les sommets sont initialement coloriés à 0, alors Colorier2 recoloriera d'abord le sommet 1 (qui a trois voisins de couleur 0) en la couleur 1. À ce moment, seuls 2 et 3 sont voisins de même couleur, on recolorie donc le sommet 2 (le sommet 3 a aussi un voisin de même couleur que lui, mais $3 > 2$) en la couleur 2 (la couleur 0 est prise par le sommet 3 et la couleur 1 par le sommet 1). L'algorithme s'arrêtera ensuite.



Question 8 *Dans le coloriage renvoyé par* Colorier2 *sur les graphes $\mathcal{G}(N, M)$ en partant du coloriage où tous les sommets sont coloriés à 0, quelle est la couleur du sommet $\lfloor N/2 \rfloor$ et quel est le nombre de couleurs utilisées pour les valeurs de N, M suivantes :*

a) $N = 5$, $M = 10$

b) $N = 100$, $M = 300$

c) $N = 1\,234$, $M = 123\,456$

d) $N = 98\,765$, $M = 234\,567$

```
def colorier2(n,m):
    gr=g(n,m)
    col=[0]*n
    nbErreurs = [len(l) for l in gr]
    aFaire = [ [] for _ in range(n+1)]
    for v in range(n):
        for i in range(len(gr[v])):
            aFaire[i+1].append(v)
    for erreur in range(n-1,0,-1):
        for v in aFaire[erreur]:
            if nbErreurs[v]==erreur:
                nbErreurs[v] = -1
                recolorie(gr,col,v)
                for vois in gr[v]:
                    nbErreurs[vois]-=1
    return col
```

Question à développer pendant l'oral 8 *Décrire l'implémentation de l'algorithme de* Colorier2 *utilisé. Donner et justifier sa complexité.*

Pour cette question il était possible (mais compliqué) d'obtenir un algorithme linéaire. En effet si on applique naïvement la définition on va recalculer les erreurs à chaque étape ce qui donne un algorithme en $O(M \times N)$. Il est possible d'être plus malin en mettant à jour le nombre d'erreurs et donc de n'avoir à chaque étape qu'à sélectionner le nœud avec la plus grosse erreur ce qui donne un algorithme en $O(N^2 + M)$.
En utilisant une file à priorité paresseuse on peut descendre la complexité à $O(\log(N) \times (N + M))$ mais c'est hors du programme et il existe une solution plus simple (mais avec une complexité similaire) que nous présentons maintenant.
L'idée consiste à créer N listes, la liste n° d contenant les nœuds dont le degré est supérieur à d par ordre d'indice. Nous allons ensuite itérer par nombre d'erreurs e décroissant. Pour chaque nombre d'erreur e, les sommets qui ont une erreur de e sont forcément dans la liste n° e. Attention il ne faut pas recolorier tous les sommets de la liste n° e car certains de ces sommets peuvent avoir déjà été recoloriés et d'autres peuvent avoir eu des voisins recoloriés. Il va donc falloir regarder parmi ces sommets ceux qui ont bien e erreurs au moment où on doit les traiter. Pour cela, on va maintenir le nombre d'erreurs de chaque sommet au fil des recoloriages.
La boucle principale du programme va partir d'un nombre d'erreurs maximal $e = N$ qu'on va faire décroître en traitant à chaque fois les sommets de la liste n° e qui ont un nombre d'erreurs de e. La

taille cumulée des listes est $O(M)$ car chaque sommet s est ajouté dans les listes n° 1 à degré de s. Chaque sommet ne sera recolorié qu'une fois, ce qui prend un temps proportionnel au degré de ce nœud donc on obtient un algorithme linéaire en $O(N + M)$.

6 Vers un algorithme pour colorier de façon optimale les graphes

Déterminer si un graphe est coloriable avec K couleurs pour $K \geq 3$ est un problème difficile pour lequel les seuls algorithmes connus ont une complexité exponentielle en le nombre de sommets. Un tel algorithme exponentiel consiste à générer tous les coloriages récursivement : tout d'abord on teste toutes les possibilités pour colorier un premier sommet, puis un second, puis un troisième, etc. À chaque fois que l'on a un coloriage complet on peut alors vérifier s'il est correct.
Bien que cette méthode soit valide, elle passe un temps important à générer des coloriages nécessairement invalides. En effet, rien ne sert de continuer à générer les coloriages récursivement si l'on a déjà deux sommets voisins coloriés avec la même couleur. Une seconde idée pour gagner du temps consiste à éviter de tester des coloriages symétriques, par exemple on peut décider que le premier sommet ne sera jamais colorié qu'avec la couleur 0 et donc que pour le second sommet, il suffit de tester la couleur 0 et la couleur 1, etc. Plus généralement, si on a un coloriage partiel avec ℓ couleurs, il suffit de tester les couleurs 0 à ℓ pour le sommet courant. D'autres idées pour réduire le temps de calcul sont possibles, on peut par exemple jouer sur l'ordre dans lequel on colorie les sommets (en coloriant à chaque étape un sommet contraint par ses voisins plutôt que de d'abord colorier le sommet 0 puis le sommet 1, etc.).

Question 9 *Calculer $P(N, M, K)$ pour les valeurs de N, M et K suivantes :*

a) $N = 5$, $M = 18$, $K = 3$

b) $N = 15$, $M = 40$, $K = 3$

c) $N = 35$, $M = 85$, $K = 3$

d) $N = 45$, $M = 100$, $K = 3$

e) $N = 55$, $M = 250$, $K = 4$

f) $N = 65$, $M = 825$, $K = 7$

g) $N = 75$, $M = 925$, $K = 7$

```python
def color(n,m,k):
    gr = g(n,m)
    utilisee = [[0 for _ in range(k)] for _ in range(n)]
    nbPris = [0 for _ in range(n)]
    def colorie(remaining,nbC):
        if remaining==[]:
            raise FOUND
        node = max(remaining,key=lambda x:(nbPris[x],len(gr[x])))
        remaining.remove(node)
        for c in range(min(nbC+1,k)):
            if utilisee[node][c] == 0:
                for v in gr[node]:
                    utilisee[v][c]+=1
                    if utilisee[v][c] == 1:
                        nbPris[v]+=1
                colorie(remaining,max(nbC,c+1))
                for v in gr[node]:
                    if utilisee[v][c] == 1:
                        nbPris[v]-=1
                    utilisee[v][c]-=1
        remaining.append(node)
    try:
        colorie(list(range(n)),0)
        return '0'
    except:
        return '1'
```

N'hésitez pas, pour cette question, à laisser tourner vos algorithmes quelques minutes mais attention, avec un algorithme trop naïf, il est impossible de réussir les cas les plus durs même en laissant tourner le calcul des heures (l'algorithme ayant une complexité exponentielle).
Pour obtenir tous les points, il faudra donc trouver et implémenter diverses optimisations qui réduisent le temps de calcul. Il est fortement conseillé de commencer par un algorithme naïf et d'implémenter petit à petit les optimisations en testant à chaque fois quels sont les cas résolubles rapidement.

Question à développer pendant l'oral 9 *Expliquer les idées (implémentées ou non) qui ont été explorées pour tester plus rapidement si les graphes sont coloriables avec K couleurs.*

Dans le code proposé, les idées implémentées pour diminuer le nombre d'appels récursifs sont les suivantes :

- compter pour chaque nœud, le nombre de couleurs disponibles qu'il a et colorier d'abord les nœuds pour lesquels on a peu de choix ce qui va permettre de plus rapidement trouver les contradictions ;
- en cas d'égalité, commencer par colorier les nœuds qui affectent le plus d'autres nœuds (donc les nœuds de plus haut degré). À nouveau l'idée est de trouver rapidement les contradictions ;

— maintenir le nombre `nbC` de couleurs utilisées et quand on colorie un nœud on essaie seulement de le colorier avec soit une des couleurs déjà utilisées, soit avec la couleur `nbC`. Cela permet d'éliminer beaucoup de coloriages symétriques.

Les idées décrites ci-haut permettent de diminuer **exponentiellement** le nombre d'appels récursifs. Une autre manière d'accélérer le code est de rendre plus rapide chaque appel récursif. Notez qu'il est beaucoup plus intéressant de diminuer le nombre d'appels que d'optimiser chaque appel ! Voici quelques idées pour optimiser chaque appel :

— maintenir la liste des nœuds qu'il reste à colorier. Ainsi quand il n'y a que peu de nœuds à colorier on peut les parcourir en $O(n)$ où n est le nombre de nœuds restants. Cette optimisation est utile car on va faire un seul appel pour colorier le premier nœud mais potentiellement des millions pour les derniers ;

— maintenir le nombre de couleurs disponibles pour chaque nœud.

Fiche réponse type : Coloriages de graphes.

$\widetilde{u_0}$: 54

Question 1

a) 378

b) 706

c) 573

d) 94

Question 2

a) 7

b) 3463

c) 62416

Question 3

a) 2

b) 3

c) 172

d) 7

Question 4

a) 6

b) 288

c) 114064

d) 234562

Question 5

a) 1 / 3

b) 3 / 7

c) 27 / 51

d) 3 / 8

Question 6

a) 1110001100

b) 1111101011

c) 0010111111

Question 7

a) 5

b) 38

c) 36

d) 12

Question 8

a) 0 / 2

b) 2 / 5

c) 41 / 46

d) 4 / 6

Question 9

a) 0000101101

b) 1010101111

c) 0100011000

d) 0110111110

e) 0000001011

f) 0000000111

g) 0000110111