BANQUE MP INTER-ENS – SESSION 2019
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) : 12,7%
- Paris-Saclay : 13,2%
- Rennes : 17,1%
- Ulm : 13,3%

Nota : pour l'année 2020, le coefficient pour Lyon passera à 16,9% (toutes options).

Jury : Nathanaël Fijalkow, Louis Jachiet, Guillermo A. Pérez, Samuel Thibault

Contenu de ce document

Nous rappelons l'organisation de l'épreuve, 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 trois corrigés : un en Python et un en OCaml pour le même sujet, et un en Python pour un sujet différent.

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 3h30, 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 s'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 des entrées pseudo-aléatoires 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 à l'examinateur une fiche réponse contenant les réponses aux questions écrites (purement numériques).

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 l'examinateur 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 pratique 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.

Remarques générales

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 leur code dans lequel les calculs produisant 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.

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 l'examinateur 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 l'examinateur 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 évalué dans le cadre de l'épreuve d'informatique fondamentale, alors que l'objet de l'oral 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. L'examinateur saura donc apprécier le recul que démontre un argument simple et intuitif par rapport à une suite d'arguments formels désincarnés.

De même, 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 fréquemment 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, l'examinateur saura apprécier un regard critique sur l'algorithme qui exploiterait l'analyse de complexité.

Langages de programmation. Les sujets sont de difficulté équivalente dans les divers langages. Nous notons une tendance générale à utiliser plutôt Python que OCaml. Il ne faut surtout pas perdre du temps en hésitant à trouver "le meilleur langage pour le sujet", car le sujet est calibré pour être implémentable avec le même niveau de difficulté en OCaml et en Python. On conseille aux candidat·es de choisir le 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, qu'en Python le nombre d'appels récursifs sont limités 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'une table est très visible dans ce type de sujet. Ceci 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 de l'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 : Qui est-ce ?

Ce sujet commençait par l'analyse de différentes stratégies pour le jeu « Qui est-ce ? » et se poursuivait par l'analyse de stratégies dans le cas où l'on autorise un des joueurs à mentir une fois.

Pour la partie écrite, les questions Q1 à Q4 ainsi que Q6 étaient des exercices de programmation qui ont été correctement traités par une majorité de candidat·es. Les questions Q1, Q3, Q4 et Q6 nécessitaient de traduire des définitions en langage informatique tandis que la question Q2 consistait à implémenter une fonction dont le pseudo-code était fourni. Les questions Q5 et Q7 nécessitaient une astuce algorithmique. En effet, comme indiqué dans le sujet, la méthode consistant à ré-utiliser plusieurs fois l'algorithme des questions Q4 et Q6 n'était pas assez efficace pour espérer une réponse à toutes les questions. Les questions Q8 à Q12 étaient plus difficiles et nécessitaient une réflexion algorithmique plus poussée et n'ont été que très peu traitées par les candidat·es. Ce sujet était très long avec beaucoup de questions très dures (Q8 à Q12). Répondre correctement à 5 questions assurait une note supérieure à la moyenne pour la partie écrite.

Pour la partie orale, la question QO1 consistait à analyser la correction d'un algorithme fourni. La plupart des candidat·es ont bien réussi cette question bien que trop souvent en se lançant dans de longues explications. Le jury rappelle qu'il est plus convaincant d'énoncer clairement un ou plusieurs invariants puis ensuite d'expliquer en quoi ils suffisent et comment les vérifier. Par ailleurs, les longues explications prennent du temps qui pourrait être utile aux candidat·es sur d'autres questions. Les questions QO2, QO4, QO5 et QO6 consistaient à analyser la complexité d'algorithmes écrits par les candidat·es ou fournis. Elles n'ont pas été bien réussies par les candidat·es. Il semble que la complexité des algorithmes n'est pas bien assimilée. Bien que les questions QO3 et QO6 étaient assez difficiles, nombre de candidat·es ont réussi à fournir une explication intéressante. À l'inverse la QO7 nécessitait surtout de bien comprendre le sujet et n'a été résolue que par une moitié des candidat·es. Les questions orales QO9 à QO13 ont été peu traitées car il s'agissait d'analyser la complexité d'algorithmes que les candidat·es n'ont pas trouvés.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Tous les points	100	95	85	87	59	87	44	0	8	0	3
Réponses partielles	100	100	100	100	82	92	72	0	10	0	5

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10	QO11	QO12	QO13
Tous les points	92	64	33	51	28	33	56	26	0	3	0	0	0
Réponses partielles	100	95	97	87	70	62	70	41	5	8	5	0	8

Table 1. Pourcentage de candidat·es qui ont répondu à chaque question du sujet 1

Sujet 2 : Plus petit ancêtre commun dans des arbres

Ce sujet proposait l'étude de différentes implémentations plus ou moins efficaces de la recherche de la position de l'élément minimum parmi un intervalle d'une liste, qui est une des méthodes pour calculer le plus petit ancêtre commun dans un arbre.

Les Q1 et Q3 n'ont pas posé problème. La partie c) de la Q2 a souvent posé problème en Python. En effet, pour rappel les "listes" de Python sont stockées sous forme de tableaux, et ainsi l'extraction d'une sous-liste (pour implémenter une approche récursive) ne peut être faite en temps constant. Sur une liste contenant presque un million d'éléments, c'est rédhibitoire. La QO1 n'a pas posé problème, les candidat·es ont seulement parfois manqué de simplicité dans leur explication.

Les Q4, Q5 et QO2 ont apporté les premières difficultés, autour de la notion de parcours d'arbre (ici en profondeur). La QO2, trop souvent mal expliquée, a montré que cette notion n'est pas suffisamment acquise, alors que cela aurait dû être une question facile.

La QO3 était l'occasion de prendre du recul sur le problème étudié et l'étendre au cas plus général des graphes. Elle a été assez bien réussie. La QO4 nécessitait d'avoir bien compris l'ensemble des notions introduites par le sujet. Il était précieux de s'aider d'un dessin pour expliquer facilement le raisonnement.

La Q6 et la QO5 portaient sur l'algorithme naïf parcourant simplement la liste. Le calcul de la complexité a cependant posé problème. Les candidat·es se sont trop souvent plongé·es dans l'exposé d'un calcul exact alors que le résultat $\mathcal{O}(n^3)$ s'obtient immédiatement à l'aide d'une approximation grossière. Les candidat·es devraient être habitué·es à reconnaître immédiatement les complexités du type $\sum_{i=1}^{n} i$ comme étant quadratiques, et ici ($\sum_{i=1}^{n}\sum_{j=1}^{n} |j-i+1|$) réaliser rapidement que la complexité est cubique, ce qui épargne tout calcul compliqué.

La Q7 et les QO6 et QO7 portaient sur une approche par mémoisation simple de $\mathcal{O}(n^2)$ valeurs. Les questions orales ont montré que l'approche était bien comprise. L'implémentation, qui commence à être un peu technique, n'a cependant été réussie que par un quart des candidat·es.

La Q8 et les QO8 et QO9 portaient sur une meilleure approche par mémoisation. Celle-ci a été beaucoup moins comprise. De nouveau, faire un dessin facilitait la compréhension de l'approche. Les candidat·es ont souvent eu du mal à se représenter ce que vaut $2^{\lfloor log_2(j-i) \rfloor}$ par rapport à $j-i$. La difficulté technique d'implémentation était accrue, un cinquième des candidat·es a pu calculer des réponses correctes.

La Q9 et les QO10, QO11 et QO12 portaient sur une approche découpant la liste en blocs. Un tiers des candidat·es a su l'expliquer, les autres ne l'ont pas abordée. Cette approche étant plutôt technique, aucun·e candidat·e ne l'a implémentée. L'analyse de sa complexité a cependant souvent été correctement faite parmi celles et ceux ayant su expliquer l'approche.

La Q10 et les QO13, QO14 et QO15 raffinaient l'approche par bloc en profitant d'une propriété des listes étudiées. Seul·es les quelques candidat·es les plus brillant·es l'ont abordée, aucun·e ne l'a implémentée et seul le plus brillant a su aborder l'analyse de sa complexité.

Pour obtenir une note de 10, il était suffisant d'implémenter les questions écrites préparatoires simples et de parcours d'arbre et savoir les expliquer facilement.

Écrit	Q1	Q2.a/b	Q2.c	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Tous les points	92	97	59	92	59	65	35	24	8	0	0
Réponses partielles	100	100	70	97	86	89	68	35	19	0	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9
Tous les points	89	65	78	54	65	84	84	27	22
Réponses partielles	100	100	100	100	95	95	89	68	35

QO10	QO11	QO12	QO13	QO14	QO15
19	14	11	5	3	0
27	16	14	8	3	3

TABLE 2. Pourcentage de candidat·es qui ont répondu à chaque question du sujet 2

SUJET 3 : SUITES UNIVERSELLES

Le sujet traitait de la notion de suites universelles (voir le sujet pour les définitions). Il s'agissait d'un sujet abstrait, très mathématique. Ces notions sont issus de travaux de recherche récents en informatique théorique, liées aux jeux de parité. Certaines questions orales nécessitaient une bonne compréhension combinatoire. L'essentiel de l'évaluation a porté sur l'aspect informatique, à savoir algorithmique, à la fois conception et mise en pratique.

Les cinq premières questions ont été correctement traitées par la plupart des candidat·es, et ne posaient pas de difficultés particulières. Elles permettaient aux candidat·es de montrer leur maîtrise du langage de programmation choisi, et de problématiques classiques (gestion de l'espace en particulier). Répondre parfaitement à toutes ces questions, y compris les questions d'oral, assurait la note 10.

Les questions 6 et 7 étaient assez ouvertes et permettaient de briller. Il y avait en effet plusieurs solutions possibles, dont certaines astucieuses et très efficaces, que certain·es candidat·es ont trouvé. La plupart des candidat·es se sont arrêté·es sur ces questions, qui ont fait la différence entre un 10 et un 14 pour la note finale. Nous conseillons aux candidat·es de présenter leurs idées même s'ils n'ont pas réussi à les mettre en pratique, de la manière la plus claire possible. Très souvent, les problèmes de mise en pratique étaient dus à une compréhension théorique imparfaite.

Les questions 8 et 9 étaient plus difficiles, elles permettaient d'obtenir les meilleures notes. Certain·es candidat·es les ont entièrement résolues. Elles nécessitaient la mise en pratique de techniques classiques (programmation dynamique en particulier), de manière non-triviale, sur un problème nouveau.

SUJET 4 : CHASSE AU TRÉSOR SUR GRAPHES

Ce sujet, très classique en termes de contenu et progression, s'intéressait au calcul du nombre de déplacements nécessaires pour déplacer un jeton le long d'un graphe afin d'atteindre un ensemble de pièces dorées dans des sommets particuliers. Il était composé de trois parties et commençait avec quelques questions simples pour s'assurer que la génération de graphes a été correctement implémentée. Dans cette première partie, il a également été demandé aux candidat·es de calculer le nombre de composants connexes afin de vérifier qu'elles et ils pouvaient utiliser correctement

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Tous les points	100	97	94	78	100	50	34	7	10
Réponses partielles	100	97	100	94	100	81	57	18	26

Oral	QO1	QO2	QO3	QO4	QO5	QO5b	QO6	QO7
Tous les points	63	92	60	89	94	28	7	26
Réponses partielles	92	97	97	97	94	86	50	47

TABLE 3. Pourcentage de candidat·es qui ont répondu à chaque question du sujet 3

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Tous les points	100	94	91	81	63	25	22	34	6
Réponses partielles	100	100	94	88	72	47	28	34	19

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7
Tous les points	81	50	38	91	44	38	16
Réponses partielles	97	100	100	91	78	53	22

TABLE 4. Pourcentage de candidat·es qui ont répondu à chaque question du sujet 4

un algorithme de parcours de graphe. Les deuxième et troisième parties traitaient le calcul du nombre optimal de mouvements et des configurations initiales optimales pour la position des pièces en or.

La première partie (questions 1–4) a été correctement traitée par la plupart des candidat·es. Cependant, nous observons que plusieurs candidat·es ont eu du mal à décrire et à analyser leur algorithme (basé sur un parcours de graphes) lors de l'examen oral.

La deuxième partie portait sur l'extraction du nombre maximal de pièces d'or que l'on pourrait collecter en utilisant un nombre donné de coups. Un algorithme glouton a été donné pour la première question et l'algorithme optimal a été considéré pour la question 6. Pour la question 7, le nombre de pièces d'or maximal pouvant être obtenu dans le pire des cas (sur les configurations initiales) a été considéré. Pour cette dernière, une mauvaise implémentation prenait trop de temps et l'analyse de leur algorithme était étonnamment difficile pour la plupart des candidat·es.

La dernière partie du sujet traitait le calcul du nombre maximal de pièces que l'on pouvait ramasser avec un nombre donné de coups. Ici, encore une fois, un algorithme glouton et l'algorithme optimal ont été considérés. Les algorithmes optimaux ici et dans la partie précédente n'étaient plus que "force brute", par contre peu de candidat·es ont su bien expliquer leur approche.

Proposition de solution pour *Sujet 3 : Suites Universelles* en Python

Note générale. De nombreu·ses candidat·es tentent d'aborder les sujets en Python mais utilisent Python comme OCaml, c'est à dire en utilisant beaucoup de fonctions récursives. Si les fonctions récursives sont possibles en Python, le langage se prête moins à cette utilisation. En particulier, nous tenons à rappeler que la taille de la pile d'appel est limitée, par défaut, à 1000 appels récursifs ! Cette limite peut être augmentée avec la fonction suivante : `sys.setrecursionlimit(`N`)` pour autoriser N appels récursifs.

Le choix de ce corrigé est de présenter un code Pythonesque (et donc de privilégier les boucles aux appels récursifs).

Question 1. On peut écrire le générateur de nombres de la façon suivante :

```
u0 = 42
a = 46613
b = 17
m = (2**31) - 1

u = [u0]
for i in range(6000000):
    u.append( (a * u[-1] + b) % m )
```

Après quoi le code pour la Q1 est :

```
def q1(n):
    return str(u[n]%1000)

print("Q1: "+(q1(1))+" "+q1(98)+" "+q1(9876))
```

Question 2. De même pour capacité :

```
def B(n,k,alpha):
    return [u[alpha*i]%n+1 for i in range(k) ]

def capacite(l):
    res = 0
    for i in l:
        res += i
    return res

b1 = B(9,5,13)
b2 = B(98,54,17)
b3 = B(987,543,41)
b4 = B(98765,54321,97)

print("Q2 : "+str(capacite(b1))+" "+str(capacite(b2))
        +" "+str(capacite(b3))+" "+str(capacite(b4)))
```

Question 3. Ou pour trouver le premier indice avec une valeur supérieure à b :

```
def premier(b,l):
    for i in range(len(l)):
        if b<=l[i]:
            return i
    return -1

print("Q3 : "+str(premier(9,b1))+" "+str(premier(86,b2))+" "+
              str(premier(975,b3))+" "+str(premier(98123,b4)))
```

Question 4. La question 4 nécessitait une légère réflexion. Il fallait remarquer que si on note $d(l_1, l_2)$ le plus long préfixe de l_2 dominé par l_1 alors $d(i_1 :: l_1, i_2 :: l_2) = d(l_1, i_2 :: l_2)$ quand $i_1 < i_2$ et $d(i_1 :: l_1, i_2 :: l_2) = d(l_1, l_2)$ sinon. Attention, on peut utiliser cette définition récursivement mais, en Python, extraire une sous-liste prend un temps linéaire en la taille de la sous-liste ! Il faut donc manipuler des indices plutôt que des sous-listes !

```
def domineJusqua(l1,l2):
    id1 = 0
    for id2 in range(len(l2)):
        while id1<len(l1) and l1[id1]<l2[id2]:
            id1+=1
        if id1==len(l1):
            return id2
        id1 += 1
    return len(l2)

print("Q4 : "+str(domineJusqua(b1,B(9,10,14)))
         +" "+str(domineJusqua(b2,B(98,68,43)))
         +" "+str(domineJusqua(b3,B(987,673,174)))
         +" "+str(domineJusqua(b4,B(98765,65431,81))))
```

Question orale 1. On peut considérer l'invariant suivant pour la boucle `for` : à chaque début de tour de boucle id_1 est la taille du plus petit préfixe nécessaire à dominer le préfixe de taille id_2 de l_2. De même on peut considérer l'invariant de boucle while suivant : à chaque début de tour de boucle `while`, id_1 n'est pas un préfixe possible pour le préfixe de taille $id_2 + 1$.

Question orale 2. La suite $(1, 2, 4, 3, 1)$ est 4-universelle. En effet elle domine L_4 qui est universelle d'après la question orale 3.

Question orale 3.
Propriété 1 : Si A est a-universelle et B est b-universelle alors $A, (a + b + 1), B$ est $a + b + 1$ universelle.

Preuve : Soit l une suite de capacité $a+b+1$ et soit p la taille minimale d'un préfixe tel que la capacité du préfixe de l de taille p soit plus grande (au sens large) que $a+1$. On décompose alors l en l_a, k, l_b avec l_a de taille $p-1$. Par définition, la capacité de l_a inférieure à a et celle de l_b inférieure à b. On a donc l_a dominée par L_a, k dominé par $a+b+1$ et l_b dominée L_b. □

Réponse à Q3 : L_n est universelle par propriété 1. Elle est aussi minimale. On va montrer par récurrence forte que L_n est une suite de capacité minimale parmi les suites n-universelles (elle n'est pas unique car, par exemple, son renversé $\bar{L_n}$ est aussi n-universel).

La preuve qui suit est compliquée et n'était pas attendue des candidat·es, nous avons noté les bonnes idées et l'honnêteté de la démarche.

Preuve : Soit U une suite n-universelle. Par n-universalité, U se décompose en $U_1, [n], U_2$ avec U_1 ne contenant pas n. Soit k_i l'entier maximal tel que U_i soit k_i-universel, il existe s_i de capacité k_i+1 non dominée par U_i. La concaténation $s_1 s_2$ n'est alors pas dominée par U et donc $k_1+k_2+2 > n$ (U est n-universelle). Par hypothèse de récurrence forte, on a donc capacité$(U_i) \geq$ capacité(L_{k_i}) et donc on peut remplacer U_1, n, U_2 en la suite de capacité moindre L_{k_1}, n, L_{k_2} tout en gardant la n-universalité (d'après propriété 1).

On veut maintenant montrer que le choix $k_1 = \lfloor n/2 \rfloor, k_2 = n-1-k_1$ est optimal, c'est à dire que $T(k_1)+T(k_2)$ est minimal. Comme T est croissante, on peut toujours réduire k_2 à $n-1-k_1$, mais il faut montrer que le meilleur équilibre est atteint pour $k_1 = \lfloor n/2 \rfloor$. Notons $T(n) =$ capacité(L_n) et $\Delta(n) = T(n) - T(n-1)$, remarquons :

— Quand $n = 2k+1$, on a $\Delta(n) = T_n - T_{n-1} = \big(T(k) + n + T(k)\big) - \big(T(k) + n - 1 + T(k-1)\big) = T(k) - T(k-1) + 1 = \Delta(\lfloor n/2 \rfloor) + 1$.

— Quand $n = 2k$ on retrouve aussi $\Delta(n) = T(k) - T(k-1) + 1 = \Delta(\lfloor n/2 \rfloor) + 1$.

Comme $\Delta(1) = 1$ on a $\Delta(n) = \lfloor ln_2(2n) \rfloor$. De cela, on déduit que pour $a > b$ on a $T(a) + T(b) \geq T(a-1) + T(b+1)$ car $a > b \Rightarrow \lfloor ln_2(2a) \rfloor \geq \lfloor ln_2(2(b+1)) \rfloor \Rightarrow \Delta(a) \geq \Delta(b+1) \Rightarrow T(a) - T(a-1) \geq T(b+1) - T(b)$.

On peut donc toujours diminuer $T(k_1) + T(k_2)$ en réduisant k_1 de 1 quand il est plus grand que k_2 ou l'augmentant quand il est plus petit et on peut donc obtenir $k_1 = \lfloor n/2 \rfloor$ en étant minimal.

□

Question 5. Pour la question 5, il était coûteux de matérialiser complètement L_n. Il fallait mieux s'appuyer sur le fait que capacite(L_n) = capacite$(L_{\lfloor n/2 \rfloor}) + n +$ capacite$(L_{n-\lfloor n/2 \rfloor} - 1)$

```
def capaciteL(n):
    if n <= 1:
        return n
    return capaciteL(n//2)+n+capaciteL(n-1-(n//2))

print("Q5 : "+str(capaciteL(6))
         +" "+str(capaciteL(987))
         +" "+str(capaciteL(98765))
         +" "+str(capaciteL(9876543)))
```

Question orale 4. L'algorithme déploie la définition en calculant la capacité sans matérialiser la suite. On remarque que cet algorithme est au plus linéaire en n car si on note $T(n)$ le nombre d'opérations on a $T(n) = O(1)+T(\lfloor n/2 \rfloor)+T(n-\lfloor n/2 \rfloor -1)$ mais $\lfloor n/2 \rfloor + (n - \lfloor n/2 \rfloor - 1) = n - 1$ donc on a bien $T(n) = O(n)$.

Question 6. Pour la question 6, on pouvait remarquer qu'il y avait 2^n suites de capacité n. En effet, on peut considérer la fonction qui associe, à n et à un nombre M entre 0 et $2^n - 1$, une suite de capacité n. Cette fonction $f(M)$ transforme $2^n + M = \sum_i 2^{b_i}$ en la suite $u_i = b_{i+1} - b_i$. On peut inverser cette fonction : soit u_i une liste de capacité n, on note $\tilde{u}[i] = \sum_{j \leq i} u_j$ et $B((u_i)_i) = \sum u_i \times 2^i$.

D'après la question précédente, il y a 2^k suites de capacité au plus k. On peut en déduire qu'il 2^{k-v+1} suites de capacité au plus k qui commence par un nombre supérieur à v (au sens large) : en effet chacune de ces suites peut s'envoyer de façon bijective sur une suite de capacité $k-v+1$ en retranchant $v-1$ à la première valeur de la suite. On a alors $2^k - 2^{k-v+1}$ suites de capacité au plus k mais qui commence par une valeur strictement inférieure à v.

Chaque suite inférieure à l peut s'écrire soit de la forme $l'ds$ où l' est un préfixe de l mais d diffère de la lettre après l' dans l soit de la forme l' où l' est un préfixe de l. Si l'on s'intéresse aux suites de capacité k où d est en position i on a $2^{k-\text{capacité}(l')} - 2^{k-\text{capacité}(l')-d+1} + 1$ suites. À cela il faut ajouter la suite qui termine en l'.

On en déduit l'algorithme suivant :

```
  def index_lexi(l, capa):
    before = 0
    for v in l:
        if capa < v:
            return -1
        before += 1+2**capa-2**(capa-v+1)
        capa -= v
    return before

b5 = B(2,4,13)
b6 = B(3,6,17)
b7 = B(3,8,41)
b8 = B(5,8,97)

print("Q6 : "+str(index_lexi(b5,7))
         +" "+str(index_lexi(b6,15))
         +" "+str(index_lexi(b7,20))
         +" "+str(index_lexi(b8,25)))
```

Question 7. De la même façon on peut traiter la question 7 de façon combinatoire. Ainsi on peut remarquer qu'il y a $\binom{n}{k}$ suites de capacité n qui ont k éléments (cela correspond à choisir k bits à 1 dans la représentation binaire d'un nombre entre 0 et $2^n - 1$.

Par dénombrement on peut alors trouver l'index de l dans l'ordre donné : on compte combien sont plus petites en taille puis on compte, comme à la question 6, l'indice parmi celles de même taille :

```
#notre calcul commence par calculer les coefficients binomiaux
k_parmi_n = [ [1]*1000 ]

for k in range(999):
    k_parmi_n.append([0]*1000)
    for n in range(999):
        k_parmi_n[k+1][n+1] = k_parmi_n[k][n]+k_parmi_n[k+1][n]

def index_size_lexi(l, capa):

    before = 1
    # d'abord compter les séquences plus petites en taille :
    for s in range(1,len(l)):
        for c in range(s,capa+1):
            before += k_parmi_n[s-1][c-1]

    # on compte pour chaque préfixe l' les suites qui divergent
    # à partir de l'
    for i in range(len(l)):
        if capa < l[i]:
            return -1
        restant = len(l)-1-i
        if restant == 0:
        before += l[i]-1 # cas spécial de la fin du mot
        # car k_parmi_n n'est pas défini pour les indices < 0
        else:
            # on fait une disjonction selon la valeur que l'on
            #  pose pour obtenir une suite plus petite
            for pose in range(1,l[i]):
                # on compte toutes les suites de capacité autorisée
                for capa2 in range(restant,capa-pose+1):
                    before += k_parmi_n[restant-1][capa2-1]
        capa -= l[i]
    return before

b5 = B(2,4,13)
b6 = B(3,6,17)
b7 = B(3,8,41)
b8 = B(5,8,97)

print("Q7 : "+str(index_size_lexi(b5,7))
         +" "+str(index_size_lexi(b6,15))
         +" "+str(index_size_lexi(b7,20))
         +" "+str(index_size_lexi(b8,25)))
```

Question orale 5. Si on place toutes les suites selon l'arbre décrit alors on constate que le premier ordre correspond à un parcours en profondeur d'abord (DFS) tandis que le second correspond à un parcours en largeur d'abord (BFS).

D'après nos calculs, il y a 2^n suites de capacité n. Notre algorithme pour la question 6 fait un nombre linéaire d'opérations, la plus coûteuse d'entre elle étant de mettre 2 à la puissance $k \leq n$. Pour $n = 25$, 2^k est petit et donc cela reste sur un mot mémoire (et donc calcul en temps $O(1)$). Au total, notre algorithme fait de l'ordre de $O(n)$ opérations c'est à dire peu.

Pour la question 7, notre algorithme itère sur chaque position I dans l puis sur chaque valeur inférieure à $l[i]$ puis sur chaque capacité restante. Le nombre d'itérations des deux premières boucles réunies (`for i` et `for pose`) est exactement la capacité de l. La troisième boucle (`for capa2`) est aussi bornée par la capacité. Notre algorithme fait donc au plus un nombre quadratique d'appels à $\binom{n}{k}$. Si $n = 25$, ces nombres restent sur la taille d'un mot mémoire et on peut les précalculer en temps $O(n^2)$ ce qui nous donne un algorithme en $O(n^2)$.

Question 8. Pour Q8, on commence par implémenter les fonctions pour créer les listes :

```
def somme(l1,l2):
    return [l1[i]+l2[i] for i in range(len(l1))]

def diff(l1,l2):
    return [l1[i]-l2[i] for i in range(len(l1))]

def filtre(l):
    return [max(1,e) for e in l]

def universelle(n):
    if n == 0:
        return []

    return universelle(n//2)+[n]+universelle(n-(n//2)-1)

b9 = filtre(
    somme(universelle(16),
          diff(
              B(8,16,13),
              B(10,16,14))))

b10 = filtre(
    somme(universelle(67),
          diff(
              B(13,67,15),
              B(18,67,16))))

b11 = filtre(
    somme(universelle(678),
```

```
            diff(
                B(21,678,17),
                B(24,678,18))))

b12 = filtre(
    somme(universelle(987),
            diff(
                B(72,987,19),
                B(68,987,20))))
```

Ensuite on teste l'universalité pour n par programmation dynamique. L'idée c'est de calculer pour chaque suffixe s de l pour quel n maximal$s$ est n-universelle.

```
def max_universelle(l):
    suffixe_univ = [0]*(len(l)+1)
    length = len(l)
    for pos in range(len(l)-1,-1,-1):
        # La variable univ_for va contenir la valeur maximale
        # pour laquelle le suffixe courant est n-universel
        # Initialement, on remarque le suffixe de longeur
        # length-pos est au plus universel pour length-pos
        univ_for = length-pos


        # Le suffixe est n-universel si et seulement si on peut
        # dominer chaque suite s_1 ... s_k.  Ce critère peut se
        # réécrire de la façon suivante : pour chaque
        # 1 <= s_1 <= n alors il existe une position p telle
        # que l[p]>= s_1 puis que le suffixe de l en p+1 soit
        # (n-s_1) universel.

        # ``ou_poser'' va contenir le plus petit indice supérieur
        # à pos tel que l[ou_poser] >= pose.
        ou_poser = pos
        pose = 1


        while pose <= univ_for: # on teste toutes les valeurs

            #on met à jour ou_poser
            while ou_poser < len(l) and pose>l[ou_poser]:
                ou_poser += 1

            # si ou_poser == len(l) cela veut dire que l'on a pas
            # trouvé et qu'il n'existe pas dans la suite de valeur
            # >= à pose donc univ_for = pose-1 au plus
            if ou_poser == len(l):
                univ_for = pose-1
```

```
            else:
                # on doit maintenant vérifier qu'après l'endroit
                # où ``pose'' est placé on ait bien un suffixe
                # (univ_for-pose)-universel
                # Si non, on en déduit une nouvelle borne
                # pour univ_for
                univ_for = min(univ_for,
                               suffixe_univ[ou_poser+1]+pose)
            pose += 1

        # une fois que la boucle while termine (pose>univ_for)
        # alors on s'est assuré que univ_for était plus grand que
        # la valeur qu'on cherchait.
        # Mais comme la boucle while a tourné pour toutes les valeurs
        # de pose <= univ_for alors on sait que le suffixe est bien
        # univ_for universel.
        suffixe_univ[pos]=univ_for
    return suffixe_univ[0]

print("Q8 : "+str(max_universelle(b9))
         +" "+str(max_universelle(b10))
         +" "+str(max_universelle(b11))
         +" "+str(max_universelle(b12)))
```

Question orale 6. L'algorithme est présenté ci-haut. On peut constater qu'à tour la première boucle on fait des itérations dans les sous-boucles au plus n fois. Au total l'algorithme est donc quadratique en temps et linéaire en mémoire.

Question 9. L'idée générale pour cette question est de regarder la relation récursive de $nbDomin(i_1 :: l_1, i_2 :: l_2)$. On a

$$nbDomin(i_1 :: l_1, i_2 :: l_2) = \begin{cases} nbDomin(l_1, i_2 :: l_2) & \text{quand } i_1 < i_2 \\ nbDomin(l_1, i_2 :: l_2) + nbDomin(l_1, l_2) & \text{quand } i_1 \geq i_2 \end{cases}$$

Voici trois manières d'implémenter cette relation. En version récursive avec mémorisation et *sys.setrecursionlimit(50000)* pour autoriser une récursion avec plus de niveaux :

```
import sys
sys.setrecursionlimit(50000)

def nb_domin_rec(l1,l2):
    memo = [[None for _ in l2] for _ in l1]
    modulo = 10**6

    def nb(i,j):
        if j==len(l2):
            return 1
```

```
        if i==len(l1):
            return 0

        if memo[i][j] == None:
            memo[i][j] = nb(i+1,j)
            if l1[i]>=l2[j]:
                memo[i][j] += nb(i+1,j+1)
            memo[i][j] %= modulo

        return memo[i][j]


    return nb(0,0)
```

Ou en version itérative :

```
def nb_domin_dyn(l1,l2):
    nb = [[0 for _ in range(1+len(l2))] for _ in range(1+len(l1))]
    modulo = 10**6

    nb[0][0]=1
    for j in range(len(l2)):
        for i in range(len(l1)):
            if l1[i] >= l2[j]:
                nb[i+1][j+1] = (nb[i+1][j+1]+nb[i][j])%modulo
            nb[i+1][j] = (nb[i+1][j]+nb[i][j])%modulo


    return nb[len(l1)][len(l2)]
```

Ou en version itérative où l'on optimise l'espace :

```
def nb_domin_fast(l1,l2):

    cur = [ 0 for _ in range(1+len(l1))]
    modulo = 10**6

    cur[0]=1
    for v in l2:
        nxt = [ 0 for _ in cur]
        for i in range(len(l1)):
            if l1[i] >= v:
                nxt[i+1] = cur[i]
            cur[i+1] = (cur[i]+cur[i+1])%modulo
        cur = nxt

    return sum(cur)%modulo
```

Question orale 7. L'algorithme est présenté ci-haut. Dans la meilleure des trois versions l'algorithme est en $O(|l_1| \times |l_2|)$ en temps et en $O(|l_1|)$ en mémoire.

PROPOSITION DE SOLUTION POUR *Sujet 3 : Suites Universelles* EN OCAML

Question 1.

```
let m = 2147483647
let a = 46613
let b = 17
let f x = (a * x + b) mod m

let create_tab n =
let t = Array.make n u0 in
for i = 1 to n-1 do
        t.(i) <- f (t.(i-1))
done ;
t ;;

let tab = create_tab 10000001 ;;

printf "1.a %d\n" (tab.(1) mod 1000) ;;
printf "1.b %d\n" (tab.(98) mod 1000) ;;
printf "1.c %d\n" (tab.(9876) mod 1000) ;;
```

Question 2. On utilise ici des listes.

```
let list_b n k alpha =
let rec boucle p =
        if p == k then []
        else ((tab.(alpha * p) mod n) + 1)::(boucle (p+1))
in boucle 0 ;;

let list_b1 = list_b 9 5 13 ;;
let list_b2 = list_b 98 54 17 ;;
let list_b3 = list_b 987 543 41 ;;
let list_b4 = list_b 98765 54321 97 ;;

let rec capacite l =
  List.fold_left (+) 0 l ;;

printf "2.a %d\n" (capacite list_b1) ;;
printf "2.b %d\n" (capacite list_b2) ;;
printf "2.c %d\n" (capacite list_b3) ;;
printf "2.d %d\n" (capacite list_b4) ;;
```

Question 3. L'utilisation de listes permet une simple et jolie fonction récursive.

```
let ind a l =
let rec boucle l i = match l with
        | [] -> -1
        | h::t -> if a <= h then i else boucle t (i+1)
```

```
in boucle l 0 ;;

printf "3.a %d\n" (ind 9 list_b1) ;;
printf "3.b %d\n" (ind 86 list_b2) ;;
printf "3.c %d\n" (ind 975 list_b3) ;;
printf "3.d %d\n" (ind 98123 list_b4) ;;
```

Question 4. Ici il s'agit de remarquer que l'algorithme glouton est optimal (ce qui était suggéré par la question précédente). On parcourt les deux listes, et à chaque nouvel élément b de ℓ_2 on cherche le prochain élément de ℓ_1 qui est supérieur ou égal à x. Ceci s'écrit récursivement de manière très élégante.

```
let rec ind_l a l = match l with
        | [] -> None
        | h::t -> if a <= h then (Some t) else ind_l a t ;;

let pref_domine l1 l2 =
let rec boucle l1 l2 c = match l2 with
        | [] -> (c+1)
        | h::t -> match ind_l h l1 with
                | None -> c
                | Some l -> boucle l t (c+1)
in boucle l1 l2 0 ;;

let list_c1 = list_b 9 10 14 ;;
let list_c2 = list_b 98 68 43 ;;
let list_c3 = list_b 987 673 174 ;;
let list_c4 = list_b 98765 65431 81 ;;

printf "4.a %d\n" (pref_domine list_b1 list_c1) ;;
printf "4.b %d\n" (pref_domine list_b2 list_c2) ;;
printf "4.c %d\n" (pref_domine list_b3 list_c3) ;;
printf "4.d %d\n" (pref_domine list_b4 list_c4) ;;
```

Question orale 1. À l'oral il fallait au moins dire qu'il y a quelque chose à prouver : on utilise l'algorithme glouton, il se pourrait qu'il ne soit pas optimal. Pour montrer la correction formellement, on procède par induction, en disant que la domination de ℓ_1 par ℓ_2 construite par l'algorithme glouton est minimale parmi toutes les dominations.

Question orale 2. La suite $(1, 2, 4, 3, 1)$ est 4-universelle, ce qui se vérifie par étude de cas. La suite $(1, 2, 4, 1)$ l'est également, et elle est de capacité minimale, ce qui est relativement pénible à montrer, par étude de cas.

Question orale 3. On procède par récurrence sur n. Considérons une suite $\ell = (a_0, \dots, a_{k-1})$ de capacité n. Soit i le plus petit indice tel que $\sum_{j \leq i} a_j \geq \lfloor n/2 \rfloor$. Par hypothèse de récurrence $(a_0, \dots, a_{i-1})$ est dominée par $L_{\lfloor n/2 \rfloor}$. Remarquons que $\sum_{j>i} a_j \leq n - 1 - \lfloor n/2 \rfloor$, donc par hypothèse de récurrence $(a_{i+1}, \dots, a_{k-1})$ est dominée par $L_{n-1-\lfloor n/2 \rfloor}$. Il s'ensuit que ℓ est dominée par L_n.

Il se trouve que L_n est de capacité minimale, mais ce n'est pas du tout évident à montrer. Nous avons apprécié les pistes de preuve, et surtout l'honnêteté de la démarche : "je pense qu'elle est minimale, mais je n'ai pas su le prouver. Voici mes idées".

Notons $f(n)$ la capacité de L_n. On procède par récurrence sur n. Considérons une suite L qui est n-universelle. Nécessairement elle contient un élément n (pour dominer la suite à un seul élément n), notons L_1 le préfixe de L (strictement) avant n, et L_2 le suffixe de L après n. Soit k maximal tel que L_1 est k-universelle. Nous allons vérifier que L_2 est $n-1-k$-universelle. Puisque L_1 n'est pas $(k+1)$-universelle, il existe une suite c de capacité $k+1$ qui n'est pas dominée par L_1. Considérons maintenant une suite ℓ de capacité $n-k-1$, et la suite $c+\ell$ obtenue en concaténant c puis ℓ : c'est une suite de capacité n, donc dominée par L. Puisque c n'est pas dominée par L_1, ceci implique que la domination de $c+\ell$ par L induit la domination de ℓ par L_2. Il s'ensuit que L_2 est bien $n-k-1$-universelle. Par hypothèse de récurrence, on obtient que L_1 est de capacité au moins $f(k)$, et L_2 de longueur au moins $f(n-k-1)$. En optimisant pour k, on obtient $k = \lfloor n/2 \rfloor$, ce qui conclut la preuve.

Question 5. Représenter L_n en entier avant de calculer sa capacité est évidemment une mauvaise idée. Un peu mieux, on peut mémoïser les réponses intermédiaires. La réponse attendue consiste à remplir un tableau de taille $n+1$ contenant la capacité de la suite L_i pour chaque i.

```
let capl n =
let tab = Array.make (n+1) None in
let rec fill x =
match tab.(x) with
        | Some p -> p
    | None -> if x <= 1
            then
        begin
                tab.(x) <- Some x ; x
        end
            else
                    begin
                        let p1 = fill (x/2) in
                    let p2 = fill (x - 1 - (x/2) ) in
                    let p = x + p1 + p2 in
                    (tab.(x) <- Some p ; p)
                end
in fill n ;;

printf "5.a %d\n" (capl 6) ;;
printf "5.b %d\n" (capl 987) ;;
printf "5.c %d\n" (capl 98765) ;;
printf "5.d %d\n" (capl 9876543) ;;
```

Question orale 4. La complexité est linéaire en n.

Question 6. La difficulté, si l'on suit l'indication donnée par l'énoncé, est d'écrire la fonction qui passe d'une liste à la suivante. Ici, c'est assez comfortable d'utiliser des listes, et de représenter les listes depuis la fin. Notons qu'il y a d'autres approches (voir le corrigé en Python).

```
let rec eq l1 l2 = match (l1,l2) with
        | [],[] -> true
        | h1::t1,h2::t2 -> h1 == h2 && eq t1 t2
        | _ -> false ;;

exception Not_found ;;

let dfs n target =
let c = capacite target in
let rev_target = List.rev target in

let rec boucle l cap nb =
        if cap < n
                then
                        let new_cap = cap + 1 in
                        if (new_cap == c && eq rev_target (1::l))
                                then nb
                                else boucle (1::l) new_cap (nb + 1)
                else
                        match l with
                        | h1::h2::t ->
                                let new_cap = cap + 1 - h1 in
                                if (new_cap == c && eq rev_target ((h2 + 1)::t))
                                        then nb
                                        else boucle ((h2 + 1)::t) new_cap (nb + 1)
                        | _ -> raise Not_found
in
try boucle [] 0 1 with Not_found -> -1 ;;

let list_e1 = list_b 2 4 13 ;;
let list_e2 = list_b 3 6 17 ;;
let list_e3 = list_b 3 8 41 ;;
let list_e4 = list_b 5 8 97 ;;

printf "6.a %d\n" (dfs 7 list_e1) ;;
printf "6.b %d\n" (dfs 15 list_e2) ;;
printf "6.c %d\n" (dfs 20 list_e3) ;;
printf "6.d %d\n" (dfs 25 list_e4) ;;
```

Question 7. C'est pareil que la question précédente, mais la fonction qui passe d'une liste à la suivante est différente. Pareil, c'est assez comfortable d'utiliser des listes, et de représenter les listes depuis la fin.

```
let bfs n target =
let c = capacite target in
let rev_target = List.rev target in

let rec next_l l cap = match l with
        | [] -> raise Not_found
        | h::t -> if (h == 1)
                        then let (new_l,new_cap) = next_l t (cap-1)
                                in (1::new_l, new_cap+1)
                        else match t with
                            | [] -> ([1;1], 2)
                            | h2::t2 -> (1::(h2+1)::t2, cap+2-h)
in

let rec boucle l cap nb =
        if cap < n
                then match l with
                        | [] -> boucle [1] 1 1
                        | h::t -> let new_cap = cap + 1 in
                                let new_l = (h+1)::t in
                        if (new_cap == c && eq rev_target new_l)
                            then (nb+1)
                            else boucle new_l new_cap (nb + 1)
                else
                        let new_l,new_cap = next_l l cap in
                        if (new_cap == c && eq rev_target new_l)
                            then (nb+1)
                            else boucle new_l new_cap (nb + 1)
in
try boucle [] 0 1 with Not_found -> -1 ;;

printf "7.a %d\n" (bfs 7 list_e1) ;;
printf "7.b %d\n" (bfs 15 list_e2) ;;
printf "7.c %d\n" (bfs 20 list_e3) ;;
printf "7.d %d\n" (bfs 25 list_e4) ;;
```

Question orale 5. En considérant l'arbre, les deux questions précédentes sont respectivement un parcours en largeur (question 6), puis en profondeur (question 7).

Notons U_n le nombre de suites de capacité au plus n, alors $U_n = 1 + \sum_{i=0}^{n-1} U_i$: en effet une suite de capacité au plus n est soit vide, soit entièrement déterminée par son premier élément $i \in [0, n]$ et une suite de capacité au plus $n - i$. De plus $U_0 = 1$, d'où $U_n = 2^n$.

Pour $n = 25$, on obtient $2^{25} = 10^{25 \cdot \log_{10}(2)} \leq 10^8$. On s'attend donc à ce que l'algorithme termine pour $n = 25$.

Question 8. Notons $\text{UNIV}(i)$ le plus grand m tel que $(a_i, \ldots, a_{k-1})$ est m-universel. Par convention $\text{UNIV}(n) = 0$. Alors

$$\text{UNIV}(i) = \begin{cases} \text{UNIV}(i+1) + 1 & \forall p \in [2, \text{UNIV}(i+1) + 1],\ \exists j \geq i, \\ & a_j \geq p \text{ et } \text{UNIV}(j+1) \geq \text{UNIV}(i+1) + 1 - p \\ \text{UNIV}(i+1) & \text{sinon.} \end{cases}$$

Ceci donne immédiatement un algorithme de programmation dynamique.

La fonction principale s'écrit comme suit.

```
let universel t =
let len = Array.length t in
let tab = Array.make (len+1) 0 in
let flag = ref true and p = ref 0 and j = ref 0 in
for i = len-1 downto 0 do
        flag := true ;
        p := 2 ;
        while(!flag && !p <= tab.(i+1) + 1) do
                j := i ;
                while(!j < len && (t.(!j) < !p ||
                tab.(!j+1) < tab.(i+1) +1 - !p)) do
                        j := !j + 1 ;
                done ;
                flag := !j < len ;
                p := !p + 1 ;
        done ;
        if !flag then tab.(i) <- tab.(i+1) + 1
        else tab.(i) <- tab.(i+1) ;
done ;
tab.(0) ;;
```

Le reste du code consiste à construire les entrées demandées.

```
let rec tab_b n k alpha =
Array.init k (fun x -> (tab.(alpha * x) mod n) + 1) ;;

let tab_l n =
let ln = l n in
let len = List.length ln in
let tab = Array.make len 0 in
List.iteri (fun i x -> tab.(i) <- x) ln ;
tab ;;

let sum tab1 tab2 =
Array.init (Array.length tab1) (fun i -> tab1.(i) + tab2.(i)) ;;
let diff tab1 tab2 =
Array.init (Array.length tab1) (fun i -> tab1.(i) - tab2.(i)) ;;
let filter_neg tab =
Array.iteri (fun i x -> if x < 1 then tab.(i) <- 1
```

```
else tab.(i) <- x) tab ; tab ;;

let tab_f1 = filter_neg (diff (sum (tab_l 16)
(tab_b 8 16 13)) (tab_b 10 16 14)) ;;
let tab_f2 = filter_neg (diff (sum (tab_l 67)
(tab_b 13 67 15)) (tab_b 18 67 16)) ;;
let tab_f3 = filter_neg (diff (sum (tab_l 678)
(tab_b 21 678 17)) (tab_b 24 678 18)) ;;
let tab_f4 = filter_neg (diff (sum (tab_l 987)
(tab_b 72 987 19)) (tab_b 68 987 20)) ;;

printf "8.a %d\n" (universel tab_f1) ;;
printf "8.b %d\n" (universel tab_f2) ;;
printf "8.c %d\n" (universel tab_f3) ;;
printf "8.d %d\n" (universel tab_f4) ;;
```

Question orale 6. L'implémentation ci-dessus est cubique.

Question 9. Notons $\textsc{domin}(i, j)$ le nombre de facons que $(a_i, \ldots, a_{k-1})$ domine $(b_j, \ldots, b_{p-1})$. Alors

$\textsc{domin}(i, j) = \textsc{domin}(i + 1, j) + (\text{si } b_j \leq a_i \text{ alors } \textsc{domin}(i + 1, j + 1) \text{ sinon } 0)$.

La réponse finale est $\textsc{domin}(0, 0)$.

```
let domine_all t1 t2 =
let n1 = Array.length t1 and n2 = Array.length t2 in
let tab = Array.init n1 (fun _ -> Array.init n2 (fun _ -> 0)) in

tab.(n1 - 1).(n2 - 1) <- if t2.(n2 - 1) <= t1.(n1 - 1) then 1 else 0 ;

for i = n1 - 2 downto 0 do
        tab.(i).(n2 - 1) <- (tab.(i+1).(n2 - 1) +
        (if t2.(n2 - 1) <= t1.(i) then 1 else 0))
        mod 1000000 ;
done ;

for i = n1 - 2 downto 0 do
        for j = n2 - 2 downto 0 do
                tab.(i).(j) <- (tab.(i+1).(j) +
                (if t2.(j) <= t1.(i) then tab.(i+1).(j+1) else 0))
                mod 1000000 ;
        done ;
done ;

tab.(0).(0) ;;
```

Il ne reste plus qu'à construire les entrées demandées.

```
let tab_b1 = tab_b 9 5 13 ;;
let tab_b2 = tab_b 98 54 17 ;;
let tab_b3 = tab_b 987 543 41 ;;
let tab_b4 = tab_b 98765 5432 97 ;;

let tab_c1 = tab_b 5 3 42 ;;
let tab_c2 = tab_b 69 23 43 ;;
let tab_c3 = tab_b 789 145 174 ;;
let tab_c4 = tab_b 56789 1234 81 ;;

printf "9.a %d\n" ((domine_all tab_b1 tab_c1) mod 1000000) ;;
printf "9.b %d\n" ((domine_all tab_b2 tab_c2) mod 1000000) ;;
printf "9.c %d\n" ((domine_all tab_b3 tab_c3) mod 1000000) ;;
printf "9.d %d\n" ((domine_all tab_b4 tab_c4) mod 1000000) ;;
```

Question orale 7. La complexité est $O(|\ell_1| \cdot |\ell_2|)$.

Proposition de solution pour *Sujet 4 : Chasse au trésor sur graphes* en Python

De manière intéressante dans ce sujet, il était utile d'utiliser des stratégies simples à écrire, même si elles ont des complexités asymptotiquement très grandes, car les valeurs demandées sont souvent relativement faibles.

Question 1. On peut écrire le générateur de nombres de la façon suivante :

```
u0 = 42

maxn = 1000000
_u = [0] * maxn
_u[0] = u0
for i in range(1,maxn):
    _u[i] = (19999999 * _u[i-1]) % 19999981
```

Les valeurs de u_n s'obtiennent alors directement :

```
def u(n):
    return _u[n]

print("1.a %d" % (u(123) % 1000))
print("1.b %d" % (u(456) % 1000))
print("1.c %d" % (u(789) % 1000))
```

Question 2. On peut commencer par calculer le nombre d'arêtes directement :

```
def v(t, p):
    if u(t) % 10000 < p:
        return 1
    else:
        return 0

def nbaretes(n, p):
    s = 0
    for i in range((n-1)*n//2):
        s += v(i, p)
    return s

print("2.a %d" % (nbaretes(10,654)))
print("2.b %d" % (nbaretes(100,543)))
print("2.c %d" % (nbaretes(1000,12)))
```

Question orale 1. Le nombre d'arêtes étant faible, on préfèrera utiliser des listes d'adjacence pour représenter les graphes, et l'on peut vérifier que l'on obtient le même résultat :

```
def gnp(n,p):
    g = [ [] for i in range(n) ]
```

```
    t = 0
    for i in range(n):
        for j in range(i+1,n):
            if v(t, p):
                g[i].append(j)
                g[j].append(i)
            t += 1
    return g

def nbaretes2(g):
    s = 0
    for i in g:
        s += len(i)
    return s

g1 = gnp(10,654)
g2 = gnp(100,543)
g3 = gnp(1000,12)
print("2.a %d" % (nbaretes2(gnp(10,654))))
print("2.b %d" % (nbaretes2(gnp(100,543))))
print("2.c %d" % (nbaretes2(gnp(1000,12))))
```

Question 3. Le calcul de la composante connexe contenant le sommet 0 se fait classiquement avec une file et un marquage.

```
def nbconnexe0(g):
    # Marquage des sommets
    m = [ False ] * len(g)

    m[0] = True
    cur = [ 0 ]
    n = 1
    while cur != []:
        s = cur.pop()
        for t in g[s]:
            if not m[t]:
                n += 1
                m[t] = True
                cur.append(t)
    return n

print("3.a %d" % (nbconnexe0(g1)))
print("3.b %d" % (nbconnexe0(g2)))
print("3.c %d" % (nbconnexe0(g3)))
```

Question orale 2. Il s'agit d'un parcours de graphe classique linéaire. Le marquage assure en effet que l'on parcours chaque sommet au plus une fois, et donc chaque sommet ne passe dans la file qu'une fois, et l'on ne parcours ses arêtes adjacentes

qu'au plus une fois. Chaque arête du graphe est ainsi parcourue au plus deux fois. On obtient ainsi une complexité $\mathcal{O}(n+m)$.

Question 4. Le calcul des différentes composantes connexes s'effectue de la même manière. Il faut par contre prendre garde à mémoriser le sommet de départ précédemment utilisé pour éviter une complexité quadratique juste pour trouver un sommet non encore marqué.

```
def nbconnexes(g):
    # Marquage des sommets
    m = [ False ] * len(g)

    def nonmarque(start):
        """Trouver un sommet non marqué à partir du sommet
        'start'"""
        for i in range(start,len(m)):
            if not m[i]:
                return i
        return None

    def marqueconnexe(start):
        """Marquer la composante connexe contenant 'start'"""
        m[start] = True
        cur = [ start ]
        while cur != []:
            s = cur.pop()
            for t in g[s]:
                if not m[t]:
                    m[t] = True
                    cur.append(t)

    n = 0
    start = 0
    while start != None:
        n += 1
        marqueconnexe(start)
        start = nonmarque(start+1)
    return n

print("4.a %d" % (nbconnexes(g1)))
print("4.b %d" % (nbconnexes(g2)))
print("4.c %d" % (nbconnexes(g3)))
```

Question orale 3. C'est le même parcours de graphe que la question précédente, sauf que l'on repart d'un sommet non marqué à chaque tour de la boucle principale. Les arguments de la question orale 2 restent valables pour la fonction `marqueconnexe`. Pour la fonction `nonmarque`, puisqu'elle est appelée à chaque fois avec le sommet que l'on avait obtenu précédemment, dans l'ensemble sa boucle `for` parcourra chaque sommet au plus une fois, dans l'ordre.

Question 5. L'ajout des arêtes pour rendre les graphes connexe peut s'effectuer ainsi :

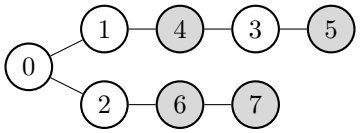
```
def gnp2(n,p):
    g = gnp(n,p)
    for i in range(len(g)-1):
        if i+1 not in g[i]:
            g[i].append(i+1)
            g[i+1].append(i)
    return g
```

L'algorithme glouton peut s'écrire ainsi : on effectue un parcours en largeur jusqu'à trouver des pièces. On conserve alors le sommet de plus petit indice portant une pièce. On itère ainsi autant de fois qu'il y a de pièces à trouver.

```
def glouton(g, k):
    n = len(g)
    # Pièces trouvées
    pieces = [ False ] * k

    def trouvepiece(s):
        """Trouve la pièce la plus proche de s
        retourne le nombre de pas et le sommet d'arrivée"""
        m = [ False ] * n
        m[s] = True
        cur = [s]
        small_steps = 0
        min_piece = n
        while cur != []:
            small_steps += 1
            newcur = []
            for s in cur:
                for t in g[s]:
                    if m[t]:
                        # Sommet déjà visité
                        continue
                    if t >= n-k and not pieces[t - (n-k)]:
                        # On a trouvé une nouvelle pièce !
                        if t < min_piece:
                            min_piece = t
                    else:
                        # Il n'y a pas de pièce
                        m[t] = True
                        newcur.append(t)
            if min_piece < n:
                # On a trouvé au moins une pièce, on y va
                pieces[min_piece-(n-k)] = True
                return small_steps, min_piece
            cur = newcur
```

```
        # Toutes les pièces sont atteignables
        assert(False)

    start = 0
    steps = 0
    for trouve in range(k):
        small_steps,start = trouvepiece(start)
        steps += small_steps

    return steps

g1 = gnp2(100,32)
g2 = gnp2(200,21)
g3 = gnp2(1000,1)
print("5.a %d" % (glouton(g1,10)))
print("5.b %d" % (glouton(g2,15)))
print("5.c %d" % (glouton(g3,20)))
```

Question orale 4. Avec le graphe suivant comportant 4 pièces :



L'algorithme glouton commencera par atteindre la pièce du sommet 4, puis celle du sommet 5, avant de revenir vers celle du sommet 6 puis celle du sommet 7. En partant plutôt d'abord du côté des sommets 6 et 7, on économise un mouvement.

Question 6. Il est utile de commencer par calculer les distances dans le graphe, de manière classique par parcours en largeur depuis chaque sommet.

Il existe d'autres algorithmes pour calculer les distances, mais ici le nombre d'arêtes est petit, un simple parcours de graphe est donc très efficace.

Par ailleurs, pour cette question 6 il n'est pas utile de calculer toutes les distances, seules les distances depuis le sommet 0 et depuis les sommets portant des pièces sont utiles. Pour la question suivante le calcul de toutes les distances sera cependant utile, sans qu'il soit en pratique contraignant en temps ici.

```
def distances(g):
    n = len(g)

    dist = [ [ None for i in range(n) ] for j in range(n) ]

    def distance(orig):
        m = [False] * n
        m[orig] = True
        cur = [orig]
        d = 0
        while cur != []:
```

```
            d += 1
            newcur = []
            for s in cur:
                for t in g[s]:
                    if m[t]:
                        # déjà visité
                        continue
                    dist[orig][t] = d
                    m[t] = True
                    newcur.append(t)
            cur = newcur

    for s in range(n):
        distance(s)

    return dist
```

On peut alors calculer le nombre optimal de mouvement par force brute, mais avec mémoisation. On peut en effet stocker, pour un sommet de départ donné et un ensemble de pièces déjà collectées, le nombre de mouvements nécessaires pour collecter le reste des pièces. On encode ici l'ensemble des pièces déjà collectées en binaire.

```
def optim(g, k):
    n = len(g)
    dist = distances(g)

    t = [ [ None ] * (1<<k) for i in range(k) ]
    for i in range(k):
        # Cas de base: on a collecté toutes les pièces
        t[i][(1<<k)-1] = 0

    def parcours_depuis(s,i):
        """Calcule le nombre optimal de mouvements à partir du
        sommet 's' et en ayant déjà collectés les pièces encodées
        dans 'i'"""
        # mémoisation
        if s != 0 and t[s-(n-k)][i] != None:
            return t[s-(n-k)][i]

        dmin = None
        for s2 in range(k):
            if (i & (1<<s2)) == 0:
                # Pièce non encore collectée, essayons d'aller la
                # chercher
                d = parcours_depuis(n-k+s2, i | (1<<s2))
                d += dist[s][n-k+s2]
                if dmin == None or d < dmin:
```

```
                    dmin = d
        if s != 0:
            t[s-(n-k)][i] = dmin
        return dmin

    dmin = parcours_depuis(0,0)
    return dmin

print("6.a %s" % (str(optim(g1,8))))
print("6.b %s" % (str(optim(g2,10))))
print("6.c %s" % (str(optim(g3,12))))
```

Question orale 5. Pour le calcul des distances, on utilise un parcours de graphe en profondeur, dont on a montré plus haut qu'il était en $\mathcal{O}(n+m)$. On effectue ce parcours de graphe pour chaque sommet de départ, la complexité totale de calcul des distances est ainsi $\mathcal{O}(n.(n+m))$. Si l'on n'avait calculé que les distances vraiment nécessaires, on aurait une complexité $\mathcal{O}(k.(n+m))$.

Le calcul du nombre optimal de mouvements (en fait il s'agit d'un calcul de voyageur de commerce) utilise une mémoisation. Il y a en tout $k.2^k$ valeurs dans le tableau de mémoisation, et pour chaque valeur une boucle `for` est parcourue k fois. Sa complexité est donc $\mathcal{O}(k^2.2^k)$, ce qui peut paraître énorme, mais k est petit. Cet algorithme est en pratique suffisant pour obtenir les réponses rapidement. Si l'on n'utilisait pas de mémoisation, on aurait typiquement une complexité en $\mathcal{O}(k!)$ pour tester toutes les combinaisons possibles.

La complexité totale est ainsi $\mathcal{O}(n.(n+m)+k^2.2^k)$.

Question 7. La répartition des pièces étant maintenant variable, le nombre de cas à traiter peut paraître beaucoup plus grand. Le sujet ne demande cependant que des cas où $k=4$ et n reste petit. On peut ainsi se contenter d'ajouter à l'algorithme précédent 4 boucles imbriquées pour placer les pièces, et il se trouve que le temps de calcul reste raisonnable.

```
def optim_quelconque(g, k):
    n = len(g)
    dist = distances(g)
    dmax = 0

    assert(k==4)
    for s1 in range(1,n):
     for s2 in range(s1+1,n):
      for s3 in range(s2+1,n):
       for s4 in range(s3+1,n):
        l = [s1,s2,s3,s4]

        # On mémoise comme précédemment
        t = [ [ None ] * (1<<k) for i in range(k) ]
        for i in range(k):
            # Cas de base: on a collecté toutes les pièces
```

```
            t[i][(1<<k)-1] = 0

        def parcours_depuis(s,i):
        """Calcule le nombre optimal de mouvements à partir du
        sommet 's' et en ayant déjà collectés les pièces encodées
        dans 'i'"""
            # mémoisation
            if s != 0 and t[l.index(s)][i] != None:
                return t[l.index(s)][i]

            dmin = None
            for s2 in range(k):
                if (i & (1<<s2)) == 0:
                    # Pièce non encore collectée, essayons d'aller
                    # la chercher
                    d = parcours_depuis(l[s2], i | (1<<s2))
                    d += dist[s][l[s2]]
                    if dmin == None or d < dmin:
                        dmin = d
            if s != 0:
                t[l.index(s)][i] = dmin
            return dmin

        dmin = parcours_depuis(0,0)
        if dmin > dmax:
            dmax = dmin
    return dmax

g1 = gnp2(10,654)
g2 = gnp2(20,543)
g3 = gnp2(50,432)
print("7.a %s" % (str(optim_quelconque(g1,4))))
print("7.b %s" % (str(optim_quelconque(g2,4))))
print("7.c %s" % (str(optim_quelconque(g3,4))))
```

Question orale 6. Il s'agit essentiellement du même algorithme que précédemment. On calcule une seule fois les distances, mais l'on effectue le calcul de voyageur du commerce pour chaque répartition de pièces possible. Il y a $\mathcal{O}(n^k)$ répartitions possibles, la complexité totale est ainsi $\mathcal{O}(n.(n+m)+n^k.k^2.2^k)$.

Question 8. Les résultats indiqués sur la feuille-réponse pour $\widetilde{u_0}$ sont plutôt petits, il est donc raisonnable d'essayer une approche par force brute, qui se trouve apporter effectiver des résultats rapidement.

```
def kmax(g,d):
    k = 1
    while True:
        dmin = optim(g,k)
```

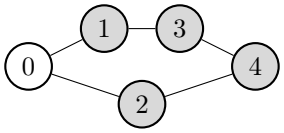
```
        if dmin > d:
            return k-1
        k += 1

print("8.a %s" % (str(kmax(g1,6))))
print("8.b %s" % (str(kmax(g2,8))))
print("8.c %s" % (str(kmax(g3,10))))
```

Question 9. On peut effectivement maintenir un chemin durant la stratégie gloutonne. Insérer un élément dans une liste Python est coûteux, mais les résultats indiqués sur la feuille-réponse pour $\widetilde{u_0}$ sont relativement petits, ce n'est donc en pratique pas gênant.

```
def kmax_glouton(g,dlim):
    n = len(g)
    dist = distances(g)

    # Chemin initial
    l = [0]
    k = 0
    d = 0
    while True:
        k += 1
        if k == n:
            return k
        dmin = None
        for i in range(len(l)-1):
            dmin2 = d + dist[l[i]][n-k] \
                      + dist[n-k][l[i+1]] \
                      - dist[l[i]][l[i+1]]
            if dmin == None or dmin2 < dmin:
                dmin = dmin2
                ii = i+1

        # essayer à la fin aussi
        dmin2 = d + dist[l[-1]][n-k]
        if dmin == None or dmin2 < dmin:
            dmin = dmin2
            ii = len(l)

        l.insert(ii, n-k)
        if dmin > dlim:
            # On dépasse la limite avec ce k-là
            return k-1
        d = dmin

g1 = gnp2(100,32)
```

```
g2 = gnp2(200,21)
g3 = gnp2(1000,1)
print("9.a %s" % (str(kmax_glouton(g1,40))))
print("9.b %s" % (str(kmax_glouton(g2,100))))
print("9.c %s" % (str(kmax_glouton(g3,500))))
```

Question orale 7. Avec le graphe suivant comportant 4 pièces :



L'algorithme glouton commence avec le chemin [0], ajoute le parcours de la pièce 4 et obtient le chemin [0,4] nécessitant 2 mouvements. Pour ajouter la pièce 3, l'algorithme a le choix entre les chemins [0,3,4] et [0,4,3], tous deux nécessitant 3 mouvements. La convention donnée dans le sujet le fait choisir le premier. Pour ajouter la pièce 2, le meilleur choix est le chemin [0,2,3,4] nécessitant 4 mouvements, alors que le chemin [0,2,4,3] ne nécessite que 3 mouvements.