A2024 – INFO MP

**ÉCOLE DES PONTS PARISTECH,
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 2024

Durée de l'épreuve : 3 heures

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

ÉPREUVE D'INFORMATIQUE MP

*Cette épreuve concerne uniquement les candidats de la filière MP.
Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :*

INFORMATIQUE - MP

L'énoncé de cette épreuve comporte 7 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.

Préliminaires

L'épreuve est formée d'un problème unique, constitué de 25 questions, qui porte sur l'algorithmique de quelques résultats mathématiques reliés à la notion de relation d'ordre. Le problème est divisé en trois sections qui peuvent être traitées séparément, à condition d'avoir lu les définitions introduites jusqu'à la question traitée.

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`).

Des rappels des extraits du manuel de documentation de OCaml sont reproduits en annexe. Ces derniers portent sur le module `Array`.

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 mais 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.

1. Relation d'ordre et tri topologique

Nous fixons un ensemble fini V et appelons n son cardinal. Nous notons systématiquement v_0, v_1, …, v_{n-1} les éléments distincts de V.

Définition : Nous appelons *relation d'ordre sur V* toute relation binaire $\trianglelefteq$ qui est

1. *réflexive*, c'est-à-dire telle que, pour tout indice i compris entre 0 et $n-1$, nous avons la relation $v_i \trianglelefteq v_i$,
2. *antisymétrique*, c'est-à-dire telle que, pour tous indices i et j compris entre 0 et $n-1$, si les relations $v_i \trianglelefteq v_j$ et $v_j \trianglelefteq v_i$ sont vérifiées, alors nous avons l'égalité $v_i = v_j$
3. *transitive*, c'est-à-dire que, pour tous indices i, j et k compris entre 0 et $n-1$, si les relations $v_i \trianglelefteq v_j$ et $v_j \trianglelefteq v_k$ sont vérifiées, alors nous avons encore la relation $v_i \trianglelefteq v_k$.

Un ensemble fini muni d'une relation d'ordre s'appelle un *ensemble ordonné* et se note $(V, \trianglelefteq)$. Nous réservons le symbole $\leqslant$ pour désigner l'ordre usuel sur les entiers.

Indication OCaml : Nous représentons tout ordre sur un ensemble de cardinal n par une matrice carrée r de dimension $n \times n$, de type

1.
```
type order = bool array array
```

où le coefficient `r.(i).(j)` vaut `true` si et seulement si la relation $v_i \trianglelefteq v_j$ est vérifiée.

☐ 1 – Écrire une fonction OCaml `check_reflexivity (r:order) : bool` qui vérifie si la relation r est une relation réflexive et qui relève du paradigme de *programmation impérative*.

☐ 2 – Écrire une fonction OCaml `check_reflexivity_bis (r:order) : bool` qui vérifie également si la relation r est une relation réflexive mais qui, par contraste avec la question 1, relève du paradigme de *programmation fonctionnelle*.

Dans le reste du sujet, le choix du paradigme de programmation est libre.

☐ 3 – Écrire une fonction OCaml `check_antisymmetry (r:order) : bool` qui vérifie si la relation r est une relation antisymétrique.

☐ 4 – Écrire une fonction OCaml `check_transitivity (r:order) : bool` qui vérifie si la relation r est une relation transitive.

☐ 5 – Écrire une fonction OCaml `is_order (r:order) : bool` qui vérifie si la relation r est une relation d'ordre en s'appuyant sur les questions 1, 3 et 4.

Définition : Nous appelons *graphe d'un ensemble ordonné* $(V, \trianglelefteq)$, ou simplement graphe de la relation d'ordre $\trianglelefteq$, le graphe orienté $G = (V, E)$ dont l'ensemble des arcs est

$$E = \{(v_i, v_j) \text{ où } (i,j) \in [\![0, n-1]\!]^2 \text{ et } v_i \trianglelefteq v_j\}.$$

Nous rappelons que le *degré sortant* d'un sommet v désigne le nombre d'arcs dont l'extrémité initiale vaut v.

☐ 6 – Écrire une fonction OCaml `outdegrees (r:order) : int array` dont la valeur de retour est le tableau d des degrés sortants des sommets du graphe de la relation d'ordre r. Autrement dit, pour tout indice i compris entre 0 et $n-1$, l'entier $d[i]$ est le degré sortant du sommet v_i.

☐ 7 – Écrire une fonction OCaml `argmin (d:int array) : int` dont la valeur de retour est un indice valide i tel que, pour tout indice j compris entre 0 et la longueur du tableau d'entiers d exclue, on a l'inégalité $d[i] \leqslant d[j]$.

Définition : Étant donné un ensemble ordonné $(V, \trianglelefteq)$ de cardinal n, nous appelons *tri topologique* de V toute permutation τ des entiers de l'intervalle $[\![0, n-1]\!]$ telle que, pour tous indices i et j compris entre 0 et $n-1$, si l'on a la relation $v_i \trianglelefteq v_j$, alors on a l'inégalité $\tau(i) \leqslant \tau(j)$. Pour tous indices i compris entre 0 et $n-1$, nous disons dans ce cas que le sommet v_i est le sommet de numéro $\tau(i)$.

☐ 8 – Montrer qu'il existe une constante entière δ_0 telle que, pour tout tri topologique τ, le sommet de numéro $n-1$, à savoir le sommet $v_{\tau^{-1}(n-1)}$, est de degré sortant δ_0 et le degré sortant des autres sommets est supérieur ou égal à δ_0.

☐ 9 – Écrire une fonction OCaml `topological_sort (r:order) : int array` dont la valeur de retour est un tri topologique de l'ensemble ordonné par r. Il est attendu que la fonction attribue les numéros par ordre décroissant et que, en cours d'exécution, elle s'appuie sur la question 8 pour identifier le prochain sommet à numéroter.

☐ 10 – Énoncer un ensemble d'invariants qui démontre que la fonction `topological_sort` de la question 9 est correcte. Il pourra s'agir d'*invariants de boucle* dans le cas d'un programme itératif ou de *postconditions* dans le cas d'un programme récursif.

☐ 11 – Exprimer la complexité en temps de la fonction `topological_sort r` en fonction du nombre d'éléments n ordonnés par la relation r. Dire comment on qualifie une telle complexité en fonction de l'espace en mémoire utilisé pour représenter l'argument d'entrée r.

2. Chaînes et antichaînes

Définition : Étant donné un ensemble ordonné $(V, \trianglelefteq)$, nous appelons *chaîne* toute partie C de V totalement ordonnée, c'est-à-dire, telle que pour tous éléments c et $c' \in C$, nous avons la relation $c \trianglelefteq c'$ ou la relation $c' \trianglelefteq c$.

Indication OCaml : Nous représentons une chaîne par le type `int list`.

Nous donnons le code suivant

```
2.  let rec is_chain (r:order) (c:int list) : bool =
3.      match c with
4.      | [] -> true
5.      | v::cc -> List.for_all (fun x -> x > v || x < v) cc
6.                 && is_chain r cc
```

où nous avons recouru à la fonction `for_all` du module `List` ainsi décrite dans la documentation de OCaml :

> `val for_all : ('a -> bool) -> 'a list -> bool`
> `for_all f [a1; ...; an]` checks if all elements of the list satisfy the predicate `f`. That is, it returns `(f a1) && (f a2) && ... && (f an)` for a non-empty list and `true` if the list is empty.

☐ 12 – Confirmer ou réfuter la spécification suivante : la fonction `is_chain r c` codée comme ci-dessus teste si la liste de sommets c est une chaîne au sens de la relation d'ordre r.
Dans le premier cas, on fournira une démonstration ; dans le second cas, on modifiera la fonction `is_chain` afin qu'elle soit correcte.

☐ 13 – Déterminer la complexité en temps de la fonction `is_chain r c` codée comme ci-dessus en fonction de la longueur γ de la liste c. Justifier le calcul.

Nous nous proposons d'écrire une fonction `is_chain_bis r c` obéissant à la spécification de la question 12 et de complexité en temps $O(\gamma \log \gamma)$ où γ est la longueur de la liste c. À cette fin, nous nous préparons à introduire une structure de donnée mutable à capacité bornée permettant de stocker une collection d'éléments de $(V, \trianglelefteq)$ à travers l'interface suivante :

```
7.  type hp
8.  val init : order -> int -> hp
9.  val is_empty : hp -> bool
10. val push : hp -> int -> bool
11. val pop : hp -> bool
```

L'appel `init r gamma` doit permet d'initialiser une collection vide, informée de la relation d'ordre r, et de capacité γ. L'appel `is_empty q` doit indiquer si la collection q est vide. L'appel `push q i` doit insérer l'élément $v_i \in V$ à la collection q ; l'appel `pop q` doit retirer un élément de la collection q. Nous envisageons que les appels à `push` et de `pop` puissent échouer, ce qui est signalé par la valeur de retour de ces fonctions. Nous écrivons la fonction `is_chain_bis r c` sous la forme suivante :

```
12. let is_chain_bis (r:order) (c:int list) : bool =
13.     let q = init r (List.length c) in
14.     let rec insert c : bool =
15.         match c with
16.         | [] -> true
17.         | hd::tl -> (push q hd) && insert tl
18.     in
19.     let rec extract_all () : bool =
20.         if is_empty q then true
21.         else (pop q) && (extract_all ())
22.     in
23.     (insert c) && (extract_all ())
```

Cette fonction insère les éléments de la liste c à une collection initialement vide q, de type `hp`, puis les retire de cette collection q.

☐ 14 – Compléter la description du type `hp` en nommant une structure de donnée, dont on rappellera brièvement l'invariant de bonne constitution, de sorte que la fonction `is_chain_bis r c` obéisse à la spécification de la question 12 et s'exécute en temps $O(\gamma \log \gamma)$, où γ est la longueur de la liste c. Justifier la réponse.

Définition : Étant donné un ensemble ordonné $(V, \trianglelefteq)$, nous appelons *antichaîne* toute collection de sommets $A \subseteq V$ dont les éléments sont deux à deux incomparables, c'est-à-dire, telle que pour tous éléments distincts a et $a' \in A$, aucune des relations $a \trianglelefteq a'$ ou $a' \trianglelefteq a$ n'est vérifiée.

Indication OCaml : Nous représentons une antichaîne par le type `int list`.

☐ 15 – Modifier la fonction `is_chain` introduite à la question 12 afin d'écrire une fonction OCaml `is_antichain` obéissant à la spécification : la fonction `is_antichain r a` teste si la liste de sommets a est une antichaîne pour la relation d'ordre r. En déduire que la fonction `is_antichain` admet pour complexité en temps la même complexité que celle obtenue à la question 13.

☐ 16 – Confirmer ou réfuter l'existence d'une fonction `is_antichain_bis r a` obéissant à la même spécification que celle de la question 15 et de complexité en temps $O(\alpha \log_2 \alpha)$, où α est la longueur de la liste a.

3. Couverture par des chaînes

Définition : Nous disons qu'une famille finie $(C_\ell)_{1\leqslant\ell\leqslant\lambda}$ de chaînes de l'ensemble ordonné $(V, \trianglelefteq)$ est une *couverture* si la réunion $\bigcup_{1\leqslant\ell\leqslant\lambda} C_\ell$ est égale à l'ensemble V.

☐ 17 – Soit $(C_\ell)_{1\leqslant\ell\leqslant\lambda}$ une couverture de l'ensemble ordonné $(V, \trianglelefteq)$ et soit A une antichaîne de $(V, \trianglelefteq)$ de cardinal α. Démontrer l'inégalité

$$\alpha \leqslant \lambda.$$

Définition : Étant donné un ensemble ordonné $(V, \trianglelefteq)$, nous appelons *graphe des poursuites d'un ensemble ordonné* $(V, \trianglelefteq)$ le graphe biparti non orienté $P = (W_0 \sqcup W_1, F)$ où les ensembles de sommets W_0 et W_1 sont deux copies distinctes de V, dont on note les éléments

$$W_0 = \{w_i\,;\ i \in [\![0, n-1]\!]\} \qquad \text{et} \qquad W_1 = \{w_{n+i}\,;\ i \in [\![0, n-1]\!]\},$$

et dont l'ensemble des arêtes est

$$F = \{(w_i, w_{n+j}) \text{ où } (i,j) \in [\![0, n-1]\!]^2,\ i \neq j,\ \text{ et } v_i \trianglelefteq v_j\}.$$

Indication OCaml : Nous représentons un graphe biparti à $2n$ sommets par un tableau de longueur $2n$ contenant des listes de voisins

```
24.  type bipartite = int list array
```

Chaque arrête est notée deux fois en mémoire.

☐ 18 – Écrire une fonction OCaml `bipartite_of_order (r:order) : bipartite` dont la valeur de retour est le graphe des poursuites tiré de la relation d'ordre r.

Définition : Nous appelons *couplage* tout ensemble d'arêtes dont les extrémités sont distinctes.

☐ 19 – Pour tout couplage M du graphe des poursuites $P = (W_0 \sqcup W_1, F)$, nous construisons la partition $(C_\ell)_{1\leqslant\ell\leqslant\lambda}$ de l'ensemble V telle que

1. pour tous indices i et j compris entre 0 et $n-1$, si le couple $(w_i, w_{n+j}) \in W_0 \times W_1$ est une arête de M, alors les deux éléments v_i et v_j appartiennent à une même part C_ℓ,
2. le cardinal λ est aussi grand que possible.

Montrer que pour tout indice ℓ compris entre 1 et λ, l'ensemble C_ℓ est une chaîne de $(V, \trianglelefteq)$.

Indication OCaml : Nous représentons un couplage dans le graphe des poursuites $P = (W_0 \sqcup W_1, F)$ par un tableau m de longueur n, de type `int array`, tel que $m[j] = i$ si le couple $(w_i, w_{n+j}) \in W_0 \times W_1$ est une arête du couplage et $m[j] = -1$ si le sommet $w_{n+j} \in W_1$ n'est pas l'extrémité d'une arrête du couplage.

☐ 20 – Déduire de la question 19 une fonction OCaml `chains_of_matching (m:int array) : int list list` dont la valeur de retour est une liste de chaînes disjointes tirée du couplage m.

☐ 21 – Une couverture $(C_\ell)_{1 \leqslant \ell \leqslant \lambda}$ de l'ensemble ordonné $(V, \trianglelefteq)$ par des chaînes disjointes étant fixée, expliquer comment construire un couplage M tel que la construction de la question 19 conduit à la partition $(C_\ell)_{1 \leqslant \ell \leqslant \lambda}$.

☐ 22 – Montrer que la construction de la question 19 fournit une couverture de l'ensemble V par un nombre minimum de chaînes si et seulement si elle s'applique à un couplage M de cardinal maximum.

☐ 23 – Écrire une fonction OCaml `maximum_matching (p:bipartite) : int array` qui calcule un couplage de cardinal maximum dans le graphe biparti p.

☐ 24 – Écrire une fonction OCaml `minimum_cover (r:order) : int list list` qui calcule une couverture par une famille de chaînes disjointes de cardinal minimum.

☐ 25 – Déterminer la complexité en temps de la fonction `minimum_cover r` en fonction du nombre d'éléments n ordonnés par la relation r.

A. Annexe : aide à la programmation en OCaml

Opérations sur les tableaux : Le module `Array` offre les fonctions suivantes :

- `length : 'a array -> int`
  Return the length (number of elements) of the given array.
- `make : int -> 'a -> 'a array`
  `Array.make n x` returns a fresh array of length n, initialized with x. All the elements of this new array are initially physically equal to x (in the sense of the == predicate). Consequently, if x is mutable, it is shared among all elements of the array, and modifying x through one of the array entries will modify all other entries at the same time.
- `make_matrix : int -> int -> 'a -> 'a array array`
  `Array.make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension $dimx$ and second dimension $dimy$. All the elements of this new matrix are initially physically equal to e. The element (x, y) of a matrix m is accessed with the notation `m.(x).(y)`.
- `init : int -> (int -> 'a) -> 'a array`
  `Array.init n f` returns a fresh array of length n, with element number i initialized to the result of $f(i)$. In other terms, `init n f` tabulates the results of f applied to the integers 0 to $n-1$.
- `copy : 'a array -> 'a array`
  `Array.copy a` returns a copy of a, that is, a fresh array containing the same elements as a.
- `mem : 'a -> 'a array -> bool`
  `mem a l` is true if and only if a is structurally equal to an element of l (i.e. there is an x in l such that `compare a x = 0`).
- `for_all : ('a -> bool) -> 'a array -> bool`
  `Array.for_all f [|a1; ...; an|]` checks if all elements of the array satisfy the predicate f. That is, it returns `(f a1) && (f a2) && ... && (f an)`.
- `exists : ('a -> bool) -> 'a array -> bool`
  `Array.exists f [|a1; ...; an|]` checks if at least one element of the array satisfies the predicate f. That is, it returns `(f a1) || (f a2) || ... || (f an)`.
- `map : ('a -> 'b) -> 'a array -> 'b array`
  `Array.map f a` applies function f to all the elements of a, and builds an array with the results returned by f : `[| f a.(0); f a.(1); ...; f a.(length a - 1) |]`.
- `mapi : (int -> 'a -> 'b) -> 'a array -> 'b array`
  Same as `Array.map`, but the function is applied to the index of the element as first argument, and the element itself as second argument.
- `iter : ('a -> unit) -> 'a array -> unit`
  `Array.iter f a` applies function f in turn to all the elements of a. It is equivalent to `f a.(0); f a.(1); ...; f a.(length a - 1); ()`.
- `iteri : (int -> 'a -> unit) -> 'a array -> unit`
  Same as `Array.iter`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

D'après `https://v2.ocaml.org/api/Array.html`

Fin de l'épreuve