Concours e3a-Polytech, France

Session 2022, filière MP

Informatique

Durée : 4 heures

Ce que traite le sujet

- Partie I : automates
- Partie II : autour des tas
- Partie III : autour de l'énumération des fractions positives

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/e3a/mp/2022/info

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

CONCOURS COMMUN INP

ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE MP

INFORMATIQUE

Durée : 4 heures

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

RAPPEL DES CONSIGNES

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

Les calculatrices sont interdites.

Le sujet est composé de trois parties, toutes indépendantes.

Partie I - Automates

L'objectif de cette partie est de proposer quelques éléments autour d'un automate, dit augmenté, construit autour d'un automate fini non déterministe et d'un sous-ensemble d'états.

Définition 1 (Automate fini non déterministe)
Un *automate fini non déterministe* est un quintuplet $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ avec :
- Q un ensemble fini non vide d'états, de cardinal $|Q|$;
- Σ un alphabet ;
- $I \subset Q$ l'ensemble des états initiaux ;
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ une fonction de transition : si $q \in Q$ et $a \in \Sigma$, $\delta(q, a)$ désigne l'ensemble des états q' de Q tels qu'il existe une transition étiquetée par a de q vers q' ;
- $F \subset Q$ l'ensemble des états finaux.

Définition 2 (Automate augmenté)
Soient $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ un automate fini non déterministe et $O \subset Q$ un sous-ensemble d'états de Q. On appelle *automate augmenté* par O la paire $(\mathcal{A}, O)$, que l'on notera par la suite $\mathcal{A}_O$.

La notion de calcul est la même entre $\mathcal{A}$ et $\mathcal{A}_O$: un calcul dans $\mathcal{A}$ est une séquence d'états $q_1 \cdots q_n$ de Q telle qu'il existe toujours au moins une transition entre deux états successifs de cette séquence. Le mot construit par concaténation des étiquettes de chacune des transitions est alors reconnu par l'automate.
L'ensemble O introduit la notion de *calcul réussi au seuil* s.

Définition 3 (Calcul réussi au seuil s)
Soit $\mathcal{A}_O$ un automate augmenté. Un calcul passant par les états $q_i \in Q, i \in [\![0, n]\!]$ est dit *réussi au seuil* $s \in \mathbb{N}$ si :

i) $q_0 \in I$;

ii) $q_n \in F$;

iii) pour tout $i \geq 0$ tel que $i + s \leq n$, $\{q_i \cdots q_{i+s}\} \cap O \neq \emptyset$.

Définition 4 (Langage reconnu par un automate augmenté au seuil s)
Soit $\mathcal{A}_O$ un automate augmenté et $s \in \mathbb{N}$. Le *langage reconnu par* $\mathcal{A}_O$ *au seuil* s, noté $L_s(\mathcal{A}_O)$, est l'ensemble des calculs réussis par $\mathcal{A}_O$ au seuil s.

On considère, pour les **questions 1** à **3**, l'automate de la **figure 1**, où $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $I = \{q_0\}$ et $F = \{q_2\}$.

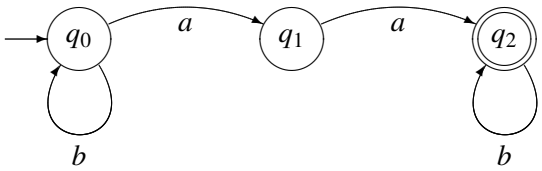


Figure 1 - Automate exemple

Q1. Donner sans justification $L_2(\mathcal{A}_{\{q_0\}})$.

Q2. Donner, en justifiant votre réponse, $L_2(\mathcal{A}_{\{q_1\}})$.

Q3. Soit $s \in \mathbb{N}$. Donner, en justifiant votre réponse, le cardinal de $L_s(\mathcal{A}_\emptyset)$ en fonction de s, où $\emptyset$ est l'ensemble vide.

Q4. Donner, sans justification, un majorant des calculs réussis au seuil s ne passant par aucun état de O.

Q5. Soit maintenant un calcul réussi au seuil s qui passe au moins par un état de O. On note $\tilde{q} = q_0 \cdots q_l$ la suite des états correspondants et $i_0 \cdots i_k \in [\![0, l]\!]$ la suite des indices dans $\tilde{q}$ correspondant aux états de O. Donner, sans justification, en fonction de s :

i) une majoration de i_0 ;

ii) un encadrement de i_k ;

iii) une majoration de $i_{j+1} - i_j$ pour $j \in [\![0, k-1]\!]$.

Q6. Soit $\mathcal{A}_O$ un automate augmenté, avec $|Q| = p$, où Q est l'ensemble des états de l'automate. Pour $s \in \mathbb{N}$, construire, en justifiant votre construction, un automate fini à $(s+1)p$ états reconnaissant $L_s(\mathcal{A}_O)$.

Partie II - Autour des tas

Cette partie comporte des questions de programmation qui seront abordées en utilisant **exclusivement le langage Python** (Informatique Pour Tous).
L'objectif est ici d'étudier et d'implémenter quelques outils autour d'une structure de données appelée *tas binomial*. Un tas binomial est une structure assez proche du tas binaire (utilisé par exemple pour réaliser une file de priorité), pour lequel la procédure de fusion de deux tas est efficace et peu complexe.

II.1 - Arbre binomial

Définition 5 (Arbre enraciné)
Un *arbre enraciné* est un graphe acyclique orienté possédant une unique racine et tel que tous les nœuds sauf la racine ont un unique parent.

La **figure 2** présente un exemple d'arbre enraciné dans lequel r est la racine.

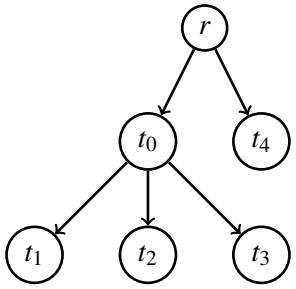


Figure 2 - Exemple d'arbre enraciné

On définit un arbre enraciné (non vide) par un couple `(r,[`$t_0, \cdots, t_{n-1}$`])`, où `r` est la valeur de la racine et `[`$t_0, \cdots, t_{n-1}$`]` est la liste de ses fils, chaque t_i, $i \in [\![0, n-1]\!]$ étant un arbre. Un arbre vide est défini par `None`. Par abus de notation, on confond dans la suite la racine de l'arbre et sa valeur, ainsi que les fils d'un arbre avec leur racine.

Q7. Écrire les fonctions Python :
- `Vide(a)` qui renvoit `True` si l'arbre `a` est vide, `False` sinon ;
- `Racine(a)` qui renvoit la racine de `a` si `a` est non vide ;
- `Fils(a)` qui renvoit la liste des arbres, fils de la racine de `a`.

Définition 6 (Arbre binomial)
Un *arbre binomial* a_k d'ordre $k \geq 0$ est un arbre enraciné dans lequel les fils de chaque nœud sont ordonnés. Il est défini récursivement comme suit :

i) a_0=`(r,[])` est constitué d'un nœud unique, la racine ;

ii) pour $k \in \mathbb{N}$, soit a_k=`(r,`$[t_0, \cdots t_{n-1}]$`)` un arbre non vide. a_k est un arbre binomial d'ordre k si :
- t_{n-1} est un arbre binomial d'ordre $(k-1)$;
- `(r,`$[t_0, \cdots t_{n-2}]$`)` est un arbre binomial d'ordre $(k-1)$;
- la racine de t_{n-1} a une valeur supérieure ou égale à `r`.

La **figure 3** donne un exemple d'arbre binomial d'ordre 3. Les valeurs dans l'arbre sont des entiers.

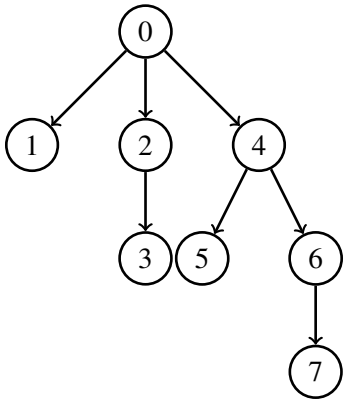


Figure 3 - Exemple d'arbre binomial d'ordre 3

Q8. Écrire une fonction Python `ArbreBinomial(a1,a2)` qui construit, à partir de deux arbres binomiaux `a1` et `a2` d'ordre $k-1$, un arbre binomial a_k d'ordre k contenant les mêmes valeurs que celles des arbres `a1` et `a2`. Dans la suite, on note `a1`$\oplus$`a2` cette opération.

Q9. Montrer par récurrence que la racine d'un arbre binomial d'ordre k a exactement k fils.

Q10. En déduire une fonction Python `Ordre(a)` qui renvoie l'ordre de l'arbre binomial `a`.

Q11. Montrer qu'un arbre binomial `a` d'ordre k possède 2^k nœuds.

Q12. Écrire une fonction récursive Python `EstUnArbreBinomial(a)` qui renvoie `True` si `a` est un arbre binomial, `False` sinon.

II.2 - Tas binomial

Un *tas* est une structure de données de type arbre qui permet en particulier de retrouver directement un élément qui doit être traité en priorité.

Définition 7 (Tas binomial)
Soient $k \geq 0$ et `T` $= \{a_0, \cdots, a_k\}$ un ensemble d'arbres. `T` est un *tas binomial* de longueur $k+1$ si, pour tout $i \in [\![0, k]\!]$, a_i est soit un arbre vide, soit un arbre binomial d'ordre i. Si $i = k$, a_i ne peut, de plus, pas être vide et a_k est donc un arbre binomial d'ordre k.

On note dans la suite |`T`| le nombre de nœuds d'un tas `T`.

Q13. Quelle structure `Python` adopter pour coder un tas ?

Définition 8 (Signature d'un tas)
Soit `T` $= \{a_0, \cdots, a_k\}$ un tas binomial de longueur $k+1$. On appelle *signature* de `T` la suite $s_0 \cdots s_k$ telle que pour tout $i \in [\![0, k]\!]$, $s_i = 0$ (respectivement $s_i = 1$) si l'arbre a_i est vide (respectivement n'est pas vide).

Q14. Soit T un tas binomial de longueur $k+1$. En utilisant sa signature, calculer |T| et montrer que $2^k \leq |T| < 2^{k+1}$. En déduire k en fonction de |T|.

Q15. Écrire une fonction Python `MinimumTas(T)` qui retourne la valeur minimum du tas T. En donner la complexité en fonction de |T|.

Les tas se construisent itérativement à partir de données. On est donc amené, pour un tas T, à ajouter un à un des éléments.
Soit p un élément que l'on souhaite ajouter à un tas binomial T non vide et déjà construit. L'insertion de la valeur p dans le tas T se fait alors selon l'algorithme 1.

Algorithme 1 - Insertion de p dans T.

Entrées : un tas T=$\{a_0 \cdots a_k\}$, une valeur p
Sorties : un tas T augmenté de la valeur p
début
- $i \leftarrow 0$
- Coder p dans un arbre binomial a d'ordre 0
- **tant que** $i<k+1$ *et* a *non vide* **faire**
  - **si** a_i *est vide* **alors**
    - $a_i \leftarrow$ a
    - Vider a
  - **sinon**
    - a a⊕a_i (***)
    - Vider a_i
  - $i \leftarrow i+1$
- **si** a *n'est pas vide* **alors**
  - Ajouter a au tas T.

Q16. Coder l'algorithme 1 sous la forme d'une fonction Python `Insertion(p,T)`.

Q17. Évaluer la complexité de cet algorithme en fonction de |T|. On suppose que l'étape marquée (***) s'effectue en temps constant.

Q18. Donner la signature du tas résultant de l'insertion de p dans T en fonction de la signature de T.

Q19. Donner, sans justification, un invariant de boucle pour la boucle de l'algorithme 1 permettant de prouver la correction de ce dernier.

Partie III - Autour de l'énumération des fractions positives

Cette partie comporte des questions nécessitant un **code OCaml**. Pour ces questions, **les réponses ne feront pas appel aux fonctionnalités impératives du langage** (en particulier pas de boucles, pas de références).

Notations
Dans la suite, on notera :
- $n \wedge d$ le PGCD (Plus Grand Commun Diviseur) de n et d ;
- $\lceil x \rceil$ la partie entière supérieure de x. Ainsi $\lceil 3.45 \rceil = 4$;
- $\lfloor x \rfloor$ la partie entière inférieure de x. Ainsi $\lfloor 3.45 \rfloor = 3$;
- log_2 la fonction logarithme de base 2. C'est la fonction réciproque de la fonction $i \mapsto 2^i$.

Une *fraction*, ou *nombre rationnel*, est le quotient de deux nombres entiers, le dénominateur étant par définition non nul. On notera $\mathbb{Q}^+$ l'ensemble des fractions positives.

L'objectif de cette partie est d'étudier une structure de données permettant d'énumérer l'ensemble des fractions positives.

Plusieurs travaux se sont intéressés à l'énumération des nombres rationnels, le plus connu étant très certainement la méthode de Cantor. Cependant, il n'est très souvent pas possible, à moins d'un travail assez complexe, de connaître une formule générale donnant le i-ième terme de cette énumération, ni même de donner le successeur dans l'énumération d'une fraction donnée.
Nous proposons dans la suite de répondre à ces questions en construisant un arbre, dit de Calkin-Wilf, permettant de manipuler cette énumération.

Définition 9 (Arbre binaire infini)
Un *arbre binaire homogène* est un arbre binaire dont tous les nœuds ont 0 successeur ou 2 successeurs, appelés fils gauche et droit. La *hauteur* h de l'arbre est la profondeur maximale des nœuds de l'arbre, c'est-à-dire la plus grande longueur d'un chemin de la racine vers une feuille de l'arbre. Lorsque h est infinie, on parle d'*arbre infini*.

Définition 10 (Arbre de Calkin-Wilf)
L'*arbre de Calkin-Wilf* est un arbre binaire homogène infini dont les nœuds sont des fractions positives. La racine de l'arbre est la fraction 1/1. Chaque sommet n/d a deux descendants : un fils gauche $n/(n+d)$ et un fils droit $(n+d)/d$.
Ainsi, si $N = n/d$, les deux fils de N sont $\dfrac{N}{1+N}$ (gauche) et $1+N$ (droit).

Q20. Dessiner l'arbre de Calkin-Wilf jusqu'à une profondeur de 3. Par convention, la racine de l'arbre est au niveau 0.

On propose le type record :

```
type fraction = {n:int; d:int}
```

pour définir une fraction de type n/d. La déclaration d'une telle fraction sera donc du type :

```
let f = {n=2;d=3};;
```

l'accès au numérateur (respectivement dénominateur) s'opérant par `f.n;;` (resp. `f.d;;`).

Q21. Proposer un type `OCaml` récursif permettant de décrire l'arbre de Calkin-Wilf.

Q22. Montrer par récurrence sur le niveau d'exploration de l'arbre que si n/d est un nœud de l'arbre, alors n et d sont premiers entre eux.

Q23. Écrire une fonction récursive `OCaml` de signature `pgcd : int * int -> int` qui calcule le PGCD de deux entiers naturels.

Q24. Écrire une fonction récursive `OCaml` de signature `fraction:int->int->fraction` qui construit une fraction positive irréductible à partir d'un numérateur n et un dénominateur d. La fonction vérifie que $n > 0$ et $d > 0$. Au besoin, elle simplifie la fraction par $n \wedge d$.

Par la suite, on ne construira des valeurs de type fraction que par l'intermédiaire de la fonction `fraction`, ce qui permettra de garantir l'invariant de type suivant : "pour toute valeur `f : fraction`, `f.n` et `f.d` sont premiers entre eux".

Q25. Soient deux nœuds k/l et n/d de l'arbre ayant des fils gauches (ou droits) identiques. Montrer qu'alors $k = n$ et $l = d$.

Q26. Soient N un nœud et $k \in \mathbb{N}$. Montrer que le nœud provenant d'une suite de k fils gauches de N est le nœud $\dfrac{N}{1+kN}$. Donner sans démonstration le nœud provenant d'une suite de k fils droits de N.

On définit alors une suite $(v_i)_{i\in\mathbb{N}}$, avec $v_0 = 0$, puis en effectuant un parcours en largeur, de gauche à droite, de l'arbre de Calkin-Wilf pour définir les termes de la suite.

Q27. Donner les huit premiers termes de la suite $(v_i)_{i\in\mathbb{N}}$.

Q28. Soient $n, d \in \mathbb{N}^*$ premiers entre eux. Montrer par récurrence sur $n + d$ que toute fraction n/d apparaît dans l'arbre.

Q29. Montrer qu'aucune fraction n'apparaît deux fois dans l'arbre.

Q30. Déduire des questions précédentes que la suite $(v_i)_{i\in\mathbb{N}}$ est une bijection de $\mathbb{N}$ dans $\mathbb{Q}^+$.

La suite $(v_i)_{i\in\mathbb{N}}$ permet donc d'affirmer que $\mathbb{Q}^+$ est dénombrable. Nous allons utiliser $(v_i)_{i\in\mathbb{N}}$ pour déterminer, dans l'énumération des fractions produite par cette suite, la position de n'importe quelle fraction positive n/d. En d'autres termes, étant donnée une fraction positive n/d, on se propose de rechercher i tel que $n/d = v_i$.

Q31. Soit $i \in \mathbb{N}^*$. Montrer que le fils gauche du nœud de valeur v_i a pour valeur v_{2i}. Montrer de même que le fils droit a pour valeur v_{2i+1}.

Pour pouvoir énoncer le i-ième terme dans cette énumération, on introduit alors une suite auxiliaire, aux nombreuses propriétés arithmétiques et liens avec d'autres objets mathématiques.

La *suite diatomique de Stern* (ou *suite de Stern-Brocot*) doit son nom à Moritz Stern (1807-1894), élève de Gauss et Achille Brocot (1817-1878), horloger qui s'intéressait aux fractions pour la fabrication d'horloges avec des engrenages comportant peu de dents, donc simples à fabriquer.

Définition 11 (Suite diatomique de Stern ou suite de Stern-Brocot)
La *suite diatomique de Stern* $(s_i)_{i\in\mathbb{N}}$ est définie par $s_0 = 0$, $s_1 = 1$ et pour tout $i \geq 1$:

$$\begin{cases} s_{2i} &= s_i \\ s_{2i+1} &= s_i + s_{i+1} \end{cases}.$$

Q32. Donner les dix premiers termes de la suite $(s_i)_{i\in\mathbb{N}}$.

Q33. Écrire une fonction récursive `OCaml` de signature `stern : int -> int` permettant de calculer les termes de cette suite.

Q34. Déduire par récurrence de la **Q31** que : $\forall i \in \mathbb{N},\ v_i = \dfrac{s_i}{s_{i+1}}$.

La suite diatomique de Stern permet d'exprimer le i-ième terme dans l'énumération des fractions positives qui est induite par le parcours en largeur de l'arbre de Calkin-Wilf décrit ci-dessus.

Voyons maintenant comment obtenir de manière rapide le successeur d'une fraction v_i donnée, sans connaître i, dans cette énumération. Trois cas peuvent se présenter :

- premier cas : les nœuds de valeur v_i et v_{i+1} sont à même profondeur k, fils d'un même nœud N,
- deuxième cas : le nœud de valeur v_i est le dernier nœud à droite à la profondeur k,
- troisième cas : les nœuds de valeur v_i et v_{i+1} sont à même profondeur k, mais ne sont pas fils d'un même nœud.

On pose pour tout $x > 0$, $f(x) = \dfrac{1}{1 + 2\lfloor x \rfloor - x}$.

Q35. Montrer que dans le premier cas, $v_{i+1} = f(v_i)$.

Q36. Montrer, en utilisant la **Q26**, que dans le deuxième cas on a encore $v_{i+1} = f(v_i)$.

On étudie enfin le dernier cas : les nœuds de valeur v_i et v_{i+1} sont sur une même profondeur k, mais ne sont pas les fils d'un même nœud. On va donc passer par la recherche d'un ancêtre commun de ces deux nœuds. Dans la suite, on s'intéressera toujours au premier ancêtre commun, c'est-à-dire celui de profondeur maximale.
En partant de la racine `r`, il est possible d'atteindre n'importe quel nœud $N = n/d$ de l'arbre par une suite de déplacements vers la gauche (`G`) ou vers la droite (`D`). Le *chemin* de `r` vers N peut donc être codé par un mot sur l'alphabet {`G`,`D`}.

Si un déplacement est représenté en OCaml par un type

```
type direction = G | D
```

alors un chemin est une liste `direction list`.

Q37. Écrire une fonction OCaml de signature `chemin : fraction -> direction list` qui calcule le chemin de la racine à un nœud quelconque de l'arbre. Cette fonction fera appel à une fonction auxiliaire récursive. Ainsi `chemin n d` calcule la liste des directions à prendre pour passer de `r` au nœud n/d. On pourra supposer l'existence d'une fonction de signature `rev : direction list -> direction list` qui renverse une liste.

Q38. Écrire une fonction OCaml de signature `noeud : direction list -> fraction` qui détermine le nœud obtenu en effectuant une série de déplacements depuis la racine. Ainsi `noeud [D;D;G;D]` renverra un couple d'entiers (n, d) correspondant au nœud de valeur n/d. Cette fonction fera appel à une fonction auxiliaire récursive.

Q39. Utiliser les deux fonctions précédentes pour écrire une fonction OCaml `ancetre` de signature `ancetre : fraction -> fraction -> fraction` qui détermine le premier ancêtre commun entre deux nœuds. Ainsi `ancetre (n,d) (p,q)` détermine le premier ancêtre commun à n/d et p/q. Cette fonction pourra utiliser une fonction auxiliaire récursive.

Q40. On suppose que le nœud de valeur v_p, le premier ancêtre commun des nœuds de valeur v_i et v_{i+1}, est à k' niveaux au-dessus d'eux. Donner le chemin entre v_p et v_i sous la forme d'une liste de `direction`. Donner de même le chemin entre v_p et v_{i+1}.

Q41. À partir de l'expression des fils gauche et droit de v_p, montrer que l'on a encore $v_{i+1} = f(v_i)$.

FIN

IMPRIMERIE NATIONALE – 22 1166 – D'après documents fournis

Travailler les chapitres de ce sujet

Prepa Help suit le programme de MP chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Bases de Python : expressions, types, instructions, fonctions
- Complexité des algorithmes itératifs et récursifs
- Récursivité
- Langage OCaml : expressions, types, filtrage, récursivité
- Arithmétique dans l'ensemble des entiers relatifs

Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière : **écrire sur WhatsApp**

Les autres années et épreuves : prepahelp.com/annales/e3a/mp