CONCOURS GÉNÉRAL DES LYCÉES

—

NUMERIQUE ET SCIENCES INFORMATIQUES

(Classes de terminale voie générale spécialité numérique et sciences informatiques)

Sujet zéro

Durée : 5 heures

—

L'usage de la calculatrice est interdit.

Attention : ce sujet 0 a une structure particulière, où l'on a cherché à faire un sujet pour chacun de 5 des 6 thèmes du programme, l'exception étant l'histoire de l'informatique, que nous avons considérée comme peu adaptée au CGL. Les 5 thèmes ne seront pas tous couverts lors des sujets suivants, et il est aussi possible d'avoir des exercices qui couvrent plusieurs thèmes en même temps. Le but ici est de montrer le genre de questions que l'on peut poser pour faire réfléchir dans chaque thème.

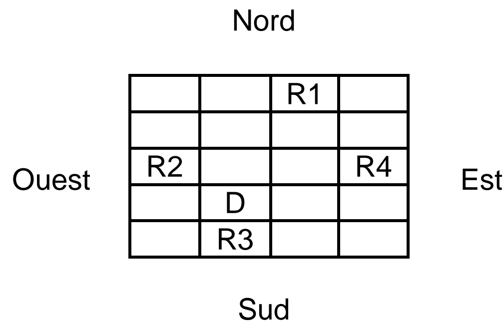
Algorithmique

Dans cet exercice, on va s'intéresser aux coûts en temps et en mémoire des algorithmes proposés, c'est-à-dire à l'ordre de grandeur du nombre d'octets utilisés, et à l'ordre de grandeur du nombre d'instructions. Comme on ne va manipuler que des tableaux et des entiers dont la valeur ne dépasse pas 1 milliard, pour le coût en mémoire, on considérera simplement l'ordre de grandeur du nombre d'entiers stockés tandis que pour le coût en temps, on considérera l'ordre de grandeur du nombre de fois où l'on accède en lecture ou écriture à un entier.

A – Simulation d'une séquence de mouvements de robots

On considère une grille de `nbLignes` lignes et `nbColonnes` colonnes, contenant `nbRobots` robots et un éventuel drapeau à atteindre. Une case ne peut contenir qu'une seule chose.

Voici un exemple où `nbLignes = 5`, `nbColonnes = 4` et `nbRobots = 4`. Les robots sont numérotés de 1 à 4, marqués R1 à R4 dans l'illustration. Le drapeau est marqué par un D.



Pour effectuer un mouvement, on va créer une fonction `bouger(direction)`, où le paramètre `direction` peut valoir "E", "O", "N" ou "S" pour Est, Ouest, Nord et Sud, respectivement. Lors de l'appel de cette fonction, tous les robots encore sur la grille doivent se déplacer d'une case dans cette direction. Lorsqu'un robot atteint le drapeau, ce robot est sauvé et est retiré. Si un robot se déplace vers l'extérieur de la grille, il est éliminé, et retiré de la grille. Lors d'un tour, tous les robots se déplacent en même temps, donc deux robots ne peuvent pas se retrouver dans la même case.

Pour accéder au contenu de la grille, on va créer une fonction `afficherEtat(indiceRobot)`, où le paramètre `indiceRobot` vaut entre 1 et `nbRobots` inclus. La fonction affiche 1 si le robot est sauvé, -1 s'il est éliminé, ou ses coordonnées actuelles (x', y') s'il est encore dans la grille. La case dans le coin haut gauche a pour coordonnées (0,0), et la case à sa droite les coordonnées (1, 0).

Voici un exemple de programme, et l'affichage correspondant si on l'exécute en partant de la grille d'exemple.

Programme	Affichage attendu de chaque instruction
`bouger("E")`	
`afficherEtat(1)`	`(3, 0)`
`afficherEtat(4)`	`-1`
`bouger("S")`	
`afficherEtat(2)`	`1`
`afficherEtat(3)`	`-1`
`afficherEtat(1)`	`(3,1)`

Après le premier mouvement, le robot R4 sort de la grille et est éliminé. On obtient le placement suivant :

			R1
	R2		
	D		
		R3	

Après le deuxième mouvement, le robot R2 est sauvé, R3 sort de la grille et est éliminé. On obtient le placement suivant :

			R1
	D		

1) On part de la grille de départ ci-dessous, et on exécute le programme suivant :

<table>
<tr><th>Grille de départ</th><th>Programme</th></tr>
<tr><td>
<table>
<tr><td></td><td></td><td></td><td></td><td>R1</td></tr>
<tr><td></td><td></td><td></td><td>D</td><td></td></tr>
<tr><td>R2</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>R3</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>R4</td><td></td></tr>
</table>
</td><td>
bouger("N")
bouger("N")
afficherEtat(3)
bouger("E")
afficherEtat(1)
bouger("O")
afficherEtat(3)
bouger("N")
afficherEtat(2)
afficherEtat(4)
</td></tr>
</table>

Quel est l'affichage attendu pour ce programme ?

2) Avec cette même grille de départ, on souhaite sauver le robot R1. Il est possible de sauver un autre robot. Indiquer lequel et écrire un programme qui permet de sauver ces deux robots

3) Dans cette question, on choisit de stocker l'état de la grille dans une variable globale, qui est une liste de listes, où l'on représente un robot par son numéro, 0 pour les cases vides, et le drapeau par la valeur -1.

Par exemple la grille d'exemple initial peut être stockée comme ceci :

```
grille = [[0, 0, 1, 0], [0, 0, 0, 0], [2, 0, 0, 4], [0, -1, 0, 0], [0,
3, 0, 0]]
```

On utilise par ailleurs une autre variable globale, qui stocke la liste des robots déjà sauvés. Par exemple, si les robots 3 et 1 ont été sauvés, on stocke l'information comme ceci : `robotsSauves = [3, 1]`

Pour mettre à jour le contenu de ces variables lors d'un appel à `bouger`, on doit passer sur chaque case de la grille.

Par exemple, pour un déplacement vers l'ouest, on peut utiliser le code suivant :

```
for ligne in range(nbLignes):
  for colonne in range(nbColonnes):
    contenuCase = grille[ligne][colonne]
    if contenuCase > 0:
       grille[ligne][colonne] = 0
       if colonne > 0:
         if grille[ligne][colonne - 1] == -1:
           robotsSauves.append(contenuCase)
         else:
           grille[ligne][colonne - 1] = contenuCase
```

Le coût en temps d'un appel à la fonction `bouger` est de l'ordre de `nbLignes×nbColonnes`.

On considère maintenant la fonction `afficherEtat`. Écrire son code en utilisant ces mêmes structures, sans les modifier.

Quel est le coût en temps d'un appel à cette fonction ?

B – Optimisation du coût en temps et mémoire

Dans cette partie, on demande d'écrire plusieurs versions des mêmes fonctions. Les questions sont de difficulté graduelle, mais il est possible que vous voyiez directement comment traiter une question difficile.

Ainsi, si vous trouvez une solution qui répond correctement à plusieurs questions, vous pouvez simplement la référencer. Par exemple, si vous êtes sûr d'avoir la réponse pour 5c, vous pouvez écrire "même réponse que 5c" pour les autres.

4) Supposons que `nbRobots` soit nettement plus petit que `nbLignes` et `nbColonnes`. Proposer une nouvelle manière de stocker les données, qui permette une implémentation plus rapide de `bouger` et `afficherEtat`

Décrire le principe de cette structure, et indiquer le coût en temps et en mémoire qu'elle permet d'obtenir pour les deux fonctions. On ne vous demande pas d'implémenter les deux fonctions.

5) Dans cette question, le coût en temps des deux fonctions doit être constant : il ne doit pas dépendre de `nbRobots`, de la taille de la grille, ou du nombre d'appels effectués jusqu'à présent. Décrire vos approches, sans les implémenter.

a) Proposer une approche qui fonctionne si l'on suppose qu'aucun déplacement n'élimine un robot en le faisant sortir de la grille.

b) Proposer une approche qui fonctionne s'il n'y a pas de drapeau. Par contre, des déplacements peuvent éliminer des robots.

c) Proposer une approche qui fonctionne dans tous les cas.

C – Tous les sous-ensembles sauvables

Dans cette question, on considère que `nbRobots` ≤ 20.

Pour représenter un sous-ensemble des `nbRobots` robots qui sont sauvés, utiliser un entier entre 0 et $2^{nbRobots}$ -1. Le i-ème bit de cet entier (en partant du bit de poids faible, à droite dans la représentation), sera 1 si le i-ème robot est déjà sauvé, et 0 sinon. Ainsi on peut encoder le fait que les robots 1, 4 et 6 sont sauvés grâce à l'entier binaire 101001, qui correspond au nombre 41 écrit en base 10.

On rappelle les éléments de syntaxe python permettant de manipuler les bits d'un entier :

- `a | b` effectue un ou bit à bit des entiers a et b. Par exemple `6 | 3` vaut `7`.
- `a & b` effectue un et bit à bit des entiers a et b. Par exemple `6 & 3` vaut `2`.
- `a ^ b` effectue un ou exclusif bit à bit des entiers a et b. Par exemple `6 ^ 3` vaut `5`.
- `a << n` décale le nombre a de n bits vers la gauche. Par exemple `6 << 3` vaut `48`.
- `a >> n` décale le nombre a de n bits vers la droite. Par exemple `48 >> 3` vaut `6`.

Voici par exemple le code d'une fonction qui renvoie un ensemble où le robot `numRobot est ajouté` s'il n'est pas déjà dedans, et retiré sinon :

```
def inverserRobot(ensemble, numRobot):
   return ensemble ^ (1 << (numRobot - 1))
```

6) Donner en base 10, le nombre correspondant à la représentation du sous-ensemble constitué des robots 5, 8 et 10.

7) Écrire une fonction python qui prend en paramètre deux entiers décrivant des ensembles de
`K <= nbRobots`
robots, et retourne l'union de ces deux ensembles.

8) Écrire une fonction python qui prend en paramètre un entier décrivant un ensemble de
`K <= nbRobots`
robots, et retourne la liste des numéros de ces robots.

9) Écrire une fonction qui prend en paramètre un entier décrivant un sous-ensemble de
`K <= nbRobots`

robots, et détermine s'il existe une séquence de mouvements qui sauve tous ces robots. Votre algorithme doit avoir un coût en temps ne dépassant pas $\texttt{nbRobots}^2$.

Expliquer aussi brièvement l'idée de votre algorithme en français.

10) Écrire un programme python qui renvoie la liste de tous les sous-ensembles de robots qu'il est possible de sauver. Expliquer aussi brièvement l'idée de votre algorithme en français.

Donner le coût en temps de votre algorithme. Votre algorithme doit avoir un coût en temps ne dépassant pas $\texttt{nbRobots} \times 2^{\texttt{nbRobots}}$.

On remarquera qu'appliquer l'algorithme en $\texttt{nbRobots}^2$ de la question précédente à chaque sous-ensemble de robots possible, donne un coût en temps de $\texttt{nbRobots}^2 \times 2^{\texttt{nbRobots}}$, donc est trop lent.

D – Plus grand sous-ensemble sauvable

Dans cette partie, on demande de calculer le plus grand nombre de robots que l'on puisse sauver.

Il n'est pas nécessaire d'implémenter vos algorithmes, mais vous devez les décrire clairement et donner leur coût en temps et en mémoire.

11) Décrire un algorithme qui fonctionne pour une grille d'une seule ligne, c'est-à-dire telle que `nbLignes` = 1. Pour obtenir tous les points, son coût en temps ne doit pas dépasser `nbColonnes`.

12) Décrire un algorithme qui fonctionne si le drapeau se trouve dans la case en bas à droite, aux coordonnées (`nbLignes` - 1, `nbColonnes` - 1). Son coût en temps ne doit pas dépasser $\texttt{nbLignes} \times \texttt{nbColonnes}$

13) Décrire un algorithme qui fonctionne quelle que soit la position du drapeau. Son coût en temps ne doit pas dépasser $\texttt{nbLignes}^2 \times \texttt{nbColonnes}^2$.

Architectures matérielles, systèmes d'exploitation et réseaux

A – Exécutions en parallèle

Dans un ordinateur, même avec un unique processeur, plusieurs programmes peuvent s'exécuter simultanément. En pratique, l'exécution de ces programmes est un entrelacement d'instructions. On va ici prendre un point de vue simplifié en imaginant deux programmes Python s'exécutant simultanément et pour lesquels les instructions de l'un ou de l'autre peuvent être exécutées. Le choix de ce qui sera exécuté à un moment donné est extérieur et n'est pas maîtrisé par la personne qui lance l'exécution. L'objectif de cette partie est d'étudier des mécanismes de synchronisation permettant de contrôler ce comportement.

On dispose des deux programmes suivants. L'instruction `ajouter`, ajoute immédiatement une lettre à la fin d'un fichier auquel les deux programmes ont accès.

Programme A1	Programme A2
`ajouter("R")` `ajouter("V")`	`ajouter("B")` `ajouter("V")`

Si l'ordinateur exécute le programme A1 en entier puis le programme A2, le résultat est : `RVBV`.

Si l'ordinateur exécute la première instruction du programme A1, puis tout le programme A2, puis la deuxième instruction du programme A1, on obtient : `RBVV`.

14) Sachant que chaque programme n'est appelé qu'une fois, donner, par ordre alphabétique, tous les résultats différents possibles de l'exécution de ces deux programmes.

B – Jetons

Dans la suite, on dispose des deux fonctions suivantes, qui communiquent par un partage de ressources symbolisé par des jetons :

`consommer("Bleu")` : attend qu'un jeton Bleu soit disponible, puis détruit ce jeton.
`produire("Bleu")` : produit un jeton Bleu et le rend disponible

Un même jeton ne peut pas être consommé en même temps par deux programmes qui s'exécutent en parallèle. Un programme peut être bloqué indéfiniment s'il cherche à consommer un jeton qui n'est jamais produit par un autre programme.

Prenons par exemple les deux programmes suivants, en considérant qu'au départ, on dispose d'un seul jeton Bleu :

Programme B1	Programme B2

`consommer("Bleu")`
`ajouter("A")`
`ajouter("B")`
`produire("Bleu")`
	`consommer("Bleu")`
	`ajouter("C")`
	`produire("Bleu")`

Si l'on commence par exécuter la première instruction du programme B1, l'unique jeton bleu est consommé. Si l'on exécute ensuite la première instruction du programme B2, celle-ci est bloquée car aucun jeton Bleu n'est disponible. Lorsque la dernière instruction du programme B1 est exécutée, elle produit un nouveau jeton Bleu. La première instruction du programme B2 peut alors se terminer, et consomme ce jeton.

Si chacun de ces deux programmes n'est appelé qu'une fois, on n'a donc que deux résultats possibles : `ABC` et `CAB`.

15) Donner par ordre alphabétique les différents résultats possibles des deux programmes ci-dessous, sachant que chacun des deux programmes n'est appelé qu'une fois.

Au départ, un jeton Bleu est disponible.	

Programme C1
`consommer("Bleu")`
`ajouter("R")`
`ajouter("V")`
`produire("Bleu")`
	Programme C2
	`consommer("Bleu")`
	`ajouter("B")`
	`ajouter("V")`
	`produire("Bleu")`
	`ajouter("C")`

16) Donner par ordre alphabétique les différents résultats possibles des deux nouveaux programmes ci-dessous. Chacun des deux programmes n'est appelé qu'une fois.

Au départ, un jeton Bleu et un jeton Rouge sont disponibles.	

Programme D1
`consommer("Bleu")`
`ajouter("R")`
`ajouter("V")`
`produire("Bleu")`
`consommer("Rouge")`
`ajouter("D")`
`ajouter("V")`
`produire("Rouge")`
	Programme D2
	`consommer("Bleu")`
	`ajouter("B")`
	`ajouter("V")`
	`produire("Bleu")`
	`consommer("Rouge")`
	`ajouter("C")`
	`produire("Rouge")`

17) Modifier les deux programmes ci-dessous, pour qu'un maximum de résultats différents puissent être produits, et que tous les résultats commençant par R se terminent par C.

Programme E1	Programme E2
`ajouter("R")` `ajouter("V")`	`ajouter("B")` `ajouter("C")`

Vos modifications doivent consister uniquement à insérer des appels à `consommer` et `produire` où vous voulez dans les deux programmes. Choisir et indiquer le nombre de jetons de chaque type disponible au départ.

Donner par ordre alphabétique, les résultats que peut produire l'exécution simultanée de vos deux programmes.

C – 10 appels en parallèle

Dans cette partie, on considère toujours deux programmes, mais chacun d'eux peut être exécuté plusieurs fois (au total, on limite à 10 le nombre total d'appels aux programmes). Par exemple, considérons les deux programmes suivants :

Programme F1	Programme F2
`ajouter("X")` `ajouter("Y")`	`ajouter("W")` `ajouter("Z")`

On peut considérer le déroulement d'une exécution avec 3 appels parallèles, le programme F1 étant appelé deux fois (appels A et B) et le programme F2 une fois (appel C). Une telle exécution peut par exemple se dérouler ainsi :

- Appel A : dans le programme F1, l'instruction 1 s'exécute
- Appel B : dans le programme F1, l'instruction 1 s'exécute
- Appel C : dans le programme F2, l'instruction 1 s'exécute
- Appel A : dans le programme F1, l'instruction 2 s'exécute, l'appel A se termine
- Appel C : dans le programme F2, l'instruction 2 s'exécute, l'appel C se termine
- Appel B : dans le programme F1, l'instruction 2 s'exécute, l'appel B se termine.

Le contenu du fichier à l'issue de cette exécution est : `XXWYZY`.

Dans chaque question, vous pourrez ajouter des instructions `consommer` et `produire` aux deux programmes et utiliser autant de types de jetons que vous le souhaitez.

Pour chacune de ces questions, plus votre solution sera courte, plus votre score sera élevé. Par ailleurs, pour obtenir tous les points, votre solution doit pouvoir produire plus de 30 résultats différents selon l'ordre d'exécution des instructions.

Programme G1 `ajouter("1")`	Programme G2 `ajouter("2")`

18) Créer deux programmes H1 et H2 en ajoutant des instructions `consommer` et `produire` aux deux programmes G1 et G2, respectivement. Indiquer le nombre de jetons de chaque type disponible au départ, pour que l'exécution de jusqu'à 10 appels au total puisse produire au moins 30 résultats différents selon l'ordre d'exécution, et que le nombre de `1` et de `2` dans le fichier soit identique à chaque moment où aucun programme n'est en cours d'exécution, c'est-à-dire lorsque que tous les appels à H1 ou H2 qui ont commencés, se sont terminés.

Supposons par exemple que l'on effectue un seul appel à chacun des deux programmes, en commençant par un appel à votre programme H1. Votre programme H1 doit être tel que cet appel ne peut pas se terminer avant le début de l'appel au programme H2, pour ne pas se retrouver avec le fichier qui contient seulement `1` alors qu'aucun programme n'est en cours d'exécution.

Expliquer brièvement pourquoi votre solution fonctionne.

19) Créer deux programmes I1 et I2 en ajoutant des instructions `consommer` et `produire` aux deux programmes G1 et G2, respectivement. Indiquer le nombre de jetons de chaque type disponible au départ, pour que l'exécution de jusqu'à 10 appels au total puisse produire au moins 30 résultats différents selon l'ordre d'exécution, et que le nombre de 1 et de 2 dans le fichier ne diffère jamais de plus de 1 pendant l'exécution.

Par exemple, on ne pourra pas avoir un résultat qui commence par `22`, mais on peut avoir un résultat qui commence par `122`.

Expliquer brièvement pourquoi votre solution fonctionne.

20) On combine les deux contraintes précédentes. Créer deux programmes J1 et J2 en ajoutant des instructions `consommer` et `produire` aux deux programmes G1 et G2. Indiquer le nombre de jetons de chaque type disponible au départ, pour que l'exécution de jusqu'à 10 appels au total puisse produire au moins 30 résultats différents selon l'ordre d'exécution. Les contraintes sont que le nombre de 1 et de 2 dans le fichier soit identique à chaque moment où les programmes déjà démarrés sont tous terminés, et que le nombre de 1 et de 2 dans le fichier ne diffère jamais de plus de 1 pendant l'exécution.

Expliquer brièvement pourquoi votre solution fonctionne.

Langages de programmation

A – Ordinateur à une seule instruction

Un OISC (One Instruction Set Computer) est un ordinateur qui n'utilise qu'une seule instruction pour fonctionner. Nous allons étudier l'OISC qui utilise l'unique instruction `subleq`. Nous verrons qu'à partir de cette unique instruction, il est possible de faire beaucoup de choses.

Plus précisément, modélisons un ordinateur par une suite infinie de cellules mémoire numérotées à partir de 0, qui peuvent chacune contenir un nombre entier relatif quelconque. On note M la mémoire de l'ordinateur et on adopte les notations habituelles M[i] pour désigner la valeur contenue dans la cellule de mémoire numérotée i.

Cet ordinateur possède aussi un pointeur d'instruction, noté P, qui est un entier naturel qui désigne l'indice de la cellule de mémoire où se trouve la prochaine instruction à exécuter. La prochaine instruction à exécuter est ainsi toujours M[P]. L'ordinateur possède aussi trois registres A, B et C qui peuvent mémoriser des nombres relatifs.

À l'initialisation de l'OISC, un programmeur ou une programmeuse peut remplir un nombre fini de cellules avec les nombres de son choix. Toutes les autres cellules sont initialisées à 0. Il ou elle positionne aussi le pointeur d'instruction P sur une cellule mémoire de son choix.

L'ordinateur exécute ensuite le programme de la manière suivante :

a) L'ordinateur copie M[P] dans le registre A.

b) L'ordinateur copie M[P+1] dans le registre B.

c) L'ordinateur copie M[P+2] dans le registre C.

d) Si un des nombres écrits dans A, B ou C est strictement négatif, l'ordinateur s'arrête. Sinon, l'ordinateur calcule M[B] - M[A], et stocke le résultat dans M[B] (la cellule d'indice B).

e) Si la soustraction a donné un résultat négatif ou nul, l'ordinateur met le la valeur du registre C dans le pointeur d'instruction P ; sinon l'ordinateur augmente le pointeur d'instruction P de 3.

On recommence ensuite à partir de l'étape a).

Exemple :

Indice de cellule	0	1	2	3	4	5	6	7	8
Contenu	6	7	6	8	2	0	-2	3	9

Le pointeur d'instruction est initialement positionné sur la cellule 0.

Voici les étapes effectuées par le programme :

- Il place 6 dans A. M[A] vaut -2.
- Il place 7 dans B. M[B] vaut 3.
- Il place 6 dans C, calcule 3 - (-2) = 5, stocke 5 dans la cellule 7.
- Il augmente le pointeur d'instruction de 3 : il contient alors 3.

				P			A	B	
Indice de cellule	0	1	2	3	4	5	6	7	8
Nouveau contenu	6	7	6	8	2	0	-2	**5**	9

- Il place 8 dans A. M[A] vaut 9.
- Il place 2 dans B. M[B] vaut 6.
- Il place 0 dans C, calcule 6 - 9 = -3, stocke -3 dans la cellule 2.
- Il met 0 dans le pointeur d'instruction.

	P		B						A
Indice de cellule	0	1	2	3	4	5	6	7	8
Nouveau contenu	6	7	**-3**	8	2	0	-2	5	9

- Il place 6 dans A. M[A] vaut -2.
- Il place 7 dans B. M[B] vaut 5.
- Il place -3 dans C et s'arrête.

À l'issue de l'exécution, voici le contenu de la mémoire :

			P				A	B	
Indice de cellule	0	1	2	3	4	5	6	7	8
Nouveau contenu	6	7	-3	8	2	0	-2	5	9

21) Décrire les étapes de l'exécution du programme suivant et donner le contenu de la mémoire à l'issue de cette exécution :

Indice de cellule	0	1	2	3	4	5	6	7	8
Contenu	100	4	5	3	0	2	2	2	3

Le pointeur d'instruction est initialement positionné sur la cellule 1.

22) Écrire en Python une fonction `OISC(memoire, P)` qui permet de simuler un tel ordinateur, en considérant que l'on n'utilise jamais que les 1000 premières cellules de la mémoire. Elle prend deux paramètres :

- `memoire` est une liste de 1000 éléments, qui représente l'état initial de la mémoire.
- `P` est la valeur du pointeur d'instruction.

La fonction modifie la mémoire en appliquant les règles et renvoie une liste qui représente l'état de cette mémoire en fin d'exécution (quand cette exécution se termine).

B – Composition d'instructions

Dans les questions suivantes, vous devez fournir votre programme sous la forme du contenu initial de la mémoire et de la valeur du pointeur P. Vous pouvez découper le tableau sur plusieurs lignes si besoin. Moins votre programme utilisera de temps pour s'exécuter, plus votre note sera élevée.

23) Écrire un programme sous la forme ci-dessus, tel que si la cellule 0 contient le nombre x et la cellule 1 le nombre y au début de l'exécution, alors la cellule 0 contient x - y à la fin de l'exécution. On se rappellera que le pointeur d'instruction n'a pas besoin de commencer en 0.

24) Écrire un programme, tel que si la cellule 0 contient le nombre x et la cellule 1 le nombre y au début de l'exécution, alors la cellule 0 contient x + y à la fin de l'exécution.

25) Écrire un programme tel que si la cellule 0 contient le nombre x et la cellule 1 le nombre y au début de l'exécution, alors la cellule 0 contient le nombre y et la cellule 1 contient le nombre x à la fin de l'exécution.

26) Écrire un programme, tel que si la cellule 0 contient le nombre x et la cellule 1 le nombre y au début de l'exécution, alors la cellule 0 contient max(x, y) à la fin de l'exécution.

27) Écrire un programme tel que, si la cellule 0 contient le nombre x et la cellule 1 le nombre y au début de l'exécution, alors la cellule 0 contient x×y à la fin de l'exécution. On suppose que x et y sont tous deux strictement positifs.

28) Pour un K que vous choisissez, écrire un programme tel que si la cellule K contient un nombre naturel N et les cellules d'indices K + 1, ..., K + N contiennent respectivement des entiers a_1, ..., a_N au début de l'exécution, alors la cellule d'indice K + N + 1 contient la somme $a_1 + ... + a_N$ à la fin de l'exécution.

29) Cet ordinateur peut-il faire tous les calculs que peut faire un ordinateur classique ? Expliquer.

Structures de données

Dans ce sujet, on souhaite réfléchir à des idées de structures de données permettant de stocker le contenu d'une grille à deux dimensions, de taille 10 000 x 10 000, dont les cases contiennent des entiers, la plupart égaux à 0. L'objectif est d'économiser de la mémoire, sans pour autant ralentir les opérations de lecture.

On considère ici que chaque entier occupe 28 octets.

On considère une machine disposant d'un système de cache simplifié : la mémoire est découpée en blocs de 1ko. Lorsqu'on lit un entier dans le même bloc de 1ko que la lecture précédente, La lecture prend un temps T1. Lorsqu'on lit dans un bloc différent, elle prend un temps T2. En pratique, T2 est un temps beaucoup plus long que T1.

On considère qu'on a 10 000 entiers non nuls dans notre grille.

Lorsque l'on parle de cases choisies aléatoirement, il s'agit de cases piochées au hasard. Deux cases choisies aléatoirement l'une après l'autre sont en général loin l'une de l'autre, donc dans des blocs différents.

30) On stocke notre grille dans une simple liste de listes.

On peut initialiser la structure comme ceci :

```
grille = []
for colonne in range(10000):
  rangee = [0] * 10000
  grille.append(rangee)
```

On définit alors les deux fonctions suivantes :

```
def lire(colonne, ligne):
     return grille[colonne][ligne]

def ecrire(colonne, ligne, valeur):
     grille[colonne][ligne] = valeur
```

a) Combien de mémoire cette structure utilise-t-elle ?

b) Quel est le temps maximal, exprimé en fonction de T1 et T2, que prendra la lecture de 10 000 cases choisies aléatoirement ?

c) Combien de temps prendra la lecture des cases d'une même ligne, dans l'ordre de la première à la dernière colonne ?

d) Proposez une modification pour améliorer le temps de lecture des cases d'une même ligne.

31) Pour stocker le contenu de la grille en utilisant moins de mémoire, on va utiliser une liste dont on limitera la longueur à 100 000 cases.

On stocke la valeur aux coordonnées (colonne, ligne) de la grille, dans la case de la liste située à l'indice `(ligne*10000 + colonne) % 100000`.

Selon cette règle, plusieurs cases de la grille, par exemple (0, 0) et (10, 0) peuvent être affectées au même indice de la liste. On parle alors de "collision". Pour gérer les collisions, on fait les deux choses suivantes :

- dans une case de la liste, on stocke non seulement la valeur, mais aussi la colonne et la ligne de la case de la grille où se trouve cette valeur.
- Lors d'une écriture, si la case à l'indice `(ligne*10000 + colonne) % 100000` est déjà occupée, on utilise la première case libre suivante, en repartant au début de la liste si l'on arrive au bout.

Voici le programme qui initialise cette structure, et la fonction d'écriture

```
data = [None] * 100000

def ecrire(colonne, ligne, valeur):
     pos = (ligne*10000 + colonne) % 100000
     while data[pos] is not None:
          pos = (pos + 1) % 100000
     data[pos] = [[colonne, ligne], valeur]
```

a) Écrire la fonction `lire` correspondante.

b) Donner un ordre de grandeur de la quantité de mémoire utilisée par cette structure.

c) Supposons que les cases non nulles de la grille sont réparties un peu partout, au hasard au sein de cette grille.

Combien de temps prendra la lecture des cases d'une même colonne, dans l'ordre de la première à la dernière ligne ?

d) On suppose maintenant que toutes les cases non nulles de la grille sont dans cette même colonne, et ont été écrites dans l'ordre, de la première à la dernière ligne.

Combien de temps prendra la lecture des cases d'une même colonne, dans l'ordre de la première à la dernière ligne ?

32) Fonction de hachage.

Pour éviter d'avoir trop d'itérations lors d'une écriture ou lecture dans le cas où de nombreuses cases consécutives de la liste sont utilisées, on va ici modifier la manière dont on calcule l'indice dans la liste où la valeur d'une case de la grille sera stockée.

L'idée est de répartir au hasard dans la liste, les endroits où l'on stocke les valeurs des cases de la grille. Si on les détermine au hasard, deux cases très proches dans la grille, se retrouveront le plus souvent très éloignées dans la liste.

Pour cela, on va supposer que l'on dispose d'une fonction `hash(x)`, qui pour un entier x donné, retourne un nombre choisi au hasard entre 0 et 1 milliard, mais toujours le même pour un même x. Les valeurs étant choisies au hasard, `hash(x)` et `hash(x+1)` ont en général des valeurs très éloignées.

Voici le programme qui initialise cette structure, et la fonction d'écriture

```
data = [None] * 100000

def ecrire(colonne, ligne, valeur):
      pos = hash(ligne*10000 + colonne) % 100000
      while not data[pos]:
           pos = (pos + 1) % 100000
      data[pos] = [[colonne, ligne], valeur]
```

a) Écrire la fonction `lire` correspondante.

b) Même question qu'en 31.c : supposons que les cases non nulles de la grille sont réparties un peu partout, au hasard au sein de cette grille. Combien de temps prendra la lecture des cases d'une même colonne, dans l'ordre de la première à la dernière ligne ?

c) Même question qu'en 31.d : On suppose maintenant que toutes les cases non nulles de la grille sont dans cette même colonne, et ont été écrites dans l'ordre, de la première à la dernière ligne. Combien de temps prendra la lecture des cases d'une même colonne, dans l'ordre de la première à la dernière ligne ?

33) Hachage multiple

On propose la modification suivante pour l'algorithme précédent :

```
def ecrire(colonne, ligne, valeur):
      pos = hash(ligne*10000 + colonne) % 100000
      while not data[pos][0]:
            pos = hash(pos) % 100000
      data[pos] = [[colonne, ligne], valeur]
```

Expliquer pourquoi cette fonction ne fonctionne pas.

34) Liste de triplets (colonne, ligne, valeur), triés par colonne, puis ligne.

Par exemple si le contenu des premières lignes et colonnes de la grille est :

0	3	0	1
0	0	5	0
2	4	0	0

La liste peut être initialisée par :

```
data = [[0, 2, 2], [1, 0, 3], [1, 2, 4], [2, 1, 5], [3, 0, 1]]
```

a) Décrire un algorithme efficace pour la lecture de toutes les valeurs d'une colonne de la grille initiale, de la première à la dernière ligne. Il n'est pas nécessaire de l'implémenter.

b) Indiquer le coût en temps de votre algorithme.

35) Liste de débuts de lignes.

On stocke un tableau de 100 000 cases dont chaque case contient une paire (coordonnées, valeur).

On stocke par ailleurs un tableau `indiceLigne` de 10 000 cases. Pour chaque ligne, `indiceLigne[ligne]` contient un entier entre 0 et 99 999 choisi aléatoirement.

```
data = [None] * 100000

indiceLigne = []
for pos in range(10000):
  indiceLigne.append(random.randint(0, 99999))

def ecrire(colonne, ligne, valeur)
      pos = (indiceLigne[ligne] + colonne) % 100000
      while not data[pos]:
            pos = (pos + 1) % 100000
      data[pos] = [[colonne, ligne], valeur]
```

a) Écrire la fonction `lire(colonne, ligne)` correspondante.

b) Même question qu'en 31.c : supposons que les cases non nulles de la grille sont réparties un peu partout, au hasard au sein de cette grille. Combien de temps prendra la lecture des cases d'une même colonne, dans l'ordre de la première à la dernière ligne ?

c) Même question qu'en 31.d : On suppose maintenant que toutes les cases non nulles de la grille sont dans cette même colonne, et ont été écrites dans l'ordre, de la première à la dernière ligne. Combien de temps prendra la lecture des cases d'une même colonne, dans l'ordre de la première à la dernière ligne ?

d) Décrire une manière de remplir 10 000 cases de la grille qui soit très lente avec cet algorithme, et indiquer le coût en temps.

Proposer une modification pour améliorer beaucoup ce cas, en modifiant le moins possible les coûts dans les cas 31.c et 31.d.

36) Lignes décomposées en blocs

Chaque ligne est stockée dans un tableau de K cases, où K est un diviseur de 10 000, et où chaque case représente un bloc de 10 000 / K cases de la ligne, est vide si aucune de ces cases ne contient une valeur, et contient une liste de 10 000 / K valeurs sinon.

```
grille = []
for ligne in range(10000):
      rangee = []
      for bloc in range(K):
            rangee.append([])
      grille.append(rangee)

def ecrire(colonne, ligne, value):
      bloc = colonne * K // 10000
      if grille[ligne][bloc] == None:
            grille[ligne][bloc] = [0]*(10000 // K)
      grille[ligne][bloc][colonne % (10000 // K)] = value
```

a) Écrire la fonction `lire` correspondante

b) Donner le coût en mémoire de cette approche, en fonction de K, en supposant que les cases sont remplies aléatoirement.

c) Même question qu'en 31.c : supposons que les cases non nulles de la grille sont réparties un peu partout, au hasard au sein de cette grille. Combien de temps prendra la lecture des cases d'une même colonne, dans l'ordre de la première à la dernière ligne ?

Quelle valeur de K faut-il utiliser pour minimiser ce coût ?

d) Même question qu'en 31.d : On suppose maintenant que toutes les cases non nulles de la grille sont dans cette même colonne, et ont été écrites dans l'ordre, de la première à la dernière ligne. Combien de temps prendra la lecture des cases d'une même colonne, dans l'ordre de la première à la dernière ligne ?

Bases de données

On nous donne une table T1 avec deux colonnes `x` et `y`. Tous les enregistrements sont distincts.

On nous donne une table T2, avec deux colonnes `i` et `z` telles que si on a deux enregistrements (i_1 , z_1) et (i_2 , z_2) alors $(i_1 < i_2)$ implique $(z_1 < z_2)$.

Voici un exemple de contenu de ces tables :

T1	
x	**y**
2	4
3	-5
-3	5
3	-3
-2	-4

T2	
i	**z**
1	5
2	10
3	17
4	21
5	28

On vous demande d'écrire plusieurs requêtes SQL, en utilisant les éléments de syntaxe SQL suivants :

- La sélection d'un sous-ensemble de champs, parmi le sous-ensemble des enregistrements d'une table qui respectent certaines conditions.

  Par exemple :
  ```
  SELECT T1.x, T1.y FROM T1 WHERE T1.x < T1.y AND T1.x > 0
  ```

  Retourne les champs x et y, de l'ensemble des enregistrements de la table T1 dont le champ y est supérieur au champ x, et tels que x est strictement supérieur à 0.

- La jointure simple entre plusieurs tables (pas forcément distinctes), éventuellement en utilisant des alias pour chacune de ces tables.

  Par exemple :
  ```
  SELECT A.x, A.y, B.z
  FROM T1 as A JOIN T2 as B ON (A.x = B.i)
  WHERE A.y > -4
  ```

  Retourne les champs x et y de la table T1 et z de la table T2, pour toutes les paires d'enregistrements des tables T1 et T2 telles que T1.x est égal à T2.i, et que T1.y est strictement supérieur à -4. Voici le résultat sur l'exemple donné plus haut :

x	y	z
2	4	10
3	-3	17

On notera que la condition de jointure peut contenir d'autres types de comparaison, et utiliser les opérateurs OR et AND, par exemple on peut écrire : `ON (A.x < B.i AND A.y > B.i)`

- L'utilisation de la jointure gauche

  Une jointure gauche (`LEFT JOIN`) entre une table A et une table B, va renvoyer tout ce que renvoie une jointure simple, mais également tous les enregistrements de A qui remplissent les conditions de la section `WHERE`, mais auxquels aucun enregistrement de B n'est associé, selon la condition de la jointure.

  Par exemple si l'on reprend l'exemple précédent mais en utilisant un `LEFT JOIN` :

```
SELECT A.x, A.y, B.z
FROM T1 as A LEFT JOIN T2 as B ON (A.x = B.i)
WHERE A.y > -4
```

  La requête retournera aussi des paires (T1.x, T1.y, NULL) pour tous les enregistrements de la table T1 tels que T1.y < -4 mais où il n'existe aucun enregistrement de la table T2 tel que T2.i = T1.x. Voici le résultat :

x	y	xi
2	4	10
-3	5	NULL
3	-3	17

  Les jointures multiples et imbriquées sont autorisées, et par exemple on peut écrire :

```
SELECT *
FROM A JOIN (B LEFT JOIN C ON B.x = C.x) ON A.y = B.y
```

  La requête effectue une jointure entre la table A, et le résultat de la jointure entre B et C.

- Les opérateurs MIN et MAX

  Par exemple:

```
SELECT MIN(T1.x), MAX(T1.x + T1.y) FROM A WHERE T1.y > 10
```

  retourne le minimum des x, et le maximum des x + y, parmi les enregistrements de la table T1 qui ont y > 10.

- Les opérateurs +, -, ABS

Par exemple:
```
SELECT T1.x + T1.y, T1.x - T1.y, ABS(T1.x - T1.y) FROM T1
```
retourne pour chaque enregistrement de la table T1, la somme des champs x et y, leur différence, et la valeur absolue de leur différence.

- Les opérateurs AND, OR

  Par exemple :
  ```
  SELECT * FROM T1 WHERE (x >= 5 AND x <= 10) OR x = 18
  ```
  retourne tous les enregistrements de T1 tels que T1.x est soit entre 5 et 10 inclus, soit égal à 18

- La condition IS NULL

  Par exemple :
  ```
  SELECT * FROM T1 WHERE x IS NULL
  ```
  retourne tous les enregistrements de T1 tels que x vaut NULL

- L'insertion du résultat d'une requête SQL dans une table :

  Par exemple :
  ```
  INSERT INTO T3 SELECT x, y FROM T1 WHERE x < y
  ```
  insère dans la table T3, les champs x et y de tous les enregistrements de T1 qui ont x < y. On suppose que T3 contient exactement deux champs de même type que x et y.

- La modification du contenu d'une table

  Par exemple :
  ```
  UPDATE T1 SET x = 0 WHERE y < 0
  ```
  met à 0 le champ x de T1 pour tous les enregistrements de T1 tels que y est négatif.

Vos requêtes ne peuvent pas contenir :
- GROUP BY
- plusieurs fois le mots clé SELECT dans la même requête
- EXISTS

Pour chacune des questions suivantes, il est possible d'obtenir le résultat en une seule requête. Vous avez cependant le droit de proposer une succession de requêtes, dont la dernière retourne le résultat demandé. Vous pouvez en particulier créer une ou plusieurs tables supplémentaires dont il faudra décrire le schéma (nom et type des champs). On les considère vides au départ. Vous pourrez ensuite la remplir lors d'une première requête, puis l'utiliser dans une deuxième requête.

Par exemple pour calculer les enregistrements de T1 dont les x sont compris entre 0 et 10 vous pouvez faire les deux requêtes suivantes :
```
INSERT INTO temporaire SELECT * FROM T1 WHERE x>=0
SELECT * FROM temporaire WHERE x<=10
```
Avec temporaire qui a le même schéma que T1.

Pour chaque question, expliquer brièvement l'idée des requêtes que vous proposez.
Moins vous utiliserez de requêtes, plus vous aurez de points.

37) Écrire une requête qui retourne la plus grande valeur de x - y, parmi les enregistrements de T1 tels que x est strictement positif, et y strictement négatif.

Sur l'exemple, cette requête retournera 8. Seuls les deuxième et quatrième enregistrements respectent les conditions.

38) Écrire une requête qui retourne la plus grande valeur de x parmi les enregistrements de T1 tels que -x, -y est également un enregistrement de T1.

Sur l'exemple, cette requête retournera 3. Tous les enregistrements sauf le quatrième respectent les conditions.

39) Écrire une requête qui retourne un enregistrement pour chaque x distinct de T1. Chaque enregistrement doit contenir la valeur de x, ainsi que la plus petite valeur de y parmi tous les enregistrements de T1 qui ont cette même valeur de x. On rappelle que l'utilisation du `GROUP BY` n'est pas autorisée.

Sur l'exemple, cette requête retournera :

2	4
3	-5
-3	5
-2	-4

40) Écrire une requête qui calcule et affiche la distance (valeur absolue de la différence) entre les deux z les plus proches parmi tous les z de la table T2.

Sur l'exemple de table T2, cette requête retournera 4, qui est la distance entre 17 et 21.

41) Même question, mais on ne suppose plus que les z sont dans l'ordre croissant dans la table.

42) Même question, mais la table ne contient que les z

43) On nous donne une table T3 constituée de deux colonnes i et w, qui contient un enregistrement pour chaque valeur de i entre 1 et N.

Écrire une requête qui retourne la longueur de la plus longue suite d'enregistrements r_1 ... r_k tels que les i soient des entiers consécutifs ($r_j.i +1 = r_{j+1}.i$) et les w soient décroissants strictement telles que ($r_j.w > r_{j+1}.w$).