

Introduction

It is much more rewarding to do more with less.
Donald Knuth

Le recueil 2025 comporte trois parties. La première contient les épreuves d'informatique de l'enseignement commun à toutes nos filières hors MPI. La deuxième contient les épreuves de MPI et de l'option informatique de MP. Enfin, la troisième contient les épreuves des CAPES et agrégation d'informatique.

Le bulletin des concours Informatique n'est proposé aux adhérents que sous forme électronique. Dans ce recueil, le nom du fichier contenant chaque sujet figure dans le sommaire et en tête de chaque page. Les fichiers sont disponibles sur le site de l'UPS à l'adresse <https://ups-cpge.fr/?module=Sujets&voir=accueil>. Ce bulletin 2025 et les précédents sont à votre disposition sur le site de l'UPS à l'adresse <https://ups-cpge.fr/?rubrique=7>.

Sébastien Balny
sebastien.balny@prepas.org

Septembre 2025

**ECOLE POLYTECHNIQUE - ESPCI
ECOLES NORMALES SUPERIEURES**

CONCOURS D'ADMISSION 2025

**JEUDI 17 AVRIL 2025
16h30 - 18h30**

**FILIERES MP-PC-PSI
Epreuve n° 8
INFORMATIQUE B (XELSR)**

Durée : 2 heures

***L'utilisation des calculatrices n'est pas
autorisée pour cette épreuve***

Le jeu de Röckse

On dispose sur les cases d'une grille $N \times N$ des *pénalités* et des gains comptés comme des pénalités négatives. Le jeu de Röckse débute à la case $(0, 0)$ et cherche un chemin vers la case $(N - 1, N - 1)$ qui *minimise les pénalités*. À chaque étape du chemin, un nombre fini de déplacements (*sauts*) est autorisé. Des *cases bonus* ajoutent, une fois atteintes, des sauts possibles pour la suite du chemin. La figure ci-dessous donne un exemple de grille et deux chemins possibles, en supposant que les sauts autorisés sur la grille sont $(i, j) \rightarrow (i, j + 1)$ (\rightarrow), $(i, j) \rightarrow (i + 1, j - 1)$ (\swarrow) et $(i, j) \rightarrow (i + 1, j + 1)$ (\searrow). On suppose par ailleurs une case bonus en $(1, 2)$ (grisée) qui, une fois atteinte, ajoute aux sauts autorisés $(i, j) \rightarrow (i + 1, j)$ (\downarrow) :

		0	1	2	3
		DÉPART	-4	-6	0
0	2				
1	1	-2	2 ^(\downarrow)	3	
2	-2	2	-3	4	
3	-1	4	-3	7	ARRIVÉE

(a) Grille de jeu

		0	1	2	3
		DÉPART	\rightarrow -4	\rightarrow -6	0
0	2				
1	1	-2	\nwarrow	2 ^(\downarrow)	3
2	-2	2	\searrow	-3	4
3	-1	4	-3	\nearrow	7

(b) Chemin non-optimal
poids : -6

		0	1	2	3
		DÉPART	\rightarrow -4	\rightarrow -6	0
0	2				
1	1	-2	\nwarrow	2 ^(\downarrow)	3
2	-2	2	\downarrow	-3	4
3	-1	4	-3	\rightarrow	7

(c) Chemin optimal
poids : -7

Considérons le chemin décrit en (c). On part de la case de départ $(0, 0)$ et, en appliquant les sauts successifs $\rightarrow, \rightarrow, \swarrow, \rightarrow$, on arrive sur la case bonus. À partir de cette case, le saut \downarrow est désormais autorisé. On applique ensuite les sauts $\downarrow, \downarrow, \rightarrow$ pour atteindre la case d'arrivée $(3, 3)$. Le chemin obtenu est décrit par la liste des cases :

```
chemin = [(0,0),(0,1),(0,2),(1,1),(1,2),(2,2),(3,2),(3,3)]
```

Son *poids* est la somme des pénalités contenues dans ces cases, soit -7. On peut montrer que c'est un chemin de poids minimal — on dit qu'il est *optimal*. Le chemin décrit en (b) est correct, mais son poids est de -6. Il n'est donc pas optimal.

Représentation des données. La grille de jeu $N \times N$ est représentée par une liste de listes d'entiers T telle que $T[i][j]$ est la pénalité à la case (i, j) . On suppose que cette représentation est bien formée, c'est-à-dire que toutes les listes ont la même taille N . Sur notre exemple :

```
T = [[2,-4,-6,0],[1,-2,2,3],[-2,2,-3,4],[-1,4,-3,7]]
```

Un saut $(i, j) \rightarrow (i + \delta i, j + \delta j)$ est représenté par le couple d'entiers $(\delta i, \delta j)$. L'ensemble des sauts possibles est ainsi une liste de couples. Sur notre exemple, l'ensemble des sauts possibles *par défaut* (avant d'avoir atteint une case bonus) est :

```
sauts = [(0,1),(1,-1),(1,1)]
```

Les sauts activés par les cases bonus sont enregistrés dans un dictionnaire **bonus** tel que **bonus[(i,j)]** est la liste des sauts activés par la case bonus (i, j) . Sur notre exemple :

```
bonus = { (1,2): [(1,0)] }
```

Complexité. La complexité d'une fonction F est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc.) nécessaires à l'exécution de F . Lorsque cette complexité dépend de plusieurs paramètres n et m , on dira que F a une complexité en $O(\phi(n, m))$ lorsqu'il existe trois constantes A , n_0 et m_0 telles que la complexité de F est inférieure ou égale à $A \cdot \phi(n, m)$, pour tout $n \geq n_0$ et $m \geq m_0$. Lorsqu'il est demandé de donner la complexité d'un programme, le candidat devra justifier sa réponse en utilisant le code du programme.

Rappels sur Python. L'utilisation de toute fonction Python sur les listes ou sur les dictionnaires autre que celles mentionnées dans ce paragraphe est interdite. Sur les **listes**, on autorise les opérations suivantes, *dont la complexité est en $O(1)$* :

- `len(l)` renvoie la longueur de la liste `l`.
- `l[i]` désigne l'élément d'indice `i` de la liste `l`, pour $0 \leq i < \text{len}(l)$.
- `l.append(e)` ajoute en place l'élément `e` à la fin de la liste `l`.

Les opérations suivantes sont également autorisées et leur *complexité est en $O(n)$* :

- `l1 + l2` renvoie une nouvelle liste (de longueur n) qui est la concaténation des listes `l1` et `l2`.
- `range(n)` renvoie la liste `[0, 1, ..., n-1]`.
- `range(n-1, -1, -1)` renvoie la liste `[n-1, ..., 0]`.
- `l.pop(0)` retire le premier élément `e` de la liste `l` (de longueur n) et renvoie `e`.
- `(e in l)` renvoie `True` si l'élément `e` est dans la liste `l` (de longueur n), et `False` sinon.
- `l[:]` renvoie une *copie* de la liste `l` (de longueur n).

On autorise également les constructions suivantes :

- La *construction for e in l* parcourt (itère sur) les éléments de la liste `l` du premier élément (d'indice 0), au dernier élément (d'indice `len(l)-1`) avec la complexité $O(\text{len}(l))$.
- La *construction [f(e) for e in l]* produit la même liste que le code suivant, et avec la complexité `len(l)` fois la complexité de `f` :

```
result = []
for e in l:
    result.append(f(e))
return result
```

Sur les **dictionnaires**, on autorise uniquement les opérations suivantes, dont *la complexité est en $O(1)$* :

- Le test `(e in d)` renvoie `True` si `e` est une clé du dictionnaire `d`, et `False` sinon.
- L'accès `d[e]` à l'élément associé à la clé `e` dans le dictionnaire `d`.

On autorise également les constructions suivantes sur les dictionnaires :

- La *construction for (k, v) in d* parcourt les éléments du dictionnaire `d`.
- La *construction d[k] = v* affecte la valeur `v` à la clé `k`.

Enfin, on supposera donnée une variable globale `INFINI` qui contient un entier supérieur ou égal à tout entier utilisé dans les programmes de ce sujet.

Organisation. Les parties peuvent être traitées indépendamment. La partie I porte sur les fonctions de base sur les chemins et les sauts. Ensuite, la partie II propose de trouver un chemin optimal avec une recherche exhaustive. La partie III utilise les résultats de la partie II pour construire une méthode de recherche gloutonne. Enfin, la partie IV étudie une résolution du problème par programmation dynamique. **On rappelle que le code doit être commenté.**

Partie I : Sauts et chemins

Question 1 Écrire une fonction `poids(T,chemin)` qui, étant donné un plateau de jeu `T` et un chemin `chemin`, renvoie le poids de ce chemin. Donner la complexité de cette fonction.

Question 2 Écrire une fonction `appliquer_sauts(i,j,sauts)` qui, étant donné une case (i,j) et une liste de sauts `sauts`, applique les sauts dans l'ordre donné par la liste `sauts`, en partant de la case (i,j) et renvoie la case atteinte. On suppose que le chemin indiqué par `sauts` reste dans la grille.

Question 3 Écrire une fonction `sauts_corrects(sauts,bonus,chemin)` qui, étant donné l'ensemble des sauts par défaut représenté par la liste `sauts`, les sauts associés aux cases `bonus` et un chemin `chemin`, renvoie `True` si les sauts utilisés dans le chemin sont corrects et `False` sinon. On suppose que le chemin reste dans la grille.

On dit qu'un ensemble de sauts C est *bien formé* s'il ne peut pas mener à un cycle. Autrement dit, s'il n'existe pas de suite de sauts construite à partir des sauts de C qui, en partant de la case $(0,0)$ permettrait d'arriver sur une case (i,j) puis de revenir à cette case. Une *condition suffisante* est que chaque saut $(\delta i, \delta j)$ de l'ensemble de sauts C soit strictement positif lexicographiquement, condition notée $(\delta i, \delta j) \gg 0$ et définie par :

$$\delta i > 0 \text{ ou bien } (\delta i = 0 \text{ et } \delta j > 0) \quad (*)$$

Une suite de sauts $\vec{\delta}$ est positive lexicographiquement, propriété notée $\vec{\delta} \gg 0$, si chaque saut $(\delta i, \delta j)$ de $\vec{\delta}$ satisfait $(*)$.

Question 4 Écrire une fonction `sauts_bien_formes(sauts,bonus)` qui, étant donné la liste des sauts par défaut `sauts` et les sauts associés aux cases `bonus`, vérifie que chaque saut de ces listes satisfait la condition $(*)$. La fonction renvoie `True` si c'est le cas et `False` sinon.

Dans le reste du sujet, on suppose que les sauts utilisés satisfont la condition $(*)$.

Partie II : Recherche exhaustive

On cherche maintenant à calculer un chemin optimal. Une première solution est de tester tous les chemins corrects et de retenir le chemin de poids minimum. L'énumération de tous les chemins corrects se fera avec la fonction auxiliaire récursive suivante :

```
trouve_complet_rec(T,sauts,bonus,sauts_max,i,j)
```

Étant donnés une grille de jeu `T`, les sauts par défaut `sauts`, les sauts associés aux cases `bonus` `bonus`, une valeur entière `sauts_max` et une case (i,j) , la fonction ci-dessus calcule un chemin \vec{p} de poids minimum partant de (i,j) et n'arrivant pas forcément à $(N-1, N-1)$, mais ayant au plus `sauts_max+1` cases. La fonction renvoie le couple $(\text{poids_min}, \text{sauts_min})$ où `poids_min` est le poids de \vec{p} et `sauts_min` est la liste des sauts pour construire \vec{p} .

La valeur `sauts_max` est utilisée pour limiter la complexité de la recherche. Ainsi, la longueur du résultat `sauts_min` sera inférieure ou égale à `sauts_max`. Cette limite est appelée *horizon*. Si l'horizon est assez grand, la fonction renvoie un chemin optimal partant de (i,j) .

Question 5 Quelle est la longueur maximale L d'un chemin de $(0,0)$ à $(N-1, N-1)$ pour n'importe quel ensemble de sauts satisfaisant $(*)$? Donner un exemple d'ensemble de sauts par défaut pour lequel se réalise ce chemin de longueur L .

Question 6 Écrire la fonction auxiliaire `trouve_complet_rec(T,sauts,bonus,sauts_max,i,j)` et la fonction principale `trouve_complet(T,sauts,bonus,sauts_max)` qui renvoie le couple (`poids,sauts`) où `poids` est le poids du chemin trouvé et `sauts` est la liste des sauts du chemin trouvé. Quelle est la complexité de cette fonction ?

Question 7 La fonction `trouve_complet_rec` perd beaucoup de temps à refaire les mêmes calculs. On peut l'améliorer en enregistrant les résultats déjà calculés pour une valeur fixée de `sauts_max`. Expliquer en quelques lignes comment procéder. Quelle serait alors la complexité ?

La recherche exhaustive d'un chemin optimal est obtenue en appelant `trouve_complet(T, sauts, bonus, L)` avec `L` la longueur maximale d'un chemin dans `T`.

Partie III : Recherche gloutonne

On peut réduire la complexité de la recherche exhaustive en limitant, à chaque étape, l'horizon de la recherche, c'est-à-dire le nombre de sauts regardés à partir de la case courante. D'où l'idée de construire un algorithme *glouton* pour trouver une solution. En partant de la case $(0, 0)$, on utilise la recherche exhaustive avec un « petit horizon » k pour déterminer la *meilleure suite locale* de k sauts, c'est-à-dire la suite de poids minimal et d'au plus k sauts. On joue les sauts de la meilleure suite locale, puis on recommence sur la case atteinte, jusqu'à la case d'arrivée $(N - 1, N - 1)$. Si la recherche exhaustive avec l'horizon k ne trouve pas une suite locale, alors l'algorithme abandonne la recherche.

Question 8 Sur l'exemple donné en introduction, quel est le chemin trouvé pour $k = 2$? Est-il optimal? Mêmes questions pour $k = 3$. En augmentant k , obtient-on forcément un chemin de poids plus petit?

Question 9 Écrire une fonction `trouve_glouton(T,sauts,bonus,k)` qui, étant donnés la grille de jeu `T`, la liste des sauts par défaut `sauts`, les sauts associés aux cases `bonus` `bonus` et l'horizon `k`, effectue cette recherche gloutonne et renvoie le couple (`poids,sauts`) où `poids` est le poids du chemin trouvé et `sauts` est la liste des sauts du chemin trouvé. Quand la recherche exhaustive avec l'horizon k ne trouve pas de suite locale, la fonction renvoie `(INFINI, [])`.

Partie IV : Recherche par programmation dynamique

On se propose maintenant de construire une méthode par programmation dynamique pour trouver un chemin optimal. Comme la solution optimale en partant de (i, j) dépend des cases bonus déjà rencontrées (dites *activées*), on va remplir un tableau `poids_opt[i][j][code_bonus]` qui contient le poids du chemin optimal en partant de la case (i, j) et où la 3^e dimension (`code_bonus`) encode l'ensemble des cases bonus activées. En parallèle, on remplira un tableau `saut_opt[i][j][code_bonus]` qui contient un saut optimal à jouer en partant de la case (i, j) . Ce tableau permettra de retrouver un chemin optimal.

1) Encodage des cases bonus activées

Soit n le nombre de cases bonus. On numérote les cases bonus de 0 à $n - 1$. On représente les cases bonus activées par une liste de booléens (*masque binaire*) $[b_0, \dots, b_{n-1}]$ où b_k vaut `True` si et seulement si la case bonus k est activée. L'ensemble des cases bonus activées est encodé par l'entier dont la représentation binaire est $\hat{b}_{n-1} \dots \hat{b}_0$, où `True` = 1 et `Faîse` = 0. Ce code est noté $\langle b_0 \dots b_{n-1} \rangle$. Par exemple, le code associé au masque `[False, False]` est 0, le code associé au masque `[True, False]` est 1, etc.

Question 10 Écrire la fonction `poids_opt[masque_bonus]` qui renvoie le code associé au masque binaire `masque_bonus` représentant l'ensemble des cases bonus activées.

2) Récurrence

On va maintenant donner une récurrence pour calculer les valeurs `poids_opt[i][j][code_bonus]` pour tout case (i, j) et valeur de `code_bonus`. Cette récurrence sera utilisée dans la section suivante pour construire un algorithme.

Pour simplifier l'explication, ignorons les cases bonus dans un premier temps. Si un chemin partant de la case (i, j) a un poids minimal, alors le chemin obtenu en lui retirant (i, j) (en partant de la case suivante) a lui-même un poids minimal. Une fois `poids_opt` calculé pour tous les successeurs possibles de la case (i, j) , il suffit de garder le plus petit et de lui ajouter le poids de la case `T[i][j]`. On obtient ainsi `poids_opt` pour la case (i, j) .

Avec les cases bonus, c'est le même principe sauf qu'il faut gérer les sauts supplémentaires des cases bonus activées. Deux cas sont possibles :

- **(i, j) n'est pas une case bonus.** Dans ce cas, il suffit de calculer `poids_opt[i_s][j_s][code_bonus]` pour tous les successeurs possibles (i_s, j_s) de la case (i, j) avec les sauts par défaut et les sauts associés à chaque case bonus activée de `code_bonus`. Le `poids_opt[i][j][code_bonus]` est alors le minimum de ces `poids_opt[i_s][j_s][code_bonus]` auquel s'ajoute la pénalité `T[i][j]` de la case (i, j) .
- **(i, j) est une case bonus.** Dans ce cas, c'est le même processus, sauf que : 1) parmi les sauts possibles, on ajoute ceux activés par la case bonus et 2) on considère les successeurs (i_s, j_s) avec un nouveau code bonus `code_bonus'` dans lequel la case bonus (i, j) est activée : le minimum doit ainsi être calculé parmi les `poids_opt[i_s][j_s][code_bonus']`. Si `code_bonus = \langle b_0 \dots b_{n-1} \rangle`, en notant k le numéro de la case bonus (i, j) , on changera b_k à `True` pour obtenir `code_bonus' = \langle b_0 \dots b'_k \dots b_{n-1} \rangle`, où $b'_k = \text{True}$.

Question 11 Écrire la définition de `poids_opt[i][j][\langle b_0 \dots b_{n-1} \rangle]`. On notera Γ l'ensemble des sauts par défaut et Δ_k l'ensemble des sauts associé à la k -ième case bonus.

3) Algorithme

On évalue itérativement les différentes cases `poids_opt[i][j][code_bonus]` de `poids_opt` à l'aide de trois boucles imbriquées itérant respectivement sur i , j et le masque de bonus (d'où on tirera `code_bonus`). La difficulté est de trouver un ordre d'évaluation correct. À partir de la récurrence, on voit que `poids_opt[i][j][\langle b_0 \dots b_{n-1} \rangle]` doit être évalué après avoir obtenu les poids des successeurs : `poids_opt[i+\delta i][j+\delta j][\langle b'_0 \dots b'_{n-1} \rangle]`.

- Comme $(\delta i, \delta j) \gg 0$ (condition (*)), alors $i + \delta i > i$ ou bien $\delta i = 0$ et $j + \delta j > j$, donc les itérations sur i et j doivent être « à l'envers », de $N - 1$ à 0.
- Soit $[b'_0, \dots, b'_{n-1}]$ est égal à $[b_0, \dots, b_{n-1}]$, soit $[b'_0, \dots, b'_{n-1}]$ est obtenu à partir de $[b_0, \dots, b_{n-1}]$ en mettant un b_k à `True`. En d'autres termes, le choix de bonus représenté par $[b_0, \dots, b_{n-1}]$ est inclus dans le choix de bonus représenté par $[b'_0, \dots, b'_{n-1}]$. La boucle sur les masques de bonus doit donc itérer par *choix de bonus décroissant au sens de l'inclusion*.

Avec 3 cases bonus, un ordre est le suivant :

```
[True, True, True], puis
[False, True, True], [True, False, True], [True, True, False], puis
[False, False, True], [False, True, False], [True, False, False], puis
[False, False, False]
```

Noter que les choix rassemblés sur une ligne ne sont pas classables entre eux. Par contre, les choix d'une ligne sont strictement supérieurs au sens de l'inclusion à l'un des choix de la ligne suivante.

Un algorithme pour calculer l'ordre d'évaluation des masques de bonus peut procéder comme suit. On commence par le choix total, dans notre exemple `[True, True, True]`. On construit la ligne suivante en insérant les choix obtenus en basculant une coordonnée `True` à `False`. Par exemple, on insère `[False, True, True]`, `[True, False, True]`, `[True, True, False]`. On itère ensuite sur chaque choix obtenu pour construire la ligne d'après. On s'arrête lorsqu'on atteint le choix vide, dans notre exemple `[False, False, False]`. On pourra utiliser une *file* pour défiler les choix de bonus à traiter et enfiler progressivement les choix de bonus de la ligne suivante.

Question 12 Écrire une fonction `combinaisons_bonus(nb_bonus)` qui, étant donné le nombre total de cases bonus `nb_bonus`, renvoie la liste de masques bonus ordonnée de manière décroissante dans le sens de l'inclusion, en codant l'algorithme ci-dessus.

On suppose qu'il existe une fonction `ranger_bonus(bonus)` qui renvoie un couple `(bonus_au_rang, rang_du_bonus)` tel que :

- `bonus_au_rang[k]` est la liste des sauts activés par la case bonus dont le numéro est `k`,
- `rang_du_bonus[(i, j)]` est le numéro de la case bonus (i, j) dans un masque de bonus.

Le résultat de cette fonction est utilisé comme paramètre pour la question suivante et pour la fonction `ajouter_bonus` décrite après.

Question 13 Écrire une fonction

```
trouver_sauts_posibles(sauts, bonus_au_rang, masque_bonus)
```

qui, étant donnés les sauts par défaut `sauts`, la liste `bonus_au_rang` renvoyée par `ranger_bonus(bonus)` et le masque des bonus activés `masque_bonus`, renvoie l'ensemble des sauts possibles.

On suppose donnée une fonction

```
ajouter_bonus(bonus, rang_du_bonus, i, j, bonus_actifs, code_bonus_actifs)
```

qui active la case bonus (i, j) dans le code des cases bonus activées `code_bonus_actifs`. Si (i, j) n'est pas une case bonus, la fonction renvoie `code_bonus_actifs`. L'argument `rang_du_bonus` est la structure renvoyée par `ranger_bonus(bonus)` et l'argument `bonus_actifs` est le masque des bonus actifs dont le code est `code_bonus_actifs`.

Question 14 Le code Python incomplet de la page suivante implémente la fonction `trouve_dynamique(T, sauts, bonus)` qui, étant donnés une grille de jeu `T`, la liste des sauts par défaut `sauts` et les sauts associés aux cases bonus `bonus`, calcule le chemin optimal par programmation dynamique en utilisant la récurrence trouvée en question 11. Le résultat de la fonction est le couple `(poids_opt, sauts_opt)` où `poids_opt` est le poids du chemin optimal trouvé et `sauts_opt` est la liste des sauts de ce chemin.

1. Donner les sept parties manquantes indiquées par « ... » dans le code.

2. Quelle est la complexité de cette fonction ?

```

def trouve_dynamique(T,sauts,bonus):
    N = len(T)
    nb_bonus = len(bonus)
    nb_code_bonus = ...
    # A COMPLETER (1)
    poids_opt = [[[INFINI for bonus_code in range(nb_code_bonus)]
                  for j in range(N)] for i in range(N)]
    saut_opt = [[[0,0,0) for bonus_code in range(nb_code_bonus)]
                  for j in range(N)] for i in range(N)]
    (bonus_au_rang,rang_du_bonus) = ranger_bonus(bonus)
    for bonus_actifs in combinaisons_bonus(nb_bonus):
        code_bonus_actifs = code_bonus(bonus_actifs)
        poids_opt[N-1][N-1][code_bonus_actifs] = ... # A COMPLETER (2)
        sauts_posibles = ... # A COMPLETER (3)
        for i in range(...): # A COMPLETER (4)
            for j in range(...): # A COMPLETER (5)
                code_bonus_dest = ajouter_bonus(bonus,rang_du_bonus,i,j,
                                                   bonus_actifs,code_bonus_actifs)
                if (i,j) in bonus:
                    sauts_posibles_final = sauts_posibles + bonus[(i,j)]
                else:
                    sauts_posibles_final = sauts_posibles
                for (delta_i,delta_j) in sauts_posibles_final:
                    i_dest = i+delta_i
                    j_dest = j+delta_j
                    if (i_dest in range(N) and j_dest in range(N)):
                        poids_opt_dest = poids_opt[i_dest][j_dest][code_bonus_dest]
                        if (poids_opt[i][j][code_bonus_actifs] > poids_opt_dest):
                            poids_opt[i][j][code_bonus_actifs] = ... # A COMPLETER (6)
                            saut_opt[i][j][code_bonus_actifs] = ... # A COMPLETER (7)
                        poids_opt[i][j][code_bonus_actifs] += T[i][j]

    return (poids_opt,saut_opt)

```

Question 15 Écrire la fonction `solution_dynamique(saut_opt,N)` qui, étant donnés la structure `saut_opt` calculée dans la question précédente et la dimension N de la grille, renvoie le chemin optimal correspondant. La fonction renvoie le chemin vide [] s'il n'existe pas de chemin entre la case de départ et celle d'arrivée.

-	F	I	N	-
---	---	---	---	---

A2025 – INFO COMMUNE

**ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2025**ÉPREUVE D'INFORMATIQUE COMMUNE**

Durée de l'épreuve : 2 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

*Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :*

INFORMATIQUE COMMUNE

Cette épreuve est commune aux candidats des filières MP, PC et PSI.

L'énoncé de cette épreuve comporte 14 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.



Autour du sac à dos

Introduction

Le problème du sac à dos, noté également KP (knapsack problem), est un problème d'optimisation combinatoire. Il modélise une situation analogue à celle du remplissage d'un sac à dos ne pouvant pas supporter plus d'un certain poids, avec tout ou partie d'un ensemble donné d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale (le profit), avec la contrainte de ne pas dépasser le poids maximum admissible (la quantité totale de ressources disponibles).

D'un point de vue mathématique, la formulation du problème du sac à dos est la suivante :

$$\text{Maximiser le profit total } P = \sum_{i=0}^{n-1} p_i x_i \text{ sous la contrainte } \sum_{i=0}^{n-1} r_i x_i \leq b$$

où

- n est le nombre d'objets candidats,
- i est l'entier caractérisant l'objet ($i \in \llbracket 0, n - 1 \rrbracket$),
- p_i est le profit (ou valeur) associé à l'objet i ,
- x_i est la variable de décision associée à l'objet i : $x_i = 1$ si l'objet i est sélectionné et $x_i = 0$ sinon,
- r_i est la quantité de ressources consommée par l'objet i (ou poids de l'objet i),
- b est la quantité totale de ressources disponibles (ou poids total).

On ne considère que des cas où chaque ressource r_i , chaque profit p_i et la quantité de ressources disponibles b , sont des entiers.

On fait aussi l'hypothèse que $\forall i \in \llbracket 0, n - 1 \rrbracket, r_i \leq b$.

Dans le problème du sac à dos multidimensionnel, noté MKP, on considère plusieurs sacs à dos ayant chacun un poids maximum admissible. L'objectif est alors de maximiser la valeur totale des objets contenus dans l'ensemble des sacs à dos.

Les applications pratiques sont nombreuses : transport de marchandises, découpe de matériaux, gestion de portefeuilles financiers, allocation de tâches à des systèmes multiprocesseurs, ...

1 Base de données - Exemple de fret maritime de conteneurs

On donne les tables NAVIRES et CONTENEURS suivantes :

- NAVIRES(idN, evp, portDepN, dateDep, portDestN)
 - idN : clé primaire, identifiant du navire (type entier, non nul)
 - evp : capacité en équivalent conteneurs de longueur 20 pieds (type entier)
 - portDepN : port de départ (type chaîne de caractères)
 - dateDep : date de départ (type date)
 - portDestN : port de destination (type chaîne de caractères)

idN	evp	portDepN	dateDep	portDestN
12	1500	"Marseille"	2025-06-23	"Hambourg"
2	16000	"Valence"	2025-06-18	"Algésiras"
5	500	"Le Havre"	2025-10-06	"Rotterdam"
8	23000	"Hongkong"	2025-07-02	"Shanghai"
...

- CONTENEURS(idC, idN, taille, portDepC, dateDisp, portDestC, val)
 - idC : clé primaire, identifiant du conteneur (type entier)
 - idN : identifiant du navire attribué (type entier, valeur 0 si non encore attribué).
 - taille : longueur du conteneur, 20 ou 40 pieds (type entier). La hauteur et la profondeur sont identiques.
 - portDepC : port de départ (type chaîne de caractères)
 - dateDisp : date de mise à disposition (type date)
 - portDestC : port de destination (type chaîne de caractères)
 - val : tarification, valeur entière de 1 à 5 par ordre croissant de profit pour l'entreprise (type entier)

idC	idN	taille	portDepC	dateDisp	portDestC	val
103	12	20	"Marseille"	2025-08-08	"Hambourg"	2
218	12	40	"Marseille"	2025-07-08	"Valence"	1
5	0	40	"Le Havre"	2025-10-01	"Shangai"	5
885	8	20	"Hongkong"	2025-07-01	"Barcelone"	1
...

Pour les deux questions qui suivent, on demande d'écrire les requêtes en langage SQL.

On notera que pour les valeurs d'attributs de type date (format AAAA-MM-JJ), les relations d'ordre ($<$, \leq , $>$, \geq , $=$, \neq) sont autorisées. Par exemple, 2025-07-02 < 2025-10-03.

- **Q1** – Écrire une requête permettant de retourner la liste des identifiants de conteneurs et leur ratio tarification/longueur trié par ordre décroissant, partant de Marseille et devant aller à Barcelone, avec une mise à disposition avant le 01/01/2025.
- **Q2** – Écrire une requête permettant de retourner les identifiants de navire et leur nombre respectif de conteneurs attribués.

2 Structure de données pour l'espace des objets

On demande d'utiliser exclusivement le langage Python pour toute la suite du sujet.

On implémente l'espace des objets i ($i \in \llbracket 0, n-1 \rrbracket$) avec une structure informatique `obj` de type liste de taille n .

La valeur de `obj[i]` est un tuple (r_i, p_i) où r_i et p_i sont les valeurs respectives des paramètres r_i et p_i de l'objet i .

Par exemple, $(10, 5)$ permet de définir pour un objet étiqueté 3 les paramètres $r_3 = 10$ et $p_3 = 5$.

Une solution au problème du sac à dos $S = [x_0, x_1, \dots, x_{n-1}]$, est implementée par une liste Python. On rappelle que $x_i = 1$ si l'objet i est sélectionné et $x_i = 0$ sinon.

Pour les deux questions qui suivent, la fonction python `sum` ne sera pas utilisée.

- Q3** – Écrire une fonction `profit(obj: [(int)], S: [int]) -> int` prenant en argument une liste `obj` et une liste `S`, et retournant la valeur du profit `P`.
- Q4** – Écrire une fonction `contrainte(obj: [(int)], S: [int], b: int) -> bool` prenant en argument une liste `obj`, une liste `S` et un nombre `b`, et retournant le booléen `True` si la contrainte de ressources du problème du sac à dos est respectée, `False` dans le cas contraire.

3 Structure de données pour l'espace des solutions

On choisit de modéliser l'espace des solutions au problème du sac à dos avec un arbre binaire (figure 1). On ne s'intéresse pas pour le moment au respect de la contrainte $\sum_{i=0}^{n-1} r_i x_i \leq b$.

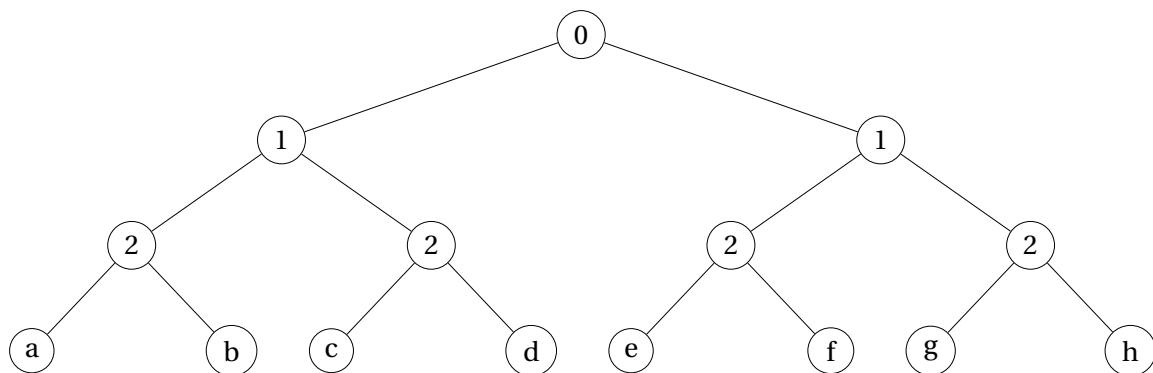


Figure 1 – Arbre binaire et espace des solutions pour le cas particulier où $n = 3$

À chaque noeud, deux choix sont possibles :

- Fils gauche : l'objet i ($i \in \llbracket 0, n-1 \rrbracket$) est sélectionné ($x_i = 1$).
- Fils droit : l'objet i ($i \in \llbracket 0, n-1 \rrbracket$) n'est pas sélectionné ($x_i = 0$).

Au niveau le plus élevé (la racine), le choix se porte sur l'objet 0. Au niveau immédiatement inférieur, le choix se fait sur l'objet 1, et ainsi de suite ... jusqu'à l'objet $n-1$.

Au niveau le plus bas, sont définies les feuilles (sans successeur). À chacune d'entre-elles correspond une solution du type $S = [x_0, x_1, \dots, x_{n-1}]$ décrivant les choix effectués le long du chemin menant de la racine à la feuille considérée.

- Q5** – Sur l'exemple de la figure 1 où $n = 3$, à quoi est égale la liste `S` pour les feuilles `b` et `c`?
- Q6** – Donner le nombre de feuilles de l'arbre binaire en fonction du nombre d'objets n . Indiquer en la justifiant la complexité temporelle en fonction du nombre d'objets n d'un algorithme de résolution du problème du sac à dos de type "force brute" qui envisagerait toutes les combinaisons possibles de sélection d'objets.

Lorsque le nombre d'objets n devient grand, un parcours de toutes les solutions n'est pas possible en un temps raisonnable.

4 Résolution approchée par un algorithme glouton

On envisage une stratégie gloutonne pour la résolution du problème du sac à dos. Lorsque le choix de sélectionner un objet est fait, il ne sera plus remis en question. Les étapes sont les suivantes :

- 1) On construit la liste Lqi qui contient pour chaque objet $i \in [0, n - 1]$ le rapport profit sur ressource consommée $q_i = p_i / r_i$. En parallèle, on construit la liste Li , telle qu'initiallement $Li[i]$ vaut i . La liste Li est telle que $Li[j]$ ($j \in [0, n - 1]$) donne l'indice dans la liste obj de l'objet décrit par le rapport situé en position j dans Lqi .
- 2) On trie la liste Lqi par ordre décroissant et on modifie simultanément la liste Li de sorte que $\forall j \in [0, n - 2], Lqi[Li[j]] \geq Lqi[Li[j + 1]]$.
- 3) En partant de la quantité totale de ressources disponibles b , on sélectionne l'objet i de rapport q_i le plus grand possible tel que $r_i \leq b$.
- 4) On met à jour la quantité de ressources disponibles $b = b - r_i$ et on réitère le processus à partir de 3) jusqu'à ce qu'il ne soit plus possible de sélectionner un objet supplémentaire.

- Q7 – On prend le cas particulier où $obj=[(2,3),(1,4),(4,4)]$ et $b=5$. Expliquer quelle liste S est construite par la stratégie gloutonne précédente.

Le code incomplet de la fonction `construitLi(obj: [(int)]) -> [int]` qui prend en argument la liste obj et qui retourne la liste Li définie précédemment, est donné ci-après. Cet algorithme correspond aux étapes 1 et 2 décrites en introduction de la stratégie gloutonne.

```

1 def construitLi(obj):
2     Lqi=[]
3     Li=[]
4     for i in range(len(obj)):
5         Lqi.append(.....)
6         Li.append(i)
7     for i in range (1,len(Lqi)):
8         x=Lqi[i]
9         j=i
10        while j>0 and Lqi[j-1]<x:
11            Lqi[j]=Lqi[j-1]
12            Li[j]=.....
13            j-=1
14            Lqi[j]=x
15            Li[j]=.....
16    return Li

```

- Q8 – Proposer le code Python complet des lignes 5, 12 et 15.

- Q9 – Identifier le meilleur des cas et le pire des cas de la méthode de tri utilisée dans la fonction `construitLi` en précisant et justifiant leur complexité temporelle respective.

On donne en page suivante le code incomplet qui calcule la solution au problème du sac à dos S sous la forme $S=[x_0, x_1, \dots]$ avec une stratégie gloutonne.

Une fois la liste Li construite en ligne 2, cet algorithme correspond aux étapes 3 et 4 décrites en introduction de la stratégie gloutonne. Les variables obj et b ont été définies en amont du code.

```

1 S=len(obj)*[0]
2 Li=construitLi(obj)
3 j=0
4 while .....:
5     if .....<=b:
6         S[.....]=.....
7         b=.....
8     j+=1

```

Q10 – Proposer le code Python complet des lignes 4, 5, 6 et 7.

La complexité temporelle optimale d'une méthode de tri dans le pire des cas en fonction du nombre n de données à trier est quasi linéaire $C(n) = \Theta(n \log(n))$. On peut en déduire que la complexité optimale de la stratégie gloutonne est aussi quasi linéaire.

Q11 – On reprend le cas particulier où $obj=[(2,3),(1,4),(4,4)]$ et $b=5$. Donner la solution optimale pour ce cas particulier. Conclure sur la pertinence d'une approche gloutonne.

5 Résolution exacte par un algorithme de programmation dynamique

On utilise une méthode dite de programmation dynamique pour résoudre le problème du sac à dos à n objets.

On note pour tout $k \in \llbracket 0, n \rrbracket$ et tout $r \in \llbracket 0, b \rrbracket$, $P_{max}(k, r)$ le profit maximum réalisable avec les objets d'indice i compris entre 0 et k non inclus pour une quantité maximale de ressources consommées r .

On pose $P_{max}(0, r) = 0$ pour tout $r \in \llbracket 0, b \rrbracket$, et on considère acquise la formule de récurrence suivante, donnant pour tout $i \in \llbracket 0, n - 1 \rrbracket$ et tout $r \in \llbracket 0, b \rrbracket$, le profit maximum :

$$P_{max}(i + 1, r) = \max\{P_{max}(i, r), P_{max}(i, r - r_i) + p_i\}$$

où

- p_i est le profit associé à l'objet i ($i \in \llbracket 0, n - 1 \rrbracket$),
- r_i est la quantité de ressources consommée par l'objet i ($i \in \llbracket 0, n - 1 \rrbracket$),
- r est la quantité maximale de ressources consommées.

On stocke les valeurs de $P_{max}(k, r)$ dans un tableau T à deux dimensions sous forme de liste de listes.

On rappelle que b est la quantité totale de ressources disponibles.

La fonction `KPprogDynamique(obj:[(int)],b:int)->int` donnée en page suivante, implémente un algorithme de programmation dynamique itératif.

```

1 def KPprogDynamique(obj,b):
2     T=[]
3     n=len(obj)
4     for k in range(n+1):
5         L=[]
6         for r in range(b+1):
7             L.append(0)
8         T.append(L)
9     for i in range(n):
10        for r in range(b+1):
11            if obj[i][0]<=r:
12                T[i+1][r]=max(T[i][r],T[i][r-obj[i][0]]+obj[i][1])
13            else:
14                T[i+1][r]=T[i][r]
15    return T[n][b]
```

□ **Q12** – On reprend le cas particulier où $\text{obj}=[(2,3),(1,4),(4,4)]$ et $b=5$. Donner sans justifications les valeurs du tableau T à l'issue de l'exécution des lignes 2 à 8. Puis, à la fin de chacune des trois itérations de la boucle `for` en ligne 9, donner la valeur de $T[i+1]$. Préciser ce que retourner la fonction `KPprogDynamique(obj,b)` et à quoi correspond cette valeur.

□ **Q13** – Déterminer en la justifiant la complexité asymptotique temporelle de la fonction `KPprogDynamique(obj,b)`.

On conserve les 14 premières lignes de la fonction proposée précédemment. On complète avec les lignes de code suivantes afin de retourner la solution S sous la forme $S=[x_0, x_1, \dots]$ et la valeur de $T[n][b]$.

```

1 def KPprogDynamique(obj,b):
2     #
3     #Lignes 2 à 14 précédentes conservées
4     #
5     S=n*[0]
6     k=n-1
7     r=b
8     while r>0 and k>=0 and T[k+1][r]!=0:
9         while k>0 and T[k+1][r]==T[k][r]:
10            k-=1
11            S[k]=1
12            r=r-obj[k][0]
13            k-=1
14    return S,T[n][b]
```

□ **Q14** – Donner les valeurs des variables S , k et r après chaque itération de la boucle `while` en ligne 8. Indiquer en justifiant si la complexité asymptotique temporelle de la fonction `KPprogDynamique(obj,b)` est ainsi modifiée.

6 Résolution exacte par un algorithme PSE

Un algorithme par séparation et évaluation (PSE) ou *branch and bound* permet d'éliminer de l'arbre des solutions, les branches qui ne peuvent pas contenir la solution optimale. Cet "élagage" de l'arbre permet d'éviter l'exploration complète de celui-ci.

On applique dans un premier temps un algorithme glouton afin d'avoir en un temps raisonnable une solution approchée de la solution optimale dont le profit est noté P_{min} .

La reformulation du problème initial est donc :

$$\text{Maximiser le profit total } P = \sum_{i=0}^{n-1} p_i x_i \text{ sous les contraintes } \sum_{i=0}^{n-1} r_i x_i \leq b \text{ et } P > P_{min}.$$

L'arbre binaire des solutions arbreSol est implémenté par une structure récursive (figure 2). Chaque nœud est lui-même un arbre binaire qui construit niveau par niveau une solution. On modélise chacun par un dictionnaire possédant trois clés de type str :

- 'S' : le vecteur courant solution :
 - arbresol['S']=[] pour la racine de l'arbre (liste vide),
 - arbresol['S']=[x_0, x_1, \dots, x_{k-1}] à la profondeur k ($k \in \llbracket 1, n \rrbracket$),
- 'g' : le nœud "fils" de gauche,
- 'd' : le nœud "fils" de droite.

À la profondeur $k = n$,

- arbresol['S']=[x_0, x_1, \dots, x_{n-1}] : le vecteur est une solution,
- arbresol['g']={} : le nœud "fils" de gauche est un dictionnaire vide,
- arbresol['d']={} : le nœud "fils" de droite est un dictionnaire vide.

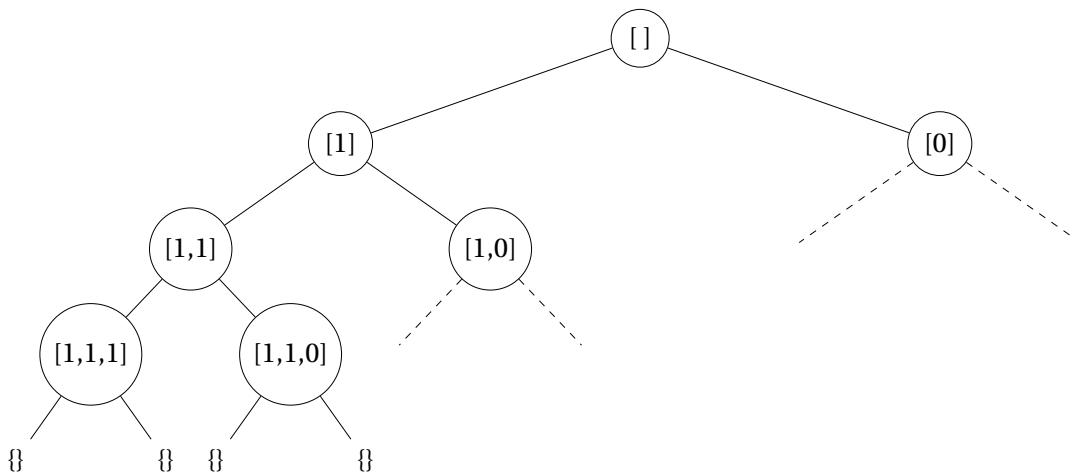


Figure 2 – Arbre binaire partiel pour $n = 3$ avec les valeurs de arbresol['S'] à chaque nœud

La fonction récursive construireArbreBinaire(a:dict,n:int) permet, en page suivante, de construire un tel arbre.

```

1 def construireArbreBinaire(a,n):
2     if n==0: return a
3     else:
4         filsGauche={'S':a['S']+[[1],{}], 'g':{}, 'd':{}}
5         filsDroit={'S':a['S']+[[0],{}], 'g':{}, 'd':{}}
6         a['g']=filsGauche
7         a['d']=filsDroit
8         construireArbreBinaire(filsGauche,n-1)
9         construireArbreBinaire(filsDroit,n-1)

```

Par exemple, le code suivant permet de construire un arbre des solutions pour $n = 3$:

```

1 arbreSol={'S':[], 'g':{}, 'd':{}}
2 construireArbreBinaire(arbreSol,3)

```

□ **Q15** – Écrire une fonction `estFeuille(a:dict)->bool` qui prend en argument un arbre `a` et qui retourne le booléen `True` si `a` est une feuille, `False` dans le cas contraire.

Le profit P_{min} obtenu par un algorithme glouton est stocké dans la **variable globale** `Pmin`.

□ **Q16** – Écrire une fonction `possible(obj:[(int)], Sk:[int], b:int)->bool` qui prend en argument la liste des objets `obj`, une liste `Sk` valeur d'une solution courante d'un nœud de niveau $k \in \llbracket 1, n \rrbracket$ ($\text{len}(Sk)=k$) et la quantité de ressources disponibles `b`. La fonction retourne le booléen `True` s'il est utile de poursuivre l'exploration des nœuds inférieurs. La fonction retourne `False` dans le cas contraire. On s'assure ici que :

- la condition de ressources est satisfaite : on utilise pour cela la fonction `contrainte(obj:[(int)], S:[int], b:int)->bool` avec un vecteur solution `S` de longueur n égal à la liste `Sk` complétée par des 0. On vérifie ainsi dans un premier temps que le simple ajout de l'objet à la profondeur k ($k \in \llbracket 1, n \rrbracket$) ne consomme pas trop de ressources disponibles,
- il est possible de trouver une meilleure solution que l'algorithme glouton : on utilise la fonction `profit(obj:[(int)], S:[int])->int` avec un vecteur solution `S` égal à la liste `Sk` complétée par des 1 en une liste de longueur n pour simuler le profit maximal de la branche avec l'ajout de tous les objets non encore traités à la profondeur k ($k \in \llbracket 1, n \rrbracket$).

On adopte un algorithme récursif `KPforceBrute(arbre:dict, obj:[(int)], b:int)`, de résolution optimale par « force brute » du problème du sac à dos. Le code Python est donné en page suivante.

La variable `Pmin` est réactualisée s'il existe une solution meilleure que celle trouvée par une stratégie gloutonne et la solution optimale correspondante est alors mise à jour dans la **variable globale** `Sol`.

```

1 def KPforceBrute(arbre,obj,b):
2     global Pmin,Sol
3     if estFeuille(arbre) :
4         if contrainte(obj,arbre['S'],b):
5             P=profit(obj,arbre['S'])
6             if P>Pmin:
7                 Pmin=P
8                 Sol=arbre['S']
9     else:
10        KPforceBrute(arbre['g'],obj,b)
11        KPforceBrute(arbre['d'],obj,b)

```

On souhaite à la fois :

- "élaguer" les branches de l'arbre des solutions qui ne peuvent pas contenir de solution de profit supérieur à la valeur P_{min} obtenue par un algorithme glouton, et
- "élaguer" les branches de l'arbre des solutions qui dépassent la ressource maximale b .

□ **Q17** – Le nom de la fonction `KPforceBrute(arbre:dict,obj:[(int)],b:int)` est modifié en `KPpse(arbre:dict,obj:[(int)],b:int)`. Cette fonction prend en argument l'arbre des solutions arbre, la liste des objets obj et un nombre ressources b. Elle affecte aux variables globales `Pmin` et `Sol` respectivement la valeur du profit maximal et le vecteur solution correspondant, conformément à l'algorithme PSE. Réécrire le code Python à partir de la ligne 9 (autant de lignes supplémentaires que nécessaire). Ne pas réécrire les autres lignes.

7 Résolution approchée par colonie de fourmis (ACO)

Principe de la méthode

L'optimisation par colonies de fourmis (ACO : *ants colony optimization*) est une méthode inspirée du comportement de fourmis lorsque celles-ci sont à la recherche de nourriture.

Le choix pour une fourmi artificielle $k \in \llbracket 0, nb_{Ants} - 1 \rrbracket$ de sélectionner un objet plutôt qu'un autre afin d'élaborer une solution S_k , se fait avec une loi de probabilité qui évolue au cours des itérations.

Les fourmis artificielles déposent des traces de phéromone sur les objets sélectionnés dans chacune des solutions S_k élaborées, en fonction du profit engendré. La probabilité de sélectionner un objet contenu dans une solution de profit important sera augmentée.

Elles élaborent alors de nouvelles solutions relativement aux traces de phéromone précédemment déposées.

De plus, ces traces subissent une loi d'évaporation au cours des itérations.

Intuitivement, cette communication indirecte fournit une information sur la qualité des solutions envisagées, afin d'attirer les fourmis vers les zones potentiellement intéressantes de l'espace des solutions.

On adopte alors la stratégie générique suivante :

- 1) Initialisation
 - a) Initialiser les traces de phéromone $\tau(o_i) \leftarrow \tau_{max}$ pour chaque objet o_i ($i \in \llbracket 0, n-1 \rrbracket$)
 - b) Initialiser la meilleure solution $S_{bestOfAll} \leftarrow \emptyset$ et le profit $P_{bestOfAll} \leftarrow 0$

- 2) Processus itératif de nb_{it} itérations (nb_{it} : valeur entière définie afin d'obtenir une solution approchée de qualité en un temps d'exécution raisonnable).

Tant que le nombre d'itérations nb_{it} n'est pas atteint, faire :

- a) Initialiser les solutions $S = \{S_0, S_1, \dots, S_{nb_{Ants}-1}\}/S_k \leftarrow \emptyset$, $k \in \llbracket 0, nb_{Ants} - 1 \rrbracket$.
- b) Pour chaque fourmi $k \in \llbracket 0, nb_{Ants} - 1 \rrbracket$, construire une solution S_k comme suit :

- i. Choisir aléatoirement un premier objet $o_0 \in \llbracket 0, n - 1 \rrbracket$ avec une loi uniforme.
- ii. $S_k \leftarrow \{o_0\}$ et la quantité de ressources disponibles b est mise à jour.
- iii. $candidats \leftarrow \{o_i \in \llbracket 0, n - 1 \rrbracket / o_i \neq o_0 \text{ et respect de la contrainte maximale de ressources}\}$.
- iv. Tant que $candidats \neq \emptyset$, faire :
 - Choisir un objet $o_i \in candidats$ avec la probabilité

$$prob(o_i) = \frac{(\tau(o_i))^\alpha (\eta(o_i))^\beta}{\sum_{o_j \in candidats} (\tau(o_j))^\alpha (\eta(o_j))^\beta}$$

La probabilité de choisir un objet $prob(o_i)$ est définie avec un facteur phéromonal $\tau(o_i)$ et un facteur heuristique $\eta(o_i)$.

Le facteur heuristique $\eta(o_i)$ doit permettre de choisir les objets les plus «intéressants» de part leurs caractéristiques intrinsèques. Il est défini par un ratio profit p_i /ressource r_i tel que :

$$\eta(o_i) = b \times \frac{p_i}{r_i}$$

avec $b = b - \sum_{o_i \in S_k} r_i$, la quantité restante de la ressource lorsque la fourmi construit la solution S_k .

La quantité de ressources disponibles b est réactualisée après chaque ajout d'objet dans la solution S_k .

- Enlever de $candidats$ chaque objet qui viole des contraintes de ressources.
- c) Mettre à jour les traces de phéromone $\tau(o_i)$ en fonction des solutions S_k et $S_{bestOfAll}$
 - i. Toutes les traces de phéromone sont diminuées afin de simuler l'évaporation. On multiplie chaque composant phéromonal par un coefficient réel de persistance $\rho \in [0, 1]$.
 - ii. On détermine la meilleure fourmi k_{max} du cycle courant de la boucle «Tant que le nombre d'itérations nb_{it} ...». C'est celle qui a trouvé la solution $S_{max} \in S = \{S_0, S_1, \dots, S_{nb_{Ants}-1}\}$, c'est-à-dire la solution ayant le profit maximal $P(S_{max}) = P_{max}$ trouvé durant le cycle courant.
 - iii. On met à jour la meilleure solution $S_{bestOfAll}$ ainsi que le meilleur profit $P_{bestOfAll} = P(S_{bestOfAll})$ trouvé depuis le début de l'exécution du processus itératif (y compris le cycle courant).
 - iv. La meilleure fourmi k_{max} du cycle courant dépose de la phéromone. La quantité déposée est égale à : $1/(1 + P_{bestOfAll} - P_{max})$. Cette quantité de phéromone est ajoutée sur chacun des composants phéromonaux $\tau(o_i)$ de S_{max} , c'est-à-dire correspondant aux objets o_i sélectionnés dans S_{max} .

- v. Si une trace de phéromone est inférieure à τ_{min} alors la mettre à τ_{min} .
- vi. Si une trace de phéromone est supérieure à τ_{max} alors la mettre à τ_{max} .

Implémentation de l'algorithme

Afin qu'une fourmi artificielle k puisse choisir un objet $o_i \in candidats$ au cours de la construction d'une solution S_k , on implémente l'ensemble des *candidats* avec une liste *candidats* d'indices $i \in \llbracket 0, n - 1 \rrbracket$ représentant la position dans la liste *obj* des objets o_i compatibles avec la quantité de ressources disponibles b .

De plus, on utilise un dictionnaire *prob* tel qu'à la clé i (indice de l'objet o_i) correspond la valeur de la probabilité $prob(o_i)$ (comprise en 0 et 1).

Les deux paramètres α et β sont définis dans deux variables globales. Par exemple, *alpha=2* et *beta=5*.

La liste *T* de taille n contient les facteurs phéromonaux associés aux objets $\tau(o_i)$ ($o_i \in \llbracket 0, n - 1 \rrbracket$).

La quantité de ressources disponibles est stockée dans la variable *b*.

Les paramètres ρ , τ_{max} et τ_{min} sont définis dans des variables globales. À titre d'exemple, *rho=0.8*, *Tmax=6* et *Tmin=0.01*.

La liste *S* contient les listes *S[k]* des solutions S_k ($k \in \llbracket 0, nb_{Ants} - 1 \rrbracket$) construites par chacune des fourmis artificielles lors d'une itération de l'algorithme.

On rappelle que chaque solution est de la forme $S_k = [x_0, x_1, \dots, x_{n-1}]$ avec $x_i \in \{0, 1\}$ ($i \in \llbracket 0, n - 1 \rrbracket$).

La meilleure de toutes les solutions construites depuis le début de l'exécution de l'algorithme et le profit associé, sont stockés dans les variables globales respectives *SbestOfAll* et *PbestOfAll*.

On donne ci-après le code incomplet de la fonction

`mettreAjourTetSolution(obj: [(int)], T: [float], S: [[int]])`, qui prend en argument la liste des objets *obj*, la liste des facteurs phéromonaux *T* et la liste des solutions *S*, et qui met à jour la liste des facteurs phéromonaux *T*, la meilleure solution *SbestOfAll* et le meilleur profit *PbestOfAll*. Cette fonction ne retourne rien et correspond à l'item 2)c) de la description de l'algorithme.

```

1 def mettreAjourTetSolution(obj, T, S):
2     global SbestOfAll, PbestOfAll
3     # 2)c)i.
4     # Utiliser le nombre de lignes nécessaires
5     .....
6     # 2)c)ii.
7     #Utiliser le nombre de lignes nécessaires
8     .....
9     # 2)c)iii.
10    if Pmax>PbestOfAll:
11        PbestOfAll=Pmax
12        SbestOfAll=S[kmax]
13    # 2)c)iv.v.vi.
14    #Utiliser le nombre de lignes nécessaires
15    .....

```

□ **Q18** – Proposer le code Python complet des lignes 4 et plus de la partie 2)c)i. de la fonction ci-dessus.

□ **Q19** – Proposer le code Python complet des lignes 7 et plus de la partie 2)c)ii.

□ **Q20** – Proposer le code Python complet des lignes 14 et plus de la partie 2)c)iv.v.vi.

La fonction `randint(a,b)` de la bibliothèque `random` a été importée. Elle permet de retourner de manière aléatoire un entier $x/a \leq x \leq b$.

La fonction `miseAjourCandidats(obj:[(int)], candidats:[int], b:int)` a déjà été implémentée. Elle permet de supprimer de la liste `candidats` tous les objets de la liste `obj` ne respectant pas la contrainte de ressource `b`. (item 2)b)iii).

La fonction `L.remove(x)` appliquée à une liste `L` permet de retirer un élément `x` de la liste.

On donne ci-après le code incomplet qui permet de trouver une solution au problème posé en appliquant un algorithme par colonie de fourmis.

```

1 #1)a)
2 n=len(obj)
3 T=[Tmax for i in range(n)]
4 #1)b)
5 PbestOfAll=0
6 SbestOfAll=n*[0]
7 #2)
8 for it in range(nbit):
9     #2)a)
10    S=[n*[0] for i in range(nbAnts)]
11    #2)b)
12    for k in range(nbAnts):
13        b2=b
14        #2)b)i.
15        .....
16        #2)b)ii.
17        .....
18        .....
19        #2)b)iii.
20        candidats=[i for i in range(n) if i!=o0]
21        miseAjourCandidats(obj,candidats,b2)
22        #2)b)iv.
23        while b2>0 and candidats!=[]:
24            #Utiliser le nombre de lignes nécessaires
25            .....
26            mettreAjourTetSolution(obj,T,S)

```

□ **Q21** – On donne le nom de variable `o0` (o zéro) pour l'indice de l'objet choisi. Proposer le code Python complet des lignes 15, 17 et 18, définies ci-dessus.

On propose ci-dessous l'ébauche d'une fonction

`construitProb(obj : [(int)], candidats : [int], b:int, T:[float]) -> {int:float},`
qui prend en argument la liste des objets `obj`, la liste des indices `candidats`, l'entier `b` et la liste `T`, et qui retourne le dictionnaire `prob` des valeurs de probabilité associées à chacun des objets $o_i \in candidats$. Cette fonction participe à la réalisation des tâches décrites à l'item 2)b)iv. de la description de l'algorithme.

```

1 def construitProb(obj,candidats,b,T):
2     m=len(candidats)
3     prob=.....
4     for i in range(m):
5         oi=.....
6         .....
7         s=sum(prob.values())
8         for i in range(m):
9             oi=.....
10            .....
11     return prob

```

□ **Q22** – Proposer le code Python complet des lignes 3, 5, 6, 9 et 10.

On dispose d'une fonction
`choixCandidat(candidats : [int], prob:{int:float}) -> int`,
qui prend en argument la liste `candidats` et le dictionnaire `prob`, et qui retourne l'indice du candidat sélectionné. Cette fonction participe à la réalisation des tâches décrites à l'item 2)b)iv. de la description de l'algorithme.

□ **Q23** – Proposer le code Python complet des lignes 24 et plus, définies en page précédente.

8 Analyse des résultats

On réalise une simulation avec 20 objets ayant chacun une quantité de ressources consommées comprise entre 1 et 100, et un profit respectif compris entre 1 et 100.

La quantité de ressources disponibles a été fixée à 200.

Les résultats sont affichés pour chaque méthode avec le format :

- Nom de la méthode : temps d'exécution en secondes
- Solution, profit maximum
- Liste T pour l'algorithme ACO.

```

1 Force brute : 1.275291500001913
2 [0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 567
3
4 Glouton : 2.300000051036477e-05
5 [0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 548
6
7 PSE : 0.026752800011308864
8 [0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0] , 567
9

```

```
10 Programmation dynamique : 0.00085039998521097
11 [0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0] , 567
12
13 ACO avec nbAnts=5 et nbit=1 : 0.00029949998133815825
14 [0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0] , 548
15 Liste T : [4.800..., 5.800..., 4.800..., 4.800..., 5.800..., ...]
16
17 ACO avec nbAnts=5 et nbit=5 : 0.0012731999740935862
18 [0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0] , 567
19 Liste T : [1.966..., 2.967..., 1.966..., 1.966..., 4.767..., ...]
20
21 ACO avec nbAnts=5 et nbit=50 : 0.011390200001187623
22 [0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0] , 567
23 Liste T : [0.01, 0.065..., 0.01, 0.01, 3.799..., ...]
```

- **Q24 –** Commenter les résultats quant aux critères de performance (temps d'exécution et qualité de la solution proposée). Commenter les résultats obtenus par l'algorithme ACO en précisant sur quels paramètres de la méthode on peut jouer, et quelles sont les conséquences sur les performances.
-

Fin du problème

SESSION 2025



PC5IN

ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE PC

INFORMATIQUE

Durée : 3 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

RAPPEL DES CONSIGNES

- Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.
- Ne pas utiliser de correcteur.
- Écrire le mot FIN à la fin de votre composition.

Les calculatrices sont interdites.

Le sujet est composé de trois parties.

L'épreuve est à traiter en langage **Python** sauf pour les bases de données.

Les différents algorithmes doivent être rendus dans leur forme définitive sur le **Document Réponse** dans l'espace réservé à cet effet en respectant les éléments de syntaxe du langage (les brouillons ne sont pas acceptés).

La réponse ne doit pas se limiter à la rédaction de l'algorithme sans explication, les programmes doivent être expliqués et commentés de manière raisonnable.

Énoncé et Annexe : 16 pages

Document Réponse (DR) : 11 pages

Seul le Document Réponse (DR) doit être rendu dans son intégralité (le QR Code doit être collé sur la première page du DR).

Gestion de Randonnées

Les fonctions seront définies avec leur signature dans le sujet :

```
ma_fonction(arg1:type1, arg2:type2) -> type3.
```

Cette notation permet de définir une fonction qui se nomme `ma_fonction` qui prend deux arguments en entrée, `arg1` de type `type1` et `arg2` de type `type2`. Cette fonction renvoie une valeur de type `type3`.

Il ne faut pas recopier les signatures des fonctions dans le DR, il faut écrire directement :

```
def ma_fonction(arg1, arg2):
    # liste d'instructions
```

Introduction

Lors d'une randonnée, des applications disponibles sur smartphones permettent d'enregistrer l'itinéraire parcouru. L'utilisation de ces données peut permettre d'analyser *a posteriori* le parcours effectué (distance, durée, dénivelés...) ou de planifier de nouvelles sorties.

L'objectif du travail proposé est de découvrir différentes facettes de ces applications.

Le sujet abordera les points suivants :

- l'organisation des différents parcours dans une base de données,
- différentes stratégies pour obtenir des informations sur le dénivelé effectué,
- la planification de randonnées en utilisant des algorithmes de type Dijkstra.

Les données recueillies lors d'une randonnée sont généralement stockées dans un fichier au format GPX (pour GPS eXchange Format). Ce fichier est appelé *trace GPX* de la randonnée.

Partie I - Gestion des randonnées dans une base de données

Les applications de randonnées permettent à un utilisateur de stocker les données de ses randonnées effectuées ou d'avoir accès à celles effectuées par d'autres utilisateurs. Ces données sont stockées dans une base contenant notamment les tables suivantes.

La table Randonnee contenant :

Id	entier, identifiant de la randonnée ;
Titre	chaîne de caractères, titre de la randonnée ;
Type	chaîne de caractères du type de l'activité : "Pied", "VTT", "Cheval" ;
Lieu	chaîne de caractères, coordonnées GPS du point de départ ;
Distance	flottant, longueur en kilomètres de la randonnée ;
DenP	entier, dénivelé positif en mètres ;
DenN	entier, dénivelé négatif en mètres ;
Duree	entier, durée en minutes de la randonnée ;
Niveau	entier compris entre 1 et 5 (1 : facile à 5 : extrême), difficulté ;
IdAuteur	entier, identifiant de l'auteur de la randonnée ;
Trace	chaîne de caractères, lien internet vers la trace GPX.

La table Auteur contenant :

Id	entier, identifiant de l'auteur (le randonneur) ;
Nom	chaîne de caractères, nom de l'auteur ;
Prenom	chaîne de caractères, prénom de l'auteur ;
Pseudo	chaîne de caractères, pseudo de l'auteur ;
Mail	chaîne de caractères, mail de l'auteur.

- Q1.** Expliquer en quoi l'attribut Titre ne peut probablement pas être une clé primaire pour la table Randonnee. Proposer un attribut de la table Randonnee qui puisse être une clé primaire.
- Q2.** Identifier un attribut qui soit une clé étrangère de la table Randonnee.
- Q3.** Écrire une requête SQL dont l'évaluation renvoie le titre, les coordonnées GPS du point de départ et la longueur des randonnées à pied.
- Q4.** Écrire une requête SQL dont l'évaluation renvoie l'identifiant de l'auteur et son nombre d'activités à pied de niveau 3, classées par ordre décroissant du nombre d'activités de chaque auteur.
- Q5.** Écrire une requête SQL dont l'évaluation renvoie le Pseudo de l'auteur et le Titre des randonnées stockées dans la base.
- Q6.** Écrire une requête SQL dont l'évaluation renvoie les nom et prénom d'un des auteurs ayant posté le plus de randonnées à cheval.

Partie II - Quelques calculs de dénivélés

Nous allons maintenant nous intéresser plus particulièrement aux données enregistrées par l'application lors d'une randonnée. En pratique, une application enregistre régulièrement les données fournies par le GPS du téléphone portable comme la latitude et la longitude exprimées en degrés ainsi que l'altitude (élévation) exprimée en mètres. La **figure 1** présente un extrait du fichier *trace GPX*.

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx
  xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/11.xsd"
  xmlns="http://www.topografix.com/GPX/1/1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <trk>
    <name>Randonnée</name>
    <type>Marche</type>
    <trkseg>
      <trkpt lat="43.331146650016307830810546875" lon="-1.62765503861010074615478515625">
        <ele>261</ele>
        <time>2022-08-01T07:37:39.000Z</time>
      </trkpt>
      <trkpt lat="43.33105503581464290618896484375" lon="-1.62789593450725078582763671875">
        <ele>267.20001220703125</ele>
        <time>2022-08-01T07:37:52.000Z</time>
      </trkpt>
      <trkpt lat="43.3310479111969470977783203125" lon="-1.62792795337736606597900390625">
        <ele>267.79998779296875</ele>
        <time>2022-08-01T07:37:54.000Z</time>
      </trkpt>
      <trkpt lat="43.3310405351221561431884765625" lon="-1.62796609103679656982421875">
        <ele>268.399993896484375</ele>
        <time>2022-08-01T07:37:57.000Z</time>
      </trkpt>
    </trkseg>
  </trk>
</gpx>
```

Figure 1 - Exemple de fichier *trace GPX*

Le module `gpxpy` de Python permet de lire et extraire simplement les données de ce type de fichier.

Q7. Écrire une ligne de code permettant l'importation du module `gpxpy`.

Un *parcours* (ou une randonnée) est alors une succession de points. Après lecture et traitement du fichier à l'aide du module `gpxpy`, l'itinéraire d'une randonnée est représenté par une liste de points où chacun de ces points est un triplet de trois flottants correspondant respectivement à la latitude, la longitude et l'altitude. Le premier point d'un itinéraire sera le *point de départ* et le dernier le *point d'arrivée*.

Pour améliorer la lisibilité de la signature de nos fonctions, nous utiliserons le type `trpt` (pour track point ou point de la trace) pour représenter les triplets de points ainsi que le type `itineraire` pour les listes de points. Ainsi, à partir du fichier de la **figure 1** on pourra introduire les variables `p0`, `p1`, `p2`, `p3` qui sont de type `trpt` et la variable `iti` qui est de type `itineraire` :

```
p0 = (43.331146, -1.627655, 261.00) # point de départ
p1 = (43.331055, -1.627895, 267.20) # point intermédiaire
p2 = (43.331047, -1.627927, 267.79) # point intermédiaire
p3 = (43.331040, -1.627966, 268.39) # point d'arrivée
iti = [p0, p1, p2, p3] # itinéraire
```

Rappelons (**figure 2**) qu'un point de la surface terrestre est défini par :

- sa *latitude*, notée ϕ , qui est la mesure angulaire entre l'équateur et ce point. Elle est représentée par un angle compris entre -90° et $+90^\circ$;
- sa *longitude*, notée λ , qui est la mesure angulaire entre le méridien de référence (méridien de Greenwich) et ce point. Elle est représentée par un angle compris entre -180° et $+180^\circ$;
- son altitude qui exprime la hauteur entre le niveau de la mer et le niveau du point. Elle est représentée par un réel et s'exprime en mètres.

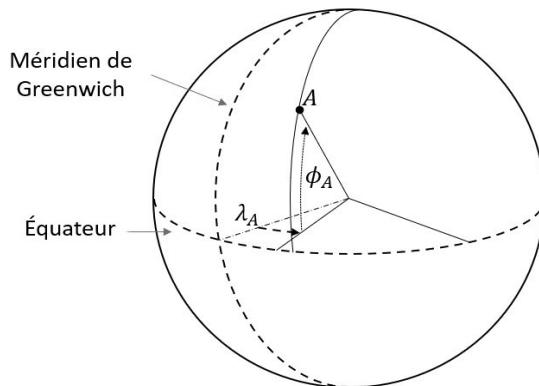


Figure 2 - Repérage, sur la surface du globe, d'un point A de latitude ϕ_A et de longitude λ_A

La syntaxe pour extraire les éléments d'un tuple est identique à celle utilisée pour les éléments d'une liste mais les tuples ne sont pas modifiables (ou mutables).

```
>>> p0 = (43.331146, -1.627655, 261.00) # point de départ
>>> p0[0]
43.331146
>>> (a, b, c) = p0
>>> a
43.331146
```

On considère la fonction `mystere`, dont l'argument `iti` est non vide :

```
1 | def mystere( iti :itinéraire ) -> float :
2 |     s = 0
3 |     for i in range(len(iti)):
4 |         (lat, long, alt) = iti[i]
5 |         s = s + alt
6 |     return (s/len(iti))
```

Q8. Donner la valeur numérique que renvoie le code suivant. Donner la signification de cette valeur dans le contexte du sujet.

```
1 | >>> p0 = (47.8741, 1.8758, 100)
2 | >>> p1 = (47.8744, 1.8759, 102)
3 | >>> p2 = (47.8748, 1.8761, 110)
4 | >>> p3 = (47.8750, 1.8759, 108)
5 | >>> l1 = [p0, p1, p2, p3]
6 | >>> mystere(l1)
```

Q9. Donner la complexité temporelle de la fonction `mystere` en fonction de la taille n de la liste passée en argument.

Q10. Écrire une fonction `altitude_maximale(itineraire) -> float` qui, étant donné une liste non vide `itineraire` de points, renvoie l'altitude maximale de l'itinéraire en mètres.

Le *dénivelé global* d'une randonnée est la différence entre l'altitude maximale et l'altitude du point de départ.

Q11. En utilisant la fonction `altitude_maximale`, écrire une fonction `denevele_global(itineraire) -> float` qui, étant donné une liste `itineraire` de points, renvoie le dénivelé global de la randonnée.

II.1 - Premier calcul de dénivelé positif

Le dénivelé entre deux points successifs p_1 et p_2 d'un itinéraire est dit positif si la différence entre l'altitude de p_2 et l'altitude de p_1 est positive. On appelle alors *dénivelé positif* la différence entre ces deux altitudes. Le *dénivelé positif cumulé* d'une randonnée est la somme de tous les dénivelés positifs entre les points successifs du parcours.

Q12. Écrire une fonction `denevele_positif_cumule(itineraire) -> float` qui, étant donné une liste `itineraire` de points, renvoie le dénivelé positif cumulé de la randonnée.

La méthode de mesure de l'altitude par le GPS est relativement imprécise due à la présence éventuelle d'une couverture nuageuse, d'un parcours sous des arbres... Or, lors du calcul du dénivelé positif cumulé, de faibles erreurs répétées peuvent induire une erreur conséquente sur le calcul cumulé.

Par exemple, si le randonneur effectue une randonnée en bord de mer sur une plage d'altitude constante mais que le GPS effectue des mesures erronées pour donner une liste d'altitudes égale à $[0, -2, 2, -2, 2, -2, 2, -2, 2]$, le dénivelé positif cumulé calculé par la fonction précédente est égal à 16 mètres alors qu'il devrait être nul.

Nous allons envisager deux méthodes pour pallier ces imprécisions : le lissage des altitudes et l'utilisation d'altitudes de référence.

II.2 - Lissage des altitudes

Le *lissage* d'une liste de longueur n des altitudes par moyenne glissante de pas p consiste à remplacer l'altitude au point numéroté i par la moyenne des altitudes des points numérotés $i, i + 1, \dots, i + p - 1$ où $j = \min\{p - 1, n - i - 1\}$.

Par exemple, lorsque $p = 2$, la liste précédente sera remplacée par :

liste de départ	0	-2	2	-2	2	-2	2	-2	2
calcul	$\frac{0-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	2
liste lissée	-1	0	0	0	0	0	0	0	2

Le dénivelé positif est alors de 3 mètres, ce qui est plus proche de la réalité de l'itinéraire.

Q13. Écrire une fonction `alt_glissante(liste_alt:list, p:int) -> list` qui étant donné une liste d'altitudes et un entier p , crée une nouvelle liste contenant la moyenne glissante des altitudes avec un pas p . On pourra utiliser la fonction `min` qui prend deux nombres flottants en entrée et renvoie le plus petit des deux.

Q14. Évaluer la complexité de la fonction `alt_glissante` en fonction de la taille n de la liste d'altitudes passée en argument et du pas p .

II.3 - Utilisation d'altitudes de référence

Une autre stratégie pour améliorer la précision sur les altitudes, une fois la randonnée effectuée et une connexion internet plus stable trouvée, consiste à se connecter à une base de référence qui, étant donné un point du globe, renvoie son altitude. Bien sûr, tous les points ne sont pas stockés dans la base. Nous supposerons que la surface du globe est quadrillée par une liste de latitudes et une liste de longitudes et qu'en chaque point de cette grille l'altitude a été mesurée précisément. Ces altitudes sont stockées dans un dictionnaire nommé `dem` (Digital Elevation Model) dont les clés sont des couples latitude / longitude et les valeurs sont les altitudes correspondantes.

On considère le code suivant :

```
1 | lat_ref = []
2 | long_ref = []
3 | for (lat, long) in dem:
4 |     lat_ref.append(lat)
5 |     long_ref.append(long)
```

On supposera dans la suite que les valeurs -90 et 90 sont dans `lat_ref` et que les valeurs -180 et 180 sont dans `long_ref`.

Q15. Indiquer le type des variables `lat_ref` et `long_ref`. Expliquer quel est le contenu de ces variables dans le contexte de ce sujet.

On considère le code suivant :

```
1 | def auxiliaire(x, y):
2 |     if x == [] : return y
3 |     if y == [] : return x
4 |     if x[len(x)-1] < y[len(y)-1] :
5 |         val = y.pop()
6 |     else :
7 |         val = x.pop()
8 |     z = auxiliaire(x,y)
9 |     z.append(val)
10|    return z
11|
12| def principal(x):
13|     if len(x) <= 1:
14|         return x
15|     else:
16|         m = len(x)//2
17|         x1 = principal(x[0:m])
18|         y1 = principal(x[m:len(x)])
19|         z = auxiliaire(x1, y1)
20|         return z
```

Q16. Précisez la signature des fonctions `auxiliaire` et `principal`. La réponse doit être justifiée.

Q17. Sur le DR, cocher la (les) cases qui correspond(ent) au(x) type(s) de programmation utilisé(s) pour coder la fonction `principal`.

Q18. Justifier brièvement que, étant donné une liste `x`, l'appel `principal(x)` termine.

Q19. L'appel `principal(x)` permet de renvoyer une liste triée par ordre croissant. Proposer un nom qui décrit le type de tri utilisé en justifiant brièvement votre choix.

Par la suite, on suppose que les listes `lat_ref` et `long_ref` sont **triées**. Il faut maintenant déterminer le point du dictionnaire `dem` le plus proche d'un point donné.

On donne une implémentation partielle de la fonction `ref(valeur, liste_ref)` qui, étant donné un flottant `valeur` et une liste non vide de flottants triés par ordre croissant `liste_ref` tels que `valeur` est strictement compris entre le premier et le dernier élément de `liste_ref`, renvoie la valeur de la liste `liste_ref` la plus proche de `valeur`.

```
1 | def ref(valeur :float, liste_ref :list) -> float :
2 |     # on détermine ind_deb et ind_fin tels que
3 |     # liste_ref[ind_deb]<valeur<=liste_ref[ind_fin]
4 |     # avec une méthode par dichotomie
5 |     ind_deb = .....
6 |     ind_fin = .....
7 |     while ind_deb < ind_fin - 1:
8 |         k = .....
9 |         if valeur <= liste_ref[k] :
10 |             ind_fin = .....
11 |         else :
12 |             .....
13 |     # on détermine le plus proche
14 |     if liste_ref[ind_fin]-valeur<valeur-liste_ref[ind_deb]:
15 |         return .....
16 |     else :
17 |         return .....
```

Q20. Compléter les lignes 5, 6, 8, 10, 12, 15 et 17 de la fonction `ref`.

Q21. Utiliser les données précédentes pour écrire une fonction `standardise(liste_parcours:itineraire) -> itineraire` qui, étant donné un itinéraire, renvoie un nouvel itinéraire où l'altitude de chaque point a été remplacée par l'altitude issue du dictionnaire `dem`.

Pour chaque point de l'itinéraire, on cherchera la latitude ϕ_r de référence la plus proche de sa latitude, la longitude λ_r de référence la plus proche de sa longitude et on remplacera son altitude par l'altitude du point de référence de coordonnées (ϕ_r, λ_r) .

Les variables `lat_ref`, `long_ref` et `dem` sont définies globalement en dehors de la fonction et peuvent être utilisées directement.

Partie III - Organisation d'un Trek

Un randonneur souhaite effectuer un long Trek, c'est-à-dire une série de randonnées sur plusieurs jours. Il organise son parcours à partir d'un site qui propose différentes randonnées d'une journée. Chaque randonnée est définie par un niveau de difficulté allant de 1 (facile) à 5 (extrême). L'ensemble des données permet au randonneur d'établir un graphe où :

- les sommets représentent les points de départ / arrivée des randonnées,
- les arêtes représentent les randonnées possibles avec comme poids le niveau de la randonnée.

On suppose qu'il y a une unique randonnée qui relie 2 sommets du graphe.

On suppose que les randonnées peuvent être effectuées du point de départ vers le point d'arrivée ou du point d'arrivée vers le point de départ sans que la difficulté ne soit modifiée. Le graphe représentant les différentes randonnées sera donc non orienté.

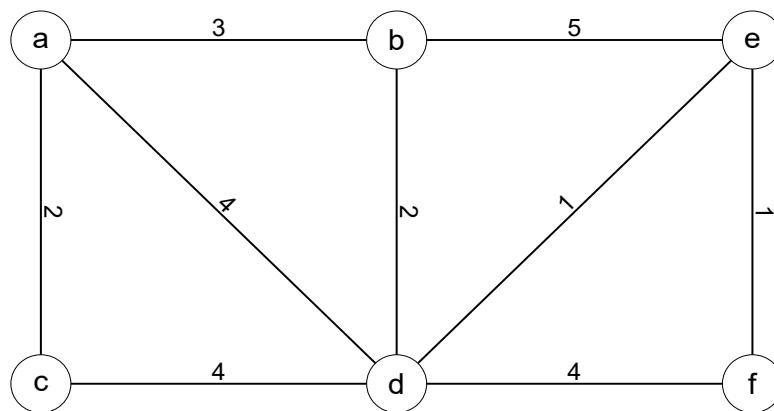


Figure 3 - Graphe G

Le graphe *G* de la **figure 3** est représenté par le dictionnaire *G* défini de la manière suivante :

```

G = dict()
G[ 'a' ] = { 'b' :3, 'c' :2, 'd' :4}
G[ 'b' ] = { 'a' :3, 'd' :2, 'e' :5}
G[ 'c' ] = { 'a' :2, 'd' :4}
G[ 'd' ] = { 'a' :4, 'b' :2, 'c' :4, 'e' :1, 'f' :4}
G[ 'e' ] = { 'b' :5, 'd' :1, 'f' :1}
G[ 'f' ] = { 'd' :4, 'e' :1}
  
```

Le randonneur souhaite aller du point *a* au point *f*. Cependant, comme il n'est pas entraîné, il choisit de trouver le chemin dont la somme des difficultés des randonnées est la plus petite, quel que soit le nombre d'étapes.

Afin d'utiliser le vocabulaire habituellement employé dans les algorithmes de parcours de graphes, on utilisera le terme "distance" au lieu du terme "difficulté" et on cherchera donc à minimiser la "distance" (au sens de "difficulté" du trek).

III.1 - Première idée : algorithme intuitif

Pour trouver son chemin, le randonneur exécute la fonction `mystere2` suivante.

```

1 | def mystere2(graph :dict , Sd:str , Sf:str) -> tuple :
2 |     """graph : graphe représentant les randonnées disponibles
3 |     Sd : sommet de départ dans le graphe
4 |     Sf : sommet d'arrivée dans le graphe
5 |     Renvoie une liste de randonnées permettant de relier Sd à Sf ainsi
6 |     que la somme des difficultés d'un tel chemin."""
7 |     dejaVisites = [] # Liste des sommets déjà visités
8 |     sTraite = Sd # Initialisation du sommet à traiter
9 |     chemin = [sTraite] # Chemin choisi initialisé
10 |    diffChemin = 0 # Difficulté du chemin choisi
11 |    # Construction itérative du chemin
12 |    while sTraite != Sf :
13 |        dejaVisites.append(sTraite)
14 |        d = float('inf') # valeur représentant l'infini
15 |        sInter = sTraite
16 |        for sommet in graph[sTraite] :
17 |            if sommet not in dejaVisites :
18 |                if graph[sTraite][sommet] < d :
19 |                    sInter = sommet
20 |                    d = graph[sTraite][sommet]
21 |        diffChemin = diffChemin + d
22 |        chemin.append(sInter)
23 |        sTraite = sInter
24 |
25 |    return chemin , diffChemin

```

Q22. Expliquer le fonctionnement de la boucle itérative comprise entre les lignes 15 à 19 et en déduire le nom du type d'algorithme utilisé.

Q23. Donner ce que renvoie l'instruction `mystere2(G, "a", "f")`. On ne demande pas d'indiquer toutes les étapes de l'algorithme.

Q24. Justifier si ce programme permet au randonneur de trouver le chemin de difficulté cumulée minimale.

III.2 - Deuxième idée : l'algorithme de Dijkstra

Pour résoudre son problème, le randonneur décide d'appliquer l'algorithme de Dijkstra. Il réalise ainsi une fonction `dijkstra` qui prend en arguments :

- la représentation du graphe sous forme de dictionnaire de dictionnaires ;
- le sommet de départ sous forme d'une chaîne de caractères ;
- le sommet d'arrivée sous forme d'une chaîne de caractères ;

et renvoie un dictionnaire dont les clés sont les sommets du graphe et les valeurs sont des couples dont :

- la première composante est la distance totale minimale du point de départ au sommet clé ;
- la seconde composante est le sommet précédent dans le graphe qui permet de réaliser la distance minimale entre le point de départ et le sommet clé.

L'implémentation rappelée ci-dessous de l'algorithme de Dijkstra utilise les quatre variables :

- `aVisiter` : liste des sommets qui doivent être visités ;
- `dejaVisites` : liste des sommets déjà visités ;
- `distance` : le dictionnaire qui sera renvoyé par la fonction ;
- `sTraite` : sommet dont on étudie les voisins pour mettre à jour les distances au point de départ.

La méthode `L.remove(elt)` permet de supprimer la première apparition de `elt` dans la liste `L`.

```

1 | def cherche_min(dico :dict, liste :list) -> str :
2 |     d = float("inf") # Initialisation
3 |     for s in liste : # parcours des elements de la liste
4 |         if s in dico and dico[s][0] < d: # recherche de la valeur
5 |             minimale
6 |             d = dico[s][0]
7 |             sTraite = s
8 |     return sTraite
9 |
10| def unpas(graph :dict, s:str, distance :dict, dejaVisites :list, avisiter :
11| list) -> None :
12|     avisiter.remove(s) # Mise a jour des sommets a visiter
13|     dejaVisites.append(s) # Mise a jour des sommets déjà visités
14|     for v in graph[s]: # Mise a jour des distances au point de depart
15|         if v not in dejaVisites :
16|             if v not in avisiter :
17|                 avisiter.append(v)
18|                 ndistance = distance[s][0] + graph[s][v]
19|                 if v not in distance or ndistance < distance[v][0] :
20|                     distance[v] = (ndistance, s)
21|
22| def dijkstra(graph :dict, Sd:str, Sf:str) -> dict :
23|     aVisiter = [Sd] # Liste des sommets à visiter
24|     dejaVisites = [] # Liste des sommets déjà visités
25|     distance = {Sd:(0, Sd)} # Dictionnaire des distances
26|     sTraite = Sd # Premier sommet a visiter
27|     while sTraite != Sf :
28|         sTraite = cherche_min(distance, aVisiter)
          unpas(graph, sTraite, distance, dejaVisites, aVisiter)
    return distance

```

Q25. On effectue l'appel `dijkstra(G, "a", "f")` où le graphe G est défini dans la **figure 3**. Dans le tableau du DR est représenté le contenu de certaines variables de l'algorithme `dijkstra` en fonction de l'étape de l'itération (comme si un `print` était effectué après la ligne 27). À partir des cases déjà remplies, compléter les cases vides du tableau. Lorsque la clé n'est pas définie dans le dictionnaire, la case du tableau contient un X.

Pour reconstruire un chemin qui réalise la distance optimale entre le point de départ "a" et le point d'arrivée "f", on utilise le code suivant. On rappelle que la variable globale G a été définie précédemment.

```

1 # On applique l'algorithme de Dijkstra :
2 sInit , sFin = "a" , "f"
3 distance = dijkstra(G, sInit , sFin)
4
5 # On construit la liste du chemin en partant de la fin
6 s = sFin
7 chemin = [s]
8 while ..... :
9     .....
10    .....
11 # On remet le chemin dans l'ordre du début vers la fin
12 chemin.reverse()
13
14 print("Un chemin de ", sInit , " à ", sFin , " est : ", chemin)
15 print("La difficulté minimale est de : ", .....)
```

Q26. Indiquer le contenu des lignes 8, 9, 10 et 15 du code précédent.

Q27. Expliquer comment pourrait être diminué le nombre de tests d'appartenance à une liste, notamment des lignes 14 et 15, de l'algorithme de Dijkstra.

Pour la fin de la sous-partie III.2, on considère le graphe de la **figure 4** où, pour simplifier, toutes les randonnées sont supposées de difficulté 1. On suppose qu'une représentation de ce graphe par un dictionnaire est fournie dans une variable globale G1 et que la liste des voisins est triée par ordre alphabétique.

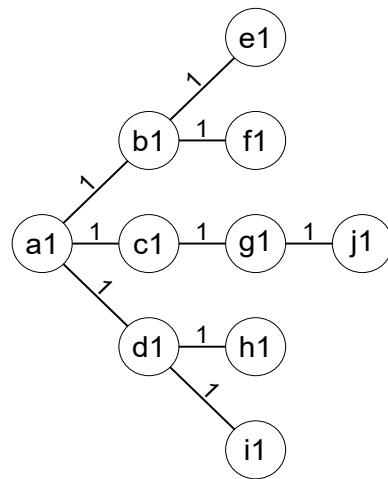


Figure 4 - Graphe G1

Q28. Lister les sommets visités par l'appel `dijkstra(G1, "a1", "j1")`, puis par l'appel `dijkstra(G1, "j1", "a1")`.

III.3 - Troisième idée : l'algorithme de Dijkstra bidirectionnel

On constate à l'aide des deux questions précédentes (**Q27** et **Q28**) que, en fonction de la structure du graphe, il est parfois préférable de chercher un chemin qui part du sommet d'arrivée vers le sommet de départ plutôt qu'un chemin qui part du sommet de départ vers le sommet d'arrivée.

L'algorithme de *Dijkstra bidirectionnel* combine ces deux stratégies : au cours de l'algorithme, on va effectuer successivement soit une recherche depuis le point de départ (appelée étape *forward*) soit une recherche depuis le point d'arrivée (appelée étape *backward*). À chaque étape, on choisit ainsi un sommet qui réalise le minimum de la distance soit au sommet de départ, soit au sommet d'arrivée et on y applique l'algorithme de Dijkstra classique.

Précisons l'algorithme. Notons \mathcal{S} l'ensemble des sommets du graphe. Pour tout sommet $i \in \mathcal{S}$, on note $dF(i)$ (respectivement $dB(i)$) la distance minimale actuellement calculée entre le sommet de départ (respectivement d'arrivée) et le sommet i . Ces distances sont actualisées au cours de l'algorithme et valent initialement à l'infini. Nous allons maintenir, c'est-à-dire mettre à jour pendant l'exécution de l'algorithme, plusieurs variables :

- $aVisiterF$, $dejaVisitesF$: associées à l'algorithme de Dijkstra partant du point de départ (partie Forward),
- $aVisiterB$, $dejaVisitesB$: associées à l'algorithme de Dijkstra partant du point d'arrivée (partie Backward),
- $F_{min} = \min\{dF(i), i \in aVisiterF\}$: distance minimale du sommet de départ à un sommet non encore visité par l'algorithme forward,
- $B_{min} = \min\{dB(i), i \in aVisiterB\}$: distance minimale du sommet d'arrivée à un sommet non encore visité par l'algorithme backward,
- $BF_{min} = \min\{dF(i) + dB(i), i \in \mathcal{S}\}$: longueur minimale d'un chemin reliant le sommet de départ au sommet d'arrivée,
- $distanceF$ (respectivement $distanceB$) : dictionnaire dont les clés sont les sommets et les valeurs sont des couples dont la première composante est la plus petite distance d'un chemin qui relie le sommet depuis le point de départ (respectivement depuis le point d'arrivée) et la seconde est le sommet précédent dans un chemin qui réalise cette distance.

À chaque étape :

- Si $BF_{min} \leq F_{min} + B_{min}$, l'algorithme termine : tout chemin qui réalise le minimum BF_{min} est un chemin optimal.
- Sinon,
 - si $F_{min} < B_{min}$, on choisit un sommet qui réalise F_{min} et on applique une étape forward de Dijkstra depuis ce sommet, c'est-à-dire qu'on appelle la fonction `unpas` avec les variables `forward` en mettant à jour les variables `dejaVisitesF`, `aVisiterF` et `distanceF`,
 - si $B_{min} < F_{min}$, on choisit un sommet qui réalise B_{min} et on applique une itération de l'algorithme de Dijkstra backward depuis ce sommet, c'est-à-dire qu'on appelle la fonction `unpas` avec les variables `backward` en mettant à jour les variables `dejaVisitesB`, `aVisiterB` et `distanceB`,
 - si $B_{min} = F_{min}$, on choisit un sommet qui réalise ce minimum dans l'ensemble qui contient le moins d'éléments parmi `aVisiterF` et `aVisiterB` et on réalise une étape de Dijkstra à partir de ce sommet. Dans le cas où les deux ensembles contiennent le même nombre d'éléments, on préférera une recherche forward.

- On actualise F_{min} , B_{min} et BF_{min} en parcourant l'ensemble des sommets pour lesquels un chemin entre ce sommet et les sommets d'arrivée et de départ a déjà été calculé.

Q29. Compléter la dernière ligne des tableaux du **DR** qui recense les étapes successives de l'algorithme de Dijkstra bidirectionnel sur le graphe G de la **figure 3** avec a comme sommet de départ et f comme sommet d'arrivée. À chaque étape, on précise si le sommet est visité par la recherche forward (F) ou par la recherche backward (B). Dans chaque case sont précisées les valeurs de $distanceF$ et $distanceB$. Pour simplifier la lisibilité, le tableau est découpé en deux parties.

Q30. Justifier si renvoyer la quantité BF_{min} dès qu'un sommet a été atteint par les recherches forward et backward permet de trouver le chemin de longueur minimale.

On donne une implémentation partielle de la fonction `dijkstra_bidirectionnel` qui, étant donné un graphe `graph`, un sommet de départ `Sd` et un sommet final `Sf`, renvoie la distance minimale reliant `Sd` à `Sf` en utilisant l'algorithme de Dijkstra bidirectionnel.

```

1 | def dijkstra_bidirectionnel(graph :dict, Sd :str, Sf :str) -> float :
2 |     aVisiterF = [Sd] # Liste des sommets à visiter Forward
3 |     dejaVisitesF = [] # Liste des sommets déjà visités Forward
4 |     aVisiterB = ... # Liste des sommets à visiter Backward
5 |     dejaVisitesB = ... # Liste des sommets déjà visités Backward
6 |     # Dictionnaire des distances Forward
7 |     distanceF = {s :float('inf'), Sd} for s in graph}
8 |     # Dictionnaire des distances Backward
9 |     distanceB = {s :float('inf'), Sf} for s in graph}
10 |    distanceF[Sd] = (0, Sd)
11 |    distanceB[Sf] = (0, Sf)
12 |    Fmin, Bmin, BFmin = 0, 0, float("inf")
13 |    while ..... :
14 |        if Fmin < Bmin or \
15 |            (Fmin == Bmin and len(aVisiterF) <= len(aVisiterB)) :
16 |                sTraite = cherche_min(distanceF, aVisiterF)
17 |                unpas(graph, sTraite, distanceF, dejaVisitesF, aVisiterF)
18 |            else :
19 |                sTraite = cherche_min(distanceB, aVisiterB)
20 |                unpas(graph, sTraite, distanceB, dejaVisitesB, aVisiterB)
21 |                Fmin = min([.....[v][0] for v in aVisiterF if v in distanceF])
22 |                Bmin = min([.....[v][0] for v in aVisiterB if v in distanceB])
23 |                L = [..... for v in distanceB if v in distanceF]
24 |                if L == []:
25 |                    BFmin = float("inf")
26 |                else :
27 |                    BFmin = min(L)
28 |    return BFmin

```

Q31. Compléter la fonction `dijkstra_bidirectionnel` qui, étant donné un graphe `graph`, un sommet de départ `Sd` et un sommet final `Sf`, renvoie la distance minimale reliant `Sd` à `Sf` en utilisant l'algorithme de Dijkstra bidirectionnel. Indiquer précisément le contenu des lignes 4, 5, 13, 21, 22 et 23.

La fonction `min(L:list)` renvoie l'élément minimal d'une liste `L`.

Pour tout couple de sommets a et b du graphe, on note $d(a, b)$ la distance minimale d'un chemin qui relie a à b . On admet que l'algorithme de Dijkstra classique est correct et que, à chaque étape, pour tous les sommets s de `dejaVisites`, la valeur $d(a, s)$ est égale à la distance calculée `distance[s][0]`. De plus, tous les sommets non visités sont à une distance de a supérieure à la plus grande des distances déjà calculées.

On s'intéresse maintenant à la correction partielle de l'algorithme de Dijkstra bidirectionnel. Supposons alors par l'absurde que l'algorithme de Dijkstra bidirectionnel a terminé mais que la distance `BFmin` ne soit pas la plus petite distance reliant le sommet de départ s au sommet d'arrivée t . Alors, il existe un chemin $s = s_0, s_1, \dots, s_n = t$ qui est de distance d strictement inférieure à `BFmin`. Soit s_i un sommet de ce chemin.

Q32. Montrer que s_i appartient soit à `dejaVisitesF` soit à `dejaVisitesB`.

Q33. En déduire qu'il existe un indice i_0 tel que s_{i_0} a été visité par la recherche forward et s_{i_0+1} l'a été par la recherche backward.

Q34. En déduire la correction partielle de l'algorithme de Dijkstra bidirectionnel.

FIN

SESSION 2025**TSI5IN**

ÉPREUVE SPÉCIFIQUE - FILIÈRE TSI

INFORMATIQUE

Durée : 3 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

RAPPEL DES CONSIGNES

- Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.
 - Ne pas utiliser de correcteur.
 - Écrire le mot FIN à la fin de votre composition.
-

Les calculatrices sont interdites.

Le sujet est composé de quatre parties indépendantes.

Important : vous pouvez librement utiliser les fonctions issues des questions précédentes, même si vous ne les avez pas traitées. Vous devez répondre directement sur le **Document Réponse**, soit à l'emplacement prévu pour la réponse lorsque celle-ci implique une rédaction, soit en complétant les différents programmes en langage Python.

Énoncé : 11 pages

Annexe : 1 page

Document Réponse (DR) : 8 pages

**Seul le Document Réponse doit être rendu dans son intégralité
(le QR Code doit être collé sur la première page du DR).**

Dans ce sujet, les fonctions sont définies avec leur signature :

```
ma_fonction(arg1:type1, arg2:type2) → type3
```

Cette notation permet de définir une fonction qui se nomme `ma_fonction` qui prend deux arguments en entrée `arg1` de type `type1` et `arg2` de type `type2`. Cette fonction renvoie une valeur de type `type3`.

Il n'est pas nécessaire de recopier les signatures des fonctions dans le **Document Réponse**, il suffit d'écrire directement :

```
def ma_fonction(arg1,arg2) :  
    #liste d'instructions
```

Détection d'objets dans le cadre de la conduite autonome

L'entreprise Easymile a mis en place à Toulouse des bus 100 % autonomes sur le campus universitaire de Paul Sabatier. Il s'agit d'un projet de recherche mené par l'IRIT et l'Université, mais dès aujourd'hui, les minibus (**figure 1**) qui peuvent accueillir vingt personnes sont en fonction et se déplacent dans un rayon de 5 km. Ces véhicules autonomes nécessitent un niveau élevé d'informations pour fonctionner en toute sécurité et sont donc équipés d'une gamme complète de capteurs.

Ces capteurs collectent et analysent les données enregistrées pour créer une image à 360 degrés de l'environnement, y compris les infrastructures, les autres véhicules, les piétons et tout ce qui se trouve sur le chemin.

Le traitement en temps réel des données permet au système du véhicule autonome de décider comment se comporter pour progresser en toute sécurité sur la route (s'arrêter, partir, ralentir, etc.).

Ce sujet s'intéresse à une partie des algorithmes mis en place afin de rendre le projet possible, notamment la partie détection d'obstacles dans l'espace.

En effet, pour détecter les objets, les piétons et les véhicules avec précision et rapidité, il est nécessaire de mettre en place des programmes robustes avec une complexité limitée.

Dans un premier temps, nous nous intéresserons à un algorithme simple de détection d'obstacles, basé sur l'apprentissage supervisé. Il faudra tout d'abord réfléchir à la méthode de localisation dans l'espace du piéton. Dans un deuxième temps, nous nous pencherons sur le traitement de l'image et la détection des obstacles. Enfin, dans le but d'améliorer et de comprendre le fonctionnement des programmes du bus, la collecte d'un grand nombre de données est réalisée à chaque déplacement, c'est l'objet de la dernière partie du sujet.



Figure 1 - Bus Easymile

Partie I - Localisation dans l'espace

Un premier calcul important pour détecter les piétons et les obstacles est la prise en compte de la distance focale de la caméra de détection. Pour cela, un calcul simplifié est réalisé à partir d'une taille réelle standard. Les caméras permettent en effet de relever une matrice de pixels qui donne ainsi une dimension aux formes et aux distances.

Sur la **figure 2**, lorsque A' est confondu avec F' , la relation entre la focale f et la distance réelle est la suivante :

$$D_{réelle} = \frac{\|\overrightarrow{OF'}\| \cdot \|\overrightarrow{AB}\|}{\|A'B'\|} = f \cdot \frac{H_{objet}}{H_{image}} \quad (1)$$

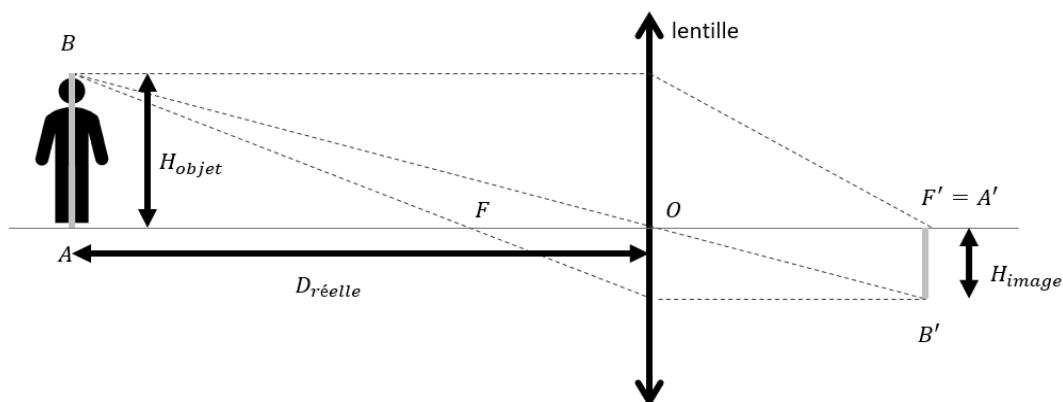


Figure 2 - Calcul simplifié de la focale f

Nous allons tout d'abord créer un dictionnaire avec 5 obstacles "classiques" qui doivent être détectés.

- Q1.** Créer un dictionnaire nommé `dico`, possédant 5 clés de type str : 'adulte', 'enfant', 'animaux', 'véhicule', 'indéterminé'. Chaque clé sera initialisée avec comme valeur une liste vide.

Pour chacune des clés du dictionnaire créé à la question **Q1**, nous allons construire un intervalle de hauteur pouvant correspondre à l'obstacle et réparti à 20 % de part et d'autre de la hauteur moyenne de celui-ci, sous la forme d'une liste de deux éléments :

- pour l'adulte, la hauteur moyenne est fixée à 175 cm, la liste associée à la clé sera alors `[175*0.8, 175*1.2]` ;
- pour l'enfant, la hauteur moyenne est fixée à 110 cm ;
- pour les animaux, la hauteur moyenne est fixée à 50 cm ;
- pour les véhicules, la hauteur moyenne est fixée à 200 cm ;
- pour les autres valeurs indéterminées, le couple associé à la clé sera composé des valeurs les plus basses `[0, 50*0.8]`.

- Q2.** Écrire une suite d'instructions qui modifie votre dictionnaire initialisé à la question **Q1** en associant le couple d'intervalles à chacune des clés du dictionnaire.

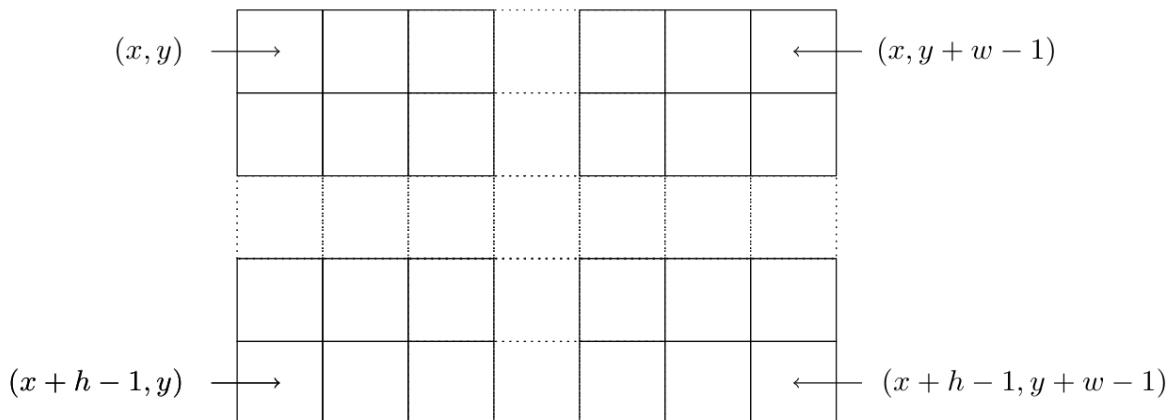
- Q3.** Écrire une fonction `obstacles_rencontres(dico : dict, hauteur : float) → list[str]` qui permet de vérifier que pour une hauteur donnée, le nom de l'obstacle est bien renvoyé. Elle prend donc en entrée une hauteur réelle en cm et le dictionnaire créé aux questions précédentes, contenant les couples clé, intervalle de hauteur. Dans le cas d'au moins une possibilité, tous les noms seront renvoyés dans une liste de chaîne de caractères. Si par contre la hauteur n'est dans aucun intervalle, la liste vide sera renvoyée.

Par exemple, pour une hauteur de 170 cm, la fonction renvoie `['adulte', 'véhicule']`.

Les algorithmes de détection d'obstacles doivent traiter les images provenant des caméras. Les images issues de la caméra sont ici considérées comme un tableau à deux dimensions qui contient des pixels.

Nous allons ici travailler dans l'une de ces images de pixels pour comprendre la détection des obstacles. On dispose d'une liste `faces_detecc` constituée de quadruplets (x, y, h, w) correspondant chacun à un contour détecté dans une image captée. Ici x, y désignent les coordonnées du coin supérieur gauche dans l'image, h la hauteur du contour, w sa largeur.

Nota : il est important de rappeler que les images sont parcourues de haut en bas avec x et h travaillant sur les lignes, y et w sur les colonnes, x, y, h et w sont en nombre de pixels. Le dernier pixel de l'image se situe donc à la coordonnée $(x + h - 1, y + w - 1)$.



- Q4.** Écrire une fonction `focale(faces_detecc : list, Dr : float, Hr : float) → list[float]` prenant pour argument une liste `faces_detecc` de contours, une distance `Dr` réelle et une hauteur `Hr` réelle données, et qui renvoie la liste contenant la valeur de la focale définie dans l'équation (1) pour chaque contour détecté.
- Q5.** Écrire une fonction `moy(L:list[float]) → float` qui renvoie la moyenne des éléments de la liste `L`.

Partie II - Traitement de l'image

II.1 - Calcul de luminance

Nous allons nous intéresser maintenant au format des images et plus particulièrement au passage en niveau de gris, c'est-à-dire le calcul de la luminance moyenne utile à la détection de contour. La teinte d'un pixel peut être représentée de plusieurs façons. Une méthode courante, basée sur la synthèse additive, consiste à la décomposer en trois composantes qui correspondent aux couleurs rouge, vert et bleu.

On parle de représentation RVB (Rouge, Vert et Bleu). Chacune des trois composantes donne l'intensité de la couleur correspondante dans la teinte finale sous forme d'un nombre entier compris entre 0 et 255. 0 indique l'absence de cette couleur et 255 l'intensité maximale. Ainsi, le triplet (0, 0, 0) désigne un pixel noir et (255,255,255) un pixel blanc.

Rappelons que les images sont représentées sous la forme d'une liste de lignes où chaque ligne est une liste de triplets RVB. Ainsi, on accède au pixel de coordonnées (x, y) de l'image I par l'expression $I[x][y]$.

- Q6.** Sachant qu'une composante de couleur est représentée sur 8 bits, préciser l'espace mémoire nécessaire pour stocker un pixel, puis pour stocker une image de taille 8 000 x 6 000 de pixels. La réponse pour l'image sera donnée en Mo (1 Mo = un million d'octets).

Pour représenter une image en niveau de gris, il suffit d'une valeur par pixel représentant l'intensité de gris entre le noir et le blanc, c'est la luminance. Pour convertir une image en couleurs en niveaux de gris, plusieurs méthodes sont possibles. Faire la simple moyenne des composantes R, V et B d'un pixel donne visuellement des résultats décevants.

On procède donc selon le protocole suivant :

- notons C une des composantes R, V ou B d'un pixel. On pose $C_1 = C/255$; elle sera transformée en une variable C_{lin} selon la définition suivante :
 - o $C_{lin} = \frac{C_1}{12,92}$ si $C_1 \leq 0,04045$;
 - o $C_{lin} = \left(\frac{C_1 + 0,055}{1,055} \right)^{2,4}$ sinon ;
- puis on calcule la valeur " linéaire " du niveau de gris Y_{lin} avec la formule :

$$Y_{lin} = 0,2126 \cdot R_{lin} + 0,7152 \cdot V_{lin} + 0,0722 \cdot B_{lin} ;$$
- enfin l'intensité Y du pixel de l'image en niveau de gris sera calculée comme suit :
 - o $Y = \lfloor 255 * 12,92 * Y_{lin} \rfloor$ si $Y_{lin} \leq 0,0031308$;
 - o $Y = \lfloor 255 * (1,055 * Y_{lin}^{1/2,4} - 0,055) \rfloor$ sinon.
($\lfloor x \rfloor$ désigne la partie entière de x)

- Q7.** Écrire une fonction `Clinear(C:int) → float`, qui prend en argument une composante de couleur C d'un pixel et qui renvoie la valeur C_{lin} décrite plus haut.

- Q8.** Écrire une fonction `Intensite(pix:tuple) → int` qui prend en argument un triplet de trois composantes correspondant à un pixel au format RVB et qui renvoie la valeur Y du niveau de gris correspondant.

- Q9.** Écrire une fonction `init(h:int, w:int) → list` prenant en argument les 2 entiers h et w caractérisant la taille d'une image et qui renvoie une liste de h listes remplie de w zéros.

Par exemple, `init(2, 3)` renvoie `[[0, 0, 0], [0, 0, 0]]`.

- Q10.** Écrire une fonction `NiveauxGris(I:list) → list` prenant en argument une image I au format RVB et qui renvoie une image de même taille en niveau de gris.

- Q11.** Estimer la complexité asymptotique pour calculer la luminance de tous les pixels d'une image ayant h lignes de pixels et w colonnes, en fonction des variables h et w .

II.2 - Le concept d'image-intégrale

On travaille ici avec des images en niveau de gris. Comme vu précédemment, la valeur du niveau de gris d'un pixel s'appelle la luminance du pixel. Lors du processus de détection des contours, on est amené à faire un très grand nombre de fois la moyenne des luminances sur des sections (des portions) de l'image d'origine. Un calcul naïf de cette moyenne entraîne alors une complexité temporelle importante.

La méthode de l'image-intégrale permet, au prix du pré-calcul initial de l'image-intégrale, d'accélérer considérablement le calcul des luminances moyennes.

La **figure 3** montre un exemple du principe de l'image-intégrale qui consiste à sommer les intensités au fur et à mesure que l'on balaye les pixels au lieu de garder la valeur d'intensité du pixel seul.

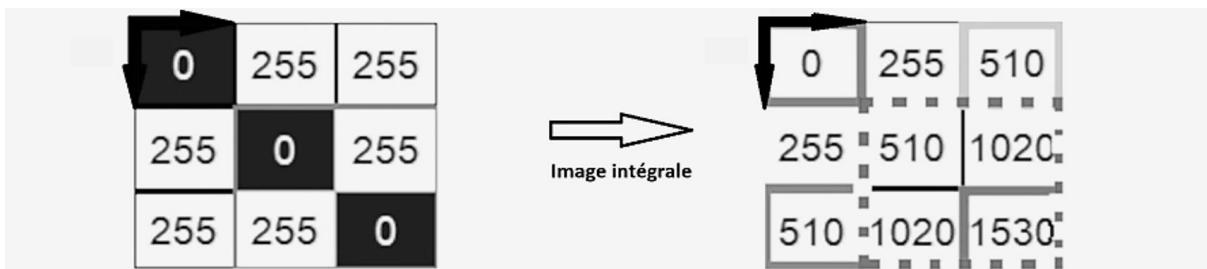


Figure 3 - Somme des pixels sur une section 3x3 avec la méthode d'image intégrale (à gauche, les 9 pixels d'origine, à droite la somme des pixels effectuée par balayage)

Dans ce qui suit, h désigne le nombre de lignes et w le nombre de colonnes de l'image. Les sections de l'image sur lesquelles on calculera la luminance moyenne auront n lignes et p colonnes. Ces quatre valeurs h, w, n, p sont fixées au début du traitement.

Q12. Expliquer les deux lignes du DR.

Q13. Créer une fonction `coef_integral(M:list,x:int,y:int) → int` qui renvoie l'intégrale de l'image jusqu'à un point (x, y) suivant la formule (2), la somme étant prise nulle quand elle ne comporte aucun terme :

$$I(x, y) = \sum_{j=0}^x \sum_{k=0}^y M_{j,k}. \quad (2)$$

Q14. À l'aide de cette fonction `coef_integral`, donner les instructions permettant de créer la matrice image-intégrale `M_int` d'une matrice `M` (liste de listes). Les coefficients `M_int[x][y]` de cette matrice sont donnés par la valeur `I[x][y]` définie à la formule (2).

La formule qui calcule ensuite la luminance moyenne \bar{m} sur une section de coin supérieur gauche (x, y) est la suivante :

$$\bar{m} = \frac{1}{n \cdot p} (M_{int_{x+n,y+p}} - M_{int_{x+n,y}} - M_{int_{x,y+p}} + M_{int_{x,y}}), \quad (3)$$

n et p étant respectivement la hauteur et la largeur de la section de l'image.

Q15. Écrire une fonction `lumi_section(M_int:list,x:int,y:int,n:int,p:int) → float` qui prend pour arguments une image intégrale `M_int`, les coordonnées `x, y` du coin supérieur gauche d'une section, la hauteur `n` et la largeur `p` de cette section et qui renvoie la luminance moyenne de la section correspondante de l'image `M`.

II.3 - Détection des obstacles : algorithme de Viola-Jones

L'algorithme de Viola-Jones implémenté ici prend en argument la matrice image intégrale `M_int`.

Le calcul de luminance moyenne est ensuite réalisé sur chacune des sections de l'image en séparant les sections en deux sur la largeur, c'est-à-dire que l'on peut définir la moyenne gauche \bar{m}_g allant de y à $y + \lfloor p/2 \rfloor$ inclus et la moyenne droite \bar{m}_d allant de $y + \lfloor p/2 \rfloor$ à $y + p$.

Le coefficient C_{obs} associé à chaque section est alors calculé suivant l'équation (4).

$$C_{obs} = \max(\bar{m}_d, \bar{m}_g) - \min(\bar{m}_d, \bar{m}_g). \quad (4)$$

Ce coefficient est comparé à une constante `C_ref` caractéristique pseudo-Haar qui vient d'un classifieur (données associées à une base de données sur chaque objet particulier). Si la corrélation dépasse un seuil défini (compris lui aussi entre 0 et 1), c'est-à-dire si $C_{obs}/C_{ref} > \text{seuil}$, le coin supérieur gauche de cette section est alors ajouté à la liste renvoyée en sortie.

L'algorithme permet ainsi de donner la liste de tous les coins supérieurs des objets détectés avec les caractéristiques pseudo Haar du classifieur.

Q16. Écrire la fonction `detection(M_int:list,n:int,p:int,C_ref:int,seuil:float) → list` qui prend en argument une matrice image intégrale `M_int`, la taille des sections associées `n, p`, la caractéristique pseudo-Haar `C_ref` et le seuil imposé. Elle réalise l'algorithme de Viola-Jones et renvoie en sortie la liste des coins des objets détectés.

On considère que les initialisations et définitions des matrices `M_int` et tailles des sections ont bien été réalisées en amont. Les fonctions natives de Python `min, max` sont autorisées.

Partie III - Optimisation - Méthode KNN

La méthode KNN est une méthode d'apprentissage dit supervisé ; les données sont déjà classées par groupes clairement identifiés et on cherche à quels groupes appartiennent de nouvelles données. Son utilisation semble pertinente ici car on dispose d'un volume important de données étiquetées.

Le principe de la méthode est simple. Après avoir calculé la distance euclidienne entre toutes les données connues des obstacles et les données d'un nouvel obstacle à classer, on extrait les K données connues les plus proches. L'appartenance du nouvel obstacle à un groupe est obtenue en cherchant le groupe majoritaire, c'est-à-dire, le groupe qui apparaît être le plus représentatif parmi les K données connues.

On note $X_i, i \in \llbracket 0, N - 1 \rrbracket$, le i^e vecteur ligne du tableau de données `data`, correspondant aux données d'un obstacle i déjà classé. Ce vecteur possède n coordonnées. Le tableau `data` contient N lignes correspondant à N obstacles différents, sa dimension est donc $N * n$. On cherche alors à déterminer à quelle classe appartient un nouvel obstacle qui lui aussi sera représenté par un vecteur `Z` de coordonnées $(z_0, z_1, \dots, z_{n-1})$.

Les calculs matriciels sont beaucoup plus faciles à manipuler avec le module `numpy` qui est explicité en **Annexe**. Il est cependant possible de réaliser la totalité des questions suivantes en utilisant des listes de listes ou des `array` de `numpy`.

Le candidat choisira librement le type qu'il préfère manipuler (liste de listes ou `array` avec les fonctions `numpy` associées) mais pour la suite des questions, nous considérons que le module `numpy` est importé comme suit :

```
import numpy as np.
```

III.1 - Données

Avant de calculer la distance euclidienne entre les différents vecteurs, il est préférable de normaliser ceux-ci pour éviter que l'un d'entre eux ait plus de poids par rapport à un autre. Nous allons ainsi ramener toutes les coordonnées d'un vecteur entre 0 et 1 grâce à une technique de normalisation reposant sur une transformation affine.

Considérons un vecteur $X = (x_0, x_1, \dots, x_{n-1})$. On note $\min(X)$ son minimum et $\max(X)$ son maximum. Le vecteur normalisé associé à X , noté $X_{norm} = (x_{norm,0}, x_{norm,1}, \dots, x_{norm,n-1})$ est défini par :

$$\forall j \in \{0, 1, \dots, n-1\}, x_{norm,j} = \frac{x_j - \min(X)}{\max(X) - \min(X)}. \quad (5)$$

- Q17.** Écrire une fonction `min_max(X : list) → tuple` qui renvoie les valeurs du minimum et du maximum d'un vecteur X passé en argument. La fonction devra être de complexité linéaire. Dans cette question uniquement, l'usage de fonctions `min`, `max` prédéfinies dans le langage Python est interdit.
- Q18.** Écrire une fonction `coeff_normalise(data:list) → list` qui prend en argument un tableau de données non normalisées `data` et qui renvoie un nouveau tableau `data_norm` contenant cette fois les données normalisées.
- Q19.** Écrire une fonction `distance(Z:list, data:list) → list` qui parcourt les N lignes du tableau `data` où les données sont normalisées et calcule les distances euclidiennes entre le n -uplet Z et chaque n -uplet X du tableau de données connues (on rappelle que X représente une ligne du tableau `data`). La fonction doit renvoyer une liste de taille N contenant les distances entre chaque n -uplet X et le n -uplet Z .

III.2 - Détermination des K plus proches voisins

Pour déterminer les K plus proches voisins avec K un entier choisi arbitrairement, il suffit d'utiliser un algorithme de tri efficace. La liste L à trier est une liste de listes à deux éléments contenant :

- la distance entre le n -uplet à classer et un n -uplet connu (on trie par ordre croissant sur ces valeurs) ;
- la valeur du type d'obstacle correspondant au n -uplet connu.

On retient l'algorithme suivant :

```
def tri (L):
    """Reçoit une liste L en argument.
    Renvoie la liste triée correspondante"""
    if len(L) <= 1:
        return T
    else:
        m= len(L) //2
        L1 = []
        for x in range(m):
            L1.append(L[x])
        L2 = []
        for x in range(m, len(L)):
            L2.append(L[x])
        return fusion ( tri (L1) , tri (L2))

def fusion (T1,T2):
    """ Reçoit en arguments deux listes triées T1 et T2.
    Renvoie une liste triée obtenue en fusionnant les deux listes T1 et T2 """
    if T1 == []:
        . . . . . . . . . . . . . . . . . #ligne 1 à compléter
    if T2 == []:
        . . . . . . . . . . . . . . . . . #ligne 2 à compléter
    if T1[0][0] <T2[0][0]:
        return [T1[0]]+fusion (T1[1:] ,T2)
    else:
        . . . . . . . . . . . . . . . . . #ligne 3 à compléter
```

Q20. Compléter sur le DR les lignes 1 à 3 de la fonction `fusion` pour que la fonction `tri` effectue le travail spécifié.

Q21. Lorsque l'on fait des tests de performance sur cet algorithme, on obtient des résultats très décevants par rapport à d'autres algorithmes de tris. Identifier la ou les causes de ces problèmes de performance. Le coût du "slicing" est explicité en **Annexe**.

L'algorithme de la méthode KNN est décrit par la fonction python suivante :

```
def KNN(data ,typeObs ,z,K,nb):
    #partie 1
    T= []
    dist = distance(z,data)
    for i in range(len(dist)):
        T.append([dist[i] , i ])
    L=tri (T)

    #partie 2
    select = [0]*nb
    for i in range(K):
        select [typeObs[L[i][1]]]+=1

    #partie 3
    ind = 0
    res = select [0]
    for k in range(1,nb):
        if select [k] > res:
            res = select [k]
            ind = k
    return ind
```

`typeObs` permet de récupérer la classe d'un objet déjà classifié, K représente le nombre de voisins proches retenus, nb correspond au nombre de type d'obstacles (5 dans l'exemple de la question **Q1**) et z est le vecteur représentant l'obstacle à classer.

Q22. Expliquer ce que font globalement les parties 1, 2 et 3 de l'algorithme. Préciser ce que représentent les variables locales `T`, `dist`, `select`, `ind`.

III.3 - Validation de l'algorithme

Pour tester l'algorithme, on utilise un nouveau jeu de données normalisées représentants 200 obstacles connus de 5 types différents (exemple de la question **Q1**).

On note `datatest` le tableau contenant ces données et `typetest` la liste des types connus pour ces obstacles. Le but est de déterminer si les prévisions de type fournies par l'algorithme KNN pour chaque obstacle correspondent aux types réels des obstacles stockés dans `typetest`. Pour cela, on applique l'algorithme sur chaque élément de ce jeu de données pour une valeur de K fixée. On définit la fonction suivante qui renvoie une matrice `mat` :

```
def test_KNN(datatest , typetest , data , typeObs , K, nb):
    typepredit = []
    for i in range(len(datatest)):
        res =KNN(data ,typeObs ,datatest [i] ,K,nb)
        typepredit.append(res)

    mat = [[0 for j in range(nb)] for i in range(nb)]
    for i in range(len(datatest)):
        mat[typetest[i]][typepredit[i]] += 1
    return mat
```

On obtient pour $K = 8$ la matrice suivante :
$$\begin{pmatrix} 27 & 4 & 3 & 5 & 2 \\ 0 & 18 & 2 & 4 & 3 \\ 5 & 2 & 40 & 7 & 2 \\ 2 & 5 & 7 & 36 & 4 \\ 4 & 3 & 1 & 1 & 19 \end{pmatrix}$$

Q23. Donner le nom de cette matrice. Indiquer l'information apportée par la diagonale de la matrice. Exploiter les valeurs de la première ligne de cette matrice en expliquant les informations que l'on peut en tirer. Donner enfin l'efficacité (ou pourcentage de réussite) de l'algorithme KNN pour cet exemple.

Il faut savoir que l'algorithme de classification qui est réellement utilisé dans les bus Easymile est basé sur un réseau de neurones à 10 couches, ce qui améliore grandement son efficacité !

Partie IV - Gestion dans la base de données

Un rendu continu de l'environnement immédiat et la prédiction des changements possibles sont fondamentaux pour comprendre et assurer le bon fonctionnement des bus. Les véhicules autonomes sont en communication permanente avec le centre de supervision.

L'ensemble des informations est enregistré dans une base de données et nous allons ici nous intéresser à seulement 3 tables qui sont mises à jour toutes les semaines :

- La table **Passagers** qui a pour attributs : Id1 clé primaire (INT), numéro du bus (INT), date_jour (JJ_MM_AA), fonctionnement (BOOL), heure d'entrée du passager (HH_MM_SS).

- La table **Detections** qui a pour attributs : Id2 clé primaire (INT), Id_bus (INT) clé étrangère correspondant à l'identifiant Id1 de la table 1, nombre d'obstacles (INT), distances (STR), date_obs (JJ_MM_AA), arrêt (BOOL).
- La table **Acquisition** qui a pour attributs : Id3 clé primaire (INT), capteurs (INT), Id_bus (INT) clé étrangère correspondant à l'identifiant Id1 de la table 1, défaillance (VARCHAR interruption/perte de signal), nombre de réparations déjà effectuées sur le capteur (INT).

Voici un extrait du contenu de ces tables :

Passagers				
Id1	Numero_bus	Date_jour	Fonctionnement	Heure_passager
...
11	1	01_01_2024	True	10_10_03
12	1	01_01_2024	True	10_10_55
13	3	02_01_2024	False	
...

	Detections				
Id2	Id_bus	Obstacles	Distances	Date_obs	Arret
...	
21	11	15	'3.4, 2.6, ..., 4.5'	01_01_2024	True
22	24	3	'2.3, 4.5, 6.5'	01_01_2024	True
23	26	1	'10.4'	02_01_2024	False
...	

	Acquisition				
Id3	Capteurs	Id_bus	Defaillance	Reparations	
...	
31	9	13	'Interruption'	2	
32	5	13	'Perte de signal'	3	
33	2	15	'Perte de signal'	1	
...	

Q24. Expliquer ce que fait la requête suivante :

```
SELECT DISTINCT(date_jour) FROM Passagers WHERE numero_bus=3 and not(fonctionnement)
```

Q25. Écrire une requête en SQL permettant d'obtenir le nombre de passagers par jour dans le bus n°1.

Q26. Écrire une requête en SQL qui permet de connaître, à une date du jour donnée, les numéros de bus et les numéros de capteurs qui ont eu une " Perte de signal ".

Q27. Écrire une requête en SQL qui permet de connaître les Id_bus qui ont eu plus de dix réparations au total et les classer par ordre décroissant du nombre de réparations.



Epreuve d'Informatique et Modélisation de Systèmes Physiques

Durée 4 h

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, d'une part il le signale au chef de salle, d'autre part il le signale sur sa copie et poursuit sa composition en indiquant les raisons des initiatives qu'il est amené à prendre.

L'usage de calculatrices est interdit.

AVERTISSEMENT

La **présentation**, la lisibilité, l'orthographe, la qualité de la **rédaction, la clarté et la précision** des raisonnements entreront pour une **part importante** dans l'**appréciation des copies**. En particulier, les résultats non justifiés ne seront pas pris en compte. Les candidats sont invités à encadrer les résultats de leurs calculs.

CONSIGNES :

- Composer lisiblement sur les copies avec un stylo à bille à encre foncée : bleue ou noire.
- L'usage de stylo à friction, stylo plume, stylo feutre, liquide de correction et dérouleur de ruban correcteur est strictement interdit. Les surveillants et surveillantes se réservent le droit de les confisquer.
- Remplir sur chaque copie en MAJUSCULES toutes vos informations d'identification : nom, prénom, numéro inscription, date de naissance, le libellé du concours, le libellé de l'épreuve et la session.
- Une feuille, dont l'entête n'a pas été intégralement renseigné, ne sera pas prise en compte.
- Il est interdit aux candidats de signer leur composition ou d'y mettre un signe quelconque pouvant indiquer sa provenance. La présence d'une information d'identification en dehors du cartouche donnera lieu à un point de pénalité et la page concernée pourra être soustraite de la correction.

" Les applications numériques sont attendues avec un (voire deux) chiffres significatifs. Le jury prendra pleinement en compte le fait que l'épreuve est sans calculatrice et de ce fait sera tout à fait tolérant sur la précision des résultats numériques, l'essentiel étant l'ordre de grandeur ".

Présentation de la problématique
AUTOUR DE LA GÉOLOCALISATION
PAR SATELLITES

La *géolocalisation par satellites* ou GNSS (pour *Géolocalisation et Navigation par un Système de Satellites*) est une technologie permettant à tout utilisateur de déterminer sa position n'importe où sur Terre à l'aide d'un récepteur adapté, qui capte et traite des signaux radio émis par des satellites en orbite autour de la Terre. Il existe quatre grands services de GNSS, dont le *Global Positioning System* (GPS) américain mis en service en 1995, et le système *Galileo* européen, partiellement opérationnel depuis 2016.

1. Fonctionnement

Un système de GNSS repose sur trois constituants principaux (figure 1) :

1. une *constellation de satellites*, suffisamment nombreux pour que plusieurs d'entre eux soient en permanence visibles en tout point de la surface de la Terre et qui émettent en permanence des signaux radio contenant des repères temporels et des informations sur leurs localisations ;
2. des *stations de contrôle* au sol, qui suivent les satellites, identifient les erreurs dans les messages qu'ils envoient (sur les positions, vitesses et temps), les corrigent et leur envoient des rectificatifs ;
3. un *récepteur* par utilisateur, qui capte les signaux des satellites et les utilise pour se localiser.

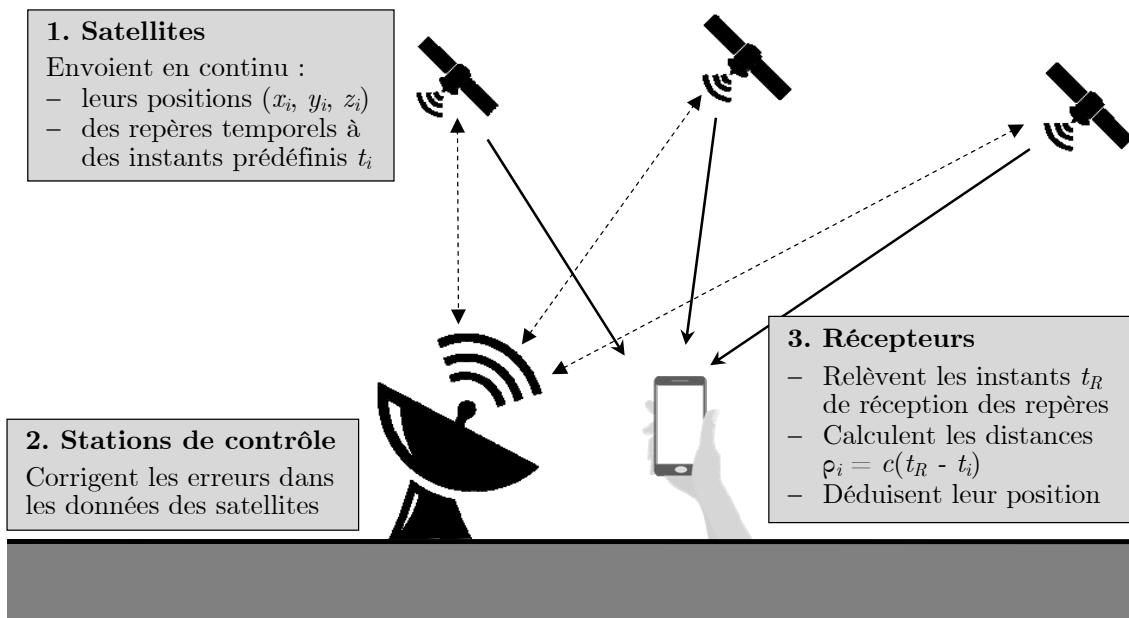


FIGURE 1 – Constituants d'un GNSS et informations transmises

Pour se localiser, le récepteur mesure la distance qui le sépare de plusieurs satellites. Il est pour cela muni d'une horloge qui lui permet, lorsqu'il détecte les repères temporels contenus dans les signaux, de relever leur instant de réception t_R . Ces repères étant émis à des instants précis t_i , il en déduit les distances $\rho_i = c(t_R - t_i)$, où c est la vitesse de propagation des ondes électromagnétiques (figure 1). En théorie, le récepteur a besoin d'au moins trois distances pour se localiser. En pratique, compte tenu des imprécisions qui affectent les mesures du temps, il en utilise au moins quatre. Le calcul géométrique de sa position à partir des distances (et des positions) des satellites s'appelle *trilateration* et est illustré figure 2.

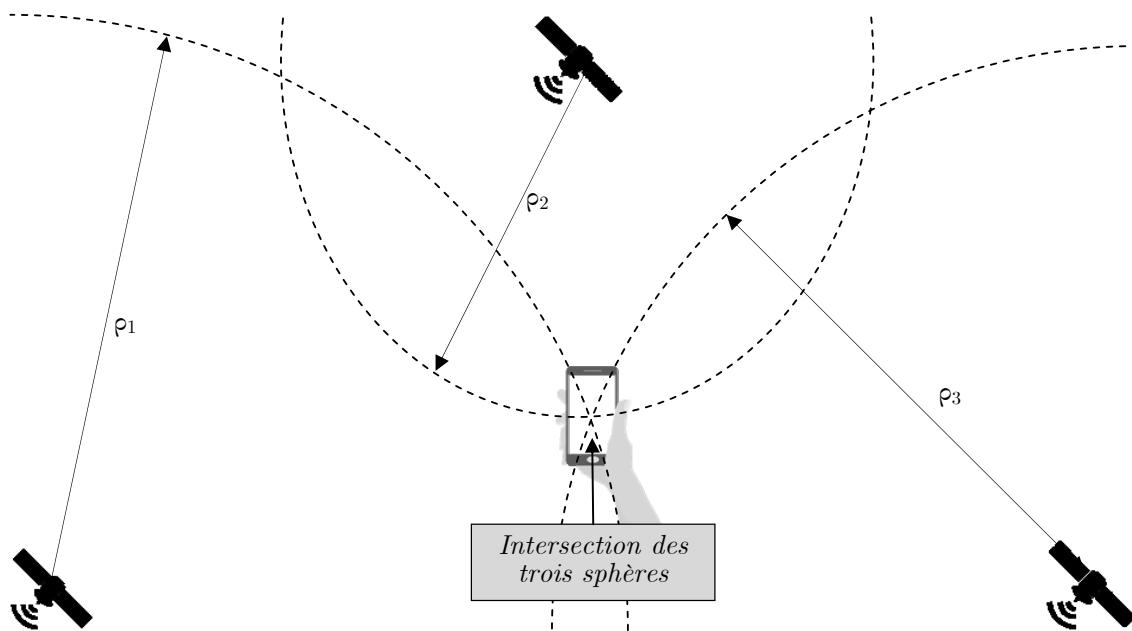


FIGURE 2 – Exemple de trilateration dans le plan

Les avantages de cette technologie sont nombreux : le positionnement est précis (quelques mètres pour la version basique du GPS américain), le service est disponible partout sur Terre sans interruption, l'information est mise à jour en moins d'une seconde une fois que le récepteur est initialisé, et le nombre d'utilisateurs est illimité puisque le récepteur est purement passif (il n'émet rien). La GNSS a ainsi connu un succès considérable : de nos jours, il existe plusieurs milliards de récepteurs GPS.

2. Travail demandé

Ce sujet comporte trois parties indépendantes :

1. la *modélisation des satellites* : il s'agit de prévoir leurs mouvements et de modéliser la propagation des signaux en direction de la Terre ;
2. la simulation du *décodage des signaux*, qui présente une partie du traitement du signal effectué par le récepteur pour identifier les satellites émetteurs ainsi que la distance le séparant de chacun d'eux ;
3. la simulation d'une *recherche d'itinéraires* qui est une application courante des GNSS, avec la recherche d'un plus court chemin et la mise en œuvre de statistiques sur des temps de trajet.

Première partie
MODÉLISATION DES SATELLITES

A. Mouvements et trajectoires

On s'intéresse au mouvement d'un satellite GPS autour de la Terre, dont la trajectoire est circulaire à une altitude $h = 20000$ km. On considère la Terre et le satellite comme des points matériels de masses respectives $M_T = 6 \cdot 10^{24}$ kg et $m = 700$ kg. Le mouvement est plan et circulaire, et on repère la position M du satellite en coordonnées polaires, le centre de la Terre étant à l'origine O (figure 3 ci-dessous). On donne la valeur de la constante de gravitation : $G = 6,7 \cdot 10^{-11}$ SI ainsi que le rayon de la Terre : $R_T = 6400$ km.

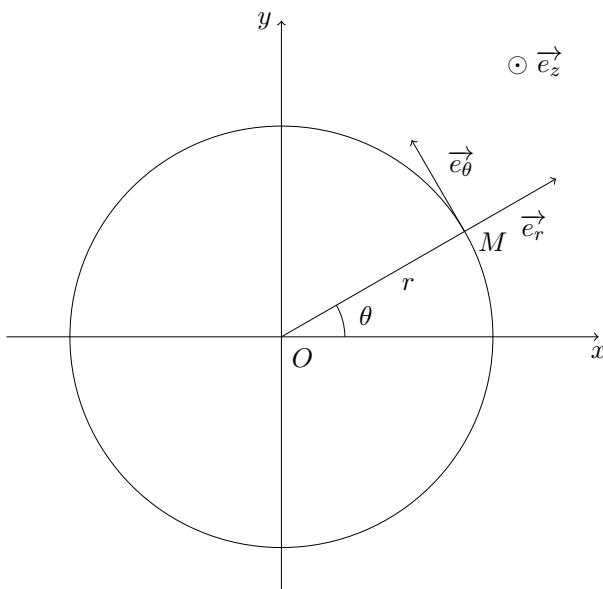


FIGURE 3 – Repérage en coordonnées polaires dans le plan du mouvement

- 1.** Sans démonstration, écrire l'expression de la force vectorielle subie par le satellite et calculer la valeur numérique de sa norme. Toujours sans démonstration, donner l'expression du champ de gravitation $\vec{g}(M)$ créé par la Terre au point M en fonction de m , M_T , h , R_T et G . On pourra utiliser l'aide numérique suivante : $2,6^2 \simeq 6,7$.
- 2.** En considérant la Terre comme sphérique et de masse volumique uniforme, justifier le fait que son champ de gravitation en un point extérieur à la Terre est le même que celui d'un point matériel situé en son centre affecté de toute la masse de la Terre.
- 3.** En utilisant la deuxième loi de Newton, justifier :
 - la relation $v^2 = \frac{GM_T}{r}$, où v est la vitesse du satellite ;
 - que le mouvement circulaire est nécessairement uniforme.
- 4.** En utilisant la question précédente, retrouver la troisième loi de Kepler dans le cas circulaire.

- 5.** Calculer la période de révolution T_{GPS} des satellites GPS en secondes (rappel : $2,6^2 \simeq 6,7$). Comparer cette valeur avec la période d'un satellite géostationnaire.

On donne ci-dessous la projection sur la surface terrestre de la trajectoire d'un satellite GPS ainsi que quelques valeurs de latitudes et longitudes.

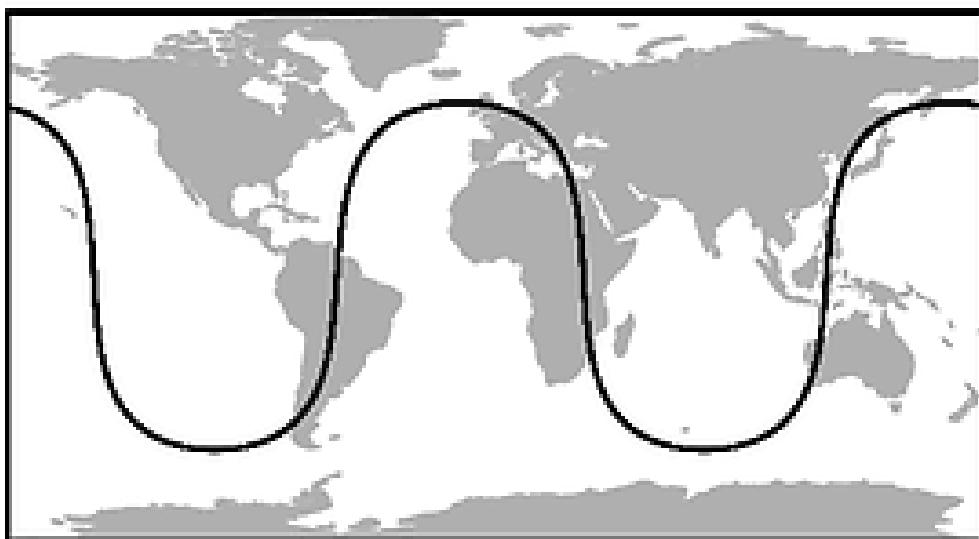


FIGURE 4 – Projection de la trajectoire d'un satellite gps

Lieu	Paris	Londres	Edimbourg	New York	Tokyo	Cap Horn
Latitude	48,5° Nord	51,3° Nord	55,6° Nord	40,4° Nord	35,4° Nord	55,6° Sud
Longitude	2,2° Est	0,1° Ouest	3,1° Ouest	74° Ouest	139,4° Est	67,2° Ouest

FIGURE 5 – Latitudes et longitudes de quelques lieux

- 6.** Evaluer approximativement, à $\pm 5^\circ$ près, l'inclinaison α du plan de l'orbite de ce satellite par rapport au plan équatorial.
- 7.** On admet que le sens de la révolution du satellite est le même que celui de la rotation propre de la Terre. Sur la figure 4, ce sens est-il vers la droite ou vers la gauche (une justification, même succincte, est attendue) ? Quelle est la période du phénomène «le satellite se retrouve à la verticale d'un même point de la Terre» ?

B. Propagation du signal

On s'intéresse dans cette partie à la propagation du signal émis par un des satellites. On considérera qu'il s'agit d'une onde électromagnétique de fréquence environ $f = 1600$ MHz. On donne la vitesse de la lumière dans le vide ainsi que la permittivité diélectrique du vide : $c = 3.10^8$ m.s $^{-1}$ et $\epsilon_0 = 8,8.10^{-12}$ F.m $^{-1}$.

- 8.** En partant des équations de Maxwell, établir l'équation d'onde vérifiée par le champ électrique dans le vide, sans charges ni courants. Quelle est la relation entre μ_0 , ε_0 et c ?

On considère que le satellite est à la verticale du récepteur. Localement, à l'échelle du récepteur, on peut modéliser l'onde qui arrive du satellite par une onde plane monochromatique de la forme :

$$\vec{E} = E_0 \cos(\omega t + kz) \vec{e}_x$$

- 9.** Faire un schéma où figurent le satellite S , le récepteur R ainsi que la base directe ($\vec{e}_x, \vec{e}_y, \vec{e}_z$). Quels sont les direction, sens de propagation et la polarisation relativement à ce système d'axes ?
- 10.** Vérifier que cette onde est bien solution de l'équation de propagation à une condition près à préciser reliant k , ω et c . Calculer numériquement k et la longueur d'onde λ associés.
- 11.** Exprimer le champ \vec{B} associé à la propagation de l'onde.
- 12.** Exprimer le vecteur de Poynting $\vec{\Pi}$ et en déduire l'expression de la puissance moyenne I par unité de surface associée à la propagation de l'onde.
- 13.** Au niveau du récepteur, l'amplitude du champ électrique est $E_0 = 2,5 \cdot 10^{-6} \text{ V.m}^{-1}$. Calculer numériquement la puissance moyenne par unité de surface à cet endroit. On pourra utiliser l'aide numérique suivante : $2,5^2 \simeq 6,3$.
- 14.** On considère que le satellite émet de manière isotrope. Calculer la puissance P émise par le satellite dans cette hypothèse (rappel : $2,6^2 \simeq 6,7$). Peut-on supposer que la puissance réelle est supérieure ou inférieure à celle-ci ?

C. Décalage dû à la traversée de la troposphère

Les informations reçues par le récepteur provenant d'un des différents satellites permettent de déterminer la distance d entre le satellite émetteur et le récepteur via le temps de propagation Δt . Si on considère que la vitesse de propagation du signal est c (vitesse de la lumière dans le vide), alors $d = c\Delta t$. Cependant, le signal traverse l'atmosphère et la vitesse de propagation dans l'air n'est pas exactement égale à $c = 3 \cdot 10^8 \text{ m.s}^{-1}$, il y a donc une correction à appliquer. On ne considère dans la suite que les 50 premiers kilomètres d'atmosphère (en partant du sol) que l'on appelle *troposphère*. On notera L cette longueur. Pour simplifier, on se place dans le cas où le satellite émetteur est à la verticale du récepteur.

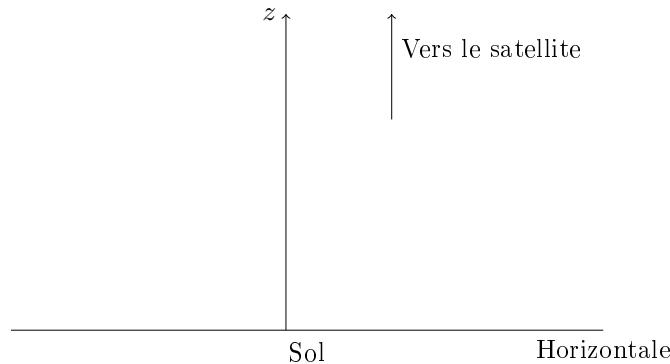


FIGURE 6 – Repérage de l'altitude

15. En considérant l'air comme un gaz parfait, donner l'expression de sa masse volumique ρ en fonction de M_{air} (masse molaire de l'air), P (pression), R (constante des gaz parfaits) et T (température). On donne $M_{\text{air}} = 29 \text{ g.mol}^{-1}$ et $R = 8,3 \text{ SI}$. Calculer numériquement la masse volumique de l'air au niveau du sol où on prendra $P = P_0 = 1 \text{ bar}$ et $T = T_0 = 270 \text{ K}$.
16. La loi de Gladstone donne une relation entre l'indice de réfraction n d'un gaz et sa masse volumique ρ , K étant une constante : $n = 1 + K\rho$
Montrer que, pour un gaz parfait, on peut écrire $n = 1 + K'\frac{P}{T}$. On prendra dans la suite $K' = 7,8 \cdot 10^{-7} \text{ SI}$. Quelle est l'unité de K' ?
17. Montrer que l'expression de la pression P en fonction de l'altitude z dans le modèle de l'atmosphère isotherme s'écrit $P = P_0 \exp\left(-\frac{z}{H}\right)$ et préciser l'expression de H .
18. On prendra $P(z=0) = P_0 = 1 \text{ bar}$, $T = 270 \text{ K}$ et $g = 9,8 \text{ m.s}^{-2}$. Calculer numériquement H ainsi que la pression à l'altitude de H . Aide numérique : $e^{-1} \simeq 0,37$
19. On s'intéresse à la variation de l'indice de réfraction n de l'air en fonction de l'altitude z .
Montrer que l'on peut écrire, à partir des résultats précédents :

$$n = 1 + \alpha P_0 \exp\left(-\frac{z}{H}\right)$$

Donner l'expression de α et sa valeur numérique.

20. Montrer que le temps dt pour traverser verticalement une petite épaisseur d'atmosphère dz s'écrit :
- $$dt = \frac{1}{c} \left(1 + \alpha P_0 \exp\left(-\frac{z}{H}\right)\right) dz$$
21. En déduire l'expression littérale du temps Δt nécessaire pour traverser verticalement les $L = 50 \text{ km}$ de la troposphère en fonction de c , L , α , P_0 et H .
22. En déduire le temps de propagation supplémentaire t_{sup} lors de la traversée de la troposphère par rapport à la même distance parcourue dans le vide. Faire l'application numérique. A quelle erreur sur l'estimation de la distance entre le satellite émetteur et le récepteur cela conduit-il ? Aide numérique : $e^{-6,25} \simeq 0,002$

Deuxième partie
TRAITEMENT DU SIGNAL
PAR LE RÉCEPTEUR

Dans cette partie, les fonctions et programmes attendus devront être écrits en langage Python. On prendra garde à la lisibilité du code, et notamment aux indentations. Le code devra être documenté. Toute fonction définie dans le sujet pourra être utilisée dans la suite de celui-ci, même sans avoir été codée.

D. Identification du satellite et estimation de la distance

Le signal capté par l'antenne du récepteur, une fois amplifié, filtré et démodulé, donne accès à un *message binaire*, c'est-à-dire une succession de 0 et de 1 logiques. Le message présenté dans cette partie est une version simplifiée des fonctionnalités de base du GPS américain. Il est constitué de deux séquences :

1. le *message de navigation*, transmis à 50 bit/s, qui donne notamment la position des satellites ;
2. le *code C/A*, pour *Coarse/Acquisition code* (« code grossier et d'acquisition » en français), transmis à $1,023 \text{ Mbit.s}^{-1}$, qui permet d'identifier le satellite émetteur ainsi que la durée de propagation depuis celui-ci jusqu'au récepteur.

Ces deux séquences sont combinées à l'aide d'une opération « OU exclusif » (figure 7).

Message de navigation 50 bit.s ⁻¹	0							1						
Code C/A 1 023 000 bit.s ⁻¹	1	0	0	1	1	0	0	1
Message transmis 1 023 000 bit.s ⁻¹	1	0	0	1	0	1	1	0

FIGURE 7 – Constitution du message émis par les satellites

Le message de navigation étant transmis beaucoup plus lentement que le code C/A, son effet se limite à inverser de temps à autre le code C/A, comme le montre la figure 7. L'objectif de cette partie est de montrer, à l'aide d'un programme informatique simple, comment le code C/A est utilisé pour identifier le satellite émetteur et estimer la durée de propagation.

D.1. Les codes C/A : définitions

Les codes C/A sont des codes binaires qu'il est commode d'écrire selon la convention NRZ (Non-Retour à Zéro), c'est-à-dire en notant les deux niveaux logiques par les nombres -1 et +1. Dans ce qui suit, les codes sont représentés par des suites $a = (a_i)_{i \in \mathbb{N}}$ d'entiers valant 1 ou -1, dont les indices vont de 0 à $n - 1$ tous deux inclus, où n est la longueur du code.

Les codes C/A sont périodiques de longueur $n = 1023$ et sont émis à une vitesse de $1,023 \text{ Mbit.s}^{-1}$: ils se répètent donc exactement toutes les millisecondes. Chaque satellite émet un code C/A différent. Leur manipulation par le récepteur GPS repose sur les deux définitions suivantes.

Définition 1 : la *corrélation* de deux codes $a = (a_i)_{i \in \mathbb{N}}$ et $b = (b_i)_{i \in \mathbb{N}}$ de même longueur n est le nombre réel suivant :

$$\langle a, b \rangle = \frac{1}{n} \sum_{i=0}^{n-1} a_i b_i$$

Définition 2 : le code $a = (a_i)_{i \in \mathbb{N}}$ retardé d'un entier k est le code dont le i -ème terme est a_{i-k} , étant entendu que les codes sont n -périodiques : les indices sont donc définis modulo n et peuvent être des entiers relatifs quelconques. Par exemple, le code $(-1, 1, 1, 1)$ retardé de 1 est $(1, -1, 1, 1)$.

Nous nous proposons ici d'implanter ces opérations en Python ; les fonctions obtenues seront utilisées dans toute la suite du sujet. Les codes C/A sont représentés par des listes d'entiers valant 1 ou -1, de longueurs *a priori* quelconques (pas forcément 1023 ; nous verrons pourquoi par la suite). Si besoin, on utilisera l'opérateur modulo (%) pour gérer la périodicité des indices.

- 23.** Écrire une fonction `corr` qui prend pour arguments deux listes `a` et `b` de *même longueur* (on ne demande pas de le vérifier) représentant des codes C/A, et qui renvoie un flottant contenant leur corrélation. Pour cette question, les fonctions intégrées de type `sum` sont interdites.
- 24.** Écrire une fonction `retarde` qui prend pour arguments une liste `a` représentant un code C/A et un entier `k`, et qui renvoie la liste représentant le code retardé de `k`.

D.2. Trois propriétés des codes C/A

L'exploitation des codes C/A pour identifier les satellites et les durées de propagation repose sur les trois propriétés suivantes, qui font appel aux définitions précédentes :

1. l'*autocorrélation* d'un code, c'est-à-dire sa corrélation avec lui-même, vaut 1 ;
2. la corrélation d'un code avec le même code *retardé* d'un nombre k non nul est quasi-nulle ;
3. la corrélation entre deux codes émis par *deux satellites différents* est quasi-nulle, et cela vaut aussi si l'un des deux codes (ou les deux) est retardé.

Les trois questions suivantes visent à expliquer ces propriétés et ne demandent pas de longs développements mathématiques.

- 25.** Montrer la première des trois propriétés ci-dessus (autocorrélation égale à 1).
- 26.** Montrer plus généralement que la corrélation de deux codes est toujours comprise entre -1 et 1. À quelle condition vaut-elle -1 ? On pourra donner un exemple.
- 27.** Combien y a-t-il de codes C/A possibles ? Donner un ordre de grandeur de ce nombre sous la forme 10^k en considérant que $2^{10} \approx 10^3$.

En pratique, on n'utilise qu'une trentaine de codes, qui sont choisis de sorte à vérifier les propriétés 2 et 3 ci-dessus. Ces codes sont générés automatiquement à l'aide d'un algorithme simple mais calculatoire, qui n'est pas détaillé ici, à partir de deux entiers naturels notés p et q ; la documentation du GPS donne les valeurs de p et q correspondant à chaque satellite (par exemple, pour le satellite numéroté 1, p vaut 1 et q vaut 5).

Dans la suite du sujet, on suppose que l'on dispose d'une fonction Python `codeCA` qui prend pour arguments deux entiers p et q et renvoie une liste de longueur 1023, dont chaque élément vaut -1 ou 1, contenant le code C/A correspondant. On se propose d'utiliser cette fonction pour illustrer les trois propriétés ci-dessus.

- 28.** En utilisant les fonctions `codeCA`, `retarde` et `corr` (les deux dernières étant définies dans la partie précédente), écrire un bloc d'instructions qui :

- construit une liste `code1` contenant le code C/A correspondant à $p=1$ et $q=5$;
- construit une liste `code2` contenant le code C/A correspondant à $p=2$ et $q=6$;
- construit une liste de listes nommée `codes1` telle que pour tout i allant de 0 à 1022, `codes1[i]` corresponde à `code1` retardé de i ;
- construit une liste nommée `correls11` telle que `correls11[i]` soit la corrélation de `codes1[i]` et de `code1`;
- construit une liste nommée `correls12` telle que `correls12[i]` soit la corrélation de `codes1[i]` et de `code2`.

On pourra, sans obligation, utiliser des constructions de listes par compréhension.

Le tracé des listes `correls11` et `correls12` en fonction de l'indice est représenté figure 8.

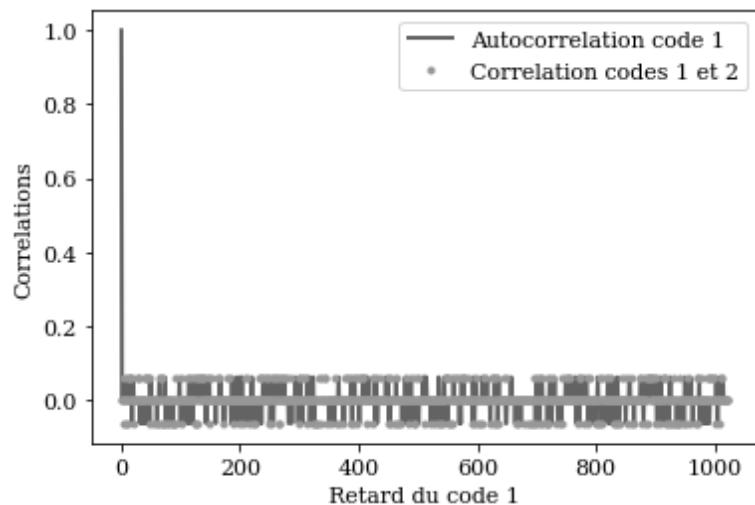


FIGURE 8 – Corrélation du code 1 retardé avec les codes 1 et 2

Ce graphe illustre bien les trois propriétés ci-dessus : seule l'*autocorrélation non retardée* de `code1` vaut 1. Si on corrèle `code1` avec une version retardée de lui-même ou avec `code2` (avec ou sans retard), la valeur obtenue est très faible.

D.3. L'acquisition

Les propriétés précédentes permettent, à partir du message binaire capté par le récepteur GPS, d'identifier le satellite qui l'a émis et d'estimer (grossièrement) la durée de la propagation. Cette opération s'appelle *acquisition*.

Pour la réaliser, le récepteur GPS est muni d'un générateur de codes C/A identique à celui embarqué dans les satellites. Il génère les codes correspondant à tous les satellites possibles (il y en a au plus 31) avec tous les retards possibles (il y en a 1023) et, pour chacun d'eux, calcule sa corrélation avec le message reçu. Les résultats précédents montrent que si les codes ne correspondent pas ou présentent un décalage, alors leur corrélation sera proche de zéro. C'est uniquement si les codes sont identiques *et* alignés que l'on obtiendra une valeur proche de 1 (ou de -1 si le code a été inversé par le message de navigation, comme le montre la figure 7). Le « bon » satellite et le « bon » retard sont donc ceux pour lesquels la corrélation est *maximale en valeur absolue*.

Pour la question suivante, on suppose que les résultats des corrélations sont stockés dans une liste de listes T , de sorte que $T[i][j]$ est la corrélation du signal reçu avec le code C/A du i -ème satellite retardé de j bits. On précise que si x est un flottant ou un entier, $\text{abs}(x)$ renvoie sa valeur absolue.

- 29.** Écrire une fonction `indmax` prenant pour argument une liste de listes T et renvoyant le couple d'indices (i, j) tel que $\text{abs}(T[i][j])$ soit maximal. On admettra que toutes les listes $T[i]$ ont la même longueur et on contrôlera, à l'aide de l'instruction `assert`, que $T[0]$ n'est pas vide.

On obtient ainsi directement le numéro du satellite ayant émis le message capté. De plus, le début de chaque code C/A est émis *exactement* un nombre entier de millisecondes après une heure « ronde » (les satellites possèdent à cet effet des horloges atomiques très précises). Si l'horloge du récepteur a été bien recalée lors des mesures précédentes, le nombre de valeurs dont il a fallu retarder le code donne donc une information sur la durée de la transmission.

Ce procédé rencontre néanmoins une limite physique importante. On rappelle que le code C/A est émis à $1,023 \text{ Mbit.s}^{-1}$ et que la vitesse de la lumière dans le vide est d'environ 3.10^8 m.s^{-1} .

- 30.** En supposant que le signal émis par les satellites se propage à la vitesse de la lumière dans le vide, donner une valeur approchée de la distance parcourue pendant la transmission d'un bit (qui est aussi la résolution de l'estimation de la distance par cette technique).

La résolution des récepteurs GPS est nettement inférieure à cette valeur. D'autres techniques sont donc nécessairement employées. L'une d'elles est de *suréchantillonner* les codes C/A, c'est-à-dire d'utiliser localement une fréquence d'échantillonnage multiple de $1,023 \text{ Mbit.s}^{-1}$. Chaque chiffre (-1 ou 1) de la séquence est donc écrit plusieurs fois au lieu d'une seule.

- 31.** Écrire une fonction `surech` prenant pour arguments une liste a et un entier k et renvoyant la liste a suréchantillonnée d'un facteur k (par exemple, l'appel `surech([1,0],3)` doit renvoyer `[1,1,1,0,0,0]`).

En suréchantillonnant les codes, il est possible de tester des retards égaux à une fraction de la durée de transmission d'un bit, et donc d'atteindre une fraction de la résolution calculée préce-

demment. Malheureusement, le procédé possède lui aussi ses limites, cette fois en ce qui concerne le temps de calcul. Si l'on appelle n le nombre d'échantillons utilisés pour représenter le code C/A, on rappelle que la corrélation de deux codes $a = (a_i)_{i \in \mathbb{N}}$ et $b = (b_i)_{i \in \mathbb{N}}$ est :

$$\langle a, b \rangle = \frac{1}{n} \sum_{i=0}^{n-1} a_i b_i$$

et que l'acquisition teste tous les retards possibles, ce qui implique de calculer n corrélations.

- 32.** Donner et justifier la complexité asymptotique du calcul de ces n corrélations en fonction de n . D'après ce résultat, si l'on souhaite diviser la résolution par 100 et donc suréchantillonner d'un facteur 100, par combien cela multiplie-t-il le temps de calcul ?

La procédure d'acquisition ne peut donc donner qu'une estimation grossière de la distance entre le récepteur et le satellite. Pour affiner cette estimation, il faut utiliser une autre méthode.

E. Affinage de la mesure : la boucle à verrouillage de retard

Une fois la procédure d'acquisition terminée, le récepteur GPS dispose du code C/A correspondant au « bon » satellite et d'une mesure grossière du temps de propagation. Il doit alors affiner cette mesure, dans le double objectif :

- de « suivre » le satellite émetteur, c'est-à-dire de décoder son signal sans interruption malgré les variations de la distance satellite-récepteur et donc du retard de propagation ;
- de mesurer cette distance récepteur-satellite avec une précision permettant la localisation.

L'algorithme mis en œuvre pour cela s'appelle *boucle à verrouillage de retard*, ou DLL pour *Delay-Locked Loop*. Il consiste, à partir du code C/A local sélectionné lors de l'acquisition, à « caler » ce code sur le signal reçu en effectuant, périodiquement et en boucle, les trois opérations suivantes (figure 9) :

1. *estimer le décalage* du signal reçu vis-à-vis du code C/A local ;
2. *intégrer* par rapport au temps, ou plus simplement sommer, les décalages estimés ;
3. *retarder* le code C/A local d'un délai proportionnel à la sortie de l'intégrateur ; le code retardé est renvoyé à l'estimateur de décalage, formant un système bouclé.

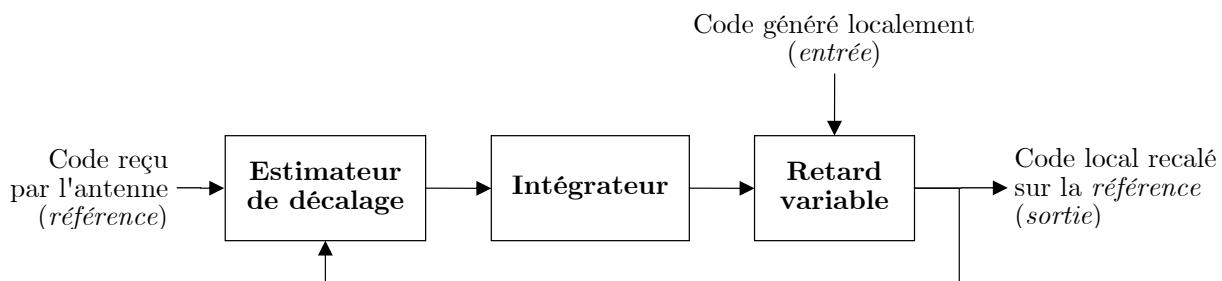


FIGURE 9 – Fonctionnement de la boucle à verrouillage de retard

E.1. Modélisation S.L.C.I. rudimentaire

Ce fonctionnement revient à *asservir le retard du code C/A local sur le retard de propagation*. Un modèle rudimentaire de cet asservissement est proposé figure 10 ; les variables de ce modèle ne sont pas les codes ou signaux eux-mêmes, mais leurs *retards* respectifs par rapport au code émis par le satellite. Plus précisément :

- $T_{\text{ref}}(p)$ représente le retard de propagation inconnu que l'on souhaite reproduire ;
- $T_e(p)$ représente le retard du code « d'entrée » obtenu lors de l'acquisition ;
- $T_s(p)$ représente le retard du code « de sortie » recalé par l'algorithme.

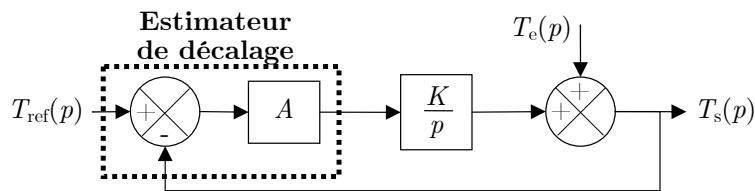


FIGURE 10 – Modèle rudimentaire de la boucle à verrouillage de retard

On souhaite vérifier qu'en régime permanent, le retard à la sortie $t_s(t)$ tend bien vers le temps de propagation du signal reçu (que l'on peut ainsi mesurer indirectement). Pour cela, on se propose d'exprimer $T_s(p)$ sous la forme $T_s(p) = H_1(p)T_{\text{ref}}(p) + H_2(p)T_e(p)$.

33. Exprimer les fonctions de transfert $H_1(p)$ et $H_2(p)$ sous forme canonique en fonction de A et de K . Justifier, d'après les formes obtenues, que ce modèle est stable.

On suppose ensuite que $T_{\text{ref}}(p)$ et $T_e(p)$ sont deux échelons d'amplitudes respectives $T_{\text{ref}0}$ et T_{e0} , et on cherche la valeur finale de $t_s(t)$, dont la transformée de Laplace est $T_s(p)$. On rappelle que la transformée de Laplace d'un échelon d'amplitude B s'écrit B/p .

34. À l'aide du théorème de la valeur finale, montrer que lorsque le temps t tend vers l'infini, $t_s(t)$ tend vers $T_{\text{ref}0}$ quelle que soit la valeur de T_{e0} . Conclure sur le fonctionnement de l'algorithme en régime établi.

35. Le résultat précédent aurait-il été encore vrai si l'algorithme n'avait pas comporté d'intégration, c'est-à-dire si le bloc K/p de la figure 5 avait été remplacé par un simple gain pur K ? On justifiera la réponse sans calcul.

Le modèle précédent ne permet pas de décrire le comportement de la boucle à verrouillage de retard en régime transitoire. En effet, il ne prend pas en compte la durée des différentes opérations (mis à part l'intégration, elles sont toutes modélisées par des gains purs) et il est formulé en temps continu alors que l'algorithme est par nature en temps discret. Pour observer le comportement transitoire de la boucle, il faut réaliser une simulation numérique.

E.2. Simulation du fonctionnement

Remarque : cette partie peut être traitée sans avoir préalablement traité la partie D. Néanmoins, elle fait appel à des notions définies dans cette partie (notamment « code C/A », « corrélation », « code retardé » et « suréchantillonnage ») et il est donc conseillé d'avoir lu attentivement l'énoncé.

On se propose dans cette partie d'implanter l'algorithme présenté précédemment. On choisit pour cela de travailler sur des codes C/A suréchantillonnés d'un facteur 100, de sorte à obtenir une résolution temporelle égale à un centième de la durée de transmission d'un bit. On appelle :

- r le code « de référence » reçu par le récepteur ;
- s le code « de sortie » généré localement et retardé par l'algorithme.

L'estimateur de décalage fait appel à deux copies du code de sortie s , l'une avancée de 50 échantillons (soit la moitié de la durée de transmission d'un bit) et notée s^+ , l'autre retardée de 50 échantillons et notée s^- . Son expression est $\epsilon = \langle s^+, r \rangle - \langle s^-, r \rangle$ où \langle , \rangle désigne la corrélation.

On rappelle les spécifications des fonctions définies dans la partie D :

- `corr(a,b)` : prend deux listes a et b et renvoie un flottant égal à leur corrélation ;
- `retarde(a,k)` : prend une liste a et un entier relatif k , et renvoie une liste correspondant à a retardé de k . Si k est négatif, cela revient à avancer a ;
- `codeCA(p,q)` : renvoie la liste de longueur 1023 contenant le code C/A (sans suréchantillonnage) généré à partir des entiers p et q ;
- `surech(a,k)` : prend une liste a et un entier positif k , et renvoie la liste « suréchantillonnée », c'est-à-dire constituée des éléments de a présents k fois consécutives.

- 36.** Proposer une fonction `decal` qui prend pour arguments deux listes s et r de même longueur (on ne demande pas de le vérifier) contenant respectivement le code de sortie et le code de référence, et renvoyant un flottant égal à l'estimateur de décalage défini ci-dessus. On appellera les fonctions `corr` et `retarde`.

La fonction `decal` étant définie, on exécute la suite d'instructions suivante, dont le résultat est donné figure 11 :

```
import matplotlib.pyplot as pp
c = codeCA(1,5) # correspond au satellite 1
cc = surech(c,100)
cr = list(retarde(cc,k) for k in range(-100,101))
d = list(-0.01*k for k in range(-100,101)) # signe moins pour avoir l'avance
e = list(decal(cr[i],cc) for i in range(len(cr)))
pp.plot(d,e)
pp.xlabel("Avance sortie sur ref (rapportée à la durée d'un bit)")
pp.ylabel("Sortie de l'estimateur de décalage")
```

- 37.** Pour chacune des variables cr , d et e , indiquer s'il s'agit d'une liste « simple » ou d'une liste de listes et préciser à quoi correspond son i -ème élément.

Sur le schéma-bloc de la figure 10, on a modélisé l'estimateur de décalage par un comparateur suivi d'un gain pur A . Puisque les variables sont les retards respectifs des différents codes par rapport à une référence commune, ce modèle repose sur l'hypothèse que la sortie de l'estimateur est proportionnelle au décalage entre les deux codes.

- 38.** Indiquer, à l'aide de la figure 11, dans quelle plage de décalages cette hypothèse est cohérente vis-à-vis du comportement de l'estimateur.

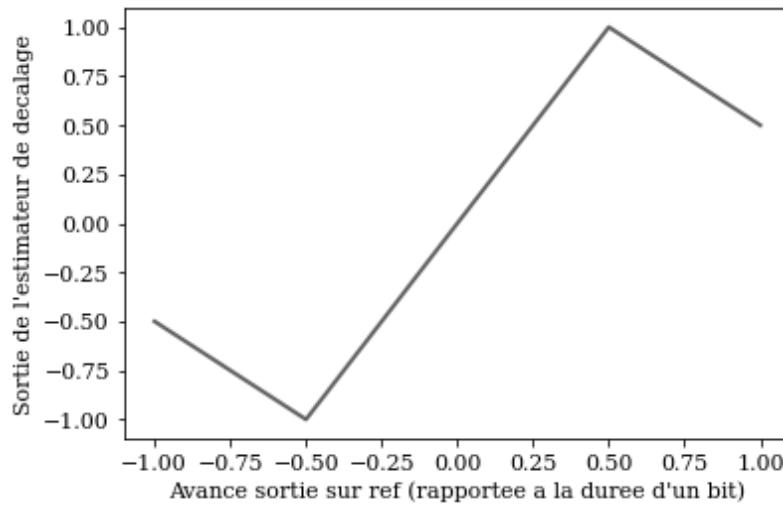


FIGURE 11 – Relation entrée/sortie de l'estimateur de décalage

On se limite à cette plage par la suite. On donne ci-dessous le code incomplet de la fonction `simu` qui simule la boucle à verrouillage de retard présentée sur la figure 9. Cette fonction prend quatre arguments en entrée :

- `ref` : liste contenant le code C/A de référence (choisi constant tout au long de la simulation) ;
- `entree` : liste contenant le code C/A d'entrée ;
- `imax` : durée de la simulation exprimée en nombre de périodes du code C/A (entier) ;
- `K` : « gain » de l'`« intégrateur »` (flottant).

```
def simu(ref, entree, imax, K):
    retard = 0.0
    sortie = entree[:]
    sorties, retards = [sortie], [retard]
    # boucle répétée à chaque période du code C/A
    for i in range(imax-1):
        # 1. on estime le décalage
        ecart = # LIGNE À COMPLÉTER NUMÉRO 1
        # 2. on "intègre" (somme) les écarts avec un facteur K
        retard = # LIGNE À COMPLÉTER NUMÉRO 2
        # 3. on retarde l'entrée pour obtenir la sortie
        sortie = # LIGNE À COMPLÉTER NUMÉRO 3
        # écriture des résultats de la simulation
        sorties.append(sortie)
        retards.append(retard)
    return sorties, retards
```

On précise que les trois opérations (estimation, sommation et retard du code d'entrée) sont effectuées une fois par période du code C/A, soit toutes les millisecondes. Chaque tour de la

boucle `for` simule donc le fonctionnement de l'algorithme sur une période. On précise également que cette simulation ne tient pas compte des contraintes liées au fonctionnement en temps réel sur le récepteur GPS, mais vise uniquement à donner une première approximation du comportement de l'algorithme en régime transitoire.

Pour simplifier le code, l'« intégrateur » est réalisé par une somme discrète : la variable `soutie` est donc la somme cumulée des décalages estimés (variable `ecart`) multipliés par le « gain » K .

- 39.** En appelant les fonctions nécessaires, compléter les trois lignes indiquées de la fonction `simu`. Ne pas oublier que la sortie de l'estimateur de décalage est un flottant, tandis que le nombre d'échantillons dont on retarde un code est nécessairement un entier.

On a réalisé une simulation sur 20 périodes (soit 20 millisecondes) à partir du code C/A du satellite 1 suréchantillonné d'un facteur 100. L'entrée (générée localement) est le code lui-même et la référence (reçue via l'antenne) est le code retardé de 20 échantillons, soit 20 centièmes de bit. Trois valeurs du gain K ont été testées. La figure 12 donne :

- à gauche, l'évolution du retard en sortie de l'intégrateur (qui doit tendre vers 20 pour aligner la sortie sur la référence),
- à droite, la corrélation de la sortie avec la référence (qui tend vers 1 si la sortie s'aligne parfaitement avec la référence).

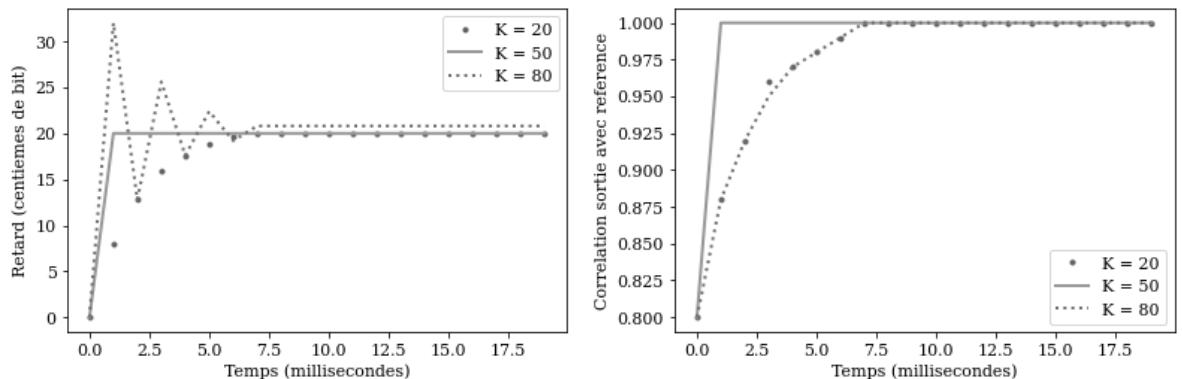


FIGURE 12 – Simulation de la boucle à verrouillage de retard pour trois valeurs de K

Les deux principales différences entre cette simulation et le modèle S.L.C.I. précédent sont la quantification du temps (qui est un nombre entier de périodes) et la quantification du retard appliqué à l'entrée (qui est un nombre entier de centièmes de bit).

- 40.** Indiquer, en justifiant, si les oscillations (obtenues pour $K = 80$) et la convergence en un temps fini (obtenue sur les trois courbes) auraient pu être prédites par le modèle S.L.C.I.

Cette simulation montre que la sortie s'aligne bien sur la référence : il est donc théoriquement possible d'utiliser les codes C/A pour mesurer des distances satellite-récepteur avec une résolution d'un centième de bit. En pratique, cette valeur théorique est rarement atteinte car il est difficile de compenser les autres sources d'erreur, comme les décalages modélisés dans la partie C.

**Troisième partie
EXPLOITATION DE LA
LOCALISATION**

Les opérations étudiées dans la partie précédente permettent au récepteur de mesurer la distance qui le séparent des satellites, tout en récupérant la position de ceux-ci contenue dans le message de navigation. Il peut alors se localiser à l'aide d'un calcul géométrique non abordé ici. Cette partie présente une exploitation de la localisation dans le cadre d'une recherche d'itinéraires.

F. Transmission de la localisation : les trames NMEA

Pour être exploitée par des applications tierces ou par d'autres matériels, la localisation déterminée par le récepteur GPS doit être transmise. Un format normalisé souvent utilisé à cet effet est le protocole NMEA 0183, qui consiste à structurer les informations sous la forme de *trames*.

Une trame NMEA est une chaîne de caractères contenant plusieurs données (les *champs* de la trame) séparées par des virgules. Il en existe différents types. Nous nous limitons dans ce sujet aux trames dites GGA. Un exemple de trame GGA est :

\$GPGGA,064036.289,4836.5375,N,00740.9373,E,1,04,3.2,200.2,M,,,0000*0E

La signification des différents champs est la suivante :

- \$GPGGA : type de trame ;
- 064036.289 : heure d'envoi au format 'HHMMSS.SSS' (ici 06 h 40 min 36,289 s) ;
- 4836.5375 : latitude en valeur absolue (ici 48 degrés 36,5375 minutes) ;
- N : sens de la latitude (N pour Nord, S pour Sud) ;
- 00740.9373 : longitude en valeur absolue (ici 7 degrés 40,9373 minutes) ;
- E : sens de la longitude (E pour Est, W pour Ouest) ;
- 1 : type de positionnement (non abordé ici) ;
- 04 : nombre de satellites utilisés pour la localisation ;
- 3.2 : indicateur de précision (non abordé ici) ;
- 200.2 : altitude (ici 200,2) ;
- M : unité de l'altitude (ici, des mètres) ;
- ,,,0000 : champs non utilisés (d'autres informations peuvent y être inscrites) ;
- * : séparateur entre les champs et la somme de contrôle ;
- 0E : somme de contrôle.

Pour extraire les champs de la trame, on propose d'utiliser la méthode `split` qui découpe une chaîne de caractères autour d'un caractère passé en argument (le « délimiteur »), et renvoie une liste de chaînes. Par exemple, si on définit la chaîne :

```
trame = '$GPGGA,064036.289,4836.5375,N,00740.9373,E,1,04,3.2,200.2,M,,,0000*0E'
```

alors l'appel `trame.split(' ',)` renvoie la liste de chaînes suivante :

```
['$GPGGA', '064036.289', '4836.5375', 'N', '00740.9373', 'E', '1', '04', '3.2', '200.2', 'M', '', '',
'', '0000*0E']
```

On rappelle que si `s` est une chaîne et `i` et `j` deux entiers positifs ou nuls, `s[i:j]` renvoie la portion de `s` dont les indices vont de `i` inclus à `j` exclu.

- 41.** Écrire une fonction `heure` qui prend pour argument la chaîne `trame` au format ci-dessus et renvoie un tuple `(h,m,s)` où `h` et `m` sont des entiers contenant respectivement les heures et les minutes, et `s` est un flottant contenant les secondes (avec les millisecondes).

Par convention, on représente les latitudes par des flottants signés contenant leur valeur en degrés, une latitude nord ('N') étant positive et une latitude sud ('S') étant négative. On rappelle qu'une minute d'angle (notée ') correspond à un soixantième de degré.

- 42.** Écrire une fonction `latitude` qui prend pour argument la chaîne `trame` au format ci-dessus, et renvoie un flottant contenant la latitude signée exprimée en degrés.

Ces informations ne sont néanmoins fiables que si la trame a été transmise sans erreur. La somme de contrôle située à la fin de la trame permet de détecter certaines erreurs de transmission.

La somme de contrôle est un *OU exclusif bit à bit* entre les codes ASCII de tous les caractères situés entre le \$ (qui est toujours au début) et le *, tous deux exclus. Le OU exclusif bit à bit de deux entiers `a` et `b` est un entier, calculé par l'instruction `a^b` en Python, dont la définition formelle n'est pas utile ici. Les deux caractères situés après le * doivent correspondre à l'écriture hexadécimale de cet entier pour que la transmission soit validée.

Exemple : vérifions la somme de contrôle de la trame simplifiée '\$0,1*2D'.

- Les trois caractères situés entre le \$ et le * sont '0', ',' et '1' ;
- Leurs codes ASCII respectifs sont 48, 44 et 49 (en décimal) ;
- L'instruction `48^44^49` renvoie l'entier 45, dont l'écriture hexadécimale est '2D' ;
- Cela correspond bien aux deux caractères situés après le * : la trame est donc valide.

Pour programmer cette vérification, on donne les indications suivantes :

- si `s` est une chaîne, `s.find('a')` recherche la première apparition du caractère 'a' dans `s` et renvoie son indice (ou -1 s'il n'est pas trouvé) ;
- si `c` est un caractère (c'est-à-dire une chaîne de longueur 1), `ord(c)` renvoie le code ASCII de `c`, qui est un entier compris entre 0 et 127 ;
- le OU exclusif bit à bit est commutatif (`a^b == b^a`) et associatif (`a^(b^c) == (a^b)^c`) ;
- le calcul du OU exclusif bit à bit peut être initialisé à zéro car quel que soit l'entier `i`, `0^i` est toujours égal à `i` ;
- si `s` est une chaîne contenant la représentation hexadécimale d'un entier (dont les chiffres vont de 0 à F), l'instruction `int(s,16)` permet d'obtenir l'entier correspondant.

- 43.** Écrire une fonction `control` qui prend pour argument la chaîne `trame` et renvoie `True` si la somme de contrôle correspond bien au résultat du calcul précédent, et `False` sinon.

G. Calcul du plus court chemin

Les récepteurs GPS utilisés pour les trajets automobiles disposent de bases de données géographiques et d'applications mettant en oeuvre des algorithmes de recherche de plus courts chemins. Un des plus classiques est l'algorithme de *Dijkstra*.

Pour pouvoir utiliser celui-ci pour déterminer les plus courts chemins, les différents lieux constituent les sommets d'un graphe et les distances entre deux lieux sont enregistrées comme les poids associés à chaque arête. Pour illustrer, les sommets (les lieux) sont codés par des nombres entiers et on utilisera le graphe pondéré représenté ci-dessous comme exemple.

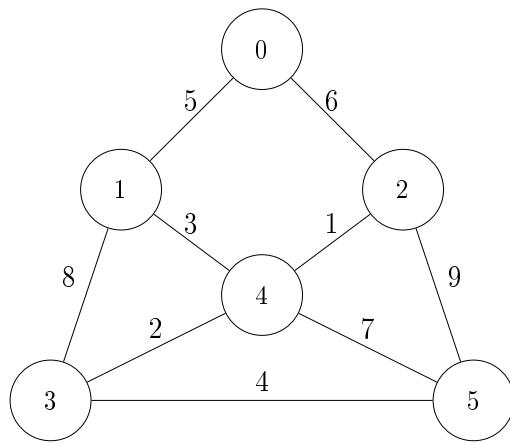


FIGURE 13 – Exemple de graphe pondéré

- 44.** Le graphe (exemple ci-dessus) sera codé par un dictionnaire d'adjacence dont on donne ci-dessous les 3 premières lignes :

```

{0 : [(1,5), (2,6)],
 1 : [(0,5), (3,8), (4,3)],
 2 : [(0,6), (4,1), (5,9)],
 ...
 ...
 ...
}
  
```

Recopier et compléter ce dictionnaire. Pourquoi un dictionnaire est-il, sur l'exemple proposé, préférable à une matrice d'adjacence ? Quelle méthode (la décrire brièvement en 3 à 4 phrases maximum) est à la base des performances des dictionnaires en termes de temps d'accès ?

Description de l'algorithme :

Soit n le nombre de sommets du graphe. On cherche les distances minimales entre un sommet de départ donné et les autres sommets de la manière suivante :

- On utilise une liste de taille n appelée `distances` dont les éléments représentent les distances entre le sommet de départ et chacun des sommets. Cette distance est initialisée à 0 pour le sommet de départ et à -1 (qui représente l'infini) pour les autres ;
- On va visiter les différents sommets du graphe, et il faut savoir lesquels ont déjà été visités ou non. On utilise une liste `deja_visites` de taille n de booléens initialement tous à `False` ;
- On itère avec une boucle `while`, la condition d'arrêt étant le fait que la liste `deja_visites` ne contient que des `True`. Pour chaque itération :
 - On cherche parmi les sommets pas encore visités celui qui est à la plus petite distance du sommet de départ. On l'appelle `en_cours` pour la suite ;
 - On visite ce sommet `en_cours` :
 - On met ce sommet à `True` dans la liste `deja_visites` ;
 - On parcourt ses voisins et pour chacun d'entre eux qui n'a pas encore été visité (appelé `voisin`) on actualise dans la liste `distance` la valeur le concernant : on la remplace par le minimum entre l'ancienne distance et la distance entre le sommet de départ et `en_cours` à laquelle on ajoute la distance entre `en_cours` et `voisin`.

- 45.** Donner les états successifs de la liste `distances` au cours de l'exécution de l'algorithme sur l'exemple de la figure 13, le sommet de départ étant le sommet 0.
- 46.** Écrire une fonction `sommet_min(dist,deja_visites)` qui prend en argument une liste `dist` d'entiers positifs ou égaux à -1 et une liste `deja_visites` de booléens et qui renvoie l'indice du minimum de `dist` parmi ceux dont la valeur dans la liste `deja_visites` correspond à `False` et qui n'ont pas pour valeur -1 . Si aucun indice dans la liste ne convient, cela signifie que l'algorithme est terminé, on renverra `None`.
- 47.** Écrire une fonction `actualiser(k,dist,deja_visites,g)` qui prend en entrée un graphe `g` sous forme d'un dictionnaire d'adjacence, deux listes `dist` et `deja_visites` de taille n et un entier `k` compris entre 0 et $n - 1$ (où n est la taille du graphe), et qui met à jour les listes `dist` et `deja_visites`.
- 48.** En utilisant les fonctions précédentes, écrire une fonction `dijkstra(g,depart)` prenant en entrée un graphe `g` sous forme de dictionnaire d'adjacence et un sommet `depart` et renvoyant la liste des distances minimales au sommet `depart`.

H. Statistiques sur les durées des trajets

L'algorithme vu dans la partie précédente permet de déterminer le trajet le plus court entre deux points donnés. Il peut également déterminer le trajet le plus rapide, c'est-à-dire qui prend le moins de temps. Pour cela, il faut disposer d'informations sur la durée des trajets. En pratique, ces informations peuvent provenir de données obtenues en temps réel sur la circulation, ou encore de statistiques préalablement collectées.

Dans cette partie, on suppose que l'on dispose de données sur des trajets individuels, stockées dans une base de données SQL constituée de deux tables. Ces tables sont :

1. **noeuds**, qui décrit chaque noeud (ou sommet) par les attributs suivants :
 - **id** (entier) : identifiant du noeud (clé primaire) ;
 - **num** (entier) : numéro de la voie de l'adresse ;
 - **voie** (chaîne) : type (rue, avenue...) et nom de la voie de l'adresse ;
 - **code** (entier) : code postal de l'adresse ;
 - **ville** (chaîne) : ville de l'adresse .
 2. **trajets**, qui décrit chaque trajet enregistré par les attributs suivants :
 - **id_from** (entier) : identifiant du noeud de départ ;
 - **id_to** (entier) : identifiant du noeud d'arrivée ;
 - **date** (chaîne) : date de départ du trajet au format "AAAA-MM-JJ" ;
 - **heure** (chaîne) : heure de départ du trajet au format "HH:MM:SS" sur 24 heures ;
 - **durée** (entier) : durée du trajet en secondes .
- 49.** Écrire une requête renvoyant les heures de départ et les durées des trajets ayant débuté le 1^{er} avril 2025 et allant du noeud 123 au noeud 456 (les numéros sont les identifiants).
- 50.** Écrire une requête renvoyant la date de départ, l'heure de départ et la durée des trois trajets les plus courts (c'est-à-dire dont les durées sont les plus petites) allant du noeud 123 au noeud 456, ordonnés par durées croissantes.
- 51.** Écrire une requête renvoyant le nombre de trajets enregistrés qui partent du numéro 7 de la voie nommée « rue des Plantes » située dans la ville de Paris.
- 52.** On considère les trajets *vers* la ville de Lyon ayant débuté le 1^{er} avril 2025. Écrire une requête renvoyant, pour *chaque* ville de départ pour laquelle *au moins dix* trajets de ce type existent, le nom de la ville suivi des durées minimale, moyenne et maximale des trajets.

Fin de l'énoncé

**ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES**

CONCOURS D'ADMISSION 2025

**MARDI 15 AVRIL 2025
14h00 - 18h00**

FILIERES MP-MPI - Epreuve n° 4

INFORMATIQUE A (XULSR)

Durée : 4 heures

**L'utilisation des calculatrices n'est pas autorisée pour
cette épreuve**

Cette composition ne concerne qu'une partie des candidats de la filière MP, les autres candidats effectuant simultanément la composition de Physique et Sciences de l'Ingénieur.

Pour la filière MP, il y a donc deux enveloppes de Sujets pour cette séance.

Arbres de classification d'arbres

Le sujet comporte 12 pages.

Vue d'ensemble du sujet.

Ce sujet porte sur la réalisation d'arbres d'identification des plantes basés sur des observations morphologiques. À partir d'une base de connaissance botaniste, il s'agit de réaliser un arbre dont les feuilles sont des diagnostics (typiquement des noms d'espèce) et les noeuds des *caractères morphologiques* (par exemple la couleur des fleurs). Un noeud a autant de fils que le caractère a de valeurs possibles (jaune, bleue, etc.).

Étant donnée une base de connaissance, il s'agit de construire un arbre dont la hauteur moyenne est la plus petite possible, ce qui revient à minimiser le nombre moyen d'observations nécessaires pour identifier une plante. La difficulté revient donc à choisir à chaque étape le caractère le plus *discriminant*, c'est-à-dire celui qui va le mieux séparer l'espace de recherche.

Pour tenir compte de la diversité au sein d'une espèce, nous adoptons une approche probabiliste basée sur un raisonnement bayésien : la valeur d'un caractère morphologique pour une espèce donnée sera probabiliste. Par exemple, la couleur des fleurs d'une certaine espèce pourra être bleue ou jaune, avec certaines probabilités.

La partie I introduit les éléments de raisonnement bayésien nécessaires à la construction des arbres. La partie II propose une approche par énumération pour générer un arbre qui minimise la hauteur moyenne. La partie III propose une heuristique basée sur l'entropie de Shannon pour éviter la construction de tous les arbres. La partie IV revient sur le calcul exact en proposant une optimisation permettant de réduire l'espace de recherche. Les parties III et IV sont indépendantes l'une de l'autre. Enfin, la partie V propose une approche logique pour étendre la notion de caractère. Cette dernière partie ne dépend que de la première.

Rappels de probabilités

Une **mesure** sur un ensemble fini X est une fonction $\varphi : X \rightarrow \mathbb{R}^+$. Sa **masse** est la quantité $\sum_{x \in X} \varphi(x)$. Une mesure de masse 1 est appelée une **distribution de probabilité**, ou simplement **distribution**. On note $D(X)$ l'ensemble des distributions sur l'ensemble fini X . Pour $x \in X$, la **dirac** en x , notée δ_x , est la distribution qui associe à x le poids 1, et zéro à tous les autres éléments. Étant donné une famille de distributions $d_1, \dots, d_n \in D(X)$ et des poids $p_1, \dots, p_n \in [0; 1]$ tels que $\sum_{1 \leq i \leq n} p_i = 1$, on note $p_1 \cdot d_1 + \dots + p_n \cdot d_n$ la distribution $d \in D(X)$ telle que $d(x) = p_1 \times d_1(x) + \dots + p_n \times d_n(x)$ pour tout $x \in X$.

Si X est un ensemble, on notera $\mathcal{P}(X)$ l'ensemble de ses parties.

Rappels d'OCaml

Dans le reste du sujet, on pourra utiliser les fonctions suivantes de la bibliothèque standard OCaml pour les listes :

- `List.hd [e1; ...; en]` renvoie e_1 , et `List.hd []` lève une exception.
- `List.tl [e1; ...; en]` renvoie $[e_2; \dots; e_n]$, et `List.tl []` lève une exception.
- `List.length l` renvoie la longueur (nombre d'éléments) de la liste l .
- `List.init n f` renvoie la liste $[f\ 0; \dots; f\ (n-1)]$ pour n positif.
- `List.assoc k [(k1, v1); ...; (kn, vn)]` renvoie le premier v_i tel que $k = k_i$ s'il existe, et lève l'exception `Not_found` sinon.
- `List.find f [e1; ...; en]` renvoie le premier e_i tel que $f\ e_i = \text{true}$ s'il existe, et lève l'exception `Not_found` sinon.
- `List.map f [e1; ...; en]` renvoie $[f\ e_1; \dots; f\ e_n]$.
- `List.filter f l` renvoie la liste des éléments e de l tels que $f\ e = \text{true}$, dans l'ordre.
- `List.concat [l1; ...; ln]` renvoie $l_1 @ \dots @ l_n$, la concaténation des listes l_i .
- `List.concat_map f l` renvoie `List.concat (List.map f l)`.
- `List.fold_left f a0 [e1; ...; en]` renvoie a_n où $a_{k+1} = f\ a_k\ e_{k+1}$ pour $0 \leq k < n$.

Pour les tableaux, on pourra utiliser les fonctions suivantes :

- `Array.length a` renvoie la longueur (nombre d'éléments) du tableau a .
- `Array.init n f` renvoie le tableau $[|f\ 0; \dots; f\ (n-1)|]$ pour n positif.
- `Array.of_list [e1; ...; en]` renvoie le tableau $[|e_1; \dots; e_n|]$.

On rappelle enfin que la fonction `log : float -> float` est disponible en OCaml pour calculer un logarithme naturel.

Partie I : Description probabiliste des plantes

On se donne un nombre de caractères $N \in \mathbb{N}^*$. Un **caractère** sera un entier $i \in \{0, 1, \dots, N-1\}$. Pour chaque caractère i , on se donne un ensemble fini V_i de **valeurs** possibles pour ce caractère.

Pour nos exemples, nous prendrons $N = 2$ et deux valeurs pour chaque caractère : $V_0 = \{4, 5\}$ représentera le nombre de pétales des fleurs, et $V_1 = \{\text{bleu}, \text{blanc}\}$ représentera leur couleur.

Nous utiliserons une représentation probabiliste des espèces pour modéliser la variabilité intra-espèce, ainsi que la variabilité des interprétations lors des observations : on peut trouver au sein d'une même espèce des individus à fleurs bleues et d'autres à fleurs blanches ; par ailleurs, une couleur bleue très claire pourra être interprétée comme bleue ou blanche en fonction de l'observateur. Ainsi, chaque espèce sera décrite par des valeurs possibles pour chaque caractère, organisées en distributions de probabilité : une **espèce** est un N -uplet $s = (s_0, \dots, s_{N-1})$ où $s_i \in D(V_i)$. Intuitivement, $s_i(v)$ représente la probabilité que la valeur v soit observée pour le caractère i chez un individu de l'espèce. On suppose donné un ensemble fini d'espèces \mathcal{S} .

Dans nos exemples, \mathcal{S} sera composé des trois espèces suivantes, décrites en utilisant des diracs δ :

$$\begin{aligned} \text{myosotis} &= (\delta_5, 0, 8 \cdot \delta_{\text{bleu}} + 0, 2 \cdot \delta_{\text{blanc}}) \\ \text{lin} &= (\delta_5, \delta_{\text{bleu}}) \\ \text{gaillet} &= (\delta_4, \delta_{\text{blanc}}) \end{aligned}$$

On définit l'ensemble Ω suivant, dont les éléments modélisent des individus, décrits par leur espèce et les valeurs de leurs caractères :

$$\Omega \stackrel{\text{def}}{=} \{(s, v_0, \dots, v_{N-1}) \mid s \in \mathcal{S}, v_i \in V_i \text{ pour tout } i\}$$

On considérera l'espace probabilisable $(\Omega, \mathcal{P}(\Omega))$ obtenu en munissant Ω de la tribu discrète. On utilisera plusieurs lois de probabilité sur cet espace, définies en fonction d'un **état** $\sigma \in D(\mathcal{S})$: intuitivement, l'état indiquera la probabilité d'observer chaque espèce. La **loi de probabilité** P^σ **induite par un état** σ est définie comme l'unique loi satisfaisant l'équation suivante pour tout $(s, v_0, \dots, v_{N-1}) \in \Omega$:

$$P^\sigma(\{(s, v_0, \dots, v_{N-1})\}) = \sigma(s) \times \prod_{0 \leq i < N-1} s_i(v_i)$$

On notera S la variable aléatoire qui à $(s, v_0, \dots, v_{N-1}) \in \Omega$ associe s . Pour chaque caractère i , on notera C_i la variable aléatoire qui à $(s, v_0, \dots, v_{N-1}) \in \Omega$ associe v_i . On pourra ainsi écrire, par exemple, $P^\sigma(C_i = v \mid S = s)$, qui représente (dans l'état σ) la probabilité (conditionnelle) que le caractère i prenne la valeur v chez un individu de l'espèce s .

Préliminaires probabilistes. On établit tout d'abord quelques résultats élémentaires sur lesquels s'appuiera notre méthode de classification.

Question 1. On considère l'état $\sigma_0 = \{\text{gaillet} : 50\%, \text{myosotis} : 30\%, \text{lin} : 20\%\}$ sur les espèces de notre exemple. Quelle est la probabilité $P^{\sigma_0}(C_1 = \text{blanc})$ d'observer des fleurs blanches dans cet état ? Pour chaque espèce s , que vaut la probabilité $P^{\sigma_0}(S = s \mid C_1 = \text{blanc})$ d'observer un individu de l'espèce s sachant que celui-ci a des fleurs blanches ? Aucune justification n'est attendue.

Question 2. Étant donnés un état $\sigma \in D(\mathcal{S})$, une espèce $s \in \mathcal{S}$, un caractère i et une valeur $v \in V_i$, exprimer simplement les probabilités $P^\sigma(S = s)$ et $P^\sigma(C_i = v)$, ainsi que $P^\sigma(C_i = v \mid S = s)$ quand elle est bien définie. Les réponses devront être justifiées.

Question 3. Étant donnés un état $\sigma \in D(\mathcal{S})$, un caractère i , une valeur $v \in V_i$ et une espèce $s \in \mathcal{S}$, exprimer $P^\sigma(S = s | C_i = v)$. Sous quelle condition cette probabilité est-elle bien définie ?

Pour un état σ , un caractère i et une valeur $v \in V_i$, on définit l'**état mis à jour** $\sigma[i := v]$ comme la distribution qui à chaque espèce s associe $P^\sigma(S = s | C_i = v)$.

Question 4. Pour un état σ , deux caractères $i \neq j$ et des valeurs $v \in V_i$ et $v' \in V_j$, montrer que $P^\sigma(S = s | C_i = v, C_j = v') = P^{\sigma[i := v]}(S = s | C_j = v')$ quand ces probabilités sont bien définies.

Le résultat précédent nous indique que, si l'on souhaite évaluer dans un état σ la probabilité d'observer une certaine espèce sachant que $C_i = v$ et $C_j = v'$, il nous suffit de calculer une probabilité conditionnée seulement par $C_j = v'$ mais dans l'état $\sigma[i := v]$. En allant plus loin, si l'on pose $\sigma' = \sigma[i := v]$ et $\sigma'' = \sigma'[j := v']$, on a $P^\sigma(S = s | C_i = v, C_j = v') = \sigma''(s)$. On admettra que ce résultat se généralise à un nombre arbitraire d'observations. Ainsi, il est possible de procéder à la reconnaissance de plantes en calculant des états mis à jours par des observations successives, jusqu'à atteindre un état de la forme δ_s ou ne plus pouvoir faire de nouvelle observation.

Choix des représentations pour le code. Un caractère sera naturellement représenté par un entier OCaml. On représentera aussi les valeurs par des entiers, en supposant que chaque V_i est de la forme $\{0, 1, \dots, k_i - 1\}$, ce qui revient à numérotter les valeurs possibles. On pose ainsi :

```
type caractere = int
type valeur = int
```

On suppose donnés le nombre de valeurs pour chaque caractère sous la forme d'un tableau :

```
val num : int array    (* num.(i) = k_i, i.e., V_i = {0, 1, ..., num.(i)-1} *)
```

Enfin, on considérera que la liste des caractères $[0; 1; \dots; N-1]$ est définie en variable globale à partir du tableau `num` :

```
let caracteres : caractere list = List.init (Array.length num) (fun i -> i)
```

Pour notre exemple avec deux caractères, on aura ainsi :

```
(* Nombre de valeurs possibles pour le nombre de pétales et pour leur couleur *)
let num = [|2; 2|]
```

```
(* Nombre de pétales *)
let nb_petales : caractere = 0
let quatre, cinq = 0, 1

(* Couleur *)
let couleur_petales : caractere = 1
let bleu, blanc = 0, 1
```

Une mesure μ sur X sera représentée par une liste de couples $(x, \mu(x))$ avec $x \in X$ et $\mu(x) > 0$, sans doublons. **En particulier, les éléments de poids nuls ne sont pas inclus dans la liste, et l'ordre n'a pas d'importance.**

```
type 'x mesure = ('x * float) list
let dirac v = [(v, 1.)]
```

Enfin, on codera comme suit les espèces et états :

```
type espece = { name: string; mesures: valeur mesure array }
type etat = espece mesure
```

Pour notre exemple, on aura ainsi :

```
let myosotis : espece =
  { name = "myosotis";
    mesures = [| dirac cinq; [(bleu, 0.8);(blanc, 0.2)] |] }
let lin : espece =
  { name = "lin";
    mesures = [| dirac cinq; dirac bleu |] }
let gaillet : espece =
  { name = "gaillet";
    mesures = [| dirac quatre; dirac blanc |] }
let sigma0 : etat = [(gaillet, 0.5);(myosotis, 0.3);(lin, 0.2)]
```

Question 5. Écrire les fonctions suivantes :

```
val proba_de      : 'x -> 'x mesure -> float
val somme_ponderee : 'x mesure -> ('x -> float) -> float
val repondere     : 'x mesure -> ('x -> float) -> 'x mesure
```

La première renvoie le poids d'un élément selon une mesure donnée. La seconde renvoie la somme d'une mesure μ pondérée par une fonction $f : X \rightarrow \mathbb{R}$, définie par $\sum_{x \in X} \mu(x)f(x)$. La troisième repondère une mesure sur X par une fonction $f : X \rightarrow \mathbb{R}$: le poids de $x \in X$ passe de $\mu(x)$ à $\mu(x)f(x)$.

Question 6. Écrire deux fonctions

```
val proba_espece : espece -> caractere -> valeur -> float
val proba_etat   : etat -> caractere -> valeur -> float
```

où `proba_espece s i v` renvoie la probabilité $P^{\delta_s}(C_i = v)$ et `proba_etat σ i v` renvoie $P^\sigma(C_i = v)$.

Question 7. Écrire une fonction

```
val reponse : etat -> caractere -> valeur -> etat
```

qui, étant donnés un état σ , un caractère i et une valeur $v \in V_i$, renvoie l'état $\sigma[i := v]$ si celui-ci est bien défini, et lève une exception sinon.

Partie II : Arbre optimal

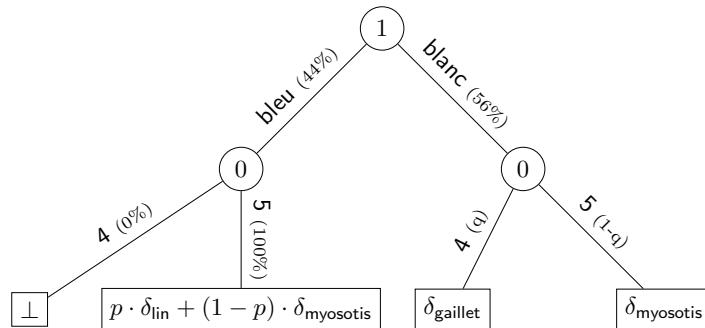
Dans cette seconde partie, on définit la notion d'arbre de décision, et on s'intéresse au calcul d'un arbre de décision optimal en un certain sens. On introduit les types de données suivants pour coder nos arbres :

```
type noeud = {caractere: caractere; enfants: (float * arbre) array}
and arbre =
| Noeud of noeud      (* On pose une question sur un caractère. *)
| Feuille of etat      (* On a atteint une feuille, on donne l'état. *)
| Impossible            (* On a atteint une feuille impossible :
                           aucune espèce ne correspond aux réponses données. *)
```

Les valeurs de type `arbre` ne représentent pas toutes des arbres de décision. Une valeur t de type `arbre` est **un arbre de décision** pour un ensemble de caractères C et un état σ lorsque :

- L'arbre t n'est pas `Impossible`.
- Si t est de la forme `Feuille` σ' , alors $\sigma' = \sigma$ et (au moins) l'une des deux conditions suivantes est satisfaite :
 - σ est une distribution dirac (il n'y a plus qu'une seule espèce possible) ;
 - $C = \emptyset$ (il n'y a plus de caractère disponible pour discriminer).
- Si t est de la forme `Noeud` `{caractere=i;enfants=e}` alors i est un élément de C , e est un tableau à `num.(i)` éléments, et pour chaque valeur possible $0 \leq v < \text{num.}(i)$, $e.(v)$ est de la forme $(P^\sigma(C_i = v), t')$ avec :
 - si $P^\sigma(C_i = v) = 0$, alors t' est `Impossible` ;
 - sinon, t' est un arbre de décision pour l'ensemble de caractères $C \setminus \{i\}$ et l'état $\sigma[i := v]$.

Dans le contexte de l'exemple de la partie I, on représente graphiquement ci-dessous un arbre de décision pour l'ensemble complet de caractères $\{0, 1\}$ et l'état σ_0 de la question 1 (pour des valeurs de p et q non précisées) :



Les noeuds sont étiquetés par des caractères et les feuilles par des états ; la notation \perp est utilisée pour représenter les cas impossibles où aucune espèce ne peut correspondre aux valeurs observées. Chaque arc est étiqueté par la valeur du caractère correspondant à l'enfant ainsi que la probabilité que cette valeur soit observée.

On définit la **hauteur moyenne** d'un arbre t , notée $h(t)$, comme suit :

- Si t est de la forme `Feuille` σ ou `Impossible`, alors $h(t) = 0$.

- Si t est de la forme `Noeud {caractere=i;enfants=e}`, et si l'on note $(p_v, t_v) = e.(v)$, alors :

$$h(t) = 1 + \sum_{0 \leq v < \text{num.}(i)} p_v \times h(t_v)$$

La hauteur moyenne correspond au nombre moyen de questions avant d'atteindre une feuille. Sur l'exemple précédent, la hauteur moyenne est de deux.

Question 8. Sur l'exemple de la question 1, énumérer tous les arbres de décision possibles pour les caractères $\{0, 1\}$ et l'état σ_0 , et donner leur hauteur moyenne — pour l'arbre déjà représenté ci-dessus, on pourra se contenter de préciser les valeurs de p et q . Quel est l'arbre de décision avec la hauteur moyenne minimale ?

Dans la suite de cette partie, on se propose de calculer naïvement un arbre de décision de hauteur moyenne minimale. On dira qu'un arbre t est **optimal pour C et σ** quand c'est un arbre de décision pour C et σ tel que $h(t)$ est minimale parmi les hauteurs moyennes de tous les arbres de décision pour C et σ . Dans un contexte où C et σ sont clairs, on dira simplement que t est optimal.

Question 9. Écrire une fonction

```
val choix_possibles : 'x list array -> 'x array list
```

qui renvoie la liste des tableaux obtenus en choisissant un élément dans chaque liste du tableau d'entrée. Par exemple :

```
choix_possibles [| [1; 2]; [3; 4; 5] |] = [
  [| 1; 3 |];
  [| 1; 4 |];
  [| 1; 5 |];
  [| 2; 3 |];
  [| 2; 4 |];
  [| 2; 5 |];
]
```

Question 10. Écrire une fonction qui, étant donné un état initial σ_0 , renvoie la liste de tous les arbres de décision possibles pour σ_0 et l'ensemble de tous les caractères :

```
val enumere : etat -> arbre list
```

On rappelle que l'ensemble $\{0, 1, \dots, N-1\}$ de tous les caractères est représenté par la liste `caracteres` déclarée en variable globale.

Question 11. Écrire une fonction

```
val argmin : 'x list -> ('x -> float) -> 'x
```

telle que `argmin l f` renvoie un élément de `l` qui minimise `f`, si la liste est non-vide. En déduire une fonction qui, étant donné un état initial σ_0 , renvoie un arbre de décision optimal pour σ_0 et l'ensemble de tous les caractères :

```
val arbre_optimal : etat -> arbre
```

Partie III : Algorithme glouton

Afin d'éviter l'énumération de tous les arbres de décision dans l'algorithme de la partie précédente, on cherche un algorithme glouton : l'idée générale est de décider localement du caractère à choisir en fonction de l'état courant, plutôt que d'envisager tous les choix possibles. Pour cela, on décide de prendre le caractère qui minimise l'**entropie moyenne**. Avant de définir cette quantité, on introduit l'entropie simple $H(\mu)$ d'une distribution $\mu \in D(X)$, qui est donnée par :

$$H(\mu) \stackrel{\text{def}}{=} -\left(\sum_{x \in X, \mu(x) \neq 0} \mu(x) \ln(\mu(x))\right)$$

Question 12. Soit X un ensemble de cardinal $k \in \mathbb{N}$. Donner (en justifiant) les valeurs minimale et maximale de H sur $D(X)$ en fonction de k , ainsi que des distributions les atteignant.

Étant donnés un caractère i et un état σ , on définit l'entropie moyenne de i dans l'état σ , notée $H_i(\sigma)$, comme l'espérance des entropies sur les distributions obtenues en faisant une observation sur le caractère i :

$$H_i(\sigma) \stackrel{\text{def}}{=} \sum_{v \in V_i} P^\sigma(C_i = v) H(\sigma[i := v])$$

Question 13. Écrire une fonction qui renvoie l'entropie moyenne d'un caractère dans un état donné :

```
val score_caractere : etat -> caractere -> float
```

Question 14. Écrire une fonction qui calcule un arbre de décision en suivant l'algorithme glouton qui construit l'arbre en choisissant à la racine le caractère minimisant l'entropie moyenne, et procède récursivement de la même façon pour les sous-arbres :

```
val glouton : etat -> arbre
```

On se propose maintenant de montrer que l'algorithme glouton n'est pas optimal. On considère une situation avec deux caractères, $V_0 = V_1 = \{\text{oui}, \text{non}\}$, et trois espèces a , b et c :

$$\begin{aligned} a &= (\delta_{\text{oui}}, 0, 5 \cdot \delta_{\text{oui}} + 0, 5 \cdot \delta_{\text{non}}) \\ b &= (\delta_{\text{non}}, \delta_{\text{oui}}) \\ c &= (\delta_{\text{non}}, \delta_{\text{non}}) \end{aligned}$$

On considère enfin l'état initial σ_0 suivant :

$$a : 20\%, b : 40\%, c : 40\%$$

L'espèce a est donc plus rare que les espèces b et c qui sont plus communes.

Question 15. Dessiner les arbres de décision possibles pour l'état σ_0 et les caractères $\{0, 1\}$, et déterminer l'arbre de décision optimal.

Question 16. Calculer les entropies moyennes $H_0(\sigma_0)$ et $H_1(\sigma_0)$. Que peut-on en conclure ?
On admettra que $\ln(5) < \frac{12}{5} \ln(2)$.

Partie IV : Optimisation de l'algorithme naïf

On cherche à optimiser l'algorithme naïf de la partie II en évitant certains calculs inutiles. Pour permettre cela, on commence par changer la méthode de recherche d'un arbre optimal, en évitant de construire la liste de tous les arbres de décision possibles. La nouvelle méthode s'appuiera sur la fonction `arbre_optimal_avec_oracle` donnée en Figure 1. Le but de cette fonction est, étant donnés un état σ et une liste de caractères C , de renvoyer $(t, h(t))$ où t est un arbre optimal pour C et σ . Il faut bien noter que cette fonction n'est pas récursive, mais qu'elle dispose d'un argument supplémentaire `oracle` dont les appels pourront correspondre à des appels récursifs : il s'agit de *récursion ouverte*.

Question 17. Donner des hypothèses (pré-conditions) aussi précises que possible sur les arguments C , σ et `oracle` de la fonction `arbre_optimal_avec_oracle`, sous lesquelles la fonction termine toujours et renvoie bien $(t, h(t))$ où t est un arbre optimal pour C et σ . On veillera à répondre à cette question après avoir réfléchi aux deux suivantes, où il faudra démontrer puis exploiter la correction de la fonction par rapport à la spécification énoncée ici.

Question 18. Démontrer que la fonction satisfait bien cette spécification, en veillant notamment à énoncer clairement l'invariant de boucle.

Question 19. Écrire une fonction, récursive cette fois-ci, et basée sur `arbre_optimal_avec_oracle`, qui renvoie un arbre de décision optimal pour un état et une liste de caractères donnés :

```
val arbre_optimal : etat -> caractere list -> arbre * float
```

Donner sa spécification et montrer qu'elle la satisfait.

Question 20. On cherche maintenant à optimiser notre algorithme en interrompant la génération d'un arbre dès que son score dépasse celui de l'`arbre_optimal` courant. Pour cela, écrire une nouvelle fonction

```
val arbre_optimal_opt : etat -> caractere list -> float -> (arbre * float) option
```

qui prend un paramètre supplémentaire représentant le score à ne pas dépasser, et renvoie `None` si tous les arbres de décision possibles sont de hauteur moyenne supérieure au score maximal, ou bien un arbre optimal de hauteur moyenne inférieure s'il en existe un. La preuve de correction n'est pas demandée.

```

1 let arbre_optimal_avec_oracle
2   (oracle : etat -> caractere list -> arbre * float)
3   (etat : etat)
4   (caracteres : caractere list) : arbre * float =
5   if caracteres = [] then
6     (Feuille etat, 0.)
7   else
8     let caracteres_a_tester = ref caracteres in
9     let score_optimal = ref infinity in
10    let arbre_optimal = ref None in
11    while !caracteres_a_tester <> [] do
12      let caractere = List.hd !caracteres_a_tester in
13      let score_total = ref 1. in
14      let enfants : (float * arbre) array =
15        Array.init num.(caractere) (fun valeur ->
16          let p_valeur = proba_etat etat caractere valeur in
17          let sous_arbre, score_sous_arbre =
18            if p_valeur = 0. then Impossible,0. else
19              let etat' = reponse etat caractere valeur in
20              let caracteres_restants =
21                List.filter (fun c -> c <> caractere) caracteres
22                in
23                  oracle etat' caracteres_restants
24                in
25                score_total := !score_total +. p_valeur *. score_sous_arbre;
26                (p_valeur, sous_arbre))
27      in
28      if !score_total < !score_optimal then
29        (arbre_optimal := Some (Noeud {caractere=caractere;enfants=enfants}));
30        score_optimal := !score_total);
31      caracteres_a_tester := List.tl !caracteres_a_tester
32    done;
33    match !arbre_optimal with
34    | Some a -> a, !score_optimal
35    | None -> assert false

```

FIGURE 1 – Fonction `arbre_optimal_avec_oracle`.

Partie V : Observations complexes et formules logiques

Pour aller plus loin, on se propose de représenter des observations complexes sous forme de formules logiques construites à partir des caractères, par exemple “la fleur est bleue et a cinq pétales”. La syntaxe de cette logique est la suivante :

$$\varphi ::= c_i \in V \mid \varphi_1 \wedge \dots \wedge \varphi_m \mid \varphi_1 \vee \dots \vee \varphi_m \quad (0 \leq i < N, V \subseteq V_i, m \geq 0)$$

Cela signifie que l’ensemble des formules est le plus petit ensemble contenant les formules **atomiques** de la forme $c_i \in V$ où $0 \leq i < N$ et $V \subseteq V_i$, et que si $\varphi_1, \dots, \varphi_m$ sont des formules pour $m \geq 0$, alors la **conjonction** $\varphi_1 \wedge \dots \wedge \varphi_m$ et la **disjonction** $\varphi_1 \vee \dots \vee \varphi_m$ sont aussi des formules. Une formule $c_i \in V$ signifie intuitivement que le caractère i prend une valeur dans l’ensemble $V \subseteq V_i$. On définit la formule \perp (la formule fausse) comme la disjonction vide, et \top (la formule vraie) comme la conjonction vide.

Par exemple, avec les ensembles V_0 et V_1 de l’exemple utilisé en partie I, $c_1 \in \{\text{bleu}\} \wedge c_0 \in \{5\}$ est une formule, correspondant à l’énoncé “la fleur est bleue et a cinq pétales”. Ou encore, si $V_i = \{0, 1, 2, 3, 4\}$, $c_i \in \{1\} \wedge c_i \in \{2, 3\}$ est une formule, correspondant à l’énoncé contradictoire “le caractère i vaut 1, et il vaut 2 ou 3”.

On définit la taille d’une formule φ , notée $|\varphi|$, comme suit :

$$\begin{aligned} |c_i \in V| &= 1 + |V| \\ |\varphi_1 \wedge \dots \wedge \varphi_m| &= 1 + m + \sum_{1 \leq i \leq m} |\varphi_i| \\ |\varphi_1 \vee \dots \vee \varphi_m| &= 1 + m + \sum_{1 \leq i \leq m} |\varphi_i| \end{aligned}$$

Dans un premier temps, nous définissons quand une situation certaine (dite déterministe) satisfait une certaine formule. On appelle **modèles déterministes** les éléments de $V_0 \times \dots \times V_{N-1}$. On notera $\vec{v} \in \vec{V}$ pour signifier que \vec{v} est un tel modèle, et dans ce cas on notera v_i la i -ème composante de \vec{v} , ce qui revient à dire que $\vec{v} = (v_0, \dots, v_{N-1})$. On définit une relation de **satisfaction** entre les modèles déterministes et les formules, notée $\vec{v} \models \varphi$, comme suit :

$$\begin{aligned} \vec{v} \models c_i \in V &\quad \text{si et seulement si } v_i \text{ appartient à l’ensemble } V \\ \vec{v} \models \varphi_1 \wedge \dots \wedge \varphi_m &\quad \text{si et seulement si } \vec{v} \models \varphi_k \text{ pour tout } k \in \{1, \dots, m\} \\ \vec{v} \models \varphi_1 \vee \dots \vee \varphi_m &\quad \text{si et seulement si } \vec{v} \models \varphi_k \text{ pour un } k \in \{1, \dots, m\} \end{aligned}$$

On étend ensuite cette relation de satisfaction au cadre probabiliste de la manière suivante. Un **modèle probabiliste** est un élément de $D(V_0) \times \dots \times D(V_{N-1})$. Un tel modèle associe une distribution de probabilité plutôt qu’une valeur déterminée à chaque caractère. On remarquera que cette modélisation suppose que les valeurs de deux caractères distincts sont probabilistes mais indépendantes l’une de l’autre. Étant donnés une formule φ et un modèle probabiliste (d_0, \dots, d_{N-1}) , on définit la probabilité que $\vec{d} = (d_0, \dots, d_{N-1})$ satisfasse φ , notée $P(\vec{d} \models \varphi)$, ainsi :

$$P(\vec{d} \models \varphi) \stackrel{\text{def}}{=} \sum_{\vec{v} \in \vec{V}, \vec{v} \models \varphi} \prod_{0 \leq i < N} d_i(v_i)$$

Question 21. Pour deux formules φ et ψ , montrer l’équivalence entre ces deux propositions :

- Pour tout modèle déterministe \vec{v} , $\vec{v} \models \varphi$ si et seulement si $\vec{v} \models \psi$.
- Pour tout modèle probabiliste \vec{d} , $P(\vec{d} \models \varphi) = P(\vec{d} \models \psi)$.

On dira que φ et ψ sont **équivalentes**, noté $\varphi \equiv \psi$, quand l'une ou l'autre des propositions précédentes est vraie.

Dans les questions qui suivent, on reprend le codage OCaml des distributions, et on représente directement les formules via le type suivant :

```
type formule =
| Feuille of caractere * valeur list
| Conjonction of formule list
| Disjonction of formule list
```

On supposera de plus que les listes codant les ensembles de valeurs dans les formules atomiques sont sans doublons. On codera les modèles déterministes (resp. probabilistes) par des tableaux de N valeurs (resp. distributions). On notera enfin K le cardinal maximal des V_i :

$$K \stackrel{\text{def}}{=} \max_{0 \leq i < N} |V_i|$$

Question 22. On considère l'algorithme qui, étant donnés un modèle probabiliste \vec{d} et une formule φ , calcule $P(\vec{d} \models \varphi)$ en suivant naïvement sa définition. Quelle est la complexité temporelle de cet algorithme, en fonction de N , K et $|\varphi|$? Un programme OCaml détaillé n'est pas demandé, mais on veillera à en décrire les aspects nécessaires pour justifier l'analyse de complexité.

Dans la suite, nous allons élaborer une méthode de calcul de $P(\vec{d} \models \varphi)$ dont la complexité ne dépend pas de N . Cette méthode s'appuie sur des formes particulières de formules. Une **clause** est une conjonction de formules atomiques concernant des caractères différents. En reprenant les exemples précédents, $c_1 \in \{\text{bleu}\} \wedge c_0 \in \{5\}$ est une clause mais pas $c_i \in \{1\} \wedge c_i \in \{2, 3\}$.

Question 23. Décrire un algorithme qui, étant donnés un modèle probabiliste \vec{d} et une clause φ , calcule $P(\vec{d} \models \varphi)$ avec une complexité qui ne dépend que de K et $|\varphi|$, mais pas de N . Justifier la correction de l'algorithme et son analyse de complexité. Il n'est pas nécessaire de donner un programme OCaml détaillé.

Question 24. Étant données deux clauses φ et ψ , montrer que $\varphi \wedge \psi$ est équivalente à une clause que l'on notera $\varphi \overline{\wedge} \psi$.

Question 25. Montrer que pour toute formule φ il existe $m \geq 0$ et des clauses $\varphi_1, \dots, \varphi_m$ telles que $\varphi \equiv \varphi_1 \vee \dots \vee \varphi_m$.

Question 26. Décrire un algorithme qui, étant donnés un modèle probabiliste \vec{d} et une formule φ , calcule $P(\vec{d} \models \varphi)$ avec une complexité temporelle (potentiellement non-polynomiale) qui ne dépend que de K et $|\varphi|$ mais pas de N .

En pratique, un tel algorithme est utile car on va avoir des modèles avec de nombreux caractères mais un nombre de valeurs possibles restreint pour chaque caractère, et l'on s'intéressera à des formules de taille limitée.

Fin du sujet.

**ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES**

CONCOURS D'ADMISSION 2025

**JEUDI 17 AVRIL 2025
08h00 - 12h00**

FILIERE MPI - Epreuve n° 7

INFORMATIQUE C (XULSR)

Durée : 4 heures

***L'utilisation des calculatrices n'est pas autorisée pour
cette épreuve***

Couplages maximaux et parfaits

Le sujet comporte 18 pages, numérotées de 1 à 18.

Début de l'épreuve.

Complexité. Par la complexité d'un algorithme, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de celui-ci dans le pire cas. Lorsque la complexité dépend d'un ou plusieurs paramètres k_0, \dots, k_r , on dit que l'algorithme a une complexité en $O(f(k_0, \dots, k_r))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de k_0, \dots, k_r suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres k_0, \dots, k_r la complexité est au plus $C \times f(k_0, \dots, k_r)$.

Fonctions utiles en OCaml. Dans les questions de programmation en OCaml, il est interdit d'utiliser des fonctions de la bibliothèque qui ne sont pas incluses dans la liste suivante.

- `Option.get: 'a option -> 'a` renvoie `v` si l'argument de type `'a option` est égal à `Some v`, et lève une exception sinon. La complexité est en $O(1)$.
- `List.length: 'a list -> int` renvoie la taille de la liste donnée en argument. La complexité est en $O(n)$, où n est la longueur de la liste donnée en argument.
- `List.hd: 'a list -> 'a` renvoie le premier élément de la liste donnée en argument si celle-ci n'est pas vide, et lève une exception sinon. La complexité est en $O(1)$.
- `List.tl: 'a list -> 'a list` renvoie la liste donnée en argument privée de son premier élément si la liste donnée en argument n'est pas vide, et lève une exception sinon. La complexité est en $O(1)$.
- `List.rev: 'a list -> 'a list` renvoie la liste donnée en argument dans l'ordre inverse. La complexité est en $O(n)$, où n est la longueur de la liste donnée en argument.
- `List.map: ('a -> 'b) -> 'a list -> 'b list` renvoie la liste de type `'b list` obtenue en appliquant la fonction de type `'a -> 'b` donnée en premier argument à chaque élément de la liste de type `'a list` donnée en second argument. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.map f lst` est en $O(n)$, où n est la longueur de `lst`.
- `List.mem: 'a -> 'a list -> bool` renvoie `true` si la valeur donnée en premier argument est un élément de la liste donnée en second argument, et renvoie `false` sinon. La complexité est en $O(n)$, où n est la longueur de la liste donnée en second argument.
- `List.assoc: 'a -> ('a * 'b) list -> 'b`: cette fonction spécifique aux listes d'association est décrite en page 4.

Fonctions utiles en C. Dans les questions de programmation en C, on pourra supposer que les en-têtes `<stdbool.h>`, `<stdio.h>`, `<stdlib.h>`, et `<math.h>` ont été inclus. En particulier, on pourra utiliser la fonction `double log2(double x)` définie par l'en-tête `<math.h>`, qui calcule le logarithme en base 2.

Présentation du sujet. Ce sujet concerne les couplages dans les graphes non orientés. La partie I, à composer dans le langage OCaml, concerne la recherche d'un couplage maximal dans un graphe. On étudiera et implémentera un algorithme dû à Edmonds. Les parties II et III, à composer dans le langage C, concernent le problème de l'existence d'un couplage parfait dans un graphe. On y implémente des méthodes algébriques et probabilistes basées sur des calculs de déterminants. La partie II concerne l'implémentation d'un algorithme pour calculer le déterminant d'une matrice carrée dû à Mahajan et Vinay. La partie III exploite cet algorithme pour implémenter un algorithme probabiliste testant l'existence d'un couplage parfait dans un graphe non orienté.

Les parties I et II sont indépendantes. La partie III dépend de la partie II et de notions introduites en partie I.

Graphes non orientés. On travaillera dans les parties I et III sur des graphes non orientés et sans boucles. Un tel graphe G est défini par une paire d'ensembles finis (V, E) , où V est l'ensemble des sommets du graphe et E l'ensemble de ses arêtes. Une arête est une paire non-ordonnée $\{u, v\}$ d'exactlyement deux sommets $u, v \in V$. Ainsi, les arêtes sont non orientées ($\{u, v\} = \{v, u\}$) et il n'y a pas de boucles ($\{v\} = \{v, v\}$ n'est jamais une arête de G). Une arête $\{u, v\}$ est dite **incidente** aux sommets u et v . On supposera toujours que V est non vide.

Étant donné un graphe $G = (V, E)$:

- Un **chemin** est une suite finie de sommets v_0, v_1, \dots, v_n avec $n \geq 0$ et telle que pour tout $i = 0, \dots, n - 1$, on a $\{v_i, v_{i+1}\} \in E$. La **longueur** du chemin v_0, v_1, \dots, v_n est n .
- Un chemin est **élémentaire** si ses sommets sont deux-à-deux distincts.
- Un **cylce** est un chemin v_0, v_1, \dots, v_n de longueur ≥ 3 , avec $v_0 = v_n$ et tel que le chemin v_1, \dots, v_n est élémentaire. Autrement dit, le seul sommet autorisé à apparaître plusieurs fois dans un cylce v_0, v_1, \dots, v_n est le sommet $v_0 = v_n$, qui apparaît exactement deux fois (une fois au début et une fois à la fin).

Partie I : Algorithme d'Edmonds

Cette partie concerne l'algorithme d'Edmonds pour la recherche de couplage maximal.

Question 1. Écrire une fonction `del`: '`a list -> 'a list -> 'a list` qui prend en entrée deux listes `lst1` et `lst2` et renvoie la liste des éléments de `lst1` n'apparaissant pas dans `lst2`. On attend une complexité en $O(m_1 m_2)$ où m_1 est la longueur de `lst1` et m_2 est la longueur de `lst2`, sans la justifier.

On suppose définie une fonction `join`: '`a list -> 'a list -> 'a list` telle que `join lst1 lst2` est une liste sans répétitions contenant l'ensemble de tous les éléments apparaissant dans `lst1` et de tous les éléments apparaissant dans `lst2`.

Nous utilisons des listes d'association pour représenter les graphes. Les listes d'association implémentent une structure de dictionnaire ; il s'agit de listes de paires (`a, b`), dont le premier élément `a` représente la clef et le second élément `b` représente la valeur associée à cette clef.

Il existe des fonctions OCaml permettant de travailler avec les listes d'association. Dans la suite, nous utiliserons seulement la fonction `List.assoc`: '`a -> ('a * 'b) list -> 'b` qui prend en arguments une clef `x` et une liste d'association `lst`, et renvoie la valeur `b` associée à cette clef, c'est-à-dire l'élément `b` du premier couple `(x, b)` appartenant à `lst`, s'il en existe. La fonction lève une exception si `x` n'est pas une clef de la liste `lst`. La complexité est en $O(n)$, où n est la longueur de `lst`.

Question 2. Écrire une fonction `keys`: '`('a * 'b) list -> 'a list` qui prend en entrée une liste d'association `lst` et qui renvoie la liste de ses clefs.

On attend une complexité en $O(m)$ où m est la longueur de `lst`, sans la justifier.

Nous représentons les graphes en utilisant le type OCaml suivant :

```
type graphe = (int * (int list)) list
```

Un graphe est représenté par une liste de paires `(a, lst)` où `a` est un sommet du graphe et `lst` est la liste d'adjacence de ce sommet. Dans la suite, on suppose toujours que :

- les sommets du graphe sont exactement les clefs de la liste d'association ;
- chaque clé de la liste d'association est unique.

Par exemple, le graphe

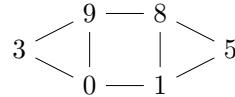
2 — 0 — 4

peut être représenté par la liste d'association `[(0, [2;4]); (4, [0]); (2, [0])]`.

Couplages

Un **couplage** C dans un graphe G est un ensemble (possiblement vide) d'arêtes de G sans intersections : un sommet de G ne peut appartenir à plus d'une arête dans C .

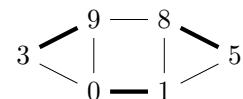
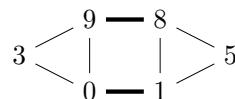
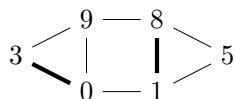
Considérons par exemple le graphe :



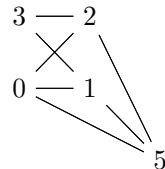
Les ensembles d'arêtes suivants sont des couplages dans ce graphe :

$$\{\{0, 3\}, \{1, 8\}\} \quad \{\{0, 1\}, \{8, 9\}\} \quad \{\{0, 1\}, \{5, 8\}, \{3, 9\}\}$$

Les arêtes d'un couplage sont représentées par des traits plus épais. Par exemple, les trois couplages ci-dessus sont représentés comme suit :



Question 3. On considère le graphe suivant :



Indiquer lesquels des ensembles suivants sont des couplages dans le graphe ci-dessus :

- (1) $\{\{1, 2\}, \{0, 5\}\}$,
- (2) $\{\{1, 3\}, \{0, 5\}\}$,
- (3) $\{\{0, 1\}, \{2, 3\}, \{0, 5\}\}$.

Un couplage est représenté par une liste de paires de sommets :

```
type couplage = (int * int) list
```

Dans toute la suite, une arête $\{a, b\}$ sera représentée par la paire $(\min a, b)$, $(\max a, b)$.

Étant donné un couplage C dans un graphe G , on dit qu'un sommet v de G est **couvert** par C s'il existe une arête dans le couplage C qui est incidente à v .

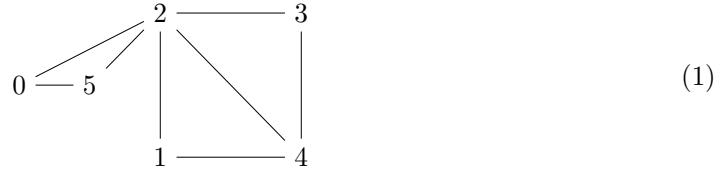
Question 4. Écrire une fonction `couverts`: `couplage -> int list` qui prend en entrée un couplage `cpl` et qui renvoie la liste des sommets couverts par `cpl`.

On attend une complexité en $O(m)$ où m est la longueur de la liste `cpl`, sans la justifier.

Couplages maximaux

Nous allons maintenant nous intéresser aux couplages maximaux, c'est-à-dire aux couplages ayant un nombre maximal d'arêtes. On remarque qu'un couplage dans un graphe à n sommets ne peut contenir plus de $\lfloor n/2 \rfloor$ arêtes.

Considérons le graphe suivant :



Le couplage $\{\{0, 5\}, \{2, 3\}, \{1, 4\}\}$ est maximal. Mais le couplage $\{\{0, 5\}, \{2, 4\}\}$ n'est pas maximal bien qu'il ne soit pas possible de lui adjoindre d'arête supplémentaire.

Question 5. Donner un autre couplage maximal dans le graphe ci-dessus.

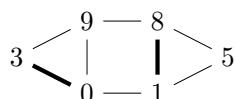
Chemins d'augmentation

L'algorithme d'Edmonds est basé sur la recherche de chemins d'augmentation. Étant donné un couplage C dans un graphe G , un **chemin d'augmentation** pour C (dans G) est un chemin élémentaire v_0, \dots, v_n dans le graphe G tel que $n > 0$ et :

- v_0 et v_n ne sont pas couverts par C ;
- c'est un chemin **alternant** : pour tout $i = 0, \dots, n-2$, on a $\{v_i, v_{i+1}\} \in C$ si et seulement si $\{v_{i+1}, v_{i+2}\} \notin C$.

Un chemin d'augmentation P pour C permet de construire un nouveau couplage $C(P)$ constitué des arêtes de C qui ne sont pas dans P et des arêtes de P qui ne sont pas dans C .

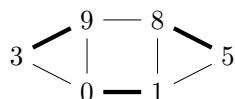
Par exemple, dans le graphe



le chemin suivant est un chemin d'augmentation :

$$9 — 3 — 0 — 1 — 8 — 5$$

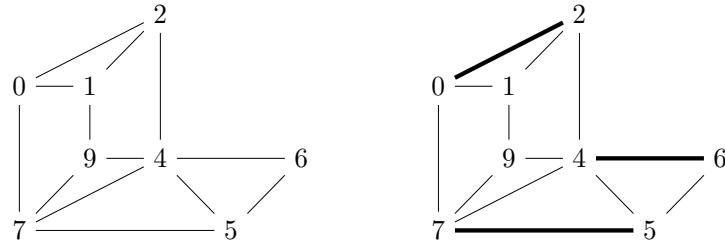
Le nouveau couplage obtenu à partir de ce chemin d'augmentation est le suivant :



Question 6. Considérons le graphe en (1) ci-dessus. Donner un chemin d'augmentation P

pour le couplage $C = \{\{0, 5\}, \{2, 4\}\}$ de manière à ce que le couplage $C(P)$ soit maximal.

Question 7. On considère le graphe et le couplage suivants :



Donner un chemin d'augmentation et le couplage augmenté correspondant.

Question 8. Soit P un chemin d'augmentation pour un couplage C dans un graphe G . Montrer que $C(P)$ est bien un couplage, et qu'il contient strictement plus d'arêtes que C .

En OCaml, un chemin est représenté par une liste de sommets, c'est-à-dire par une valeur de type `int list`.

Question 9. Écrire une fonction

```
separer : int list -> ((int * int) list) * ((int * int) list)
```

qui prend en entrée un chemin d'augmentation `chm` et renvoie deux listes `lstin` et `lstout` où `lstin` contient les arêtes composant le chemin `chm` qui appartiennent au couplage, et où `lstout` contient les arêtes composant le chemin `chm` qui n'appartiennent pas au couplage.

On attend une complexité en $O(m)$, où m est la longueur de la liste `chm`, sans la justifier.

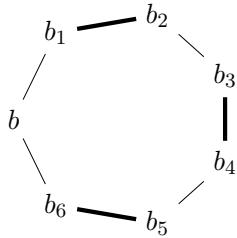
Question 10. Écrire une fonction `augmente`: `couplage -> int list -> couplage` qui prend en entrée un couplage `cpl` et un chemin d'augmentation `chm`, et qui renvoie le couplage `cpl(chm)`.

L'algorithme d'Edmonds, que nous allons voir ci-dessous, repose essentiellement sur le lemme de Berge. Ce résultat, que nous admettrons, énonce qu'un couplage C dans un graphe G est maximal si et seulement s'il n'existe pas de chemin d'augmentation pour C dans G .

Bourgeons

L'algorithme d'Edmonds va de plus effectuer des opérations de **contraction de bourgeons**. Soit C un couplage dans un graphe G . Un bourgeon est un cycle dans G de longueur impaire $2m + 1$ dont exactement m arêtes sont dans C . La **base** b d'un bourgeon B est le seul sommet

de B dont les deux arêtes adjacentes dans B ne sont pas dans C :



La contraction d'un bourgeon consiste à identifier tous les sommets de celui-ci. On représente un bourgeon B par une valeur de type `int list` qui contient exactement les sommets de B , dans un ordre quelconque, mais sans répétition. Par exemple, le bourgeon illustré ci-dessus peut être représenté par la liste `[b1; b6; b2; b5; b3; b4]`.

Étant donné un graphe $G = (V, E)$ et une liste de sommets $L = \ell_1, \dots, \ell_k$, le **graphe contracté** G/L est défini comme suit :

- L'ensemble des sommets du graphe G/L est $(V \setminus L) \cup \{w\}$ où w , le **nouveau sommet de** G/L , est un entier qui n'est pas un sommet de G .
 - Les arêtes de G/L sont les arêtes de G entre sommets n'étant pas dans L , ainsi que les arêtes de la forme $\{u, w\}$ où u n'appartient pas à L mais est adjacent dans G à un sommet de L . L'ensemble des arêtes de G/L est donc défini comme
- $$\{\{u, v\} \in E \mid u \notin L \text{ et } v \notin L\} \cup \{\{u, w\} \mid u \notin L \text{ et } u \text{ adjacent dans } G \text{ à un sommet de } L\}$$

On suppose donnée une fonction `renomme`: `int list -> int list -> int -> int list` telle que `renomme lst rnm w` renvoie une liste obtenue en remplaçant dans `lst` tous les éléments de `rnm` par `w`, et en supprimant tous les doublons.

Question 11. Écrire une fonction `contracteG`: `graphe -> int list -> int -> graphe` qui prend en entrée un graphe `grph`, une liste de sommets `lst`, et le nom `w` du nouveau sommet, et qui renvoie le graphe contracté `grph/lst`.

Soit C un couplage dans un graphe G , et soit B un bourgeon. On définit un couplage C/B dans le graphe contracté G/B . En notant w le nouveau sommet de G/B , l'ensemble C/B a pour éléments :

- les arêtes $\{a, b\} \in C$ telles que $a \notin B$ et $b \notin B$;
- les arêtes $\{a, w\}$ telles que $a \notin B$ et telles qu'il existe un $b \in B$ avec $\{a, b\} \in C$.

On remarquera que C/B ne dépend que de C , B , et du nom w du nouveau sommet.

Question 12. Montrer que C/B est un couplage dans G/B .

Question 13. Écrire une fonction

`contracteC` : `couplage -> int list -> int -> couplage`

qui prend en entrée un couplage `cpl`, un bourgeon `brg`, et le nom `w` du nouveau sommet, et qui renvoie le couplage `cpl/brg`.

On admettra dans la suite qu'il existe un chemin d'augmentation pour le couplage C/B dans G/B si et seulement s'il existe un chemin d'augmentation pour le couplage C dans G .

Recherche de chemins d'augmentation

La recherche de chemins d'augmentation va utiliser des forêts, c'est-à-dire des listes d'arbres. Les types ci-dessous représentent des arbres et des forêts dont les nœuds sont étiquetés par des entiers. Ces entiers seront par la suite des sommets d'un graphe.

```
type arbre = N of int * foret
and foret = arbre list
```

Question 14. Écrire une fonction `find: foret -> int -> (int list) option` telle que `find foret v` renvoie `None` si l'entier `v` n'étiquette aucun des nœuds de `foret`, et telle que `find foret v` renvoie `Some c` sinon, où `c` est la liste des étiquettes le long d'un chemin allant d'une racine à un nœud étiqueté `v` dans `foret`.

On attend une complexité en $O(n)$, où n est le nombre de nœuds de `foret`, sans la justifier.

Question 15. Écrire une fonction `extend: foret -> int -> int -> foret`, qui prend en arguments une forêt `foret` et deux entiers `u` et `v`, et qui renvoie une copie de `foret` dans laquelle pour chaque nœud étiqueté par `u`, on a créé un nouvel enfant étiqueté par `v`.

On attend une complexité en $O(n)$, où n est le nombre de nœuds de `foret`, sans la justifier.

La **profondeur** d'un nœud dans un arbre est définie inductivement comme suit : la racine est à profondeur 0, et pour $h \geq 0$, les nœuds à profondeur $h + 1$ sont les enfants des nœuds à profondeur h .

L'algorithme de recherche de chemins d'augmentation prend en entrée un graphe G et un couplage C dans G . Il manipule un ensemble de sommets T et peut être décrit comme suit.

- (1) On construit une forêt F dont les nœuds consistent exactement en une racine u pour chaque sommet u de G non couvert par C .
- (2) On fixe $T = \emptyset$.
- (3) Tant qu'il existe un sommet $u \notin T$ tel que u apparaît à profondeur paire dans F :

On ajoute u à l'ensemble T .

Pour chaque voisin v de u dans G tel que $v \notin T$:

- (a) Si v n'est pas dans la forêt F , il est couvert par une arête $\{v, z\}$ de C , et on ajoute les arêtes $\{u, v\}$ et $\{v, z\}$ à F .
- (b) Si v est dans la forêt F :
 - (i) Si u, v apparaissent à profondeurs paires dans des arbres distincts de racines respectives r_u, r_v , alors on renvoie la suite des sommets de G vus dans F en allant de r_u à u (r_u et u inclus), puis en allant de v à r_v (v et r_v inclus).
 - (ii) Si u et v apparaissent dans le même arbre à profondeurs paires, alors on construit le bourgeon B correspondant, et on cherche un chemin d'augmentation pour le couplage C/B dans le graphe G/B . Si on en trouve un, alors on renvoie un chemin d'augmentation pour C dans G .

Question 16. Montrer que la suite de sommets renvoyée par l'algorithme dans le cas (3)(b)(i) est un chemin d'augmentation pour C dans G .

On admet dans toute la suite que l'algorithme ci-dessus renvoie un chemin d'augmentation pour C si et seulement s'il en existe un.

Nous allons étudier une implémentation OCaml de cet algorithme. On suppose données les fonctions suivantes.

- La fonction `frais: int list -> int` renvoie un entier n'appartenant pas à la liste donnée en argument.
- La fonction `prochain: foret -> int list -> int option` prend en arguments une forêt f et une liste de sommets lst . Un appel `prochain f lst` renvoie `Some d` où d est un entier apparaissant dans f à profondeur paire et n'appartenant pas à lst si un tel entier d existe. Sinon, `prochain f lst` renvoie `None`.
- La fonction `apparie: couplage -> int -> int` prend en arguments un couplage cpl et un sommet u couvert par le couplage, et renvoie le sommet apparié, c'est-à-dire l'unique v tel que $(\min u v, \max u v)$ appartient à cpl .
- La fonction `gonfle: graphe -> int list -> int list -> int -> int list` prend en arguments un graphe $grph$, un bourgeon brg , un chemin d'augmentation dans le graphe contracté $grph/brg$, et le nom du nouveau sommet nv , et renvoie le chemin d'augmentation correspondant dans $grph$.

La figure 1 page 11 représente une implémentation *incomplète* d'une fonction

```
recherche : graphe -> couplage -> (int list) option
```

Les questions ci-dessous demandent de donner le code de la ligne 15, des lignes 17–18 et des lignes 24–26, ainsi que d'implémenter la fonction `extrait` (ligne 20).

La fonction `recherche` prend en arguments un graphe et un couplage. Elle doit renvoyer `Some p` s'il existe un chemin d'augmentation p , et renvoyer `None` sinon. Certaines fonctions appelées dans le code de `recherche` peuvent lever des exceptions, mais on admettra que ces exceptions ne sont jamais levées si les données d'entrée sont bien formées.

Question 17. *Donner le code devant apparaître à la ligne 15 de la figure 1.*

Question 18. *Donner le code devant apparaître dans le bloc aux lignes 17–18 de la figure 1.*

Question 19. *Écrire une fonction `extrait: int list -> int list -> int list` telle que l'appel `extrait cu cv` ligne 20 renvoie le bourgeon associé aux chemins `cu` et `cv`.*

Question 20. *Donner le code devant apparaître dans le bloc aux lignes 24–26 de la figure 1.*

Question 21. *Écrire une fonction `edmonds: graphe -> couplage` qui prend en entrée un graphe et qui renvoie un couplage maximal.*

```

1 let rec recherche graphe couplage =
2   let nc = del (keys graphe) (couverts couplage) in
3   let foret = ref (List.map (fun x -> N(x,[])) nc) in
4   let rec aux_voisins u liste_voisins =
5     let cu = Option.get (find !foret u) in
6     match liste_voisins with
7     | [] -> None
8     | v :: tl ->
9       match find !foret v with
10      | None -> let z = apparie couplage v in
11          foret := extend (extend !foret u v) v z;
12          aux_voisins u tl
13      | Some cv when (List.length cv) mod 2 = 0
14        (* nombre de sommets pair = profondeur impaire *)
15        -> ...
16      | Some cv when (List.hd cu) <> (List.hd cv)
17        -> ...
18        ...
19      | Some cv
20        -> let bourgeon = extrait cu cv in
21          let nv = frais (keys graphe) in
22          let ng = contracteG graphe bourgeon nv in
23          let nc = contracteC couplage bourgeon nv in
24          ...
25          ...
26          ...
27
28   in
29   let rec aux_sommets traites =
30     match prochain !foret traites with
31     | None -> None
32     | Some u -> let liste_voisins = del (List.assoc u graphe) traites in
33       match aux_voisins u liste_voisins with
34       | None -> aux_sommets (u::traites)
35       | Some ch -> Some ch
36
37   in
38   aux_sommets []

```

FIGURE 1 – Fonction recherche.

Partie II : Calculs de déterminants

Cette partie concerne l'algorithme de Mahajan-Vinay pour le calcul de déterminants de matrices carrées. Elle est à composer dans le langage C.

Préliminaires : matrices linéarisées

On travaille avec des matrices carrées de taille arbitraire indexées à partir de 0. On utilise une représentation linéarisée de ces matrices. Une matrice A de dimension $n \times n$ (c'est-à-dire avec n lignes et n colonnes) est représentée par un tableau A avec n^2 éléments de manière à ce que le coefficient $A_{i,j}$ de la matrice (indice de ligne i et indice de colonne j) corresponde à l'élément $A[i*n+j]$ du tableau.

On suppose dans la suite que les opérations arithmétiques ont une complexité en $O(1)$.

Question 22. Écrire des fonctions

```
int read_sqmatrix (int n, int *A, int i, int j)
void write_sqmatrix (int n, int *A, int i, int j, int val)
```

telles que

- `read_sqmatrix(n,A,i,j)` renvoie la valeur du coefficient $A_{i,j}$ où A est la matrice $n \times n$ représentée par A ;
- `write_sqmatrix(n,A,i,j,val)` modifie la valeur du coefficient $A_{i,j}$ pour qu'il soit égal à val .

On supposera que A est un tableau avec n^2 éléments et que $i, j \in \{0, \dots, n-1\}$.

On attend une complexité en $O(1)$ pour ces fonctions, sans la justifier.

Algorithme de Mahajan-Vinay

L'algorithme de Mahajan-Vinay est basé sur une formulation du déterminant d'une matrice A en termes de **suites de marches fermées** dans un graphe associé à A .

Soit A une matrice carrée de dimension $n \times n$. On associe à A son **graphe d'adjacence** $G(A)$. Le graphe $G(A)$ est **orienté et pondéré**. Il est défini comme suit :

- les sommets de $G(A)$ sont les entiers de 0 à $n-1$,
- les arcs sont les paires ordonnées $e = (i, j)$ telles que $A_{i,j} \neq 0$,
- le poids $\omega(e)$ d'un arc $e = (i, j)$ est égal à $A_{i,j}$.

Les sommets de $G(A)$ sont donc des entiers. L'ordre usuel sur ces entiers est utilisé dans la suite de cette partie.

Question 23. Dessiner le graphe d'adjacence de la matrice suivante :

$$\begin{pmatrix} 2 & 3 \\ -2 & 1 \end{pmatrix}$$

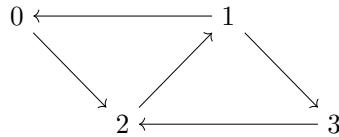
Une **marche fermée** dans le graphe $G(A)$ est une suite de sommets $C = v_0 \dots v_m$ avec $m > 0$ et qui satisfait les conditions suivantes :

- pour tout $i = 0, \dots, m - 1$, il existe un arc (v_i, v_{i+1}) dans $G(A)$;
- le sommet $v_0 = v_m$ est le plus petit entier de la suite, appelé la **tête** de C , notée $t(C)$;
- le sommet v_0 n'apparaît qu'au début et à la fin de la suite : pour tout $i = 1, \dots, m - 1$, on a $v_i \neq v_0$.

La **longueur** $|C|$ de la marche fermée C est le nombre d'arcs qui la composent. La longueur de $C = v_0 \dots v_m$ est donc égale à m . Le **poids** $\omega(C)$ d'une marche fermée C est le produit des poids des arcs (avec multiplicités) qui la composent :

$$\omega(C) = \prod_{i=0}^{m-1} \omega(v_i, v_{i+1}).$$

Exemple. Considérons le graphe suivant, dont les arcs ont tous poids 1 :



Dans ce graphe, les suites de sommets suivantes sont des marches fermées :

$$0 \rightarrow 2 \rightarrow 1 \rightarrow 0 \quad \text{et} \quad 0 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0.$$

On remarque qu'il est possible de passer plusieurs fois par le même sommet. Cependant la suite

$$0 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow 2 \rightarrow 1 \rightarrow 0$$

n'est pas une marche fermée car la tête ne doit apparaître qu'au début et à la fin. La suite

$$2 \rightarrow 1 \rightarrow 0 \rightarrow 2$$

n'est pas non plus une marche fermée car la tête n'est pas le plus petit sommet.

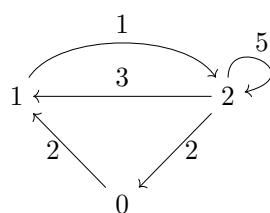
Question 24. Donner une marche fermée de longueur 3 et de tête 1 dans le graphe ci-dessus.

Une **suite de marches fermées** est une suite S de marches fermées C_1, \dots, C_k telle que :

- les têtes sont croissantes : $t(C_1) < t(C_2) < \dots < t(C_k)$;
- le nombre total d'arcs (en comptant les multiplicités) est égal au nombre de sommets n de $G(A)$: $\sum_{i=1}^k |C_i| = n$.

Le **poids** d'une suite de marches fermées est le produit des poids des marches fermées qui la composent : $\omega(S) = \prod_{i=1}^k \omega(C_i)$. La **tête** de S , notée $t(S)$, est la tête de la première marche fermée de S : $t(S) = t(C_1)$.

Question 25. On considère le graphe suivant :



- (1) Donner les marches fermées de longueur au plus 3 et leurs poids.
(2) Donner les suites de marches fermées et leurs poids.

On définit aussi le **signe** d'une suite de marches fermées. Si $S = C_1 \dots C_k$, alors on pose :

$$\text{sgn}(S) = (-1)^{n+k}.$$

On dit que S est une suite de marches fermées **positive** lorsque $\text{sgn}(S) = 1$, et que S est une suite de marches fermées **négative** lorsque $\text{sgn}(S) = -1$.

Le résultat fondamental de Mahajan et Vinay est que le déterminant d'une matrice carrée A peut être exprimé comme suit :

$$\det(A) = \sum_{\substack{S \text{ suite de marches fermées} \\ \text{dans } G(A)}} \text{sgn}(S) \omega(S).$$

Soit A une matrice avec n lignes et n colonnes. Pour implémenter le calcul du déterminant de A selon la formule ci-dessus, Mahajan et Vinay introduisent un nouveau graphe orienté et pondéré H_A , que nous allons maintenant décrire.

Le graphe H_A possède trois sommets distingués : s , t_0 , et t_1 . La construction de H_A permet d'assurer que les chemins de s à t_0 (respectivement t_1) dans H_A correspondent aux suites de marches fermées positives (respectivement négatives) dans $G(A)$. Les autres sommets sont des quadruplets $\langle p, h, u, i \rangle$ avec $p \in \{0, 1\}$, $0 \leq h \leq u \leq n-1$, et $i \in \{0, \dots, n-1\}$. Ceux-ci représentent des étapes de tentatives de construction de suites de marches fermées. Le sommet $\langle p, h, u, i \rangle$ représente le cas où des marches fermées C_1, \dots, C_k ont été construites, et où on cherche à construire une marche fermée commençant par $C_{k+1} = v_0, v_1, \dots, v_m$, où

- le signe de la suite de marches fermées C_1, \dots, C_k est $(-1)^p$;
- h est la tête de la marche fermée en cours de construction, c'est-à-dire $h = v_0$;
- u est le sommet courant, c'est-à-dire $u = v_m$;
- i est le nombre d'arcs parcourus jusque là, c'est-à-dire $i = m + \sum_{j=1}^k |C_j|$.

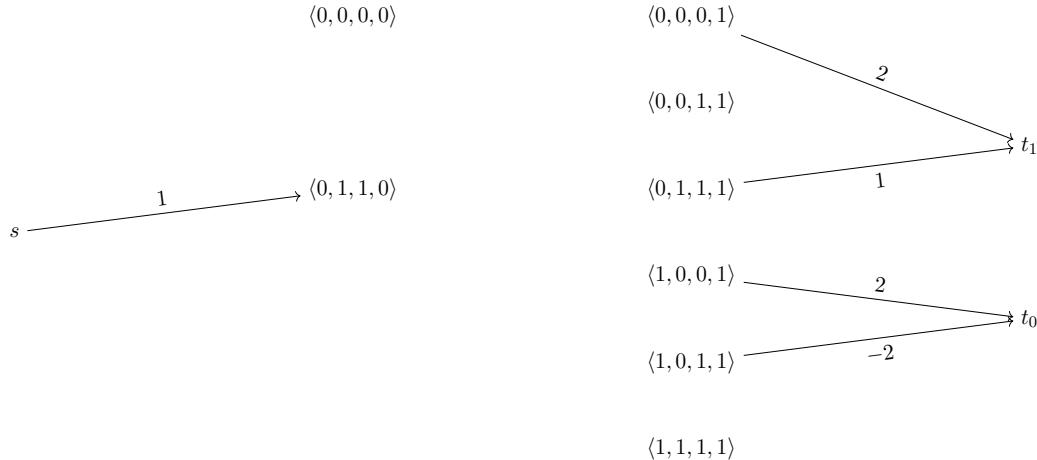
Le graphe H_A a les arcs suivants.

- (1) Un arc de poids 1 de s vers chaque sommet de la forme $\langle n \bmod 2, h, h, 0 \rangle$. Ces arcs permettent d'initialiser les suites de marches fermées.
- (2) Pour chaque coefficient $A_{u,v} \neq 0$, chaque $h < v$ et chaque $i < n-1$, un arc de poids $A_{u,v}$ de $\langle p, h, u, i \rangle$ vers $\langle p, h, v, i+1 \rangle$. Ces arcs correspondent à l'extension de la marche fermée en cours de construction (la marche C_{k+1} dans l'explication ci-dessus).
- (3) Pour chaque coefficient $A_{u,h} \neq 0$, chaque $h' \in \{h+1, \dots, n-1\}$, et chaque $i < n-1$, un arc de poids $A_{u,h}$ de $\langle p, h, u, i \rangle$ vers $\langle 1-p, h', h', i+1 \rangle$. Ces arcs correspondent à la fermeture de la marche fermée en cours de construction, et à l'initialisation d'une nouvelle marche fermée de tête h' .
- (4) Pour chaque coefficient $A_{u,h} \neq 0$ et chaque $p \in \{0, 1\}$, un arc de poids $A_{u,h}$ de $\langle p, h, u, n-1 \rangle$ vers t_{1-p} . Ces arcs correspondent à la fermeture de la dernière marche fermée de la suite.

Question 26. On considère la matrice A suivante :

$$\begin{pmatrix} 2 & 3 \\ -2 & 1 \end{pmatrix}$$

La figure ci-dessous représente certains sommets du graphe H_A correspondant, ainsi que certains arcs. Indiquer les arcs manquants entre les sommets représentés, ainsi que leurs poids.



Question 27. Soit A une matrice carrée avec n lignes et n colonnes. Montrer que si $C = h, u_1, \dots, u_m, h$ est une marche fermée dans $G(A)$, alors

$$\langle p, h, h, i \rangle \rightarrow \langle p, h, u_1, i + 1 \rangle \rightarrow \dots \rightarrow \langle p, h, u_m, i + m \rangle$$

est un chemin dans H_A pour tout $p \in \{0, 1\}$ et tout $i \in \{0, \dots, n - 1 - m\}$.

En déduire qu'à chaque suite de marches fermées positive dans $G(A)$ correspond un chemin de s à t_0 dans H_A .

On peut démontrer que les chemins de s à t_0 dans H_A sont en fait en bijection avec les suites de marches fermées positives dans $G(A)$. Dans toute la suite, on admet ce résultat ainsi que le résultat analogue pour les chemins de s à t_1 dans H_A et les suites de marches fermées négatives dans $G(A)$.

Nous ne construirons pas le graphe H_A mais travaillerons de manière implicite avec celui-ci. Afin de calculer le déterminant de la matrice A en utilisant la formule de Mahajan-Vinay, nous allons calculer la somme des poids des chemins de s vers t_0 et t_1 dans H_A . Soit F_A la fonction qui à chaque sommet x de H_A associe la somme des poids des chemins de s à x dans H_A .

Nous allons commencer par calculer la valeur de $F_A(x)$ pour l'ensemble des sommets de la forme $x = \langle p, h, v, i \rangle$. On remarque que les sommets du graphe H_A peuvent être organisés en couches selon la valeur du dernier entier. Formellement, la **couche** i de H_A est l'ensemble des sommets de H_A de la forme $\langle p, h, v, i \rangle$.

Question 28. Exprimer la valeur de $F_A(\langle p, h, v, i \rangle)$ en fonction des valeurs de la couche $i - 1$, c'est-à-dire des $F_A(\langle p', h', v', i - 1 \rangle)$.

Implémentation

On veut représenter $F_A(x)$ pour chaque x de la forme $\langle p, h, v, i \rangle$. Pour cela, on suppose donnés les types et fonctions C suivants.

- Un type **Fourtable**.
- Une fonction **Fourtable *create (int n)**. Un appel `create(n)` alloue dans le tas une valeur de type **Fourtable** et renvoie un pointeur **t** vers celle-ci. La valeur ***t** ainsi créée a la taille nécessaire pour enregistrer les valeurs de F_A lorsque A est une matrice $n \times n$. Les valeurs contenues dans ***t** sont toutes initialisées à 0. On supposera que `free(t)` a une complexité en $O(1)$.
- Une fonction **void write(Fourtable *t, int val, int p, int h, int v, int i)** qui enregistre dans ***t** l'entier **val** comme valeur correspondant au sommet $\langle p, h, v, i \rangle$.
- Une fonction **int read(Fourtable *t, int p, int h, int v, int i)** qui renvoie la valeur correspondant au sommet $\langle p, h, v, i \rangle$ dans ***t**.

On suppose de plus que les fonctions **read** et **write** ont une complexité en $O(1)$, et que la fonction **create** a une complexité en $O(n^3)$, où **n** est la valeur donnée en argument.

Notons que $F_A(\langle p, h, u, 0 \rangle)$ ne dépend que de p, h, u et de la dimension de la matrice A .

Question 29. Écrire une fonction **Fourtable *initialise (int n)** telle que **initialise(n)** renvoie un pointeur vers un **Fourtable** contenant les valeurs de F_A pour tous les sommets de la couche 0 de H_A , où A est une matrice $n \times n$.

On attend une complexité en $O(n^3)$ pour cette fonction, sans la justifier.

Question 30. Écrire une fonction **Fourtable *remplit (int n, int *A)** qui prend en entrée la représentation linéarisée A d'une matrice A de dimension $n \times n$, et renvoie un pointeur vers un **Fourtable** contenant les valeurs de F_A pour tous les sommets de la forme $\langle p, h, v, i \rangle$ avec $p \in \{0, 1\}$, $0 \leq h \leq v \leq n - 1$, et $0 \leq i \leq n - 1$.

On attend une complexité $O(n^4)$ pour cette fonction, qu'il faudra justifier.

Question 31. Écrire une fonction **int determinant (int n, int *A)** qui prend en entrée la représentation linéarisée A d'une matrice A de dimension $n \times n$ et renvoie son déterminant. On veillera à libérer correctement la mémoire allouée dans le tas.

On attend une complexité en $O(n^4)$ pour cette fonction, qu'il faudra justifier.

Partie III : Méthode algébrique et probabiliste pour les couplages parfaits

Cette partie concerne un algorithme qui teste si un graphe possède un couplage parfait, c'est-à-dire un couplage qui couvre tous ses sommets. Elle est à composer dans le langage C. On pourra réutiliser le matériel de la partie précédente.

On ne considère que des graphes **non orientés** (voir page 3). Un tel graphe G d'ensemble de sommets $\{0, \dots, n-1\}$ est représenté par sa **matrice d'adjacence** A , de dimension $n \times n$, et dont les coefficients sont donnés par

$$A_{i,j} = \begin{cases} 1 & \text{si } \{i, j\} \text{ est une arête de } G, \\ 0 & \text{si } \{i, j\} \text{ n'est pas une arête de } G. \end{cases}$$

Nous allons utiliser des méthodes algébriques, basées sur le déterminant de la matrice de Tutte. La matrice de Tutte d'un graphe est une matrice symbolique, et son déterminant est un polynôme réel à plusieurs variables. Un polynôme réel à m variables X_1, \dots, X_m est une somme finie de monômes, c'est-à-dire de termes de la forme $aX_1^{d_1} \cdots X_m^{d_m}$ avec $a \in \mathbb{R}$ et $d_1, \dots, d_m \in \mathbb{N}$. Lorsque $d_i = 0$, la variable X_i est omise dans $aX_1^{d_1} \cdots X_m^{d_m}$. Par exemple, le monôme $aX_1^0 \cdots X_m^0$ est noté a .

Considérons un graphe non orienté G dont l'ensemble de sommets est $\{0, \dots, n-1\}$. La **matrice de Tutte** $T(G)$ de G est une matrice $n \times n$ dont les éléments sont des monômes sur l'ensemble de variables $\{X_{i,j} \mid 0 \leq i < j \leq n-1\}$. Les coefficients de $T(G)$ sont donnés par

$$T(G)_{i,j} = \begin{cases} 0 & \text{si } i = j \text{ ou si } \{i, j\} \text{ n'est pas une arête de } G, \\ X_{i,j} & \text{si } i < j \text{ et } \{i, j\} \text{ est une arête de } G, \\ -X_{j,i} & \text{si } i > j \text{ et } \{i, j\} \text{ est une arête de } G. \end{cases}$$

Un théorème de Tutte énonce que G possède un couplage parfait si, et seulement si, le déterminant de $T(G)$ est différent du polynôme nul (c'est-à-dire du polynôme dont toutes les valeurs sont 0).

Une partie du problème consiste donc à décider si un polynôme est nul ou non. Nous allons implémenter un algorithme **probabiliste** qui teste si un polynôme s'annule en un certain nombre de points choisis aléatoirement.

Dans la suite, on suppose donnée une fonction `int rand_range(int n)` qui prend en entrée un entier `n` et renvoie un entier tiré aléatoirement entre 0 et `n` (inclus) selon la distribution uniforme. On suppose que cette fonction s'exécute en temps constant.

On suppose que l'instruction suivante, qui crée un tableau d'entiers `t` de longueur `m`, s'exécute en temps $O(m)$:

```
int t [m];
```

Question 32. Écrire une fonction `int tirage_tutte(int n, int *A)` qui prend en entrée la représentation linéarisée `A` d'une matrice d'adjacence `A` de dimension `n × n`, et renvoie le

déterminant d'une instantiation de la matrice de Tutte du graphe représenté par A obtenue en tirant aléatoirement les valeurs des variables $X_{i,j}$ dans l'ensemble $\{0, 1, 2, \dots, n*n - 1, n*n\}$. On attend une complexité en $O(n^4)$ pour cette fonction, sans la justifier.

On va utiliser la fonction définie ci-dessus pour décider si un polynôme réel à plusieurs variables est nul ou non. On se propose d'appliquer le lemme de Schwartz-Zippel. Ce résultat permet, pour un polynôme non nul P , de borner le nombre de zéros de P (dans un ensemble fini donné) en fonction du degré total de P (et de la taille de cet ensemble). Le degré total d'un monôme $aX_1^{d_1} \dots X_n^{d_n}$ est la somme $d_1 + \dots + d_n$. Le **degré total** d'un polynôme est le degré total maximal des monômes qui le composent.

Soit P un polynôme réel non nul à m variables et de degré total $d \geq 0$, et soit S un sous-ensemble fini non vide de \mathbb{R} . On note $\Pr[P(r_1, r_2, \dots, r_m) = 0]$ la probabilité pour que P s'annule en r_1, r_2, \dots, r_m , où r_1, r_2, \dots, r_m sont des réels tirés aléatoirement dans S , de manière uniforme et indépendante. Le lemme de Schwartz-Zippel énonce que

$$\Pr[P(r_1, r_2, \dots, r_m) = 0] \leq \frac{d}{|S|}.$$

On souhaite obtenir un algorithme probabiliste décidant si un graphe possède un couplage parfait. En s'aidant du lemme de Schwartz-Zippel, on pourra assurer la correction de cet algorithme et en borner l'erreur. On pourra utiliser la fonction `log2` définie lorsque l'entête `<math.h>` est incluse.

Question 33. Écrire une fonction

```
bool parfait (int n, int *A, int p)
```

qui prend en entrée la matrice d'adjacence linéarisée A d'un graphe G ayant n sommets, et décide si G possède un couplage parfait avec une probabilité d'erreur inférieure à 2^{-p} , qu'il faudra justifier.

On attend une complexité en $O(p \frac{n^4}{\log_2(n)})$, qu'il faudra justifier.

Fin du sujet.

A2025 – INFO I MPI

**ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2025**PREMIÈRE ÉPREUVE D'INFORMATIQUE**

Durée de l'épreuve : 3 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

*Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :*

INFORMATIQUE I - MPI

Cette épreuve concerne uniquement les candidats de la filière MPI.

L'énoncé de cette épreuve comporte 13 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines-Ponts.



Étude des tableaux associatifs

Préliminaires

L'épreuve est composée d'un problème unique, comportant 34 questions. Le problème consiste en l'étude et l'implémentation d'un tableau associatif en C, suivant différentes méthodes.

Le problème est divisé en 4 sections distinctes :

- l'implémentation du tableau associatif en utilisant une structure d'arbre (section 1),
- l'ajout d'un « ramasse miettes » pour gérer la mémoire (section 2),
- l'implémentation du tableau associatif en utilisant un algorithme probabiliste (section 3),
- la conception d'un analyseur syntaxique pour créer un tableau associatif à partir d'une chaîne de caractères (section 4).

Les quatre sections sont indépendantes.

Travail attendu

Les résultats attendus pour certaines questions pourront être admis afin de faciliter la résolution des questions suivantes.

Le langage C constitue l'unique langage autorisé pour le traitement des questions de programmation.

Une attention particulière sera portée quant à la correction et la lisibilité du code proposé. Si une tolérance raisonnable peut être envisagée sur certains aspects syntaxiques dans le cadre d'une épreuve réalisée sur support papier (par exemple, l'oubli d'un « ; »), un code difficilement lisible ou l'emploi de fonctions ou structures de contrôle non conformes aux standards du langage C, notamment celles empruntées à d'autres langages, sera sanctionné.

Un soin particulier sera également apporté à la gestion de la mémoire, à la bonne initialisation des structures ainsi qu'au contrôle systématique des allocations mémoires.

La gestion rigoureuse des erreurs sera valorisée (l'utilisation d'assert est recommandée), de même que la qualité, la simplicité et la lisibilité du code.

En conformité avec le programme officiel, il sera admis que certains en-têtes standards soient présupposés comme inclus dans le code : `<assert.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdio.h>`, `<stdlib.h>`, et `<string.h>`.

Les tableaux associatifs

Les tableaux associatifs sont des structures de données permettant un accès rapide à une *valeur* à partir d'une *clef*.

Afin de simplifier l'écriture du code, nous considérerons que les clefs sont toujours de type `char*` et les valeurs de type `int`.

1. Implémentation par un arbre binaire de recherche

Le but de cette section est d'étudier une implémentation d'un tableau associatif reposant sur un arbre binaire de recherche.

Nous adopterons une approche fonctionnelle pour l'implémentation des structures de données, en les considérant comme immuables. Ainsi, chaque fonction de manipulation renverra systématiquement une nouvelle version modifiée de la structure, sans altérer l'originale.

Cette méthode présente l'avantage d'éliminer les effets de bord lors des appels de fonction. Toutefois, elle implique une augmentation du nombre d'allocations mémoire, puisque chaque modification nécessite la création d'une copie de la structure de données.

Dans cette section nous ne nous occuperons pas de la récupération de la mémoire, car nous introduirons en section 2 un « ramasse miettes » pour gérer la mémoire automatiquement.

Arbre binaire de recherche

Définitions : Soit A un arbre binaire de recherche. On notera avec une lettre minuscule (par exemple x ou y) un noeud de A , et avec une lettre majuscule (par exemple T_1 , T_2 ou T_3) un sous-arbre de A . Soit x un noeud, on notera x_{gauche} et x_{droit} , respectivement le sous-arbre gauche et le sous-arbre droit de x .

On rappelle que la taille d'un arbre est égale au nombre de noeuds qu'il contient.

Pour un arbre de racine x , on notera $h(x)$ sa hauteur, et on définit récursivement la hauteur d'un arbre par :

- la hauteur d'un arbre vide est -1 ,
- $h(x) = \max(h(x_{\text{gauche}}), h(x_{\text{droit}})) + 1$.

Pour un arbre de racine x , on définit la valeur d'équilibre par :

$$e(x) = h(x_{\text{gauche}}) - h(x_{\text{droit}})$$

On qualifiera d'*équilibré* un arbre binaire de recherche A si pour tout noeud x de A ,

$$-1 \leq e(x) \leq 1$$

Définition : On définit un noeud d'un arbre binaire de recherche par la structure suivante :

```

1. struct abrNoeud_s {
2.     char* clef;
3.     int valeur;
4.
5.     struct abrNoeud_s* fils_gauche;
6.     struct abrNoeud_s* fils_droit;
7. };
8. typedef struct abrNoeud_s abrNoeud;
```

On utilisera la valeur du champ `clef` pour ordonner les noeuds de l'arbre, en utilisant l'ordre alphabétique.

- 1 – Nous nous intéressons aux valeurs du champ `clef` de la structure ci-dessus. Expliquer comment une chaîne de caractères est représentée en mémoire en C ? Combien d'octets sont nécessaires pour représenter la chaîne "arbre" par exemple ?

Première épreuve d'informatique MPI 2025

2 – Sachant que le code ASCII du caractère ‘a’ est 97, quelle est sa représentation en binaire ? Quelle est sa représentation en hexadécimal ?

3 – Écrire une fonction C `copie_chaine` qui renvoie une copie de la chaîne de caractères passée en paramètre. La signature de la fonction est :

```
char* copie_chaine(char* chaine)
```

Indication C : On considérera que la chaîne de caractères passée en paramètre (`chaine`) n'est jamais NULL.

4 – Écrire une fonction C `abr_creer_noeud` qui crée un noeud à partir d'une clef, d'une valeur et de 2 sous-arbres donnés, et qui renvoie ce nouveau noeud créé. La signature de la fonction est :

```
abrNoeud* abr_creer_noeud(char* clef, int valeur,
                            abrNoeud* fils_gauche,
                            abrNoeud* fils_droit)
```

Indication C : le champ `clef` du noeud créé doit contenir une copie de la chaîne de caractères passée en paramètre. Par contre, il n'est pas nécessaire de dupliquer les sous-arbres passés en paramètre.

5 – Écrire une fonction C `abr_equilibre` qui renvoie la valeur d'équilibre de l'arbre dont le noeud racine est passé en paramètre. La signature de la fonction est :

```
int abr_equilibre(abrNoeud* noeud)
```

6 – Pour un arbre de taille N , quelle est la complexité de la fonction `abr_equilibre` ? Sans proposer de code, expliquer quelles seraient les modifications à apporter à la structure de donnée et/ou à l'algorithme pour améliorer cette complexité ?

7 – Soit A un arbre binaire de recherche équilibré et de hauteur h , quelle approximation peut-on faire du nombre minimal N de noeuds que peut contenir cet arbre ?

8 – En déduire une borne supérieure de la hauteur h d'un arbre binaire de recherche équilibré de taille N .

9 – Écrire une fonction C récursive `abr_rechercher` qui, étant donné un arbre dont le noeud racine est passé en paramètre ainsi qu'une clef, renvoie la valeur associée à cette dernière. La signature de la fonction est :

```
int abr_rechercher(abrNoeud* noeud, char* clef)
```

Indication : On considérera que la clef donnée est toujours dans l'arbre binaire de recherche.

Indication C : l'opérateur `==` permet de comparer des entiers ou des pointeurs mais pas de comparer le contenu de deux chaînes de caractères.

Première épreuve d'informatique MPI 2025

- 10 – En application de la question 8, quelle est la complexité de la fonction abr_rechercher pour un arbre équilibré de taille N ?
- 11 – Quel est l'intérêt d'avoir un arbre binaire de recherche qui soit tout le temps équilibré ?

2. Ajout d'un ramasse miettes

Un « ramasse miettes » est un ensemble de fonctions qui permet de gérer la mémoire de manière automatique. L'idée est de libérer la mémoire qui avait été allouée dynamiquement pour des structures ou des tableaux lorsque ces derniers ne sont plus utilisés.

Dans un contexte fonctionnel où les données sont immuables, il est parfois compliqué de déterminer le moment opportun pour libérer la mémoire allouée aux nœuds. En effet, l'immuabilité des structures permet de partager le même sous-arbre à plusieurs endroits d'un arbre, sans indication explicite de ce partage, rendant ainsi difficile de savoir si un nœud doit être libéré ou non. L'utilisation d'un ramasse miettes dans un tel cas est donc tout indiquée. C'est ce que nous allons étudier dans cette section.

Pour simplifier le problème, nous considérons qu'une structure ou un tableau devient inutilisé dès lors qu'il n'est plus atteignable à partir des variables globales du programme. Autrement dit, en partant des variables globales, il est impossible d'y accéder en suivant des pointeurs, même de manière transitive.

Pour pouvoir décrire l'algorithme de ramasse miettes qui nous intéresse, nous avons besoin de considérer qu'un nœud peut être « marqué ». Cela consiste à mémoriser une information dans la structure elle-même. Dans notre situation, nous utiliserons la structure suivante :

```

1. struct abrNoeud_s {
2.     char* clef;
3.     int valeur;
4.
5.     struct abrNoeud_s* fils_gauche;
6.     struct abrNoeud_s* fils_droit;
7.
8.     // partie propre au ramasse-miettes
9.     bool marque;
10.    };
11.    typedef struct abrNoeud_s abrNoeud;

```

Un nœud est alors dit « marqué » si et seulement si le champ `marque` a pour valeur `true`. Nous considérons qu'initialement aucun nœud n'est marqué. L'algorithme de ramasse miettes fonctionne en deux temps :

1. Marquer tous les nœuds atteignables par clôture transitive depuis les variables globales. C'est-à-dire tous les nœuds atteignables directement ou bien atteignables à partir d'un nœud lui-même atteignable.
2. Libérer tous les nœuds alloués mais non marqués et effacer la marque des nœuds marqués.

 Première épreuve d'informatique MPI 2025

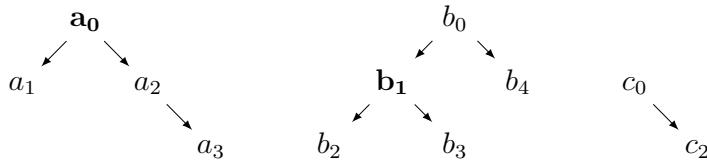


FIGURE 1 – Exemples d'arbres binaires de recherche, où les a_i , b_i , c_i sont des noeuds.

- 12 – On considère que *seuls* a_0 et b_1 sont directement référencés par des variables globales (Figure 1). Quels sont les noeuds marqués par l'étape 1 de l'algorithme ? Quels noeuds seront libérés par le ramasse miettes ?

En déduire (sans démonstration) la complexité en temps de l'algorithme de marquage et de l'algorithme de libération.

Pour pouvoir libérer tous les noeuds non marqués il est nécessaire de maintenir une liste des noeuds de l'arbre alloués en mémoire. Nous proposons d'utiliser la structure de données suivante :

```

1.  struct rmListeMemoire_s {
2.      abrNoeud* element;
3.      struct rmListeMemoire_s* suivant;
4.  };
5.  typedef struct rmListeMemoire_s rmListeMemoire;
  
```

- 13 – Écrire une fonction C permettant d'insérer, en temps constant, un noeud d'arbre binaire de recherche dans une liste. La signature de la fonction est :

```
rmListeMemoire* rm_inserer_tete(abrNoeud* element,
                                 rmListeMemoire* liste)
```

Indication : la donnée référencée par liste n'a pas besoin d'être modifiée.

Nous considérons maintenant la structure de donnée RM permettant de mémoriser dans le champ arbres les noeuds racines des différents arbres binaires de recherche accessibles (dans notre cas il peut y avoir jusqu'à 16 noeuds racines) ainsi que la liste des noeuds alloués :

```

1. #define MAX_NB_ARBRES 16
2. struct RM_s {
3.     abrNoeud* arbres[MAX_NB_ARBRES]; /* arbres accessibles */
4.     int nb_elements; /* nombre total de noeuds alloués */
5.     rmListeMemoire* premierElement;
6. };
7. typedef struct RM_s RM;
8.
9. RM* rm; /* variable globale pour l'ensemble du programme */
  
```

Le champ premierElement est une référence vers le premier élément d'une liste. Cette dernière est gérée de la façon suivante :

- lorsqu'un noeud d'un arbre binaire de recherche est alloué, il est ajouté à la liste,

Première épreuve d'informatique MPI 2025

— lorsqu'un nœud d'un arbre binaire de recherche est libéré, il est retiré de la liste.

- 14 – La fonction `main` commence par l'instruction `rm = rm_creer_rm();`
 Écrire le code de la fonction `rm_creer_rm` permettant d'allouer et d'initialiser la structure de données qui est affectée à la variable `rm`. La signature de la fonction est :

```
RM* rm_creer_rm(void)
```

Le code suivant correspond aux deux étapes de l'algorithme de ramasse miettes :

```
1. void rm_ramasse_miettes(RM* rm) {
2.     assert(rm != NULL);
3.     int nb_noeuds = rm->nb_elements;
4.     for (int i = 0; i < MAX_NB_ARBRES; i++) {
5.         if (rm->arbres[i] != NULL) {
6.             rm_marquer_elements_accessiblees(rm->arbres[i]);
7.         }
8.     }
9.     rm_recuperer_elements_inaccessiblees(rm);
10.    printf("noeuds_collectes:%d,%d", nb_noeuds - rm->nb_elements);
11.    printf("noeuds_<encore>_accessiblees:%d\n", rm->nb_elements);
12. }
```

- 15 – Écrire une fonction C correspondant à la première étape de l'algorithme : marquer un nœud donné ainsi que tous les noeuds atteignables à partir de ce dernier. La signature de la fonction est :

```
void rm_marquer_elements_accessiblees(abrNoeud* element)
```

Première épreuve d'informatique MPI 2025

Pour libérer tous les nœuds non marqués on considère la fonction suivante :

```

1. void rm_recuperer_elements_inaccessibles(RM* rm) {
2.     assert(rm != NULL);
3.     rmListeMemoire* objet = rm->premierElement;
4.     rmListeMemoire* objets_conserves = NULL;
5.
6.     while (objet != NULL) {
7.         if (!objet->element->marque) {
8.             /* Ce noeud n'a pas été marqué,
9.                on le retire de la liste et on le libère */
10.            rmListeMemoire* a_detruire = objet;
11.            objet = a_detruire->suivant;
12.            // PARTIE A
13.            // ici il faut libérer la mémoire
14.            // FIN PARTIE A
15.            rm->nb_elements--;
16.        } else {
17.            // PARTIE B
18.            /* Ce noeud a été marqué,
19.               on enlève la marque et on passe au noeud suivant */
20.            objet->element->marque = false;
21.            rmListeMemoire* a_garder = objet;
22.            objet = a_garder->suivant;
23.            // FIN PARTIE B
24.        }
25.    }
26.    rm->premierElement = objets_conserves;
27. }
```

L'algorithme est le suivant :

- on parcourt tous les éléments de la liste premierElement,
- si le noeud n'est pas marqué, il faut libérer la mémoire et enlever le nœud de la liste,
- si le nœud est marqué il faut enlever la marque et le laisser dans la liste.

16 – Compléter le code de la partie A de la fonction ci-dessus pour libérer la mémoire lorsque le noeud est marqué.

Indication : Seul le code entre *// PARTIE A* et *// FIN PARTIE A* est à compléter. Il n'est pas nécessaire de recopier le code autour.

17 – Quelle spécificité ont les nœuds de la liste référencée par le champ premierElement à la fin de l'exécution de la fonction ?

18 – Expliquer pourquoi le code proposé dans la partie B de la fonction ci-dessus n'est pas correct, et proposez une correction.

Indication : Seul le code entre *// PARTIE B* et *// FIN PARTIE B* est à compléter. Il n'est pas nécessaire de recopier le code autour.

3. Implémentation par l'utilisation d'un algorithme probabiliste

Nous allons nous intéresser dans cette section à une autre manière d'implémenter un tableau associatif, en utilisant un algorithme probabiliste : les « *skip lists* ».

3.1. Principe des skip lists

Une skip list est une liste chaînée optimisée pour accélérer la recherche d'un élément. Pour ce faire, certains noeuds stockent dans un tableau des pointeurs vers d'autres noeuds situés plus loin dans la liste. Chaque case du tableau correspond à un « niveau » et un pointeur de niveau i mène vers un noeud qui possède lui aussi un pointeur de niveau i .

Le niveau 0 contient l'ensemble des éléments, classés par ordre alphabétique en fonction de leur clef. Les niveaux supérieurs regroupent un sous-ensemble des éléments, de façon à espacer progressivement les noeuds. Par exemple, idéalement, le niveau 1 contiendrait un noeud sur deux, le niveau 2 un noeud sur quatre, et ainsi de suite (voir Figure 2).

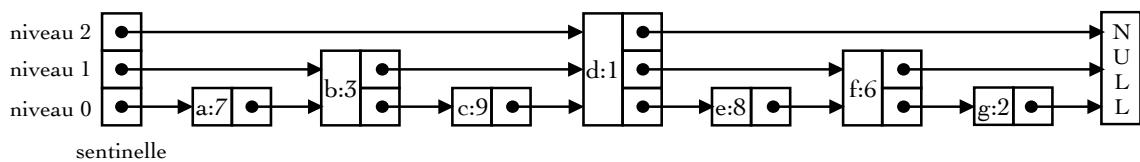


FIGURE 2 – Exemple d'une skip list idéale.

Le premier noeud de la skip list est une sentinelle : un élément spécial qui ne contient pas de donnée utile mais qui possède des pointeurs pour tous les niveaux de la liste.

Pour chercher un élément, on démarre à partir de la sentinelle au niveau le plus élevé. À ce niveau, on suit les pointeurs vers les noeuds suivants tant que la clef du noeud actuel est inférieure à celle recherchée. Dès qu'on rencontre un noeud dont la clef est supérieure, on descend d'un niveau et on répète la même opération. Lorsque la recherche atteint le niveau 0, elle est terminée et, si l'élément recherché y est présent, il est renvoyé (voir Figure 3).

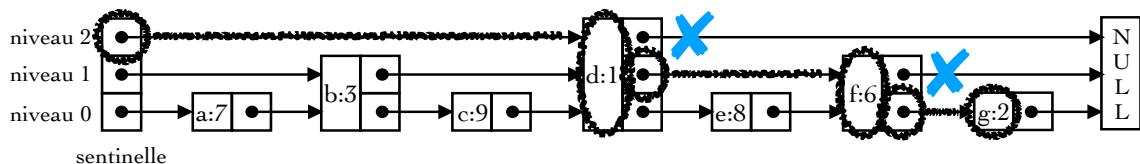


FIGURE 3 – Parcours lors de la recherche de la clef "g".

En pratique, chaque noeud a une probabilité de $\frac{1}{2}$ d'être présent dans le niveau supérieur.

3.2. Implémentation

Définition : On définit une skip list et un nœud d'une skip list par les structures suivantes :

```

1.  struct slNoeud_s {
2.      char* clef;
3.      int valeur;
4.
5.      struct slNoeud_s* suivant_par_niveau[];
6.  };
7.  typedef struct slNoeud_s slNoeud;
8.
9.  struct slListe_s {
10.     int niveau_actuel;
11.     slNoeud* sentinelle;
12. };
13. typedef struct slListe_s slListe;
```

On définit une constante, MAX_NIVEAU, qui correspond au nombre maximal de niveaux qu'un nœud peut avoir. La sentinelle de la skip list est créée comme un nœud particulier, avec une clef vide, une valeur nulle, et un tableau de pointeurs de taille MAX_NIVEAU+1 (de 0 à MAX_NIVEAU inclus) tous initialisés à NULL.

□ 19 – Écrire, en suivant l'algorithme décrit section 3.1, une fonction C sl_rechercher qui recherche, dans la skip list passée en paramètre, la clef donnée et qui renvoie :

- soit la valeur associée à la clef si elle est présente dans la skip list,
- soit -1 si la clef n'est pas présente dans la skip list.

La signature de la fonction est :

```
int sl_rechercher(slListe* liste, char* clef)
```

□ 20 – On rappelle que la fonction rand() renvoie un entier aléatoire (de type **int**) compris entre 0 et un certain entier supérieur à 32767.

Quelles sont les propriétés de la valeur renvoyée par la fonction suivante ?

```

1.  int sl_mystere() {
2.      int niveau = 0;
3.      while ((rand() % 2) == 1 && niveau < MAX_NIVEAU) {
4.          niveau++;
5.      }
6.      return niveau;
7. }
```

On suppose que la fonction suivante est disponible :

```
slNoeud* sl_creer_noeud(int niveau, char* clef, int valeur)
```

Cette fonction crée un nouveau nœud de niveau niveau avec la clef et la valeur passées en paramètre. Les pointeurs du nouveau nœud sont tous initialisés à NULL.

Première épreuve d'informatique MPI 2025

La fonction `sl_ajoute_valeur`, qui ajoute une clef et une valeur dans la skip list passée en paramètre, a la structure suivante :

```

1. void sl_ajoute_valeur(slListe* liste, char* clef, int valeur) {
2.     int niveau = sl_mystere();
3.     slNoeud* nouveau_noeud = sl_creer_noeud(niveau, clef, valeur);
4.
5.     slNoeud* noeuds_a_mettre_a_jour[MAX_NIVEAU + 1];
6.     slNoeud* courant = liste->sentinelle;
7.
8.     // PARTIE A
9.     /* parcourt l'ensemble des niveaux de la liste pour trouver
10.        dans chaque niveau le noeud après lequel insérer
11.        le nouveau noeud */
12.     // FIN PARTIE A
13.
14.    if (niveau > liste->niveau_actuel) {
15.        // PARTIE B
16.        /* si le niveau du nouveau noeud est plus grand
17.           que le niveau actuel de la liste
18.           préparer la sentinelle */
19.        // FIN PARTIE B
20.    }
21.
22.    // PARTIE C
23.    /* insérer le nouveau noeud */
24.    // FIN PARTIE C
25. }
```

21 – En vous basant sur les instructions du commentaire, écrire le code correspondant à la partie A de la fonction `sl_ajoute_valeur`.

Indication C : Les noeuds devant être mis à jour sont stockés dans le tableau `noeuds_a_mettre_a_jour`.

Indication : Seul le code entre `// PARTIE A` et `// FIN PARTIE A` est à compléter. Il n'est pas nécessaire de recopier le code autour.

22 – En vous basant sur les instructions du commentaire, écrire le code correspondant à la partie B de la fonction `sl_ajoute_valeur`.

Indication C : On réutilisera la variable `noeuds_a_mettre_a_jour`.

Indication : Seul le code entre `// PARTIE B` et `// FIN PARTIE B` est à compléter. Il n'est pas nécessaire de recopier le code autour.

23 – En vous basant sur les instructions du commentaire, écrire le code correspondant à la partie C de la fonction `sl_ajoute_valeur`.

Indication : Seul le code entre `// PARTIE C` et `// FIN PARTIE C` est à compléter. Il n'est pas nécessaire de recopier le code autour.

Première épreuve d'informatique MPI 2025

24 – L'algorithme des skip lists est un algorithme probabiliste. Est-ce un algorithme de type « Monte-Carlo » ou de type « Las Vegas » ? Justifier.

25 – En supposant une skip list idéale comme dans la Figure 2, où chaque niveau possède exactement la moitié des éléments du niveau inférieur idéalement répartis, quelle est la complexité en temps de la recherche d'un élément dans une skip list de taille n ?

On considère désormais une skip list non idéale, où les éléments sont répartis de manière aléatoire avec une probabilité $p = \frac{1}{2}$ d'être présent dans le niveau supérieur.

26 – Quelle est l'espérance du nombre d'éléments au niveau k ?

27 – Quelle est l'espérance du niveau le plus haut d'une skip list de taille n ?

On acceptera dans la suite le résultat suivant : cette espérance est proportionnelle à $\log_2(n)$.

28 – Quel est en moyenne le nombre de noeuds parcourus horizontalement dans un niveau k lors d'une recherche dans une skip list de taille n ?

29 – Des questions 27 et 28, déduisez la complexité en moyenne en temps de la recherche d'un élément dans une skip list de taille n .

4. Analyse syntaxique

On souhaite construire une fonction `analyse` qui prend en paramètre une chaîne de caractères décrivant un tableau associatif sous la forme de couples `clef:valeur`.

Un exemple d'utilisation pourrait être : `analyse("{id:5,valeur:12,ok:0}")`

Le format de la chaîne de caractères passée en paramètre est spécifié par les règles suivantes :

- la chaîne doit commencer par une accolade ouvrante, être suivie d'une liste de couples `clef:valeur`, et terminée par une accolade fermante,
- chaque couple est composé d'une clef et d'une valeur séparées par le caractère ‘:’,
- ni la clef ni la valeur ne peuvent être absentes,
- la clef est une suite finie non vide de lettres minuscules (de ‘a’ à ‘z’),
- la valeur est un entier non signé, sans 0 non significatif,
- les couples sont séparés par le caractère ‘,’.

Le but de cette partie est d'étudier la grammaire et l'automate de ce langage de description.

Première épreuve d'informatique MPI 2025

Définition : On utilise la notation $x \mid \dots \mid y$ pour indiquer la disjonction de symboles allant de x à y avec x et y des chiffres ou x et y des lettres.

$$N \rightarrow 0 \mid \dots \mid 9$$

On considère connues les règles de production suivantes : $M \rightarrow 1 \mid \dots \mid 9$

$$L \rightarrow a \mid \dots \mid z$$

30 – Écrire la règle de production C correspondant à la description d'une clef.

31 – Écrire la règle de production V correspondant à la description d'une valeur.

On considère la grammaire suivante :

$$T \rightarrow \{S\}$$

$$S \rightarrow K \mid K, S$$

$$K \rightarrow C : V$$

32 – Le mot `{id:5,valeur:12,ok:0}` appartient-il au langage engendré par cette grammaire ? Le cas échéant, dessinez un arbre de dérivation.

Même question pour le mot `{james:007}`.

33 – Décrire un automate (avec un schéma ou une table de transitions) déterministe et sans transition vide, correspondant au langage reconnu par la règle T . On s'autorisera à étiqueter les transitions avec des disjonctions de lettres ou de chiffres.

34 – Vérifiez en précisant les étapes de l'automate que les séquences `{code:102}`, `{v:0}`, et `{a:1,b:2}`, sont bien reconnues par l'automate proposé. Dans quel état est l'automate lorsqu'on lui demande de reconnaître la séquence `{james:007}` ?

A2025 – INFO II MPI

**ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2025**DEUXIÈME ÉPREUVE D'INFORMATIQUE**

Durée de l'épreuve : 4 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

*Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :*

INFORMATIQUE II - MPI

Cette épreuve concerne uniquement les candidats de la filière MPI.

L'énoncé de cette épreuve comporte 9 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines-Ponts.



Épreuve d'Informatique 2 MPI 2025

Le jeu de Shannon

Préliminaires

L'épreuve est composée d'un problème unique, comportant 33 questions.

Dans ce problème, nous nous intéressons au *jeu de Shannon*, qui est un jeu de stratégie dans lequel l'un des joueurs doit parvenir à établir une certaine connexion en posant des pièces de jeu, tandis que son opposant doit l'en empêcher en essayant de le bloquer. Le problème est divisé en trois sections. Dans la première section (page 1), nous introduisons les règles du jeu et une étude de tournois. Dans la deuxième section (page 5), nous analysons les stratégies de jeu conduisant l'un des joueurs à la victoire. Dans la troisième section (page 7), nous étudions la construction de deux arbres couvrants disjoints dans un graphe, ce qui est un prérequis de l'une des stratégies d'un joueur, vue dans la deuxième section.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères différentes désigne la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple n).

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question précédente, même sans avoir réussi à établir ce résultat.

Selon les consignes, il faudra coder des fonctions à l'aide du langage de programmation C exclusivement, en reprenant le prototype de fonction fourni par le sujet, ou en pseudo-code (c.-à-d. dans une syntaxe souple mais conforme aux possibilités offertes par le langage C). Inclure les entêtes tels que `<assert.h>`, `<stdbool.h>`, etc., n'est pas demandé.

Quand l'énoncé demande de coder une fonction, sauf indication explicite, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté et de la concision des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrira le rôle.

1 Le jeu de Shannon

1.1 Implémentation du jeu en langage C

Un graphe non orienté et connexe $G = (V, E)$ est fixé, ainsi que deux sommets particuliers : le sommet s et le sommet t . La lettre n désigne le nombre de sommets.

Définition : Le *jeu de Shannon* sur le graphe G est un jeu qui se joue entre deux joueurs, *Positus* et *Minus*, qui agissent à tour de rôle. Chaque coup consiste à marquer une arête qui n'a pas été marquée précédemment. Dans ce jeu, *Positus* remporte la victoire s'il parvient à marquer un ensemble d'arêtes dans lequel les sommets s et t sont reliés par un chemin (appelé aussi une *chaîne* d'arêtes). *Minus* gagne le jeu si toutes les arêtes ont été marquées sans que *Positus* ne soit parvenu à relier les sommets s et t .

Épreuve d’Informatique 2 MPI 2025

En pratique, *Positus* « verrouille » des arêtes tandis que *Minus* « supprime » des arêtes encore jouables.

Indication C : Dans l’ensemble du sujet, les sommets du graphe $G = (V, E)$ sont numérotés entre 0 et $n - 1$, $S = \{0, 1, 2, \dots, n - 1\}$. Par convention, les sommets s et t correspondent aux sommets de numéros 0 et 1. Les arêtes des graphes sont de la forme $\{u, v\}$ et donc E est un sous-ensemble de $P_2(S) = \{X \subset S \mid |X| = 2\}$. Nous notons $M \subseteq E$ l’ensemble des arêtes marquées. Nous introduisons les déclarations suivantes.

```

1. #define s 0
2. #define t 1
3.
4. struct link {
5.     int node;
6.     struct link * next;
7. };
8. typedef struct link link_t;
9.
10. struct game {
11.     int n;           // Nombre de sommets
12.     int nm;          // Nombre d'arêtes non marquées
13.     link_t ** adj_arr; // Voisins non marqués
14.     int * cmps;        // Composantes connexes créées par Positus
15. };
16. typedef struct game game_t;
17.
18. game_t gm;
```

Le type `link_t` permet de représenter des maillons de listes simplement chaînées. Le graphe $(V, E \setminus M)$ est représenté par un tableau de listes d’adjacences. Le type `game_t` permet de représenter un jeu de Shannon en cours de déroulement. Dans les enregistrements de ce type,

- le champ `n` désigne le nombre de sommets,
- le champ `nm` désigne le nombre d’arêtes non marquées,
- le champ `adj_arr` renvoie à un tableau de pointeurs où, pour tout entier u compris entre 0 et $n - 1$, le pointeur `adj_array[u]` désigne le premier maillon d’une liste simplement chaînée contenant tous les voisins du sommet u dans le graphe $(V, E \setminus M)$,
- le champ `cmps` renvoie à un tableau permettant de représenter les sommets reliés entre eux par des arêtes marquées par *Positus* selon une convention détaillée aux questions 6 et 7.

Une variable globale `gm` est utilisée dans tout le sujet pour désigner le jeu en cours. On notera que les arêtes marquées au cours du jeu sont perdues et ne figurent plus dans la variable `gm`.

- 1 – Écrire une fonction C `void gm_init(int n)` qui initialise la variable globale `gm` afin qu’elle contienne un jeu sur le graphe à n sommets et zéro arêtes. Il est attendu que le tableau `gm.cmps` contienne initialement les entiers $0, 1, \dots, n - 1$, dans cet ordre.

 Épreuve d’Informatique 2 MPI 2025

2 – Écrire une fonction C **void** gm_addege(**int** u, **int** v) dont l’effet est d’ajouter une arête entre les sommets u et v au graphe stocké dans la variable globale gm. On veillera à représenter une arête par deux arcs orientés (u, v) et (v, u) .

3 – Écrire une fonction C **bool** gm_remove(**int** u, **int** v) dont l’effet est de retirer une arête entre les sommets u et v au graphe gm et dont la valeur de retour est true si la suppression s’est bien déroulée, et false sinon.

Nous rappelons que la structure « unir & trouver » (ou, en anglais, *union find*) permet de représenter les classes d’équivalence d’une relation d’équivalence pour un ensemble E . On choisit d’implémenter cette structure à l’aide d’un tableau associatif A , de taille n pour un ensemble E de cardinal n dont les éléments sont numérotés de 0 à $n - 1$. Le tableau rend compte d’une forêt où chaque arbre est orienté des feuilles vers la racine et représente une classe d’équivalence. Pour un élément de numéro u , $A[u]$ vaut u s’il s’agit de la racine, que l’on appellera **identifiant** de la classe, ou $A[u]$ vaut le numéro de son parent dans l’arbre dans tout autre cas.

4 – Écrire une fonction C **int** uf_find(**int** n, **int** * A, **int** u) dont la valeur de retour est l’identifiant de la classe à laquelle appartient l’élément u dans la relation représentée par A . Le tableau A ne doit pas être modifié.

5 – Écrire une fonction C **bool** uf_unite(**int** n, **int** * A, **int** u, **int** v) dont l’effet est de réunir les classes des éléments u et v dans la relation d’équivalence représentée par A . Il n’y a pas d’effet si les éléments u et v appartiennent déjà à la même classe. La valeur de retour est true si A a été modifié et false sinon.

Nous appelons $M^+ \subseteq E$ l’ensemble des arêtes marquées par le joueur *Positus*. Nous notons \rightsquigarrow_P la relation d’équivalence dans laquelle deux sommets u et v sont en relation si et seulement s’il existe une chaîne entre u et v formée d’arêtes de M^+ . Les classes d’équivalence sont ainsi les composantes connexes du graphe (V, M^+) . Nous utilisons le champ cmps de la variable gm afin de stocker cette relation d’équivalence selon une structure « unir & trouver ».

6 – Écrire une fonction C **int** gm_find(**int** u) dont la valeur de retour est l’identifiant de la composante connexe à laquelle appartient le sommet u dans le graphe (V, M^+) .

7 – Écrire une fonction C **void** gm_unite(**int** u, **int** v) dont l’effet est de réunir les composantes connexes des sommets u et v .

8 – Écrire une fonction C **bool** has_P_won(**void**) dont la valeur de retour indique si le jeu est dans un état gagnant pour le joueur *Positus*.

9 – Écrire une fonction C **bool** has_P_lost(**void**) dont la valeur de retour indique si le jeu est dans un état perdant pour le joueur *Positus*.

Épreuve d'Informatique 2 MPI 2025

- 10 – Écrire une fonction C `void play_P(int u, int v)` dont l'effet est de transformer la variable `gm` en jouant l'arête entre les sommets u et v par le joueur *Positus*. On se défendra, à l'aide d'une assertion, du cas où le coup joué par *Positus* est illicite.

- 11 – Écrire une fonction C `void play_M(int u, int v)` dont l'effet est de transformer la variable `gm` en jouant l'arête entre les sommets u et v par le joueur *Minus*. On se défendra, à l'aide d'une assertion, du cas où le coup joué par *Minus* est illicite.

1.2 Étude de tournois en langage SQL

Un *tournoi* de jeux de Shannon est une compétition dans laquelle s'affrontent exactement deux joueurs (A et B) au cours de plusieurs matchs. Le vainqueur d'un tournoi est le joueur qui a gagné le plus grand nombre de matchs. Le rôle de chaque joueur (*Positus* ou *Minus*) peut changer entre chaque match.

On dispose d'une base de donnée SQL composée de trois tables (les clés primaires sont soulignées)

- Joueurs(JoueurID, NomJ)
- Tournois(TournoiID, NomT, JoueurAID, JoueurBID)
- Matchs(MatchID, TournoiID, RoleJoueurA, VainqueurID)

dont voici certains extraits :

JoueurID	NomJ	TournoiID	NomT	JoueurAID	JoueurBID
2	Alice	82	TournoiDu12Dec	2	8
8	Bob	59	TournoiDu17Fev	7	8
7	Charlie	61	TournoiDu29Mai	8	2

MatchID	TournoiID	RoleJoueurA	VainqueurID
119	82	Positus	2
934	82	Minus	8
925	82	Minus	2
384	61	Positus	2

Par exemple, le tournoi du 12 décembre a opposé Alice et Bob, il s'est joué en 3 matchs. Alice, qui a gagné deux des trois matchs, a été victorieuse.

- 12 – Écrire une requête SQL renvoyant le nombre de matchs joués dans chaque tournoi.

- 13 – Écrire une requête SQL renvoyant le nombre de tournois gagnés par chaque joueur.

- 14 – Écrire une requête SQL renvoyant le nombre de fois où chaque joueur a joué le rôle de *Positus*.

Épreuve d’Informatique 2 MPI 2025

2 Stratégies de jeu

2.1 Stratégies gagnantes

- 15 – Démontrer, à l'aide d'un variant, que le jeu se termine toujours et justifier qu'il ne peut pas se terminer par un match nul.
- 16 – Rappeler la définition du terme *stratégie gagnante*.
- 17 – Nous supposons que *Positus* dispose d'une stratégie gagnante s'il joue en second. Démontrer qu'il dispose aussi d'une stratégie gagnante s'il joue en premier.
- 18 – Démontrer qu'il existe trois types d'instances du jeu de Shannon :
 - un type d'instance dans laquelle *Positus* dispose d'une stratégie gagnante qu'il soit premier ou second joueur,
 - un type dans laquelle *Minus* dispose d'une stratégie gagnante qu'il soit premier ou second joueur,
 - et enfin un type dans laquelle le joueur qui commence la partie dispose d'une stratégie gagnante.

On donnera un exemple de graphe G illustrant chacun des trois types.

2.2 Une stratégie pour *Positus*

Définition : Selon la terminologie usuelle, nous appelons *arbre couvrant* tout sous-ensemble de $n - 1$ arêtes $T \subseteq E$ ne contenant pas de cycle. Il est rappelé les faits suivants :

- le graphe (V, T) est connexe ;
- tout ajout d'une arête $e \in E \setminus T$, à un arbre couvrant T , crée un unique cycle noté $C(T, e)$;
- si T est un arbre couvrant, alors, pour tout couple de sommets u et v de V , il existe un unique chemin simple reliant u et v avec des arêtes de T .

- 19 – Soient deux arbres couvrants $T \subseteq E$ et $T' \subseteq E$ et une arête α appartenant à T . Montrer qu'il existe une arête β appartenant à T' telle que l'ensemble $T'' = T \cup \{\beta\} \setminus \{\alpha\}$ est encore un arbre couvrant.

Indication C : Nous représentons tout arbre couvrant par un tableau T de taille n tel que $T[0]$ vaut 0 et, pour tout autre sommet u , $T[u]$ vaut le parent du sommet u dans l'arbre enraciné en 0.

Pour représenter un arc, nous déclarons :

Épreuve d'Informatique 2 MPI 2025

```

19. struct pair {
20.     int first;
21.     int second;
22. };
23. typedef struct pair pair_t;

```

- 20 – Écrire une fonction C **pair_t** **edge_find**(**int** n, **int** * T, **int** * Tprime, **pair_t** alpha ainsi spécifiée :

Précondition : Les variables T et Tprime contiennent des arbres couvrants T et T' sur les mêmes n sommets.

Valeur de retour : Une arête β , appartenant à l'arbre T' , telle que $T'' = T \cup \{\beta\} \setminus \{\alpha\}$ est encore un arbre couvrant.

- 21 – Écrire une fonction C **void** **edge_swap**(**int** n, **int** * T, **pair_t** alpha, **pair_t** beta) ainsi spécifiée :

Précondition : La variable T correspond à un arbre couvrant T à n sommets. La variable α est un arc de l'arbre T orienté des feuilles vers la racine. La variable β est une arête qui n'appartient pas à l'arbre T telle que $T'' = T \cup \{\beta\} \setminus \{\alpha\}$ est un arbre couvrant.

Effet : Le tableau T est transformé afin de contenir l'arbre T'' .

Dans ce qui suit, nous supposons que *Minus* joue en premier.

Définition : Pour tout ensemble de sommets $U \subseteq V$ qui contient les sommets s et t , nous notons G_U le *graphe induit par U*, c'est-à-dire le sous-graphe (U, E_U) où,

$$E_U = \{\{u, v\} \in E \mid u, v \in U\}.$$

Pour tout entier naturel k , nous notons $M_k^+ \subseteq E$ l'ensemble des k arêtes marquées par *Positus*, après que k coups ont été joués par *Positus*. De même, nous notons $M_k^- \subseteq E$ l'ensemble des k arêtes marquées par *Minus* après que k coups ont été joués par *Minus*. Enfin, nous appelons (\boxtimes_k) l'assertion suivante :

Après que $2k$ coups ont été joués par *Minus* et *Positus*, il existe un ensemble de sommets U contenant les sommets s et t ainsi que deux arbres couvrant le graphe induit G_U , que l'on note $T_k \subseteq E_U$ et $T'_k \subseteq E_U$, tels que l'on a $T_k \cap T'_k = M_k^+$ et $(T_k \cup T'_k) \cap M_k^- = \emptyset$. (\boxtimes_k)

- 22 – Soit k un entier naturel. Dans cette question, nous supposons que $2k$ coups ont été joués et déjà été joués et que l'assertion (\boxtimes_k) est vérifiée ; pour son $(k+1)^{\text{e}}$ coup, le joueur *Minus* choisit de marquer une arête appartenant à l'arbre T_k . Montrer que le joueur *Positus* peut alors trouver une arête à marquer telle que l'assertion (\boxtimes_{k+1}) est vérifiée.

- 23 – Soit k un entier naturel. Dans cette question, nous supposons que $2k$ coups ont été joués et que l'assertion (\boxtimes_k) est vérifiée ; le joueur *Minus* marque une arête n'appartenant ni à l'arbre T_k , ni à l'arbre T'_k . Montrer que le joueur *Positus* peut marquer une arête telle que l'assertion (\boxtimes_{k+1}) est vérifiée.

Épreuve d'Informatique 2 MPI 2025

- 24 – Démontrer que, si au cours de la partie, il existe un entier naturel k_0 tel que l'assertion (\boxplus_{k_0}) est satisfaite, alors *Positus* possède une stratégie gagnante.
- 25 – Dans cette question, nous supposons que la paire $\{s, t\}$ n'appartient pas à E et nous notons \hat{G} le graphe obtenu à partir de G en ajoutant une arête supplémentaire entre les sommets s et t . Nous supposons que l'assertion (\boxplus_0) est vraie pour le nouveau graphe \hat{G} . Démontrer que le joueur *Positus* dispose d'une stratégie gagnante lorsqu'il joue en premier.

3 Arbres couvrants disjoints

Dans toute cette section, nous supposons que la paire $\{s, t\}$ n'appartient pas à l'ensemble des arêtes E . Nous notons \hat{G} le graphe obtenu à partir de G en ajoutant une arête supplémentaire entre les sommets s et t . Nous appelons \hat{E} l'ensemble d'arêtes $E \cup \{s, t\}$.

3.1 Une condition suffisante pour (\boxplus_0)

Soient $T \subseteq \hat{E}$ et $T' \subseteq \hat{E}$ deux arbres couvrants de \hat{G} et α_0 une arête n'appartenant pas à $T \cup T'$. Notons $L_0 = \{\alpha_0\}$. Notons $L_1 \subseteq \hat{E}$ l'unique cycle $C(T, \alpha_0)$ présent dans l'ensemble $T \cup \{\alpha_0\}$.

Définition : Pour tout couple d'arbres couvrants, $T \subseteq \hat{E}$ et $T' \subseteq \hat{E}$, notons $\delta(T, T')$ le cardinal de l'intersection $T \cap T'$. Deux arbres couvrants T_0 et T'_0 sont *éloignés* si le couple (T_0, T'_0) atteint le minimum global de la fonction δ parmi tous les couples d'arbres couvrants de \hat{G} .

Définition : Une arête $\alpha \in \hat{E}$ est *principale* s'il existe un couple d'arbres couvrants T et T' éloignés, tel que l'arête α n'appartient pas à $T \cup T'$.

- 26 – En supposant qu'il existe une arête $\alpha_1 \in L_1$ appartenant à $T \cap T'$, construire deux arbres couvrants \tilde{T} et \tilde{T}' du graphe \hat{G} tels que

$$\delta(\tilde{T}, \tilde{T}') < \delta(T, T').$$

Si L_1 ne contient aucune arête de $T \cap T'$, nous appelons $L_2 \subseteq \hat{E}$ la réunion des cycles $\bigcup_{\alpha \in L_1} C(T', \alpha)$ qui se créent lorsque l'on ajoute à T' l'une des arêtes de L_1 .

- 27 – En supposant qu'il existe une arête $\alpha_2 \in L_2$ appartenant à $T \cap T'$, construire deux arbres couvrants \tilde{T} et \tilde{T}' du graphe \hat{G} tels que

$$\delta(\tilde{T}, \tilde{T}') < \delta(T, T').$$

Plus généralement, nous construisons par récurrence la suite croissante $(L_k)_{k \in \mathbb{N}} \subseteq \hat{E}$ d'ensembles d'arêtes où

Épreuve d’Informatique 2 MPI 2025

- pour tout entier $k \geq 1$, si l’ensemble L_{2k} ne contient aucune arête de $T \cap T'$, alors on pose

$$L_{2k+1} = \bigcup_{\alpha \in L_{2k}} C(T, \alpha),$$

- pour tout entier $k \geq 0$, si l’ensemble L_{2k+1} ne contient aucune arête de $T \cap T'$, alors on pose

$$L_{2k+2} = \bigcup_{\alpha \in L_{2k+1}} C(T', \alpha).$$

28 – Démontrer que l’un ou l’autre des cas suivants se produit :

- la suite $(L_k)_{k \in \mathbb{N}} \subseteq E^{\mathbb{N}}$ n’est définie que pour un nombre fini de termes et dans ce cas montrer qu’il existe deux arbres couvrants \tilde{T} et \tilde{T}' du graphe \hat{G} tels que

$$\delta(\tilde{T}, \tilde{T}') < \delta(T, T') ;$$

- ou bien la suite $(L_k)_{k \in \mathbb{N}} \subseteq E$ est définie pour tout rang k et donc qu’il existe un rang à partir duquel la suite est égale à une constante $\Lambda(\alpha_0) \subseteq \hat{E}$.

29 – Démontrer que, si l’arête $\{s, t\}$ est une arête principale du graphe \hat{G} , alors l’assertion (\boxtimes_0) est vérifiée.

3.2 Recherche des arêtes principales

30 – Nommer un algorithme qui construit un arbre couvrant pour le graphe G et qui soit cohérent avec les choix opérés par le type **game_t** de représenter un graphe par des listes d’adjacence.

Soit T et T' un couple d’arbres couvrants éloignés. On pose

$$P = \bigcup_{\alpha_0 \in E \setminus (T \cup T')} \Lambda(\alpha_0).$$

On appelle U l’ensemble des arêtes incidentes à P . On appelle n_U le cardinal de U , m_P le cardinal de P , et $c = |E \setminus (T \cup T')|$ le nombre d’arêtes hors de T et T' . On note κ le nombre de composantes connexes de (U, P) .

Soit \tilde{T} et \tilde{T}' un autre couple d’arbres couvrants éloignés.

Notons \tilde{c}_1 le cardinal de $(E \setminus (\tilde{T} \cup \tilde{T}')) \cap P$.

31 – Montrer que

$$c \geq \tilde{c}_1 \geq m_P - 2(n_U - \kappa).$$

En déduire que le cas 2 de la question 28 se produit pour toute arête α_0 uniquement quand T et T' sont éloignés.

32 – D’après la question 28, pour toute arête α_0 n’appartenant pas à $T \cup T'$, l’ensemble $\Lambda(\alpha_0) \subseteq \hat{E}$ est bien défini. Démontrer que P est l’ensemble des arêtes principales.

Épreuve d'Informatique 2 MPI 2025

- 33 – Expliquer comment on peut construire l'ensemble des arêtes principales et en déduire une stratégie gagnante pour un joueur. Une réponse en pseudo-code est permise, mais seule la logique de programmation sera évaluée.

FIN DE L'ÉPREUVE

A2025 – INFO MP

**ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2025**ÉPREUVE D'INFORMATIQUE MP**

Durée de l'épreuve : 3 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

*Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :*

INFORMATIQUE - MP

Cette épreuve concerne uniquement les candidats de la filière MP.

L'énoncé de cette épreuve comporte 10 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.



Épreuve d'informatique d'option MP 2025

Préliminaires

L'épreuve est formée d'un problème unique, constitué de 28 questions, et porte sur l'apprentissage automatique (ou *machine learning*) d'un langage régulier que l'on souhaite déterminer à partir de requêtes d'appartenance et de requêtes d'équivalence.

Les applications de l'apprentissage automatique d'un langage sont étendues. En analysant le modèle appris, il est possible de détecter les divergences entre une spécification et sa mise en œuvre ou entre différentes mises en œuvre tant pour des systèmes matériels que logiciels : par exemple, protocoles de cartes à puce, réseaux de zombies sur internet (ou *botnets*), logiciels patrimoniaux (ou *legacy software*) pour ne citer que quelques illustrations.

Le problème est divisé en quatre sections reliées entre elles. Une question peut être traitée à condition d'avoir lu les définitions introduites jusque là. Dans la première section (page 2), nous établissons quelques prolégomènes. Dans la deuxième section (page 4), nous nous intéressons à une relation d'équivalence induite par le langage à apprendre. Dans la troisième section (page 6), nous étudions un certain type d'arbre de décision. Dans la quatrième section (page 8), nous construisons un automate reconnaissant le langage à apprendre.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères différentes désigne la même entité mais du point de vue mathématique pour la police en italique (par exemple n , T , δ) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`, `t`, `delta`).

Des rappels des extraits du manuel de documentation de OCaml portant sur le module List sont reproduits en annexe (page 10).

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Il faudra coder des fonctions à l'aide du langage de programmation OCaml exclusivement, en reprenant l'en-tête de fonctions fourni par le sujet, sans s'obliger à recopier la déclaration des types. Il est permis d'utiliser la totalité du langage OCaml sauf indication contraire. Il est recommandé de s'en tenir aux fonctions les plus courantes afin de rester compréhensible. Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté et de la concision des programmes : il est attendu que l'on choisisse des noms de variables intelligibles ou encore que l'on structure de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

 Épreuve d'informatique d'option MP 2025

1. Alphabets, mots et automates

Dans l'ensemble du sujet, nous fixons un *alphabet* binaire $\Sigma = \{a, b\}$ muni de l'ordre alphabétique ; la notation ε désigne le *mot vide* ; la notation Σ^* désigne l'*ensemble des mots* sur l'alphabet Σ muni de l'ordre lexicographique induit. La *concaténation* de deux mots $u \in \Sigma^*$ et $v \in \Sigma^*$ est notée uv .

Indication OCaml : Nous adoptons les types suivants :

1. `type letter = A | B`
2. `type word = letter list`

- 1 – Écrire une fonction OCaml `cmp_letter (x1:letter) (x2:letter) : int` dont la valeur de retour est nulle si les lettres x_1 et x_2 sont égales, est strictement négative si la lettre x_1 précède strictement la lettre x_2 dans l'ordre alphabétique et est strictement positive sinon.
- 2 – Écrire une fonction OCaml `cmp_word (w1:word) (w2:word) : int` dont la valeur de retour est nulle si les mots w_1 et w_2 sont égaux, est strictement négative si le mot w_1 précède strictement le mot w_2 dans l'ordre lexicographique et est strictement positive sinon. On s'interdira l'usage de la fonction `compare` du module `List` et toute autre approche similaire.
- 3 – Décrire une structure de données immuables qui implémente un dictionnaire dont les clés sont des mots de Σ^* et dont les valeurs sont d'un type quelconque. Il est attendu un nom de structure classique, un type OCaml, une condition concernant l'ensemble des clés et un invariant de bonne constitution.

Indication OCaml : Dans la suite du sujet, nous munissons d'un module OCaml `WordMap` et d'un type `'a wordmap` qui implémentent des dictionnaires de clés de type `word` et de valeurs d'un type quelconque `'a`. Nous disposons de la constante et des deux fonctions suivantes :

- `WordMap.empty`, de type `'a wordmap`, qui désigne le dictionnaire vide.
- `WordMap.add`, de type `word -> 'a -> 'a wordmap -> 'a wordmap`, telle que la valeur de retour de `WordMap.add w v d` est un dictionnaire contenant les associations du dictionnaire d ainsi qu'une association supplémentaire entre la clé w et la valeur v .
- `WordMap.find`, de type `word -> 'a wordmap -> 'a`, telle que la valeur de retour de `WordMap.find w d` est la valeur associée à la clé w dans le dictionnaire d .

- 4 – À titre d'exemple, déclarer deux valeurs OCaml `delta_a` et `delta_b`, de type `word wordmap`, et une valeur OCaml `un_a`, de type `bool wordmap`, égales aux dictionnaires respectifs

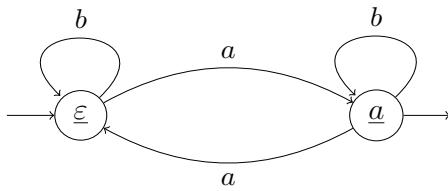
$$\delta_a : \begin{cases} \Sigma^* &\rightarrow \Sigma^* \\ \varepsilon &\mapsto a \\ a &\mapsto \varepsilon \end{cases}; \quad \delta_b : \begin{cases} \Sigma^* &\rightarrow \Sigma^* \\ \varepsilon &\mapsto \varepsilon \\ a &\mapsto a \end{cases} \text{ et } \mathbb{1}_{\{a\}} : \begin{cases} \Sigma^* &\rightarrow \{\text{faux, vrai}\} \\ \varepsilon &\mapsto \text{faux} \\ a &\mapsto \text{vrai} \end{cases}.$$

 Épreuve d'informatique d'option MP 2025

Définition : Le terme *automate* s'entend systématiquement dans le sens d'un automate fini déterministe complet, c'est-à-dire un quadruplet $(Q, q_0, \delta, \mathbb{1}_F)$ où

- l'ensemble Q est un ensemble fini d'états,
- l'état q_0 est un élément particulier de Q appelé l'état initial,
- l'application δ est une application $Q \times \Sigma \rightarrow Q$ appelée fonction de transition,
- et l'application $\mathbb{1}_F : Q \rightarrow \{\text{faux}, \text{vrai}\}$ est la fonction indicatrice d'une partie F de Q appelée ensemble des états finals.

5 – À titre d'exemple, décrire en langue française le langage reconnu par l'automate figuré ci-dessous puis énoncer une expression régulière qui le dénote. On ne demande pas de justification.



6 – Dire s'il existe une expression régulière admettant l'automate figuré à la question 5 comme automate de Glushkov associé.

Indication OCaml : Dans ce sujet, l'ensemble des états des automates est systématiquement choisi parmi les mots de Σ^* . Typographiquement, nous convenons de souligner les mots servant d'état dans ce sujet et adoptons les types OCaml suivants

```

3.   type state = word
4.   type automaton = { initial : state ;
5.                           transitions : state -> letter -> state ;
6.                           finals : state -> bool }
  
```

7 – À titre d'exemple, définir une valeur OCaml `automaton_example`, de type `automaton`, égale à l'automate figuré à la question 5.

On copiera et on complètera le code suivant :

```

1.   let automaton_example =
2.     {initial = .... ;
3.      transitions = ( fun q x -> WordMap.find q (.....)
4.                         );
5.      finals = ....
6.    }
  
```

Épreuve d'informatique d'option MP 2025

Définition : Pour tout automate $A = (Q, q_0, \delta, \mathbb{1}_F)$, pour tout état q de Q et pour tout mot w de Σ^* , nous notons $\delta^*(q, w)$ l'état d'arrivée dans l'automate A au terme du parcours démarrant en l'état q et suivant les transitions de A étiquetées par les lettres successives du mot w . Ainsi, la fonction δ^* est une application $Q \times \Sigma^* \rightarrow Q$ qui étend l'application δ .

8 – Écrire une fonction OCaml `delta_star` (`a:automaton`) (`q:state`) (`w:word`) : `state` dont la valeur de retour est $\delta^*(q, w)$.

9 – Écrire une fonction OCaml `accepts` (`a:automaton`) (`w:word`) : `bool` dont la valeur de retour est le booléen `vrai` si le mot w est reconnu par l'automate A et le booléen `faux` sinon. Il est demandé d'utiliser la fonction de la question précédente.

2. Relation de séparabilité par rapport à un langage

Il a été fixé, jusqu'à la fin de ce sujet, un certain langage régulier $L \subseteq \Sigma^*$, que nous ne connaissons pas mais que nous souhaitons apprendre. Autrement dit, notre objectif est de produire en temps fini et efficacement un automate qui reconnaît le langage L . Nous accédons au langage L par l'intermédiaire d'un oracle qui répond à deux types de requêtes :

- des *requêtes d'appartenance*, prenant la forme de la question « le mot $w \in \Sigma^*$ appartient-il au langage fixé L ? »
- et des *requêtes d'équivalence*, prenant la forme de la question : « l'automate A reconnaît-il le langage fixé L ? ».

En cas de réponse négative à une requête d'équivalence, l'oracle nous pourvoit d'un contre-exemple $c \in \Sigma^*$ pour lequel l'automate A se trompe. Autrement dit, soit le mot c appartient au langage à apprendre L mais n'est pas reconnu par l'automate A , soit le mot c n'appartient pas au langage à apprendre L mais est reconnu par l'automate A .

Indication OCaml : L'oracle est empaqueté dans un module Oracle ainsi signé :

```

7.  type answer = Yes | Counterexample of word
8.
9.  val mem : word -> bool
10. val equiv : automaton -> answer

```

et s'utilise avec la syntaxe `Oracle.mem w` pour une requête d'appartenance ou `Oracle.equiv a` pour une requête d'équivalence.

Définition : Nous notons $\mathbb{1}_\Lambda : \Sigma^* \rightarrow \{\text{faux}, \text{vrai}\}$ la *fonction indicatrice* de tout langage $\Lambda \subseteq \Sigma^*$. Pour tout mot $u \in \Sigma^*$, $\mathbb{1}_\Lambda(u)$ prend la valeur `vrai` si et seulement si le mot u appartient au langage Λ .

Épreuve d'informatique d'option MP 2025

Définition : Deux mots quelconques $u_1 \in \Sigma^*$ et $u_2 \in \Sigma^*$ sont dits *séparés par le mot \bar{v} par rapport à un langage $\Lambda \subseteq \Sigma^*$* si l'on a

$$\mathbb{1}_\Lambda(u_1\bar{v}) \neq \mathbb{1}_\Lambda(u_2\bar{v}),$$

autrement dit, si concaténation u_1v appartient au langage Λ tandis que la concaténation $u_2\bar{v}$ n'appartient pas au langage Λ ou si la concaténation $u_1\bar{v}$ n'appartient pas au langage Λ tandis que la concaténation $u_2\bar{v}$ appartient au langage Λ . Nous disons alors que le mot \bar{v} est un *discriminant*.

Indication OCaml : Typographiquement, nous convenons de surligner les mots servant de discriminant dans ce sujet et marquons le rôle de discriminant en adoptant l'alias de type

11. type disc = word

- 10 – Écrire une fonction OCaml `separated_by (u1:word) (u2:word) (v:disc) : bool` qui teste si les deux mots u_1 et u_2 sont séparés par rapport au langage à apprendre L par le mot discriminant \bar{v} .

Définitions : Deux mots quelconques $u_1 \in \Sigma^*$ et $u_2 \in \Sigma^*$ sont dits *séparables par rapport à un langage $\Lambda \subseteq \Sigma^*$* , ou simplement *séparables*, s'il existe un discriminant qui les sépare. Ils sont dits *inséparables* sinon. Nous notons $u_1 \equiv_\Lambda u_2$ la relation d'inséparabilité par rapport au langage Λ .

- 11 – Montrer que la relation d'inséparabilité par rapport à un langage quelconque est une relation d'équivalence sur l'ensemble des mots Σ^* .
- 12 – Soient $A = (Q, q_0, \delta, \mathbb{1}_F)$ un automate quelconque, $L_A \subseteq \Sigma^*$ le langage reconnu par l'automate A et $(u_1, u_2) \in (\Sigma^*)^2$ deux mots tels que $\delta^*(q_0, u_1) = \delta^*(q_0, u_2)$. Montrer que les mots u_1 et u_2 sont inséparables par rapport au langage L_A .
- 13 – Citer un théorème liant langages réguliers d'une part et langages reconnus par un automate d'autre part. Déduire de la question 12 que le nombre de classes d'équivalences de la relation d'inséparabilité par rapport au langage à apprendre L est fini.

 Épreuve d'informatique d'option MP 2025

3. Arbre discriminant

Définition : Un *arbre discriminant* est un arbre de décision binaire, dont les sommets internes sont étiquetés par des discriminants et dont les feuilles sont étiquetées par des états distincts deux à deux.

```
12. type disctree = Node of disctree * disc * disctree
   13.           | Leaf of state
```

La figure 1 présente un exemple d'arbre discriminant T_0 , dans lequel les trois sommets internes sont figurés par des cercles et les quatre feuilles par des rectangles.

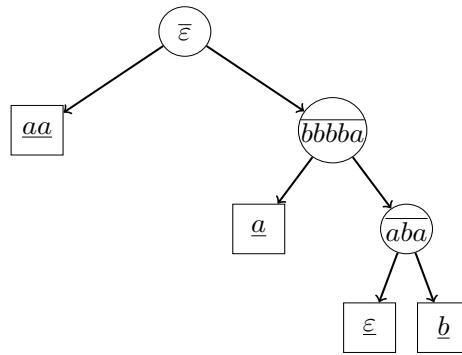


FIGURE 1 – Un exemple d'arbre discriminant : l'arbre T_0

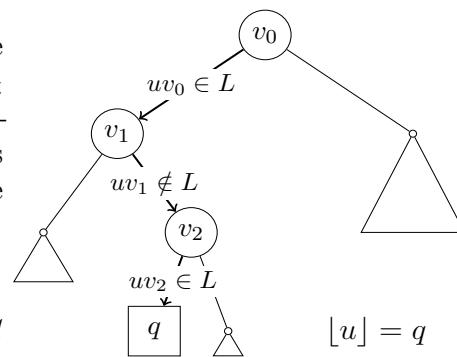
- 14 – Écrire une fonction OCaml `states_of_disctree (t:disctree) : state list` dont la valeur de retour est une liste des étiquettes des feuilles de l'arbre discriminant T . Il est attendu que la complexité en temps de `states_of_disctree` soit linéaire en le nombre de feuilles et qu'on le justifie.

Par exemple, avec l'arbre discriminant T_0 de la figure 1, la valeur `states_of_disctree t0` peut égaler la liste $[aa, a, \varepsilon, b]$, la liste $[\varepsilon, a, aa, b]$ ou encore toute autre permutation.

Définition : Cribler un mot $u \in \Sigma^*$ à travers un arbre discriminant T pour un langage L consiste à construire l'unique chemin $s_0 - s_1 - \dots - s_k$ dans l'arbre T où :

- le sommet s_0 est la racine de T ,
- pour tout indice i compris entre 0 et $k - 1$, le sommet s_i est un sommet interne et, en notant v_i l'étiquette du sommet interne s_i , si le mot concaténé uv_i appartient au langage à apprendre L , alors le sommet s_{i+1} est le fils gauche de s_i , sinon, le sommet s_{i+1} est le fils droit de s_i ,
- le sommet s_k est une feuille de l'arbre T .

et renvoyer l'état q porté par la feuille s_k . L'état q s'appelle le *cribat du mot u* et est noté $[u]$.



$$[u] = q$$

Épreuve d'informatique d'option MP 2025

Par exemple, en supposant que le langage à apprendre L est dénoté par l'expression régulière ab^*a , le criblat du mot $abbba$ par l'arbre discriminant \mathcal{T}_0 (figure 1) est l'état $[abbba] = \underline{aa}$, qui est l'extrémité du chemin $\bar{\varepsilon} - \underline{aa}$ obtenu en observant que $abbba$ appartient à L . En revanche, le criblat du mot abb est l'état $[abb] = \underline{a}$, qui est l'extrémité du chemin $\bar{\varepsilon} - \overline{bbbba} - \underline{a}$ obtenu en observant que abb appartient à L mais $abbbbbba$ n'appartient pas à L .

- 15 – Donner le criblat du mot ba pour l'exemple précédent.
- 16 – Écrire une fonction OCaml `sift (t:discree) (u:word) : state` dont la valeur de retour est le criblat $[u]$ du mot u à travers l'arbre discriminant T pour le langage L .
- 17 – Déterminer la complexité en temps dans le pire des cas de la fonction `sift t u` en fonction de tout ou partie des grandeurs suivantes : la longueur $|u|$ du mot u , le nombre n de sommets de l'arbre T , la hauteur h de l'arbre T , le maximum $M = \max_{v \in T} |v|$ des longueurs des mots discriminants de l'arbre T , le coût $\mu(|w|)$ d'un appel `Oracle.mem w`, le coût $\nu(|\mathcal{A}|)$ d'un appel `Oracle.equiv a`.
- 18 – Démontrer, en exhibant un discriminant, que deux mots de criblats distincts sont toujours séparables.
- 19 – Démontrer que deux mots inséparables ont les mêmes criblats.

Définition : Nous disons qu'un arbre discriminant est *accessible* si, pour tout état $\underline{w} \in \Sigma^*$, apparaissant dans la liste des étiquettes des feuilles, le criblat $[w]$ du mot w est l'état \underline{w} .

- 20 – Démontrer qu'il existe une constante dépendant uniquement du langage à apprendre L , qui majore le nombre de feuilles d'un arbre discriminant accessible.

Définition : Nous disons qu'un arbre discriminant est *démêlant* s'il comporte au moins un sommet interne et si l'étiquette de la racine est le discriminant mot vide $\bar{\varepsilon}$.

- 21 – Démontrer que si deux mots u_1 et $u_2 \in \Sigma^*$ ont le même criblat par rapport à un arbre discriminant démêlant, alors on a l'égalité $\mathbb{1}_L(u_1) = \mathbb{1}_L(u_2)$.

 Épreuve d'informatique d'option MP 2025

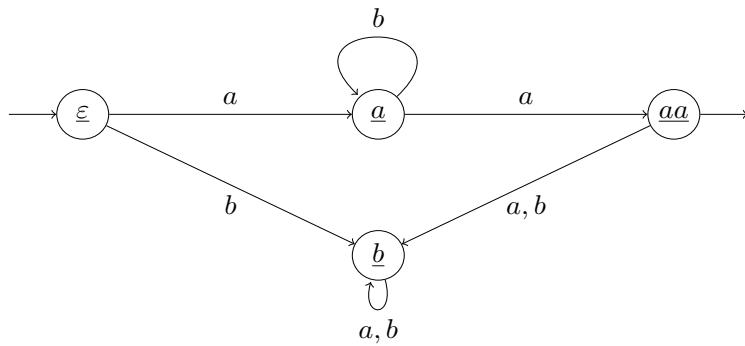
4. Automate tiré d'un arbre discriminant

Définitions : Nous disons qu'un arbre discriminant T est un *crible* s'il est accessible, démêlant et si le mot vide $\underline{\varepsilon}$ est un état qui apparaît dans la liste des étiquettes des feuilles.

L'automate associé à un crible T pour un langage L est l'automate A ainsi défini :

- l'ensemble des états de A est l'ensemble des étiquettes des feuilles de T ;
- l'état initial de A est l'état $\underline{\varepsilon}$;
- la fonction de transition associe l'état \underline{w} et la lettre x à l'état $\lfloor wx \rfloor$ obtenu en criblant le mot wx à travers l'arbre discriminant T pour le langage L ;
- l'ensemble des états finals est l'ensemble des mots \underline{w} où \underline{w} est une étiquette d'une feuille du sous-arbre gauche de T .

Par exemple, l'automate associé au crible \mathcal{T}_0 de la figure 1 pour le langage dénoté par ab^*a est l'automate ainsi figuré



□ 22 – Écrire une fonction OCaml `automaton_of_disctree (t:disctree) : automaton` dont la valeur de retour est l'automate associé au crible T .

On recopiera et on complètera le code suivant :

```

7.   let automaton_of_disctree (t:disctree) : automaton =
8.     { initial = .... ;
9.       transitions = .... ;
10.      finals = ....
11.    }
    
```

□ 23 – En supposant que le langage à apprendre L n'est ni le langage vide, ni le langage plein Σ^* , décrire une construction, à l'aide des fonctions `Oracle.mem` et `Oracle.equiv`, d'un premier crible formé d'un seul sommet interne et de deux feuilles.

Dans les questions 24, 25 et 26, nous notons $A = (Q, \underline{\varepsilon}, \delta, \mathbb{1}_F)$ l'automate associé à un certain crible T déjà construit. Nous supposons que `Oracle.equiv` a renvoie `Oracle.Counterexample c`, où c est le mot non vide $c = c_0c_1 \cdots c_{\gamma-1} \in \Sigma^*$. Nous nous proposons de construire un nouveau crible T' ayant un état de plus que le crible T .

 Épreuve d'informatique d'option MP 2025

Pour tout indice i compris entre 0 et γ , nous appelons $\pi_i = c_0c_1 \cdots c_{i-1}$ le préfixe de c de longueur i et $\sigma_i = c_{i+1}c_1 \cdots c_{\gamma-1}$ le suffixe de longueur $\gamma - i - 1$ de sorte que le mot c admette la factorisation $c = \pi_i c_i \sigma_i$. Nous notons les états

$$\underline{p}_i = \lfloor \pi_i \rfloor \in \Sigma^* \quad \text{et} \quad \hat{p}_i = \delta^*(q_\varepsilon, \pi_i) \in \Sigma^*$$

et posons encore

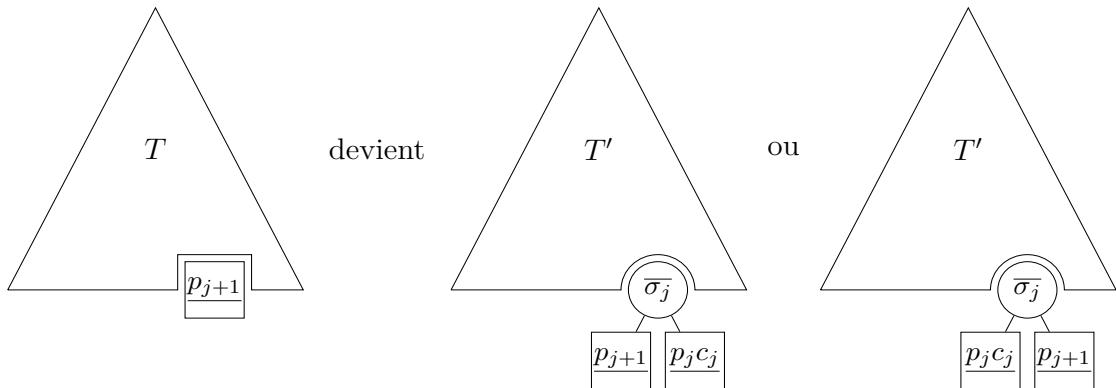
$$\tau_i = p_i c_i \sigma_i \in \Sigma^* \quad \text{et} \quad \hat{\tau}_i = \hat{p}_i c_i \sigma_i \in \Sigma^*.$$

- 24 – Démontrer que l'on a $\mathbb{1}_L(\tau_\gamma) \neq \mathbb{1}_L(\hat{\tau}_\gamma)$.

On admet qu'il existe un indice i compris entre 0 et $\gamma - 1$ tel que les états \underline{p}_i et \hat{p}_i sont égaux, les états \underline{p}_{i+1} et \hat{p}_{i+1} sont distincts et, alors, les mots $p_i c_i$ et p_{i+1} sont séparés par le discriminant $\overline{\sigma}_i$.

- 25 – Écrire une fonction OCaml `split (t:disctree) (c:word) : state * state * disc` dont la valeur de retour est le triplet $(\underline{p}_j c_j, \underline{p}_{j+1}, \overline{\sigma}_j)$ où l'entier j est le plus petit des indices i compris entre 0 et $\gamma - 1$ obtenu par la question 24.

On admet de la question 24 que l'arbre discriminant T' obtenu en substituant un arbre constitué d'un sommet interne d'étiquette $\overline{\sigma}_j$, d'une feuille d'étiquette l'ancien état \underline{p}_{j+1} et d'une feuille d'étiquette un nouvel état $p_j c_j$ à la place de la feuille d'étiquette \underline{p}_{j+1} dans l'arbre discriminant T jouit encore des propriétés de crible. Schématiquement,



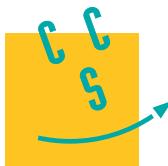
- 26 – Écrire une fonction OCaml `substitute (t:disctree) ((pp,p,s):state * state * disc) : disctree` ainsi spécifiée :

Précondition : Le triplet (pp, p, s) est la valeur de retour de la fonction `split` écrite à la question 25.

Valeur de retour : Crible T' décrit ci-dessus.

- 27 – Concevoir un algorithme complet qui apprend un langage régulier à base de requêtes d'appartenance et de requêtes d'équivalence en renvoyant un automate. Justifier sa terminaison à l'aide d'un variant de boucle.

- 28 – Démontrer que l'automate construit à la question 27 possède un nombre d'état minimal parmi l'ensemble des automates qui reconnaissent le langage à apprendre.



Informatique

MPI

2025

CONCOURS CENTRALE-SUPÉLEC

4 heures

Calculatrice autorisée

Rush Hour

L'épreuve est composée de deux problèmes indépendants. L'objectif du premier problème est d'écrire un programme permettant de résoudre le casse-tête de déplacements nommé *Rush Hour* dont le but est de faire sortir une voiture d'un parking. Dans le second problème, on souhaite connaître le maximum de véhicules qu'il est possible de placer sur le terrain d'une *casse automobile*.

Les candidats devront répondre aux questions en utilisant le langage C dans le premier problème et le langage OCaml dans le second problème. Ils sont invités, lorsque c'est pertinent, à décrire succinctement le fonctionnement de leurs programmes, un dessin étant souvent plus parlant qu'un long discours. Le barème tient compte de la clarté et de la concision des programmes. Nous recommandons de choisir des noms de variables intelligibles et de structurer, si cela s'avère nécessaire, de longs codes par des fonctions auxiliaires dont on décrira le rôle.

On identifiera une même grandeur écrite dans deux polices différentes, en italique lorsque nous la considérerons d'un point de vue mathématique (par exemple n), et en police à largeur fixe lorsque nous la considérerons d'un point de vue informatique (par exemple `n`). Si $a, b \in \mathbb{Z}$, on note $\llbracket a, b \rrbracket$ l'ensemble $\{k \in \mathbb{Z} \mid a \leq k < b\} = \{a, a+1, \dots, b-1\}$. Un graphe non orienté est un couple $G = (S, A)$ où S est un ensemble fini non vide d'éléments appelés sommets et A un ensemble de parties de S à deux éléments, appelées arêtes. Si $x, y \in S$ sont deux sommets distincts de S , l'arête $\{x, y\}$ sera notée $x - y$. Un graphe orienté est un couple $G = (S, A)$ où S est un ensemble fini non vide d'éléments appelés sommets et A un ensemble d'éléments (x, y) de $S \times S$ tels que $x \neq y$, appelés arcs. Si $x, y \in S$ sont deux sommets distincts de S , l'arc (x, y) sera noté $x \rightarrow y$.

Par complexité en temps d'un algorithme, on entend nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de l'algorithme dans le pire des cas.

Partie A – Rush Hour

Le jeu de plateau *Rush Hour* simule un embouteillage dans un parking à l'heure de pointe. Dans ce jeu à un joueur, on doit extraire le véhicule gris d'une grille dans laquelle plusieurs autres véhicules noirs bloquent la sortie. Il est possible de déplacer chaque véhicule, d'une ou plusieurs cases, vers l'avant ou l'arrière. Les véhicules sont soit des voitures, soit des camions. Les voitures occupent 2 cases tandis que les camions occupent 3 cases. La sortie est repérée par une flèche qui se situe sur le côté droit de la grille. La figure 1 illustre une position de départ du jeu.

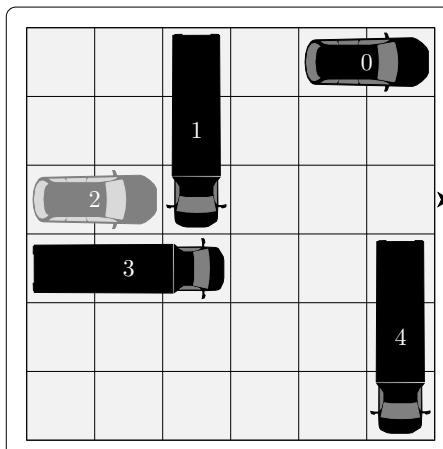


Figure 1.

Pour résoudre ce casse-tête, on commence par déplacer la voiture 0 d'une case en arrière. On se retrouve alors dans la configuration de la figure 2.a. Puis on déplace le camion 4 de 3 cases en arrière, le camion 3 de 3 cases en avant et le camion 1 de 3 cases en avant. On se retrouve alors dans la configuration de la figure 2.b qui nous permet de déplacer la voiture grise numérotée 2 de 3 cases vers l'avant et nous retrouver sans la configuration de la figure 2.c.

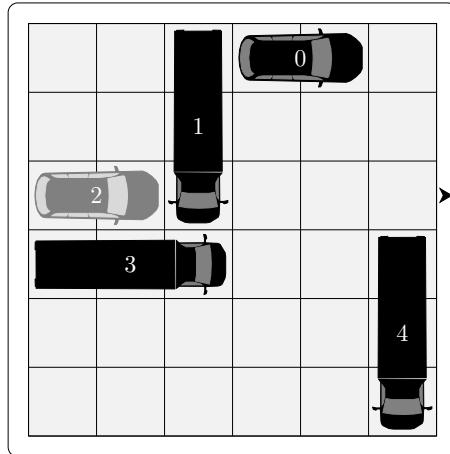


Figure 2.a

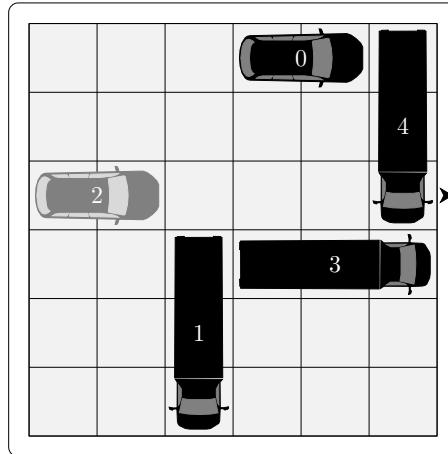


Figure 2.b

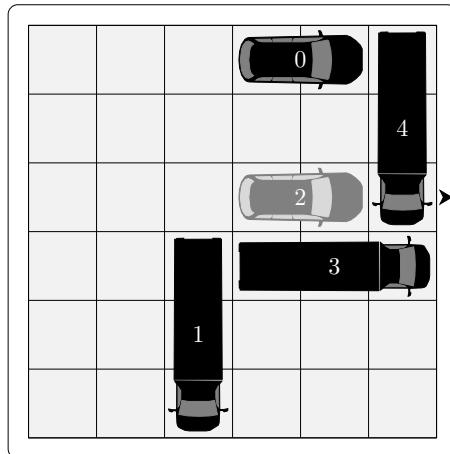


Figure 2.c

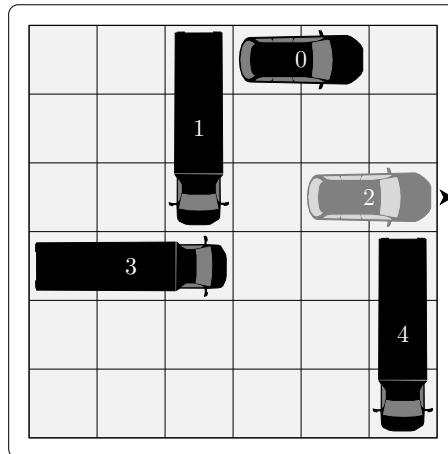


Figure 2.d

On déplace alors le camion 1 de 3 cases vers l'arrière, puis le camion 3 de 3 cases vers l'arrière et le camion 4 de 3 cases vers l'avant. On avance enfin la voiture grise numérotée 2 d'une case. Elle se retrouve devant la sortie indiquée par la flèche sur la droite de la grille. Cette configuration gagnante est indiquée à la figure 2.d.

Dans ce problème, nous supposerons que la grille du jeu est carrée et comporte $M \times M$ cases. Comme tous les véhicules occupent au moins 2 cases, le nombre de véhicules présents sur la grille est majoré par $(M \times M)/2$. Dans notre programme, cet entier est noté **MAX_V**. Les cases sont repérées par leurs coordonnées $(i, j) \in \llbracket 0, M \rrbracket^2$, l'indice i indiquant le numéro de la ligne et l'indice j indiquant le numéro de la colonne. La ligne sur laquelle se trouve la sortie est indiquée par S . Dans l'exemple de la figure 3 disponible page suivante, on a $M = 6$ et $S = 2$. Dans la suite du sujet, nous supposerons que la voiture grise est l'unique voiture placée horizontalement sur la ligne de la sortie. Comme c'est la seule voiture pouvant sortir de la grille, contrairement aux schémas présents dans ce sujet sur lesquels elle est en gris, elle ne fera l'objet d'aucun marquage particulier dans notre algorithme.

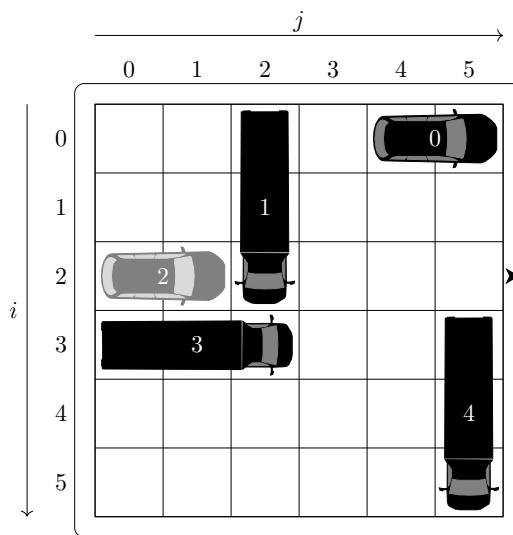


Figure 3.

Nous supposerons que les entêtes suivants ont été inclus et que les constantes ci-dessous sont définies au début du programme.

```

1 #include <assert.h>
2 #include <stdbool.h>
3 #include <stdint.h>
4 #include <stdlib.h>
5
6 #define M 6
7 #define MAX_V (M * M) / 2
8 #define S 2

```

Les directives `#define` présentes ici ont pour but de définir des synonymes pour les constantes entières utilisées dans le sujet. Chaque fois que l'identifiant `M` sera présent dans la suite, il sera directement substitué par la constante 6 par le compilateur.

I – Introduction

La position de chaque véhicule sur la grille est représentée par la structure de donnée suivante :

```

1 struct vehicule_s {
2     int i;
3     int j;
4     int longueur;
5     char direction;
6 };
7 typedef struct vehicule_s vehicule;

```

Le couple (i, j) représente les coordonnées de la case occupée par le véhicule située la plus en haut à gauche sur la grille. Par exemple, sur la figure 3, les coordonnées du véhicule 0 sont $(0, 4)$ et celles du véhicule 1 sont $(0, 2)$. Le champ `longueur` indique la longueur, en nombre de cases, du véhicule : 2 pour une voiture et 3 pour un camion. Enfin, le champ `direction` indique la direction du véhicule : 'H' si le véhicule est placé de manière horizontale, comme le véhicule 0 de la figure 3, et 'V' si le véhicule est placé de manière verticale, comme le véhicule 1 de la figure 3.

La configuration des véhicules sur le plateau de jeu est représentée par la structure suivante :

```

1 struct etat_s {
2     vehicule v[MAX_V];
3     int n;
4     int distance;
5 };
6 typedef struct etat_s etat;

```

Le nombre de véhicules présents sur la grille est donné par le champ `n` et vérifie $n \leq MAX_V$. Pour tout $k \in [0, n]$, le champ `v[k]` donne la position et les caractéristiques du véhicule d'indice k . Enfin, le champ `distance` n'est pas utilisé pour le moment et sera initialisé à 0.

- Q1.** Écrire une fonction `etat *e_nouveau(void)` renvoyant un état ne possédant aucune voiture sur son plateau, ainsi qu'une fonction `void e_libere(etat *e)` libérant la mémoire associée à cet état.
- Q2.** Écrire une fonction `bool e_egal(etat *e1, etat *e2)` déterminant si deux états sont égaux, c'est-à-dire possédant le même nombre de véhicules n et tels que pour tout $k \in [0, n]$ les véhicules d'indice k ont mêmes positions et mêmes caractéristiques. Deux états peuvent être égaux en ayant des valeurs différentes associées au champ `distance`.
- Q3.** Écrire une fonction `void e_ajoute_vehicule(int i, int j, int longueur, char direction, etat *e)` ajoutant à l'état e un véhicule à la position (i, j) dont la longueur et la direction sont données par les variables homonymes. On ne cherchera pas à vérifier que les cases sont libres, mais on utilisera une assertion pour vérifier que le nombre n de véhicules présents sur la grille reste inférieur ou égal à `MAX_V`.

Afin de faciliter la manipulation des états, nous allons définir une structure `plateau` représentant le plateau de jeu. Cette structure est définie comme suit :

```

1 const int vide = -1;
2
3 struct plateau_s {
4     int tab[M][M];
5 };
6 typedef struct plateau_s plateau;
```

- Q4.** Écrire une fonction `plateau *p_etat(etat *e)` renvoyant un plateau dont les cases sont initialisées à la valeur `vide`, excepté les cases sur lesquelles se trouvent les véhicules de l'état e qui sont chacune initialisée avec l'indice du véhicule la recouvrant.

Pour chaque état, seul un nombre fini de déplacements est valide. Pour manipuler cette collection, nous allons utiliser une liste chaînée, dont le maillon `déplacement` est défini comme ceci :

```

1 struct deplacement_s {
2     int k;
3     int pas;
4     struct deplacement_s *suivant;
5 };
6 typedef struct deplacement_s deplacement;
```

Un tel maillon permet de représenter un déplacement de la voiture d'indice k de `pas` cases dans le sens des i (ou des j) croissants. Par exemple, dans la solution du casse-tête de la figure 1, le premier mouvement de la voiture 0 d'une case vers la gauche aura une valeur de $k = 0$ et de $pas = -1$. Le second mouvement du camion 4 de 3 cases vers le haut aura une valeur de $k = 4$ et de $pas = -3$. Le champ `suivant` pointe vers le prochain maillon de la liste chaînée. La liste vide est représentée par le pointeur `NULL`.

- Q5.** Écrire une fonction `deplacement *d_cons(int k, int pas, deplacement *lst)` ajoutant en tête de la liste chaînée `lst`, un déplacement de la voiture d'indice k de `pas` cases.

Pour la suite du sujet, on suppose disposer d'une fonction `deplacement *e_deplacements(etat *e)` renvoyant la liste des déplacements valides dans l'état e .

- Q6.** Implémenter une fonction `void d_libere(deplacement *lst)` libérant la mémoire occupée par les différents éléments de la liste `lst`.

II – Implémentation d'une file

Dans cette partie, nous allons réaliser une file à l'aide d'un tableau circulaire. Sur l'exemple ci-dessous, nous avons une file d'entiers représentée par un tableau possédant 8 cases. On dit que la file a une capacité de 8 éléments.

8					17	25	9
0	1	2	3	4	5	6	7
prochain							

Ce tableau représente la file $17 \leftarrow 25 \leftarrow 9 \leftarrow 8$. Le prochain élément à sortir de la file est 17 et le dernier élément à avoir été ajouté est 8. L'indice **prochain**, qui est égal à 5 sur cet exemple, indique l'indice du prochain élément qui sortira de la file. Cette file possède 4 éléments. On dit que sa taille est égale à 4.

La file que nous allons implémenter est une file d'états. Elle est représentée par la structure suivante :

```

1 struct file_s {
2     etat **tab;
3     int prochain;
4     int taille;
5     int capacite;
6 };
7 typedef struct file_s file;

```

- Q7.** Écrire une fonction `file *f_nouveau(void)` qui renvoie une file de taille et de capacité nulle. Écrire d'autre part une fonction `void f_libere(file *f)` libérant la mémoire associée à la file *f*. Cette fonction ne devra pas libérer la mémoire associée aux états présents dans la file.
- Q8.** Écrire une fonction `void f_change_capacite(file *f, int c)` prenant en entrée une file de taille *m* ainsi qu'un entier $c \geq m$. Cette fonction devra modifier la capacité de la file *f* afin qu'elle devienne égale à *c*.
- Q9.** Écrire les fonctions `void f_enfile(file *f, etat *e)` et `etat *f_defile(file *f)` permettant respectivement d'ajouter un nouvel élément *e* à l'entrée de la file *f* et d'enlever et récupérer un élément à la sortie de la file. Lorsqu'on souhaite ajouter un nouvel élément dans une file dont la capacité est égale à la taille, on multiplie cette capacité par 2 (sauf pour le cas particulier où la capacité est nulle où on la change en 1) avant d'ajouter notre nouvel élément.

On souhaite montrer que la complexité amortie des fonctions `f_enfile` et `f_defile` est en $\mathcal{O}(1)$, c'est-à-dire que si l'on part d'une file vide, la complexité d'une succession de *n* appels, chaque appel se faisant à l'une de ces fonctions, a une complexité en $\mathcal{O}(n)$. Pour les deux questions suivantes, et seulement pour ces deux questions, la complexité d'une opération sera calculée comme le nombre d'accès aux éléments du tableau `tab`, que ces accès soient en lecture ou en écriture. Nous allons utiliser pour cela la méthode du potentiel. On définit le potentiel Φ d'une file *f* par

$$\Phi(f) = |4 \cdot \text{taille}(f) - 2 \cdot \text{capacite}(f)|.$$

On note f_0, f_1, \dots, f_n les états successifs de la file. L'état f_0 est son état initial qui est une file de taille et de capacité nulle. Pour tout $k \in \llbracket 1, n \rrbracket$, f_k est l'état de la file après la *k*-ième opération.

$$f_0 \xrightarrow{\text{op}_1} f_1 \xrightarrow{\text{op}_2} f_2 \cdots f_{n-1} \xrightarrow{\text{op}_n} f_n.$$

Pour tout $k \in \llbracket 1, n \rrbracket$, on note C_k la complexité de l'opération op_k qui fait passer la file de l'état f_{k-1} à l'état f_k .

- Q10.** Montrer qu'il existe un entier $K \geq 0$ que l'on déterminera, tel que

$$\forall k \in \llbracket 1, n \rrbracket, \quad C_k + \Phi(f_k) - \Phi(f_{k-1}) \leq K.$$

- Q11.** En déduire que

$$\sum_{k=1}^n C_k \leq Kn$$

puis conclure.

III – Table de hachage

Dans cette partie, nous allons implémenter une structure d'ensemble à l'aide d'une table de hachage en adressage ouvert. Les éléments de cet ensemble sont de type `etat *`. Nous allons commencer par définir une fonction de hachage. Pour tout état e , on définit

$$\text{hash}(e) = \sum_{\substack{0 \leq i < M \\ 0 \leq j < M}} d_{i,j} 3^{iM+j}$$

où pour tout $i, j \in \llbracket 0, M \rrbracket$

$$d_{i,j} = \begin{cases} 0 & \text{si la case } (i, j) \text{ est inoccupée,} \\ 1 & \text{si la case } (i, j) \text{ est occupée par un véhicule horizontal,} \\ 2 & \text{si la case } (i, j) \text{ est occupée par un véhicule vertical.} \end{cases}$$

On suppose disposer d'une fonction `plateau *p_hash(etat *e)` qui étant donné un état e , renvoie un plateau p dont les éléments `tab[i][j]` sont égaux à $d_{i,j}$.

Q12. Écrire une fonction `int64_t e_hash(etat *e)` prenant en entrée un état e et renvoyant $\text{hash}(e)$.

Q13. Montrer que si e_1 et e_2 sont deux états accessibles à partir d'un même état initial e et que $\text{hash}(e_1) = \text{hash}(e_2)$, alors $e_1 = e_2$.

Pour stocker les états dans notre table de hachage, nous allons utiliser un tableau possédant un nombre de cases appelé capacité de la table de hachage. Cette capacité sera de la forme 2^p . La taille de la table de hachage est le nombre d'états qu'elle contient. Lorsqu'on souhaite ajouter un état e dans notre table de hachage, on commencera par calculer $h = \text{hash}(e) \bmod 2^p$. Si la case d'indice h est vide, on placera e dans cette case. Sinon, on cherchera la première case vide en partant de l'indice h et en avançant de un en un de manière circulaire dans le tableau, jusqu'à trouver une case vide. Par exemple, si l'on part d'une table de hachage vide de capacité 8, et qu'on ajoute successivement les états e_0, e_1, e_2, e_3 et e_4 ayant des hash respectifs de $h_0 = 1, h_1 = 7, h_2 = 3, h_3 = 3$ et $h_4 = 7$, on obtiendra la table de hachage suivante.

e_4	e_0		e_2	e_3			e_1
0	1	2	3	4	5	6	7

Lorsque la table de hachage est trop peuplée, on double sa capacité et on réinsère les éléments de l'ancienne table de hachage dans la nouvelle. Nous utiliserons la structure suivante pour représenter notre table de hachage.

```

1 struct tableh_s {
2     etat **tab;
3     int taille;
4     int capacite;
5 };
6 typedef struct tableh_s tableh;
```

Le tableau `tab` contient les différents états présents dans la table de hachage. Nous utiliserons le pointeur `NULL` pour représenter les cases vides de la table. On suppose que l'on dispose d'une fonction `tableh *t_nouveau(void)` créant une table de hachage vide ayant une capacité de 1 et d'une fonction `void t_libere(tableh *t)` libérant la mémoire associée à la table de hachage t , ainsi que la mémoire associée à tous les états qu'elle contient.

Q14. Écrire une fonction `bool t_contient(tableh *t, etat *e)` déterminant si l'état e est présent dans la table de hachage t .

On suppose disposer d'une fonction `void t_augmente_capacite(tableh *t)` qui prend en entrée une table de hachage t de capacité $c \geq 1$ et qui augmente sa capacité à $2c$.

Q15. Écrire une fonction `void t ajoute(tableh *t, etat *e)` qui ajoute un état e à la table de hachage t . Si m est la taille de la table et c sa capacité, on fera attention à ce que l'on ait toujours $m \leq 2c/3$.

Q16. Expliquer en quoi ce serait une mauvaise idée de remplacer le 3^{iM+j} par 4^{iM+j} dans la définition de $\text{hash}(e)$.

IV – Recherche d'une solution optimale

On supposera disposer dans cette partie d'une fonction `bool e_gagne(etat *e)` déterminant si la voiture grise est devant la sortie, ainsi que d'une fonction `etat *e_copie(etat *e)` effectuant une copie d'un état `e`.

Q17. Écrire une fonction `int resout(etat *e)` prenant en entrée un état `e` et renvoyant le nombre minimal de mouvements pour résoudre le jeu. La fonction devra renvoyer `-1` dans le cas où il n'y a pas de solution. On souhaite de plus que la mémoire associée à l'état `e` soit libérée à la fin de l'appel à `resout`. On pourra utiliser le champ `distance` d'un état pour indiquer le nombre de coups minimal nécessaires pour atteindre cet état à partir de l'état initial.

Q18. Montrer que la fonction `resout` n'engendre aucun problème mémoire.

Partie B – La casse automobile

Dans cette partie, nous disposons d'un terrain sur lequel nous souhaitons entreposer le plus grand nombre de véhicules hors d'usage. Ce terrain est modélisé par un ensemble de cases du plan, extraites d'une grille régulière. La taille d'un terrain désigne le nombre de cases le constituant. Deux types de véhicules sont disponibles : les voitures, qui occupent 2 cases, et les camions, qui occupent 3 cases.

Le premier objectif de cette partie est d'implémenter un algorithme efficace, ayant une complexité polynomiale en la taille du terrain, permettant de déterminer le maximum de voitures qu'il est possible de disposer sur ce dernier. Cet algorithme nous permet notamment de décider s'il est possible de couvrir tout le terrain à l'aide de voitures. La figure 4 donne un exemple de couverture complète d'un terrain possédant 14 cases à l'aide de voitures.

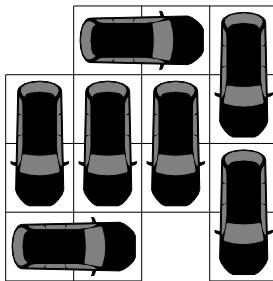


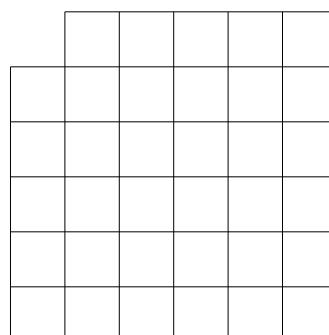
Figure 4.

Le second objectif de cette partie est de démontrer que, si l'on remplace les voitures, qui occupent 2 cases, par des camions, qui occupent 3 cases, le problème devient NP-complet.

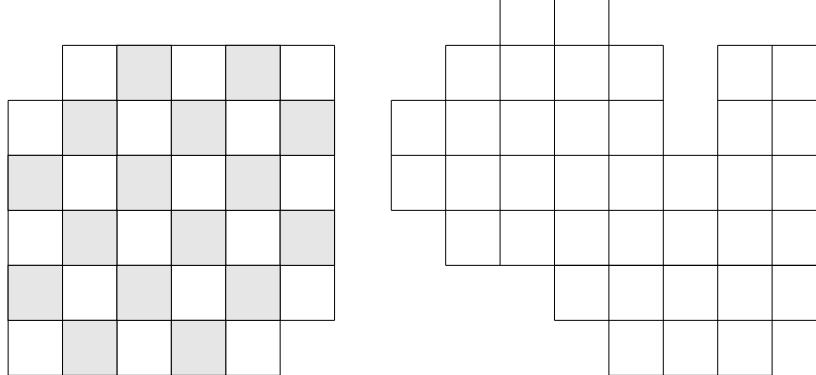
I – Remplir une casse avec le maximum de voitures

Dans cette partie B.I, tous les véhicules considérés sont des voitures et occupent 2 cases.

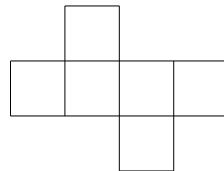
Q19. Prouver qu'il est impossible de couvrir tout le terrain suivant à l'aide de voitures. En déduire une condition nécessaire sur le terrain pour qu'il soit possible de le couvrir entièrement à l'aide de voitures.



Q20. Prouver qu'il est impossible de couvrir entièrement le terrain de gauche ci-dessous à l'aide de voitures. Ce terrain a une taille de 34 cases et la coloration de ses cases en blanc et en noir à la manière d'un échiquier n'a aucune incidence sur la disposition des voitures. Est-il possible de couvrir entièrement le terrain de droite ci-dessous ? On attend soit un exemple de couverture, soit une preuve de son impossibilité. En déduire une nouvelle condition nécessaire sur le terrain pour qu'il soit possible de le couvrir entièrement à l'aide de voitures.



Q21. En considérant le terrain ci-dessous, prouver que la condition nécessaire trouvée dans la question précédente n'est pas suffisante pour assurer l'existence d'une couverture complète à l'aide de voitures.



Dans la suite de cette partie, nous allons écrire un algorithme permettant de déterminer le plus grand nombre de voitures qu'il est possible de disposer sur un terrain, ainsi qu'une telle disposition. Pour cela, on définit le graphe non orienté $G = (S, A)$ dont l'ensemble des sommets S est constitué des cases du terrain, une arête $x - y$ appartenant à A si et seulement si x et y sont deux cases adjacentes, c'est-à-dire lorsque y est l'une des 4 cases au-dessus, en dessous, à gauche ou à droite de x . Nous choisissons arbitrairement de colorier une case du terrain en noir, puis de colorier le reste du terrain à la manière d'un échiquier en alternant les couleurs blanches et noires. On note N l'ensemble des sommets noirs et B l'ensemble des sommets blancs. Puisqu'une arête relie toujours deux sommets de couleurs différentes, $G = (N \sqcup B, A)$ est un graphe biparti. On numérote les r cases du terrain en commençant par les p cases noires, puis les q cases blanches. On a donc $r = p + q$, $N = \llbracket 0, p \rrbracket$ et $B = \llbracket p, r \rrbracket$. En OCaml, ce graphe sera représenté par un tableau de listes d'adjacence ($g : \text{int list array}$) ainsi que l'entier ($p : \text{int}$). Pour tout $x \in \llbracket 0, r \rrbracket$, $g.(x)$ est la liste des voisins du sommet x .

On rappelle qu'un couplage de $G = (S, A)$ est une partie C de A telle que chaque sommet n'est l'extrémité que d'au plus une arête de C . Un sommet $x \in S$ est dit libre lorsqu'il n'est l'extrémité d'aucune arête de C . Deux sommets $x, y \in S$ sont dits conjoints lorsqu'ils sont les extrémités d'une même arête de C . En OCaml, un couplage sera représenté par le tableau (`conjoint : int option array`) défini par

- `conjoint.(x) = None` lorsque le sommet x est libre.
- `conjoint.(x) = Some y` et `conjoint.(y) = Some x` lorsque x et y sont conjoints.

Un chemin $x_0 - x_1 - \dots - x_k$ de longueur k sera représenté en OCaml par la liste `[x0; ...; xk]` et sera représenté par la variable (`w : int list`). Un chemin est dit élémentaire lorsque ses sommets sont deux à deux distincts. Un chemin est dit alternant pour C lorsqu'il est élémentaire et ses arêtes $x_i - x_{i+1}$ alternent entre des éléments de C et $A \setminus C$. On dit qu'il est augmentant lorsqu'il est alternant et que ses extrémités sont libres.

Si X et Y sont deux parties de A , on définit la différence symétrique $X \Delta Y = (X \cup Y) \setminus (X \cap Y)$. On admet que cette opération est associative et commutative. Si X est une partie de A et w est un chemin, $X \Delta w$ est défini en considérant le chemin w comme un ensemble d'arêtes.

Q22. Montrer que si C est un couplage et que w est un chemin augmentant pour C , alors $C\Delta w$ est un couplage.

Q23. Écrire une fonction `delta (conjoint : int option array) (w : int list) : unit` prenant en entrée un couplage C représentée par le tableau `conjoint` et un chemin w que l'on supposera augmentant et modifiant `conjoint` de manière à ce qu'il représente le couplage $C\Delta w$.

On définit le graphe orienté G_C possédant les mêmes sommets que G et possédant un arc orienté pour chaque arête $n - b$ de G où $n \in N$ et $b \in B$: si $n - b \in C$, cet arc est $b \rightarrow n$, sinon cet arc est $n \rightarrow b$. D'autre part, dans la suite de cette partie, on utilisera l'enregistrement

```

1 type couplage = {
2   graphe : int list array;
3   p : int;
4   conjoint : int option array;
5 }
```

pour représenter un couplage ainsi que le graphe biparti auquel il est associé. Si `c` est une variable de type `couplage`, `c.graphe` permet d'accéder au tableau de listes d'adjacence du graphe, `c.p` permet d'accéder à l'entier p et `c.conjoint` permet d'accéder au tableau `conjoint`. On construit un couplage à l'aide de la syntaxe

```
{ graphe = g; p = p; conjoint = c }
```

Enfin, on note respectivement N_ℓ et B_ℓ l'ensemble des sommets libres de N et de B .

Q24. Montrer qu'un chemin $x_0 - x_1 - \dots - x_k$ de G dont le premier élément x_0 est dans N_ℓ et le dernier élément x_k est dans B_ℓ est augmentant pour C si et seulement si $x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_k$ est un chemin de G_C .

Q25. Écrire une fonction `oriente (c : couplage) : int list array` prenant en entrée un couplage C et renvoyant le tableau de listes d'adjacence représentant le graphe orienté G_C .

Pour tout $x \in S$, on note $d(x)$ la plus petite longueur parmi les longueurs des chemins alternants reliant un élément $n \in N_\ell$ à x . Si un tel chemin n'existe pas, on pose par convention $d(x) = +\infty$. Afin de manipuler efficacement cette distance, nous utiliserons par la suite l'enregistrement

```

1 type distance = {
2   d : int array;
3   dmin : int;
4 }
```

où `d` est un tableau tel que `d.(x) = d(x)` pour tout $x \in [0, r]$, la valeur $+\infty$ étant représenté par l'entier `max_int`, et `dmin` est la longueur d_{\min} du plus court chemin augmentant pour C , avec par convention `dmin = max_int` lorsqu'un tel chemin n'existe pas.

Q26. Écrire une fonction `distance (c : couplage) : distance` prenant en entrée un couplage C et renvoyant l'enregistrement (`dist : distance`) associé à la distance d . On pourra utiliser le module `Queue` de la bibliothèque standard OCaml offrant une implémentation d'une file impérative et dont la signature est rappelée ci-dessous.

- `Queue.create () : 'a Queue.t`, création d'une file vide.
- `Queue.push (x : 'a) (f : 'a Queue.t) : unit`, enfiler un élément x dans la file f .
- `Queue.pop (f : 'a Queue.t) : 'a`, dépiler et renvoyer le prochain élément en attente de la file f .
- `Queue.is_empty (f : 'a Queue.t) : bool`, renvoie `true` lorsque la file f est vide, `false` sinon.

Nous souhaitons construire le plus grand ensemble de chemins augmentants pour C tel que ces chemins n'aient aucun sommet en commun et soient tous de longueur d_{\min} . Un tel ensemble est appelé ensemble de chemins de Hopcroft-Karp. Pour construire cet ensemble, on commence par définir le graphe G_{HK} possédant les mêmes sommets que G_C , en ne conservant que les arcs $x \rightarrow y$ tels que $d(y) = d(x) + 1$ et $d(y) \leq d_{\min}$. Ensuite, on initialise notre ensemble de chemins à l'ensemble vide. Puis, pour chaque sommet $x \in N_\ell$, on cherche un chemin reliant x à un sommet $y \in B_\ell$ dans G_{HK} à l'aide d'un parcours en profondeur. Tant qu'un tel chemin existe, on retire de G_{HK} les sommets empruntés par notre chemin, on ajoute ce chemin à notre ensemble de chemins et on réitère l'opération. Pour notre implémentation, on ne

cherchera pas à générer le graphe G_{HK} , mais on travaillera uniquement à partir du graphe G_C .

On souhaite donc écrire une fonction `ensemble_chemins` (`c : couplage`) (`dist : distance`) : `int list list` prenant en entrée un couplage C , ainsi que l'enregistrement `dist` calculé par la fonction `distance` et renvoyant un ensemble de chemins de Hopcroft-Karp sous forme de liste. Pour cela, on introduit l'exception

```
1 exception Chemin of int list
```

et on définit la fonction

```
1 let ensemble_hc (c : couplage) (dist : distance) : int list list =
2   let r = Array.length c.graphe in
3   let visite = Array.make r false in
4   let go = oriente c in
5   let ensemble = ref [] in
6   let rec chemin (x : int) (w : int list) : unit =
7     <code1>
8   in
9   <code2>
10  ! ensemble
```

Q27. Compléter la partie `<code1>` de la fonction `ensemble_hc` afin de définir la fonction
`chemin (x : int) (w : int list) : unit.`

Cette fonction doit effectuer un parcours en profondeur dans G_{HK} à partir du sommet x en ayant déjà parcouru le chemin w codé à l'envers (le dernier sommet visité est en tête de liste), et renvoyer une exception `Chemin w` lorsqu'un chemin augmentant w a été trouvé.

Q28. Compléter la partie `<code2>` de la fonction `ensemble_hc`.

Dans la question suivante, on admet le lemme de Berge : un couplage dans un graphe est maximum pour l'inclusion si et seulement si il n'existe pas de chemin augmentant.

Q29. Définir la fonction `couplage_maximum` (`g : int list array`) (`p : int`) : `int option array` prenant en entrée un graphe biparti donné par son tableau de listes d'adjacence ainsi que par son nombre de sommets noirs et renvoyant un couplage maximum pour ce graphe.

II – Remplir une casse avec le maximum de camions

Dans cette partie, on souhaite montrer que le problème qui consiste, étant donné un terrain t ainsi qu'un entier n , de déterminer s'il est possible de placer au moins n camions sur le terrain t , est NP-complet. Dans la suite, on nomme ce problème 3CASSE.

Q30. Définir ce qu'est un problème de décision appartenant à la classe NP. Définir ce qu'est un problème NP-difficile, puis un problème NP-complet.

On définit les types

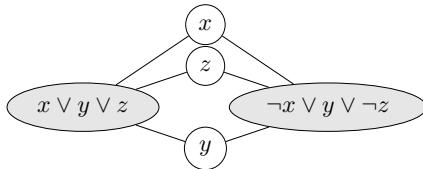
```
1 type terrain = bool array array
2 type direction = Horizontal | Vertical
3 type camion = {i : int; j : int; d : direction}
```

le type `terrain` représentant un terrain à l'aide d'un tableau bidimensionnel de booléens. Pour chaque case de cette grille, le booléen associé est égal à `true` si et seulement si elle fait partie du terrain. Le type `camion` représente quant à lui la position d'un camion sur le terrain, de manière similaire à ce qui a été décrit dans la partie A.I.

Q31. Écrire une fonction `est_valide` (`t : terrain`) (`n : int`) (`lst : camion list`) : `bool` prenant en entrée un terrain t , un entier n et une liste de positions de camions `lst` et renvoyant `true` si la liste possède n camions et qu'il est possible de les placer sur le terrain t comme spécifié. Si ce n'est pas le cas, la fonction renverra `false`.

Q32. En déduire que le problème 3CASSE est dans la classe NP.

On souhaite montrer que 3CASSE est NP-difficile. Pour cela, nous allons partir d'une variante du problème 3SAT, nommée Planar 3SAT et qui est également NP-complet. Tout comme pour 3SAT, les instances de ce problème de décision sont des formules logiques en forme normale conjonctive. Pour une formule φ on considère le graphe non orienté dont les sommets sont formés des variables et des clauses et dont les arêtes relient une variable à une clause si et seulement si la variable est présente dans la clause. Le problème Planar 3SAT est semblable à 3SAT mais restreint les formules à celles dont le graphe est planaire. Par exemple, la formule $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z)$ est planaire puisqu'on peut la représenter par le graphe suivant :



Étant donnée une formule φ de Planar 3SAT, nous allons construire une instance de 3CASSE. Pour cela, nous allons construire un terrain à l'aide de *gadgets*. L'essentiel de ces gadgets peut accueillir un nombre fixe de camions. Les gadgets représentant des clauses sont des exceptions et peuvent accueillir un camion en plus lorsque cette clause est vraie. Les gadgets sont reliés entre eux afin de créer des chemins, transportant des *signaux*. Chaque signal représente la valeur d'un littéral x ou de sa négation $\neg x$.

Concrètement, les gadgets sont reliés entre eux en superposant une de leur case, par laquelle sera transmis ce signal. Les gadgets sont agencés de manière à ce que tout camion placé sur le terrain se trouve entièrement sur un unique gadget. Par exemple, le terrain de la figure 5 est composé de 4 gadgets $\alpha, \beta, \gamma, \delta$ reliés par les cases a, b, c et d . Les gadgets α et γ sont reliés par la case a qui appartient aux deux gadgets. Sur ce terrain, nous avons placé deux camions. Le camion 1 appartient au gadget γ . Bien qu'il empiète sur le gadget α , on ne considère pas qu'il appartienne à ce gadget. Le camion 2 appartient quant à lui au gadget δ .

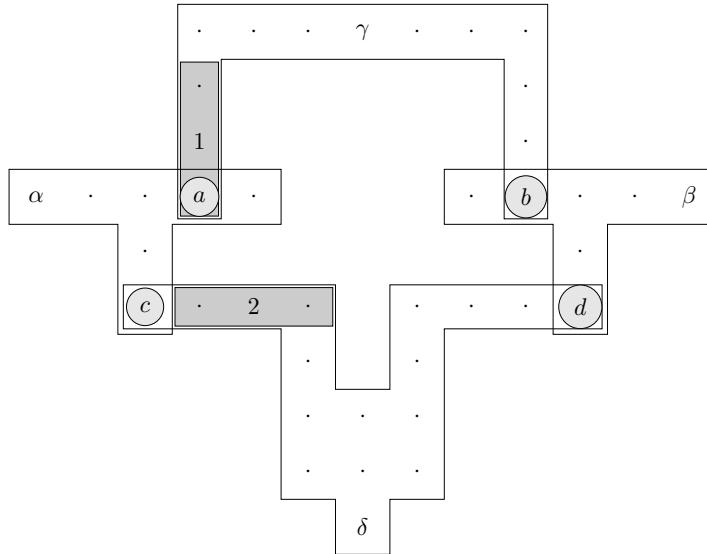
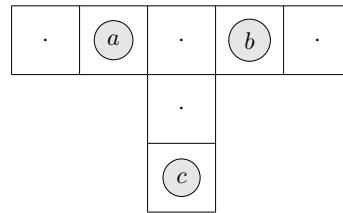


Figure 5.

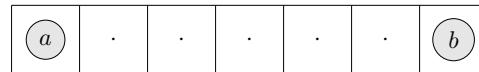
La disposition des camions sur le terrain va permettre de représenter les signaux. Si ε est un gadget et qu'un camion n'appartient pas à ce gadget empiète sur la case e du gadget ε , on dit que le gadget ε possède une protrusion en e . On dit dans ce cas que, pour le gadget ε , le signal e est faux. Dans le cas contraire, on dit que ce signal est vrai. Par exemple, avec la disposition de camions ci-dessus, pour le gadget α , on dit que le signal a est faux et que le signal c est vrai.

Q33. Afin de représenter une clause $a \vee b \vee c$, nous allons utiliser le gadget suivant sur lequel il peut y avoir des protrusions sur les cases a, b, c .



Montrer que ce gadget permet seulement d'accueillir 0 ou 1 camion et que l'un des signaux a, b, c est vrai si et seulement si il est possible d'y accueillir un camion.

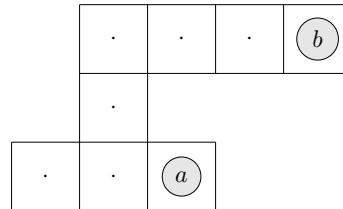
Q34. Afin de transmettre une valeur, nous allons utiliser le gadget suivant, sur lequel on souhaite placer deux camions.



Montrer que, si le signal a est faux, pour placer deux camions sur ce gadget, il est nécessaire de rendre le signal b faux pour le gadget suivant. Montrer que, si le signal a est vrai, il est possible de placer ces deux camions de manière à ce que le signal b soit vrai pour le gadget suivant.

L'idée du gadget précédent peut être généralisée afin d'obtenir un gadget permettant de transmettre un signal tout en se décalant d'un nombre de cases qui est un multiple de 3. Il est cependant nécessaire de transmettre des signaux tout en se décalant d'un nombre de cases quelconque.

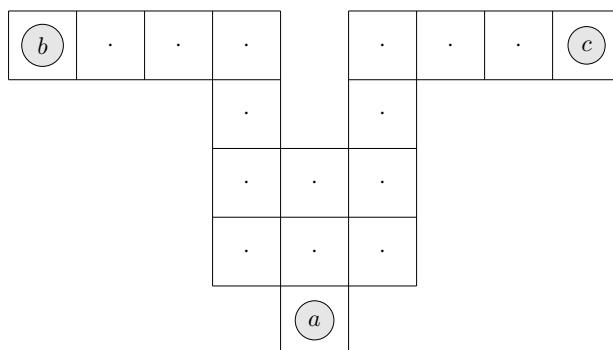
Q35. Afin de transmettre un signal tout en le décalant de deux cases, nous allons utiliser le gadget suivant, sur lequel on souhaite placer deux camions.



Montrer que, si le signal a est faux, pour placer deux camions sur ce gadget, il est nécessaire de rendre le signal b faux pour le gadget suivant. Montrer que, si le signal a est vrai, il est possible de placer ces deux camions de manière à ce que le signal b soit vrai pour le gadget suivant.

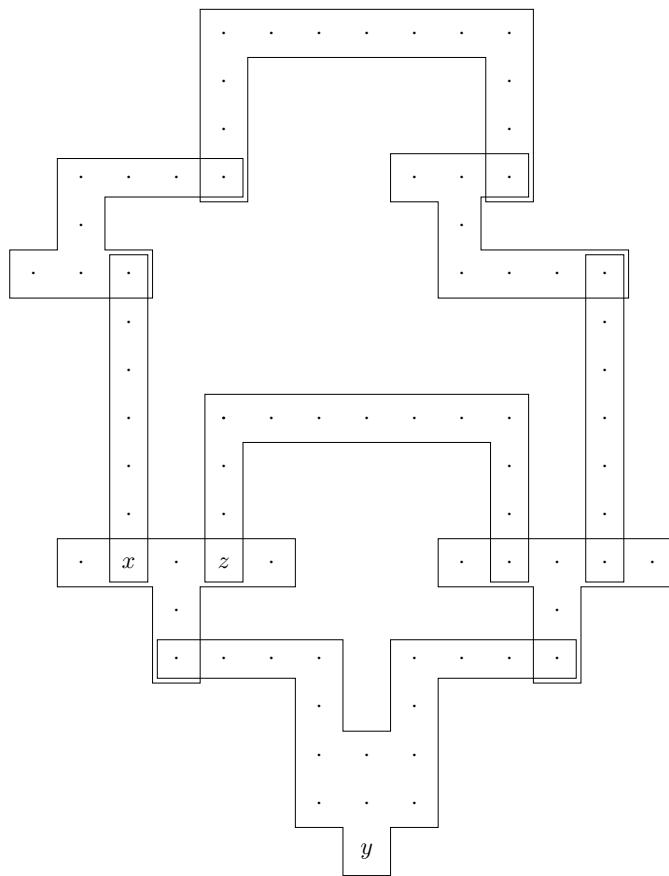
Un signal représentant un littéral x (ou sa négation $\neg x$) peut être utilisé dans plusieurs clauses. C'est pourquoi il est nécessaire de pouvoir le dupliquer.

Q36. Afin de diviser un signal en deux, nous allons utiliser le gadget suivant, sur lequel on souhaite placer cinq camions.



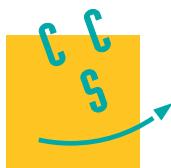
Montrer que, si le signal a est faux, pour placer cinq camions sur ce gadget, il est nécessaire de rendre les signaux b et c faux pour les gadgets suivants. Montrer que, si le signal a est vrai, il est possible de placer cinq camions de manière à ce que les signaux b et c soient vrais pour les gadgets suivants.

Q37. En utilisant les idées développées dans les questions 33 à 36 ainsi que le fait qu'il est possible de satisfaire la formule $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z)$, déterminer le plus grand nombre de camions qu'il est possible de placer sur le terrain suivant. On justifiera la réponse en donnant une telle disposition ainsi que les valeurs des variables x , y et z associées.



Q38. Montrer que le problème 3CASSE est NP-complet.

◊ Fin ◊



CONCOURS CENTRALE-SUPÉLEC

Option informatique

MP

2025

4 heures

Calculatrice autorisée

Meilleurs itinéraires dans un réseau ferroviaire

La recherche de l'itinéraire le plus court entre deux points d'un graphe pondéré est un problème facilement résolu par plusieurs algorithmes. Cette recherche est plus difficile dans un réseau ferroviaire, où les trajets sont restreints par des horaires de départ et d'arrivée à chaque arrêt, et du fait de l'existence de correspondances.

De plus, la notion de « meilleur trajet » se fait souvent en considérant non pas uniquement la durée totale du trajet, mais également d'autres critères tels que le prix du billet, le nombre de correspondances, l'heure de départ, les services à bord, etc. Il est rare d'obtenir un trajet qui minimise simultanément tous les critères : la minimisation d'un critère particulier se faisant souvent au détriment d'un autre. Certains trajets sont cependant moins bons sur tous les critères : ceux-ci n'ont pas besoin d'être considérés.

Ce sujet comporte quatre parties :

- la partie I étudie l'implémentation d'une file de priorité et d'un tri par tas ;
- la partie II définit et étudie la notion d'optimum de Pareto. Les notations introduites dans cette partie sont utilisées dans les parties suivantes ;
- la partie III étudie le calcul d'optimums de Pareto dans le cas d'un langage sur un alphabet fini ;
- la partie IV étudie le calcul d'optimums de Pareto dans un graphe pondéré.

Les parties III et IV sont indépendantes.

On trouvera dans l'annexe A en page 10 quelques points de rappels de syntaxe et de fonctions OCAML.

Consignes aux candidates et candidats Il doit être répondu aux questions de programmation en utilisant le langage OCAML. En cas d'écriture d'une fonction non demandée par l'énoncé, il doit être précisé son rôle, ainsi que sa signature (son type). Lorsque cela est pertinent, la description du fonctionnement des programmes qui ont été écrits est également incitée. On autorise toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `failwith` ainsi que les opérateurs comme `@`). Sauf précision de l'énoncé, l'utilisation d'autres modules est interdite.

Lorsqu'une question de programmation demande l'écriture d'une fonction, la signature de la fonction demandée est indiquée dans la question. Les candidates et candidats peuvent écrire une fonction dont la signature est compatible avec celle demandée. Par exemple, si l'énoncé demande l'écriture d'une fonction `int list -> int` qui renvoie le premier élément d'une liste d'entiers, écrire une fonction `'a list -> 'a` qui renvoie le premier élément d'une liste d'éléments quelconques sera considéré comme correct.

Il est possible d'admettre le résultat d'une question, y compris de supposer qu'une fonction demandée a été écrite, afin de traiter les questions suivantes.

I – Une file de priorité

L'objectif de cette première partie est d'implémenter une file de priorité persistante en utilisant une structure de tas d'appariement. Cette structure de tas permet également d'obtenir un tri de complexité optimale.

On peut voir les tas d'appariement comme des arbres n -aires auto-ajustés, c'est-à-dire pour lesquels les opérations sur la structure de données réorganisent l'arbre tout en réalisant globalement une forme d'ajustement. Ils disposent d'une implémentation très simple, tout en étant efficace en pratique, par exemple pour une utilisation comme file de priorité dans l'algorithme de Dijkstra, qui sera étudié dans la partie IV.

Contrairement à une implémentation classique en utilisant un tas binaire implémenté dans un tableau, les opérations sur un tas d'appariement à n éléments pourront avoir un coût en $\mathcal{O}(n)$ dans le pire cas. Cependant, on peut montrer qu'une séquence de n opérations consécutives sur cette structure de données a un coût total en $\mathcal{O}(n \log n)$, ce qui rend l'utilisation de cette structure tout aussi efficace lorsque l'on effectue une séquence d'opérations.

Dans toute cette partie on utilise la relation d'ordre totale implicite en OCAML sur les objets de type `'a` donnée par les opérateurs `<` ou `<=` ou encore par les fonctions `min` et `max`. En particulier, si les objets considérés sont des tuples, alors cet ordre correspond à l'ordre lexicographique sur les composantes de ces tuples.

I.1 – Tas d'appariement

Pour implémenter un tas d'appariement, on utilise des arbres n -aires étiquetés, dont les noeuds possèdent un nombre quelconque d'enfants. La figure 1 présente un exemple de tas d'appariement comportant 12 noeuds.

Dans ce sujet, un *arbre* est soit un arbre vide, soit un noeud formé d'une étiquette et d'une liste éventuellement vide d'arbres *non vides*, qui sont les *enfants* de ce noeud *parent*. Une *feuille* est un noeud qui ne possède pas d'enfants, c'est-à-dire dont la liste d'enfants est la liste vide. On représente en OCAML un arbre à l'aide du type suivant :

```
type 'a arbre =
| Vide
| Noeud of 'a * 'a arbre list
```

On remarquera que le constructeur `Vide` sert *uniquement* à dénoter l'arbre vide et ne peut pas apparaître dans une liste d'enfants puisque les enfants d'un noeud sont systématiquement supposés non vides.

Un *tas d'appariement* est un arbre vérifiant la propriété de *tas minimum* : l'étiquette d'un noeud est inférieure ou égale à celle de tous ses enfants.

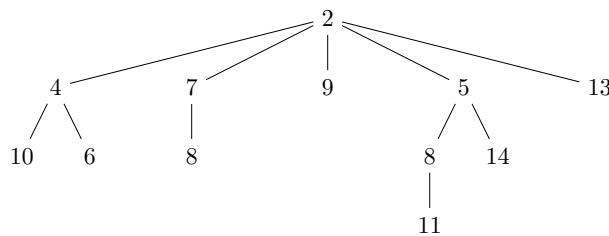


Figure 1 – Un tas d'appariement à 12 noeuds.

Q1. Écrire une fonction `min_valeur` : `'a arbre -> 'a` qui renvoie la valeur minimale d'un tas supposé non vide. Quelle est sa complexité dans le pire cas ?

Pour fusionner deux tas non vides, il suffit d'ajouter le tas dont l'étiquette de la racine est la plus grande comme nouvel enfant de l'arbre dont l'étiquette de la racine est la plus petite. Un nouvel enfant est ajouté en tête de la liste des enfants, donc devient le premier enfant à gauche. La complexité de cette opération est donc en $\mathcal{O}(1)$ dans le pire cas. La figure 2 illustre un certain nombre de fusions : les deux arbres au-dessus d'une accolade sont fusionnés pour obtenir l'arbre sous celle-ci.

Q2. Écrire une fonction `fusion` : `'a arbre -> 'a arbre -> 'a arbre` qui réalise la fusion de deux tas. On gèrera les cas où l'un ou les deux tas sont vides.

Pour ajouter un élément à un tas, on crée un nouveau tas contenant cet élément et on le fusionne avec le tas initial.

Q3. Écrire une fonction `insere` : `'a -> 'a arbre -> 'a arbre` qui réalise l'insertion d'un élément dans un tas. Quelle est sa complexité dans le pire cas ?

Pour supprimer l'élément minimal d'un tas, il suffit de supprimer la racine et de fusionner les enfants de celle-ci. Pour obtenir une forme d'auto-ajustement, on procède en deux étapes pour fusionner une liste de k arbres :

- on fusionne, *de gauche à droite*, les arbres deux par deux, tant que c'est possible ;
- on fusionne progressivement, en un seul arbre, les $\lceil k/2 \rceil$ arbres ainsi obtenus en prenant les arbres un par un, *de la droite vers la gauche*, en partant de l'arbre de droite.

Cette opération en deux étapes est appelée *fusion par paires* et donne le nom à cette structure de *tas d'appariement*.

La figure 2 illustre la suppression de la racine d'étiquette 2 du tas de la figure 1. Lors de la première étape de la fusion par paires, on effectue la fusion des enfants de gauche à droite deux par deux. Les arbres de racines 4 et 7 sont fusionnés entre eux, puis les arbres de racines 9 et 5 sont fusionnés entre eux. L'arbre de racine 13 étant seul, il reste inchangé. Lors de la deuxième étape de la fusion par paires, on fusionne les arbres ainsi obtenus de droite à gauche. Les arbres de racine 5 et 13 sont fusionnés, puis le résultat de cette fusion est fusionné avec l'arbre de racine 4.

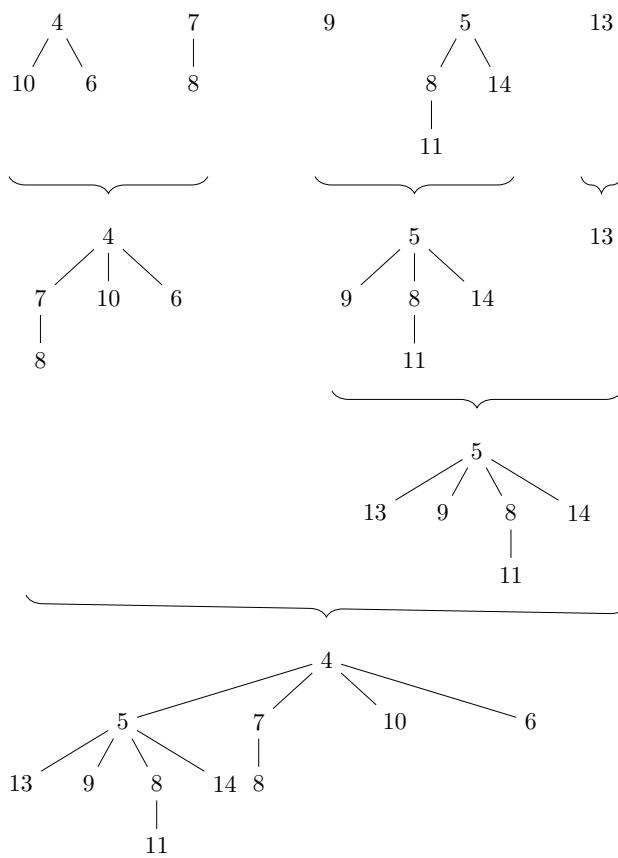


Figure 2 – Fusion par paires des enfants de la racine d'étiquette 2 du tas de la figure 1.

- Q4.** On insère successivement et dans cet ordre les éléments 0, 1, 2, 3, 4, 5 et 6 dans un tas initialement vide, puis on supprime deux fois le minimum. Représenter le tas d'appariement obtenu.
- Q5.** Écrire une fonction `fusion_par_paires : 'a arbre list -> 'a arbre` qui réalise la fusion par paires d'une liste d'arbres. Cette fonction devra renvoyer `Vide` pour une liste vide.
- Q6.** En déduire une fonction `suppression_min : 'a arbre -> 'a arbre` qui supprime l'élément minimal d'un tas. Quelle est sa complexité dans le pire cas ?

On admet qu'une séquence quelconque de n opérations `min_valeur`, `insere` et `suppression_min` sur cette structure de données, à partir d'un tas initialement vide, a une complexité totale en $\mathcal{O}(n \log n)$.

I.2 – Tri par tas

On peut utiliser les fonctions précédentes pour implémenter un tri par tas de complexité $\mathcal{O}(n \log n)$ pour trier une liste de n éléments.

- Q7.** Écrire une fonction `tri : 'a list -> 'a list` permettant de trier une liste dans l'ordre croissant en utilisant un tri par tas d'appariement.
- Q8.** Quelle est la complexité dans le meilleur cas de cette fonction `tri` ? Donner, en justifiant, une forme de liste correspondant à cette complexité.

II – Optimums de Pareto

Dans un problème d'optimisation classique, on cherche généralement à minimiser une certaine quantité parmi un ensemble de solutions possibles. Par exemple, on peut vouloir déterminer le trajet le plus court parmi tous les trajets possibles entre deux points. Lorsque l'on dispose de plusieurs objectifs à minimiser simultanément, ce problème devient cependant bien plus délicat : certains objectifs peuvent être contradictoires et il peut être impossible de les minimiser tous simultanément.

Considérons l'exemple de la figure 3, correspondant à la recherche d'un billet de train entre la gare de Lyon Part Dieu et la gare de Tours sur le réseau ferroviaire national. Pour chaque trajet possible, on retient dans cet exemple trois critères : sa durée, le nombre de correspondances (c'est-à-dire le nombre de fois où l'on doit descendre d'un train pour continuer le trajet avec un autre) et son prix. Idéalement, on souhaite obtenir à la fois un trajet dont la durée est minimale, qui comporte le moins de correspondances possibles et qui est le moins cher.

On observe sur la figure 3 qu'il n'est pas possible de minimiser simultanément tous ces critères. En revanche, il est clairement inutile de considérer le trajet (e) qui a la même durée et le même nombre de correspondances que le trajet (d), mais qui est strictement plus cher que celui-ci.

identifiant	durée (h)	nombre de correspondances	prix (€)
(a)	6	0	60
(b)	5	1	50
(c)	4	2	110
(d)	5	1	40
(e)	5	1	90
(f)	5	2	100
(g)	4	1	120

Figure 3 – Une liste de trajets proposés par un moteur de recherche d'itinéraires pour aller d'une gare à une autre.

On ne sait pas *a priori* quels seront les critères privilégiés par le voyageur. L'objectif est alors de lui présenter l'ensemble des solutions minimales, au sens où aucune autre solution n'est strictement meilleure : il s'agit des *optimums de Pareto*. Par exemple, dans la figure 3 on retiendrait les quatre trajets (a), (c), (d) et (g), qui sont ici les quatre optimums de Pareto. Remarquons que le trajet (a) minimise le nombre de correspondances, que le trajet (d) minimise le prix et que les deux trajets (c) et (g) minimisent la durée, sans pour autant qu'aucun des deux ne soit meilleur que l'autre sur les deux autres attributs.

- Q9.** Montrer que les trajets (b) et (f) ne sont pas des optimums de Pareto, en explicitant pour chacun d'entre eux un trajet strictement meilleur.

II.1 – Notion d'optimum de Pareto et ordre partiel

On rappelle qu'une *relation d'ordre* \preceq sur un ensemble E est une relation binaire réflexive ($\forall x \in E, x \preceq x$), antisymétrique ($\forall (x, y) \in E^2, x \preceq y \wedge y \preceq x \Rightarrow x = y$) et transitive ($\forall (x, y, z) \in E^3, x \preceq y \wedge y \preceq z \Rightarrow x \preceq z$). Une relation d'ordre est *totale* si tous les éléments sont deux à deux comparables, c'est-à-dire si $\forall (x, y) \in E^2, x \preceq y \vee y \preceq x$. On dit que l'ordre est *partiel* sinon. On note \prec la relation d'ordre stricte associée, définie par : $x \prec y$ si $x \preceq y \wedge x \neq y$.

Si $X \subseteq E$ est une partie de E , un *élément minimal* de X est un élément $x \in X$ qui n'est strictement supérieur à aucun autre élément de X . On note $\min(X)$ l'ensemble des éléments minimaux de X , c'est-à-dire :

$$\min(X) = \{x \in X \mid \nexists y \in X, y \prec x\} = \{x \in X \mid \forall y \in X, y \preceq x \Rightarrow y = x\}.$$

Notons que si la relation d'ordre \prec n'est pas totale, il peut y avoir plusieurs éléments minimaux, qui sont alors deux à deux non comparables.

Soient $d \in \mathbb{N}^*$ et d ensembles totalement ordonnés $(A_1, \leq_1), (A_2, \leq_2), \dots, (A_d, \leq_d)$, appelés attributs. Pour $i \in \llbracket 1, d \rrbracket$, \leq_i est donc un ordre total sur A_i . On peut munir $E = A_1 \times A_2 \times \dots \times A_d$ de différentes relations d'ordre construites à partir des ordres \leq_i .

Ordre lexicographique On définit l'ordre lexicographique \sqsubseteq sur E par $(x_1, x_2, \dots, x_d) \sqsubseteq (x'_1, x'_2, \dots, x'_d)$ si et seulement si $(x_1, x_2, \dots, x_d) = (x'_1, x'_2, \dots, x'_d)$ ou bien, en notant $k = \min\{i \in \llbracket 1, d \rrbracket \mid x_i \neq x'_i\}$, $x_k <_k x'_k$. L'ordre lexicographique est un ordre total sur E .

Ordre produit On définit l'ordre produit \preceq par $(x_1, x_2, \dots, x_d) \preceq (x'_1, x'_2, \dots, x'_d)$ si $\forall i \in \llbracket 1, d \rrbracket, x_i \leq_i x'_i$. En général, l'ordre produit est seulement un ordre partiel sur E . Un optimum de Pareto est un élément minimal pour cet ordre.

- Q10.** Soit X une partie de l'ensemble E telle que X admet un minimum pour l'ordre lexicographique. Montrer que cet élément minimum pour l'ordre lexicographique est un optimum de Pareto sur X , c'est-à-dire un élément minimal de X pour l'ordre produit.

Pour simplifier, on suppose dans toute la suite de cette partie que $\forall i \in \llbracket 1, d \rrbracket, (A_i, \leq_i) = (\mathbb{Z}, \leq)$ pour l'ordre total \leq usuel sur les entiers. On s'intéresse donc, dans cette partie, à l'ordre partiel produit sur \mathbb{Z}^d .

II.2 – Cas de la dimension 1

Dans le cas de la dimension $d = 1$, l'ordre produit coïncide avec l'ordre total \leq sur \mathbb{Z} . Toute partie finie non vide de \mathbb{Z} admet exactement un optimal de Pareto qui est le plus petit élément de cette partie.

Q11. Écrire une fonction `element_minimal : int list -> int` qui renvoie l'élément minimal d'une liste non vide d'entiers.

II.3 – Cas de la dimension 2

Dans cette partie, on s'intéresse à des éléments de \mathbb{Z}^2 ordonnés avec l'ordre produit \preceq . On manipule donc des objets du type :

```
type tuple = int * int
```

On représente un ensemble de couples par une liste sans doublons. On considère l'exemple suivant :

```
let ex1 = [(4, 1); (5, 7); (3, 3); (1, 3); (2, 9); (1, 4); (2, 2)]
```

Q12. Donner, sans justification, le ou les éléments minimaux de l'ensemble représenté par la liste de couples `ex1`.

Q13. Écrire une fonction `inferieur : tuple -> tuple -> bool` telle que `inferieur x y` pour deux couples $(x, y) \in (\mathbb{Z}^2)^2$ s'évalue à `true` si $x \preceq y$ et `false` sinon, c'est-à-dire si $y \prec x$ ou si x et y ne sont pas comparables.

Q14. On considère $X \subseteq \mathbb{Z}^2$ un ensemble qui contient $n \geq 1$ couples d'entiers. Donner, en justifiant, un encadrement de $\text{card}(\min(X))$, dont les bornes sont atteintes.

On suppose disposer d'une fonction de tri de signature `tri_lexico : tuple list -> tuple list` qui permet de trier une liste d'éléments dans l'ordre croissant pour l'ordre lexicographique. On suppose que cette fonction termine toujours et que la complexité de cette fonction de tri est en $\mathcal{O}(n \log n)$ sur une liste de taille n . La fonction réalisée dans la partie I convient à cet effet.

On considère la fonction suivante, de type `tuple list -> tuple list`, associée à la spécification : si `ens` est une liste de couples représentant un ensemble X d'éléments de \mathbb{Z}^2 , l'appel `elements_minimaux ens` s'évalue en une liste contenant exactement les éléments minimaux de X .

```
1 let elements_minimaux ens =
2   let rec w u =
3     match u with
4     | x :: y :: z ->
5       if inferieur x y then w (x :: z)
6       else x :: w (y :: z)
7     | v -> v
8   in w (tri_lexico ens)
```

Les noms de variables de cette fonction ont été volontairement rendus inintelligibles.

Pour représenter la trace d'exécution d'un appel à une fonction, on note ligne par ligne dans l'ordre chronologique d'exécution : $\rightarrow f \ args$ pour indiquer qu'on commence un appel à `f` sur les paramètres `args`; ou $\leftarrow \ valeur$ de `retour` pour indiquer la fin de l'appel et la valeur renvoyée. On indente les appels imbriqués. Voici un exemple de fonction et la trace associée à l'appel `cardinal [32; 52]` :

```
let rec cardinal l =           -> cardinal [32; 52]
  match l with                  -> cardinal [52]
  | [] -> 0                      -> cardinal []
  | _ :: tl -> 1 + cardinal tl    <- 0
                                    <- 1
                                    <- 2
```

Q15. Donner la trace d'exécution de la fonction `w` lors de l'appel à `elements_minimaux ex1`. On pourra se contenter de noter uniquement le début de la liste à chaque appel s'il n'y a pas ambiguïté sur la valeur de la fin, par exemple `ex1 = [(4, 1); (5, 7); ...]`.

Q16. Justifier soigneusement la terminaison de la fonction `elements_minimaux`.

Q17. Déterminer la complexité de la fonction `elements_minimaux`.

Q18. Justifier soigneusement la correction de la fonction `elements_minimaux`.

II.4 – Cas général

On se place désormais dans le cadre de la dimension $d \in \mathbb{N}^*$ et même, plus généralement, dans le cadre d'un ordre partiel quelconque sur un ensemble. On suppose ainsi uniquement disposer d'un type abstrait `element` et d'une fonction `inferieur` : `element -> element -> bool` qui détermine si un élément est inférieur ou égal à un autre pour cet ordre partiel.

Si (E, \preceq) est un ensemble partiellement ordonné et si $X \subseteq E$, la *largeur* de X est le nombre maximal d'éléments simultanément deux à deux non comparables de X ; c'est-à-dire le cardinal maximal d'une partie de X composée uniquement d'éléments deux à deux non comparables :

$$\text{largeur}(X) = \max\{\text{card}(Y) \mid Y \subseteq X \text{ et } \forall(x, y) \in Y^2, x \neq y \Rightarrow x \not\preceq y \wedge y \not\preceq x\}.$$

Q19. Écrire une fonction `existe_plus_petit` de signature `element -> element list -> bool` telle que l'appel `existe_plus_petit u liste` s'évalue à `true` si et seulement s'il existe un élément de la liste inférieur ou égal à l'élément `u`.

Q20. Écrire une fonction `ajoute_plus_petit` de signature `element -> element list -> element list` telle que `ajoute_plus_petit u liste` ajoute l'élément `u` à une liste en supprimant de cette liste tous les éléments supérieurs ou égaux à `u`.

Q21. En déduire une fonction `elements_minimaux` : `element list -> element list` qui renvoie une liste composée des éléments minimaux d'une liste passée en paramètre. On suppose que la liste passée en paramètres ne comporte pas de doublons et représente un ensemble X d'éléments. Cette fonction doit effectuer au plus $\mathcal{O}(nk)$ appels à la fonction `inferieur`, où n est le cardinal de X et k la largeur de X , ce que l'on justifiera brièvement.

III – Éléments minimaux d'un langage régulier

Dans cette partie, on se propose de généraliser la notion d'optimum de Pareto dans le cas d'un nombre variable d'attributs. On pourra ainsi être amené à comparer des tuples de longueurs différentes. À cet effet, on modélise un tuple par un mot sur un alphabet fini.

III.1 – Éléments minimaux d'un langage

On considère un alphabet Σ fini non vide muni d'un ordre total \leq sur les lettres. On rappelle que Σ^* désigne l'ensemble de tous les mots sur Σ et que $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ désigne l'ensemble des mots de longueur non nulle sur Σ .

On considère la relation d'ordre \preceq sur Σ^* définie par : $u_1 u_2 \dots u_n \preceq v_1 v_2 \dots v_m$ si et seulement si $n \leq m$ et $\forall i \in \llbracket 1, n \rrbracket, u_i \leq v_i$. Si Σ comporte au moins deux lettres, cette relation d'ordre est seulement partielle. On remarquera que la restriction de \preceq aux mots de Σ^d , c'est-à-dire aux tuples de longueur d fixée, correspond à l'ordre partiel étudié dans la partie précédente.

Pour les exemples, on considérera l'alphabet $\Sigma_3 = \{a, b, c\}$, avec $a < b < c$.

Pour Σ_3 , on a $ab \prec bc \prec bca \prec ccab \prec ccbca \prec cccbcb$. En revanche ab, ca, bac et $aaab, babab$ sont deux à deux incomparables.

On note $\min(L)$ le langage des éléments minimaux de L , c'est-à-dire : $\min(L) = \{u \in L \mid \nexists v \in L, v \prec u\}$.

Par exemple, $\min(\Sigma_3^*) = \{\varepsilon\}$, $\min(\Sigma_3^+) = \{a\}$, $\min(\{ab, bc, ca, bac, ccab\}) = \{ab, ca, bac\}$.

Q22. Montrer que $\varepsilon \in L \iff \min(L) = \{\varepsilon\}$.

Il est ainsi trivial de déterminer les éléments minimaux d'un langage contenant ε . **On supposera dans toute la suite que les langages considérés ne contiennent pas ε .**

Q23. Montrer que si $L \subseteq \Sigma^+$ est non vide, alors $\min(L) \neq \emptyset$. Autrement dit, l'ordre partiel \preceq est bien fondé.

On considère, sur Σ_3 , les langages $L_1 = \{a^i b^j \mid (i, j) \in (\mathbb{N}^*)^2\}$ et $L_2 = \{a^i b^j c^k \mid (i, j, k) \in (\mathbb{N}^*)^3, i \leq k \wedge j \leq k\}$.

Q24. Donner $\min(L_1)$ sans justification puis montrer que les langages L_1 et $\min(L_1)$ sont réguliers.

Q25. (a) Déterminer $\min(L_2)$, en expliquant brièvement comment ce langage est obtenu.

(b) Montrer que les langages L_2 et $\min(L_2)$ ne sont pas réguliers. On détaillera uniquement la preuve montrant que L_2 n'est pas régulier, et on justifiera brièvement cette propriété pour $\min(L_2)$.

Q26. Montrer que $\min(L)$ peut être régulier même si $L \subseteq \Sigma^+$ n'est pas régulier.

On se propose dans la suite de cette partie de montrer que si $L \subseteq \Sigma^+$ est régulier alors $\min(L)$ est également régulier.

III.2 – Automates finis non déterministes

Un *automate* sur un alphabet Σ est un quadruplet $\mathcal{A} = (Q, I, T, \Delta)$ avec Q un ensemble fini non vide, $I \subseteq Q$ un ensemble d'états initiaux, $T \subseteq Q$ un ensemble d'états terminaux et $\Delta \subseteq Q \times \Sigma \times Q$ une relation de transition. Un chemin d'un tel automate est une suite de transitions $((q_{i-1}, u_i, q_i))_{i \in [\![1, n]\!]} \in \Delta^n$, d'état de départ $q_0 \in Q$, d'état d'arrivée $q_n \in Q$ et d'étiquette $u = u_1 u_2 \dots u_n \in \Sigma^*$. Un mot $u \in \Sigma^*$ est reconnu par \mathcal{A} s'il existe un chemin d'un état initial à un état terminal dans \mathcal{A} . On note $\mathcal{L}(\mathcal{A})$ le langage des mots reconnus par \mathcal{A} . Un exemple d'automate est représenté sur la figure 4.

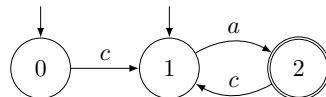


Figure 4 – L'automate \mathcal{A}_1 sur Σ_3 . Les états initiaux, ici 0 et 1, sont indiqués par une flèche et les états terminaux, ici 2, sont doublement cerclés.

Q27. Appliquer l'algorithme d'élimination des états à l'automate \mathcal{A}_1 de la figure 4, en éliminant successivement, dans cet ordre, les états 0, 1 puis 2. En déduire une expression régulière qui dénote le langage $\mathcal{L}(\mathcal{A}_1)$.

III.3 – Automates marqués

Un *automate marqué* sur un alphabet Σ est un couple $\mathcal{A}^M = (\mathcal{A}, M)$ formé d'un automate $\mathcal{A} = (Q, I, T, \Delta)$ sur Σ et d'un sous-ensemble $M \subseteq \Delta$ de transitions dites *marquées*. Un mot $u \in \Sigma^*$ est reconnu par un automate marqué \mathcal{A}^M s'il existe un chemin dans \mathcal{A} d'un état initial à un état terminal qui passe par au moins une transition marquée de M . On note $\mathcal{L}(\mathcal{A}^M)$ le langage des mots reconnus par un automate marqué \mathcal{A}^M .

Un exemple d'automate marqué \mathcal{A}_2^M est représenté sur la figure 5. Le langage $\mathcal{L}(\mathcal{A}_2^M)$ reconnu par cet automate marqué est le langage des mots de Σ_3^* d'au moins deux lettres qui commencent et terminent par a ou b et qui comportent au moins un a . On remarquera que même si l'état 0 est à la fois un état initial et un état terminal, le mot vide n'est pas reconnu par cet automate marqué, puisque l'on impose aux chemins de passer par au moins une transition marquée.

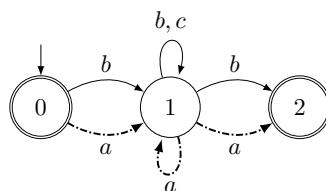


Figure 5 – Un automate marqué \mathcal{A}_2^M sur Σ_3 . Les états initiaux, ici seulement 0, sont indiqués par une flèche et les états terminaux, ici 0 et 2, sont doublement cerclés. Les transitions marquées, ici celles d'étiquette a , sont représentées par une ligne épaisse en traits et points.

On se propose de commencer par montrer que tout langage reconnaissable par un automate marqué est reconnaissable par un automate ordinaire.

Soit $\mathcal{A}^M = (\mathcal{A}, M)$ un automate marqué sur Σ avec $\mathcal{A} = (Q, I, T, \Delta)$. On construit un automate ordinaire $\ddot{\mathcal{A}}$ qui reconnaît le même langage que l'automate marqué \mathcal{A}^M . Pour cela on duplique l'automate \mathcal{A} , avec une première copie de \mathcal{A} , sans états terminaux, avec les mêmes transitions non marquées, mais dont les transitions marquées mènent vers une deuxième copie de \mathcal{A} , identique à \mathcal{A} mais sans états initiaux.

Formellement, on note $\ddot{Q} = \{\ddot{q} \mid q \in Q\}$ une copie des états de Q . L'ensemble des états de $\ddot{\mathcal{A}}$ est $\ddot{Q} \cup Q$. L'ensemble des états initiaux de $\ddot{\mathcal{A}}$ est $\ddot{I} = \{\ddot{q} \mid q \in I\}$ et l'ensemble des états terminaux de $\ddot{\mathcal{A}}$ est T . On note $\ddot{\Delta} = \{(\ddot{q}, x, q') \mid (q, x, q') \in M\} \cup \{(\ddot{q}, x, \ddot{q}') \mid (q, x, q') \in \Delta \setminus M\}$. Les transitions de $\ddot{\mathcal{A}}$ sont $\ddot{\Delta} \cup \Delta$.

Q28. Appliquer cette construction sur l'automate marqué \mathcal{A}_2^M de la figure 5 et représenter l'automate ordinaire $\ddot{\mathcal{A}}_2$ ainsi obtenu.

Q29. Montrer, de manière générale, que $\mathcal{L}(\mathcal{A}^M) = \mathcal{L}(\ddot{\mathcal{A}})$.

III.4 – Automate des minimaux

Soit $L \subseteq \Sigma^+$ un langage régulier. On note $\check{L} = \{u \in \Sigma^+ \mid \exists v \in L, v \prec u\}$ le langage des mots qui sont strictement plus grands qu'au moins un mot de L .

Q30. Montrer qu'il suffit de montrer que \check{L} est régulier pour montrer que $\min(L)$ est régulier.

On se propose de construire un automate marqué $\check{\mathcal{A}}^M$ qui reconnaît \check{L} à partir d'un automate \mathcal{A} qui reconnaît L .

Considérons un automate $\mathcal{A} = (Q, I, T, \Delta)$ qui reconnaît L .

- On commence par supprimer dans Δ toutes les transitions sortantes d'un état terminal de \mathcal{A} . Formellement, on note $\Delta' = \{(q, x, q') \mid (q, x, q') \in \Delta \text{ et } q \notin T\}$ l'ensemble des transitions de \mathcal{A} qui ne partent pas d'un état terminal ;
- Ensuite, pour chaque transition $(q, x, q') \in \Delta'$, on ajoute (si nécessaire) et on marque les transitions (q, y, q') , pour chaque lettre $y \in \Sigma$ telle que $x < y$. Formellement, on note $M' = \{(q, y, q') \mid \exists (q, x, q') \in \Delta', \exists y \in \Sigma, x < y\}$;
- Enfin, on ajoute et on marque les transitions (q, x, q) pour chaque état terminal $q \in T$ et pour chaque lettre $x \in \Sigma$. Formellement on note $M'' = \{(q, x, q) \mid q \in T, x \in \Sigma\}$.

On pose $\check{\Delta} = \Delta' \cup M' \cup M''$ et $M = M' \cup M''$. On note $\check{\mathcal{A}} = (Q, I, T, \check{\Delta})$ et $\check{\mathcal{A}}^M = (\check{\mathcal{A}}, M)$ l'automate marqué obtenu.

Q31. Appliquer cette construction sur l'automate \mathcal{A}_1 de la figure 4, avec l'alphabet Σ_3 , et représenter l'automate marqué $\check{\mathcal{A}}_1^M$ ainsi obtenu.

Q32. Montrer, de manière générale, que $\check{L} \subseteq \mathcal{L}(\check{\mathcal{A}}^M)$.

Q33. Montrer, de manière générale, que $\mathcal{L}(\check{\mathcal{A}}^M) \subseteq \check{L}$.

Q34. En déduire que si $L \subseteq \Sigma^+$ est un langage régulier alors $\min(L)$ est également un langage régulier.

IV – Meilleurs chemins dans un graphe

On s'intéresse à présent à la recherche d'itinéraires dans un réseau ferroviaire, afin d'optimiser différents critères pour un trajet. Le réseau ferroviaire est modélisé par un graphe orienté $G = (V, E)$ dont les noeuds, dans V , sont les gares. Il existe un arc d'une gare $v_1 \in V$ vers une gare $v_2 \in V$ lorsqu'il est possible de faire le trajet directement de v_1 à v_2 sans changer de train.

Chaque arc $e \in E$ est étiqueté par un couple d'entiers positifs $(t(e), p(e))$ où $t(e)$ donne le temps de trajet de la gare de départ de e vers la gare d'arrivée de e , et $p(e)$ donne le prix du billet à payer pour prendre le train représenté par e . Un exemple d'un tel réseau est donné en figure 6. On remarquera que les trajets obtenus dans la figure 3 correspondent à certains des chemins entre le sommet $s = 0$ et le sommet $t = 5$.

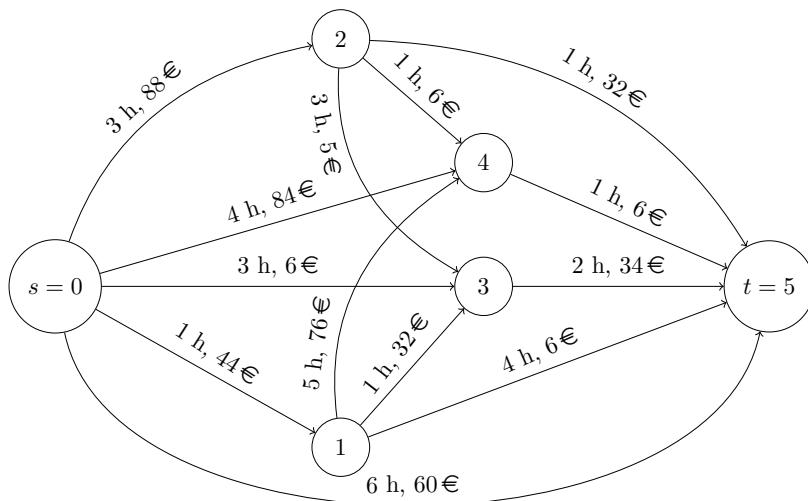


Figure 6 – Un exemple de réseau ferroviaire.

Soit $\mathcal{C} = (e_1, e_2, \dots, e_T)$ un chemin orienté dans G , donné par la liste de ses arcs. Le *poids* de ce chemin, noté $w(\mathcal{C})$ est le triplet :

$$w(\mathcal{C}) = \left(\sum_{k=1}^T t(e_k), T - 1, \sum_{k=1}^T p(e_k) \right).$$

La deuxième composante du triplet donne le nombre de correspondances, c'est-à-dire de changements de trains. Les poids sont représentés par un triplet d'entiers, qui sont ordonnés en OCAML selon l'ordre lexicographique :

```
type poids = int * int * int
```

On définit l'opération \oplus sur les triplets d'entiers par : $(T, C, P) \oplus (T', C', P') = (T + T', C + C', P + P')$. On rappelle cependant que cet opérateur n'est pas défini *a priori* par le langage OCAML.

On représente un arc du graphe par un enregistrement de type **arc** :

```
type arc = {dest : int; temps : int; prix : int}
```

Ainsi, si **train** est une valeur de type **arc** représentant un arc e , l'expression **train.dest** donne le numéro de la gare d'arrivée de e , l'expression **train.temps** permet d'obtenir $t(e)$ et l'expression **train.prix** permet d'obtenir $p(e)$.

On représente les gares du réseau ferroviaire par des entiers de 0 à $n - 1$, où n est le nombre total de gares. Le graphe du réseau ferroviaire est représenté par ses listes d'adjacence :

```
type reseau_ferroviaire = arc list array
```

Si **reseau** est une valeur de type **reseau_ferroviaire** et i un entier entre 0 et $n - 1$, alors **reseau.(i)** est la liste des arcs partant du sommet i , dans un ordre arbitraire.

L'algorithme de Dijkstra permet, dans un graphe orienté dont les arcs sont pondérés par des entiers positifs, de calculer pour un sommet s donné les longueurs des plus courts chemins de s à tout autre sommet du graphe. On adapte cet algorithme de la façon suivante, afin qu'il calcule, dans un réseau ferroviaire, les trajets dont les poids sont des optimums de Pareto.

Algorithme 1 Algorithme de Dijkstra pour les réseaux ferroviaires

```

1 : Initialiser un tableau  $d$  qui à chaque sommet associe la liste vide []
2 : Soit  $Q$  une file de priorité vide
3 : Ajouter  $s$  à  $Q$  avec priorité  $(0, -1, 0)$ 
4 : tant que  $Q$  n'est pas vide faire
5 :   Défiler  $v$  de  $Q$  de priorité  $\pi_v$  minimale pour l'ordre lexicographique
6 :   si aucun élément de  $d[v]$  n'est plus petit que  $\pi_v$  alors
7 :     Supprimer de  $d[v]$  tous les éléments plus grands que  $\pi_v$ 
8 :     Ajouter  $\pi_v$  dans  $d[v]$ 
9 :     pour chaque arc  $e$  de  $v$  vers  $w$  faire
10 :      Enfiler  $w$  dans  $Q$  avec la priorité  $\pi_v \oplus (t(e), 1, p(e))$ 
11 :    fin pour
12 :  fin si
13 : fin tant que
```

Dans cette version de l'algorithme de Dijkstra, un sommet peut être présent plusieurs fois dans la file.

Q35. Les files de priorité de la partie I stockent *a priori* uniquement les priorités. Justifier qu'on peut les utiliser pour implémenter l'algorithme 1 si on stocke dans chaque noeud le couple (π_v, v) .

On implémente le tableau **d** par un objet de type :

```
d : poids list array
```

On considère le réseau de transport de la figure 6. Après quatre itérations de la boucle de l'algorithme 1, avec $s = 0$, la file de priorité Q contient les valeurs suivantes :

Priorité	(3, 0, 88)	(4, 0, 84)	(4, 2, 110)	(5, 1, 40)	(5, 1, 50)	(6, 0, 60)	(6, 1, 120)
Sommet	2	4	5	5	5	5	4

et le tableau **d** contient les valeurs suivantes :

d.(0)	d.(1)	d.(2)	d.(3)	d.(4)	d.(5)
[(0, -1, 0)]	[(1, 0, 44)]	[]	[(3, 0, 6); (2, 1, 76)]	[]	[]

Q36. Détails l'évolution de Q et **d** lors des 3 itérations suivantes de l'algorithme 1.

Q37. Écrire une fonction `dijkstra_pareto : reseau_ferroviaire -> int -> poids list array` qui prend en paramètres un graphe orienté décrit par ses listes d'adjacence, un sommet d'origine s et qui calcule la liste des poids optimaux de Pareto des chemins de s à tous les autres sommets du graphe. On pourra librement utiliser toutes les fonctions définies dans les parties précédentes, en particulier celles de la partie I.1 ainsi que celles de la partie II.4 en supposant que le type `element` correspond ici aux triplets donnant les poids.

Q38. Montrer qu'un chemin non élémentaire d'un sommet u à un sommet v , c'est-à-dire qui passe deux fois par un même sommet, a un poids qui n'est pas un optimum de Pareto parmi les poids des chemins de u à v .

Q39. Soient s et t deux sommets du graphe et $\mathcal{C}_{s,t}$ un chemin de s à t , dont le poids est un optimum de Pareto parmi tous les chemins de s à t . Montrer que le poids de tout chemin $\mathcal{C}_{s,u}$, allant de s à u , préfixe de $\mathcal{C}_{s,t}$, est un optimum de Pareto parmi les poids des chemins de s à u .

Q40. Montrer que, dans l'algorithme 1, si π_v satisfait la condition du test en ligne 6, alors π_v est un optimum de Pareto parmi les poids des chemins de s à v . En déduire une optimisation de l'algorithme 1.

Q41. Démontrer la correction de l'algorithme, c'est-à-dire : d'une part qu'à la fin de l'exécution, pour tout sommet v , $d[v]$ contient exactement tous les poids des chemins de s à v optimaux de Pareto ; et d'autre part que l'algorithme termine.

Q42. Proposer une fonction en OCAML permettant d'obtenir la liste de tous les chemins entre un sommet s et un sommet t dont le poids est un optimum de Pareto. On indiquera la signature de la fonction ainsi qu'une rapide description de la structure utilisée.

A – Annexe : documentation OCaml

I.1 – Fonctions usuelles

- `val min : 'a -> 'a -> 'a` renvoie le minimum de ses deux paramètres. Lorsque les paramètres sont des tuples, l'ordre utilisé est l'ordre lexicographique.
- `val max : 'a -> 'a -> 'a` renvoie le maximum de ses deux paramètres. Lorsque les paramètres sont des tuples, l'ordre utilisé est l'ordre lexicographique.

I.2 – Fonctions sur les listes

- `val List.for_all : ('a -> bool) -> 'a list -> bool.`
L'appel `List.for_all f [a0; a1; ...; an]` renvoie `true` lorsque tous les appels `f a0, f a1, ..., f an` renvoient `true`, et `false` sinon.
- `val List.exists : ('a -> bool) -> 'a list -> bool.`
L'appel `List.exists f [a0; a1; ...; an]` renvoie `true` lorsqu'au moins l'un des appels `f a0, f a1, ..., f an` renvoie `true`, et `false` sinon.
- `val List.filter : ('a -> bool) -> 'a list -> 'a list.`
L'appel `List.filter f [a0; a1; ...; an]` renvoie la liste extraite de `[a0; a1; ...; an]` formée uniquement des éléments x tels que $f x$ est `true`. Les éléments apparaissent dans la liste résultat dans le même ordre que dans la liste en paramètre.
- `val List.iter : ('a -> unit) -> 'a list -> unit.`
L'appel `List.iter f [a0; a1; ...; an]` exécute `f a0`, puis `f a1`, etc., jusqu'à `f an`, dans cet ordre.

I.3 – Manipulation des tableaux

- `val Array.make : int -> 'a -> 'a array.`
L'appel `Array.make n x0` crée un tableau de n entrées, chacune initialisée à la valeur $x0$.
- `val Array.length : 'a array -> int` renvoie la longueur d'un tableau.
- Si t est un tableau et i un indice valide dans le tableau, $t.(i) <- x0$ modifie la case d'indice i en lui affectant la valeur $x0$.

SESSION 2025**MPI5IN**

ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE MPI

INFORMATIQUE

Durée : 4 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

RAPPEL DES CONSIGNES

- Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.
 - Ne pas utiliser de correcteur.
 - Écrire le mot FIN à la fin de votre composition.
-

Les calculatrices sont interdites.

Le sujet est composé de trois parties, toutes indépendantes.

Partie I - Grammaire non contextuelle

Soit la grammaire non contextuelle $G = (\Sigma, V, P, S)$, l'ensemble des règles de production P étant défini par :

$$\begin{aligned} S &\rightarrow SaS \mid A \\ A &\rightarrow AbA \mid B \\ B &\rightarrow BcB \mid \varepsilon \end{aligned}$$

où Σ (respectivement V) est l'ensemble de symboles terminaux (respectivement non terminaux), $S \in V$ est le symbole initial de G et ε dénote le mot vide.

Soit u le mot abc .

Q1. Donner une dérivation à gauche de u . Que peut-on en déduire sur le mot u ?

Q2. Donner deux arbres de dérivation pour le mot aa . En déduire que G est ambiguë.

Une grammaire est dite *récursive gauche directe* si elle possède une règle de la forme $A \rightarrow A\alpha$, où α est une suite de symboles terminaux et non terminaux. A est dite *variable récursive gauche*. Une telle grammaire pose divers problèmes en analyse syntaxique descendante, on cherche alors à éliminer cette récursivité. Ceci est toujours possible pour les langages réguliers.

Q3. Identifier dans G les variables récursives gauches.

Pour éliminer la récursivité gauche, on utilise l'algorithme suivant :

Soit $A \rightarrow A\alpha \mid \beta$ une règle, où α est une suite de symboles terminaux et non terminaux et β est une suite de symboles terminaux et non terminaux ne commençant pas par A .

On remplace cette règle par :

- (i). une règle commençant par A : $A \rightarrow \beta A'$,
- (ii). une règle pour une nouvelle variable A' : $A' \rightarrow \alpha A' \mid \varepsilon$.

Q4. Construire la grammaire non récursive gauche directe G' équivalente à G .

Q5. Prouver que le langage reconnu par G' (ou G) est $\{a, b, c\}^*$.

Partie II - Problème de bin-packing

Cette partie comporte des questions nécessitant un code OCaml.

Soient $n, k \in \mathbb{N}^2$. On appelle *rangement* de n objets dans k boîtes une fonction $\mathcal{R} : \llbracket 1, n \rrbracket \rightarrow \llbracket 1, k \rrbracket$. L'ensemble $B_j = \mathcal{R}^{-1}(j) = \{i \in \llbracket 1, n \rrbracket, \mathcal{R}(i) = j\}$ est le contenu de la boîte $j \in \llbracket 1, k \rrbracket$.

Étant données des tailles $a = (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0, 1])^n$, a_i étant la taille de l'objet i , on dit qu'un rangement \mathcal{R} de n objets dans k boîtes est *valide* pour les tailles a si pour tout $j \in \llbracket 1, k \rrbracket$ on a $\sum_{i \in B_j} a_i \leq 1$.

On considère alors le problème de décision BIN-PACKING qui, étant donnés $n, k \in \mathbb{N}^2$ et $a \in (\mathbb{Q} \cap]0, 1])^n$, décide s'il existe un rangement \mathcal{R} de n objets dans k boîtes valide pour les tailles a .

On peut décrire une solution de ce problème par un couple (k, \mathcal{R}) .

Q6. Montrer que BIN-PACKING est dans NP.

Soit le problème PARTITION suivant :

Pour n entiers $c_1 \cdots c_n$, existe-t-il un sous-ensemble S de $\llbracket 1, n \rrbracket$, tel que $\sum_{i \in S} c_i = \sum_{i \notin S} c_i$?

On admet que le problème PARTITION est NP-complet.

Q7. Montrer par réduction depuis PARTITION que le problème BIN-PACKING est NP-complet.

On s'intéresse par la suite au problème MIN-BIN-PACKING qui demande de déterminer le plus petit k pour lequel il existe un rangement \mathcal{R} de n objets de taille a dans k boîtes. On introduit pour résoudre ce problème une heuristique dite *Premier Casier Décroissant* (**algorithme 2**), se basant sur l'heuristique *Premier Casier* décrite dans l'**algorithme 1**. Dans cet algorithme, le rangement \mathcal{R} est modélisé par une liste de boîtes \mathcal{L} .

Algorithme 1 - Algorithme Premier Casier

Entrées : $n \in \mathbb{N}, (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0, 1])^n$

Sorties : Une solution (k, \mathcal{R})

début

\mathcal{L} : liste de boîtes ouvertes

$\mathcal{L} = \emptyset$

pour $i=1$ à n **faire**

 Rechercher la première boîte de \mathcal{L} dans laquelle l'objet i peut être placé

si une telle boîte existe alors

 l'objet i est placé à l'intérieur

sinon

 une nouvelle boîte est ajoutée à \mathcal{L} et l'objet i y est placé

$k := \text{longueur}(\mathcal{L})$

 Déduire \mathcal{R} de \mathcal{L} .

Algorithme 2 - Algorithme Premier Casier Décroissant

Entrées : $n \in \mathbb{N}, (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0, 1])^n$

Sorties : Une solution (k, \mathcal{R})

début

$\sigma = \text{permutation telle que } a_{\sigma(1)} \geq a_{\sigma(2)} \cdots \geq a_{\sigma(n)}$

$k, \mathcal{R}_1 := \text{Premier Casier}(n, a_{\sigma(1)}, \dots, a_{\sigma(n)})$

pour $i = 1$ à n **faire**

$\mathcal{R}(i) := \mathcal{R}_1(\sigma(i))$

Nous allons montrer que l'algorithme *Premier Casier Décroissant* est une $\frac{3}{2}$ -approximation pour le problème MIN-BIN-PACKING.

On note dans la suite k le nombre de boîtes rentrées par l'**algorithme 2**, k^* le nombre de boîtes optimal (défini comme le nombre minimal de boîtes permettant de répondre au problème MIN-BIN-PACKING) et $k_1 = \lceil 2k/3 \rceil$, où $\lceil . \rceil$ désigne la partie entière supérieure. B_{k_1} fait référence à la boîte déduite du rangement renvoyé par l'**algorithme 2**.

Q8. Montrer que si B_{k_1} contient un objet i de taille $a_i > \frac{1}{2}$, alors $k^* \geq k_1 \geq \frac{2}{3}k$.

Q9. Montrer que sinon, les boîtes $B_{k_1} \cdots B_k$ contiennent au moins $2(k - k_1) + 1$ objets, aucun d'eux ne pouvant rentrer dans les boîtes $B_1 \cdots B_{k_1-1}$.

Q10. Justifier que $\sum_{i=1}^n a_i > \min(k_1 - 1, 2(k - k_1) + 1)$. En admettant que pour $k \in \mathbb{N}$, $\frac{2}{3}k + \frac{2}{3} \geq \lceil 2k/3 \rceil$, en déduire que $k^* \geq \left\lceil \frac{2}{3}k \right\rceil \geq \frac{2}{3}k$.

On définit des types record pour représenter les objets et les boîtes en OCaml :

```
type objet = {
  id : int; (*identifiant de la boîte *)
  taille : float; (*taille de la boîte *)
}

type boîte = {
  charge : float; (*somme des tailles des objets de la boîte *)
  objets : objet list; (*liste courante des tailles des objets dans la boîte *)
}
```

Q11. Écrire une constante boîte_vide : boîte représentant une boîte vide.

Q12. Écrire une fonction de signature

ajoute_boîte : objet -> boîte -> boîte
qui renvoie la boîte obtenue en ajoutant un objet à une boîte donnée.

Q13. Écrire une fonction récursive de signature

trouve_boîte : boîte list -> objet -> int
telle que trouve_boîte b1 o retourne l'indice (à partir de 0) de la première boîte dans b1 pouvant contenir o ou la taille de la liste b1 si aucune boîte ne peut contenir o.

Q14. Écrire une fonction récursive de signature

transforme : 'a -> ('a -> 'a) -> int -> 'a list -> 'a list
telle que transforme d f i l retourne la liste l où l'élément x à l'indice i (à partir de 0) a été remplacé par f x. Si i est plus grand que la taille de la liste, f d est ajouté à la fin.

Q15. Écrire une fonction de signature

premier_casier : objet list -> boîte list
qui applique l'**algorithme 1**. On retournera directement la liste de boîtes, on ne cherchera pas à calculer la fonction \mathcal{R} . On pourra utiliser la fonction transforme précédemment écrite.

Q16. Écrire une fonction de signature

premier_casier_decroissant : objet list -> boîte list
qui applique l'**algorithme 2**.
On pourra utiliser List.sort : ('a -> 'a -> int) -> 'a list -> 'a list qui trie la liste passée en argument dans l'ordre croissant suivant la fonction passée en argument (qui renvoie -1, 0 ou 1 pour inférieur, égal ou supérieur respectivement). On pourra utiliser la fonction de comparaison générique compare : 'a -> 'a -> int. Là encore, on retournera directement la liste des boîtes.

Partie III - Algorithmes de couplage

Cette partie comporte des questions nécessitant un code en C.

On s'intéresse ici à un problème d'appariement entre deux groupes d'individus U et V , que l'on suppose de même cardinalité n .

On cherche à appairer les éléments de V aux éléments de U , chaque élément devant être apparié à exactement un élément de l'autre ensemble. Chaque élément de V (respectivement de U) a, de plus, un ordre de préférence total entre tous les éléments de U (respectivement de V).

Définition 1 (Couplage, couplage parfait)

Un ensemble de paires $A \subseteq V \times U$ est un couplage si tout $x \in V$ et tout $y \in U$ apparaissent dans au plus un élément de A. Le couplage est dit parfait si tout $x \in V$ et tout $y \in U$ apparaissent dans un et un seul élément de A.

Notations

Dans toute la suite on notera :

- $|E|$ le cardinal de l'ensemble E ,
- $>^x$ l'ordre associé à l'élément $x \in V \cup U$: si $u_1 \in U$ préfère strictement $v_1 \in V$ à $v_2 \in V$ alors $v_1 >^{u_1} v_2$. Par extension, on définit de même la notion de préférence \geq^x .
- $m_A(x)$ le partenaire de $x \in V \cup U$ dans le couplage parfait A .

Définition 2 (Couplage parfait stable)

Un couplage parfait A est dit stable si, pour tous couples (v_1, u_1) et (v_2, u_2) de A , il est impossible que l'on ait $u_2 >^{v_1} u_1$ et $v_1 >^{u_2} v_2$.

Définition 3 (Pareto-optimalité)

Soient A et A' deux couplages parfaits stables et $E \subseteq V \cup U$. On dit que A Pareto-domine A' sur E si et seulement si pour tout $x \in E$, $m_A(x) \geq^x m_{A'}(x)$ et il existe $x \in E$ tel que $m_A(x) >^x m_{A'}(x)$. Un couplage est Pareto-optimal s'il est stable et n'est pas Pareto dominé sur $V \cup U$.

En économie, la Pareto-optimalité d'un système exprime le fait que l'on ne peut améliorer la satisfaction d'un individu du système sans diminuer la satisfaction d'un autre.

III.1 - Recherche de couplages stables

L'algorithme de Gale-Shapley (**algorithme 3**) permet de montrer qu'il existe toujours au moins un couplage parfait stable et d'en trouver un de façon efficace. L'idée de l'algorithme est tout d'abord de choisir un des deux ensembles (ici pour illustration V). Les éléments de V font des propositions aux éléments de U . Quand un élément de V fait une proposition à un élément de U , celui-ci peut soit la refuser définitivement, soit l'accepter provisoirement en attendant une éventuelle meilleure offre. Toute nouvelle acceptation vaut rejet définitif des propositions précédemment acceptées.

Algorithme 3 - Algorithme de Gale-Shapley

Entrées : $n = |V| = |U|$, les listes de préférences des éléments de V et U

Sorties : Un couplage parfait stable A

début

$A := \emptyset$

tant que il existe un élément de V non apparié faire

 Choisir un tel v

$u :=$ élément de U que v préfère parmi ceux à qui il n'a pas encore proposé

si u n'est pas apparié dans A **alors**

$A := A \cup \{(v, u)\}$

sinon

si u est apparié dans A à v' ET $v >^u v'$ **alors**

$A := A \setminus \{(v', u)\}$

$A := A \cup \{(v, u)\}$

retourner A

Soit le jeu de données \mathcal{D} suivant : $V = \{v_1, v_2, v_3\}$, $U = \{u_1, u_2, u_3\}$ et :

V	U
$u_2 >^{v_1} u_1 >^{v_1} u_3$	$v_1 >^{u_1} v_3 >^{u_1} v_2$
$u_1 >^{v_2} u_2 >^{v_2} u_3$	$v_2 >^{u_2} v_1 >^{u_1} v_3$
$u_1 >^{v_3} u_2 >^{v_3} u_3$	$v_2 >^{u_3} v_1 >^{u_3} v_3$

Q17. Donner la trace de l'**algorithme 3** sur ces données. Le choix de v dans l'algorithme se fera par ordre croissant des indices.

Q18. Montrer que l'**algorithme 3** termine.

Q19. Donner, sans le prouver, un invariant de boucle permettant de prouver que l'**algorithme 3** retourne un couplage parfait.

Q20. Montrer que le couplage parfait calculé est stable.

Q21. Montrer que cet algorithme effectue moins de n^2 itérations de boucles.

Pour des préférences données, il peut exister plus d'un couplage parfait stable. Soient A et A' deux tels couplages pour un problème donné. On dira qu'un élément de $V \cup U$ préfère A à A' s'il n'a pas le même partenaire dans les deux couplages et s'il préfère celui de A à celui de A' .

Q22. Montrer que si v préfère le couplage A , alors son partenaire u dans A préfère A' .

On peut montrer qu'en même temps le partenaire u' de v dans A' préfère A .

La correction de l'algorithme de Gale-Shapley ne dépend pas de l'élément $v \in V$ qui fait sa proposition à chaque tour. En fait, on montre dans la suite que, quel que soit l'élément $v \in V$ non encore apparié qui fait une proposition à chaque tour, l'algorithme calcule toujours le même couplage parfait stable. Dans la suite, on dira qu'un élément $u \in U$ est *réalisable* pour un élément $v \in V$ s'il existe un couplage parfait stable qui contient (v, u) .

Q23. Montrer qu'au cours de l'**algorithme 3**, chaque $v \in V$ n'est rejeté que par les éléments $u \in U$ qui ne sont pas réalisables pour v .

Soit $M(v) \in U$ l'élément réalisable le mieux classé sur la liste des préférences de v et $P(u) \in V$ l'élément réalisable le moins bien classé sur la liste des préférences de u . On note enfin $A^* = \{(v, M(v)), v \in V\}$.

Q24. Montrer que lors de l'exécution de l'**algorithme 3** :

- 1) Tout $v \in V$, s'il est apparié, l'est avec $u \in U$ supérieur ou égal à $M(v)$ dans sa liste de préférences.
- 2) Tout $v \in V$, s'il n'est pas apparié, n'a fait des propositions qu'à des $u \in U$ strictement supérieurs à $M(v)$ dans sa liste de préférences.

Q25. En déduire que quel que soit l'élément $v \in V$ choisi dans la boucle Tant que de l'**algorithme 3**, l'algorithme termine en produisant l'ensemble A^* .

Ainsi, l'algorithme calcule le meilleur couplage parfait stable possible du point de vue des éléments de V . Il s'avère également que l'algorithme de Gale-Shapley fait correspondre u avec $P(u)$, pour tout $u \in U$. L'algorithme avantage donc de manière claire le groupe ayant l'initiative (ici V) : le couplage stable produit est dit *Pareto-V-optimal* en ce sens qu'il n'est pas Pareto-dominé sur V .

III.2 - Couplages Pareto-optimaux

Dans cette sous-partie, on utilise également le jeu de données \mathcal{D} .

L'algorithme de Gale-Shapley ne retourne pas nécessairement un couplage Pareto-optimal. On s'intéresse ici à un algorithme susceptible de produire cette propriété.

Pour construire un couplage Pareto-optimal, on représente les données à l'aide d'un graphe orienté $\mathcal{G} = (S, E)$ où :

- les sommets S sont les éléments de $V \cup U$,
- les arcs E représentent les premiers choix : $(v, u) \in E$ si et seulement si u est le premier choix de v dans sa liste de préférences et $(u, v) \in E$ si et seulement si v est le premier choix de u .

Q26. Montrer que \mathcal{G} contient au moins un circuit, c'est-à-dire un chemin dont les deux sommets extrémités sont identiques.

Q27. Justifier par l'absurde que tout $x \in V \cup U$ est dans au plus un circuit.

On propose alors l'**algorithme 4** pour calculer un couplage, dont on admettra qu'il est Pareto-optimal pour V .

Algorithme 4 - Algorithme de couplage par graphe orienté

Entrées : $n = |V| = |U|$, les listes de préférences des éléments de V et U

Sorties : Un couplage A

début

$A := \emptyset$

tant que il existe un élément de V non apparié faire

Construire le graphe \mathcal{G} sur les données courantes

pour chaque arc (v, u) présent dans un circuit de \mathcal{G} faire

$A = A \cup \{(v, u)\}$

Supprimer du problème tous les $v \in V$ et les $u \in U$ appariés

Mettre à jour les listes de préférences des individus restants

retourner A

Q28. Donner les graphes et la construction itérative de A produits par l'**algorithme 4** sur \mathcal{D} .

Q29. En utilisant les données \mathcal{D} , montrer que l'**algorithme 4** ne produit pas toujours un couplage stable.

On s'intéresse alors à une autre forme de stabilité pour assurer à la fois une propriété de Pareto et une stabilité dans les couplages.

Définition 4 (P-stabilité)

Un couplage A est P-stable si il n'existe aucune paire $(v, v') \in V^2$ telle que v préfère $m_A(v')$ et v' préfère $m_A(v)$.

Q30. Montrer par l'absurde que l'**algorithme 4** produit un couplage P-stable.

Q31. Montrer que l'**algorithme 3** ne produit pas nécessairement un couplage P-stable.

III.3 - Implémentation

Dans la suite, les individus sont numérotés de 0 à $n - 1$. Ainsi $V = \{0, \dots, n - 1\}$ et $U = \{0, \dots, n - 1\}$.

On suppose disposer de deux matrices de taille $n \times n$ L_v et L_u donnant respectivement les listes de préférences des éléments de V et U : ainsi, $L_u[i][j]$ est le j -ème élément de V préféré par l'élément $i \in U$.

On définit ces tableaux en C par l'intermédiaire du code suivant :

```
/* La directive #define est utilisée pour définir une valeur pour la constante n qui
   servira à déclarer des tableaux de taille fixe.*/
#define n 4

int main(void) {
    int Lv[n][n], Lu[n][n];
    (...)
```

III.3 .1 - Algorithme de Gale-Shapley

Pour connaître rapidement le classement des éléments de V dans les listes de préférences des éléments de U , on définit un tableau $Rang$ de taille $n \times n$, tel que $Rang[i][j]$ donne le rang de $j \in V$ dans la liste de préférences de l'élément $i \in U$. Par convention, les rangs commencent à 0.

Q32. Écrire une fonction de prototype

```
void calcule_rang(int Lu[n][n], int Rang[n][n])
qui construit le tableau Rang.
```

Afin de pouvoir ajouter ou supprimer rapidement des couples (v, u) au couplage parfait stable A , on définit deux tableaux `CoupleV` et `CoupleU` tels que `CoupleV[i]` ou `CoupleU[i]` est le partenaire actuel de i . On définit enfin un dernier tableau `USuiv` de taille n stockant pour chaque élément de V le rang du prochain élément de U à qui il peut faire une proposition.

Q33. Proposer des valeurs d'initialisation pour ces trois tableaux.

Q34. Avec ces structures de données, écrire une fonction de prototype

void gale_shapley(int Lv[n][n], int Lu[n][n], int CoupleU[n], int CoupleV[n])
qui réalise l'**algorithme 3**. On ne retournera pas le couplage parfait stable, on remplira seulement les variables `CoupleU` et `CoupleV` passées en argument. Pour le choix de v , on prendra les éléments de V par ordre croissant de leur numéro en choisissant à chaque fois le premier libre.

III.3 .2 - Algorithme de couplage par graphe orienté

Sans explicitement coder tout l'**algorithme 4** qui requiert de rechercher les circuits dans \mathcal{G} , on se propose de coder quelques fonctions utiles à la réalisation de l'algorithme.

Pour pouvoir numérotter les sommets du graphe de manière continue, les individus sont maintenant numérotés de 0 à $2n - 1$ de la manière suivante : $V = \{0, \dots, n - 1\}$ et $U = \{n, \dots, 2n - 1\}$.

On code le graphe orienté \mathcal{G} par listes d'adjacence. On propose les types suivants :

```
struct noeud {
    int individu;           // Element de U union V
    struct noeud* suivant;
};

typedef struct noeud *liste;

struct graphe_s {
    int nb_sommets;
    liste *liste_adjacence;
};

typedef struct graphe_s graphe;
```

Q35. Écrire une fonction de prototype

void ajoute_arc(graphe *g, int i, int j)
qui ajoute l'arc (i, j) au graphe \mathcal{G} .

Q36. Écrire une fonction de prototype

void supprime_arc(graphe *g, int i, int j)
qui supprime l'arc (i, j) du graphe \mathcal{G} .

Q37. Écrire une fonction de prototype

void supprime_sommet(graphe *g, int s)
qui supprime le sommet s du graphe \mathcal{G} . Pour ce faire, on supprimera uniquement tous les arcs dont ce sommet est origine ou destination.

Q38. Écrire une fonction de prototype

graphe *construit_graphe(int Lv[n][n], int Lu[n][n])
qui construit le graphe \mathcal{G} utilisé dans l'**algorithme 4** et dont la définition est donnée juste avant la **Q26**. On prendra bien garde à la numérotation des éléments de U .

FIN

SESSION 2025**MP7IN**

ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE MP

INFORMATIQUE

Durée : 4 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

RAPPEL DES CONSIGNES

- Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.
 - Ne pas utiliser de correcteur.
 - Écrire le mot *FIN* à la fin de votre composition.
-

Les calculatrices sont interdites.

Le sujet est constitué d'un unique problème et comporte cinq parties qui ne sont pas complètement indépendantes.

Répartition de la gestion d'un réseau minimisant la bande passante

Le Listenbourg (**figure 1**) est une république fictive de la péninsule ibérique, comportant environ 66 millions d'habitants, née sur les réseaux sociaux en octobre 2022. Au Listenbourg deux fournisseurs d'accès, MaxDébit et MinLatence, cherchent à se partager la gestion d'un tout nouveau réseau : l'Ultranet.

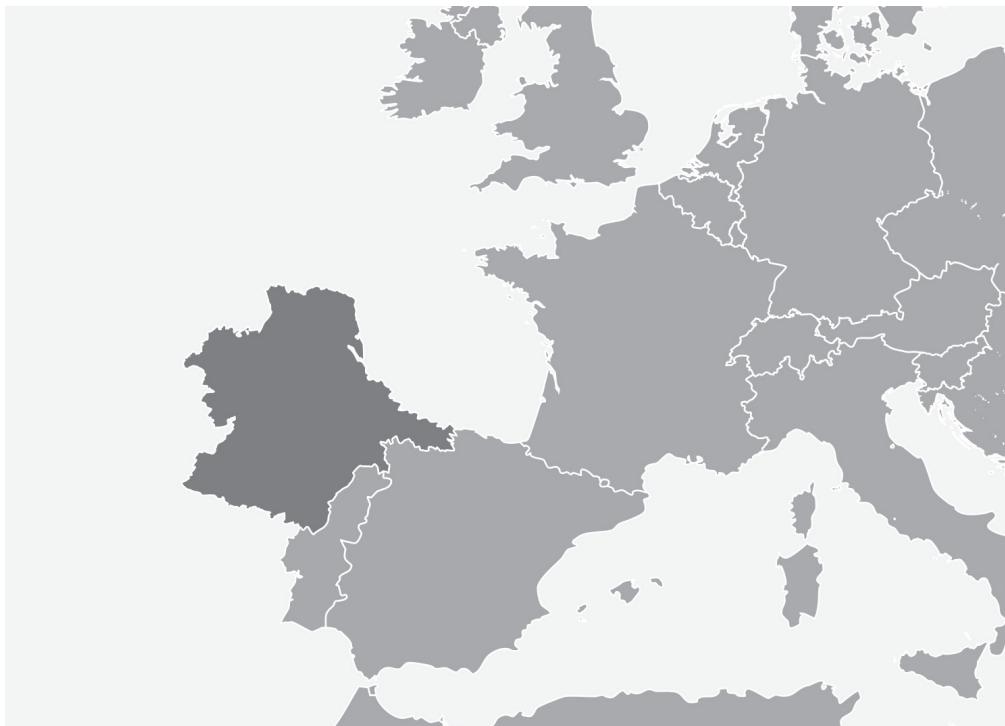


Figure 1 - La république imaginaire du Listenbourg

Le graphe simplifié du réseau Ultranet de ce pays est représenté à la **figure 2**. Le nom des villes correspondant aux identifiants des sommets est enregistré dans la base de données (**figure 3**). Une arête relie deux villes lorsqu'un câblage physique direct est établi entre elles et la bande passante permise par ce câblage est indiquée comme poids. Il y a au plus une arête entre deux villes. La structure du réseau Ultranet a été intégralement terminée lors du grand plan « Réseau pour Tous » mené par le premier ministre Petro Sank-Ærixh et il s'agit désormais d'en confier l'exploitation aux deux fournisseurs d'accès concurrents.

Dans ce sujet, on s'intéresse à une procédure de partage de la gestion du réseau entre les deux opérateurs. On suppose qu'une liaison entre deux villes ne peut être exploitée que par l'un ou par l'autre des deux fournisseurs d'accès qui en aura alors l'usage exclusif. Le gouvernement du Listenbourg ne souhaite pas s'embarrasser avec une procédure complexe d'appel d'offre et impose la procédure simple suivante pour répartir l'exploitation exclusive des liaisons physiques du réseau :

- tant qu'il reste des liaisons à attribuer, chaque opérateur, à tour de rôle et en commençant par MaxDébit, choisit parmi les liaisons restantes une liaison qu'il souhaite exploiter exclusivement.

Une fois la procédure de répartition terminée, c'est-à-dire une fois que toutes les liaisons ont été attribuées, chaque fournisseur d'accès dispose de son propre sous-réseau exclusif, dont la gestion lui revient entièrement. L'objectif d'un fournisseur d'accès est d'obtenir un sous-réseau permettant de proposer la meilleure bande passante possible entre deux villes quelconques du pays.

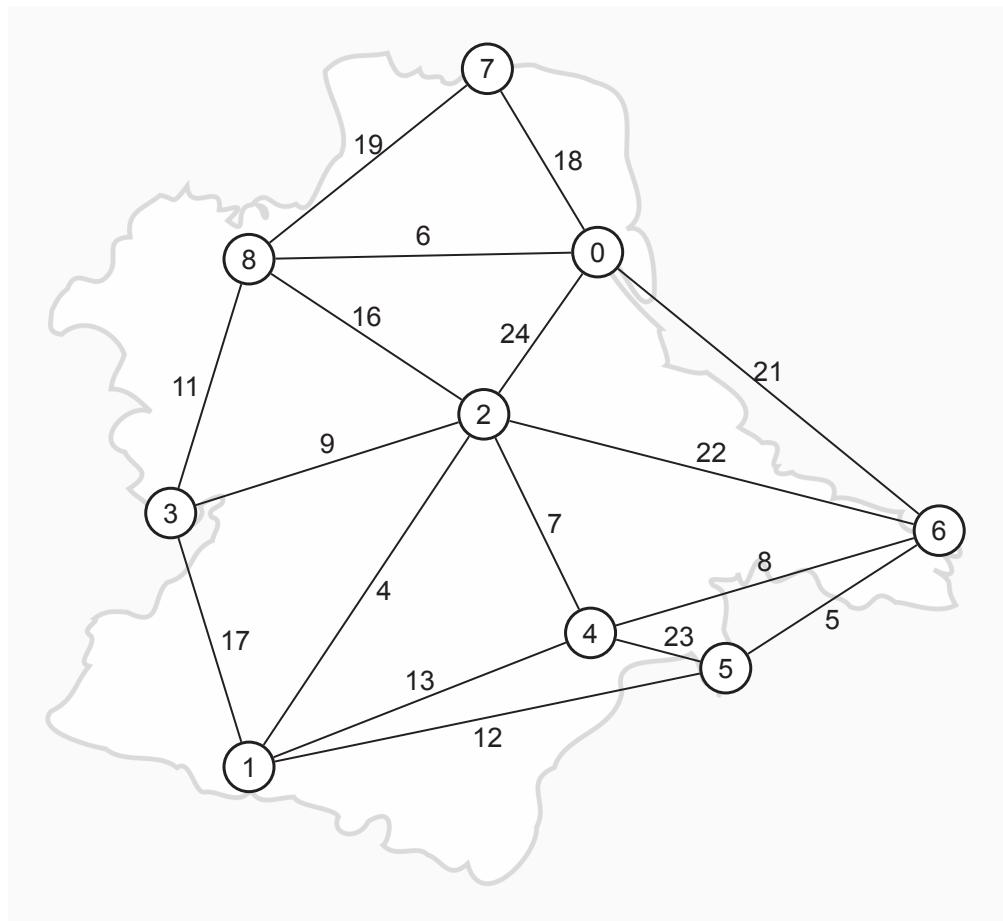


Figure 2 - Réseau simplifié du Listenbourg

La première partie du sujet s'intéresse à l'étude d'une base de données du réseau mise à disposition des fournisseurs d'accès (informatique de tronc commun, langage SQL). La deuxième partie introduit et étudie quelques propriétés de l'arbre couvrant de poids maximal (option informatique). La troisième partie s'intéresse à la recherche de l'arbre couvrant de poids maximal en utilisant l'algorithme de Borůvka (option informatique, langage OCaml). La quatrième partie se propose de montrer que la bande passante limite d'un réseau correspond au poids de l'arête de poids minimal de l'arbre couvrant de poids maximal du graphe (option informatique). La cinquième partie propose de modéliser la procédure de répartition de la gestion du réseau comme un jeu à deux joueurs (informatique de tronc commun, langage Python).

Les cinq parties sont essentiellement indépendantes dans leur traitement mais des notions utiles à la résolution d'une partie peuvent être introduites dans les parties précédentes. La **partie I** est indépendante des quatre autres. La **partie II** introduit des notions utiles à la résolution des **parties III et IV**. La **partie IV** introduit des concepts utilisés dans la **partie V**.

Dans tout le sujet, il est toujours admis d'utiliser un résultat ou un programme correspondant à une question précédente, même si cette question n'a pas été résolue.

Partie I - Base de données du réseau

Le ministère des affaires environnementales, des ressources, de l'agriculture et des forêts du Listenbourg met à disposition des deux fournisseurs d'accès une base de données relationnelle. Celle-ci comporte deux tables dont le schéma est le suivant :

- **villes**(id : entier, nom : texte, pop : flottant)
- **liaisons**(id1 : entier, id2 : entier, bp : flottant)

Un enregistrement de la table **villes** comporte l'identifiant unique d'une ville (id), le nom de cette ville (nom) et sa population en millions d'habitants (pop). *A priori*, plusieurs villes du Listenbourg peuvent porter le même nom. Un enregistrement de la table **liaisons** correspond à une liaison physique directe entre deux villes données par leur identifiant (id1, id2) ainsi que la bande passante en $Tb.s^{-1}$ correspondant à ce câblage (bp). On garantit que dans la représentation d'une liaison, l'identifiant de la première ville est toujours strictement inférieur à celui de la deuxième ville. Rappelons qu'il ne peut exister qu'au plus une liaison entre deux villes.

Un exemple simplifié du contenu de cette base de données, correspondant au graphe simplifié du réseau du Listenbourg de la **figure 2**, est donné **figure 3**.

villes			liaisons		
id	nom	pop	id1	id2	bp
0	Lurenberg	12.0	0	2	24.0
1	Veroja	10.0	0	6	21.0
2	Aschlöss	8.0	0	7	18.0
3	Stratord	5.2	0	8	6.0
4	Gossard	5.1	1	2	4.0
5	La Galinera	5.0	1	3	17.0
6	Sainte Marie	5.0	1	4	13.0
7	Atlanitkischer	4.7	1	5	12.0
8	Gasparländ	3.5	2	3	9.0
			2	4	7.0
			2	6	22.0
			2	8	16.0
			3	8	11.0
			4	5	23.0
			4	6	8.0
			5	6	5.0
			7	8	19.0

Figure 3 - Contenu de la base de données correspondant au réseau simplifié de la **figure 2**

- Q1.** Au vu de la situation modélisée, proposer des clés primaires pour les tables **villes** et **liaisons**, en justifiant succinctement. Identifier les clés étrangères présentes dans ce schéma relationnel.
- Q2.** Pour chacune des questions suivantes on demande d'écrire une requête SQL portant sur le schéma relationnel proposé qui fonctionnerait quel que soit le contenu de la base de données et pas uniquement sur l'exemple de contenu proposé à la **figure 3**.
- (a) Écrire une requête SQL qui donne les noms des villes comportant strictement plus de cinq millions d'habitants ;
 - (b) Écrire une requête SQL qui donne la bande passante moyenne des liaisons incidentes à la ville d'identifiant 2 ;
 - (c) Écrire une requête SQL qui donne les noms des deux villes reliées par la liaison de bande passante maximale. *On suppose que toutes les liaisons ont une bande passante différente.*
- Q3.** Déterminer le résultat de la requête SQL suivante sur l'exemple de contenu de la base de données représentée à la **figure 3**.

```

SELECT nom, COUNT(*) AS d
FROM villes
JOIN (SELECT id1 AS x, id2 FROM liaisons UNION SELECT id2 AS x, id1 FROM liaisons)
ON id = x
GROUP BY id
HAVING d >= 4
ORDER BY id ASC

```

Partie II - Arbre couvrant de poids maximal

II.1 - Acyclicité, connexité et arbres

Un *graphe* (non orienté) est un couple $G = (S, A)$ formé d'un ensemble S fini non vide de *sommets* et d'un ensemble A d'*arêtes* constitué de parties de S de cardinal 2. On note $|S|$ le nombre de sommets de G et $|A|$ le nombre d'arêtes de G . Les voisins d'un sommet $x \in S$ sont notés $\mathcal{V}(x)$ et leur nombre $d(x) = |\mathcal{V}(x)|$ est le *degré* de x .

Un chemin dans un graphe est une suite $c = x_0 x_1 \dots x_n$ de sommets avec $\forall i \in \llbracket 0, n-1 \rrbracket, \{x_i, x_{i+1}\} \in A$. La longueur $|c|$ d'un tel chemin est $n \in \mathbb{N}$. Un chemin est *élémentaire* si tous les sommets empruntés par ce chemin sont deux à deux distincts. Un chemin est *simple* si toutes les arêtes empruntées par ce chemin sont deux à deux distinctes.

Un *circuit* est un chemin simple $x_0 x_1 \dots x_n$ avec $n \geq 3$ et $x_0 = x_n$. Un graphe est *acyclique* s'il ne comporte pas de circuit.

Un graphe est *connexe* si deux sommets quelconques sont toujours reliés par un chemin. Une *composante connexe* est un ensemble $C \subseteq S$ de sommets dont tous les couples de sommets sont reliés par au moins un chemin.

Un *arbre* est un graphe connexe acyclique.

Si $a = \{x, y\}$ est une partie de S de cardinal 2, on note $G + a$ le graphe $(S, A \cup \{a\})$, c'est-à-dire le graphe G auquel on a ajouté l'arête a et $G - a$ le graphe $(S, A \setminus \{a\})$, c'est-à-dire le graphe G auquel on a retiré l'arête a .

Q4. Montrer que si $G = (S, A)$ est un graphe connexe et que si $a = \{x, y\} \in A$ est une arête de G appartenant à un cycle de G , alors le graphe $G - a$ est encore connexe.

On admet la proposition suivante :

Proposition 1 : relation entre $|S|$ et $|A|$ pour les graphes connexes ou acycliques

Soit $G = (S, A)$ un graphe :

- (a) si G est connexe alors $|A| \geq |S| - 1$;
- (b) si G est acyclique alors $|A| \leq |S| - 1$.

Q5. Montrer que si $G = (S, A)$ est connexe et que $|A| = |S| - 1$ alors G est un arbre.

II.2 - Sous-graphe et arbre couvrant

Dans tout ce sujet un *sous-graphe* d'un graphe G est un graphe $G' = (S, A')$ avec $A' \subseteq A$. Notons qu'un sous-graphe de G comporte exactement les mêmes sommets que G . On identifie un sous-graphe $G' = (S, A')$ avec le sous-ensemble d'arêtes $A' \subseteq A$. Un *arbre couvrant* de G est un sous-graphe de G qui est un arbre.

Q6. Montrer qu'un graphe $G = (S, A)$ est connexe si et seulement s'il admet un arbre couvrant.

Q7. Soit $T = (S, A')$ un arbre couvrant de G avec $A' \subsetneq A$ et $a \in A \setminus A'$.

- (a) Justifier que $T + a$ n'est pas acyclique.
- (b) Soit $a' \in A'$ une arête d'un cycle de $T + a$. Montrer que $T + a - a'$ est un arbre couvrant de G .

II.3 - Graphes pondérés et arbre couvrant de poids maximal

Un *graphe* pondéré est un triplet (S, A, p) où (S, A) est un graphe et $p : A \rightarrow \mathbb{R}$ une fonction de pondération des arêtes. Si $\{x, y\} \in A$ est une arête, $p(\{x, y\})$ est appelé le *poids* de cette arête. Le poids d'un graphe pondéré est la somme des poids de ses arêtes :

$$p(G) = \sum_{a \in A} p(a).$$

On étend la notion de sous-graphe aux graphes pondérés de la manière suivante : un sous-graphe d'un graphe pondéré $G = (S, A, p)$ est un graphe pondéré $G' = (S, A', p_{A'})$ où $A' \subseteq A$ et où $p_{A'}$ est la restriction de p à A' . On se permettra de ne pas systématiquement indiquer la fonction de pondération.

Le plan « Réseau pour Tous » du gouvernement a permis d'interconnecter toutes les villes du Listenbourg. On considèrera donc en général un graphe pondéré $G = (S, A, p)$ non orienté *connexe*, avec la fonction de pondération correspondant à la bande passante associée à chaque liaison. Chaque province ayant opté pour son propre câblage, deux villes ne sont jamais reliées par le même type de câblage et la bande passante de deux liaisons est *toujours* différente. Ceci se traduit donc par une fonction de pondération p injective : dans toute la suite du sujet, les poids des arêtes de G sont toujours deux à deux distincts.

On se propose de démontrer la **proposition 2** suivante :

Proposition 2 : existence et unicité de l'arbre couvrant de poids maximal

Soit $G = (S, A, p)$ un graphe *connexe* muni d'une fonction de pondération *injective*. Alors, il existe un unique arbre couvrant de poids maximal de G . On note $T^* = (S, A^*)$ cet arbre couvrant.

Q8. Montrer l'existence d'un arbre couvrant de poids maximal de G .

Q9. Pour montrer l'unicité, on se propose de procéder par l'absurde. Supposons l'existence de deux arbres couvrants de poids maximal $T_1 = (S, A_1)$ et $T_2 = (S, A_2)$ différents. Considérons une arête de poids maximal parmi les arêtes qui appartiennent à un seul des deux arbres, c'est-à-dire une arête de poids maximal dans $(A_1 \setminus A_2) \cup (A_2 \setminus A_1)$, qui est bien non vide puisque $A_1 \neq A_2$. Sans perte de généralité, on peut noter a_2 une telle arête et supposer que $a_2 \in A_2 \setminus A_1$.

- (a) Montrer qu'il existe une arête $a_1 \in A_1 \setminus A_2$ sur un cycle de $T_1 + a_2$.
- (b) Conclure en considérant le sous-graphe $T_1 + a_2 - a_1$.

Partie III - Recherche d'un arbre couvrant de poids maximal

III.1 - Rappels et fonctions élémentaires en OCaml

On rappelle que, dans le langage OCaml :

- on peut représenter des tableaux par le type `'a array` ;
- si `e` est une expression de type `'a`, l'expression `Array.make n e` permet de créer un tableau de type `'a array` de $n \in \mathbb{N}$ cases dont toutes les cases contiennent la valeur de l'expression `e` ;

```
Array.make : int -> 'a -> 'a array
```

- si `a` est un tableau de type `'a array` de taille $n \in \mathbb{N}$, l'expression `Array.length a` permet d'obtenir la valeur n ;

```
Array.length : 'a array -> int
```

- si $0 \leq i < n$, l'expression `a.(i)` s'évalue en la valeur contenue à la case `i` du tableau `a` et si `e` est une expression de type `'a`, l'expression `a.(i) <- e` permet de remplacer la valeur contenue à la case `i` du tableau `a` par la valeur de l'expression `e` ;
- la fonction `List.iter` a le comportement suivant : si `f` est une fonction de type `'a -> unit`, alors l'expression `List.iter f [a1; a2; ...; an]` est équivalente à `f a1; f a2; ...; f an`

```
List.iter : ('a -> unit) -> 'a list -> unit
```

- une boucle `for i = a to b do ... done` permet de faire évoluer une variable `i` entre les valeurs entières `a` et `b` **incluses** ;
- il existe deux fonctions `max` et `min` de type `'a -> 'a -> 'a`.

Q10. Écrire une fonction `max_tab : int array -> int` telle que, pour un tableau d'entiers `a` de taille $n \geq 1$, `max_tab a` renvoie la valeur du plus grand entier contenu dans `a`. On attend une complexité linéaire en n , ce que l'on justifiera en une phrase.

III.2 - Représentation des graphes en OCaml

On représente un graphe pondéré $G = (S, A, p)$, avec $S = \llbracket 0, n - 1 \rrbracket$ et $n \in \mathbb{N}^*$ par un tableau de listes d'adjacence pondérées.

```
type graphe = (int * float) list array
```

Si g représente un graphe et si $0 \leq i < n$ est un sommet, alors $g.(i)$ contient la liste des couples $\{(j, p(\{i, j\}))\}_{j \in V(i)}$, c'est-à-dire la liste des couples de la forme (j, w) avec $w = p(\{i, j\})$ et j voisin de i .

Remarquons que puisque les graphes sont non orientés, si i est dans la liste d'adjacence de j , alors j est dans la liste d'adjacence de i avec le même poids.

Par exemple, si g est le graphe de la **figure 2**, on a $g.(0)$ qui vaut par exemple :

```
[(2, 24.0); (6, 21.0); (7, 18.0); (8, 6.0)]
```

Le nombre de sommets du graphe correspond donc à la taille du tableau de listes d'adjacences :

```
let nb_sommets g = Array.length g
```

Q11. Donner une valeur possible pour $g.(7)$.

On représente une arête par un triplet (w, i, j) où $w = p(\{i, j\})$ et où $i < j$. Remarquons que chaque arête est ainsi représentée exactement une fois et qu'une arête commence par son poids.

Q12. Écrire en OCaml une fonction `ajoute_arete : graphe -> (float * int * int) -> unit` telle que `ajoute_arete g (w, i, j)` permet d'ajouter au graphe g l'arête (w, i, j) . On suppose que cette arête n'était pas déjà présente et la complexité doit être en temps constant.

Q13. Écrire en OCaml une fonction `toutes_les_aretes : graphe -> (float * int * int) list` qui permet d'obtenir la liste des arêtes d'un graphe avec la convention ci-dessus. On attend une complexité linéaire en $|S| + |A|$, ce que l'on justifiera rapidement.

Si a_1 et a_2 sont deux arêtes de type `float * int * int`, l'expression $a_1 < a_2$ compare les arêtes pour l'ordre lexicographique des triplets donc pour la valeur du poids de l'arête en premier. Comme dans ce sujet on suppose toujours que tous les poids sont deux à deux distincts, on peut directement utiliser l'ordre OCaml `<` sur les arêtes pour comparer les poids des arêtes. Ainsi par exemple, `min a1 a2` s'évalue en l'arête de poids minimal entre a_1 et a_2 .

Q14. Écrire une fonction `min_arete : (float * int * int) list -> (float * int * int)` de manière récursive en OCaml qui renvoie l'arête de poids minimal d'une liste d'arêtes supposée non vide. Quelle est sa complexité ?

III.3 - Recherche des composantes connexes

Pour un graphe $G = (S, A, p)$ avec $S = \llbracket 0, n - 1 \rrbracket$, on cherche à construire un tableau c de taille n représentant les composantes connexes du graphe. Les composantes connexes sont indexées entre 0 et $p - 1$ où p est le nombre de composantes connexes du graphe. Pour $0 \leq i < n$ la case $c.(i)$ comporte l'indice k de la composante connexe du sommet i (ou la valeur -1 si cette composante n'a pas encore été déterminée).

On considère la fonction `explorer` : `graphe -> int array -> int -> int -> unit` suivante pour un graphe g , un tableau c de taille n , un identifiant de composante connexe $0 \leq k < p$ et un sommet de départ s , vérifiant la convention suivante : « un sommet $0 \leq i < n$ a été visité si et seulement si $c.(i) \geq 0$ ».

```

1 let rec explorer g c k s =
2   if c.(s) < 0 then begin
3     c.(s) <- k;
4     List.iter (fun (v, _) -> explorer g c k v) g.(s)
5   end

```

Q15. Décrire précisément ce que réalise un appel à `explorer g c k s`.

Q16. Écrire en OCaml une fonction `composantes_connexes` : `graphe -> int array` telle que, pour un graphe g , `composantes_connexes g` s'évalue en le tableau c des indices des composantes connexes. Cette fonction doit avoir une complexité en $O(|S| + |A|)$ que l'on ne demande pas de justifier.

Q17. En déduire une fonction `est_connexe` : `graphe -> bool` telle que `est_connexe g` s'évalue à `true` si le graphe g est connexe et à `false` sinon.

III.4 - Algorithme de Borůvka

Dans toute cette sous-partie, on considère un graphe $G = (S, A, p)$ connexe avec une fonction de pondération p injective. On note $T^* = (S, A^*)$ l'unique arbre couvrant de poids maximal de G .

Considérons $H = (S, B)$ un sous-graphe acyclique de G , avec donc $B \subseteq A$. On considère $C \subseteq S$ une composante connexe de H . Une arête **sûre** pour C est une arête de A ayant exactement une extrémité dans C dont le poids est maximal parmi les arêtes qui ont exactement une extrémité dans C .

Par exemple, pour le sous-graphe H de la **figure 4** et pour la composante connexe $\{1, 4, 5, 6\}$ l'arête de poids 22 est sûre (maximale parmi les arêtes de poids 4, 7, 17, 21 et 22 qui sont celles ayant exactement une extrémité dans cette composante). Sauf si H est connexe, chaque composante connexe de H comporte exactement une arête sûre. Une même arête peut être sûre pour deux composantes connexes C et C' . Dans l'exemple de la **figure 4**, l'arête de poids 22 est également l'arête sûre pour la composante connexe $\{0, 2\}$.

Q18. Déterminer l'arête sûre pour la composante $\{3\}$ ainsi que celle pour la composante $\{7, 8\}$.

Q19. On suppose dans cette question que $H = (S, B)$ est un sous-graphe acyclique de l'arbre couvrant $T^* = (S, A^*)$ de poids maximal de G , et donc que $B \subseteq A^*$. Montrer que A^* contient toutes les arêtes sûres des composantes connexes de H .

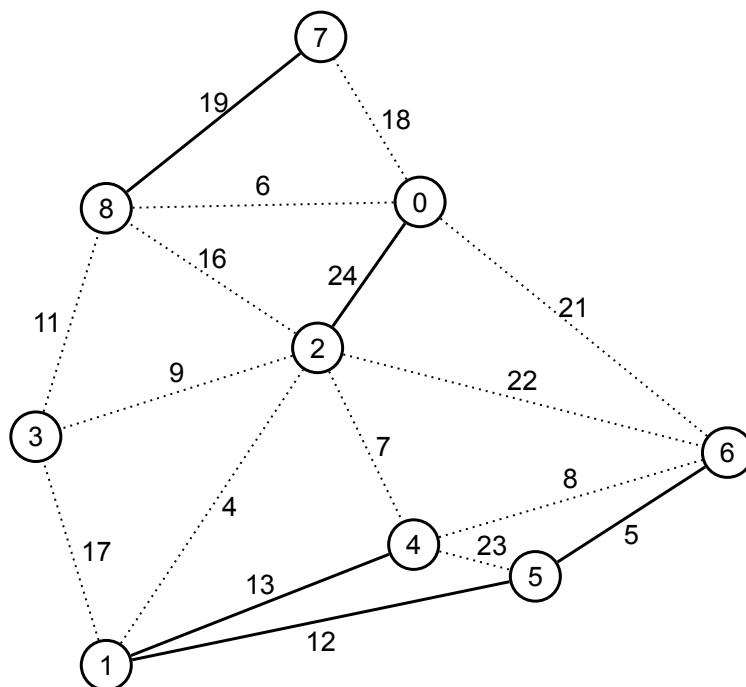


Figure 4 - Un sous-graphe H du graphe de la **figure 2**, comportant quatre composantes connexes. Les arêtes qui n'appartiennent pas au sous-graphe sont représentées en pointillés

L'idée de l'algorithme de Borůvka, dont le pseudocode est donné par l'**algorithme 1**, est de construire progressivement un sous-graphe acyclique $H = (S, B)$ de l'arbre couvrant de poids maximal $T^* = (S, A^*)$. Au départ $B = \emptyset$. À chaque étape, on ajoute au sous-graphe acyclique H en cours de construction toutes les arêtes sûres pour les composantes connexes de H .

Algorithme 1 : algorithme de Boruvka

Entrée : Graphe $G = (S, A, p)$ pondéré injectivement et connexe

Sortie : $H = (S, B)$ l'arbre couvrant de poids maximal de G

$$1 \ H = (S, \emptyset);$$

2 tant que H n'est pas connexe faire

3 ajouter à H toutes les arêtes sûres pour les composantes connexes de H ;

4 fin

- Q20.** Donner le déroulé de l'**algorithme 1** sur le graphe G de la **figure 2**. Indiquer à chaque étape et pour chaque composante connexe son arête sûre et indiquer l'ensemble des arêtes sûres ajoutées à chaque étape.

- Q21.** Montrer que l'algorithme de Borůvka termine.

- Q22.** Montrer que l'algorithme de Borůvka renvoie l'arbre couvrant de poids maximal $T^* = (S, A^*)$ de G .

On considère la fonction suivante qui prend en entrée un graphe g représentant le graphe $G = (S, A, p)$ connexe pondéré et un graphe h représentant un sous-graphe acyclique $H = (S, B)$ de G supposé non encore connexe.

```

1 let aretes_sures g h =
2   let c = composantes_connexes h in
3   let p = max_tab c + 1 in
4   let aretes = toutes_les_aretes g in
5   let sures = Array.make p (min_arete aretes) in
6   let traite a =
7     let (_, i, j) = a in
8     if c.(i) <> c.(j) then begin
9       sures.(c.(i)) <- max a sures.(c.(i));
10      sures.(c.(j)) <- max a sures.(c.(j));
11    end
12  in
13  List.iter traite aretes;
14  sures

```

Q23. Déterminer le type de cette fonction. Expliquer ce que réalise cette fonction et préciser la structure et le contenu de la valeur renvoyée. Déterminer sa complexité en fonction de $|A|$ en justifiant. *On remarquera que d'après la proposition 1, puisque le graphe G est connexe, on a $|S| - 1 \leq |A|$ et donc qu'un $O(|S|)$ est un $O(|A|)$.*

On suppose disposer d'une fonction :

```
ajoute_aretes_sures : graphe -> (float * int * int) array -> unit
```

telle que si h représente un sous-graphe acyclique de G et que $sures$ est la valeur renvoyée par $aretes_sures g h$ alors $ajoute_aretes_sures h$ ajoute à h toutes les arêtes contenues dans $sures$. On garantit que cette fonction est en complexité $O(|A|)$ et qu'elle n'ajoute qu'une seule fois une même arête à h même si cette arête est sûre pour deux composantes connexes différentes.

Q24. Écrire en OCaml une fonction $borukva : \text{graphe} \rightarrow \text{graphe}$ telle que $borukva g$ sur un graphe g s'évalue en un sous-graphe h de G correspondant à son arbre couvrant de poids maximal.

Q25. On s'intéresse maintenant à la complexité temporelle de cet algorithme.

- (a) Montrer qu'à chaque passage dans la boucle **tant que** à la ligne 2 de l'**algorithme 1**, le nombre de composantes connexes de H est divisé au moins par 2.
- (b) En déduire la complexité temporelle dans le pire cas de l'algorithme de Borůvka.
- (c) Quelle est la complexité temporelle de cet algorithme dans le meilleur cas ?

Partie IV - Chemin de bande passante maximale

Dans cette partie, on considère un graphe pondéré $G = (S, A, p)$, non nécessairement connexe, mais dont la fonction de pondération est injective.

On définit la *bande passante* d'un chemin $c = x_0x_1 \dots x_n$ comme la quantité :

$$\bar{b}(c) = \min_{0 \leq i \leq n-1} p(\{x_i, x_{i+1}\})$$

c'est-à-dire le poids minimal d'une arête de ce chemin. En effet, la bande passante d'un chemin est limitée par la plus petite bande passante des arêtes de ce chemin. La bande passante d'un chemin de longueur nulle est $+\infty$.

Pour deux sommets $x, y \in S$, un *chemin de bande passante maximale* entre x et y est un chemin c qui relie x à y dont la bande passante est maximale parmi tous les chemins entre x et y . Un chemin de bande passante maximale est ainsi un chemin permettant de maximiser la bande passante entre deux villes, ce que cherchent à réaliser les opérateurs MaxDébit et MinLatence. On note :

$$\bar{b}_{\max}(x, y) = \max \{\bar{b}(c) \mid c \text{ chemin de } x \text{ à } y\}$$

la bande passante d'un chemin de bande passante maximale de x à y . On a $\bar{b}_{\max}(x, y) = -\infty$ si x et y ne sont pas reliés par au moins un chemin. Notons que cette quantité est bien définie, l'ensemble des arêtes étant fini, il y a un nombre fini de valeurs possibles pour la bande passante d'un chemin de x à y .

Q26. Déterminer un chemin de bande passante maximale entre Lurenberg (d'identifiant 0) et Veroja (d'identifiant 1) sur le graphe de la **figure 2**. Quelle est sa bande passante ?

On admet que dans un arbre il existe un unique chemin élémentaire entre tout couple de sommets.

Q27. Soient $x, y \in S$, montrer que l'unique chemin de x à y dans l'arbre couvrant de poids maximal T^* est un chemin de bande passante maximale entre x et y .

Un fournisseur d'accès cherche à pouvoir offrir un chemin avec la bande passante la plus grande possible entre tous les couples de villes possibles. Plus précisément, il s'intéresse à la quantité :

$$\bar{b}_{\lim}(G) = \min \{\bar{b}_{\max}(x, y) \mid x, y \in S\}$$

appelée la *bande passante limite* d'un graphe G .

Q28. Justifier que

$$\bar{b}_{\lim}(G) = \begin{cases} -\infty & \text{si } G \text{ n'est pas connexe} \\ \min \{p(a) \mid a \in A^*\} & \text{avec } T^* = (S, A^*) \text{ l'arbre couvrant de poids maximal de } G \text{ sinon.} \end{cases}$$

Partie V - Jeu sur un graphe

Dans cette partie, on suppose que $G = (S, A, p)$ est un graphe pondéré connexe muni d'une fonction de pondération injective.

V.1 - Procédure de partage vue comme un jeu

On peut voir la procédure de partage du réseau décrite dans l'introduction du sujet comme un jeu à deux joueurs. Les deux joueurs sont MaxDébit (joueur max associé à la valeur 1) et MinLatence (joueur min associé à la valeur -1).

Une *configuration* est la donnée de deux sous-graphes G_1 et G_{-1} de G , d'arêtes disjointes, correspondant aux choix des arêtes effectués jusque-là par les deux joueurs, ainsi que la donnée du joueur qui doit jouer (il s'agit du joueur qui *contrôle* cette configuration). Un *coup* possible consiste à choisir une nouvelle arête du graphe non encore attribuée.

Une configuration est finale lorsque toutes les arêtes ont été attribuées et donc lorsque les arêtes de G_1 et G_{-1} partitionnent celles de G . La configuration initiale est contrôlée par MaxDébit et est composée de deux sous-graphes sans arêtes.

Pour tester la viabilité de cette procédure, le gouvernement commence par effectuer un essai sur le réseau de l'île de Korße rattachée au Listenbourg, représenté à la **figure 5**.

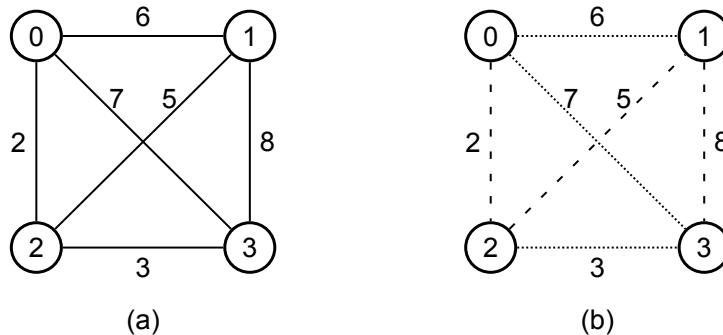


Figure 5 - (a) Le graphe correspondant au réseau de l'île de Korße. (b) Exemple de configuration finale (contrôlée par MaxDébit) atteinte à l'issue d'une partie. Les arêtes choisies par le joueur MaxDébit sont représentées par un trait discontinu et celle par le joueur MinLatence par une ligne en pointillés

Un exemple possible de partie de ce jeu, dont l'issue est représentée à la **figure 5**, est la suivante :

- l'arête {1, 3} est choisie par MaxDébit ;
- l'arête {0, 1} est choisie par MinLatence ;
- l'arête {1, 2} est choisie par MaxDébit ;
- l'arête {2, 3} est choisie par MinLatence ;
- l'arête {0, 2} est choisie par MaxDébit ;
- l'arête {0, 3} est choisie par MinLatence.

Ici les deux opérateurs obtiennent la gestion d'un sous-réseau qui est exactement un arbre couvrant, mais ce n'est pas le cas en général, le sous-graphe correspondant au sous-réseau obtenu par un joueur pouvant même ne pas être connexe.

La **partie IV** montre que l'objectif d'un fournisseur d'accès est de choisir un sous-graphe G' de manière à maximiser la quantité $\bar{b}_{\lim}(G')$, c'est-à-dire de maximiser la bande passante limite du sous-graphe. D'après la **partie IV**, pour cela, il s'agit tout d'abord d'obtenir un sous-graphe G' connexe (sans quoi $\bar{b}_{\lim}(G') = -\infty$) puis, le cas échéant, de considérer l'arête de poids minimal de l'arbre couvrant de poids maximal de G' , $\bar{b}_{\lim}(G')$ étant alors le poids de cette arête. Un fournisseur d'accès estime qu'il est gagnant si la bande passante limite de son sous-graphe $\bar{b}_{\lim}(G')$ est strictement supérieure à celle de son concurrent. Une configuration finale est donc gagnante pour MaxDébit si $\bar{b}_{\lim}(G_1) > \bar{b}_{\lim}(G_{-1})$ et gagnante pour MinLatence si $\bar{b}_{\lim}(G_1) < \bar{b}_{\lim}(G_{-1})$. Remarquons que si seul l'un des deux joueurs réussit à obtenir un sous-graphe connexe, celui-ci est nécessairement gagnant. Il peut également y avoir match nul si aucun des deux joueurs n'obtient un graphe connexe à l'issue de la procédure d'attribution.

En reprenant l'exemple proposé à la **figure 5**, MaxDébit obtient un sous-graphe dont la plus petite arête de son arbre couvrant est de poids 2 et MinLatence obtient un sous-graphe dont la plus petite arête de son arbre couvrant est de poids 3. Le joueur MinLatence remporte donc la partie.

Q29. Justifier que ce jeu termine toujours, c'est-à-dire que toute partie est nécessairement finie.

Q30. Justifier que la seule possibilité de match nul est celle où aucun des deux joueurs n'obtient un graphe connexe à l'issue de la partie.

Q31. Montrer, sur un exemple, que la stratégie consistant à toujours choisir l'arête de poids maximal parmi les arêtes restantes n'est pas une stratégie gagnante pour le joueur MaxDébit.

V.2 - Implémentation en Python

On rappelle qu'en Python une boucle `for i in range(a, b)` permet de faire évoluer une variable `i` entre les valeurs `a` **incluses** et `b` **exclues**.

On modélise en Python le graphe pondéré $G = (S, A, p)$ par un objet `g` de type `Graphe` dont l'implémentation n'est pas précisée ici. Si `g` est un objet de type `Graphe`, `g.nb_sommets` est un entier correspondant à $n = |S|$ et `g.arestes` est une liste de longueur $m = |A|$ contenant les triplets (w, i, j) avec $0 \leq i < j < n$ et $w = p(\{i, j\})$ correspondant aux arêtes pondérées. On note $(a_k)_{k \in \llbracket 0, m-1 \rrbracket}$ les arêtes dans l'ordre de cette énumération.

Par exemple, si `g` représente le graphe de l'île de Korße représenté à la **figure 5**, on aura par exemple :

```
>>> g.nb_sommets
4
>>> g.arestes
[(6.0, 0, 1), (2.0, 0, 2), (7.0, 0, 3), (5.0, 1, 2), (8.0, 1, 3), (3.0, 2, 3)]
```

On représente une configuration par un couple (t, etat) . La première composante `t` est un entier valant 1 si c'est au joueur MaxDébit de jouer et -1 si c'est au joueur MinLatence de jouer. La deuxième composante `etat` est une liste de longueur m avec, pour $k \in \llbracket 0, m-1 \rrbracket$, `etat[k] == 1` si l'arête a_k a déjà été choisie par le joueur MaxDébit, `etat[k] == -1` si l'arête a_k a déjà été choisie par le joueur MinLatence et `etat[k] == 0` si l'arête a_k n'a pas encore été choisie.

Par exemple, la configuration finale de la **figure 5** est représentée par le couple :

```
(1, [-1, 1, -1, 1, 1, -1])
```

En effet, l'état est contrôlé par le joueur MaxDébit, les arêtes d'indices 1, 3 et 4 ont été choisies par le joueur MaxDébit et les arêtes d'indices 0, 2, 5 par le joueur MinLatence.

La fonction suivante permet de renvoyer la configuration initiale correspondant à un graphe `g` :

```
def configuration_initiale(g):
    etat = [0] * len(g.arestes)
    return (1, etat)
```

La fonction suivante permet de savoir quel joueur contrôle une configuration. Elle renvoie 1 si c'est à MaxDébit de jouer et -1 si c'est à MinLatence de jouer.

```
def tour(c):
    t, etat = c
    return t
```

Q32. Écrire en Python une fonction `finale(c)` qui prend une configuration `c` et qui renvoie `True` si et seulement si la configuration est finale, c'est-à-dire si la partie est terminée.

On rappelle que si `etat` est une liste Python, l'instruction `suiv = etat.copy()` permet de créer une copie de la liste `etat` dans la variable `suiv`.

Q33. Écrire en Python une fonction `configurations_suivantes(c)` qui, étant donné une configuration `c` supposée non finale, renvoie la liste de toutes les configurations accessibles en un coup à partir de cette configuration, c'est-à-dire les configurations obtenues lorsque le joueur qui contrôle cette configuration choisit une nouvelle arête. La configuration `c` ne doit pas être modifiée : il est nécessaire d'en faire des copies.

V.3 - Calcul des attracteurs

On suppose disposer d'une fonction `gagnant(g, c)` qui, étant donné un graphe g et une configuration finale c , renvoie 1, 0 ou -1 suivant que la configuration finale est gagnante pour MaxDébit, un match nul ou gagnante pour MinLatence. Pour cela, cette fonction va calculer les sous-graphes G_1 et G_{-1} obtenus par les deux joueurs, vérifier si ceux-ci sont connexes, le cas échéant en calculer les arbres couvrants de poids maximal, puis le poids de l'arête de poids minimal de ces arbres couvrants, en déduire $\bar{b}_{\text{lim}}(G_1)$ et $\bar{b}_{\text{lim}}(G_{-1})$, les comparer et enfin renvoyer le gagnant (ou 0 pour match nul).

La fonction suivante permet d'attribuer un score dans $\{-1, 0, 1\}$ à une configuration non nécessairement finale.

```

1 def score(g, c):
2     if est_finale(c):
3         return gagnant(g, c)
4     t, etat = c
5     score_fils = [score(g, suiv) for suiv in configurations_suivantes(c)]
6     if t == 1:
7         return max(score_fils)
8     else:
9         return min(score_fils)

```

Q34. Indiquer à quoi correspondent les configurations (non nécessairement finales) de score -1 , de score 0 et de score 1.

Q35. Considérons une configuration non finale de score 1. Expliquer comment construire une stratégie gagnante pour MaxDébit à partir de cette configuration. On ne demande pas d'implémenter cette fonction en Python ni de montrer que cette stratégie est effectivement gagnante, mais uniquement d'expliquer comment l'obtenir.

Les trois sous-parties suivantes sont complètement indépendantes.

V.4 - Memoïsation

L'implémentation précédente de la fonction `score` va conduire à recalculer un grand nombre de fois le score pour une même configuration. On se propose d'adopter une technique de mémoïsation. On se donne un dictionnaire `cache` qui sera, pour simplifier, une variable globale. Ce dictionnaire permet de mémoriser le score des configurations pour lesquelles celui-ci a déjà été calculé. Comme les listes ne peuvent pas être utilisées comme clés pour les dictionnaires, on utilisera la fonction `fige` ci-après pour convertir une configuration avec des listes en configuration avec des n -uplets avant de l'utiliser comme clé du dictionnaire `cache`.

```

def fige(c):
    t, etat = c
    return (t, tuple(etat))

```

Par exemple, en considérant la configuration obtenue après les trois premiers coups lors de la partie donnée en exemple dans la **sous-partie V.1**, on obtient :

```

>>> c_ex = (-1, [-1, 0, 0, 1, 1, 0])
>>> fige(c_ex)
(-1, (-1, 0, 0, 1, 1, 0))

```

Q36. Compléter le squelette de la fonction `score_memo` ci-dessous pour que celle-ci se comporte exactement comme la fonction `score` de la **sous-partie V.3**, mais en utilisant le dictionnaire `cache` pour ne pas recalculer plusieurs fois le score d'une même configuration.

```
cache = {}

def score_memo(g, c):
    # On peut utiliser dans cette fonction la variable globale `cache`
    ...
```

V.5 - Algorithme MinMax avec heuristique et exploration jusqu'à une profondeur p

Même en utilisant une fonction mémoisée, le coût du calcul du score de toutes les configurations avec l'approche précédente peut vite se révéler prohibitif. On se propose dans cette sous-partie d'utiliser l'algorithme MinMax avec une heuristique en limitant l'exploration jusqu'à une profondeur $p \in \mathbb{N}$. On suppose disposer d'une heuristique, c'est-à-dire une évaluation approximative de la qualité d'une configuration non finale sous la forme d'un score dans $[-1, 1]$, avec un score positif si la configuration semble favorable pour MaxDébit et un score négatif si la configuration semble favorable pour MinLatence. On suppose donc disposer d'une fonction `heuristique` telle que `heuristique(g, c)` pour un graphe `g` et une configuration non finale `c` renvoie un score dans $[-1, 1]$.

Q37. Écrire en Python une fonction `score_minmax(g, c, p)` sur le modèle de la fonction `score` de la **sous-partie V.3** (sans la mémoisation de la **sous-partie V.4**) telle que `score_minmax(g, c, p)` calcule un score pour une configuration `c`, mais en limitant l'exploration à une profondeur $p \in \mathbb{N}$, et en utilisant l'heuristique comme substitut lorsque l'on atteint la limite de profondeur.

V.6 - Vol de stratégie

Q38. Montrer que, quel que soit le graphe connexe à pondération injective, il n'existe pas de stratégie gagnante pour MinLatence. *On pourra remarquer qu'avoir un coup arbitraire en plus ne peut qu'être bénéfique et procéder par l'absurde en supposant que le deuxième joueur dispose d'une stratégie gagnante pour construire une stratégie gagnante pour le premier joueur.*

Ainsi, il existe pour MaxDébit une stratégie lui assurant de ne pas perdre : elle lui assure soit de gagner soit de forcer un match nul. Le gouvernement du Listenbourg semble avoir privilégié un des deux opérateurs.

FIN

EBE NSI 1

CAPES externe et CAFEP-CAPES
Section Numérique et Sciences Informatiques
Épreuve Disciplinaire – Session 2025

Autour du problème de la couverture par ensembles

Présentation du sujet. Ce sujet traite du problème algorithmique de la couverture par ensembles. Il se décompose en 7 parties. La partie 1 présente un problème concret permettant de fixer les idées sur ce qu'est le problème de la couverture par ensembles. Dans la partie 2 on modélise le problème, les instances et leurs solutions en PYTHON. La partie 3 se penche sur deux variantes d'un algorithme par brute force pour ce problème, tandis que la partie 4 propose un algorithme glouton. La partie 5 introduit la notion de solution partielle et ne contient pas de question. La partie 6 présente la notion de graphe associé à une solution partielle. La partie 7 présente le paradigme de séparation et évaluation et utilise les résultats précédemment obtenus afin de fournir une autre solution algorithmique au problème.

Il est tout à fait possible de sauter des questions ou des parties du sujet, il est toutefois nécessaire de lire toutes les questions et définitions pour comprendre les questions suivantes.

Le sujet est complété de deux annexes (page 17). L'annexe A précise les quelques notations mathématiques utilisées dans le sujet. L'annexe B fournit une description succincte de la librairie PYTHON typing utilisée tout au long du sujet pour les annotations de type.

Travail attendu. Si une fonction ou un résultat est introduit dans l'énoncé, vous pouvez l'utiliser dans les questions suivantes, y compris si vous n'avez pas proposé de définition de cette fonction, de justification de ce résultat. Dans la partie 1, les questions de programmation doivent être traitées en SQL. Dans les parties suivantes les questions de programmation doivent être traitées en PYTHON, et chacune d'elle est accompagnée de la signature de la fonction à implémenter (au besoin on se référera à l'annexe B).

On rappelle que les réponses aux questions de programmation doivent être compréhensibles. Une réponse même correcte risque de ne rapporter aucun point si le style ne permet pas une compréhension aisée de la solution proposée. Pour rendre vos programmes compréhensibles vous pouvez utiliser des noms de variables pertinents, des fonctions auxiliaires et des commentaires. Chaque fonction auxiliaire (comprendre, non explicitement demandée par l'énoncé) doit être accompagnée d'une description rapide de son comportement et du sens attaché à ses arguments. Si la correction de votre programme ou de votre algorithme repose sur une idée non triviale, il faut donner cette idée en français avant le code.

Lorsqu'il vous est demandé de fournir la complexité algorithmique pire cas d'une fonction, vous devez fournir le résultat sous la forme d'un \mathcal{O} et veiller à ne pas donner une majoration trop grossière (par exemple, on ne dira pas $\mathcal{O}(n^2)$ lorsque $\mathcal{O}(n)$ aurait suffit).

1. Un exemple introductif

On s'intéresse dans cette section à un problème concret d'organisation de révisions. Un groupe de personnes préparant le CAPES de NSI a recueilli des énoncés d'épreuves d'informatique de différents concours et cherche à sélectionner un ensemble d'énoncés qui couvre tout le programme du CAPES avec un minimum d'énoncés. Pour cela les différents énoncés ont été annotés à l'aide de mots-clés (par exemple *binaire*, *glouton*, *sql*, ...), et les différents mots-clés ont été raccrochés aux thèmes du programme du CAPES (par exemple *Algorithmique*, *Types construits*, ...). On suppose que chaque mot-clé n'est raccroché qu'à un seul thème du programme, et l'objectif est de couvrir chacun des thèmes au programme. Les données issues de ce travail documentaire ont été rassemblées dans quatre tables d'une base de données.

- La table *ProgrammeCapes* qui liste les thèmes au programme du CAPES de NSI.
Cette table contient un seul champ *Theme*.
- La table *Enonce* qui liste les énoncés disponibles.
Cette table contient cinq champs : *Id*, *Concours*, *Discipline*, *Annee* et *Epreuve*.
- La table *Balise* qui associe aux énoncés des mots-clés.
Cette table a deux champs *Id* et *MotCle*. On suppose que tous les identifiants *Id* apparaissant dans cette table sont des identifiants d'énoncés de la table *Enonce*.
- La table *Anrage* qui associe à un mot-clé le thème du programme auquel il se rapporte.
Cette table a deux champs *MotCle* et *Theme*. On suppose que tous les thèmes apparaissant dans cette table sont des thèmes au programme figurant dans la table *ProgrammeCapes*.

On donne ci-après quelques enregistrements extraits de ces tables.

Extrait de la table *ProgrammeCapes*

Theme
Algorithmique
Types et valeurs de base
Types construits

Extrait de la table *Enonce*

Id	Concours	Discipline	Annee	Epreuve
1	CAPES	NSI	2020	écrit 1
4	CAPES	NSI	2021	écrit 2
9	CAPES	NSI	2025	écrit 1
10	Agrégation	Informatique	2023	composition

Extrait de la table *Balise*

Id	MotCle
9	SQL
9	algorithmes gloutons
9	graphe
9	dictionnaire

Extrait de la table *Anrage*

MotCle	Theme
dictionnaire	Types construits
SQL	Bases de données
fichier csv	Traitement de données en table
arbres	Structures de données
terminaison	Algorithmique

Q. 1 Quelle requête SQL permet d'obtenir la liste des thèmes au programme du CAPES ?

Q. 2 Quelle requête donne le nombre d'années d'épreuve de CAPES présentes dans la base ? Le résultat de la requête doit être réduit à ce nombre.

Q. 3 Quelle requête SQL permet d'obtenir la liste, triée par ordre alphabétique, des mots-clés associés au thème *Algorithmique* qui apparaissent dans au moins un énoncé de la base ?

Q. 4 Quelle requête SQL permet d'obtenir la liste des thèmes couverts par les énoncés de la base de données ?

Q. 5 Quelle requête SQL permet de lister les thèmes au programme qui ne sont pas couverts par au moins un énoncé de la base de données ?

Grâce à la question précédente, on peut vérifier que la base de données contient assez d'énoncés pour couvrir tout le programme. On travaille sous cette hypothèse dans la question suivante.

Q. 6 Quelle requête SQL permet d'obtenir la liste qui associe aux identifiants d'énoncés du CAPES les thèmes qu'ils couvrent ? On veillera à éviter les doublons dans la liste résultat, par exemple si un énoncé couvre le thème *Algorithmique* parce qu'il est à la fois annoté par le mot-clé *gouloton* et le mot-clé *programmation dynamique*, l'association de ce sujet au thème *Algorithmique* ne doit apparaître qu'une seule fois dans le résultat.

2. Présentation du problème

Le problème concret présenté à la section précédente consiste, connaissant un ensemble de sujets et les thèmes qu'ils abordent, à trouver un sous-ensemble de ces sujets qui couvre l'ensemble du programme du CAPES de NSI.

Un exemple. Dans cet exemple, on suppose qu'il y a 12 thèmes à couvrir, numérotés de 0 à 11. On notera donc X_e l'ensemble des thèmes à couvrir, à savoir : $X_e = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$. On dispose de 6 sujets, numérotés de 0 à 5. Pour chaque sujet on connaît les thèmes qu'il couvre, par exemple le sujet 2 couvre les thèmes 0, 3, 6, 9. Ainsi chaque sujet donne un sous-ensemble des thèmes à couvrir, pour chaque $i \in \llbracket 0, 5 \rrbracket$, on note P_i le sous-ensemble de thèmes que couvre le sujet numéro i . En particulier $P_2 = \{0, 3, 6, 9\}$. L'ensemble des 6 sujets à disposition est donc un ensemble de sous-ensembles de thèmes, on le notera ici \mathcal{F}_e . Ainsi $\mathcal{F}_e = \{P_0, P_1, P_2, P_3, P_4, P_5\}$. La définition complète de cet exemple est fournie par la Figure 1, à la fois de manière schématique et de manière mathématique.

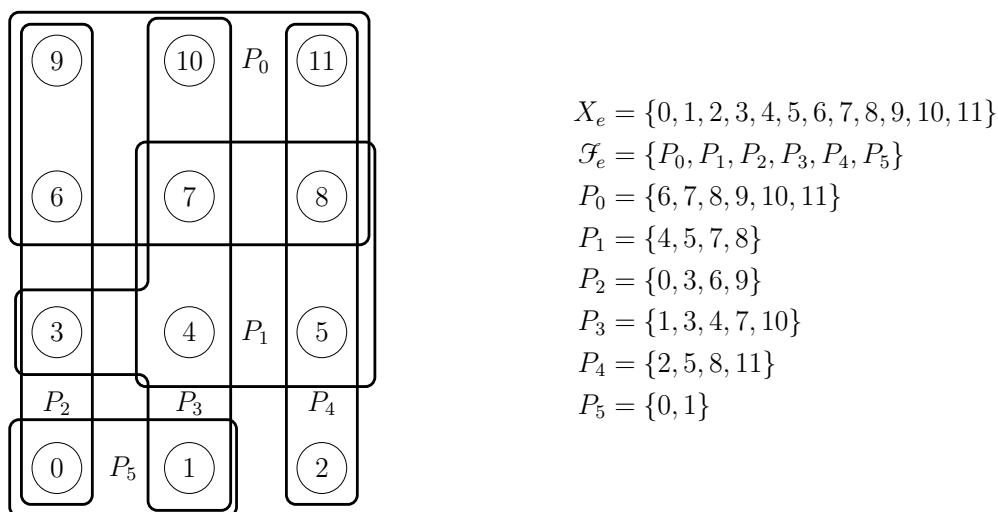


FIGURE 1 – L'exemple (X_e, \mathcal{F}_e)

La sélection des sujets P_0 et P_1 ne permet pas de couvrir tous les thèmes au programme : les thèmes 0, 1, 2 et 3 ne sont pas couverts. Au contraire, la sélection des sujets P_0, P_2, P_3 et P_4 couvre bien tous les thèmes au programme. Ainsi il est possible de couvrir tous les thèmes au programme avec 4 sujets, ce n'est pas cependant pas le minimum. En effet la sélection de sujets P_2, P_3 et P_4 couvre aussi le programme, en utilisant seulement 3 sujets.

Abstraction. Ce problème concret relève d'un problème d'optimisation connu sous le nom anglais SETCOVER que l'on traduit en COUVENS, pour Couverture par Ensemble. Étant donné un ensemble fini $X \clubsuit$ et un ensemble \mathcal{F} de sous-ensembles de $X \heartsuit$, on appelle *couverture* un sous-ensemble \mathcal{C} de \mathcal{F} tel que $X = \bigcup_{P \in \mathcal{C}} P \spadesuit$. Le problème COUVENS consiste à trouver une couverture la plus petite possible, à savoir, de plus petit cardinal \diamond . Ainsi une instance du problème COUVENS est la donnée d'un couple (X, \mathcal{F}) . La Figure 1 définissait l'instance (X_e, \mathcal{F}_e) qui sera utilisée en exemple tout au long du sujet.

Couvivable. Pour un instance (X, \mathcal{F}) , il se peut qu'il n'existe aucune couverture. Toutefois, dès lors que chaque élément de X apparaît dans au moins un ensemble de \mathcal{F} , il existe une couverture : par exemple celle sélectionnant tous les sous-ensembles possibles, à savoir \mathcal{F} . Cette contrainte s'exprime par le fait que $\bigcup_{P \in \mathcal{F}} P = X$, on dira alors que X est *couvivable* par \mathcal{F} .

Formalisation. On formalise les trois problèmes qui nous intéresseront dans ce sujet. Le premier est un problème de recherche de solution optimale, le deuxième un problème de calcul de la valeur optimale, et le dernier un problème de décision.

CouvENS : $\begin{cases} \textbf{Entrée} : \text{Un ensemble fini } X, \text{ un ensemble } \mathcal{F} \text{ de sous-ensembles de } X \text{ tel} \\ \quad \quad \quad \text{que } X \text{ est couvivable par } \mathcal{F}. \\ \textbf{Sortie} : \text{Une couverture } \mathcal{C} \text{ de } X \text{ par } \mathcal{F}, \text{ qui soit de cardinal minimal.} \end{cases}$

COUVENSTAILLE : $\begin{cases} \textbf{Entrée} : \text{Un ensemble fini } X, \text{ un ensemble } \mathcal{F} \text{ de sous-ensembles de } X \text{ tel} \\ \quad \quad \quad \text{que } X \text{ est couvivable par } \mathcal{F}. \\ \textbf{Sortie} : \text{Le cardinal minimal d'une couverture de } X \text{ par } \mathcal{F}. \end{cases}$

COUVENSSEUIL : $\begin{cases} \textbf{Entrée} : \text{Un ensemble fini } X, \text{ un ensemble } \mathcal{F} \text{ de sous-ensembles de } X \text{ tel} \\ \quad \quad \quad \text{que } X \text{ est couvivable par } \mathcal{F}, \text{ un entier } K \in \mathbb{N}. \\ \textbf{Sortie} : \text{Existe-t-il } \mathcal{C} \text{ une couverture de } X \text{ par } \mathcal{F}, \text{ de cardinal } \leqslant K? \end{cases}$

On donne ci-dessous des exemples de relation entrée/sortie pour ces trois problèmes.

- Pour le problème COUVENS sur l'instance (X_e, \mathcal{F}_e) , une solution peut être l'ensemble $\{P_2, P_3, P_4\}$ qui est bien une couverture de cardinal minimal. C'est en fait la seule solution possible.
- Pour le problème COUVENSTAILLE sur l'instance (X_e, \mathcal{F}_e) , l'unique solution est 3.
- Pour le problème COUVENSSEUIL sur l'instance $(X_e, \mathcal{F}_e, 4)$, l'unique solution est vrai (**True**) tandis que sur l'instance $(X_e, \mathcal{F}_e, 2)$, l'unique solution est faux (**False**).

♣. les thèmes dans notre exemple

♡. l'ensemble des sujets dans notre exemple, chaque sujet étant un ensemble de thèmes

♠. une sélection de sujets couvrant tous les thèmes dans notre exemple

◊. dans notre exemple, on cherche une sélection, la plus petite possible, c'est-à-dire avec le moins de sujets possible

Q. 7 On considère l'instance $X = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, et $\mathcal{F} = \{Q_0, Q_1, Q_2, Q_3, Q_4\}$ où

- $Q_0 = \{0, 1, 3, 4, 6, 7\}$,
- $Q_1 = \{1, 2, 4, 5, 7\}$,
- $Q_2 = \{7, 8\}$,
- $Q_3 = \{6, 8\}$,
- $Q_4 = \{0, 1, 3, 4, 5\}$.

Donner des sorties pour les problèmes COUVENS et COUVENSTAILLE sur l'entrée (X, \mathcal{F}) ainsi que pour COUVENSSEUIL sur l'entrée $(X, \mathcal{F}, 2)$.

2.1. Manipulation des problèmes

- Q. 8** On suppose fourni, pour cette question seulement, un algorithme `couv_ens_taille` résolvant COUVENSTAILLE, c'est-à-dire, prenant en arguments un ensemble X et un ensemble \mathcal{F} de parties de X et renvoyant le cardinal d'une couverture de X de cardinal minimal. Proposer le pseudo-code d'un algorithme utilisant `couv_ens_taille` et résolvant COUVENSSEUIL. Donner le nombre d'appels à `couv_ens_taille` effectués en fonction des valeurs d'entrées X, \mathcal{F} et K .
- Q. 9** On suppose fourni, pour cette question seulement, un algorithme `couv_ens_seuil` résolvant COUVENSSEUIL, c'est-à-dire prenant en arguments un ensemble X , un ensemble \mathcal{F} de parties de X et un entier K et renvoyant un booléen indiquant s'il existe une couverture de X par \mathcal{F} de cardinal $\leq K$.
- Proposer le pseudo-code d'un algorithme utilisant `couv_ens_seuil`, résolvant le problème COUVENSTAILLE. Cet algorithme devra effectuer un nombre le plus faible possible d'appels à `couv_ens_seuil`.
 - Donner, sans justifier, en fonction de X et \mathcal{F} , un ordre de grandeur du nombre d'appels à `couv_ens_seuil` effectués par cet algorithme.
 - Proposer un invariant de boucle, qui justifierait la correction de cet algorithme.

2.2. Représentation en machine

Restriction. Dans tout le reste du sujet on se limite au cas où X est un intervalle d'entiers de la forme $\{0, 1, \dots, n - 1\} = \llbracket 0, n - 1 \rrbracket$ pour un certain entier $n \in \mathbb{N}$.

Représentation en machine. On présente ici les différentes manières dont les objets manipulés dans ce sujet seront encodés en PYTHON.

- Étant donné que l'on se restreint au cas où $X = \llbracket 0, n - 1 \rrbracket$ pour un certain n , l'entrée X sera représentée par un entier n .
- Une partie de X sera représentée par la liste de ses éléments. Bien que dans tous les exemples les éléments d'une partie $P \in \mathcal{F}$ soient listés dans l'ordre croissant, ceci n'est pas requis. On n'utilisera pas cette hypothèse par la suite. L'ensemble \mathcal{F} des parties de X autorisées pour la couverture sera alors représenté par la liste de ses éléments, de type `List[List[int]]`. Aussi dans la suite, on suppose défini le type ci-dessous.

```
1 | Parties = List[List[int]]
```

Ainsi les données X_e et \mathcal{F}_e de l'exemple seront représentées en PYTHON par les valeurs `n_ex` et `f_ex` du code ci-dessous.

```
1 | p0_ex = [6, 7, 8, 9, 10, 11]
2 | p1_ex = [4, 5, 7, 8]
3 | p2_ex = [0, 3, 6, 9]
4 | p3_ex = [1, 3, 4, 7, 10]
5 | p4_ex = [2, 5, 8, 11]
```

```

6 | p5_ex = [0, 1]
7 | f_ex  = [p0_ex, p1_ex, p2_ex, p3_ex, p4_ex, p5_ex]
8 | n_ex  = 12

```

Dans la suite, lorsqu'une fonction prendra en arguments n et f représentant une instance (X, \mathcal{F}) , on supposera que n est positif et que les listes de f sont bien à valeurs dans $\llbracket 0, n - 1 \rrbracket$.

- Pour manipuler les couvertures, on s'autorisera deux encodages :

- Un sous-ensemble \mathcal{C} de $f = [P_0, P_1, \dots, P_{m-1}]$ pourra être représenté au moyen d'une liste des indices dans la liste \mathcal{F} des parties présentes dans \mathcal{C} . Un tel objet est alors de type `List[int]`. On dira alors que \mathcal{C} est représenté par *liste d'indices*. Par exemple le sous-ensemble $\mathcal{C}_1 = \{P_0, P_3, P_4, P_2\}$ de l'exemple pourra être représenté par la liste `[0, 4, 3, 2]` ou encore `[2, 3, 4, 0]`.
- Un sous-ensemble \mathcal{C} de $f = [P_0, P_1, \dots, P_{m-1}]$ pourra aussi être représenté par un tableau de booléens t , de taille $m = |\mathcal{F}|$, indiquant dans chaque case i si P_i est présent ou non dans \mathcal{C} . Un tel objet est alors de type `List[bool]`. On dira alors que \mathcal{C} est représenté par sa *fonction indicatrice*. Par exemple le sous-ensemble $\mathcal{C}_1 = \{P_0, P_3, P_4, P_2\}$ de l'exemple sera représenté par le tableau `[True, False, True, True, False]`.

- Q. 10** Définir une fonction `card1` prenant en argument un sous-ensemble \mathcal{C} , représenté par liste d'indices, et renvoyant son cardinal.

```
card1(c: List[int]) -> int
```

- Q. 11** Définir une fonction `card2` prenant en argument un sous-ensemble \mathcal{C} , représenté par fonction indicatrice, et renvoyant son cardinal.

```
card2(c: List[bool]) -> int
```

2.3. Vérification des solutions

- Q. 12** Définir une fonction `verification1` prenant en arguments n et f représentant une instance (X, \mathcal{F}) et c représentant un sous-ensemble \mathcal{C} de \mathcal{F} par liste d'indices, et renvoyant si \mathcal{C} est une couverture de X par \mathcal{F} .

```
verification1(n: int, f: Parties, c: List[int]) -> bool
```

- Q. 13** En utilisant les notations $X = \llbracket 0, n - 1 \rrbracket$, $\mathcal{F} = \{P_0, P_1, \dots, P_{m-1}\}$, pour tout $i \in \llbracket 0, m - 1 \rrbracket$, $n_i = |P_i|$, $l = \sum_{i=0}^{m-1} n_i$, et finalement $q = |\mathcal{C}|$, donner la complexité algorithmique pire cas de la fonction `verification1` proposée à la question précédente (**Q. 12**). On attend une justification succincte.

- Q. 14** Définir une fonction `verification2` prenant en arguments n et f représentant une instance (X, \mathcal{F}) et c représentant un sous-ensemble \mathcal{C} de \mathcal{F} par sa fonction indicatrice, et renvoyant si \mathcal{C} est une couverture de X par \mathcal{F} .

```
verification2(n: int, f: Parties, c: List[bool]) -> bool
```

- Q. 15** En utilisant les notations de la **Q. 13** donner la complexité algorithmique pire cas de la fonction `verification2` (**Q. 14**). On attend une justification succincte.

- Q. 16** Définir une fonction `est_couvrible` prenant en arguments n et f représentant une instance (X, \mathcal{F}) et testant si X est couvrable par \mathcal{F} .

```
est_couvrible(n: int, f: Parties) -> bool
```

Important : dans toute la suite on s'intéresse à des instances (X, \mathcal{F}) où X est couvrable par \mathcal{F} .

2.4. Obtenir des instances en PYTHON

Dans cette section, on envisage l'import d'une instance du problème concret de la section 1 sous la forme d'un couple (n, f) où n est de type `int` et f de type `Parties` comme expliqué plus haut.

On suppose que la table de données obtenue à la question 6 a été exportée au format csv dans un fichier nommé `assoc_id_theme.csv`, dont les premières lignes sont reproduites ci-contre. Ce fichier texte contient un enregistrement par ligne, les valeurs sont séparées par des virgules.

assoc_id_theme.csv
9,Algorithmique
3,Algorithmique
5,Algorithmique
9,Bases de données

Un tel formatage permet d'extraire les données en PYTHON grâce au module `csv`, et en particulier grâce à la fonction `csv.reader`. On donne ici un exemple d'utilisation de cette fonction très proche de celui fourni par la documentation du module `csv`. Le fichier `oeufs.csv` contient le texte ci-dessous.

oeufs.csv
brouillés, au plat, au plat, miroir
bénédicte, au plat, coque, parfait, mollet

Le code ci-dessous importe dans l'objet `filereader` les données de la table enregistrée dans le fichier `oeufs.csv`, ainsi itérer sur `filereader` revient à itérer sur les lignes du fichier, soit dans notre cas sur les enregistrements de la table.

```
>>> import csv
>>> with open('oeufs.csv', newline='') as csvfile:
...     filereader = csv.reader(csvfile, delimiter=',')
...     for row in filereader:
...         for i in range(len(row)):
...             print(row[i], end=' / ')
...     print()
brouillés / au plat / au plat / miroir /
bénédicte / au plat / coque / parfait / mollet /
```

Q. 17 À l'aide de l'exemple de code fourni ci-dessus, proposer le code d'un programme PYTHON qui extrait du fichier `assoc_id_theme.csv` un dictionnaire nommé `dico_id` qui associe à chaque identifiant d'énoncé (de type `int`) la liste des thèmes (de type `List[str]`) que couvre cet énoncé.

De manière similaire on pourrait extraire du fichier `assoc_id_theme.csv` un dictionnaire nommé `num_theme` de type `Dict[str, int]` qui associe à chaque thème un numéro unique entre 0 et $n - 1$ où n est le nombre de thèmes en jeu.

Q. 18 Définir une fonction `cree_instance` prenant en arguments `dico` un dictionnaire qui associe aux identifiants d'énoncés la liste des thèmes qu'ils couvrent et `num` un dictionnaire numérotant les thèmes apparaissant dans `dico` et renvoyant le couple (n, f) où n est le nombre de thèmes à couvrir et où f est une liste de listes de thèmes couverts par un même énoncé. Dans f , les thèmes doivent être représentés par leur numéro, soit un entier entre 0 et $n - 1$.

```
cree_instance (dico: Dict[int, List[str]], num: Dict[str, int]) ->
    Tuple[int, Parties]
```

3. Un algorithme par brute force

Dans cette section on se concentre sur la mise en place d'un algorithme de résolution du problème CouvEns par brute force : étant donné une instance (X, \mathcal{F}) , on parcourt l'espace de tous les choix de $\mathcal{C} \subseteq \mathcal{F}$, pour trouver une couverture de cardinal minimal.

3.1. Version naïve

Pour cette première approche on choisit de représenter les couvertures $\mathcal{C} \subseteq \mathcal{F}$ par leur fonction indicatrice.

Q. 19 Définir une fonction suivant prenant en argument un tableau de m booléens représentant la décomposition en base 2 d'un entier e de $\llbracket 0, 2^m - 1 \rrbracket$ (les bits de poids faibles sont à droite), et modifiant le tableau pour :

- qu'il contienne la décomposition en base 2 de l'entier $e + 1$, dans le cas où $e < 2^m - 1$;
- qu'il contienne uniquement des **False** sinon.

Dans le premier cas, cette fonction renvoie **True**, dans le second elle renvoie **False**.

```
suivant(cand: List[bool]) -> bool
```

Q. 20 Définir une fonction couv_ens_naif prenant en arguments n et f représentant une instance (X, \mathcal{F}) et renvoyant un couple (q, c) tel que c est la représentation par fonction indicatrice d'une couverture de cardinal minimal de X par \mathcal{F} et q est le cardinal de cette couverture.

```
couv_ens_naif(n: int, f: Parties) -> Tuple[int, List[bool]]
```

3.2. Énumération par cardinal croissant

Dans le problème CouvEns, on cherche un sous-ensemble de cardinal minimal, aussi on s'intéresse dans cette partie à énumérer les sous-ensembles d'un ensemble \mathcal{F} fixé par cardinaux croissants, afin d'arrêter la recherche dès que l'on trouve une couverture. Pour cette seconde approche on choisit de représenter les couvertures $\mathcal{C} \subseteq \mathcal{F}$ par leur liste d'indices.

On rappelle que le nombre de sous-ensembles de cardinal k d'un ensemble de cardinal m est le coefficient binomial $\binom{m}{k}$. Les coefficients binomiaux vérifient les relations de récurrence ci-dessous.

$$\forall m \in \mathbb{N}, \forall k \in \mathbb{N}, \binom{m}{k} = \begin{cases} 1 & \text{si } k = 0 \\ \binom{m-1}{k-1} + \binom{m-1}{k} & \text{si } k \in \llbracket 1, m \rrbracket \\ 0 & \text{sinon} \end{cases}$$

Afin d'énumérer, par cardinaux croissants, les sous-ensembles d'un ensemble de cardinal m , nous allons nous reposer sur la connaissance des coefficients $\binom{p}{k}$ pour tout $p \in \llbracket 0, m \rrbracket$ et tout $k \in \llbracket 0, m \rrbracket$. L'objectif de la question suivante est de pré-calculer efficacement ces coefficients.

Q. 21 Définir une fonction binomiaux prenant en argument un entier m et renvoyant une matrice b , indiquée par les entiers $\llbracket 0, m \rrbracket \times \llbracket 0, m \rrbracket$, telle que pour tout $p \in \llbracket 0, m \rrbracket$ et tout $k \in \llbracket 0, m \rrbracket$, $b[p][k]$ contient le coefficient $\binom{p}{k}$. On précisera, sans la justifier, la complexité algorithmique pire cas de la fonction implémentée.

```
binomiaux(m: int) -> List[List[int]]
```

Pour former un sous-ensemble de cardinal $k \neq 0$ de l'ensemble $\{0, 1, \dots, m - 1\}$ on peut :

- former un ensemble à $k - 1$ éléments de $\{0, 1, \dots, m - 2\}$ et y ajouter l'élément $m - 1$;
- ou bien former un ensemble à k éléments de $\{0, 1, \dots, m - 2\}$.

On remarque qu'on peut ainsi former $\binom{m-1}{k-1}$ ensembles de type a) et $\binom{m-1}{k}$ ensembles de type b).

Le but de la question suivante est d'utiliser les remarques précédentes pour construire une fonction `ensemble_num_i` qui, pour trois entiers i , k et m donnés, retourne le i -ème sous-ensemble à k éléments de l'ensemble $\llbracket 0, m - 1 \rrbracket$, où i -ème s'entend pour une numérotation bien choisie. La Figure 2 donne la numérotation des $10 = \binom{4}{2}$ sous-ensembles à 3 éléments de $\llbracket 0, 4 \rrbracket$. Chaque flèche de l'arbre correspond à une décision : à gauche on choisit de former un sous-ensemble de type a), à droite on choisit de former un sous-ensemble de type b). Chaque chemin de la racine à une feuille correspond à une suite de décisions qui forment un ensemble de taille 3, indiqué au dessous. Les ensembles formés sont alors numérotés de gauche à droite.

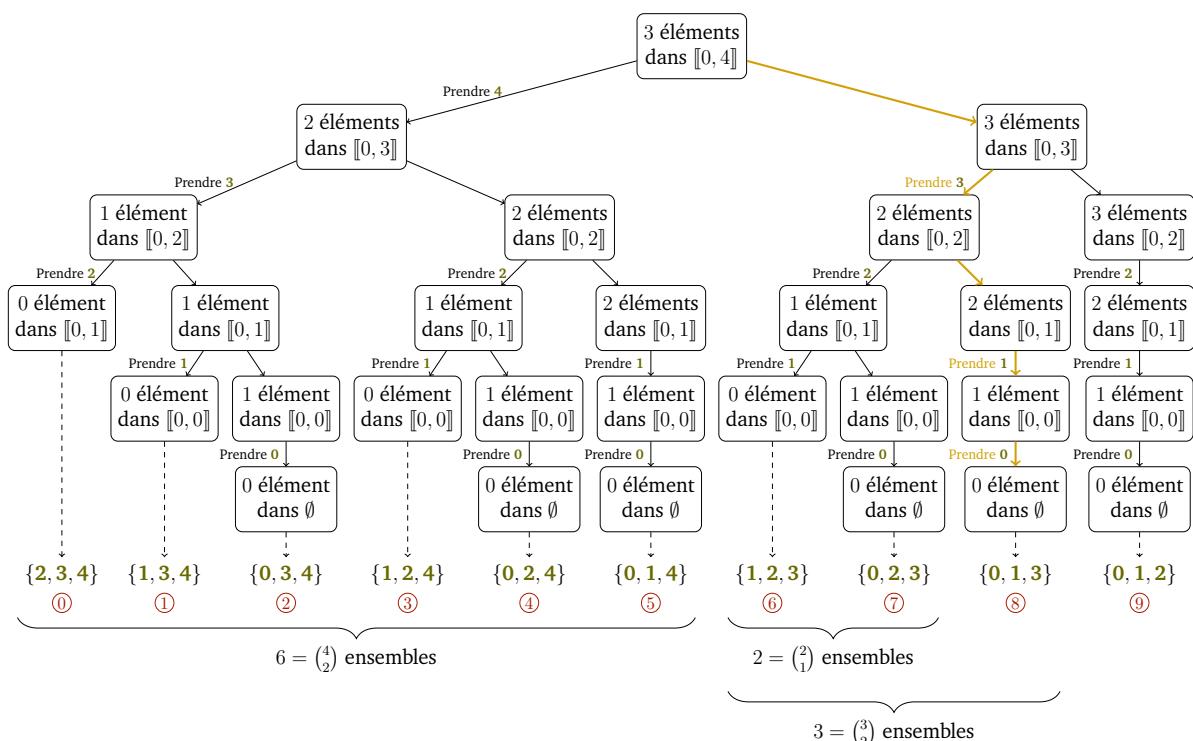


FIGURE 2 – Numérotation des sous-ensembles à 3 éléments de $\llbracket 0, 4 \rrbracket$.

Exemple. On cherche à construire l'ensemble de numéro ⑧ parmi les ensembles à 3 éléments de $\llbracket 0, 4 \rrbracket$.

- Comme l'indique la Figure 2, il y a $\binom{4}{2} = 6$ ensembles à 3 éléments de $\llbracket 0, 4 \rrbracket$ qui contiennent le nombre 4. Autrement dit, en choisissant de construire un ensemble de type a), on obtiendrait un ensemble de numéro entre ① et ⑤. Aussi choisit-on de construire un ensemble de type b), c'est-à-dire ne contenant pas 4. On cherche donc l'ensemble de numéro $8 - 6 = 2$ parmi les ensembles à 3 éléments de $\llbracket 0, 3 \rrbracket$.
- Comme l'indique la Figure 2, il y a $\binom{3}{2} = 3$ ensembles à 3 éléments de $\llbracket 0, 3 \rrbracket$ qui contiennent le nombre 3. Autrement dit, en choisissant de construire un ensemble de type a), on obtiendrait un ensemble de numéro entre 0 et 2. Aussi choisit-on ici de construire un ensemble de type a), c'est-à-dire contenant 3. On cherche donc l'ensemble de numéro 2 parmi les ensembles à 2 éléments de $\llbracket 0, 2 \rrbracket$, sachant qu'on lui ajoutera 3.

- Comme l'indique la Figure 2, il y a $\binom{2}{1} = 2$ ensembles à 2 éléments de $\llbracket 0, 2 \rrbracket$ qui contiennent le nombre **2**. Autrement dit, en choisissant de construire un ensemble de type *a*), on obtiendrait un ensemble de numéro entre 0 et 1. Aussi choisit-on de construire un ensemble de type *b*), c'est-à-dire ne contenant pas **2**. On cherche donc l'ensemble de numéro $2 - 2 = 0$ parmi les ensembles à 2 éléments de $\llbracket 0, 1 \rrbracket$.
- Il y a $\binom{1}{1} = 1$ ensemble à 2 éléments de $\llbracket 0, 1 \rrbracket$ qui contient le nombre **1**. Aussi choisit-on de construire un ensemble de type *a*), c'est-à-dire contenant **1**. On cherche donc l'ensemble de numéro 0 parmi les ensembles à 1 éléments de $\llbracket 0, 0 \rrbracket$, sachant qu'on lui ajoutera **1**.
- Il y a $\binom{1}{1} = 1$ ensemble à 1 élément de $\llbracket 0, 0 \rrbracket$ qui contient le nombre **0**. Aussi choisit-on de construire un ensemble de type *a*), c'est-à-dire contenant **0**. Il reste donc à trouver l'ensemble de numéro 0 parmi les ensembles à 0 élément de $\llbracket 0, 0 \rrbracket$, sachant qu'on lui ajoutera **0**. Un tel ensemble est l'ensemble vide.

Finalement on ajoute **3**, **1** et **0** à l'ensemble vide, construisant ainsi l'ensemble **{0, 1, 3}**.

Q. 22 Définir une fonction `ensemble_num_i` prenant en arguments :

- trois entiers naturels *i*, *k* et *m* vérifiant $k \leq m$ et $i \in \llbracket 0, \binom{m}{k} - 1 \rrbracket$ et
- une matrice *b* suffisamment grande pour contenir les coefficients binomiaux $\binom{r}{s}$ pour tout couple (r, s) de $\llbracket 0, m \rrbracket^2$.

et renvoyant l'ensemble de numéro *i* parmi les ensembles à *k* éléments de $\llbracket 0, m - 1 \rrbracket$.

```
ensemble_num_i(i: int, k: int, m: int, b: List[List[int]]) -> List[int]
```

Q. 23 En déduire une fonction `couv_ens_binom` prenant en arguments *n* et *f* représentant une instance (X, \mathcal{F}) et renvoyant un couple (*q*, *c*) tel que *c* est la représentation par liste d'indices d'une couverture de cardinal minimal de *X* par \mathcal{F} et *q* est le cardinal de cet ensemble.

```
couv_ens_binom(n: int, f: Parties) -> Tuple[int, List[int]]
```

4. Un algorithme glouton

Éléments couverts. Lorsque (X, \mathcal{F}) est une instance d'un problème de couverture, et que $\mathcal{C} \subseteq \mathcal{F}$ est un sous-ensemble de \mathcal{F} , on dit d'un élément $x \in X$ qu'il est *couvert par* \mathcal{C} dès lors qu'il existe une partie $P \in \mathcal{C}$ telle que $x \in P$. Naturellement, \mathcal{C} est une couverture de *X*, si et seulement si tout élément de *X* est couvert par \mathcal{C} .

Les algorithmes de résolution du problème COUVENS considérés jusqu'ici ont des complexités algorithmiques exponentielles. On se propose ici d'étudier l'algorithme glouton suivant.

- On initialise l'ensemble \mathcal{C} à l'ensemble vide.
- Tant qu'il reste des éléments de *X* qui ne sont pas couverts par \mathcal{C} :
 - on choisit dans \mathcal{F} l'ensemble *P* ayant le plus d'éléments non encore couverts par \mathcal{C} ,
 - on ajoute *P* à \mathcal{C} .

On donne ci-dessous le pseudo-code de cet algorithme.

Algorithme 1 : Glouton**Entrée :** Une instance (X, \mathcal{F}) .**Sortie :** Calcule une couverture de X par \mathcal{F}

```

1  $\mathcal{C} \leftarrow \emptyset;$                                 // La solution en cours de construction
2  $R \leftarrow X;$                                          //  $R$  pour Reste à couvrir
3 tant que  $R \neq \emptyset$  faire
4   Choisir  $P \in \mathcal{F}$  tel que  $|P \cap R|$  soit maximal;
5    $\mathcal{C} \leftarrow \{P\} \cup \mathcal{C};$ 
6    $R \leftarrow R \setminus P;$ 
7 renvoyer  $\mathcal{C}$ 

```

4.1. Non optimalité de l'algorithme glouton

Q. 24 Donner une instance (X, \mathcal{F}) sur laquelle l'algorithme glouton renvoie une couverture de X qui n'est pas une solution optimale (de cardinal non minimal donc). Détaillez l'exécution de l'algorithme glouton sur cette instance, justifier que la solution obtenue n'est pas optimale.

La question précédente justifie que cet algorithme glouton ne répond pas de manière exacte au problème COUVENS. Dans la suite on cherche à quantifier ce défaut d'optimalité. Plus précisément, on montre que le rapport entre la valeur de la solution fournie par l'algorithme glouton (le cardinal de l'ensemble des parties sélectionnées) et la valeur optimale n'est pas borné. Pour cela, il suffit d'exhiber, pour tout $k \in \mathbb{N}$, une instance (X_k, \mathcal{F}_k) telle qu'en notant \mathcal{C}^g la solution renvoyée par l'algorithme glouton et en notant \mathcal{C}^* une solution optimale, $\frac{|\mathcal{C}^g|}{|\mathcal{C}^*|} \geq \frac{k}{2}$.

Étant donné un entier $k \in \mathbb{N}$, on considère :

- l'ensemble $X_k = \{0, 1, \dots, 2^{k+1} - 3\}$;
- les parties $E_k = \{0, 2, \dots, 2^{k+1} - 4\}$ et $O_k = \{1, 3, \dots, 2^{k+1} - 3\}$ ♦;
- et finalement $A_k = \{2^k - 2, 2^k - 1, 2^k, \dots, 2^{k+1} - 3\}$.

Ces parties sont représentées sur le schéma ci-dessous.

E_k	0 2 4 6 ... $2^k - 4$	$2^k - 2$ 2^k ... $2^{k+1} - 4$	A_k
O_k	1 3 5 7 ... $2^k - 3$	$2^k - 1$ $2^k + 1$... $2^{k+1} - 3$	

Q. 25 a) Donner les cardinaux des ensembles X_k , E_k , O_k et A_k .

b) Quel est le comportement de l'algorithme glouton sur l'instance $(X_k, \{E_k, O_k, A_k\})$?

Q. 26 a) Proposer un ensemble \mathcal{F}_k contenant au moins E_k, O_k, A_k , tel que l'algorithme glouton renvoie, sur l'instance (X_k, \mathcal{F}_k) , une solution de cardinal k . Expliciter le déroulé de l'algorithme sur l'instance proposée.
b) Conclure.

♦. E pour Even : pair en anglais, et O pour Odd : impair en anglais

4.2. Correction de l'algorithme glouton

On admet dans cette section que la propriété " $R \subseteq X$ " est un invariant de la boucle **Tant que** de l'algorithme glouton 1.

Q. 27 Démontrer, au moyen d'un variant bien choisi, la terminaison de l'algorithme glouton.

Q. 28 Démontrer, au moyen d'un invariant bien choisi, la correction de l'algorithme glouton. Il faut donc démontrer que, pour une instance (X, \mathcal{F}) , l'ensemble \mathcal{C} renvoyé par l'algorithme sur cette instance est une couverture de X .

5. Notion de solution partielle

Dans les sections qui suivent, on s'intéresse à des algorithmes de résolution du problème CouvEns qui nécessitent de représenter une suite de décisions prises lors de la fabrication d'une solution. En effet un algorithme de résolution du problème CouvEns pourrait procéder de la manière suivante.

- sélectionne-t-on P_1 dans la solution ?
 - Si oui, sélectionne-t-on P_2 dans la solution ?
 - ▷ Si oui, ...
 - ▷ Si non, ...
 - Si non, sélectionne-t-on P_2 dans la solution ?
 - ▷ Si oui, ...
 - ▷ Si non, ...

Dans un tel algorithme il est nécessaire de représenter ce que nous appellerons des solutions partielles : par exemple on a décidé que P_1 serait dans la solution, on a décidé que P_2 ne serait pas dans la solution, pour les autres éléments on n'a pas encore décidé.

Solution partielle. Étant donné une instance (X, \mathcal{F}) du problème CouvEns, on appelle *solution partielle*, la donnée de deux ensembles $\mathcal{O} \subseteq \mathcal{F}$ et $\mathcal{N} \subseteq \mathcal{F}$ tels que $\mathcal{O} \cap \mathcal{N} = \emptyset$.

- \mathcal{O} représente l'ensemble des parties que l'on a sélectionnées dans la solution en construction,
- \mathcal{N} représente l'ensemble des parties que l'on a rejetées pour cette solution.

\mathcal{F} étant représenté par une liste PYTHON $F = [P_0, P_1, \dots, P_{m-1}]$, une solution partielle sera représentée en PYTHON au moyen d'un tableau t à valeurs dans **True**, **False** ou **None**. \mathcal{O} est alors l'ensemble des P_i tels que $t[i]$ vaut **True**, \mathcal{N} est l'ensemble des P_i tels que $t[i]$ vaut **False**. Ainsi une solution partielle est représentée en PYTHON par le type suivant.

```
1 | Partiel = List[Optional[bool]]
```

Aussi, toujours avec l'exemple (X_e, \mathcal{F}_e) , la solution partielle dans laquelle on a sélectionné P_0 , mais rejeté P_3 sera représentée par le tableau PYTHON **[True, None, None, False, None, None]**.

Solution compatible. On dit d'une solution \mathcal{C} du problème pour (X, \mathcal{F}) qu'elle est *compatible* avec une solution partielle $(\mathcal{O}, \mathcal{N})$ dès lors que $\mathcal{O} \subseteq \mathcal{C}$ et $\mathcal{C} \cap \mathcal{N} = \emptyset$. Par exemple, pour l'instance (X_e, \mathcal{F}_e) , la solution partielle $(\mathcal{O} = \{P_3\}, \mathcal{N} = \{P_0\})$ représente les choix : P_3 est sélectionnée, P_0 est rejetée. Ainsi la solution $\{P_2, P_3, P_4\}$ est compatible avec cette solution partielle, en revanche la solution $\{P_0, P_1, P_4, P_5\}$ ne l'est pas, à double titre : d'une part car P_3 n'est pas présente et d'autre part car P_0 est présente.

6. Graphe associé à une solution partielle

Un sous-ensemble \mathcal{E} de parties de X induit une relation binaire sur les éléments de X : deux éléments x et y de X sont en relation dès lors qu'il existe un ensemble $P \in \mathcal{E}$ tel que x et y sont dans P . Une telle relation définit un graphe non orienté sur l'ensemble de sommets X . L'étude de tels graphes est l'objet de cette section.

Graphe associé à une solution partielle. Étant donné une instance (X, \mathcal{F}) , on définit le graphe associé à une solution partielle $(\mathcal{O}, \mathcal{N})$, comme étant le graphe non orienté $G = (S, A)$ où

- S est l'ensemble des éléments de X non couverts par \mathcal{O} (autrement dit, $S = X \setminus \bigcup_{P \in \mathcal{O}} P$) et
- A est l'ensemble des arêtes reliant deux sommets dès lors qu'ils sont contenus dans un même ensemble encore autorisé (autrement dit, $\{x, y\} \in A$ si et seulement s'il existe $P \in \mathcal{F} \setminus \mathcal{N}$ tel que $x \in P$ et $y \in P$).

En reprenant l'instance (X_e, \mathcal{F}_e) , le graphe associé à la solution partielle $\mathcal{O} = \{P_3\}$ et $\mathcal{N} = \{P_0\}$ est le graphe $G_e = (S, A)$ représenté en figure 3 d'ensemble de sommets $S = \{0, 2, 5, 6, 8, 9, 11\}$ et d'ensemble d'arêtes $A = \{\{0, 6\}, \{0, 9\}, \{2, 5\}, \{2, 8\}, \{2, 11\}, \{5, 8\}, \{5, 11\}, \{6, 9\}, \{8, 11\}\}$.

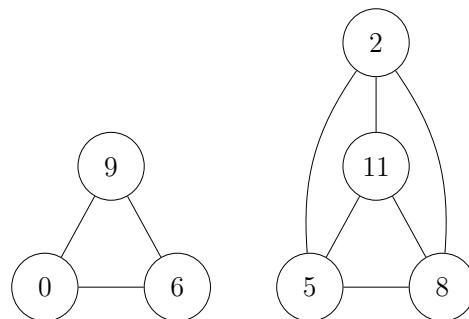


FIGURE 3 – Graphe G_e associé à la solution partielle $(\{P_3\}, \{P_0\})$ de l'instance (X_e, \mathcal{F}_e)

Représentation des graphes. Un graphe non orienté $G = (S, A)$, où $S = \llbracket 0, v-1 \rrbracket$ pour un certain $v \in \mathbb{N}$, sera représenté par listes d'adjacence. On choisit ici d'utiliser un dictionnaire PYTHON qui associe à chaque sommet du graphe l'ensemble PYTHON de ses voisins. Aussi dans la suite on suppose défini le type suivant.

1 | Graph = Dict[int, Set[int]]

Le graphe de la figure 3 est donc représenté en PYTHON par le dictionnaire suivant.

{0: {9, 6}, 2: {8, 11, 5}, 5: {8, 2, 11},
6: {0, 9}, 8: {2, 11, 5}, 9: {0, 6}, 11: {8, 2, 5}}

Q. 29 Définir une fonction `fabrique_graphe` prenant en arguments une instance (X, \mathcal{F}) du problème CouvENS et une solution partielle `sol_partielle` et calculant le graphe associé à cette solution partielle.

`fabrique_graphe(n: int, f: Parties, sol_partielle: Partiel) -> Graph`

6.1. Utilisation du graphe pour la décomposition en sous-problèmes

Q. 30 Définir une fonction PYTHON composantes_connexes prenant en argument un graphe et renvoyant la liste de ses composantes connexes. Une composante connexe sera représentée au moyen d'un ensemble.

```
composantes_connexes(g: Graph) -> List[Set[int]]
```

Dans le reste de cette sous-section, on fixe une instance (X, \mathcal{F}) du problème CouvEns. On considère le graphe $G = (S, A)$ associé à la solution partielle vide (\emptyset, \emptyset) . Connaissant $X = X_1 \cup X_2 \cup \dots \cup X_p$, la décomposition en composantes connexes de G , on cherche à décomposer \mathcal{F} en $\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_p$, de sorte que résoudre les instances (X_i, \mathcal{F}_i) permet de résoudre l'instance initiale.

- Q. 31** *a)* Proposer une définition des \mathcal{F}_i .
b) Justifier succinctement que les \mathcal{F}_i sont disjoints.
c) Expliquer comment recomposer une solution \mathcal{C} à l'instance (X, \mathcal{F}) à partir des solutions \mathcal{C}_i aux instances (X_i, \mathcal{F}_i) .

Q. 32 On s'intéresse ici à l'algorithme qui consiste à décomposer une instance comme proposé en **Q. 31** et à résoudre chaque petite instance (X_i, \mathcal{F}_i) au moyen de l'algorithme par brute force naïf étudié précédemment. Comparer le nombre de cas que doit traiter l'algorithme par brute force appliqué directement sur l'instance initiale, au nombre de cas que doit traiter l'algorithme décrit ci-dessus.

6.2. Utilisation du graphe pour l'obtention d'un minorant

Lorsqu'on recherche la solution à un problème par raffinements successifs de solutions partielles, il est utile d'avoir un outil de mesure de la qualité de la solution partielle fabriquée jusqu'ici.

Supposons qu'on soit en train de considérer une solution partielle \mathcal{C}_1 qui ne peut être complétée que par des solutions utilisant au moins 10 ensembles alors qu'on a déjà croisé une solution complète utilisant 8 ensembles : on peut rejeter la solution partielle \mathcal{C}_1 sachant que la développer ne mènerait à aucune solution optimale. Aussi, afin d'évaluer la qualité d'une solution partielle, on se propose ici de mettre en place un algorithme calculant une minoration de la taille des solutions compatibles avec cette solution partielle.

Sommets non reliés. Dans le graphe $G = (S, A)$ associé à une solution partielle $(\mathcal{O}, \mathcal{N})$ d'une instance (X, \mathcal{F}) deux sommets x et y ne sont pas reliés (par une arête de A) si et seulement s'ils ne peuvent être couverts par un même ensemble de $\mathcal{F} \setminus \mathcal{N}$. Ainsi si deux tels sommets x et y existent, cette solution partielle ne peut être complétée qu'en ajoutant au moins deux ensembles de \mathcal{F} : l'un pour couvrir x , l'autre pour couvrir y .

Ensemble stable d'un graphe. Étant donné un graphe $G = (S, A)$, on dit d'un ensemble V de sommet du graphe qu'il est *stable* si les sommets de V sont deux à deux non reliés par une arête. V est donc stable si et seulement si pour tout couple (x, y) de sommets de V l'arête $\{x, y\}$ n'est pas dans A .

Q. 33 Expliquer en quoi la recherche d'un ensemble stable du graphe associé à une solution partielle fournit un minorant sur le cardinal d'une solution compatible avec cette solution partielle. Donner le minorant obtenu par la découverte d'un ensemble stable de cardinal $p \in \mathbb{N}$.

Afin de trouver un ensemble stable de sommets qui soit le plus grand possible (afin d'obtenir la meilleure minoration possible), on propose d'utiliser un algorithme glouton. L'algorithme construit un ensemble stable I en y ajoutant à chaque itération un nouveau sommet. Pour choisir un sommet à ajouter on maintient un ensemble de sommets candidats : un candidat est un sommet qu'on pourrait ajouter à I sans casser son caractère stable. Le choix glouton de l'algorithme est le suivant : à chaque étape on souhaite faire diminuer le moins possible l'ensemble des candidats, afin qu'il soit non vide le plus longtemps possible, pour construire un ensemble I le plus gros possible.

Q. 34 Définir une fonction `choix_glouton` prenant en arguments un graphe et un ensemble de sommets candidats et renvoyant un des sommets candidats qui suit le choix glouton décrit ci-dessus. On devra *mettre à jour* l'ensemble des candidats, au vu du choix glouton effectué.

```
choix_glouton(g: Graph, candidats: Set[int]) -> int
```

Q. 35 En déduire une fonction `stable` prenant en argument un graphe et renvoyant un ensemble stable en itérant, tant que cela est possible, le choix glouton ci-dessus.

```
stable(g: Graph) -> Set[int]
```

- Q. 36**
- Cet algorithme fournit-il un ensemble stable qui est de cardinal maximal (autrement dit, un ensemble stable tel qu'il n'existe aucun ensemble stable de cardinal strictement supérieur) ? On attend une justification succincte ou un contre-exemple.
 - Cet algorithme fournit-il un ensemble stable qui est maximal pour l'inclusion (autrement dit, un ensemble stable qui n'est contenu dans aucun ensemble stable autre que lui-même) ? On attend une justification succincte ou un contre-exemple.

Q. 37 Déduire des questions précédentes une fonction `minorant_couv_ens` prenant en argument une instance du problème (X, \mathcal{F}) et une solution partielle et renvoyant un minorant du cardinal des solutions du problème COUVENS, compatibles avec cette solution partielle.

```
minorant_couv_ens(n: int, f: Parties, sol_partielle: Partiel) -> int
```

7. Séparation et évaluation

Une stratégie d'exploration de l'espace des solutions pour l'instance (X, \mathcal{F}) est de décider successivement si on sélectionne ou non P_0, P_1, \dots . Une telle exploration forme un arbre : la racine correspond à la solution partielle (\emptyset, \emptyset) (aucune décision n'a été prise). Un nœud de profondeur i de cet arbre correspond à une solution partielle de la forme (\mathcal{O}, n) avec $\mathcal{O} \cup n = [\![0, i-1]\!]$, autrement dit on a décidé pour chaque j dans $[!\![0, i-1]\!]$ si on sélectionne ou non l'ensemble P_j .

En visitant tous les nœuds de cet arbre on réalisera en fait un algorithme par brute force. Le schéma algorithmique de séparation et évaluation permet de limiter le nombre de nœuds visités en maintenant, tout au long de l'exploration, la meilleure solution rencontrée jusque lors. Notons \mathcal{C}^b cette meilleure solution. Si une solution partielle (\mathcal{O}, n) ne peut être complétée qu'en des solutions de cardinal strictement supérieur à $|\mathcal{C}^b|$, il n'est pas intéressant d'explorer le sous-arbre issu de ce nœud (\mathcal{O}, n) . Il s'agit donc de trouver un minorant μ des valeurs des solutions complétant (\mathcal{O}, n) afin de tester si $\mu > |\mathcal{C}^b|$ et ce sans explorer lesdites solutions. Pour cela on utilisera les résultats de la section 6.2.

L'algorithme glouton de la section 4 permet de calculer une solution pour (X, \mathcal{F}) non nécessairement optimale, donnant ainsi une première valeur pour C^b . Afin d'améliorer cette valeur, on va calculer pour chaque noeud (\mathcal{O}, n) , une solution pour (X, \mathcal{F}) non nécessairement optimale complétant (\mathcal{O}, n) .

Q. 38 En s'inspirant de l'algorithme glouton de la section 4 (Algorithme 1, page 10), écrire le pseudo-code d'un algorithme fournissant une solution compatible avec une solution partielle donnée si cela est possible. Dans le cas contraire l'algorithme renverra None.

On suppose définie dans la suite une fonction `glouton_compatible` implémentant cet algorithme et dont la signature est la suivante.

```
glouton_compatible(n: int, f: Parties, partiel: Partiel) -> Optional[Tuple[int,  
→ List[bool]]]
```

Pour une instance (X, \mathcal{F}) et une solution partielle (\mathcal{O}, n) cette fonction renvoie `None` si (\mathcal{O}, n) ne peut être complétée en une solution. En pareil cas, il n'est pas intéressant d'explorer le sous-arbre issu du noeud (\mathcal{O}, n) . Dans le cas contraire, elle renvoie un couple (v, s) où s représente par sa fonction indicatrice une solution \mathcal{C} pour (X, \mathcal{F}) , non nécessairement optimale, et où v est le nombre d'ensembles utilisés par cette solution \mathcal{C} (i.e. le nombre de `True` dans le tableau s).

Q. 39 Définir une fonction `separation_et_evaluation` implémentant l'algorithme décrit ci-avant.

```
separation_et_evaluation(n: int, f: Parties) -> Tuple[int, List[bool]]
```

∴ FIN DU SUJET ∴

CAPES externe et CAFEP-CAPES

EBE NSI 2

Section Numérique et Sciences Informatiques

Epreuve Disciplinaire appliquée – Session 2025

Préambule : cette épreuve est constituée de deux parties A et B indépendantes. Les réponses aux questions doivent être précises et rédigées avec soin. Certaines questions demandent des productions d'enseignement à destination des élèves (cours, exercices, projets, activités sur machines, activités débranchées, etc.). Ces questions devront être traitées avec la plus grande attention.

Partie A : la photographie numérique

Le but de cette partie est d'étudier le traitement de la photographie numérique à travers le continuum des programmes de SNT, première NSI et terminale NSI.

Elle est composée de cinq parties indépendantes qui couvrent plusieurs notions des référentiels.

Partie 1 : L'éthique et la photographie numérique

- Pour traiter les aspects éthiques à considérer lors de la prise et de la publication de photographies numériques et notamment pour les sensibiliser au harcèlement, vous décidez d'organiser une discussion avec la classe de SNT mais souhaitez leur faire faire des recherches sur la protection de la vie privée et la diffusion d'images sensibles.

Proposez un questionnaire (de cinq ou six questions) auquel devront répondre vos élèves en effectuant des recherches sur le Web autour des notions de vie privée, de publication, de photographie numérique, de droit à l'image...

- Expliquez en quelques lignes en quoi la photographie numérique soulève des enjeux éthiques, et pourquoi il est important de sensibiliser les élèves à ces enjeux.

Partie 2 : Notions juridiques et la cyberviolence

Nous nous intéressons ici à deux parties du cours de SNT : le Web et Les réseaux sociaux dont voici les deux extraits :

Contenus	Capacités attendues
Notions juridiques	Connaître certaines notions juridiques (licence, droit d'auteur, droit d'usage, valeur d'un bien).

Contenus	Capacités attendues
Cyberviolence	Connaître les dispositions de l'article 222-33-2-2 du code pénal. Connaître les différentes formes de cyberviolence (harcèlement, discrimination, sexting...) et les ressources disponibles pour lutter contre la cyberviolence

3. Vous voudriez faire une activité sur les implications du droit d'auteur dans le domaine de la photographie numérique. Après avoir fait faire une recherche par petits groupes sur un cas réel de violation du droit d'auteur et les conséquences juridiques, vous aimerez conclure l'activité par une synthèse sur ces deux points. Que proposeriez-vous ? L'article L335-2 sur les droits d'auteur se trouve en annexe.
4. Vous organisez un débat sur la cyberviolence et le cyberharcèlement. Rédigez six questions pour orienter par une recherche la préparation du débat autour des capacités attendues du référentiel. L'article 222-33-2-2 du code pénal se trouve en annexe.

Partie 3 : La représentation matricielle d'une image et son traitement en itératif

Vous souhaitez faire travailler les élèves de SNT sur deux parties du référentiel : Notions transversales de programmation et La photographie numérique dont voici les deux extraits :

Contenus	Capacités attendues
Affectations, variables, Séquences, Instructions conditionnelles, Boucles bornées et non bornées, Définitions et appels de fonctions	Écrire et développer des programmes pour répondre à des problèmes et modéliser des phénomènes physiques, économiques et sociaux.

Contenus	Capacités attendues
Traitement d'image	Traiter par programme une image pour la transformer en agissant sur les trois composantes de ses pixels.

Et vous souhaitez faire travailler les élèves de première NSI sur les types construits et notamment sur cet extrait de programme :

Contenus	Capacités attendues
Tableau indexé, tableau donné en compréhension	Utiliser des tableaux de tableaux pour représenter des matrices : notation a [i] [j].

5. Comment peut-on représenter une image en niveaux de gris à l'aide de matrices en Python ? Quelle serait la notation pour accéder à un pixel spécifique de l'image ?
6. Comment, dans la représentation matricielle classique d'une image, le codage d'une image en couleur diffère-t-il de celui d'une image en niveaux de gris ?
7. Vous souhaitez introduire le cours sur le traitement d'images en seconde SNT. Proposez une activité débranchée pour leur faire découvrir la notion de négatif et la notion de niveau de gris pour une image. On pourra s'aider si besoin d'un quadrillage représentant une image.
8. Créez une fiche méthode pour l'utilisation du module Image de la bibliothèque PIL (pillow) à destination de la classe de seconde SNT. Un extrait de la bibliothèque pillow est fourni en annexe.
9. On vous propose un code dont vous vous inspirez pour le proposer à la classe de seconde SNT.

Proposez des questions pour guider vos élèves dans leur travail pour compléter les trous de ce code.

Puis détaillez une correction de vos questions, ainsi que le code correctement complété et commenté, tels que vous les donneriez à des élèves de seconde.

Code d'obtention d'un négatif :

```
from PIL import Image

# Ouvre l'image
img = Image.open(____)

# Parcours tous les pixels de l'image
for y in range(img.height):
    for x in range(img.width):
        # Obtenir la couleur du pixel (R, G, B)
        r, g, b = img.getpixel((x, y))

        # Inverser les valeurs des couleurs pour obtenir un négatif
        r = _____
        g = _____
        b = _____

        # Modifier le pixel avec les nouvelles valeurs
        img.putpixel((x, y), (r, g, b))

# Sauvegarder l'image négative
img.save(____)
```

10. Préparez les différentes étapes d'un cours aboutissant au programme précédent : vous détaillerez l'introduction à ce travail, les objectifs et le nombre de séances, ainsi que le travail sur la notion de double boucle.
11. Proposez un code pour obtenir une image en niveau de gris à partir d'une image. L'image de départ sera en RGB et sera notée : "image.jpg".
12. Afin d'introduire les tableaux de tableaux en première NSI, vous souhaitez mettre en place une auto-évaluation sous forme de QCM. Proposez 3 questions / réponses qui vous indiqueront la compréhension des élèves sur cette notion.
13. Un élève de première NSI vous rend une partie de son travail sur les tableaux de tableaux. Il vous indique que son script a un comportement qu'il juge étrange, car en modifiant un tableau de pixels (image2), l'autre tableau (image) est également modifié. Expliquez-lui la raison de ce comportement et proposez une correction.

```
image = [ [255,0,0],[0,255,0], [0,0,255],[255,255,255] ]
image2 = image

for x in range(len(image2)):
    for y in range(len(image2[x])):
        if image2[x][y] == 255:
            image2[x][y] = 125
```

14. Vous décidez, ensuite, de produire une petite activité sur les carrés magiques. Vous voulez vous inspirer d'un exercice du site France-IOI qui vous propose :

Entrée

La première ligne de l'entrée contient un entier N : le nombre de cases du côté de la grille de nombres.

Chacune des N lignes suivantes contient N entiers séparés par des espaces : les nombres d'une ligne de la grille.

Sortie

Vous devez afficher une ligne sur la sortie, contenant le mot "yes" si le carré fourni est un carré magique, et "no" sinon.

Définition d'un carré magique

Un **carré magique** est une grille ($N \times N$) dans laquelle :

- Tous les nombres sont uniques de 1 à N .
- La somme des nombres de chaque ligne, de chaque colonne et des deux diagonales est identique.

La somme commune à toutes ces lignes, colonnes et diagonales est appelée **constante magique**. Pour un carré magique de taille ($N \times N$), cette constante est égale à : $S = \frac{N(N^2+1)}{2}$

Exemple*entrée :*

```

3
6 1 8
7 5 3
2 9 4

```

Sortie :

yes

Décrivez le déroulement de l'activité qui permettra aux élèves de se familiariser avec les tableaux de tableaux. Pour cela, comment modéliser l'entrée ? Donner les différentes fonctions à créer pour décomposer le problème.

15. Vous aimerez amener les élèves à créer un damier noir et blanc, de dimension 600 x 400, et dont chaque carré à un côté de longueur 20 pixels, à l'aide d'une fonction dont le début du code est le suivant :

```

from PIL import Image
def damier():
    damier = Image.new('RGB', (600,400))
    for x in range(600):
        for y in range(400):
            ...

```

Expliquez comment amener les élèves à calculer le numéro de colonne et de ligne et à déterminer la couleur du pixel dans la fonction damier ci-dessus.

Partie 4 : La rotation des images

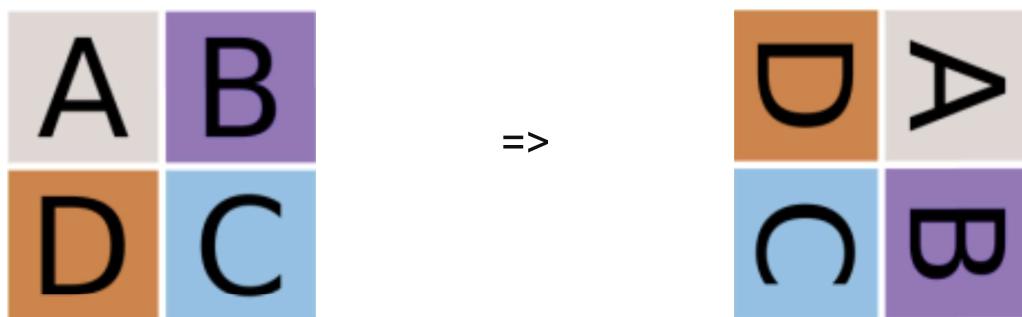
Nous nous intéresserons dans cette partie au programme de terminale et en particulier :

Contenus	Capacités attendues
Méthode « diviser pour régner ».	Écrire un algorithme utilisant la méthode « diviser pour régner ».

et

Contenus	Capacités attendues
Récursivité.	Écrire un programme récursif. Analyser le fonctionnement d'un programme récursif.

16. Expliquez la méthode “diviser pour régner” dans le cadre de la rotation d’un quart de tour d’une image et en quoi cet algorithme a un coût en mémoire constant. On pourra utiliser l’image carrée suivante (celle de gauche) qui deviendrait celle de droite (après rotation d’un quart de tour) :



17. Vous souhaitez mettre en œuvre, sous forme de mini-projet, la rotation d’une image d’un quart de tour avec une “méthode diviser pour régner”. Programmez plusieurs fonctions pour implémenter cet algorithme en récursif. On se limitera au cas où l’image est de taille NxN où N est une puissance de 2.

On pourra s’aider des fonctions suivantes (les arguments des fonctions sont décrits sous l’encadré) :

```
from PIL import Image

# Cette fonction échange deux pixels dans l'image source.
def echangePixel(source, x1, y1, x2, y2):
    pass

# Cette fonction échange deux quadrants dans l'image source.
def echangeQuadrant(source, x1, y1, x2, y2, n):
    pass

# Cette fonction effectue une rotation d'un quart de tour d'un quadrant de l'image source.
def tourneQuadrant(source, x, y, n):
    pass

# Cette fonction effectue une rotation d'un quart de tour de l'image source.
def rotationDpR(source):
    largeur, hauteur = source.size # dimensions de l'image
    tourneQuadrant(source, 0, 0, largeur) # Effectue une rotation sur l'ensemble de l'image
```

- avec
- `echangePixel(source, x1, y1, x2, y2)`
 - `source` : C'est l'image d'origine que nous manipulons pour échanger les pixels.

- x1 : Coordonnée en x du premier pixel à échanger (colonne). Cela représente la position horizontale (axe x) du premier pixel.
- y1 : Coordonnée en y du premier pixel à échanger (ligne). Cela représente la position verticale (axe y) du premier pixel.
- x2 : Coordonnée en x du deuxième pixel à échanger (colonne). Cela représente la position horizontale (axe x) du deuxième pixel.
- y2 : Coordonnée en y du deuxième pixel à échanger (ligne). Cela représente la position verticale (axe y) du deuxième pixel.
- echangeQuadrant(source, x1, y1, x2, y2, n)
 - source : L'image source dans laquelle les quadrants vont être échangés.
 - x1 : Coordonnée en x du coin supérieur gauche du premier quadrant à échanger. C'est la position horizontale du coin supérieur gauche du premier quadrant à échanger.
 - y1 : Coordonnée en y du coin supérieur gauche du premier quadrant à échanger. C'est la position verticale du coin supérieur gauche du premier quadrant.
 - x2 : Coordonnée en x du coin supérieur gauche du deuxième quadrant à échanger. Position horizontale du coin supérieur gauche du deuxième quadrant à échanger.
 - y2 : Coordonnée en y du coin supérieur gauche du deuxième quadrant à échanger. Position verticale du coin supérieur gauche du deuxième quadrant.
 - n : Taille du côté des quadrants échangés (le quadrant est de taille n x n). La taille de chaque quadrant à échanger (chaque quadrant est un carré de côté n).
- tourneQuadrant(source, x, y, n)
 - source : L'image source dans laquelle nous effectuons la rotation d'un quadrant.
 - x : Coordonnée en x du coin supérieur gauche du quadrant à faire tourner. Position horizontale du coin supérieur gauche du quadrant à faire tourner.
 - y : Coordonnée en y du coin supérieur gauche du quadrant à faire tourner. Position verticale du coin supérieur gauche du quadrant à faire tourner.
 - n : Taille du côté du quadrant à faire tourner (le quadrant est de taille n x n). La taille du quadrant à faire tourner, cette fonction va travailler sur des sous-parties de l'image.
- rotationDpR(source)
 - source : L'image complète sur laquelle la rotation d'un quart de tour va être appliquée.

18. Proposez un code plus simple, non récursif et de même complexité spatiale

Partie 5: Les filtres d'image en Programmation Orientée Objet (POO)

Nous nous intéresserons dans cette partie au programme de terminale et en particulier :

Contenus	Capacités attendues
Vocabulaire de la programmation objet : classes, attributs, méthodes, objets.	Écrire la définition d'une classe. Accéder aux attributs et méthodes d'une classe.

On cherche à faire implémenter quelques filtres dans le cadre du programme de terminale.

19. Proposez une activité d'introduction à la POO pour permettre aux élèves à s'approprier ces notions : classes, attributs, méthodes, objets.
20. Vous remarquez que les élèves confondent les classes et les instances (objet). Comment pourriez-vous les aider à préciser ce vocabulaire ?
21. Comment leur expliquer le mot 'self' qui apparaît en POO ?

Vous décidez de proposer aux élèves une activité qui lie la POO et les filtres des images dont le diagramme de classe est le suivant :

Classe Filtre: <ul style="list-style-type: none"> • constructeur : <ul style="list-style-type: none"> ◦ fichier : string ◦ img : Image ◦ pix : list • Méthodes : <ul style="list-style-type: none"> ◦ __init__(fichier: string) ◦ taille() : tuple ◦ largeur() : int ◦ hauteur() : int ◦ nombre_pixels() : int ◦ get_pixel(col: int, ligne: int) : tuple

- img permet l'ouverture de l'image
- pix permet d'accéder à un pixel de l'image que l'on a ouvert avec img
- taille() retourne la taille en pixels d'une image sous forme de tuple largeur, hauteur
- largeur() retourne la largeur d'une image en pixels
- hauteur() retourne la hauteur d'une image en pixels
- nombre_pixels() retourne le nombre total de pixels d'une image
- get_pixel (col, ligne) retourne la valeur du pixel de coordonnées (col, ligne), ou None si les coordonnées sont incorrectes.

22. Vous voulez proposer des aides supplémentaires et/ou du code pour accompagner les élèves dans cette activité, afin de les guider de manière progressive et claire.

Proposez des supports qui permettent aux élèves de :

- Comprendre précisément les objectifs et attendus : quelles méthodes doivent être complétées.
Comment utiliser chaque méthode (exemples attendus).
 - S'assurer de la bonne compréhension des méthodes et attributs : compléter une structure de code avec des trous en respectant les étapes données. Utiliser des questions progressives qui guideront les élèves sur l'utilisation des attributs et méthodes de la classe.
23. Le but, à présent, est de faire faire une méthode couleur_vers_gris(fichier: string) qui permet de transformer la photo couleur en nuance de gris. Comment amener les élèves à faire une double boucle dans le codage de cette méthode ?
24. Proposez un algorithme en pseudo-code qui permette à l'élève d'implémenter la méthode : couleur_vers_gris(fichier: string) et décrivez ce qu'il fait.
25. Proposez une implémentation de l'algorithme précédent.

Partie B : optimisation par la programmation dynamique

Vous souhaitez aborder la notion de programmation dynamique avec comme objectifs :

- Comprendre le concept de programmation dynamique.
- Reconnaître les problèmes pouvant être résolus efficacement avec la programmation dynamique.
- Savoir appliquer la méthode de résolution de problèmes par programmation dynamique.

Vous avez décidé d'accompagner vos élèves de terminale à travers une activité concrète. Étant donné que votre classe présente une hétérogénéité de niveaux, cette activité sera abordée en plusieurs séances. Cela vous permettra d'adapter les séances en fonction des difficultés rencontrées par chaque élève.

La propriétaire d'un cheval se penche sur le calendrier des courses hippiques de la saison à venir :

- chaque jour il y a une course hippique à laquelle il peut participer ;
- chacune des courses hippiques offre un prix pour le vainqueur ;
- après avoir participé à une course, le cheval doit se reposer et ne peut participer à une nouvelle course que quatre jours plus tard.

La propriétaire cherche à choisir les courses auxquelles il participera en maximisant

le montant total des prix qu'il pourra gagner s'il est vainqueur.

On suppose que le calendrier des courses est implémenté par un tableau indexé ‘calendrier’ pour lequel la valeur d’indice ‘*i*’ est le prix offert pour la course du ‘*i*ème jour de la saison (en commençant à zéro).

Le planning des courses choisies est implémenté par un tableau indexé ‘planning’ pour lequel la valeur d’indice ‘*i*’ est le jour choisi de la saison (en commençant à zéro). Cette valeur doit correspondre à un jour référencé dans le calendrier.

Exemple :

calendrier = [2, 1, 10, 6, 8, 20, 22, 10, 4, 20, 2, 4, 14, 2, 6] est de taille 15.

La propriétaire peut choisir comme planning [0, 4, 8, 12] auquel cas le montant total des prix est $2 + 8 + 4 + 14 = 28$.

Partie 1 : Planning

26. Afin de faire comprendre la contrainte concernant uniquement le repos du cheval dans le planning choisi, proposez quatre exemples pertinents que les élèves doivent résoudre à la main.

Vous proposez aux élèves d’écrire une fonction ‘planning_correct’ qui admet comme argument ‘planning’ une liste représentant un planning et qui renvoie ‘True’ si le planning est correct sinon ‘False’. Dans un premier temps et afin de simplifier l’écriture de cette fonction, vous proposez de ne considérer que les jours de repos comme contrainte.

27. Écrivez cette fonction.

28. Vous trouverez ci-dessous des réponses données par des élèves.

Elève 1

```
def planning_correct(planning):
    for i in range(len(planning)-1):
        p = planning[i+1] - planning[i]
        return p > 4

planning_correct([0, 4, 8, 12])
False
```

Elève 2

```
def planning_correct(planning):
    for i in planning:
        if i+4 >= i:
            return False
        else:
            return True
```

Quelles sont les erreurs commises par chaque élève ? Quels commentaires proposez-vous sur la copie afin de lui suggérer une piste de réflexion ?

29. Complétez la fonction ‘planning_correct’ de la question 27 qui admet comme arguments ‘planning’ une liste représentant un planning et ‘calendrier’ une liste représentant un calendrier afin de tenir compte de toutes les contraintes et qui renvoie ‘True’ si le planning est correct sinon ‘False’.
30. Proposez quatre plannings avec le calendrier = [2, 1, 10, 6, 8, 20, 22, 10, 4, 20, 2, 4, 14, 2, 6] afin que les élèves vérifient leur fonction.
31. Au regard des erreurs des élèves et dans le cadre d'une remédiation, quelles sont les trois notions importantes que vous pourriez aborder avec les élèves ?
32. Proposez quatre exercices courts afin de vérifier les acquis des élèves.

Partie 2 : Gain possible

La partie 1 est corrigée. La fonction ‘planning_correct’ est donnée aux élèves.

33. Vous proposez aux élèves d'écrire une fonction ‘gain_possible’ qui admet comme arguments ‘calendrier’ une liste représentant un calendrier et ‘planning’ une liste représentant un planning et qui renvoie le gain possible dans le cas où le planning est correct sinon -1.

Vous trouverez ci-dessous des réponses données par des élèves.

Elève 1

```
def gain_possible(calendrier, planning):
    assert isinstance(calendrier, list) and isinstance(planning, list)
    if planning_correct == True :
        for g in range(calendrier):
            for i in range(planning):
                return calendrier[planning[i]]
    else :
        return -1
```

Elève 2

```
def gain_possible(calendrier, planning):
    gain = 0
    if planning_correct(planning) == True:
        for elem in calendrier:
            gain += elem
        return gain
    return -1
```

Elève 3

```
def gain_possible(calendrier, planning):
    for i in planning:
        for elem in calendrier:
            if
                gain = gain + elem
            else:
```

Indiquez les erreurs commises par chaque élève et proposez une correction en tenant compte ces erreurs, en justifiant vos choix.

Partie 3 : Résolution par un algorithme glouton

Nous nous intéresserons dans cette partie à un algorithme vu en classe de première

Contenus	Capacités attendues
Algorithmes gloutons	Résoudre un problème grâce à un algorithme glouton

Vous décidez de reprendre un exemple classique de rendu de monnaie vu en première afin de préparer les élèves à une solution gloutonne de l'activité.

On rappelle que le problème du rendu de monnaie consiste à trouver le nombre minimal de pièces ou de billets nécessaires pour payer une somme donnée.

On veut programmer une caisse automatique pour qu'elle rende la monnaie de façon optimale, c'est-à-dire avec le nombre minimal de pièces et de billets.

Les valeurs des pièces et billets à disposition sont : 1, 2, 5, 10, 20, 50, 100 et 200 euros. On suppose que la machine dispose d'autant d'exemplaires de pièces et billets que nécessaires pour rendre la monnaie.

34. Qu'est-ce qu'un algorithme glouton ?
35. Détaillez l'algorithme dans le cas du rendu de monnaie.
36. Un élève vous demande si la solution proposée est optimale pour tous les systèmes monétaires. Donnez un contre-exemple.
37. Comment expliquez-vous à vos élèves la différence fondamentale entre l'approche de l'algorithme glouton et l'approche par force brute pour résoudre le problème du rendu de monnaie ?
38. Détaillez l'algorithme glouton dans le cas du calendrier des courses hippiques.
39. Pour résoudre ce problème, un élève propose d'écrire une fonction qui mettra à jour le calendrier en mettant à 0 la valeur de l'indice 'jour' choisie ainsi que les 3 valeurs des indices avant et après si possible. Et pour cela, il propose cette fonction :

```
def supprime_jours(calendrier, jour):
    # Test validation planning
    assert 0 <= jour < len(calendrier), "jour non valide"
    for i in range(0,4):
        if jour + i < len(calendrier) and jour - i >= 0:
            calendrier[jour + i] = 0
            calendrier[jour - i] = 0
    return calendrier
```

Quelle est l'erreur de l'élève ? Proposez un exemple de calendrier et jour pour lui faire comprendre son erreur en le justifiant.

40. Vous décidez de proposer une évaluation sous la forme d'une épreuve pratique de terminale (voir annexe).

L'exercice 1 portera sur la notion de listes.

L'exercice 2 s'appuiera sur le programme suivant.

```
def max_gain_courses(prix_courses):
    total_gain = 0
    while sum(prix_courses) != 0:
        meilleur_jour = prix_courses.index(max(prix_courses))
        total_gain += prix_courses[meilleur_jour]
        prix_courses = supprime_jours(prix_courses, meilleur_jour)
    return total_gain
```

Proposez deux exercices en tenant compte des attendus. Vous devrez justifier brièvement vos choix.

Partie 4 : Récursivité

On donne la fonction récursive total_prix qui admet comme arguments ‘calendrier’ une liste représentant un calendrier et jour un entier représentant l’indice de la dernière course possible et qui renvoie le gain maximum.

Ainsi total_prix(calendrier, 9) calcule le gain maximum possible lorsqu’on choisit les 10 premières dates du calendrier.

```
def total_prix(calendrier, jour):
    if jour < 4:
        prix_total_max = max(calendrier[i] for i in range(jour + 1))
        return prix_total_max
    else:
        prix_total_max = max(total_prix(calendrier, jour - 1),
                             total_prix(calendrier, jour - 4) + calendrier[jour])
    return prix_total_max
```

41. Intéressons-nous au test d’arrêt :

```
if jour < 4:
    prix_total_max = max(calendrier[i] for i in range(jour + 1))
    return prix_total_max
```

Pourriez-vous expliquer le choix de cette condition `jour < 4` et le calcul de la variable `prix_total_max` ?

Pourquoi cette condition est-elle nécessaire et comment influence-t-elle le comportement de la fonction ?

42. Dessinez soigneusement l’arbre des appels récursifs lorsque l’appel principal est `total_prix(calendrier, 9)`. Combien d’appels sont effectués en tout (appel principal compris) ?

43. Comment peut-on modifier la fonction `total_prix` afin de réduire le nombre d’appels ?

Partie 5 : Programmation dynamique

Nous nous intéresserons dans cette partie à la programmation dynamique de la classe de terminale :

Contenus	Capacités attendues
Programmation dynamique	Utiliser la programmation dynamique pour écrire un algorithme.

Il existe deux facettes équivalentes de la programmation dynamique :

- La programmation descendante récursive avec mémoisation appelée top-down ;
- La programmation ascendante itérative, appelée bottom up.

44. Comment expliquez-vous aux élèves, en quelques phrases, ces deux approches. Vous illustrerez vos explications par des exemples.

45. Vous proposez aux élèves la fonction total répondant au paradigme de programmation dynamique ci-dessous et l'affichage des différents calculs.

```
cal = [2, 25, 30, 2, 15, 2, 25, 6, 15, 25, 6, 2, 6, 6, 6, 2, 15, 6, 6, 6, 15, 15, 25, 25, 6, 15]

def maxi_debut(cal):
    maxi = cal[0]
    m =[maxi]
    for i in range(1,4):
        maxi = max(maxi,cal[i])
        m.append(maxi)
    return m

def total(cal,n):
    t = maxi_debut(cal)
    for i in range(4,n + 1):
        t.append (max( t[i - 1] , cal[i] + t[i - 4]))
        print(t)
    return t

total(cal,25)
```

```
[2, 25, 30, 30, 30]
[2, 25, 30, 30, 30, 30]
[2, 25, 30, 30, 30, 30, 55]
[2, 25, 30, 30, 30, 30, 55]
[2, 25, 30, 30, 30, 30, 55, 55]
[2, 25, 30, 30, 30, 30, 55, 55, 55]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 61]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 61, 67]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 61, 67, 67]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 61, 67, 67, 76]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76, 76]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76, 76, 91]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76, 76, 91, 91]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76, 76, 91, 91, 101]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76, 76, 91, 91, 101, 101]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76, 76, 91, 91, 101, 101, 101]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76, 76, 91, 91, 101, 101, 101, 106]
[2, 25, 30, 30, 30, 30, 55, 55, 55, 55, 61, 61, 61, 61, 61, 67, 67, 76, 76, 76, 91, 91, 101, 101, 101, 106]
```

En quels termes expliquez-vous aux élèves, les valeurs obtenues à la première ligne de l'affichage ?

Choisissez des lignes pertinentes consécutives de l'affichage pour expliquer les principes de la programmation dynamique.

Les trois parties de ce sujet couvrent, sous des angles disjoints, des problématiques au cœur de la réalisation d'un éditeur de texte.

Dépendances. Bien qu'elles partagent un thème commun, les trois exercices sont indépendants et doivent être traités tous les trois. On veillera à bien indiquer sur la copie les changements de partie. On veillera également à bien se conformer aux attendus de chaque exercice en terme de précondition de fonction et de justification de complexité.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Gestion de fichiers dans un éditeur de texte

Préliminaires

Dans le cadre de cet exercice, on s'intéresse à la consultation et la modification par un éditeur d'un très gros fichier texte—i.e., de plusieurs giga-octets—stocké sur le disque local d'un ordinateur. L'objectif de cet exercice est de proposer les méthodes d'accès et les structures de données qui permettent de manipuler le contenu de ce fichier de manière aussi efficace que possible. Le langage de programmation considéré pour l'ensemble de cet exercice est le langage C. Il est attendu que toute proposition de code source soit clairement et précisément documentée.

Partie I. Lecture d'un fichier

```

1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int editer_document(void* contenu) {
6     /* Code non détaillé */
7 }
8
9 int charger_document(int fd) {
10    /* Code non détaillé */
11 }
12
13 int main (int argc, char *argv[]) {
14     int fd = open(argv[0], O_RDONLY);
15     char* contenu;
16     if (charger_document(fd, &contenu) != EXIT_FAILURE) {
17         editer_document(contenu);
18     }
19     close(fd);

```

```
20 }     return 0;  
21 }
```

Question 1.1. Le code source sur la page précédente est erroné et devrait permettre d'ouvrir un fichier texte en lecture/écriture. Décrire 4 erreurs de programmation contenues dans ce code source en expliquant les problèmes, sans nécessairement les corriger, que peuvent engendrer ces erreurs. L'annexe A vous rappelle la syntaxe de la fonction `open()`.

Question 1.2. Proposer une correction de ce code source erroné pour permettre de lire un document en lecture/écriture comme attendu.

Question 1.3. On se propose d'utiliser la primitive `mmap` pour manipuler le contenu du fichier édité, décrite en annexe C. Dans quelle situation l'usage de `mmap` est-il préconisé ?

Question 1.4. Écrire le code source de la méthode `charger_document` pour charger le contenu du document en utilisant la primitive `mmap`. Vous pouvez vous aider des annexes B et C.

Partie II. Manipulation du contenu

Nous considérons désormais que le fichier a pu être chargé en mémoire par l'éditeur de texte et que l'utilisateur souhaite disposer de plusieurs primitives pour lui permettre d'opérer des modifications sur le document manipulé.

Pour l'ensemble des fonctions de cette partie, il est convenu que toute modification valide opérée sur le document doit donner lieu à un enregistrement automatique du document modifié sur le disque.

Question 1.5. Proposer le code d'une fonction `remplacer_texte(char* addr, char* texte, char* remplacement)` qui cherche toutes les occurrences d'un texte `texte` et les remplace par le texte `remplacement`.

La fonction doit renvoyer une erreur si le texte de remplacement est plus long que le texte remplacé, ou le nombre d'occurrences du texte remplacées sinon. Si le texte de remplacement est plus court que le texte remplacé, la fonction doit compléter les caractères manquants par des caractères espaces.

Question 1.6. Expliquer pourquoi il est avisé, ou non, de mettre en œuvre une fonction `chercher_texte(char* addr, char* texte)` comme une simple invocation de la fonction `remplacer_texte(addr, texte, texte)`.

Question 1.7. Proposer le code source d'une fonction `insérer_texte(char* addr, char* texte)` qui insère le texte `texte` à partir de la position déterminée par `addr`.

Partie III. Gestion des modifications

Enfin, nous souhaitons ajouter à l'éditeur de texte la capacité de mémoriser et d'annuler un historique de modifications opérées par l'utilisateur sur le texte d'un document chargé.

Question 1.8. Proposer le code source d'une structure de donnée `modification` pour tracer les modifications de type remplacement et insertion d'un document édité.

Question 1.9. Proposer le code source d'une structure de données `historique`, et les fonctions adéquates, pour mémoriser un ensemble de modifications et permettre d'annuler ces modifications dans le bon ordre.

Question 1.10. Modifier la signature et le code source de la fonction `remplacer_texte(...)` pour mémoriser toutes les modifications apportées au texte lors de l'appel à cette fonction.

Question 1.11. Proposer la signature et le code source d'une fonction `annuler_modifications(...)` pour annuler les `n` dernières modifications apportées à un document en cours d'édition.

A OPEN

A.1 NOM

`open` - Ouvrir ou créer éventuellement un fichier ou un périphérique

A.2 SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
```

A.3 DESCRIPTION

Étant donné le chemin *pathname* d'un fichier, `open()` renvoie un descripteur de fichier, un petit entier positif ou nul utilisable par des appels système ultérieurs (`read()`, `write()`, `lseek()`, `fcntl()`, etc.). Le descripteur de fichier renvoyé par un appel réussi sera le plus petit descripteur de fichier non encore ouvert pour le processus.

Un appel à `open()` crée une nouvelle *description de fichier ouvert*, une entrée dans la table des fichiers ouverts du système. Cette entrée enregistre la position dans le fichier et les attributs

d'état du fichier Un descripteur de fichier est une référence à l'une de ces entrées ; cette référence n'est pas affectée si *pathname* est ultérieurement supprimé ou modifié pour se référer à un fichier différent. La nouvelle description de fichier ouvert n'est initialement pas partagée avec un autre processus, mais ce partage peut survenir après un **fork()**.

Le paramètre *flags* doit inclure l'un des *mode d'accès* suivants : **O_RDONLY**, **O_WRONLY** ou **O_RDWR**. Ceux-ci réclament respectivement l'ouverture du fichier en lecture seule, écriture seule, ou lecture-écriture.

A.4 VALEUR RENVOYÉE

open() renvoie le nouveau descripteur de fichier s'il réussit, ou -1 s'il échoue, auquel cas *errno* contient le code d'erreur.

B FSTAT

B.1 NOM

fstat - Obtenir l'état d'un fichier (*file status*)

B.2 SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int fstat(int fd, struct stat* buf);
```

B.3 DESCRIPTION

Cette fonction renvoie des informations à propos d'un fichier. **fstat()** récupère l'état du fichier pointé par le descripteur *fd*, obtenu avec **open()** et remplit le tampon *buf*. Cette fonction renvoie une structure *stat* contenant les champs suivants :

```
struct stat {
    dev_t      st_dev;      /* ID du périphérique contenant le fichier */
    ino_t      st_ino;      /* Numéro inœud */
    mode_t     st_mode;     /* Protection */
    nlink_t    st_nlink;    /* Nb liens matériels */
    uid_t      st_uid;      /* UID propriétaire */
```

```

    gid_t      st_gid;      /* GID propriétaire */
    dev_t      st_rdev;     /* ID périphérique (si fichier spécial) */
    off_t      st_size;     /* Taille totale en octets */
    blksize_t  st_blksize;  /* Taille de bloc pour E/S */
    blkcnt_t   st_blocks;   /* Nombre de blocs alloués */
};


```

Le champ *st_dev* indique le périphérique sur lequel réside le fichier. Le champ *st_rdev* indique le périphérique que ce fichier représente. Le champ *st_size* indique la taille du fichier (s'il s'agit d'un fichier régulier ou d'un lien symbolique) en octets. La taille d'un lien symbolique est la longueur de la chaîne représentant le chemin d'accès qu'il vise, sans l'octet nul final.

Le champ *st_blocks* indique le nombre de blocs de 512 octets alloués au fichier (cette valeur peut être inférieure à *st_size*/512 si le fichier contient des trous). Le champ *st_blksize* indique la taille de bloc « préférée » pour les entrées-sorties du système de fichiers (l'écriture dans un fichier par petits morceaux peut induire de nombreuses étapes lecture-modification-écriture peu efficaces).

B.4 VALEUR RENVOYÉE

Cet appel système renvoie zéro s'il réussit. En cas d'échec, -1 est renvoyé, et *errno* contient le code de l'erreur.

C MMAP

C.1 NOM

mmap, **munmap** - Établir/supprimer une projection en mémoire (map/unmap) des fichiers ou des périphériques.

C.2 SYNOPSIS

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

C.3 DESCRIPTION

mmap() crée une nouvelle projection dans l'espace d'adressage virtuel du processus appelant. L'adresse de départ de la nouvelle projection est indiquée dans *addr*. L'argument *length* indique la longueur de la projection.

Si *addr* est NULL, le noyau choisit l'adresse à laquelle créer la projection ; c'est la méthode la plus portable pour créer une nouvelle projection. Si *addr* n'est pas NULL, le noyau le considère comme un conseil l'endroit où placer la projection ; sous Linux, la projection sera créée à la prochaine plus haute frontière de page. L'adresse de la nouvelle projection est renvoyée comme résultat de l'appel.

Le contenu d'une projection de fichier est initialisé avec *length* octets à partir de la position *offset* dans le fichier (ou autre objet) référencé par le descripteur de fichier *fd*. *offset* doit être un multiple de la taille de page telle qu'elle est renvoyée par *sysconf(_SC_PAGE_SIZE)*.

L'argument *prot* indique la protection mémoire que l'on désire pour cette projection, et ne doit pas entrer en conflit avec le mode d'ouverture du fichier. Il s'agit soit de **PROT_NONE** (le contenu de la mémoire est inaccessible) soit d'un OU binaire entre les constantes suivantes :

PROT_EXEC On peut exécuter du code dans la zone mémoire.

PROT_READ On peut lire le contenu de la zone mémoire.

PROT_WRITE On peut écrire dans la zone mémoire.

PROT_NONE Les pages ne peuvent pas être accédées.

Le paramètre *flags* détermine si les modifications de la projection sont visibles par les autres processus projetant la même région et si les modifications sont appliquées au fichier sous-jacent. Ce comportement est déterminé en incluant exactement une des valeurs suivantes dans *flags* :

MAP_SHARED Partager cette projection. Les modifications de la projection sont visibles par les autres processus qui projettent ce fichier, et sont appliquées au fichier sous-jacent. En revanche, ce dernier n'est pas nécessairement mis à jour tant qu'on n'a pas appelé **msync()** ou **munmap()**.

MAP_PRIVATE Créer une projection privée, utilisant la méthode de copie à l'écriture. Les modifications de la projection ne sont pas visibles par les autres processus projetant le même fichier et ne sont pas appliquées au fichier sous-jacent. Il n'est pas précisé si les changements effectués dans le fichier après l'appel **mmap()** seront visibles.

La projection doit avoir une taille multiple de celle des pages. Pour un fichier dont la longueur n'est pas un multiple de la taille de page, la mémoire restante est remplie de zéros lors de la projection, et les écritures dans cette zone n'affectent pas le fichier. Les effets de la modification de la taille du fichier sous-jacent sur les pages correspondant aux zones ajoutées ou supprimées ne sont pas précisés.

C.3.1 munmap()

L'appel système **munmap()** détruit la projection dans la zone de mémoire spécifiée, et s'arrange pour que toute référence ultérieure à cette zone mémoire déclenche une erreur d'adressage. La projection est aussi automatiquement détruite lorsque le processus se termine. À l'inverse, la fermeture du descripteur de fichier ne supprime pas la projection.

L'adresse *addr* doit être un multiple de la taille de page. Toutes les pages contenant une partie de l'intervalle indiqué sont libérées, et tout accès ultérieur déclenchera **SIGSEGV**. Aucune erreur n'est détectée si l'intervalle indiqué ne contient pas de page projetée.

C.4 VALEUR RENVOYÉE

mmap() renvoie un pointeur sur la zone de mémoire, s'il réussit. En cas d'échec il renvoie la valeur **MAP_FAILED** (c'est-à-dire *void **) -1 et *errno* contient le code d'erreur.

munmap() renvoie 0 s'il réussit. En cas d'échec, -1 est renvoyé et *errno* contient le code d'erreur (probablement **EINVAL**).

Structure de corde

Cette partie doit être traitée dans le langage OCaml.

L'édition de grands textes nécessite l'usage de structures de données adaptées de sorte à rendre certaines opérations courantes efficaces. La structure de corde est une structure de données qui vise à remplacer avantageusement la structure de chaîne de caractères, représentée par le type `string` en OCaml.

Question 2.1. En remarquant que les opérations d'insertion et de suppression de texte sont très courantes lors de l'édition de texte, expliquer pourquoi l'usage du type `string` est une mauvaise idée.

La structure de corde est une structure d'arbre binaire strict, i.e. dont chaque nœud interne a exactement deux fils. Les noeuds internes d'une corde représentent la concaténation et les feuilles sont étiquetées par des chaînes de caractères.

Ainsi, la chaîne "Je passe un concours de recrutement d'enseignants." pourrait être représentée par l'arbre en Figure 1.

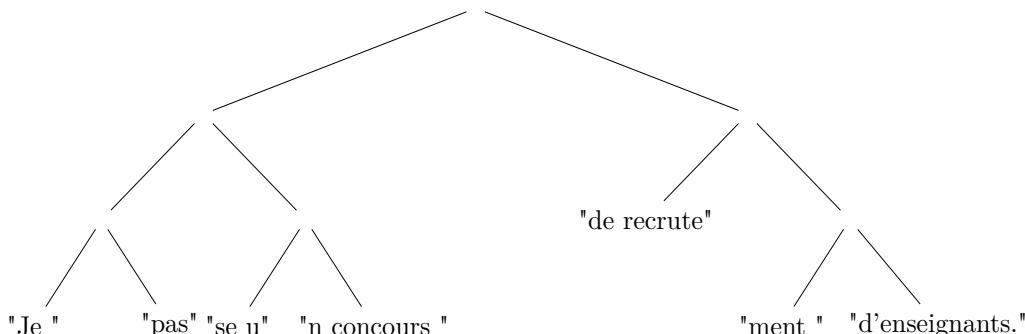


Figure 1: Une corde représentant la chaîne de caractères "Je passe un concours de recrutement d'enseignants."

Nous représentons la structure de corde en OCaml à l'aide du type suivant.

```
type rope =
| Leaf of int * string
| Concat of int * int * rope * rope
```

Une feuille (`Leaf`) est définie par une chaîne de caractères et sa longueur et un nœud interne (`Concat`) par la longueur totale du texte qu'il représente, la hauteur de l'arbre qu'il implémente, et ses sous-arbres gauche et droit.

Avec cette représentation, la corde de la Figure 1 s'implémente comme suit en OCaml:

```

Concat (51, 3,
    Concat (21, 2,
        Concat (6, 1, Leaf (3, "Je "), Leaf (3, "pas")),
        Concat (15, 1, Leaf (4, "se u"), Leaf (11, "n concours "))
    ),
    Concat (29, 2,
        Leaf (10, "de recrute"),
        Concat (19, 1, Leaf (5, "ment "),
            Leaf (14, "d'enseignants.")))
)

```

Par convention, une feuille est de hauteur nulle et un texte vide est représenté par la feuille `Leaf (0, "")`, appelée « corde vide ».

Nous disons qu'une structure de données est persistante si, lorsque l'on applique une opération sur celle-ci, une nouvelle version de la structure est produite de sorte à pouvoir conserver l'ancienne version.

Question 2.2. Expliquer en quoi le type `rope` est adapté pour une implémentation persistante de la structure de corde et quel serait l'intérêt d'une telle implémentation dans un contexte d'édition de texte.

Nous allons maintenant étudier l'implémentation de quelques opérations sur la structure de corde. Dans toute la suite, le terme complexité désignera la complexité temporelle dans le pire des cas, qui devra être exprimée soit en fonction de la longueur n des textes représentés par les cordes en argument, soit en fonction de la hauteur h des cordes en argument. Toute complexité, qu'elle soit demandée dans une question ou imposée par l'énoncé, devra être soigneusement justifiée.

Question 2.3. Écrire deux fonctions `length` et `height` de type `rope -> int` qui déterminent respectivement la longueur du texte représenté par une corde et la hauteur de la corde. Donner la complexité de ces fonctions.

Question 2.4. Écrire une fonction `get : int -> rope -> char` qui prend un indice i et une corde r et qui renvoie le caractère d'indice i dans le texte représenté par la corde r .

Par convention, les indices commencent à 0, comme pour le type `string`. On supposera que l'entier i est valide sans le vérifier. Cette fonction doit être de complexité $\mathcal{O}(h)$.

La complexité de l'accès à un caractère donné d'un texte ne met pas en valeur l'intérêt des cordes par rapport aux chaînes de caractères. Cependant, c'est une opération assez peu courante car un éditeur de textes affiche plutôt de grandes portions de texte sans réaliser des accès caractère par caractère. Un parcours de la structure de corde permet justement d'accéder à des portions de texte et d'y appliquer des transformations locales, comme des insertions et des suppressions. Nous allons maintenant voir que ces opérations sont efficaces sur la structure de corde.

Nous commençons par la concaténation de deux cordes. Nous supposons définie une variable globale `max_string_length` qui représente la longueur maximale autorisée pour une chaîne de caractères dans une feuille d'une corde.

Question 2.5. Écrire une fonction `concat : rope -> rope -> rope` qui concatène deux cordes.

Cette fonction doit être de complexité $\mathcal{O}(1)$ et on cherchera à limiter la croissance des arbres en effectuant les simplifications suivantes:

- la concaténation, à gauche ou à droite, de la corde vide avec une corde `r` vaudra `r`;
- la concaténation de deux feuilles sera remplacée par une unique feuille si la longueur de la chaîne obtenue ne dépasse pas `max_string_length`;
- par extension, la concaténation d'une feuille et d'un nœud `Concat` dont l'un des sous-arbres est une feuille sera simplifiée si la concaténation se fait du côté de cette feuille et si la longueur maximale des feuilles n'est pas dépassée.

Nous supposons maintenant qu'une fonction `split : int -> rope -> rope * rope` est implémentée. Cette fonction prend un entier `i` en argument et une corde `r` et renvoie le couple formé des cordes représentant le préfixe de longueur `i` du texte représenté par `r` et le suffixe correspondant. Par exemple, pour la corde de la Figure 1, si l'on fait appel à la fonction `split` avec l'entier 9, on obtient un couple de cordes représentant respectivement les chaînes "Je passe " et "un concours de recrutement d'enseignants.".

Question 2.6. En déduire une fonction `insert_in_rope : int -> rope -> rope -> rope` telle que `insert_in_rope i r1 r2` renvoie une corde représentant le texte associé à `r1` où l'on a inséré le texte associé à `r2` juste après le préfixe de longueur `i`.

On supposera que l'entier `i` est valide sans le vérifier. On garantira que la complexité de cette fonction ne présente qu'un surcoût constant par rapport à la complexité de la fonction `split`.

Question 2.7. De même, déduire de la fonction `split` une fonction `delete_in_rope : int -> int -> rope -> rope` telle que `delete_in_rope i len r` renvoie une corde représentant le texte associé à `r` où l'on a effacé le facteur de longueur `len` qui succède au préfixe de longueur `i`. On supposera que les entiers `i` et `len` sont valides sans le vérifier. On garantira que la complexité de cette fonction ne présente qu'un surcoût constant par rapport à la complexité de la fonction `split`.

Question 2.8. Écrire la fonction `split`.

On supposera que l'entier en argument est valide. On pourra utiliser la fonction `String.sub` de type `string -> int -> int -> string`, telle que `String.sub s i len` renvoie le facteur de longueur `len` de `s` commençant à l'indice `i`, et de complexité $\mathcal{O}(\text{len})$. On garantira une complexité $\mathcal{O}(h)$, en supposant que les chaînes de toutes les feuilles de la corde en argument sont de longueur au plus `max_string_length`, considérée comme une constante.

Ainsi, comme elles reposent sur la fonction `split` sans surcoût de complexité, les insertions et suppressions dans un texte représenté par une corde sont effectuées avec une complexité linéaire

en la hauteur de la corde. Cependant, lorsque nous appliquons des modifications à un texte, l'usage des fonctions implémentées dans les questions précédentes ne permet malheureusement pas de contrôler l'évolution de la hauteur de la corde manipulée, ce qui fait que les performances se dégradent à mesure que le texte est modifié. Il peut alors être utile de rééquilibrer la corde pour limiter sa hauteur.

Il est possible de montrer que les opérations d'insertion et de suppression sur une corde équilibrée sont de complexité logarithmique en la longueur du texte qu'elle représente. Cela rend la structure de corde efficace pour l'édition de grands textes.

Pour pouvoir représenter l'historique des versions d'une valeur de la structure de corde, nous considérons un couple de piles non persistantes implémenté à l'aide du type suivant:

```
type 'a undo_redo = 'a list ref * 'a list ref
```

Ainsi, une valeur du type `'a undo_redo` est un couple de piles (`undo`, `redo`) où:

- la pile `undo` est constituée des versions antérieures à la version courante, de la plus récente à la plus ancienne dans l'ordre de dépilement;
- la pile `redo` est constituée des versions ultérieures à la version courante, de la plus récente à la plus ancienne dans l'ordre de dépilement.

Par convention, la version courante est placée au sommet de la pile `redo`.

Par exemple, nous pouvons représenter un fichier vide sans historique par la valeur suivante:

```
(ref [], ref [Leaf (0, "")])
```

Après insertion du mot « informatique » dans ce fichier, nous pourrions obtenir un couple de piles (`undo`, `redo`) dont les contenus respectifs seraient `[Leaf (0, "")]` et `[Leaf (12, "informatique")]`.

Question 2.9. Écrire des fonctions `undo` et `redo`, de type `'a undo_redo -> unit`, qui permettent respectivement de revenir en arrière et d'aller vers l'avant dans l'historique en argument. On lèvera une exception lorsque l'opération est impossible.

Question 2.10. Écrire une fonction `insert : rope undo_redo -> int -> rope -> unit` telle que `insert h i r` insère dans le texte associé à la corde courante de `h` le texte associé à `r` juste après le préfixe de longueur `i`.

On supposera que l'entier `i` est valide sans le vérifier.

Question 2.11. De même, écrire une fonction `delete : rope undo_redo -> int -> int -> unit` telle que `delete h i len` efface de la corde courante de `h` le facteur de longueur `len` qui succède au préfixe de longueur `i`.

On supposera que les entiers `i` et `len` sont valides sans le vérifier.

Recherche de motif

Toute la programmation de cette partie est en langage PYTHON. On considère que les opérations suivantes se font en temps $\mathcal{O}(1)$ dans le pire des cas : ajouter ou rechercher dans un dictionnaire (`dict`) ou dans un ensemble (`set`), ajouter en fin de liste (`list`), et supprimer en fin de liste. C'est une simplification puisqu'il faudrait parler de complexité amortie et en moyenne. On considère également que les entiers utilisés restent petits, et que les opérations arithmétiques se font donc en temps $\mathcal{O}(1)$ dans le pire cas. Finalement, on utilisera qu'ajouter n éléments dans un ensemble ou n clés dans un dictionnaire initialement vide, sans faire de suppression, crée une structure qui prend un espace mémoire en $\mathcal{O}(n)$.

Définition : l'*espace additionnel* pris par un algorithme ou une fonction est la quantité de mémoire utilisée quand on ne tient pas compte de la mémoire pour stocker l'entrée et la sortie.

Le problème auquel on s'intéresse dans cette partie consiste à déterminer la liste, éventuellement vide, de toutes les occurrences de X dans T , où X et T sont deux mots non vides. Les occurrences sont représentées par les indices de T où commencent les occurrences (en prenant comme convention que la première lettre de T est à l'indice 0).

Formellement, pour un entier positif ou nul i , on dit que T a une occurrence de X en position i si il existe un mot Y de longueur i tel que YX est un préfixe de T .

Le problème RECHERCHE (X,T) consiste à calculer la liste de toutes les positions des occurrences de X dans T , dans l'ordre croissant. Ainsi, une solution à ce problème devra renvoyer :

- `[0, 2, 8]` pour $X=aba$ et $T=abababbaabaa$
- `[]` pour $X=abc$ et $T=abababbaabaa$

Dans toute la suite on note $n = |T|$ la longueur de T et $m = |X|$ la longueur de X . Pour les questions de complexités, on pourra considérer que m est significativement plus petit que n .

Considérations importantes :

- Comme c'est l'objectif de cette partie, vous n'êtes pas autorisé à utiliser l'un des différents tests de présence d'une sous-chaîne fourni par le langage PYTHON, comme "`ab`" in "`bababa`", "`bababa`".`find("ab")`, On pourra cependant se servir de `in` pour tester si un élément est dans une liste, un dictionnaire, ou un ensemble. On pourra également utiliser `in` pour parcourir une structure itérable : `for i in range(10)`, `for l in "abababb"`, `for x in [1,4,3,7]` sont autorisés.
- Sauf si c'est explicitement précisé différemment dans l'énoncé, toutes les complexités sont des complexités en temps et dans le pire cas.

- Quand on vous demande d'écrire une fonction PYTHON avec des contraintes de complexité en temps ou en espace additionnel, on ne vous demande pas de justifier que votre fonction a les bonnes complexités.
- On considère que les arguments passés à une fonction vérifient toujours les pré-conditions de l'énoncé, il n'est pas nécessaire de les tester.

Question 3.1. Écrire une fonction PYTHON nommée `occurrence_position(X, T, i)` qui renvoie `True` si T a une occurrence de X en position i , et `False` sinon. Votre fonction doit avoir une complexité en $\mathcal{O}(m)$ en temps et utiliser un espace additionnel en $\mathcal{O}(1)$.

L'algorithme de la fenêtre est une solution au problème RECHERCHE obtenue en appelant `occurrence_position(X, T, i)` pour des valeurs de `i` croissante, comme présenté dans la fonction `fenetre` suivante :

```

1 def fenetre(X, T):
2     solution = []
3     for i in range(len(T)-len(X)+1):
4         if occurrence_position(X, T, i):
5             solution.append(i)
6     return solution

```

Question 3.2. Donner sous forme de \mathcal{O} et en fonction de n et de m , une borne atteinte pour la complexité de `fenetre` et proposez des valeurs pour X et T qui montrent qu'elle est atteinte.

Pour les valeurs de i considérées dans `fenetre`, si T n'a pas une occurrence de X , c'est qu'il existe un entier $k \in \{0, \dots, m-1\}$ tel que $T[i+k] \neq X[k]$. On peut ainsi créer une fonction PYTHON appelée `difference_position(X, T, i)` qui renvoie un entier k tel que $T[i+k] \neq X[k]$ si une telle valeur existe et -1 sinon.

Question 3.3. Écrire en PYTHON deux versions de `difference_position(X, T, i)` : l'une nommée `difference_position_debut(X, T, i)` qui renvoie la plus petite valeur de k possible quand T n'a pas d'occurrence de X en position i , et l'autre nommée `difference_position_fin(X, T, i)` qui renvoie la plus grande valeur de k possible. Vos fonctions doivent avoir une complexité en $\mathcal{O}(m)$ en temps et utiliser un espace additionnel en $\mathcal{O}(1)$.

On souhaite modifier la fonction `fenetre` en tenant compte du résultat donné par la fonction `difference_position(X, T, i)` (ou ses variantes de la question précédente). Pour cela on remarque que si elle renvoie `k` et que le caractère $T[i+k]$ n'est pas un des caractères de X , alors il ne peut pas y avoir d'occurrence de X dans T en positions $i, i+1, \dots, i+k$. Il y a donc des cas où on peut augmenter i de plus que 1 lors d'une itération de la boucle.

Question 3.4. Écrire la fonction PYTHON nommée `fenetre_saut(X, T)` qui modifie `fenetre(X, T)` en utilisant l'idée décrite ci-dessus. Votre fonction doit avoir la même complexité pire cas que `fenetre(X, T)` en temps et un espace additionnel en $\mathcal{O}(m)$.

Question 3.5. Pour tout $m \geq 1$ et $n \geq m$, proposez des mots X_m et T_n les plus favorables possibles pour la complexité temporelle, pour chacune des trois fonctions suivantes (on rappelle que n est beaucoup plus grand que m par hypothèse) :

- la fonction `fenetre` ;
- la fonction `fenetre_saut` qui utilise `difference_position_debut(X, T, i)`;
- la fonction `fenetre_saut` qui utilise `difference_position_fin(X, T, i)`.

Indiquer à chaque fois une estimation de la complexité obtenue pour les X_m et T_n proposés. On vous demande de justifier votre réponse.

On cherche à présent à construire un algorithme pour RECHERCHE qui se base sur un automate déterministe et complet. On va construire une famille d'automates $(\mathcal{A}_u)_{u \in \Sigma^*}$, un pour chaque mot sur l'alphabet Σ . Pour tout mot $u \in \Sigma^*$ l'automate \mathcal{A}_u a pour ensemble d'états $Q_u = \{0, 1, \dots, |u|\}$, pour état initial 0 et pour état terminal $|u|$. On définit inductivement sur la longueur de u sa fonction de transition $\delta_u : Q_u \times \Sigma \rightarrow Q_u$ de la façon suivante :

- si $u = \varepsilon$ est le mot vide, alors $Q_\varepsilon = \{0\}$ et on a $\delta_\varepsilon(0, a) = 0$ pour tout $a \in \Sigma$;
- sinon, on écrit $u = v\alpha$, où $v \in \Sigma^*$ est un mot et $\alpha \in \Sigma$ est une lettre, on note $p = \delta_v(|v|, \alpha)$ et on définit pour tout état $q \in Q_u$ et toute lettre $\sigma \in \Sigma$:

$$\delta_u(q, \sigma) = \begin{cases} |u| & \text{si } q = |u| - 1 \text{ et } \sigma = \alpha, \\ |u| & \text{si } q = |u| \text{ et } (p, \sigma) = (|u| - 1, \alpha) \\ \delta_v(p, \sigma) & \text{si } q = |u| \text{ et } (p, \sigma) \neq (|u| - 1, \alpha) \\ \delta_v(q, \sigma) & \text{sinon.} \end{cases} \quad (1)$$

On représente δ_u par un dictionnaire qui a pour clés les couples (état,lettre) auxquels on associe leur image par δ_u : une transition $p \xrightarrow{a} q$, c'est-à-dire $\delta_u(p, a) = q$ correspond dans le code à `delta[(p,a)] = q`.

Question 3.6. Représenter graphiquement \mathcal{A}_{aabaa} , ainsi que la représentation en PYTHON de δ_{aabaa} pour $\Sigma = \{a, b\}$.

Question 3.7. Écrire une fonction PYTHON nommée `automate(u, alphabet)`, qui prend en argument un mot u sous forme d'une chaîne de caractères et l'alphabet `alphabet` sous forme d'une liste de caractères et qui renvoie le dictionnaire représentant δ_u . Votre fonction doit avoir une complexité temporelle en $\mathcal{O}(|u|)$, en considérant la taille de l'alphabet comme fixée.

De façon classique, on étend inductivement la fonction de transition δ d'un automate déterministe d'ensemble d'états Q sur l'alphabet Σ pour tout mot $w \in \Sigma^*$, toute lettre $a \in \Sigma$ et tout

état $p \in Q$ par (ε est le mot vide) :

$$\begin{cases} \delta(p, \varepsilon) = p, \\ \delta(p, wa) = \delta(\delta(p, w), a). \end{cases}$$

Question 3.8. Soient u et v deux mots de Σ^* et $\alpha \in \Sigma$ une lettre, tels que $u = v\alpha$. Soit $p = \delta_v(|v|, \alpha)$. Montrez que pour tout mot $w \in \Sigma^*$ on a ou bien $\delta_u(0, w) = \delta_v(0, w)$, ou bien $\delta_u(0, w) = |u|$ et $\delta_v(0, w) = p$.

Question 3.9. Montrer que pour tout mot u l'automate \mathcal{A}_u reconnaît le langage Σ^*u .

Question 3.10. En utilisant l'automate \mathcal{A}_u sur un alphabet bien choisi, écrire une fonction PYTHON appelée `recherche_automate(X, T)` qui résout le problème RECHERCHE en temps $\mathcal{O}(n + m)$.

* *
*

Mastermind

Ce sujet est consacré à l'étude d'un jeu de stratégie, inspiré du jeu de société Mastermind. On s'intéresse en particulier à la recherche de stratégies optimales et aux moyens d'effectuer cette recherche efficacement.

Préliminaires

Organisation du sujet Le sujet est constitué de quatre parties. Les définitions, concepts et idées introduits dans une partie sont généralement utiles pour les parties suivantes ; cependant, il est toujours possible d'admettre le résultat d'une question (ou de supposer qu'une fonction demandée à une question a été écrite) pour traiter les questions suivantes.

Attendus Il est attendu des candidates et des candidats des réponses construites. Leur évaluation portera aussi sur la précision, le soin et la clarté de la rédaction.

Programmation Au début de chaque partie comportant des questions de programmation, le langage à utiliser (Python, OCaml ou C suivant la partie) est précisé : cette consigne devra impérativement être respectée.

Pour les parties en Python, on se limitera aux fonctions disponibles sans effectuer de `import`.

Pour les parties en C, on supposera que les fichiers d'entête suivants ont été inclus : `stdlib.h`, `stdio.h`, `string.h`, `assert.h`, `stdbool.h` et `stdint.h`.

Pour les parties en OCaml, on pourra utiliser librement les fonctions des modules `List`, `Array` et `Hashtbl` de la bibliothèque standard, ainsi que les fonctions du module initialement ouvert (celles n'étant pas préfixées par un nom de module, comme `incr`, `min`, ...). On rappelle la spécification de quelques fonctions d'ordre supérieur pouvant être utiles :

- `List.map` : $('a \rightarrow 'b) \rightarrow 'a\ list \rightarrow 'b\ list$
`List.map f [x1; ...; xn]` renvoie $[f\ x1; ...; f\ xn]$;
- `List.iter` : $('a \rightarrow unit) \rightarrow 'a\ list \rightarrow unit$
`List.iter f [x1; ...; xn]` équivaut à $f\ x1; ...; f\ xn; ()$;
- `List.fold_left` : $('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow 'b\ list \rightarrow 'a$
`List.fold_left f init [x1; ...; xn]` renvoie $f\ (... (f\ (f\ init\ x1)\ x2)\ ...)$;
- `List.filter` : $('a \rightarrow bool) \rightarrow 'a\ list \rightarrow 'a\ list$
`List.filter pred u` renvoie la liste constituée des éléments x de u tels que $pred\ x = true$, dans l'ordre de la liste u .

Ces fonctions (sauf `List.filter`) sont également présentes dans le module `Array`, avec un type adapté.

Pour le module `Hashtbl`, on rappelle les fonctions essentielles :

- ('a, 'b) Hashtbl.t est le type d'une table de hachage dont les clés sont de type 'a et les valeurs de type 'b;
- Hashtbl.create : int → ('a, 'b) Hashtbl.t
Hashtbl.create 1 renvoie une table de hachage vide (l'argument entier est une indication sur la taille initiale du tableau sous-jacent et pourra systématiquement être pris égal à 1);
- Hashtbl.find_opt : ('a, 'b) Hashtbl.t → 'a → 'b option
Hashtbl.find_opt h x renvoie Some y si y est la valeur associée à la clé x dans h, ou None si la clé x n'est pas présente dans h;
- Hashtbl.replace : ('a, 'b) Hashtbl.t → 'a → 'b → unit
Hashtbl.replace h x y associe la clé x à la valeur y dans la table h, en remplaçant l'ancienne association pour x s'il y en avait une (et en créant l'association dans le cas contraire);
- Hashtbl.iter : ('a → 'b → unit) → ('a, 'b) Hashtbl.t → unit
Hashtbl.iter f h équivaut à f x1 y1; ... ; f xn yn; (), où $(x_1, y_1), \dots, (x_n, y_n)$ sont les associations présentes dans la table h, dans un ordre non spécifié.

On supposera que les fonctions Hashtbl.create, Hashtbl.find_opt et Hashtbl.replace s'exécutent en temps constant, et que Hashtbl.iter prend un temps proportionnel à la somme des complexités des appels à f effectués.

Présentation informelle du jeu

Dans sa version standard, ce jeu fait intervenir un *arbitre* et un joueur, que l'on appellera J_1 . L'arbitre commence par choisir une *combinaison* secrète, qui est constituée d'une suite de N jetons colorés (N est un paramètre du jeu, fixé à l'avance). La couleur de chaque jeton de la combinaison est choisie librement parmi un ensemble de P couleurs, l'ordre des jetons à l'intérieur de la combinaison est important et les répétitions sont autorisées.

Par exemple, si $N = 5$ et $P = 3$ (disons que les couleurs possibles sont le rouge, le vert et le bleu, notées R, V, B), l'arbitre peut choisir la combinaison (R, R, V, B, R). Le but du joueur J_1 est de deviner cette combinaison en utilisant le moins d'essais possibles. La partie se déroule comme suit :

- J_1 propose une combinaison c : par exemple, $c = (V, B, V, R, B)$;
- l'arbitre répond à J_1 en indiquant le nombre de jetons bien placés et mal placés dans sa combinaison;
- ici, il y a un V bien placé :

R	R	V	B	R
V	B	V	R	B

- pour déterminer le nombre de jetons mal placés, on commence par éliminer les jetons bien placés puis l'on oublie l'ordre – ici, il y a deux jetons mal placés (un R et un B);

R	R	R	B
R		V	B

- si l'arbitre répond que tous les jetons sont bien placés (réponse $(N, 0)$), la partie s'arrête;
- sinon, J_1 fait une nouvelle proposition et l'on continue.

Formalisation

- On se donne deux entiers strictement positifs N et P – dans la version standard du jeu, on a $N = 4$ et $P = 6$.
- On appelle *combinaison* un N -uplet d'éléments de $\{0, \dots, P-1\}$. *Attention, il s'agit bien d'un N -uplet et non d'une combinaison au sens usuel en mathématiques : l'ordre compte et les répétitions sont autorisées.*
- On notera C (ou $C(N, P)$ s'il y a risque d'ambiguïté) l'ensemble de ces combinaisons : $C = \{0, \dots, P-1\}^N$. On a donc $|C| = P^N$.
- On appellera *coulours* les éléments de $\{0, \dots, P-1\}$ et *jetons* les éléments du N -uplet.
- Étant données deux combinaisons $x = (x_0, \dots, x_{N-1})$ et $y = (y_0, \dots, y_{N-1})$, la *similarité* $sim(x, y)$ entre x et y est le couple d'entiers (b, m) où :
 - b est le nombre de jetons *bien placés* de y (par rapport à x), c'est-à-dire le nombre d'indices $i \in \{0, \dots, N-1\}$ tels que $x_i = y_i$;
 - m est le nombre de jetons *mal placés* de y (par rapport à x). Ce nombre est tel que $b+m$ soit égal au nombre de jetons en commun entre x et y si l'on ne tient pas compte de l'ordre.
- Autrement dit, on commence par compter le nombre de jetons bien placés puis on les élimine et l'on compte le nombre de jetons identiques sans tenir compte de l'ordre, comme illustré au paragraphe précédent.
- Quelques exemples :
 - $sim((2, 3, 0, 2, 3), (3, 3, 3, 1, 2)) = (1, 2)$ – en effet, il y a au total trois éléments communs (un « 2 » et deux « 3 »), dont l'un est bien placé (le « 3 » à l'indice 1, c'est-à-dire en deuxième position) ;
 - $sim((1, 2, 3), (2, 2, 2)) = (1, 0)$ (un seul élément commun, le « 2 » à l'indice 1 qui est bien placé) ;
 - $sim((1, 2, 3, 1, 4), (2, 4, 3, 1, 1)) = (2, 3)$ (à l'ordre près, tous les éléments sont communs, et deux d'entre eux sont bien placés).
- De manière immédiate, on a $x = y$ si et seulement si $sim(x, y) = (N, 0)$. On remarque de plus que sim est symétrique.

Question 1. Donner sans justification les valeurs de :

- $sim((2, 1, 3, 4), (1, 2, 3, 4))$;
- $sim((0, 0, 1, 1), (1, 1, 3, 0))$;
- $sim((0, 3, 3, 2, 3), (3, 0, 3, 4, 4))$.

Notations et définitions supplémentaires

- On note R l'ensemble des similarités (ou *réponses*) possibles :

$$R = \{sim(x, y) \mid x, y \in C\}.$$

- Pour $X \subseteq C$ et $c \in C$, on note

$$sim(X, c) = \{sim(x, c) \mid x \in X\}.$$

- Pour $X \subseteq C$ et $c \in C$ et $r \in R$, on note

$$filtre(X, c, r) = \{x \in X \mid sim(c, x) = r\}.$$

- Pour une liste (éventuellement vide) $h = [(c_1, r_1), \dots, (c_k, r_k)]$ (appelée *historique*) de couples (combinaison, similarité), on définit l'ensemble des combinaisons *compatibles avec* h par

$$\text{compat}(h) = \{x \in C \mid \forall i \in \{1, \dots, k\}, \sim(x, c_i) = r_i\}.$$
- Un tel historique est dit *admissible* si $\text{compat}(h) \neq \emptyset$ et si les r_i avec $1 \leq i < k$ sont différents de $(N, 0)$.
- Un historique admissible est dit *inachevé* si $k = 0$ (l'historique est vide) ou $r_k \neq (N, 0)$.

Jeu à un joueur Le principe du jeu est le suivant :

- une combinaison $but \in C$ est choisie par l'arbitre et gardée secrète ;
- le joueur J_1 doit deviner cette combinaison en un minimum d'essais ;
- pour ce faire, il propose à chaque coup une combinaison $c \in C$;
- l'arbitre indique alors au joueur J_1 la valeur de $\sim(x, but)$;
- si cette valeur est $(N, 0)$ (c'est-à-dire si $x = but$), la partie s'arrête ;
- sinon, le joueur J_1 propose une nouvelle combinaison ;
- le score du joueur J_1 est le nombre total d'essais effectués pour trouver but . Le joueur J_1 cherche donc à minimiser ce score. Si la partie ne se termine jamais, on considère que ce score est infini.

Exemple. On prend ici $N = 3$ et $P = 3$.

- L'arbitre choisit $but = (2, 2, 0)$ comme combinaison secrète à deviner.
- Le joueur J_1 joue $(0, 0, 0)$, l'arbitre répond $\sim((0, 0, 0), but) = (1, 0)$.
- Le joueur J_1 joue $(0, 1, 2)$, l'arbitre répond $\sim((0, 1, 2), but) = (0, 2)$.
- Le joueur J_1 joue $(2, 2, 0)$, l'arbitre répond $(3, 0)$ et la partie se termine. Le score de la partie vaut 3.

Partie I. Programmation des fonctions élémentaires

Dans cette partie, on se propose d'implémenter **en utilisant le langage Python** les fonctions de base permettant de jouer, ainsi qu'une stratégie très simple pour le joueur J_1 .

On suppose définies deux constantes globales \mathbb{N} et \mathbb{P} , et l'on représente :

- une combinaison par une `list` de longueur \mathbb{N} à valeurs dans $\{0, \dots, \mathbb{P} - 1\}$;
- une réponse par un couple (b, m) d'entiers.

Une fonction prenant en entrée une combinaison pourra toujours supposer qu'elle est bien formée (c'est-à-dire de longueur N et ne contenant que des entiers de $\{0, \dots, P - 1\}$), et une fonction renvoyant une combinaison devra s'assurer qu'elle est bien formée.

I.1 Calcul de la similarité

Question 2. Écrire une fonction `bien_ou_mal_places` prenant en entrée deux combinaisons x et y et renvoyant l'entier $b+m$, où $(b, m) = \sim(x, y)$. On demande une complexité en $\mathcal{O}(N + P)$, que l'on justifiera brièvement.

Question 3. Écrire une fonction `sim` prenant en entrée deux combinaisons x et y et renvoyant $sim(x, y)$. Préciser sa complexité, en la justifiant brièvement.

Question 4. Écrire une fonction `compatible` prenant en entrée une combinaison c et un historique h , sous la forme d'une liste de couples (combinaison, réponse), et indiquant si $c \in compat(h)$.

I.2 Stratégie naïve

On suppose que le joueur J_1 utilise la stratégie suivante : il joue systématiquement la plus petite combinaison (dans l'ordre lexicographique) qui est compatible avec l'historique actuel.

Question 5. Écrire une fonction `avance` prenant en entrée une combinaison c et la modifiant en la remplaçant par la combinaison suivante dans l'ordre lexicographique (cette fonction ne renverra donc rien). Si cette fonction est appelée sur la liste $[P - 1, \dots, P - 1]$ (dernière combinaison dans l'ordre lexicographique), elle lèvera l'exception `ValueError`. Préciser sa complexité dans le pire cas (on ne demande pas de justification).

Question 6. On note $\Phi(c)$ le nombre d'occurrences de $P - 1$ dans la combinaison c et $T(c)$ le nombre d'éléments de c modifiés pendant l'appel `avance(c)`. En notant c' l'état de la liste c après l'appel, majorer la quantité $T(c) + \Phi(c') - \Phi(c)$.

Question 7. Que peut-on dire du coût total d'une série de k appels successifs sur une combinaison initialement égale à $(0, \dots, 0)$ (en supposant que les k appels ont lieu sans déclencher d'exception) ? Quel type de calcul de complexité vient-on d'effectuer ?

Question 8. Écrire une fonction `joue_naif` prenant en entrée la combinaison but choisie par l'arbitre et renvoyant l'historique de la partie que l'on obtient si le joueur J_1 utilise la stratégie naïve. Par exemple, pour $N = 3$ et $P = 4$, l'appel `joue_naif([3, 3, 1])` doit renvoyer la liste :

```
[[[0, 0, 0], (0, 0)), ([1, 1, 1], (1, 0)), ([1, 2, 2], (0, 1)),
 ([3, 1, 3], (1, 2)), ([3, 3, 1], (3, 0))]
```

Partie II. Généralités sur les stratégies

Dans toute cette partie, D désigne une partie non vide de C .

II.1 Arbre de stratégie

Un *D-arbre de stratégie* est un arbre dont chaque nœud interne est étiqueté par une combinaison $c \in C$ et chaque arête est étiqueté par une réponse $r \in R$ et qui vérifie les conditions suivantes :

- en notant c l'étiquette de la racine, on a un sous-arbre pour chaque $r \in sim(D, c)$ (et l'arête reliant la racine à ce sous-arbre est étiquetée r) ;
- si $(N, 0) \in sim(D, c)$, alors le sous-arbre correspondant est une feuille (que l'on étiquettera parfois c , même si cette étiquette est redondante) ;
- pour chaque $r \in sim(D, c)$, $r \neq (N, 0)$:
 - $|filtre(D, c, r)| < |D|$;
 - le sous-arbre correspondant à r est un $filtre(D, c, r)$ -arbre de stratégie.

On note \mathcal{T}_D l'ensemble des D -arbres de stratégie.

Exemples

- Si $D = \{c\}$ (singleton), alors la combinaison à la racine est nécessairement c (tout autre choix conduirait à violer la condition $|filtre(D, c, r)| < |D|$) et l'unique D -arbre de stratégie est donc :

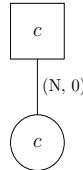


FIGURE 1 – Arbre pour un singleton.

- Pour $N = 3$, $P = 4$ et $D = \{(1, 2, 2), (3, 3, 1)\}$, de nombreux arbres sont possibles, dont les deux représentés ci-dessous :

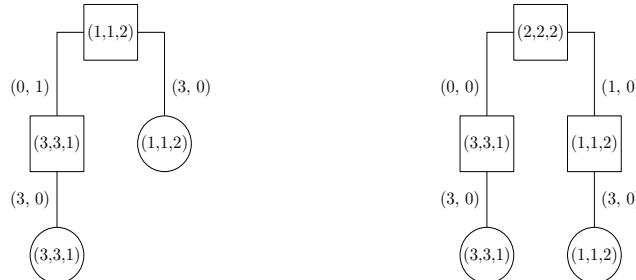
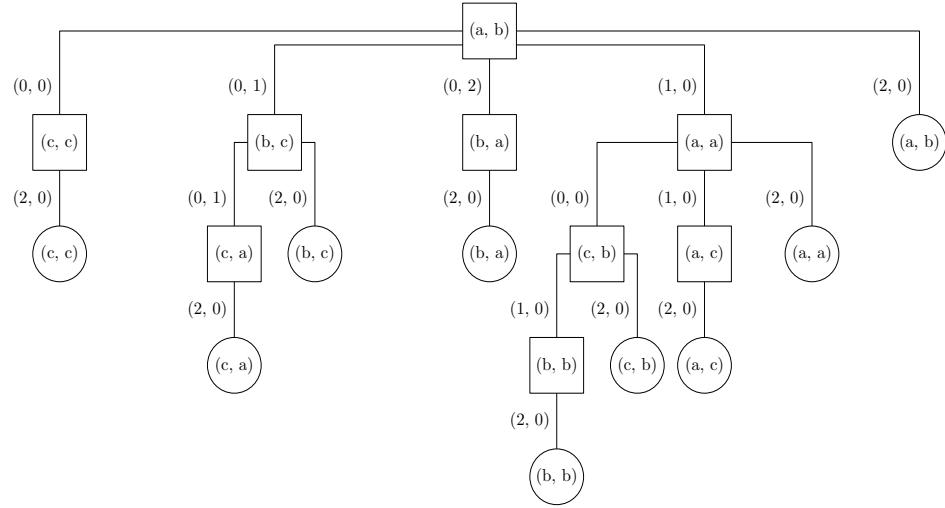


FIGURE 2 – Deux arbres possibles pour $D = \{(1, 2, 2), (3, 3, 1)\}$.

On notera en revanche qu'aucun D -arbre ne peut ici avoir de racine étiquetée par $c = (0, 0, 0)$: pour $r = (0, 0)$, on aurait $filtre(D, c, r) = D$.

- Pour $N = 2$ et $P = 3$, l'arbre de la figure 3 en page suivante est un C -arbre de stratégie (dans cet arbre, on a noté les couleurs a, b, c au lieu de 0, 1, 2 pour éviter les confusions entre combinaisons et réponses).

Question 9. On note n le nombre de nœuds internes et f le nombre de feuilles d'un D -arbre de stratégie T . Montrer que $f = |D|$ et $n \geq |D|$.

FIGURE 3 – L'arbre de stratégie T_0 .**Jeu suivant un arbre de stratégie**

- On dit qu'un historique $h = ((c_1, r_1), \dots, (c_k, r_k))$ est *joué suivant l'arbre T* s'il existe un chemin partant de la racine et étiqueté $c_1, r_1, c_2, \dots, c_k, r_k$ (en alternant étiquette des nœuds internes et étiquettes des arêtes). Notons que s'il existe, ce chemin est unique.
- Pour un D -arbre de stratégie T et une combinaison $x \in D$, il existe une unique feuille de T étiquetée x , et donc un unique chemin de la racine à cette feuille. L'historique associé à ce chemin (qui se termine par $(x, (N, 0))$) est appelé *partie jouée suivant T pour le but x* .
- On note alors $score(T, x)$ le score de cette partie.

Intuitivement, un D -arbre de stratégie indique à J_1 comment jouer, s'il sait déjà que le but appartient à D :

- la racine lui indique le premier coup à jouer ;
- il reçoit une réponse r et il sait maintenant que le but est dans $filtrer(D, c, r)$;
- il descend dans le sous-arbre correspondant, qui est soit une feuille (si $r = (N, 0)$, et la partie est alors terminée), soit un $filtrer(D, c, r)$ -arbre qui lui indiquera comment jouer la suite de la partie.

Exemple On reprend l'arbre T_0 de la figure 3, et l'on suppose que $but = (c, a)$. La partie jouée suivant T_0 se déroule alors ainsi :

- le joueur J_1 joue (a, b) (le coup à la racine) ;
- comme $sim((a, b), but) = (0, 1)$, l'arbitre répond $(0, 1)$ et l'on descend dans la branche correspondante ;
- J_1 joue ensuite (b, c) ;
- l'arbitre répond $sim((b, c), but) = (0, 1)$, on descend dans cette branche ;
- J_1 joue (c, a) , l'arbitre répond $(2, 0)$ et l'on descend dans cette branche ;
- on arrive sur une feuille, ce qui indique que la partie est terminée.

II.2 Stratégie optimale

- Le *score dans le pire cas* $\text{pire}_D(T)$ d'un D -arbre de stratégie T est défini par :

$$\text{pire}_D(T) = \max_{x \in D} \text{score}(T, x).$$

- Le *poids total* $\text{poids}_D(T)$ d'un D -arbre de stratégie T est défini par :

$$\text{poids}_D(T) = \sum_{x \in D} \text{score}(T, x).$$

- On définit :

$$\begin{aligned} \text{pire}_{\text{opt}}(D) &= \min\{\text{pire}_D(T) \mid T \in \mathcal{T}_D\} \\ \text{poids}_{\text{opt}}(D) &= \min\{\text{poids}_D(T) \mid T \in \mathcal{T}_D\} \end{aligned}$$

- Un D -arbre de stratégie est dit *optimal dans le pire cas* s'il vérifie $\text{pire}_D(s) = \text{pire}_{\text{opt}}(D)$, *optimal en moyenne* s'il vérifie $\text{poids}_D(s) = \text{poids}_{\text{opt}}(D)$.

Question 10. À quoi correspondent $\text{pire}_D(T)$ et $\text{poids}_D(T)$ sur un D -arbre de stratégie T ?

Question 11. Déterminer un C -arbre de stratégie optimal dans le pire cas pour $N = 2$ et $P = 3$, et justifier son caractère optimal. *On pourra modifier l'arbre T_0 de la figure 3.*

Question 12. Déterminer $\text{pire}_{\text{opt}}(D)$ et $\text{poids}_{\text{opt}}(D)$ pour D de cardinal 1 ou 2.

Question 13. Un arbre est dit *optimiste* si chaque nœud interne a un enfant feuille. On note $\text{pire}'_{\text{opt}}(D)$ le minimum des $\text{pire}_D(T)$ pour T un D -arbre de stratégie optimiste. A-t-on nécessairement $\text{pire}'_{\text{opt}}(D) = \text{pire}_{\text{opt}}(D)$?

II.3 Jeu à deux joueurs

On considère une variante à deux joueurs de notre jeu, où l'arbitre est remplacé par un joueur J_2 qui cherche à maximiser le score de la partie. Le k -ième tour de jeu consiste en un coup $c_k \in C$ du joueur J_1 et une réponse r_k du joueur J_2 , et l'on note $h = ((c_1, r_1), \dots, (c_{k-1}, r_{k-1}))$ l'historique des couples (coup, réponse) depuis le début de la partie (au premier tour, $k = 1$ et cette liste est donc vide).

- Le joueur J_1 choisit librement $c_k \in C$;
- le joueur J_2 choisit une réponse $r_k \in \text{sim}(\text{compat}(h), c_k)$;
- si $r_k = (N, 0)$, la partie se termine avec un score final de k , sinon on remplace h par $h, (c_k, r_k)$ et l'on passe au tour suivant.

Question 14. Dans le jeu à deux joueurs, à quoi correspond $\text{pire}_C(T)$ (où T est un C -arbre de stratégie) ? Même question pour $\text{pire}_{\text{opt}}(C)$.

II.4 Représentation informatique

On utilise dans cette partie le langage OCaml. On suppose disposer d'un type `combi` permettant de représenter une combinaison et d'un type `reponse` pour les réponses (on pourra tester l'égalité de deux réponses avec l'opérateur `=`). On suppose également disposer d'une fonction `sim` permettant de calculer la similarité entre deux combinaisons :

```
val sim : combi -> combi -> reponse
```

On représente un arbre de stratégie par un objet du type suivant :

```
type strat =
| Gagne
| Noeud of combi * ((reponse * strat) list)
```

On a choisi de ne pas étiqueter les feuilles (l'étiquette serait nécessairement la même que celle du noeud parent, donc redondante).

Question 15. Écrire une fonction `joue_un_joueur` prenant en entrée un D -arbre de stratégie s et un but x et renvoyant $score(s, x)$. On lèvera une exception si $x \notin D$.

```
val joue_un_joueur : strat -> combi -> int
```

Question 16. Écrire deux fonctions `pire` et `poids` prenant en entrée un D -arbre de stratégie s et renvoyant respectivement $pire_D(s)$ et $poids_D(s)$. On précisera leur complexité en fonction du nombre total n de noeuds de l'arbre et/ou de $|D|$ et l'on cherchera à écrire des versions raisonnablement efficaces.

```
val pire : strat -> int
val poids : strat -> int
```

II.5 Calcul naïf de $pire_{opt}(D)$ et $poids_{opt}(D)$

On définit $split(X)$ comme l'ensemble des $c \in C$ tels quel $|sim(X, c)| > 1$.

Question 17. Pour $X \subseteq C$ et $X \neq \emptyset$, on définit récursivement f par :

$$f(X) = \begin{cases} 1 & \text{si } |X| = 1 \\ 1 + \min_{c \in split(X)} \max_{r \in sim(X, c)} f(filtre(X, c, r)) & \text{sinon} \end{cases}$$

Montrer que f est bien définie et que $f(X) = pire_{opt}(X)$.

Question 18. Donner (sans justification) une définition similaire pour $poids_{opt}$.

On suppose que l'on dispose d'une fonction `reponses_posibles` prenant en entrée un ensemble X de combinaisons, sous forme d'une liste, ainsi qu'une combinaison c , et renvoyant l'ensemble $sim(X, c)$ sous forme d'une liste d'éléments distincts.

```
val reponses_posibles : combi list -> combi -> reponse list
```

On suppose de plus que l'on dispose d'une constante `toutes_combis` de type `combi list` correspondant à l'ensemble C et d'une constante `tous_bons` de type `reponse` correspondant à la réponse $(N, 0)$ (autrement dit, `sim x y = tous_bons` si et seulement si les combinaisons x et y sont égales).

Question 19. Écrire une fonction `pire_opt_naif` prenant en entrée un ensemble $X \neq \emptyset$ de combinaisons sous forme de liste et renvoyant $\text{pire}_{\text{opt}}(X)$. On écrira la fonction la plus simple possible, sans attention particulière portée à l'efficacité.

```
val pire_opt_naif : combi list -> int
```

Question 20. En supposant toujours que l'on dispose de la fonction `reponses_posibles`, écrire une fonction `strategie_poids_naive` prenant en entrée un ensemble X de combinaisons et renvoyant l'arbre d'une stratégie optimale en moyenne pour X (c'est-à-dire de poids égal à $\text{poids}_{\text{opt}}(X)$).

```
val strategie_poids_naive : combi list -> strat
```

Partie III. Recherche efficace de stratégies

III.1 Premières optimisations

La recherche d'une stratégie optimale demande de calculer un très grand nombre de valeurs $\text{sim}(x, y)$ (y compris plusieurs fois pour le même couple (x, y)). Pour accélérer cette recherche, on décide donc de pré-calculer ces valeurs et de les stocker. Dans cette sous-partie, on utilise **le langage C** afin de pouvoir contrôler plus finement les représentations mémoire des objets.

Question 21. Justifier que $|R|$ (le nombre de réponses effectivement possibles) est inférieur ou égal à $\frac{N(N+3)}{2}$.

Question 22. Sachant que N et P seront tous les deux inférieurs ou égaux à 8, proposer deux définitions de type :

- une pour un type `combi_int_t` permettant de représenter un entier de $\{0, \dots, |C| - 1\}$;
 - l'autre pour un type `rep_int_t` permettant de représenter un entier de $\{0, \dots, |R| - 1\}$.
- On choisira des types de taille adaptée et l'on écrira explicitement les `typedef`.

On suppose définis un type `rep_couple_t` ainsi que trois constantes globales `N`, `P` et `NB_COMBIS` (égale à $|C|$) :

```
struct rep_couple_t {
    int bien;
    int mal;
};

typedef struct rep_couple_t rep_couple_t;
```

```
const int N = 4; // valeur purement indicative
const int P = 6; // idem
const int NB_COMBIS = ...; //
```

On suppose de plus que l'on dispose de deux fonctions

```
rep_couple_t similarite(int x[], int y[]);
rep_int_t rep_int_of_rep_couple(rep_couple_t couple);
```

La fonction `similarite` prend en entrée deux combinaisons (sous la forme de deux tableaux de N entiers de $\{0, \dots, P - 1\}$) et renvoie leur similarité sous la forme d'un couple ; on suppose qu'elle s'exécute en temps $\mathcal{O}(N + P)$.

La fonction `rep_int_of_rep_couple` prend un couple (b, m) (valide) et renvoie le code entier correspondant ; on suppose qu'elle s'exécute en temps constant.

Question 23. Écrire deux fonctions (réciproques l'une de l'autre) permettant la conversion entre la représentation « tableau » et la représentation « entier » d'une combinaison :

```
combi_int_t int_of_combi(int x[]);
int *combi_of_int(combi_int_t x);
```

Question 24. Proposer une méthode permettant d'obtenir une fonction `sim` prenant en entrée deux combinaisons (sous forme de `combi_int_t`) et renvoyant leur similarité (sous forme de `rep_int_t`) qui s'exécute en temps constant. On donnera le code de cette fonction, ainsi que celui des éventuelles autres fonctions utilisées, constantes définies, et code d'initialisation à exécuter avant le premier appel à `sim`.

```
rep_int_t sim(combi_int_t x, combi_int_t y);
```

Question 25. Quelle quantité de stockage votre solution utilise-t-elle ? Jusqu'à quelle valeur de $|C|$ cela vous paraît-il raisonnable, sur un ordinateur personnel « standard » ?

Important Dans toute la suite du sujet, on suppose qu'une technique similaire a été mise en œuvre en OCaml de manière à ne plus manipuler que des entiers. Ainsi, les deux types `combi` et `reponse` sont en fait définis ainsi :

```
type combi = int
type reponse = int
```

- On suppose de plus disposer de deux constantes globales `nb_combis` et `nb_reponses` correspondant respectivement à $|C|$ et $|R|$.
- Les combinaisons sont numérotées de 0 à `nb_combis` - 1 et les réponses de 0 à `nb_reponses` - 1. On suppose de plus que la constante `tous_bons`, de type `reponse`, correspond au codage de la réponse $(N, 0)$.
- La constante `toutes_combis` est toujours disponible, et de type `combi list`.
- La fonction `sim` est toujours disponible (et utilise les nouveaux types) mais s'exécute à présent en temps constant.

III.2 Stratégie gloutonne

Pour une partie $X \neq \emptyset$ de C et une combinaison c , on définit :

$$f(X, c) = \max_{x \in X} |\{y \in X \mid y \neq c \text{ et } \text{sim}(c, x) = \text{sim}(c, y)\}|.$$

Question 26. En supposant que $\text{compat}(h) = X$ (où h est l'historique actuel) et que la combinaison à deviner est x , que représente la quantité $|\{y \in X \mid y \neq c \text{ et } \text{sim}(c, x) = \text{sim}(c, y)\}|$?

Un D -arbre de stratégie est dit *glouton* si c'est une feuille ou si l'étiquette c de sa racine vérifie $f(D, c) = \min_{c' \in C} f(D, c')$ et si chacun de ses sous-arbres est glouton. On conviendra que, si plusieurs c réalisent ce minimum, on choisira le plus petit (ce qui permet d'assurer l'unicité de l'arbre glouton).

Question 27. Expliquer pourquoi cette stratégie est raisonnable, et pourquoi on peut la qualifier de *gloutonne*.

Question 28. On fait, dans cette question uniquement, l'hypothèse que $\text{pire}_{opt}(D)$ est une fonction croissante de $|D|$ (ce qui est en général faux). Que peut-on alors dire d'un arbre glouton ?

Question 29. Écrire une fonction `repartit` prenant en entrée un ensemble X de combinaisons (sous forme d'une liste) et une combinaison c et renvoyant une liste $u = [u_1; \dots; u_k]$, de type `combi list list`, et telle que :

- les liste u_i sont non vides et disjointes ;
- l'union des éléments des listes u_i vaut $X \setminus \{c\}$;
- pour chaque u_i , il existe $r_i \neq (N, 0)$ tel que $v_i = \text{filtre}(X, c, r_i)$.

L'ordre des éléments, tant dans la liste u qu'à l'intérieur de chaque liste u_i , n'est pas spécifié. On demande une complexité en $\mathcal{O}(|X|)$, que l'on justifiera.

```
val repartit : combi list -> combi -> combi list list
```

Question 30. Écrire une fonction `choix_glouton` prenant en entrée un ensemble X de combinaisons ainsi qu'un ensemble `coups`, et renvoyant une combinaison $c \in \text{coups}$ telle que $f(X, c) = \min_{c' \in \text{coups}} f(X, c')$ (X et `coups` seront supposés non vides). Déterminer sa complexité.

```
val choix_glouton : combi list -> combi list -> combi
```

Question 31. Écrire une fonction `pire_glouton` prenant en entrée un ensemble X de combinaisons et renvoyant $\text{pire}_X(s_g)$, où s_g est l'arbre glouton pour X .

Question 32. Déterminer la complexité d'un appel à `pire_glouton` avec $X = C$ en fonction de $|C|$ et de $\text{pire}_C(s_g)$.

III.3 Majorations et minorations

Élagage $\alpha\beta$. On définit les deux fonction mutuellement récursives suivantes :

Algorithme 1 $\alpha\beta$.

```

fonction EVAL1(buts,  $\alpha$ ,  $\beta$ )
  si  $|buts| = 1$  alors
    renvoyer 1
   $h_{min} \leftarrow \beta$ 
  pour  $c \in split(X)$  faire
     $h \leftarrow EVAL2(buts, c, \alpha - 1, h_{min} - 1)$ 
     $h_{min} \leftarrow \min(h + 1, h_{min})$ 
    si  $h_{min} \leq \alpha$  alors
      renvoyer  $h_{min}$ 
  renvoyer  $h_{min}$ 

fonction EVAL2(buts,  $c$ ,  $\alpha$ ,  $\beta$ )
   $repartition = \{(r, filtre(buts, c, r)) \mid r \in sim(buts, c)\}$ 
   $h_{max} \leftarrow \alpha$ 
  pour  $(r, buts') \in repartition$  faire
    si  $r \neq (N, 0)$  alors
       $h \leftarrow EVAL1(buts', h_{max}, \beta)$ 
       $h_{max} \leftarrow \max(h_{max}, h)$ 
      si  $h_{max} \geq \beta$  alors
        renvoyer  $h_{max}$ 
  renvoyer  $h_{max}$ 
```

Question 33. Donner la spécification de ces deux fonctions. Comment les utiliser pour calculer $pire_{opt}(C)$?

Question 34. Justifier rapidement leur correction. Expliquer l'intérêt de cette approche par rapport à celle de la partie II.5.

Question 35. Expliquer comment modifier ces fonctions pour calculer une stratégie optimale dans le pire cas pour le joueur J_1 .

Question 36. Quelles modifications faudrait-il apporter aux fonctions EVAL1 et EVAL2 si l'on souhaitait calculer $poids_{opt}(C)$?

Tri des coups. Les boucles principales des fonctions EVAL1 et EVAL2 (*pour* $c \in C$ et *pour* $(r, buts') \in repartition$) s'effectuent pour l'instant dans un ordre non spécifié. On s'intéresse dans cette partie à l'impact que peut avoir le fait de parcourir ces deux ensembles dans un ordre spécifique.

Question 37. Quel serait le pire ordre de parcours possible ? Comment se comporteraient alors les deux fonctions ? On ne demande pas de justification formelle.

Question 38. Est-il réaliste d'espérer parcourir les ensembles dans l'ordre optimal ? Proposer une manière simple (et ayant un coût raisonnable) d'obtenir un ordre satisfaisant pour la fonction EVAL2.

Question 39. Proposer un ordre intéressant de traitement des coups pour EVAL1.

Partie IV. Prise en compte des symétries

Intuitivement, il est clair qu'il n'est pas nécessaire d'étudier à la fois les coups initiaux $(0, 0, 0, 0)$ et $(1, 1, 1, 1)$ lorsqu'on recherche une stratégie optimale : rien ne différenciant *a priori* la couleur 0 de la couleur 1, toute stratégie commençant par l'un doit pouvoir être transformée en une stratégie commençant par l'autre sans modifier aucune des grandeurs qui nous intéressent. Le but de cette partie est de formaliser cette intuition, de l'étendre au-delà du premier coup et d'étudier comment la mettre en œuvre en pratique pour diminuer (massivement) le temps de calcul d'une stratégie optimale.

IV.1 Définitions et premières propriétés

- On appelle *permutation des positions* une permutation σ de l'ensemble $\{0, \dots, N - 1\}$ des positions.
- On appelle *permutation des couleurs* une permutation τ de l'ensemble $\{0, \dots, P - 1\}$ des couleurs.
- On dit qu'une application α de C dans C est une *transformation* s'il existe une permutation des positions σ et une permutation des couleurs τ telles que :

$$\forall x \in C, \alpha(x) = (\tau(x_{\sigma(0)}), \dots, \tau(x_{\sigma(N-1)})).$$

On note alors $\alpha = (\sigma, \tau)$.

- On note \mathcal{S} l'ensemble des transformations.

Exemple. Pour $N = 4$, $P = 6$, $x = (1, 2, 5, 1)$, $\sigma = (2, 1, 3, 0)$ (c'est-à-dire $\sigma(0) = 2$, $\sigma(1) = 1 \dots$) et $\tau = (0, 4, 2, 1, 5, 3)$, on obtient $(\sigma, \tau)(x) = (3, 2, 4, 4)$.

Question 40. Montrer que si $\alpha, \beta \in \mathcal{S}$, alors :

- α est une bijection, et $\alpha^{-1} \in \mathcal{S}$;
- $\alpha \circ \beta \in \mathcal{S}$.

IV.2 Implémentation

Question 41. Discuter le coût en temps et en espace d'un pré-calcul permettant de disposer d'une fonction `applique` prenant en entrée une transformation α et une combinai-

son c et renvoyant $\alpha(c)$. Cette fonction devra s'exécuter en temps constant et avoir le type `transfo -> combi -> combi`, où l'on précisera le type `transfo` (on ne demande pas d'écrire le code du pré-calcul ni de la fonction `applique`).

On suppose dans la suite que l'on dispose d'une telle fonction, ainsi que d'une liste `toutes_transfos` contenant l'ensemble des transformations.

```
val applique : transfo -> combi -> combi
val toutes_transfos : transfo list
```

IV.3 Équivalence de coups

Une partie \mathcal{T} de \mathcal{S} est dite *close* si elle vérifie les trois conditions suivantes :

- (i) $\text{Id}_C \in \mathcal{T}$;
- (ii) si $\alpha, \beta \in \mathcal{T}$, alors $\alpha \circ \beta \in \mathcal{T}$;
- (iii) si $\alpha \in \mathcal{T}$, alors $\alpha^{-1} \in \mathcal{T}$.

Si \mathcal{T} est close, on définit la relation $\sim_{\mathcal{T}}$ sur C par $x \sim_{\mathcal{T}} y$ s'il existe $\alpha \in \mathcal{T}$ tel que $y = \alpha(x)$.

Question 42. Montrer que si \mathcal{T} est close, alors $\sim_{\mathcal{T}}$ est une relation d'équivalence.

Pour une combinaison x , on note :

- $[x]_{\mathcal{T}}$ la classe d'équivalence de x modulo la relation $\sim_{\mathcal{T}}$;
- $\text{repr}_{\mathcal{T}}(x)$ le plus petit élément de $[x]_{\mathcal{S}}$ pour l'ordre lexicographique (qu'on appelle *représentant de x modulo $\sim_{\mathcal{T}}$*).

On dit que x est \mathcal{T} -canonique si $x = \text{repr}_{\mathcal{T}}(x)$.

Question 43. Écrire une fonction `coups_canoniques` prenant en entrée un ensemble \mathcal{T} (supposé clos) de transformations, sous forme de liste, et renvoyant la liste des coups \mathcal{T} -canoniques. On considérera que l'ordre sur les entiers pour le type `combi` coïncide avec l'ordre lexicographique sur les combinaisons. On demande une complexité en $\mathcal{O}(|C| + k|\mathcal{T}|)$, où k est le nombre de coups \mathcal{T} -canoniques, et l'on justifiera cette complexité.

```
val coups_canoniques : transfo list -> combi list
```

Question 44. Donner, sans justification, l'ensemble des combinaisons \mathcal{S} -canoniques dans le cas $N = 4$ et $P = 6$.

Question 45. Montrer que pour la recherche d'un C -arbre de stratégie optimal, on peut se limiter aux arbres dont l'étiquette à la racine est \mathcal{S} -canonique.

IV.4 Extension aux coups suivants

Le gain apporté par cette réduction des coups à considérer est très important, mais il ne s'applique pour l'instant qu'au premier coup. On cherche à présent à étendre ce raisonnement aux coups suivants.

Pour $X \subseteq C$, on note $Fix(X)$ l'ensemble des transformations laissant X invariant :

$$\alpha \in Fix(X) \iff \alpha(X) = X$$

Question 46. Écrire une fonction `fix` prenant en entrée un ensemble X de combinaisons et renvoyant $Fix(X)$.

```
val fix : combi list -> transfo list
```

Question 47. Montrer que $Fix(X)$ est clos.

Question 48. En déduire une modification des fonctions EVAL1 et/ou EVAL2 permettant de restreindre les coups à considérer.

* *
*

Système de gestion de projets informatiques

Préliminaires

L'énoncé s'inspire d'un système de gestion de projets informatiques tel que *SourceForge* ou *GitHub*. Chaque partie s'intéresse à une problématique à résoudre, présente des objectifs concrets et amène une réflexion sur les moyens de les atteindre.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Les copies sont évaluées sur la précision, le soin et la clarté de la rédaction. On veillera toujours à rendre saillante la logique générale du code. Les pré-conditions des fonctions demandées n'ont pas besoin d'être vérifiées dans le code mais peuvent être énoncées sous forme de commentaires.

Dépendances. Ce sujet contient six parties. Les différentes parties et un grand nombre de leurs questions sont largement indépendants. Il est possible d'aborder les différentes parties dans un ordre quelconque en groupant les questions d'une même partie et en indiquant clairement quelle question est répondue.

Langages. Les parties I et V sont à traiter en Python. La partie II est à traiter en SQL. La partie III est à traiter en HTML/CCS et JavaScript. La partie IV est à traiter en C.

Partie I. Mise en place de chaînes de traitement d'intégration et de déploiement continu (*chaînes CI/CD*)

Une chaîne d'intégration et de déploiement continu (*chaîne CI/CD*) permet de gérer le processus de production de nouvelles versions d'un produit logiciel. La chaîne assure la bonne mise en relation entre les activités des équipes de développement et celles des équipes opérationnelles. Un système d'intégration et de déploiement continu permet la définition et l'exécution *automatique* de chaînes CI/CD.

Une chaîne CI/CD est composée de plusieurs *travaux* où chaque travail définit une suite d'*actions*. Les travaux d'une chaîne sont tous distincts. Ils peuvent être indépendants et s'exécuter en parallèle, ou avoir des dépendances qui imposent qu'un travail attende la terminaison d'un autre. Au sein d'un travail, les actions s'exécutent en séquence l'une après l'autre, chaque action ne débutant qu'après la fin de l'action qui la précède. Un exemple de chaîne CI/CD pour une application informatique est une chaîne qui a plusieurs travaux de construction indépendants (création de différentes versions de l'application), ainsi qu'un travail de packaging qui dépend de ces derniers. Chaque travail de construction est typiquement composé d'une suite d'actions qui mélange compilation et tests.

L'exécution d'une chaîne CI/CD, qui se déclenche suite à un événement ou de manière périodique, consiste en l'exécution des travaux qui la composent.

Question 1. Quel est l'intérêt de la mise en place des chaînes CI/CD ? La réponse s'appuiera sur un minimum de deux arguments.

Nous représentons les entités constitutives d'une chaîne CI/CD par trois classes Python : la classe `CICDPipeline`, la classe `Work` et la classe `Action`, qui obéissent aux relations décrites ci-dessus. Chaque instance de l'une des trois classes dispose d'un attribut entier `id` distinct (un identifiant) dont la valeur est fournie par l'usager à la création. Chacune des trois classes est munie d'une méthode `run`. La méthode `run` définit les traitements que la chaîne, le travail ou l'action doit effectuer.

Question 2. Proposer un diagramme de classes contenant les trois classes `CICDPipeline`, `Work` et `Action`, ainsi que d'éventuelles classes supplémentaires permettant de factoriser certaines fonctionnalités le cas échéant.

Question 3. Quels sont les concepts de la programmation orientée-objet qui permettent à la méthode `run` de s'appliquer à des instances relevant de trois classes différentes ?

Question 4. Écrire une implémentation de la classe `Work` en Python.

Travaux indépendants.

Dans la question 5, nous considérons le cas simplifié des chaînes CI/CD composées de travaux *indépendants*. Une telle chaîne lance l'exécution de ses travaux en parallèle à l'aide de fils d'exécution (ou *threads*) : elle crée un fil d'exécution par travail. L'exécution de la chaîne se termine quand tous les travaux, et donc les fils d'exécution correspondants, se sont terminés.

Question 5. Écrire en Python une première version de la classe `CICDPipeline`. Des rappels sur la manipulation de fils d'exécution en Python sont donnés en Annexe A.

Mise en place des dépendances.

Dans les questions 6 à 13, nous considérons le cas général d'une chaîne où les travaux peuvent avoir des dépendances. L'exécution d'un travail qui a des dépendances ne peut démarrer avant la terminaison de l'exécution de tous les travaux dont il dépend.

Question 6. Modifier le code Python de la classe `Work` écrite à la question 4 en adjoignant un troisième argument au constructeur : une liste `dependencies` des travaux dont dépend le travail construit et qui est passée comme attribut à l'objet créé.

Le bon fonctionnement de l'exécution d'une chaîne repose sur la synchronisation des fils d'exécution utilisés pour exécuter les travaux. Dans ce but, une chaîne CI/CD associe un sémaphore distinct à chacun de ses travaux. Quand un travail t_1 doit attendre la terminaison d'un autre travail t_2 , il se bloque sur le sémaphore associé à t_2 . Il est débloqué par t_2 à la fin de l'exécution de t_2 .

Question 7. Modifier le code Python de la classe `CICDPipeline` écrite à la question 5 afin d'y inclure une gestion des sémaphores : définir un attribut, procéder à son initialisation dans le constructeur et fournir un accesseur d'entête `get_semaphore(work_id)`. Des rappels sur la synchronisation de fils d'exécution en Python sont donnés en Annexe B.

Pour que tous les travaux qui attendent un certain travail puissent être débloqués, il est nécessaire de connaître leur nombre. Comme pour les sémaphores, l'information sur le nombre de travaux que chaque travail doit débloquer est à gérer au niveau de la chaîne CI/CD correspondante.

Question 8. Dans la continuité de la question 7, modifier la classe `CICDPipeline` pour pouvoir gérer l'association entre un travail de la chaîne et le nombre de travaux qui en dépendent. Rajouter également un accesseur d'entête `get_waiting_for(work_id)`.

Question 9. Pour se synchroniser avec les autres travaux, un travail a besoin d'avoir accès aux informations (sémaphores et compteurs) de sa chaîne CI/CD. Modifier la classe `Work` pour y intégrer une référence vers sa chaîne CI/CD.

Question 10. Décrire, en langue française, une solution pour la synchronisation de travaux avec dépendances. La mettre en œuvre en réécrivant la méthode `run` de la classe `Work`.

Interblocages.

Question 11. Les dépendances entre travaux étant introduites exclusivement par l'intermédiaire du constructeur de la classe `Work`, selon la spécification de la question 6, démontrer que l'exécution de la méthode `run` de la classe `CICDPipeline` ne connaît pas d'interblocages.

Lors de la définition de grandes chaînes CI/CD avec de nombreux travaux, il peut être commode de rajouter des dépendances au fur et à mesure de la définition de différents travaux.

Question 12. Enrichir le code Python de la classe `Work` écrite à la question 6 en adjointant un transformateur d'entête `set_dependency(prec_work)` de sorte que la commande `work.set_dependency(prec_work)` ajoute le travail `prec_work` comme dépendance devant précéder le travail `work`.

L'utilisation de la méthode `set_dependency` risque d'introduire des interblocages.

Question 13. Citer un problème d'algorithmique classique permettant de modéliser la détection d'interblocages. Nommer un algorithme classique le résolvant. Écrire en langage Python un script permettant de détecter si une chaîne fait l'objet d'un interblocage.

Question 14. Proposer une modification du code écrit à la question 12 pour informer l'usager de l'introduction d'un interblocage dès que possible.

Partie II. Modélisation et stockage persistant

Les projets informatiques. Les chaînes CI/CD sont définies dans le cadre de *projets informatiques*. Un projet informatique est un projet de développement et de gestion d'un produit logiciel. Un projet dispose d'un identifiant unique au sein du système, ainsi que d'un nom permettant une manipulation plus facile par les équipes de développement et d'intégration. Un projet peut disposer de plusieurs chaînes CI/CD.

Les utilisateurs. Les *utilisateurs* qui travaillent sur un projet informatique sont dotés d'un *rôle*. Le rôle délimite les opérations qu'un utilisateur peut effectuer et inclut, entre autres, les rôles de *propriétaire*, *développeur* et *invité*. Un utilisateur peut être impliqué dans plusieurs projets avec des rôles distincts.

Les opérations de lecture/écriture. Les utilisateurs peuvent effectuer deux types d'opérations sur les projets : des consultations de projet (*opération de lecture*) et des modifications (*opération d'écriture*). Chaque opération d'un utilisateur sur un projet, qu'elle soit de lecture ou d'écriture, est consignée de manière persistante.

Question 15. Élaborer une modélisation relationnelle des données du système en faisant apparaître les tables User, Project, CICD_Pipeline, Role et une table de jointure User_Project.

Question 16. Écrire une requête SQL qui permet de créer la table contenant les données des projets informatiques.

Question 17. Écrire une requête SQL qui permet d'identifier les cinq projets ayant le plus d'utilisateurs.

Question 18. Écrire une requête SQL qui permet de trier les utilisateurs d'un projet MyProject selon le nombre de leurs contributions (modifications).

Partie III. Application WEB

Dans cette partie nous nous intéressons à l'application web qui permet de travailler avec les projets informatiques.

L'application web doit permettre aux utilisateurs d'accéder aux informations sur leurs projets et leur activité. Chaque utilisateur doit disposer de sa propre adresse pour consulter les informations le concernant. Toutefois, les informations concernant un projet informatique en particulier doivent être accessibles à la même adresse pour tous les utilisateurs.

Question 19. Proposer une structuration d'URL pour l'application web qui permette de consulter les informations d'un projet informatique P pour un utilisateur U.

The screenshot shows a web application interface. On the left, there is a sidebar with two menu items: "Mes projets" and "Mon activité". The main content area has a header "Claude Martin / APP-AGREG-2025". Below the header, the title "Projet : APP-AGREG-2025" is displayed. Underneath the title, the ID "ID : 204567" and the last modification date "Dernière modification : 10/01/2025" are shown. A section titled "Utilisateurs" contains a table listing users and their roles:

Nom	Rôle
Jeanne Sauvignon	administrateur
Denis Pelletier	développeur
Claude Martin	développeur
Alain Da Vinci	invité

Below the user table, there is a section titled "Activité" which contains another table showing consultation and modification counts:

Consultations	Modifications
1356	2756

FIGURE 1 – Page web visualisée par l'utilisatrice *Claude Martin* participant au projet informatique APP-AGREG-2025. La page donne l'identifiant du projet et la dernière date de modification, avant de lister les utilisateurs impliqués et le nombre de consultations et de modifications. À gauche, un menu permet d'accéder aux informations sur les projets de *Claude Martin* et sur l'historique de ses contributions.

Question 20. Donner le code HTML/CSS statique qui correspond à l'esquisse d'IHM décrite dans la Figure 1.

Question 21. Enrichir le code HTML/CSS précédent pour proposer une solution supportant la mise-à-jour dynamique, i.e. sans rechargement de la page, du nom du projet, du nombre de consultations et du nombre de modifications.

Question 22. Écrire en JavaScript un script qui permet à l'application cliente de recevoir les informations de mise-à-jour du nombre de consultations et du nombre de modifications.

Question 23. Quels sont les couches réseau et les protocoles impliqués entre la partie cliente et la partie serveur de l'application web lorsqu'un utilisateur veut consulter la page à l'adresse <https://agregsystem.fr> ?

Partie IV. Gestion de versions

Dans cette partie nous allons nous pencher sur un des aspects des plus importants d'un système de gestion de projets informatiques : la gestion des versions. La gestion des versions concerne l'ensemble des fichiers qui définissent un produit logiciel : fichiers source, scripts, fichiers de configuration, etc. Ces fichiers sont typiquement organisés de manière arborescente, avec plusieurs répertoires et sous-répertoires.

Pour gérer les différentes versions, le système utilise deux types d'objets : les *objets binaires* (*blobs*) et les *arbres*. Les *blobs* représentent les fichiers, alors que les *arbres* représentent les répertoires. Pour chaque objet, le système calcule une clé unique d'identification obtenue par hachage du contenu de l'objet.

Nous supposons pour la suite que le système fournit les définitions suivantes, écrites en langage C :

```
// types d'objets de gestion de versions
typedef enum {BLOB_TYPE, TREE_TYPE} object_t;

// structure décrivant un objet de gestion de versions
struct object_info {
    object_t type;
    unsigned long size;
    hash_t oid;
};

// fonction de hachage de contenu
hash_t hash(void* content, unsigned long size);

// structure de blob
struct blob {
    struct object_info header;
    void* content;
};
```

La structure `object_info` est utilisée aussi bien pour les *blobs*, que pour les *arbres*. Elle contient trois champs : le type de l'objet, donné à l'aide du type énuméré `object_t`, la taille en octets du contenu de l'objet, ainsi qu'un identifiant, `oid`, qui correspond au haché de l'objet. L'identifiant est obtenu à l'aide de la fonction `hash`, que nous supposons fournie.

La structure pour représenter un *blob* contient, en plus des informations de type, de taille et d'identifiant, le contenu du fichier correspondant, `content`. Le champ `content` est par conséquent une suite d'octets de taille variable.

Question 24. Écrire en C une fonction

```
struct blob* init_blob(void* content, unsigned long size);
```

qui initialise une structure `struct blob` à partir du contenu et de la taille d'un fichier passés en argument. La fonction renvoie la structure initialisée qui doit être allouée par la fonction. En cas d'échec, la fonction renvoie `NULL`.

Des fonctions C qui peuvent vous être utiles pour cette question sont données en Annexe C.

Arbres. Pour les *arbres*, le système de gestion de versions s'inspire de la logique des répertoires UNIX. Dans UNIX, le contenu d'un répertoire est une liste comprenant des informations sur des fichiers et des sous-répertoires. Dans le système de gestion de versions, le contenu d'un *arbre* est une liste comprenant des informations sur des blobs et des sous-arbres.

Pour la suite, nous supposons que le contenu d'un arbre est représenté à l'aide d'un tableau de taille prédéfinie. Chaque élément de ce tableau correspond à un objet distinct de l'arbre et contient deux parties : la structure `struct object_info`, qui décrit l'objet, et le nom du fichier ou du répertoire représenté par cet objet.

Question 25. En suivant la logique de la structure `struct blob` et les indications ci-dessus, déclarer en C une structure de données `struct tree` pour représenter un arbre. Donner le code C d'une fonction `struct tree* empty_tree()` qui crée un arbre vide. Bien spécifier comment un objet non défini est représenté dans le contenu de l'arbre.

Question 26. Donner le code C des deux fonctions suivantes qui permettent de changer le contenu d'un arbre

```
bool tree_add_object(struct tree* tree, char* name, struct object_info* header);
bool tree_remove_object(struct tree* tree, hash_t oid);
```

Question 27. Écrire en C une fonction

```
bool diff(void* o1, void* o2);
```

qui vérifie si les deux objets `o1` et `o2` passés en paramètre sont identiques.

Chemins. Dans UNIX, l'emplacement d'un fichier dans l'arborescence de fichiers est défini par son *chemin* depuis la racine. UNIX définit des conventions pour représenter un tel chemin sous forme d'une chaîne de caractères. Ainsi, la racine de l'arborescence de fichiers est notée `/` et le chemin `/code/test.c` correspond au fichier `test.c` qui se trouve dans le répertoire `code` qui est lui-même un sous-répertoire de la racine.

Le système de gestion de versions sauvegarde tous les objets de blob ou d'arbre dans une table associative entre l'objet et son identifiant. Dans la suite, nous supposerons les fonctions de manipulation de cette table fournies et déclarées comme suit :

```
bool put_hashtable(hash_t oid, void* o);
void* get_hashtable(hash_t oid);
```

La fonction `put_hashtable` lie un identifiant à son objet dont il est le haché (*blob* défini par `struct blob` ou *arbre* défini par `struct tree`) correspondant. La fonction `get_hashtable` permet d'avoir accès à l'objet identifié par un haché donné.

Nous supposons également fournies les fonctions suivantes :

```
struct tree* get_root();  
char** parse_path(char* path);
```

La fonction `get_root` récupère l'objet arbre correspondant au répertoire racine `(/)`. La fonction `parse_path` renvoie la liste des chaînes de caractères qui composent le chemin `path`, terminée par `NULL`. Ainsi `parse_path("/code/test.c")` renvoie `{"code", "test.c", NULL}`.

Question 28. Écrire en C une fonction `object_t get_type(char* path)` qui permet de connaître le type de l'objet (*blob* ou *arbre*) accessible à l'emplacement défini par le chemin `path` passé en paramètre. Nous supposons que le paramètre `path` est une chaîne de caractères non vide qui respecte les conventions de nommage des chemins UNIX.

Question 29. Déclarer en C une structure de données `struct commit` qui permet de contenir les informations relatives à la définition d'une nouvelle version d'un arbre : l'identifiant de l'arbre en question, la liste des commits précédant le commit en question, le nom de l'utilisateur qui a créé le commit et un commentaire.

Partie V. Performances à l'exécution

Dans cette partie, nous nous intéressons à la durée d'exécution d'une chaîne CI/CD. Nous supposons que toute action d'un travail CI/CD s'exécute en une unité de temps.

Question 30. Enrichir le code Python de la classe `Work`, introduite à la question 4, afin que la commande `work.duration()` renvoie le temps d'exécution du travail `work`.

Le système de gestion de projets informatique exécute les chaînes CI/CD sur des entités dédiées appelées *exécutants*. Les exécutants fournissent les ressources de calcul et de stockage nécessaires aux traitements CI/CD, et permettent une éventuelle exécution parallèle. Pour la suite, nous supposons que les dépendances d'une chaîne CI/CD sont stabilisées et ne peuvent plus changer.

Question 31. Enrichir le code Python de la classe `CICDPipeline`, introduite à la question 5, afin que la commande `pipeline.sequential_duration()` renvoie le temps d'exécution de la chaîne `pipeline` dans le cas où un seul fil d'exécution peut être exécuté à la fois.

Question 32. Enrichir le code Python de la classe `CICDPipeline`, introduite à la question 5, afin que la commande `pipeline.parallel_duration()` renvoie le temps d'exécution de la chaîne `pipeline` dans le cas où les fils d'exécution peuvent être de nombre illimité. On décrira et on justifiera avec soin l'algorithme employé.

Partie VI. Autour du CI/CD

Question 33. En quoi consiste la phase d'intégration d'un projet informatique ? Quels sont ses liens avec les tests et la gestion de versions ?

Question 34. Le déploiement correspond au processus de transfert du produit logiciel modifié vers l'environnement de production. Citer trois risques majeurs pour un déploiement.

Question 35. Que signifie le fait que l'intégration et le déploiement soient *continus* ? Quels en sont les bénéfices ?

* *
*

ANNEXE A : **threading** — Parallélisation à base de Threads

Extrait de la documentation Python 3

Code source : Lib/threading.py

Objets **Threads**

La classe Thread représente une activité exécutée dans un fil d'exécution distinct. Il existe deux manières de spécifier l'activité : en passant un objet appelable au constructeur ou en redéfinissant la méthode `run()` dans une sous-classe. Aucune autre méthode (à l'exception du constructeur) ne doit remplacée dans une sous-classe. En d'autres termes, remplacez uniquement les méthodes `__init__()` et `run()` de cette classe.

Une fois qu'un objet fil d'exécution est créé, son activité doit être lancée en appelant la méthode `start()` du fil. Ceci invoque la méthode `run()` dans un fil d'exécution séparé.

D'autres fils d'exécution peuvent appeler la méthode `join()` d'un fil. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode `join()` est appelée soit terminé.

Un fil d'exécution a un nom. Le nom peut être passé au constructeur, et lu ou modifié via l'attribut `name`.

```
class threading.Thread(group=None, target=None, name=None, args=(),  
                      kwargs=*, daemon=None)
```

Les arguments sont :

`group` devrait être `None`;

`target` est l'objet appelable qui doit être invoqué par la méthode `run()`. La valeur par défaut est `None`, ce qui signifie que rien n'est appelé.

`name` est le nom du fil d'exécution. Par défaut, un nom unique est construit de la forme "Thread-N" où N est un entier. Si `target` est défini, le nom est de la forme "Thread-N (`target.__name__`)"

`args` est une liste ou un tuple d'arguments pour l'invocation de `target`. La valeur par défaut est `()`.

`kwargs` est un dictionnaire d'arguments nommés pour l'invocation de l'objet appelable. La valeur par défaut est `.`.

Si la sous-classe réimplémente le constructeur, elle doit s'assurer d'appeler le constructeur de la classe de base (`Thread.__init__()`) avant de faire autre chose au fil d'exécution.

start()

Lance l'activité du fil d'exécution.

Elle ne doit être appelée qu'une fois par objet de fil. Elle fait en sorte que la méthode `run()` de l'objet soit invoquée dans un fil d'exécution.

run()

Méthode représentant l'activité du fil d'exécution.

Exemple :

```
>>> from threading import Thread  
>>> t = Thread(target=print, args=[1])  
>>> t.run()  
1  
>>> t = Thread(target=print, args=(1,))  
>>> t.run()  
1
```

join(timeout=None)

Attend que le fil d'exécution se termine. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode `join()` est appelée se termine – soit normalement, soit par une exception non gérée – ou jusqu'à ce que le délai optionnel `timeout` soit atteint.

ANNEXE B : **threading** — Synchronisation de Threads en Python

Extrait de la documentation Python 3

Code source : Lib/threading.py

Verrous

Un verrou primitif n'appartient pas à un fil d'exécution lorsqu'il est verrouillé. En Python, c'est actuellement la méthode de synchronisation la plus bas-niveau qui soit disponible, implémentée directement par le module d'extension `_thread`.

Un verrou primitif est soit « verrouillé » soit « déverrouillé ». Il est créé dans un état déverrouillé. Il a deux méthodes, `acquire()` et `release()`. Lorsque l'état est déverrouillé, `acquire()` verrouille et se termine immédiatement. Lorsque l'état est verrouillé, `acquire()` bloque jusqu'à ce qu'un appel à `release()` provenant d'un autre fil d'exécution le déverrouille. À ce moment `acquire()` le verrouille à nouveau et rend la main. La méthode `release()` ne doit être appelée que si le verrou est verrouillé, elle le déverrouille alors et se termine immédiatement. Déverrouiller un verrou qui n'est pas verrouillé provoque une `RuntimeError`.

Lorsque plusieurs threads sont bloqués dans `acquire()` en attendant que l'état passe à déverrouillé, un seul thread continue lorsqu'un appel `release()` réinitialise l'état à déverrouillé ; lequel des threads en attente se débloque n'est pas défini et peut varier selon les implémentations.

`class threading.Lock`

La classe implémentant des verrous primitifs. Une fois qu'un fil d'exécution a acquis un verrou, tout appel consécutif pour acquérir le verrou est bloquant, jusqu'à libération du verrou. Un verrou peut être libéré par n'importe quel fil d'exécution.

acquire(blocking=True, timeout=-1)

Acquierte un verrou, bloquant ou non bloquant.

release()

Libérer un verrou. Peut être appelé par n'importe quel fil d'exécution, non seulement par celui qui a acquis le verrou.

Sémaphores

```
class threading.Semaphore(value=1)
```

Cette classe implémente les sémaphores.

Un sémaphore gère un compteur interne qui est décrémenté à chaque appel `acquire()` et incrémenté à chaque appel `release()`. Le compteur ne peut jamais descendre en dessous de zéro ; lorsque `acquire()` trouve qu'il est nul, il bloque, attendant qu'un autre thread appelle `release()`.

```
class threading.Semaphore(value=1)
```

```
    acquire(blocking=True, timeout=None)
```

Acquiert un sémaphore.

```
    release(n=1)
```

Libérer un sémaphore, incrémenté le compteur interne de un.

ANNEXE C : Fonctions utiles en C

NAME

memcpy - copy memory area

SYNOPSIS

```
#include <string.h>

void *
memcpy(void *restrict dst, const void *restrict src, size_t n);
```

DESCRIPTION

The **memcpy()** function copies n bytes from memory area src to memory area dst. If dst and src overlap, behavior is undefined. Applications in which dst and src might overlap should use **memmove(3)** instead.

RETURN VALUES

The **memcpy()** function returns the original value of dst.

NAME

memmove - copy byte string

SYNOPSIS

```
#include <string.h>

void *
memmove(void *dst, const void *src, size_t len);
```

Transducteurs finis

Les transducteurs finis sont des machines théoriques similaires aux automates finis permettant de transformer des mots donnés en entrée. Ils sont utilisés pour l'analyse morphologique et phonologique dans le domaine de la linguistique, mais peuvent également avoir des applications en analyse syntaxique. Certains cas particuliers de transducteurs peuvent également avoir des applications dans des machines simples, tels que des distributeurs automatiques.

Le sujet est découpé en trois parties. La première présente les transducteurs séquentiels et contient quelques exemples et implémentations. La deuxième généralise le concept en introduisant les transducteurs sous-séquentiels. Enfin, la troisième et dernière partie établit deux théorèmes qui caractérisent les fonctions séquentielles et sous-séquentielles et leurs démonstrations.

Préliminaires

Notations Si Σ est un alphabet, on note Σ^* l'ensemble des mots sur Σ . Pour $u \in \Sigma^*$, on note $|u|$ la taille de u . L'unique mot de taille 0 sera noté ε , indépendamment de l'alphabet. Pour n un entier naturel, on note :

- Σ^n l'ensemble des mots de taille **exactement** n , défini par induction par $\Sigma^0 = \{\varepsilon\}$ et $\Sigma^{n+1} = \Sigma\Sigma^n$;
- $\Sigma^{\leq n}$ l'ensemble des mots de taille **au plus** n , c'est-à-dire $\Sigma^{\leq n} = \{\varepsilon\} \cup \Sigma^1 \cup \Sigma^2 \cup \dots \cup \Sigma^n$.

On définit un **automate fini déterministe complet** comme un quintuplet $A = (Q, \Sigma, \delta, q_0, F)$ où :

- Q est un ensemble fini dont les éléments sont appelés **états** ;
- Σ est un alphabet appelé **alphabet d'entrée** ;
- δ est une fonction de $Q \times \Sigma$ dans Q appelée **fonction de transition** ;
- $q_0 \in Q$ est appelé **état initial** ;
- $F \subseteq Q$ est l'ensemble des **état finaux**.

On étend inductivement l'ensemble de définition de δ à $Q \times \Sigma^*$ par :

- $\delta(q, \varepsilon) = q$;
- si $u \in \Sigma^*$ et $a \in \Sigma$, alors $\delta(q, ua) = \delta(\delta(q, u), a)$.

On dit qu'un mot $u \in \Sigma^*$ est **reconnu** par l'automate A si et seulement si $\delta(q_0, u) \in F$. On note $L(A)$ l'ensemble des mots reconnus par A . Un langage L est dit **rationnel** (ou régulier) s'il existe un automate A tel que $L = L(A)$.

Dépendances. Ce sujet contient plusieurs parties. Chaque partie utilise des définitions et des résultats des parties précédentes. Sauf mention explicite du contraire, les questions restent néanmoins indépendantes, au sens où toute question peut être traitée en admettant les résultats énoncés dans les questions précédentes.



Attendus. Les questions de programmation doivent être traitées en langage OCaml. On pourra utiliser toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`). L'utilisation d'autres modules est interdite.

Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Partie I. Transducteurs séquentiels

Informellement, un transducteur séquentiel est un automate fini déterministe complet qui écrit un mot en sortie lors de la lecture d'un mot. Dans la définition suivante, il n'y a pas de notion d'état final.

On définit un **transducteur séquentiel** (ou simplement transducteur quand il n'y a pas d'ambiguité) comme un sextuplet $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ tel que :

- Q est un ensemble fini dont les éléments sont appelés **états** ;
- Σ est un alphabet appelé **alphabet d'entrée** ;
- Γ est un alphabet appelé **alphabet de sortie** ;
- δ est une fonction de $Q \times \Sigma$ dans Q appelée **fonction de transition** ;
- λ est une fonction de $Q \times \Sigma$ dans Γ^* appelée **fonction de sortie** ;
- $q_0 \in Q$ est appelé **état initial**.

Comme pour un automate, on étend l'ensemble de définition de δ à $Q \times \Sigma^*$. On fait de même pour λ :

- $\lambda(q, \varepsilon) = \varepsilon$;
- si $u \in \Sigma^*$ et $a \in \Sigma$, alors $\lambda(q, ua) = \lambda(q, u)\lambda(\delta(q, u), a)$.

On appelle **transition de T** et on note $q \xrightarrow{a|v} q'$ un quadruplet (q, a, q', v) tel que $(q, a) \in Q \times \Sigma$, $\delta(q, a) = q'$ et $\lambda(q, a) = v$. L'interprétation d'une telle transition est « si, depuis l'état q , on lit la lettre a en entrée, alors on va vers l'état q' , et on écrit le mot v en sortie ».

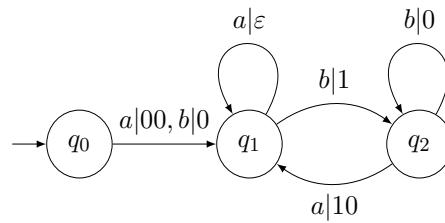
Un **calcul dans T** est une suite de transitions de la forme $p_0 \xrightarrow{a_0|v_0} p_1 \xrightarrow{a_1|v_1} \dots \xrightarrow{a_{k-1}|v_{k-1}} p_k$ telle que $p_i \xrightarrow{a_i|v_i} p_{i+1}$ est une transition de T pour $i \in \llbracket 0, k-1 \rrbracket$. On dit qu'un tel calcul est d'origine p_0 , d'extrémité p_k , d'entrée $a_0a_1\dots a_{k-1}$ et de sortie $v_0v_1\dots v_{k-1}$.

Enfin, on définit la **fonction associée** à T par $\varphi_T : \Sigma^* \longrightarrow \Gamma^*$. Autrement dit,

$$\begin{array}{ccc} & a_0 | v_0 & \\ u & \longmapsto & \lambda(q_0, u) \\ & a_1 | v_1 & \\ & \dots & \\ & a_{k-1} | v_{k-1} & \end{array}$$

$\varphi_T(u) = v$ si un calcul d'entrée u est de sortie v . On dit qu'une fonction $f : \Sigma^* \rightarrow \Gamma^*$ est **séquentielle** s'il existe un transducteur séquentiel T tel que $f = \varphi_T$.

Schématiquement, on représente un transducteur séquentiel comme dans la figure 1, qui représente le transducteur $T_0 = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ où $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $\Gamma = \{0, 1\}$ et δ et λ sont définies par les tableaux en figure 2. Sur ce transducteur, on a $\varphi_{T_0}(aabba) = 001010$ et $\varphi_{T_0}(baaab) = 01$.

FIGURE 1 – Le transducteur séquentiel T_0

δ	a	b
q_0	q_1	q_1
q_1	q_1	q_2
q_2	q_1	q_2

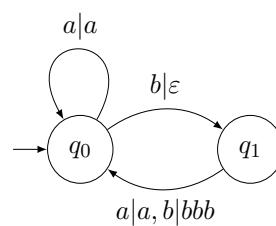
λ	a	b
q_0	00	0
q_1	ϵ	1
q_2	10	0

FIGURE 2 – La table de transition et la table de sortie de T_0

I.1 Premiers exemples

Question 1 Pour T_0 le transducteur défini par la figure 1, déterminer $\varphi_{T_0}(bbbbaaa)$. Déterminer un antécédent par φ_{T_0} de 01101.

Question 2 On considère le transducteur T_1 de la figure 3. Pour un mot $u \in \{a, b\}^*$ quelconque, décrire en français comment est construit $\varphi_{T_1}(u)$.

FIGURE 3 – Le transducteur séquentiel T_1

Question 3 Représenter graphiquement un transducteur séquentiel T sur $\Sigma = \Gamma = \{a, b\}$ dont la fonction séquentielle remplace toute séquence de a consécutifs en un seul a et ne modifie pas les séquences de b consécutifs. Justifier brièvement la correction de la construction.

Par exemple, on doit avoir $\varphi_T(aaaaabaaaabbba) = ababbba$ et $\varphi_T(babba) = babba$.

Pour $b \in \mathbb{N} \setminus \{0, 1\}$ et $n \in \mathbb{N}^*$, une représentation en base b de n est une suite notée $[a_0 a_1 \dots a_{m-1}]_b$ telle que $m \in \mathbb{N}^*$, $\forall i \in \llbracket 0, m-1 \rrbracket$, $a_i \in \llbracket 0, b-1 \rrbracket$ et :

$$n = \sum_{i=0}^{m-1} a_i b^i$$

Si de plus on impose $a_{m-1} \neq 0$, alors cette suite est unique.

Par exemple, la représentation en base 4 de 237 est $[1323]_4$ car $237 = 1 + 3 \times 4 + 2 \times 16 + 3 \times 64$. Notons que nous représentons ici les chiffres de poids faible à gauche.

Question 4 Représenter graphiquement un transducteur sur $\Sigma = \{0, 1, 2, 3\}$ et $\Gamma = \{0, 1\}$ qui transforme un mot u représentant une écriture de $n \in \mathbb{N}$ en base 4 en un mot v représentant une écriture de n en base 2. Justifier brièvement la correction de la construction.

Question 5 Représenter graphiquement un transducteur sur $\Sigma = \{0, 1\}$ et $\Gamma = \{0, 1, 2, 3\}$ qui transforme un mot u représentant une écriture de $n \in \mathbb{N}$ en base 2, tel que u est de taille paire (terminant éventuellement par un 0), en un mot v représentant une écriture en base 4 de n . Justifier brièvement la correction de la construction.

Par exemple, l'image de 011110 doit être 132, car $[011110]_2 = [132]_4$.

I.2 Implémentation

On représente un transducteur séquentiel en OCaml en utilisant les types suivants :

```
type lettre = int
type mot = lettre list
type etat = int
type transducteur = (etat * mot) array array
```

On choisit de représenter un alphabet Σ par $\Sigma = \{0, 1, \dots, |\Sigma| - 1\}$ (de même pour Γ). Un mot de Σ^* sera représenté par une liste d'entiers. Par exemple, l'objet $[0; 1; 1; 2; 1; 0]$ représente le mot 011210 sur l'alphabet $\{0, 1, 2\}$.

Lorsque le transducteur séquentiel $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ est représenté par la matrice t , dont la première dimension est n et la deuxième dimension est p , on a :

- $|Q| = n$;
- $|\Sigma| = p$;
- $q_0 = 0$;
- pour tous $q \in Q$ et $a \in \Sigma$, $t.(q).(a)$ est un couple (q', v) où q' représente $q' = \delta(q, a)$ et v représente $v = \lambda(q, a)$.

Par exemple, le transducteur séquentiel T_0 de la figure 1 peut être représenté par le morceau de code suivant :

```
let t0 = [| [|(1, [0; 0]); (1, [0])|];
           [|(1, []); (2, [1])|];
           [|(1, [1; 0]); (2, [0])|]|]
```

On a assimilé ici la lettre a à 0 et la lettre b à 1.

Question 6 Écrire une fonction `calcul (t : transducteur) (u : mot) : mot` telle que si t est un transducteur séquentiel représentant $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ et u est un mot $u \in \Sigma^*$, alors `calcul t u` renvoie le mot v représentant $\lambda(q_0, u)$.

À T fixé, cette fonction devra avoir une complexité linéaire en $|u|$ et on demande de le justifier.

Question 7 Écrire une fonction `liste_mots (p : int) (m : int) : mot list` qui prend en argument un entier p et un entier m et renvoie une liste contenant tous les mots de taille m sur l'alphabet $\{0, 1, \dots, p - 1\}$ dans un ordre arbitraire.

Question 8 En déduire une fonction `antecedent (t : transducteur) (m : int) (v : mot option) : mot` telle que si t est un transducteur séquentiel représentant $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$, m est un entier naturel et v est un mot $v \in \Gamma^*$, alors `transducteur t m v` renvoie `Some u` où u est un mot le plus court possible vérifiant $|u| \leq m$ et $\varphi_T(u) = v$ s'il en existe un, et `None` sinon.

On n'impose aucune restriction sur la complexité temporelle de cette fonction.

I.3 Morphismes de mots

Soit $\Phi : \Sigma^* \rightarrow \Gamma^*$. On dit que Φ est un **morphisme de mots** si et seulement si pour tous mots u et v de Σ^* , $\Phi(uv) = \Phi(u)\Phi(v)$.

Question 9 Soit $\Phi : \Sigma^* \rightarrow \Gamma^*$ un morphisme de mots. Déterminer en justifiant la valeur de $\Phi(\varepsilon)$.

Question 10 Soit Φ et Ψ deux morphismes de mots de Σ^* dans Γ^* . Montrer que si pour tout $a \in \Sigma$, $\Phi(a) = \Psi(a)$, alors $\Phi = \Psi$.

Question 11 Soit $\Phi : \Sigma^* \rightarrow \Gamma^*$ un morphisme de mots. Montrer que Φ est une fonction séquentielle.

Question 12 Soit T un transducteur séquentiel dont tous les états sont **accessibles** (c'est-à-dire que pour tout état $q \in Q$, il existe $u \in \Sigma^*$ tel que $\delta(q_0, u) = q$). Montrer qu'il y a équivalence entre :

1. φ_T est un morphisme de mots
2. pour tout $q \in Q$ et $a \in \Sigma$, $\lambda(q, a) = \lambda(q_0, a)$.

I.4 Machines de Mealy et de Moore

Un transducteur séquentiel $(Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ est appelé **machine de Mealy** si pour toute transition $q \xrightarrow{a|v} q'$, on a $|v| = 1$. C'est-à-dire que pour chaque lettre lue en entrée, la machine écrit une lettre en sortie. La fonction de sortie peut être vue comme définie de $Q \times \Sigma$ dans Γ .

Une machine de Mealy est appelée **machine de Moore** si de plus pour toute paire de transitions $q_1 \xrightarrow{a_1|b_1} q$ et $q_2 \xrightarrow{a_2|b_2} q$, on a $b_1 = b_2$. Autrement dit, la lettre écrite en sortie ne dépend que de l'état dans lequel on arrive lors de l'exécution d'une transition.

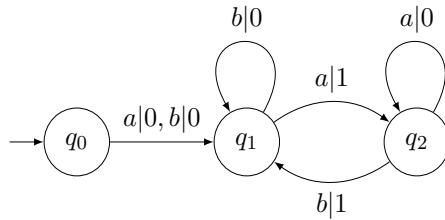


FIGURE 4 – Une machine de Mealy

On rappelle qu'une représentation en base b commence par les chiffres de poids faible.

Question 13 Représenter graphiquement une machine de Mealy sur $\Sigma = \Gamma = \{0, 1\}$ qui transforme un mot u représentant l'écriture de $x \in \mathbb{N}$ en base 2 en le mot v représentant l'écriture en base 2 de $2x$ modulo $2^{|u|}$. Justifier brièvement la correction de la construction.

Question 14 Représenter graphiquement une machine de Moore sur $\Sigma = \Gamma = \{0, 1\}$ qui transforme un mot u représentant l'écriture de $x \in \mathbb{N}$ en base 2 en le mot v représentant l'écriture en base 2 de $x + 1$ modulo $2^{|u|}$. Justifier brièvement la correction de la construction.

Question 15 Écrire une fonction `est_mealy` ($t : \text{transducteur}$) : `bool` qui détermine si un transducteur est une machine de Mealy ou non. Déterminer sa complexité temporelle dans le pire cas.

Question 16 Soit M une machine de Mealy. Montrer qu'il existe une machine de Moore M' telle que $\varphi_{M'} = \varphi_M$.

Une machine de Mealy est appelée **machine de Moore alternative** si pour toute paire de transitions $q \xrightarrow{a_1|b_1} q_1$ et $q \xrightarrow{a_2|b_2} q_2$, on a $b_1 = b_2$. Autrement dit, la lettre écrite en sortie ne dépend que de l'état duquel **on part** lors de l'exécution d'une transition. La définition est similaire à celle d'une machine de Moore, mais c'est cette fois-ci l'état de départ qui détermine la lettre en sortie.

Question 17 Montrer qu'il existe une machine de Mealy M à moins de deux états telle qu'aucune machine de Moore alternative n'est équivalente à M .

Partie II. Transducteurs sous-séquentiels

Dans cette partie, on étend la définition des transducteurs séquentiels en autorisant l'écriture d'un préfixe et d'un suffixe lors du calcul de l'image d'un mot.

II.1 Définition et exemples

On définit un transducteur sous-séquentiel comme un septuplet $(Q, \Sigma, \Gamma, \delta, \lambda, q_0, \rho)$ tel que $(Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ est un transducteur séquentiel et ρ est une fonction de Q dans Γ^* appelée **fonction de suffixe**.

Si $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0, \rho)$ est un transducteur sous-séquentiel, on définit la fonction associée à T comme la fonction $\varphi_T : \Sigma^* \longrightarrow \Gamma^*$. Autrement dit l'image de u est $u \mapsto \lambda(q_0, u)\rho(\delta(q_0, u))$ constituée de la sortie du calcul d'entrée u suivie du suffixe associé à l'état final du calcul.

On représente graphiquement un transducteur sous-séquentiel de la même façon qu'un transducteur séquentiel, en rajoutant des flèches sortantes étiquetées par les suffixes. Par exemple, la figure 5 représente un transducteur sous-séquentiel T_2 vérifiant $\rho(q_0) = 00$, $\rho(q_1) = \varepsilon$ et $\rho(q_2) = 1$.

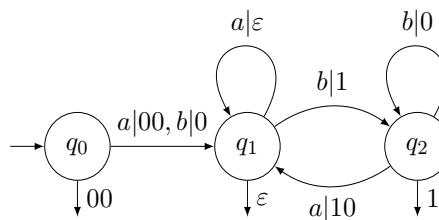


FIGURE 5 – Le transducteur sous-séquentiel T_2

On a par exemple $\varphi_{T_2}(\varepsilon) = 00$ et $\varphi_{T_2}(aabbaa) = 001010$.

Enfin, si f est une fonction de Σ^* dans Γ^* , on dit que f est une **fonction sous-séquentielle** s'il existe un transducteur sous-séquentiel T tel que $f = \varphi_T$.

Question 18 Expliquer comment modifier la machine de Moore de la question 14 en un transducteur sous-séquentiel qui fait une addition exacte, c'est-à-dire sans modulo.

Question 19 On considère le transducteur sous-séquentiel T_3 de la figure 6. Montrer que si u est un mot non vide qui représente une écriture de $x \in \mathbb{N}$ en base 2, alors $\varphi_{T_3}(u)$ représente une écriture de $3x$ en base 2.

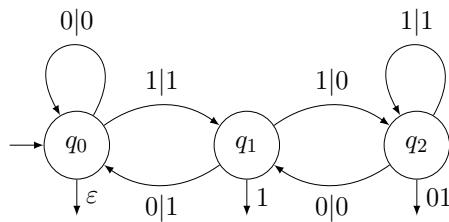


FIGURE 6 – Le transducteur sous-séquentiel T_3

Question 20 Représenter graphiquement sans justifier un transducteur sous-séquentiel à 5 états sur $\Sigma = \Gamma = \{0, 1\}$ qui transforme un mot u non vide représentant une écriture de $x \in \mathbb{N}$ en base 2 en un mot v représentant une écriture en base 2 de $5x$.

II.2 Langages rationnels

Dans cette sous-partie, on considère $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0, \rho)$ un transducteur sous-séquentiel.

Question 21 Montrer que si L est un langage rationnel sur Σ , alors $\varphi_T(L)$ est un langage rationnel sur Γ .

Indication : on pourra admettre qu'un automate fini dont les transitions sont étiquetées par des mots (plutôt que des lettres) reconnaît aussi un langage rationnel.

Question 22 Montrer que la réciproque de la question précédente est fausse.

Soit L un langage sur Γ . On appelle image réciproque de L par T , notée $\varphi_T^{-1}(L)$, l'ensemble des mots dont l'image par φ_T est dans L , c'est-à-dire $\varphi_T^{-1}(L) = \{u \in \Sigma^* \mid \varphi_T(u) \in L\}$.

Question 23 Montrer que si L est un langage rationnel sur Γ , alors $\varphi_T^{-1}(L)$ est un langage rationnel sur Σ . On explicitera un automate fini, construit à partir de T et d'un automate

reconnaissant L , et on montrera qu'il reconnaît exactement $\varphi_T^{-1}(L)$.

Indication : on commencera par traiter le cas où T est séquentiel.

II.3 Fonctions sous-séquentielles

Question 24 Soit $f : \Sigma^* \rightarrow \Gamma^*$ et $g : \Gamma^* \rightarrow \Lambda^*$ deux fonctions sous-séquentielles. Montrer que $g \circ f$ est une fonction sous-séquentielle.

On pourra commencer par traiter le cas de fonctions séquentielles.

Soit $f : \Sigma^* \rightarrow \Gamma^*$ une fonction. On dit que f **conserve les préfixes** si $f(\varepsilon) = \varepsilon$ et si pour tous mots u et v de Σ^* , si u est un préfixe de v , alors $f(u)$ est un préfixe de $f(v)$.

Question 25 Montrer par un exemple qu'il existe une fonction sous-séquentielle qui ne conserve pas les préfixes.

Soit $f : \Sigma^* \rightarrow \Gamma^*$ une fonction. On cherche à montrer l'équivalence entre les deux propositions suivantes :

- (a) f est une fonction séquentielle.
- (b) f est une fonction sous-séquentielle et f conserve les préfixes.

Question 26 Montrer l'implication (a) \Rightarrow (b) de l'équivalence précédente.

Question 27 Soit $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0, \rho)$ un transducteur sous-séquentiel tel que φ_T conserve les préfixes. On suppose que tous les états de Q sont accessibles. Montrer que pour tout $q \in Q$ et $a \in \Sigma$, il existe un mot qu'on notera $\lambda'(q, a) \in \Gamma^*$ tel que $\rho(q)\lambda'(q, a) = \lambda(q, a)\rho(\delta(q, a))$.

Question 28 Avec les notations de la question précédente, on étend l'ensemble de définition de λ' à $Q \times \Sigma^*$ comme pour les fonctions de sortie des transducteurs. Montrer que pour $q \in Q$ et $u \in \Sigma^*$, $\rho(q)\lambda'(q, u) = \lambda(q, u)\rho(\delta(q, u))$.

Question 29 En déduire l'implication (b) \Rightarrow (a) de l'équivalence précédente.

Partie III. Théorèmes de Ginsburg-Rose et Choffrut

III.1 Distance préfixe

Pour u et v deux mots de Σ^* , on note $u \wedge v$ le plus long préfixe commun à u et v . On définit la **distance préfixe** entre u et v par $d(u, v) = |u| + |v| - 2|u \wedge v|$. Par exemple, si $u = ababa$ et $v = abbbbaab$, alors $u \wedge v = abb$ et $d(u, v) = |u| + |v| - 2|u \wedge v| = 6 + 9 - 2 \times 3 = 9$.

Question 30 Montrer que la fonction de distance préfixe est une distance, c'est-à-dire que pour tous u, v, w mots de Σ^* , d vérifie les quatre propriétés suivantes :

- $d(u, v) = d(v, u)$;
- $d(u, v) \geq 0$;
- $d(u, v) = 0 \Leftrightarrow u = v$;
- $d(u, w) \leq d(u, v) + d(v, w)$.

Question 31 Soit $X \subseteq \Sigma^*$ un ensemble de mots non vide. Soit u le plus long préfixe commun à tous les mots de X . Montrer que pour $v \in X$, $d(u, v) \leq \max_{(x,y) \in X^2} d(x, y)$.

On dit qu'une fonction $f : \Sigma^* \rightarrow \Gamma^*$ est **lipschitzienne** s'il existe un entier K strictement positif tel que :

$$\forall (u, v) \in \Sigma^* \times \Sigma^*, d(f(u), f(v)) \leq Kd(u, v)$$

On cherche dans cette partie à montrer les théorèmes suivants :

- **Théorème de Ginsburg et Rose (1966)** : Soit $f : \Sigma^* \rightarrow \Gamma^*$. La fonction f est une fonction séquentielle si et seulement si elle vérifie les trois propriétés suivantes :
 - (a) f conserve les préfixes ;
 - (b) pour tout langage rationnel L sur Γ^* , $f^{-1}(L)$ est rationnel ;
 - (c) f est lipschitzienne.
- **Théorème de Choffrut (1978)** : Soit $f : \Sigma^* \rightarrow \Gamma^*$. La fonction f est une fonction sous-séquentielle si et seulement si elle vérifie les deux propriétés suivantes :
 - (a) pour tout langage rationnel L sur Γ^* , $f^{-1}(L)$ est rationnel ;
 - (b) f est lipschitzienne.

Question 32 On suppose vrai le théorème de Choffrut. Montrer le théorème de Ginsburg et Rose.

Question 33 Soit $f : \Sigma^* \rightarrow \Gamma^*$ une fonction sous-séquentielle. Montrer que f est lipschitzienne. En déduire le sens direct du théorème de Choffrut.

Indication : si $f = \varphi_T$ avec $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0, \rho)$, on posera $K = K_1 + 2K_2$, où K_1 et K_2 sont définies par $K_1 = \max\{|\lambda(q, a)| \mid q \in Q, a \in \Sigma\}$ et $K_2 = \max\{|\rho(q)| \mid q \in Q\}$.

La suite de cette partie a pour objectif de montrer le sens réciproque du théorème de Choffrut. On considère pour la suite une fonction $f : \Sigma^* \rightarrow \Gamma^*$ qui vérifie :

- (a) pour tout langage rationnel L sur Γ^* , $f^{-1}(L)$ est rationnel ;
- (b) f est lipschitzienne.

On cherche à montrer que f est une fonction sous-séquentielle.

III.2 Découpage de $f(uv)$

Pour $u \in \Sigma^*$, on définit $\pi(u)$ comme le plus long préfixe commun aux mots $f(uv)$, pour $v \in \Sigma^{\leq 1}$. Ainsi, pour tout $v \in \Sigma^{\leq 1}$, il existe $\sigma_u(v)$ tel que :

$$f(uv) = \pi(u)\sigma_u(v)$$

Question 34 Montrer qu'il existe un entier naturel M tel que pour tout $u \in \Sigma^*$ et tous v_1, v_2 dans $\Sigma^{\leq 1}$, on a :

$$d(f(uv_1), f(uv_2)) \leq M$$

Question 35 En déduire que pour tout $u \in \Sigma^*$ et $v \in \Sigma^{\leq 1}$, $|\sigma_u(v)| \leq M$ puis que l'ensemble $S = \{\sigma_u \mid u \in \Sigma^*\}$ est fini.

Pour $s \in S$, on pose alors $X_s = \{u \in \Sigma^* \mid \sigma_u = s\}$.

III.3 Construction d'un transducteur sous-séquentiel

On admet temporairement que pour $s \in S$, l'ensemble X_s est un langage rationnel sur Σ .

Question 36 Soit L_1 et L_2 deux langages rationnels sur Σ . Montrer qu'il existe un automate fini déterministe complet $A = (Q, \Sigma, \delta, q_0, F)$ ainsi que deux sous-ensembles $Q_1 \subseteq Q$ et $Q_2 \subseteq Q$ tels que $L_1 = \{u \in \Sigma^* \mid \delta(q_0, u) \in Q_1\}$ et $L_2 = \{u \in \Sigma^* \mid \delta(q_0, u) \in Q_2\}$.

Un automate fini déterministe est dit **standard** si aucune transition ne pointe vers l'état initial.

Question 37 Montrer qu'il existe un automate fini déterministe complet et standard $A = (Q, \Sigma, \delta, q_0, F)$ tel qu'en posant, pour tout $s \in S$, $Q_s = \{\delta(q_0, u) \mid u \in X_s\}$, alors on a :

- $\forall s, s' \in S, Q_s \cap Q_{s'} \neq \emptyset \Leftrightarrow s = s'$;
- $\bigcup_{s \in S} Q_s = Q$.

Pour $q \in Q$, on pose $U_q = \{u \in \Sigma^* \mid \delta(q_0, u) = q\}$. On pose de plus $\beta(q)$ le plus long suffixe des $\pi(u)$, pour $u \in U_q$. Il existe donc une fonction α telle que pour $u \in U_q$:

$$\pi(u) = \alpha(u)\beta(q)$$

On veut montrer, par disjonction de cas, que pour tout $u \in U_q$ et $a \in \Sigma$, $\alpha(u)$ est un préfixe de $\alpha(ua)$. Pour cela, on note $s \in S$ tel que $q \in Q_s$, $q' = \delta(q, a)$ et s' tel que $q' \in Q_{s'}$.

Question 38 Montrer que $\alpha(u)\beta(q)s(a) = \alpha(ua)\beta(q')s'(\varepsilon)$.

On suppose pour les deux questions suivantes que $|s(a)| \leq |s'(\varepsilon)|$.

Question 39 Montrer qu'il existe $v \in \Gamma^*$ tel que $\pi(u) = \pi(ua)v$, et que v ne dépend pas du mot u choisi dans U_q .

Question 40 Montrer que nécessairement $|\beta(q')v| \leq |\beta(q)|$, puis qu'il existe $w \in \Gamma^*$ tel que $\alpha(u)w = \alpha(ua)$, et que w ne dépend pas du mot u choisi dans U_q .

On admet que le cas $|s(a)| > |s'(\varepsilon)|$ se traite de manière similaire. En conclusion, il existe une fonction $\lambda : Q \times \Sigma \rightarrow \Gamma$ telle que pour tout $q \in Q$, $u \in U_q$ et $a \in \Sigma$, $\alpha(u)\lambda(q, a) = \alpha(ua)$.

On pose, pour $q \in Q$ tel que $q \in X_s$, $\rho(q) = \beta(q)s(\varepsilon)$.

Question 41 En considérant le transducteur sous-séquentiel $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0, \rho)$, montrer le théorème de Choffrut.

III.4 Rationalité des X_s

On cherche dans cette partie à montrer que pour tout $s \in S$, l'ensemble X_s est un langage rationnel. Pour cela, on définit successivement plusieurs familles intermédiaires de langages permettant de reconstruire X_s .

Pour $i \in \{0, 1, \dots, 2M\}$ et $z \in \Gamma^M$, l'entier M étant celui défini à la question 34, on définit $B(i, z)$ par :

$$B(i, z) = \{x \in \Gamma^* \mid |x| \equiv i \pmod{2M+1} \text{ et } (x \in \Gamma^* z \text{ ou } z \in \Gamma^* x)\}$$

c'est-à-dire que $x \in B(i, z)$ si et seulement si la taille de x est de la forme $|x| = (2M+1) \times k + i$ avec k entier, et x est un suffixe de z ou que z est un suffixe de x .

Question 42 Montrer que $B(i, z)$ est un langage rationnel sur Γ .

Pour $i \in \{0, 1, \dots, 2M\}$, $z \in \Gamma^M$, $s \in S$ et $v \in \Sigma^{\leq 1}$, on définit $C_s(i, z, v)$ par :

$$C_s(i, z, v) = \{u \in \Sigma^* \mid f(uv) \in B(i, z)s(v)\}$$

Question 43 Montrer que $C_s(i, z, v)$ est un langage rationnel sur Σ .

Enfin, pour $s \in S$, on définit Y_s par :

$$Y_s = \bigcup_{i=0}^{2M} \bigcup_{z \in \Gamma^M} \bigcap_{v \in \Sigma^{\leq 1}} C_s(i, z, v)$$

Le résultat de la question 43 et les propriétés de stabilité des langages rationnels garantissent que Y_s est un langage rationnel sur Σ .

Question 44 Montrer que pour $s \in S$, $X_s \subseteq Y_s$.

On veut maintenant démontrer que $Y_s \subseteq X_s$. On considère $y \in Y_s$, et $i \in \{0, 1, \dots, 2M\}$, $z \in \Gamma^M$ tels que $y \in \bigcap_{v \in \Sigma^{\leq 1}} C_s(i, z, v)$.

Question 45 Soient $v_1, v_2 \in \Sigma^{\leq 1}$ et $b_1, b_2 \in B(i, z)$ tels que $f(yv_1) = b_1 s(v_1)$ et $f(yv_2) = b_2 s(v_2)$. Montrer que $|b_1| = |b_2|$. Pour la suite, on pose n_s cette taille commune.

Indication : on pourra utiliser, après l'avoir montrée, l'inégalité $\|u - v\| \leq d(u, v)$.

Question 46 Montrer qu'il existe $v_1, v_2 \in \Sigma^{\leq 1}$ tels que $s(v_1) \wedge s(v_2) = \varepsilon$. En déduire que $|\pi(y)| \leq n_s$.

On admet que, de même, il existe $v_1, v_2 \in \Sigma^{\leq 1}$ tels que $\sigma_y(v_1) \wedge \sigma_y(v_2) = \varepsilon$.

Question 47 En considérant $v_1, v_2 \in \Sigma^{\leq 1}$ tels que $\sigma_y(v_1) \wedge \sigma_y(v_2) = \varepsilon$, montrer que $s(v_1)$ est un suffixe de $\sigma_y(v_1)$ et que $s(v_2)$ est un suffixe de $\sigma_y(v_2)$. En déduire que $|\pi(y)| = n_s$.

Question 48 En déduire que $Y_s \subseteq X_s$.

* *
*

Banque X-E.N.S. MP-PC-PSI (2025)	
Informatique B - XELSR (2 h) [I25-XENS-MP-PC-PSI-B]	3
Le jeu de Rockse	
Mines-Ponts MP-PC-PSI (2025)	
Épreuve d'informatique commune (2 h) [I25-CCMP-MP-PC-PSI]	11
Autour du sac à dos	
CCINP PC-PSI (2025)	
Informatique (3 h) [I25-CCINP-PC-PSI]	26
Gestion de Randonnées	
CCINP TPC-TSI (2025)	
Informatique (3 h) [I25-CCINP-TPC-TSI]	42
Détection d'objets dans le cadre de la conduite autonome	
Banque PT (2025)	
Épreuve d'informatique et Modélisation (4 h) [I25-BANQUE-PT]	53
Autour de la géolocalisation par satellites	
Banque X-E.N.S. MP-MPI (2025)	
Informatique A - XULSR (4 h) [I25-XENS-MP-MPI-A]	74
Arbres de classification d'arbres	
Banque X-E.N.S. MP-MPI (2025)	
Informatique C - XULSR (4 h) [I25-XENS-MP-MPI-C]	87
Coupages maximaux et parfaits	
Mines-Ponts MPI (2025)	
Informatique 1 (3 h) [I25-CCMP-MPI-1]	106
Étude des tableaux associatifs	
Mines-Ponts MPI (2025)	
Informatique 2 (3 h) [I25-CCMP-MPI-2]	119
Le jeu de Shannon	
Mines-Ponts MP (2025)	
Option informatique (3 h) [I25-CCMP-MP-INFO]	129
Partie 1. Alphabets, mots et automates	
Partie 2. Relation de séparabilité par rapport à un langage	
Partie 3. Arbre discriminant	
Partie 4. Automate tiré d'un arbre discriminant	

Centrale-Supélec MPI (2025)

Informatique (4 h) [I25-CCS-MPI]	139
Rush Hour	

Centrale-Supélec MP (2025)

Option informatique (4 h) [I25-CCS-MP-INFO]	152
Meilleurs itinéraires dans un réseau ferroviaire	

CCINP MPI (2025)

Informatique (4 h) [I25-CCINP-MPI]	162
Partie 1. Grammaire non contextuelle	
Partie 2. Problème du bin-packing	
Partie 3. Algorithmes de couplage	

CCINP MP (2025)

Option informatique (3 h) [I25-CCINP-MP-INFO]	170
Répartition de la gestion d'un réseau minimisant la bande passante	

CAPES externe NSI (2025)

Epreuve 1 (5 h) [I25-CAPES-1]	186
Autour du problème de la couverture par ensembles	

CAPES externe NSI (2025)

Epreuve 2 (5 h) [I25-CAPES-2]	202
Partie A : La photographie numérique / Partie B : optimisation par la programmation dynamique	

Agrégation externe d'informatique (2025)

Composition en informatique (5 h) [I25-AGREG-1]	217
Gestion de fichiers dans un éditeur de texte	

Agrégation externe d'informatique (2025)

Étude d'un problème informatique (6 h) [I25-AGREG-2]	232
Mastermind	

Agrégation externe d'informatique (2025)

Épreuve spécifique (6 h) [I25-AGREG-3]	248
Étude de cas informatique : Système de gestion de projets informatiques	
Fondements de l'informatique : Transducteurs finis	

SOMMAIRE

Informatique commune	3
MPI & MP option informatique	74
CAPES & agrégation d'informatique	186