ECOLE POLYTECHNIQUE - ESPCI
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2026

MARDI 14 AVRIL 2026
14h00 - 18h00

FILIERES MP-MPI - Epreuve n° 4

INFORMATIQUE A

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.

Bases de données de vecteurs

Les bases de données de vecteurs contiennent des éléments de $\mathbb{R}^d$ pour d un entier strictement positif. Dans ce sujet, on s'intéresse à l'opération suivante que ces bases de données permettent d'effectuer : étant donnés un vecteur $v \in \mathbb{R}^d$ appelé *requête* et une fonction de distance $f : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, renvoyer les plus proches voisins de v dans la base de données pour la distance f. Le sujet étudie différentes implémentations de bases de données de vecteurs.

La partie I s'intéresse au cas des vecteurs de bits avec une application à la reconnaissance automatique d'extraits musicaux : la similarité entre deux tels vecteurs dépend du nombre de bits qu'ils ont en commun. La partie II s'intéresse à une structure de données appelée *liste à sauts* permettant la recherche efficace d'éléments à l'aide de pointeurs à plusieurs étages permettant d'accélérer la recherche. Dans cette partie, on s'intéresse aussi à la recherche de plus proches voisins dans des graphes hiérarchiques où chaque nœud correspond à un vecteur et le voisinage est défini à plusieurs étages. Enfin, la partie III s'intéresse à une approximation de la distance euclidienne $f(x, y) = ||x - y||$ entre deux vecteurs, en plongeant ces vecteurs dans un espace discret de plus petite dimension : les vecteurs de $\mathbb{R}^d$ sont plongés dans $\{0, \ldots, k-1\}^m$ où k est un entier et $m < d$ à l'aide de l'algorithme des k-moyennes.

Les différentes parties sont indépendantes : il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie. À titre d'indication, il n'y a pas de question de code qui nécessite plus d'une douzaine de lignes pour être résolue.

Rappels

Notations. L'ensemble des nombres entiers de 0 à $n-1$ est noté $[n]$. Dans tout ce sujet, on note « log » le logarithme en base 2.

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

Langage OCaml. Ce sujet utilise les entiers, les listes, les files et les tableaux d'OCaml. On utilisera uniquement les fonctions incluses dans la bibliothèque standard du langage. La fonction de bibliothèque `Random.float : float -> float` tire uniformément une valeur entre 0 et un flottant donné en argument.

On rappelle que les entiers OCaml sont représentés sur 63 bits avec le bit de poids fort placé à gauche. Le sujet utilise les opérations suivantes sur la représentation binaire des entiers :

- `a lsl b` décale l'entier `a` de `b` bits vers la gauche, par exemple `1 lsl 5` vaut `32` ;
- `a land b` fait un « et bit à bit » des entiers `a` et `b`, par exemple `6 land 3` vaut `2`.

On rappelle quelques opérations sur les tableaux en OCaml :

- `Array.length tab` renvoie la longueur du tableau `tab` en temps $O(1)$.
- `Array.make k v` crée un tableau de `k` éléments, tous initialisés à `v`, en temps $O(\texttt{k})$.
- La valeur à la case numéro `i` du tableau `tab` est `tab.(i)`. Les cases sont numérotées à partir de zéro et l'accès est fait en temps $O(1)$.
- `Array.of_list liste` renvoie un tableau initialisé avec les valeurs de la liste `liste`, tandis que `Array.to_list t` réalise l'opération inverse ; les deux opérations ont la complexité en temps $O(k)$ avec k la longueur du tableau ou de la liste en argument.
- `Array.make_matrix n m v` initialise une matrice $n \times m$, c'est-à-dire un tableau ayant `n` éléments, chaque élément étant un tableau de `m` valeurs, toutes égales à `v` ; sa complexité en temps est $O(n \times m)$.
- `Array.init_matrix n m (fun i j -> i + j)` (depuis OCaml 5.2) initialise une matrice $n \times m$ dont l'élément (i, j) vaut $i + j$; sa complexité en temps est $O(n \times m)$.
- L'accès à l'élément (i, j) d'une matrice `a` de taille $n \times m$ est `a.(i).(j)` ; il est fait en temps $O(1)$.

On rappelle quelques opérations de base sur les listes en OCaml :

- `List.map f liste` renvoie une liste où la fonction `f` a été appliquée à chaque élément de la liste `liste`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.map` est en $O(k)$, où k est la longueur de `liste`.
- `List.fold_left f acc liste` déroule de la gauche vers la droite le calcul de `f` (qui prend deux arguments) sur la liste, calcul initialisé à `acc`. Par exemple, `List.fold_left f init [b1; ...; bn]` vaut `f (... (f (f init b1) b2) ...) bn`. Sa complexité en temps est $O(n)$, où n est la longueur de la liste, si f est en $O(1)$.

On rappelle enfin les opérations de base sur les files en OCaml. Toutes ces opérations s'effectuent en temps constant.

- `Queue.create ()` renvoie une file vide.
- `Queue.length q` renvoie le nombre d'éléments de la file `q`.
- `Queue.push element q` enfile `element` dans la file `q`.
- `Queue.pop q` défile et renvoie le premier élément de la file `q`.
- `Queue.drop q` (depuis OCaml 5.3) défile le premier élément de la file `q`.
- `Queue.peek q` renvoie le premier élément de la file `q` sans le défiler.

Les trois dernières opérations ci-dessus lèvent l'exception `Queue.Empty` si `q` est vide.

Partie I. Reconnaissance automatique d'extraits musicaux

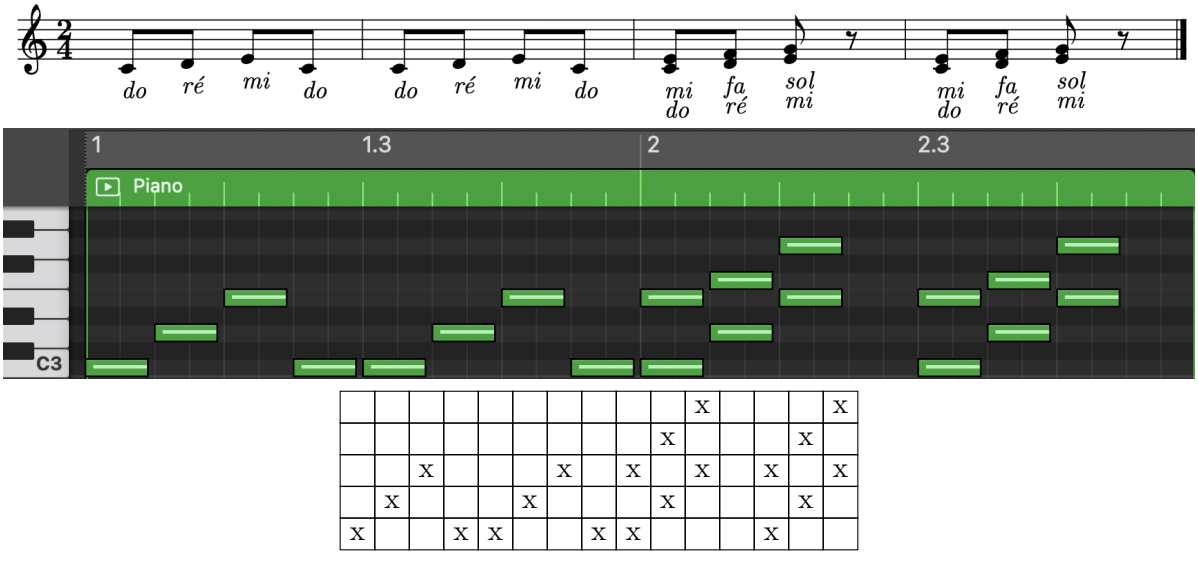
On représente des musiques par leur *spectrogramme*, aussi appelé diagramme temps-fréquence. L'axe des abscisses indique le temps et dépend de la durée de la musique. L'axe des ordonnées représente les fréquences des sons, cf. figure 1. (Il n'est pas nécessaire de savoir lire la musique pour ce sujet.) Pour simplifier, on suppose qu'un spectrogramme d'une musique est représenté en OCaml par une matrice de caractères d'une hauteur constante F, où F représente le nombre de fréquences. La case $(i, j) = (f, t)$ de la matrice vaut le caractère `'x'` si et seulement s'il y a une note de fréquence $i = f$ au temps $j = t$. (Dans la figure 1, les autres cases sont remplies avec le caractère espace `' '`.)

```
type spectrogram = char array array
```

Question I.1. *Pour cette question, on ne demande pas d'écrire de code.* Supposons qu'on cherche à déterminer si une musique est exactement l'extrait d'une autre musique. À quel problème connu se ramènerait-on ? Donner et justifier la complexité dans le pire cas d'une recherche naïve d'une musique de durée D dans une musique de durée T avec $T > D$, en fonction de F le nombre de fréquences possibles.

Les spectrogrammes contiennent principalement des cases vides ; on dit qu'ils sont creux ou *sparse*. Pour économiser de la mémoire, on souhaite représenter les musiques par un autre format appelé *représentation concise* dans la suite de cette partie. Cette représentation est une liste de paires d'entiers (f, t) telles que la case (f, t) du spectrogramme vaut le caractère `'x'`.

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



									x				x
								x				x	
		x				x		x		x		x	
	x				x				x			x	
x			x	x			x	x			x		

FIGURE 1 – Une partition de Frère Jacques et son spectrogramme, sous forme graphique et sous forme de matrice de caractères.

Question I.2. Écrire une fonction

```
val to_sparse : spectrogram -> sparse
```

qui renvoie la représentation concise du spectrogramme en argument.

On définit l'*indice de Jaccard* de deux ensembles non vides et finis A et B, noté $J(A, B)$, comme le quotient entre la taille de leur intersection et la taille de leur union :

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \in [0, 1].$$

J mesure la similarité entre deux ensembles : plus ils sont similaires et plus J est proche de 1.

Dans ce qui suit, on souhaite déterminer la taille de l'intersection de 2 sous-ensembles. Dans les questions de I.3 à I.5, on se restreint aux sous-ensembles de [62], l'ensemble des entiers de 0 à 61. On va représenter un sous-ensemble de [62] comme un vecteur de bits encodé par les 62 bits de poids faible d'un entier OCaml, numérotés de 0 à 61. Le i-ième bit d'un entier n est à 1 si et seulement si l'élément i est dans le sous-ensemble représenté par n.

Question I.3. Écrire une fonction

```
val convert : int list -> int
```

qui convertit une liste d'entiers de [62] sans doublons vers l'entier qui la représente sous forme de vecteur de bits.

Question I.4. Écrire une fonction

```
val inter : int -> int -> int
```

qui, étant donnés deux entiers a et b représentant des sous-ensembles de [62], renvoie un entier qui représente l'ensemble intersection des deux sous-ensembles.

Question I.5. Soit la fonction

```
val size : int -> int
```

qui renvoie le nombre d'éléments du sous-ensemble de [62] encodé dans l'entier en argument.

(a) Écrire le code de la fonction `size` de manière à obtenir une complexité linéaire dans la taille de l'ensemble représenté par son argument. Justifier la complexité de votre code.

(b) Prouver la correction de votre code à l'aide d'un invariant.

Indication : Si `b` est un entier non nul, alors en OCaml `b land (- b)` renvoie un entier dont le seul bit à 1 est celui de poids faible de `b`. Par exemple, pour `b` égal à 6, `b land (-b)` vaut 2.

Question I.6. Soit la fonction

```
val sparse_inter : 'a list -> 'a list -> int
```

qui, étant données deux listes a et b triées dans l'ordre strictement croissant selon la fonction de comparaison `(<)`, renvoie le nombre d'éléments en commun dans ces deux listes. On suppose que les opérations de comparaison du type `'a` sont en $O(1)$.

(a) Écrire le code de la fonction `sparse_inter`. Votre fonction doit fonctionner quel que soit le type des éléments de a et b, que ce soient des entiers ou des d-uplets. De plus, la complexité du code doit être linéaire dans la longueur des deux listes en argument.

(b) Justifier la complexité de votre algorithme.

On pourrait utiliser l'algorithme de la question précédente pour déterminer la similarité entre deux musiques, à partir de leur représentation concise. Mais si un enregistrement contient des fréquences parasites (comme du bruit ambiant), cette valeur de similarité ne sera pas fiable. Dans ce qui suit, on cherche à définir une représentation plus robuste des musiques.

Soit $\Delta > 0$ un entier fixé. On appelle *signature* $S_\Delta(M)$ d'une musique M de durée T l'ensemble de triplets (f_1, f_2, δ) tels que M est passée de la fréquence f_1 à la fréquence f_2 dans un intervalle de temps $\delta > 0$ borné par Δ. Ainsi :

$$(f_1, f_2, \delta) \in S_\Delta(M) \iff 0 < \delta \leq \Delta \text{ et } \exists t \in [T], M \text{ contient } (f_1, t) \text{ et } (f_2, t + \delta) \qquad (1)$$

où lorsqu'on dit « M contient », on fait référence à sa représentation concise.

On représente une valeur signature d'une musique par une liste de triplets d'entiers (f_1, f_2, δ) :

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

Question I.7. Soit la fonction

```
val tosignature : sparse -> int -> signature
```

telle que `tosignature sp delta` prend en arguments une représentation concise `sp` de M et un entier strictement positif représentant Δ, et renvoie la signature $S_\Delta(M)$ de la musique M.

(a) On vous demande de coder cette fonction en commençant par convertir la représentation concise `sp` en un tableau de paires d'entiers à l'aide de `Array.of_list`. La complexité de votre code doit être $O(n^2)$, où n est la longueur de l'argument `sp`.

(b) Justifier la complexité de votre code.

Question I.8. Soit la fonction

```
val signature_size : sparse -> int -> int
```

telle que `signature_size sp delta` calcule la **taille** $|S_\Delta(M)|$ de la signature (donc le **nombre** de triplets (f_1, f_2, δ) de $S_\Delta(M)$) d'une musique M à partir de sa représentation concise `sp` et d'un entier strictement positif représentant Δ. On suppose que la représentation concise `sp` est triée strictement croissant en utilisant l'ordre $(f_1, t_1) < (f_2, t_2)$ si et seulement si $t_1 < t_2$ et pour cette question seulement, on suppose que tous les t_i sont distincts.

(a) Écrire la fonction `signature_size` de manière à calculer, par *un seul parcours* de l'argument `sp`, son résultat. À titre d'information, `signature_size [(1, 1); (5, 5)] 3` doit renvoyer 0 ; `signature_size [(1, 1); (2, 2); (3, 3); (4, 4); (5, 5)] 10` doit renvoyer 10.
Indication : s'aider d'une file.

(b) Justifier brièvement la correction de votre algorithme, puis indiquer et justifier sa complexité.

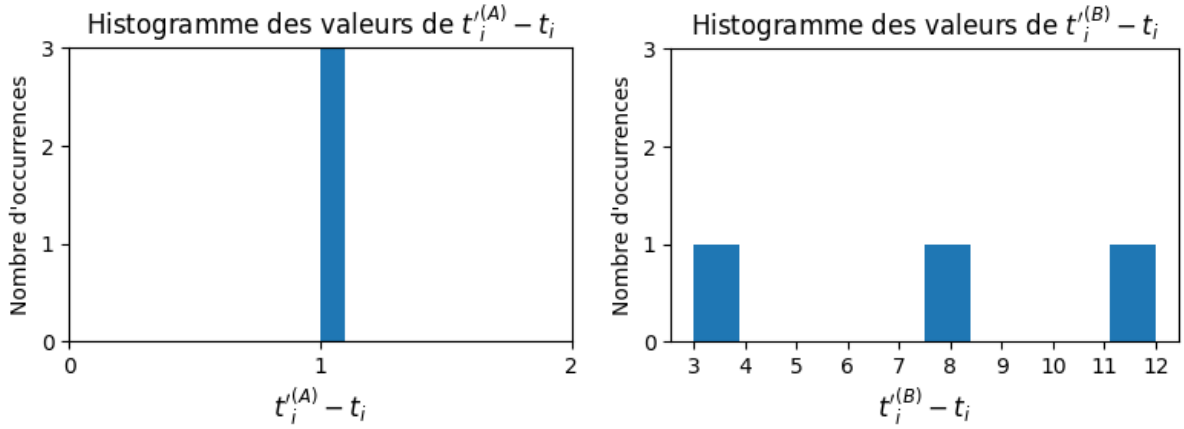


FIGURE 2 – Histogramme des valeurs $t'_i - t_i$ pour trois triplets de M partagés avec deux musiques A et B.

Ainsi, une méthode pour déterminer la similarité entre deux musiques, qui est celle utilisée dans les premières versions des services en ligne qu'on connaît aujourd'hui, consiste à calculer les signatures des musiques puis calculer leur similarité définie par l'indice de Jaccard, à partir du nombre de triplets en commun.

En pratique, un extrait M peut avoir autant de triplets (f_1, f_2, δ) en commun avec une musique A qu'avec une musique B. Pour être plus précis, on peut représenter une musique par un dictionnaire dont les clés sont les triplets (f_1, f_2, δ) et les valeurs sont les temps auxquels ils apparaissent, c'est-à-dire les valeurs des temps t vérifiant l'équation (1) de la définition de signature. Cela nous permet de comparer les temps respectifs où les triplets apparaissent dans chaque musique.

Par exemple, on suppose qu'on a un extrait M et deux musiques A et B tels que $|S_\Delta(M) \cap S_\Delta(A)| = |S_\Delta(M) \cap S_\Delta(B)| = 3$ et les valeurs respectives des dictionnaires pour les clés communes de M, A et B sont (t_1, t_2, t_3), $(t'^{(A)}_1, t'^{(A)}_2, t'^{(A)}_3)$ et $(t'^{(B)}_1, t'^{(B)}_2, t'^{(B)}_3)$. À gauche de la figure 2, on a l'histogramme des valeurs $|t'^{(A)}_i - t_i|$, tandis qu'à droite on a l'histogramme des valeurs $|t'^{(B)}_i - t_i|$, pour $1 \leqslant i \leqslant 3$.

Question I.9. Quelle musique parmi A et B correspond le mieux à l'extrait M et pourquoi ?

Les signatures peuvent avoir beaucoup d'éléments ; ainsi, les bases de données les stockant peuvent prendre beaucoup de mémoire.

Question I.10. Soit E un ensemble à N éléments confondu avec $[N]$, $\mathcal{P}(E)$ l'ensemble des parties de E et $h : E \to [N]$ une fonction de hachage sur les éléments de E. La méthode « minHash » définit, à partir d'une fonction de hachage h sur E, une fonction de hachage $\mathsf{minHash}(h)$ sur les éléments de $\mathcal{P}(E)$ comme ceci :

$$\mathsf{minHash}(h) : \mathcal{P}(E) \to [N] \qquad \mathsf{minHash}(h)(A) = \min_{a \in A} h(a).$$

(a) Pour faciliter l'analyse, on suppose pour cette question que $h : E \to [N]$ est une numérotation aléatoire des éléments de E (une permutation à N éléments car E est confondu

avec $[N]$). Justifier que, si h est la variable aléatoire, alors on a :

$$\forall A, B \subset E, \Pr(\mathsf{minHash}(h)(A) = \mathsf{minHash}(h)(B)) = \frac{|A \cap B|}{|A \cup B|} = J(A, B).$$

(b) En déduire, à la lumière de l'ensemble de cette partie, et en supposant l'accès à une fonction de hachage convenable, une méthode pour déterminer les plus proches musiques voisines d'un extrait audio.

Partie II. Listes à sauts et graphes hiérarchiques navigables

Dans cette partie, on s'intéresse à la recherche d'un nœud dans un graphe orienté ou non orienté tel que la relation de voisinage est définie à différents étages. Dans un premier temps, on étudie une telle structure de données pour des graphes orientés représentant des listes chaînées triées. Ensuite, on s'intéresse à des graphes non orientés quelconques.

Les *skip lists* (listes à sauts) généralisent la représentation des ensembles par des listes chaînées triées. Dans ce sujet, chaque nœud d'une *skip list* contient une valeur entière strictement positive et peut avoir jusqu'à L successeurs où L est le nombre maximal d'étages, numérotés de 0 à $L-1$. Par exemple, la *skip list* de la figure 3 représente la liste $[11, 15, 17, 28, 31, 55, 56, 61]$ à 8 valeurs. Elle contient 9 nœuds dont le nœud factice de valeur 0. Les successeurs du nœud de valeur 15 sont à l'étage 1 le nœud de valeur 31 et à l'étage 0 le nœud de valeur 17.

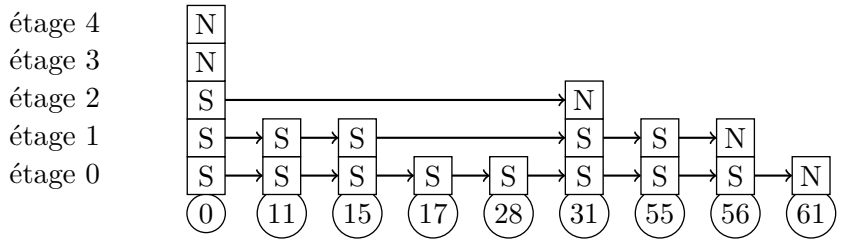


FIGURE 3 – Une *skip list* à $L = 5$ étages, 9 nœuds et $n = 8$ valeurs entières positives (le 0 correspond au nœud factice donc ne compte pas). Chaque nœud contient un tableau de 5 successeurs dont les entrées sont soit `Some node`, étiqueté par S et lié au nœud `node`, soit `None`, étiqueté par N ou parfois non représenté sur l'image.

Le type OCaml `skip_node` d'un nœud est donné ci-dessous. Un nœud `node` a une valeur entière positive `value` et un tableau de successeurs de taille L. Le successeur d'un nœud à l'étage `level` est donné par `node.successors.(level)` : il vaut soit `Some other_node` où `other_node` est un autre nœud de type `skip_node`, soit `None`.

```
type skip_node = {
  value : int;
  mutable successors : skip_node option array;
}
```

On dit qu'une valeur est *présente* à l'étage `level` si et seulement si un nœud pointe vers elle. **On assurera que si une valeur est présente à l'étage ℓ, alors elle est également présente aux étages ℓ' inférieurs à ℓ et elle a un seul prédécesseur à l'étage ℓ.** De plus, les valeurs présentes à un étage forment une sous-liste (forcément triée) de la liste à l'étage 0. Dans l'exemple de la figure 3, la valeur 31 est présente aux étages 0, 1 et 2 mais pas à l'étage 3. Il y a 8 valeurs à l'étage 0, 5 à l'étage 1 et une valeur à l'étage 2.

Une *skip list* a pour attributs :

— un nœud initial (`init_node`) qui est un nœud factice de valeur 0 ;

— un étage (`init_level`) qui est l'étage où toute recherche d'élément commence et qui correspond au plus grand étage où une valeur de la *skip list* est présente. Ainsi, pour une *skip list* `skl`, on pourra commencer la recherche par comparer avec la valeur du nœud `skl.init_node.successors.(skl.init_level)` ;
— un étage maximum (`max_level`), qui vaut $L - 1$ où L est le nombre maximal d'étages que peut contenir la *skip list*.

```
type skip_list = {
  mutable init_node  : skip_node;
  mutable init_level : int; (* Étage où commencer une recherche *)
  mutable max_level  : int  (* Étage maximal possible en mémoire *)
}
```

Par exemple, on peut initialiser une *skip list* ayant le nombre maximal d'étages $L = 5$ (étages de 0 à 4) de la façon suivante :

```
let create_node max_level v = {
  value = v;
  successors = Array.make (max_level + 1) None
}
let skl = { init_node = create_node 4 0; init_level = 0; max_level = 4 }
```

L'étage 0 correspond à la liste chaînée triée usuelle, comme dans l'exemple de la figure 3. Les étages supérieurs permettent d'économiser des étapes lors de la recherche, en pointant vers des nœuds plus loin dans la liste.

Question II.1. Écrire la fonction

```
val skip_length : skip_list -> int
```

qui compte le nombre de valeurs de la *skip list* passée en argument ; on ne comptera pas la valeur du nœud factice.

Pour insérer un nœud dans une *skip list*, on doit d'abord tirer aléatoirement l'étage maximal $\ell \geqslant 0$ où sa valeur sera présente. Ce tirage est effectué en appelant la fonction `random_level` ci-dessous pour une probabilité p ($0 < p < 1$) et un nombre maximal d'étages fixés. La valeur du nœud sera présente exactement aux étages $0, \ldots, \ell$, tout en maintenant la liste chaînée triée.

```
let random_level p max_level =
  let rec loop level =
    if level < max_level && Random.float 1. < p then (* avec probabilité p *)
      loop (level + 1)
    else
      level
  in
  loop 0
```

Question II.2. Si la *skip list* contient n valeurs, combien de valeurs en moyenne sont présentes à l'étage ℓ ? Pour faciliter l'analyse dans cette question, on suppose que l'étage maximal est une valeur si grande que le nombre d'étages est considéré comme non borné.

Question II.3. Soit la fonction

```
val search : skip_list -> int -> skip_node option
```

qui, pour une *skip list* `skl` et une valeur `v`, détermine si la valeur `v` est présente dans `skl`. La fonction renvoie **`Some`** `node` avec `node` le nœud de `skl` qui contient `v` si un tel nœud existe, et **`None`** sinon.

(a) Écrire la fonction `search`. On ne demande pas de le faire naïvement à l'étage 0, mais plutôt de tirer parti des successeurs quand c'est utile.

(b) Justifier brièvement sa correction.

(c) Quelle est sa complexité dans le pire cas ?

On s'intéresse au chemin pris par l'algorithme de recherche entre la position de départ, donnée par le nœud factice `skl.init_node` et l'étage initial `skl.init_level`, et la position finale, donnée par le nœud et l'étage où la recherche s'arrête. Soit $(n_0, \ell_0), (n_1, \ell_1), \ldots, (n_k, \ell_k)$ un tel chemin, représenté par la séquence de paires (nœud, étage) qui sont traversés pendant la recherche. On suppose que l'étage final ℓ_k d'un tel chemin est 0 ; si ce n'est pas le cas, on ajoute à la fin du chemin les paires $(n_k, \ell_k - 1)...(n_k, 0)$. Par exemple, pour la recherche de la valeur 28 dans la *skip list* de la figure 3, le chemin est $(0, 2), (0, 1), (11, 1), (15, 1), (15, 0), (17, 0), (28, 0)$; on obtient le même chemin pour la recherche de la valeur 30.

Question II.4. Dans cette question, on parcourt à l'envers le chemin pris par l'algorithme de recherche, depuis la dernière position (dont l'étage est 0) à la première position. On définit C_ℓ le nombre moyen d'étapes pour aller de la dernière position vers une position à l'étage ℓ dans tout chemin de recherche inversé. À l'aide d'une relation de récurrence, montrer que $C_\ell = \ell/p$, où $0 < p < 1$ est le paramètre utilisé pour monter d'un étage dans la fonction `random_level` ci-dessus appelée lors de l'insertion d'un élément. Pour cette question, et la suivante, on suppose que le nombre d'étages n'est pas borné.

Question II.5. En déduire que le nombre d'étapes en moyenne d'une recherche d'une valeur dans une *skip list* à n valeurs est $O(\log n)$.

Question II.6. Soit la fonction d'insertion

```
val insert : skip_list -> int -> unit
```

qui, pour une *skip list* `skl` et une valeur `v` qui est absente de `skl`, insère la valeur `v` dans `skl`.

(a) Combien de successeurs suffit-il de modifier au maximum ?

(b) Écrire la fonction `insert` ; il n'est pas nécessaire de recopier les parties qui sont redondantes avec la fonction `search` mais surtout d'indiquer ce qui diffère. Vous pourrez également supposer que vous avez déjà accès aux fonctions `create_node` et `random_level` ci-dessus. On ne vous demande pas de modifier la valeur `init_level` de la *skip list*.

On s'intéresse à présent à la représentation d'une base de données de vecteurs de $\mathbb{R}^d$ sur laquelle est définie une relation de voisinage. Un vecteur sera représenté par un tableau de flottants de type `vector`. Les n vecteurs de la base de données sont stockés dans une matrice `db` de taille $n \times d$ et de type `matrix` :

```
type vector = float array
type matrix = vector array
val db : matrix
```

Nous utiliserons un *graphe non orienté hiérarchique* pour représenter la relation de voisinage, c'est-à-dire un graphe structuré en étages, de manière similaire aux *skip lists*. La définition en OCaml du type `hgraph` est :

```
type hgraph = int list array array
```

Un graphe hiérarchique $G = (V_\ell, E_\ell)_{0 \leqslant \ell < L}$ à n nœuds est tel qu'à l'étage $\ell \in \{0, \ldots, L-1\}$ les nœuds sont V_ℓ et les arêtes E_ℓ. Comme pour les *skip lists*, si un nœud est présent à un étage alors il est présent aux étages inférieurs : $[n] = V_0 \supseteq V_1 \supseteq \cdots \supseteq V_{L-1}$.

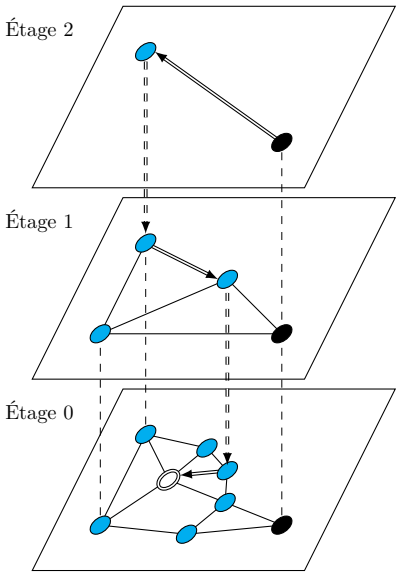


FIGURE 4 – Recherche dans un graphe hiérarchique à trois étages. Le nœud de départ, à l'étage 2 est en noir et le nœud recherché à l'étage 0 en double cercle. Les flèches indiquent le chemin suivi par la recherche : en pointillé pour un changement d'étage et en plein pour un changement de nœud parmi les voisins au même étage.

En OCaml, on représente un tel graphe hiérarchique par une matrice de listes d'adjacence de taille $n \times L$. L'entrée `g.(i).(level)` est la liste des voisins du nœud `i` à l'étage `level`, tous dans $[n]$. On a en outre que la relation « est voisin de » est symétrique et le nœud `i` est dit *présent* à l'étage `level` si et seulement si `g.(i).(level)` n'est pas vide. Les coordonnées (en $\mathbb{R}^d$) du nœud `i` sont données par `db.(i)`.

On suppose qu'on a accès à une fonction de distance entre les vecteurs

```
val dist : vector -> vector -> float
```

dont la complexité en temps est en $O(d)$.

Un exemple de graphe hiérarchique est donné dans la figure 4. Il y a 2 nœuds présents à l'étage 2 et 4 à l'étage 1. La figure illustre avec des flèches doubles les étapes d'une méthode gloutonne pour la recherche du plus proche voisin d'un vecteur requête `query` (non indiqué sur la figure). La méthode consiste à partir d'un nœud de départ, par exemple celui d'indice 0 indiqué en noir, et à chaque tour prendre parmi les voisins de l'étage en cours celui qui réduit strictement la distance à `query`. Pour le voisin ainsi trouvé, soit on est à l'étage 0 et on renvoie l'indice du vecteur dans `db`, soit on descend à l'étage inférieur (toujours possible) et on continue la recherche gloutonne.

Question II.7. Soit la fonction

```
val closest : vector -> hgraph -> int
```

qui, pour un vecteur requête `query` et un graphe hiérarchique `g` renvoie l'indice `i` dans `db` du vecteur candidat le plus proche de `query` en utilisant la méthode de recherche gloutonne.

(a) Écrire la fonction `closest`.

(b) Quelle est la complexité en temps de votre implémentation en fonction du nombre de nœuds n, du nombre total d'arêtes $|E| = \sum_\ell |E_\ell|$, du nombre d'étages L et de la dimension d des vecteurs ?

(c) A-t-on besoin de marquer les nœuds pour que votre algorithme termine ? Justifiez.

Partie III. Quantification de produit

On s'intéresse maintenant à répondre à une requête sur la base de données de vecteurs en utilisant un algorithme d'approximation, basé sur la méthode des k-moyennes. Dans un premier temps, on implémente cet algorithme. Ensuite, on s'en sert pour approcher les distances entre vecteurs de grande dimension. Enfin, on s'intéresse au gain offert par un pré-calcul.

On dispose de n vecteurs de dimension d dans une matrice `db` et d'une fonction `dist2` telle que `dist2 p1 p2` calcule en $O(d)$ le **carré** de la distance euclidienne entre les vecteurs `p1` et `p2`.

```
type vector = float array        (* vecteur *)
type matrix = vector array       (* tableau de vecteurs *)
type square = float array array (* matrice carrée n × n *)

val dist2  : vector -> vector -> float
```

Question III.1. En utilisant `Array.init_matrix`, écrire une fonction

```
val compute_all_dist2 : matrix -> square
```

qui, pour un tableau de n vecteurs `db` en argument, calcule la matrice carrée (de taille $n \times n$) des carrés des distances euclidiennes $D_{ij} = ||p_i - p_j||^2$ où p_i est le i-ième élément de `db`.

Question III.2. Quelle est la complexité de déterminer le plus proche voisin parmi les n éléments de `db` d'un vecteur `query` donné en entrée ? On ne demande pas d'écrire le code.

Pour implémenter l'algorithme des k-moyennes, nous utiliserons deux valeurs auxiliaires :

- `centroids : matrix` est une matrice de taille $k \times d$ qui contient k vecteurs de dimension d, qu'on appelle des *centroïdes*.
- `centroid_of : int array` est un tableau de taille n tel que `centroid_of.(i)` est l'indice j ($0 \leq j < k$) de l'élément de `centroids` qui est le plus proche de `db.(i)` pour la distance euclidienne.

Question III.3. Écrire une fonction

```
val pick : matrix -> matrix -> int array -> unit
```

telle que, pour un tableau de n vecteurs `db` et un tableau de centroïdes `centroids`, `pick db centroids centroid_of` calcule le centroïde le plus proche de chaque vecteur de `db`, en mettant à jour en place le tableau `centroid_of`.

Pour la suite du sujet, on suppose que l'on a accès aux fonctions suivantes :

```
val vector_sum : vector -> vector -> vector (* somme de deux vecteurs *)
val matrix_sum : square -> square -> square (* somme de deux matrices *)
val vector_div : vector -> float -> vector  (* division par un scalaire *)
```

Pour deux vecteurs u et v, l'appel à `vector_sum u v` renvoie le vecteur $u+v$. Pour deux matrices A et B, l'appel à `matrix_sum a b` renvoie la matrice $A + B$. Enfin, l'appel à `vector_div u z` renvoie le vecteur dont la i-ième coordonnée est u_i/z où u est un vecteur et z est un scalaire flottant.

Question III.4. Écrire une fonction

```
val update : int -> matrix -> int array -> matrix -> unit
```

telle que `update k db centroid_of centroids` calcule les k nouveaux centroïdes (et les modifie en place), en fonction des n vecteurs `db` et l'affectation `centroid_of` des vecteurs aux centroïdes.

Question III.5. Est-il possible que votre code mette à jour moins de k centroïdes ? Justifiez votre réponse.

Question III.6. Écrire une fonction

```
val kmeans : matrix -> int -> int -> int array * matrix
```

telle que `kmeans db k n_iter` effectue `n_iter` itérations de l'algorithme des k-moyennes, et renvoie la paire `(centroid_of, centroids)` obtenue. Pour simplifier, on initialisera les centroïdes aux k premiers vecteurs de la base `db`.

À présent, on vise à améliorer le temps d'exécution de l'algorithme de calcul de distances de la question III.1 pour des vecteurs de très grande dimension d. Pour cela, on divise chaque vecteur en m paquets disjoints de même taille d/m où m est un diviseur de d. L'algorithme des k-moyennes est appliqué sur chaque paquet de taille $n \times d/m$ (les d/m premières dimensions de tous les vecteurs, puis les d/m suivantes et ainsi de suite). Un vecteur original (de dimension d) est approché par la liste des centroïdes de ses m paquets. Formellement, cet algorithme calcule une fonction

$$q : \mathbb{R}^d \to [k]^m$$

telle que $q(p_i) = (c_1(p_i), \ldots, c_m(p_i))$ où $c_\ell : \mathbb{R}^d \to [k]$ affecte à un vecteur l'indice du centroïde de son ℓ-ième paquet parmi les ℓ-ième paquets de tous les vecteurs de `db`. Le principe utilisé par cette méthode, c'est-à-dire employer les centroïdes comme approximation des parties d'un vecteur s'appelle *quantification de produit*. Dans ce qui suit, on va calculer la distance approchée entre deux vecteurs quelconques de la base `db`.

On suppose que l'on dispose d'une fonction

```
val submatrix : matrix -> int -> matrix list
```

qui, pour une matrice `db` (de dimension $n \times d$) et une valeur m (diviseur de d), renvoie une liste de m matrices de dimension $n \times (d/m)$, telle que la ℓ-ième matrice a comme i-ème ligne le ℓ-ième paquet du vecteur `db.(i)`.

Question III.7. Écrire la fonction

```
val approx_dist : matrix -> int -> int -> int -> square
```

telle que `approx_dist db k n_iter m`, pour un tableau de n vecteurs `db`, calcule la matrice (de taille $n \times n$) des distances approchées

$$D'_{ij} = \sum_{\ell=1}^{m} d_\ell(c_\ell(p_i), c_\ell(p_j))$$

où pour toute paire d'entiers (a, b) dans l'ensemble $[k] \times [k]$, $d_\ell(a, b)$ est le carré de la distance euclidienne entre le a-ième centroïde et le b-ième centroïde dans le ℓ-ième paquet. On vous encourage à utiliser plusieurs fonctions. Ici, `n_iter` désigne encore le nombre d'itérations de l'algorithme de k-moyennes à faire sur chaque paquet.

On souhaite pré-calculer dans un dictionnaire les valeurs de $d_\ell(a, b)$ pour tous $(\ell, a, b) \in \{1, \ldots, m\} \times [k] \times [k]$. On suppose que les valeurs $c_\ell(p_i)$ sont pré-calculées et stockées pour tous les paquets et tous les vecteurs, et que les k centroïdes sont connus et fixés pour tous les paquets.

Question III.8.

(a) Quelle est la complexité en temps du pré-calcul du dictionnaire en fonction de k, d et m ?

(b) Borner le nombre de valeurs distinctes que peuvent prendre les D'_{ij} en fonction de k et m lorsque $(i, j) \in [n]^2$.

(c) Une fois le pré-calcul effectué, quelle est la complexité de calculer les plus proches voisins du vecteur `db.(i)` parmi les vecteurs de `db` pour la distance approchée ?

(d) Même question pour calculer les plus proches voisins parmi `db` pour la distance approchée d'un nouveau vecteur de coordonnées $p \in \mathbb{R}^d$.

Fin du sujet.