Sélection de sujets posés lors de la session 2023

Sujets sans corrigés

Exercices de type A

Exercice 1 *Grammaires algébriques (type A)*

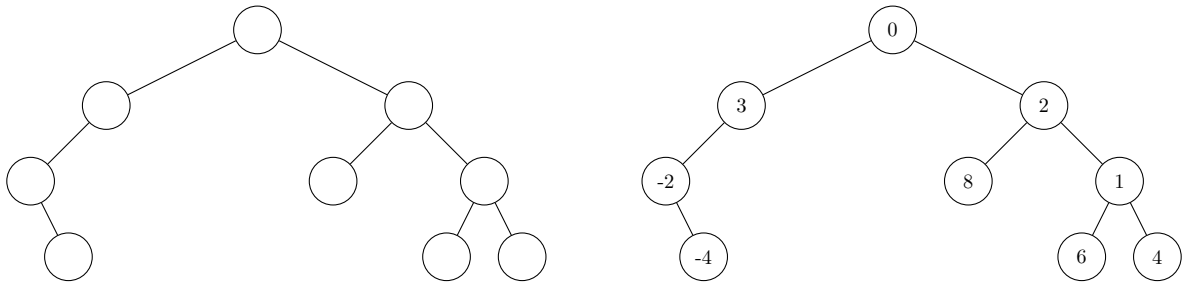
On considère la grammaire algébrique G sur l'alphabet $\Sigma = \{a, b\}$ et d'axiome S dont les règles sont :

$$S \to SaS \mid b$$

1. Cette grammaire est-elle ambiguë ? Justifier.
2. Déterminer (sans preuve pour cette question) le langage L engendré par G. Quelle est la plus petite classe de langages à laquelle L appartient ?
3. Prouver que $L = L(G)$.
4. Décrire une grammaire qui engendre L de manière non ambiguë en justifiant de cette non ambiguité.
5. Montrer que tout langage dans la même classe de langages que L peut être engendré par une grammaire algébrique non ambiguë.

Exercice 2 *Minima locaux dans des arbres (type A)*

Dans cet exercice, on considère des arbres binaires étiquetés par des entiers relatifs deux à deux distincts. Un nœud est un minimum local d'un arbre si son étiquette est plus petite que celle de son éventuel père et celles de ses éventuels fils. Considérons par exemple l'étiquetage (b) de l'arbre binaire non étiqueté (a) :



(a) (b)

1. Déterminer le ou les minima locaux de l'arbre (b).
2. Donner une définition inductive permettant de définir les arbres binaires ainsi que la définition de la hauteur d'un arbre. Quelle est la hauteur de l'arbre (b) ?
3. Montrer que tout arbre non vide possède un minimum local.
4. Proposer un algorithme permettant de trouver un minimum local d'un arbre non vide et déterminer sa complexité.

On considère un arbre binaire non étiqueté que l'on souhaite étiqueter par des entiers relatifs distincts deux à deux de manière à maximiser le nombre de minima locaux de cet arbre.

5. Proposer sans justifier un étiquetage de l'arbre (a) qui maximise le nombre de minima locaux.
6. Proposer un algorithme qui, étant donné un arbre binaire non étiqueté en entrée, permet de calculer le nombre maximal de minima locaux qu'il est possible d'obtenir pour cet arbre. Déterminer la complexité de votre algorithme.
7. Montrer que, pour un arbre de taille $n \in \mathbb{N}$, le nombre maximal de minima locaux est majoré par $\left\lfloor \frac{2n+1}{3} \right\rfloor$. *On pourra remarquer que les nœuds non minimaux couvrent l'ensemble des arêtes de l'arbre.*

Exercice 3 *Formules propositionnelles croissantes (type A)*

On fixe un entier $n \geq 1$ et $E = \{x_1, \cdots, x_n\}$ un ensemble de variables propositionnelles. Étant données deux applications $a : E \to \{V, F\}$ et $b : E \to \{V, F\}$ on dit que a est plus petite que b (que l'on note $a \leq b$) si :

$$\forall x \in E, a(x) = V \implies b(x) = V.$$

Dans un but de simplification des calculs, on pourra faire les abus de notation suivants : assimiler V à 1 et F à 0 et vice versa. Avec cet abus, la propriété $a \leq b$ se traduit par :

$$\forall x \in E, a(x) \leq b(x).$$

1. Étant donnée une valuation sur E, rappeler comment on l'étend naturellement en une valuation sur les formules propositionnelles.

On dit qu'une formule propositionnelle P est *croissante* si pour tout a, b des valuations vérifiant $a \leq b$, on a :

$$a(P) = V \implies b(P) = V.$$

2. Montrer que si P, Q sont des formules croissantes, alors $P \wedge Q$ et $P \vee Q$ sont des formules croissantes.
3. Soit C une clause conjonctive satisfiable contenant au moins un littéral. Montrer qu'elle est croissante si et seulement si elle ne contient aucun littéral de la forme $\neg x$ avec $x \in E$.
4. On considère une formule propositionnelle P qui n'est ni une tautologie, ni une antilogie.
   (a) Montrer que si P est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme $\neg x$ avec $x \in E$, alors P est une formule propositionnelle croissante.
   (b) Réciproquement, montrer que si P est une formule propositionnelle croissante, alors elle est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme $\neg x$ avec $x \in E$.

Exercice 4 *Activation de processus (type A)*

Soit un système temps réel à n processus asynchrones $i \in [\![1, n]\!]$ et m ressources r_j. Quand un processus i est actif, il bloque un certain nombre de ressources listées dans un ensemble P_i et une ressource ne peut être utilisée que par un seul processus. On cherche à activer simultanément le plus de processus possible.

Le problème de décision `ACTIVATION` correspondant ajoute un entier k aux entrées et cherche à répondre à la question : "Est-il possible d'activer au moins k processus en même temps ?"

1. Soit $n = 4$ et $m = 5$. On suppose que $P_1 = \{r_1, r_2\}$, $P_2 = \{r_1, r_3\}$, $P_3 = \{r_2, r_4, r_5\}$ et $P_4 = \{r_1, r_2, r_4\}$. Est-il possible d'activer 2 processus en même temps ? Même question avec 3 processus.
2. Dans le cas où chaque processus n'utilise qu'une seule ressource, proposer un algorithme résolvant le problème `ACTIVATION`. Évaluer la complexité de votre algorithme.

On souhaite montrer que `ACTIVATION` est NP-complet.

3. Donner un certificat pour ce problème.
4. Écrire en pseudo code un algorithme de vérification polynomial. On supposera disposer de trois primitives, toutes trois de complexité polynomiale :
   (a) `appartient(c,i)` qui renvoie `Vrai` si le processus `i` est dans l'ensemble d'entiers `c`.
   (b) `intersecte(`$\mathtt{P}_i$`,R)` qui renvoie `Vrai` si le processus `i` utilise une ressource incluse dans un ensemble de ressources `R`.
   (c) `ajoute(`$\mathtt{P}_i$`,R)` qui ajoute les ressources $\mathtt{P}_i$ dans l'ensemble `R` et renvoie ce nouvel ensemble.

En théorie des graphes, le problème `STABLE` se pose la question de l'existence dans un graphe non orienté $G = (S, A)$ d'un ensemble d'au moins k sommets ne contenant aucune paire de sommets voisins. En d'autres termes, existe-t-il $S' \subset S$, $|S'| \geq k$ tel que $s, p \in S' \Rightarrow (s, p) \notin A$?

5. En utilisant le fait que `STABLE` est NP-complet, montrer par réduction que le problème `ACTIVATION` est également NP-complet.

Exercices de type B

Exercice 5 *Langages locaux (type B)*

Consignes : Cet énoncé est accompagné d'un code compagnon en OCaml localite.ml *fournissant le type décrit par l'énoncé et quelques fonctions auxiliaires : il est à compléter en y implémentant les fonctions demandées. On privilégiera un style de programmation fonctionnel.*

On considère un alphabet Σ. Si L est un langage sur Σ, on note :

- $P(L) = \{a \in \Sigma \mid a\Sigma^\star \cap L \neq \emptyset\}$ l'ensemble des premières lettres des mots de L.
- $D(L) = \{a \in \Sigma \mid \Sigma^\star a \cap L \neq \emptyset\}$ l'ensemble des dernières lettres des mots de L.
- $F(L) = \{m \in \Sigma^2 \mid \Sigma^\star m \Sigma^\star \cap L \neq \emptyset\}$ l'ensemble des facteurs de longueur 2 des mots de L.
- $N(L) = \Sigma^2 \setminus F(L)$ l'ensemble des mots de taille 2 qui ne sont pas facteurs de mots de L.

On rappelle qu'un langage L est dit *local* si et seulement si l'égalité de langages suivantes est vérifiée :

$$L \setminus \{\varepsilon\} = (P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L) \Sigma^\star)$$

1. Calculer les ensembles $P(L), D(L), F(L)$ et $N(L)$ dans le cas où L est le langage dénoté par l'expression régulière $a^\star(ab)^\star + aa$ sur l'alphabet $\{a, b\}$. Ce langage est-il local ? On vérifiera la cohérence entre les réponses à cette question et celles obtenues via les fonctions demandées dans la suite de l'énoncé.

On cherche dans la suite de l'exercice à concevoir un algorithme répondant à la spécification suivante :

Entrée : Une expression régulière e sur un alphabet Σ ne faisant pas intervenir le symbole $\emptyset$.
Sortie : Vrai si le langage dénoté par e est local, faux sinon.

Par défaut, dans la suite de l'énoncé, "expression régulière" signifie "expression régulière ne faisant pas intervenir le symbole $\emptyset$". Les expressions régulières seront manipulées en OCaml via le type somme suivant :

```
type regexp =
  | Epsilon
  | Letter of string (*La chaîne en question sera toujours de longueur 1*)
  | Union of regexp * regexp
  | Concat of regexp * regexp
  | Star of regexp
```

On propose tout d'abord de calculer les ensembles $P(L)$, $D(L)$ et $F(L)$ à partir d'une expression régulière dénotant L. Ces ensembles seront représentés en OCaml par des listes de chaînes de caractères qui vérifieront les propriétés suivantes :

- Elles sont triées dans l'ordre croissant selon l'ordre lexicographique.
- Elles sont sans doublons.

L'énoncé fournit une fonction `union` permettant de calculer l'union sans doublons de deux listes triées. La définition inductive d'une expression régulière invite à calculer inductivement les ensembles $P(L)$, $D(L)$ et $F(L)$. C'est ce que propose la fonction `compute_P` fournie par l'énoncé.

2. En supposant que la fonction `contains_epsilon : regexp -> bool` renvoie `true` si et seulement si le langage dénoté par l'expression régulière en entrée contient le mot ε, justifier brièvement la correction de `compute_P`.
3. Implémenter la fonction `contains_epsilon`.

4. Sur le modèle de `compute_P`, implémenter une fonction `compute_D : regexp -> string list` permettant le calcul de l'ensemble $D(L)$ étant donnée une expression régulière dénotant le langage L.

5. Expliquer en langage naturel comment calculer récursivement l'ensemble $F(L)$ étant donnée une expression régulière e dénotant le langage L. Si $e = e_1e_2$ on pourra exprimer $F(L)$ en fonction notamment de $P(L_2)$ et $D(L_1)$ où L_1 (resp. L_2) est le langage dénoté par e_1 (resp. e_2).

6. Écrire une fonction `prod : string list -> string list -> string list` calculant le produit cartésien des deux listes en entrée, qu'on pourra supposer triées dans l'ordre lexicographique croissant et sans doublons, puis qui pour chaque couple de chaînes dans la liste obtenue les concatène. Par exemple :

```
prod ["a";"c";"e"] ["b";"c"] = ["ab";"ac";"cb";"cc";"eb";"ec"]
```

7. En déduire une fonction `compute_F : regexp -> string list` déterminant l'ensemble $F(L)$ étant donnée une expression régulière dénontant le langage L.

Dans les questions qui suivent, on ne demande PAS d'implémenter les algorithmes décrits.

8. Décrire en pseudo-code ou en langage naturel un algorithme permettant de calculer un automate reconnaissant $(P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L)\Sigma^\star)$ étant donnée une expression régulière dénotant L.

9. Décrire un algorithme permettant de détecter si le langage dénoté par une expression régulière est local ou non.

10. Pourquoi est-il légitime de ne considérer que les expressions régulières ne faisant pas intervenir $\emptyset$? Comment modifier l'algorithme obtenu dans le cas où cette contrainte n'est plus vérifiée ?

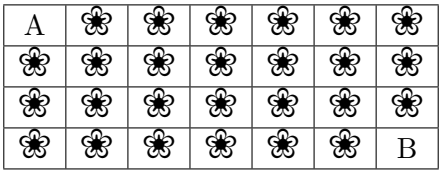
Extrait du code compagnon, non présent dans l'énoncé original.

```
10 type regexp =
11   | Epsilon
12   | Letter of string
13   | Union of regexp * regexp
14   | Concat of regexp * regexp
15   | Star of regexp
16
17 (** Entrée : deux listes l1 et l2 dont les éléments sont de même nature et sont triés
   ↪  dans l'ordre croissant
18 Sortie : une liste l contenant tous les éléments de l1 et de l2 sans doublons et triée
   ↪  dans l'ordre croissant *)
19 let rec union (l1:'a list) (l2:'a list) :'a list = match l1, l2 with
20   |[], l |l, [] -> l
21   |t::q, a::b when t < a -> t::(union q (a::b))
22   |t::q, a::b when t = a -> t::(union q b)
23   |t::q, a::b -> a::(union (t::q) b)
24
25 (** Entrée : une expression régulière e
26 Sortie : la liste des premières lettres des mots du langage dénoté par e
27 ATTENTION : On ne pourra tester cette fonction qu'après avoir implémenté
   ↪  contains_epsilon *)
28 let rec compute_P (e:regexp) :string list = match e with
29   |Epsilon -> []
30   |Letter a -> [a]
31   |Union (e1, e2) -> union (compute_P e1) (compute_P e2)
32   |Concat (e1, e2) when contains_epsilon e1 -> union (compute_P e1) (compute_P e2)
33   |Star e1 |Concat (e1, _) -> (compute_P e1)
```

Exercice 6 *Récolte dynamique de fleurs (type B)*

Consignes : Cet énoncé est accompagné d'un code compagnon en C bouquet_enonce.c *fournissant certaines des fonctions mentionnées dans l'énoncé : il est à compléter en y implémentant les fonctions demandées. La ligne de compilation* gcc -o main.exe *.c -lm *vous permet de créer un exécutable* main.exe *à partir du ou des fichiers C fournis. Vous pouvez également utiliser l'utilitaire* make. *En ligne de commande, il suffit de taper* make. *Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande* ./main.exe. *Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant* make clean *et relancer* make.

Une petite fille se trouve en haut à gauche (case A) d'un champ modélisé par un tableau rectangulaire de taille $m \times n$ et doit se rendre dans la case B en bas à droite du champ où réside sa grand-mère (figure ci-dessous).



Chaque case du tableau, *y compris les cases A et B*, contient un certain nombre de fleurs. La petite fille, qui connaît depuis sa position initiale le nombre de fleurs de chaque case, doit se déplacer vers B de case en case, les seuls mouvements autorisés étant vers le bas ou vers la droite. À chaque déplacement, elle récolte les fleurs de la case atteinte. L'objectif pour elle est alors de faire le bouquet avec le plus de fleurs possible lors de son déplacement pour l'offrir à sa grand-mère.

1. On considère le champ suivant :

0 (A)	1	2	3
1	2	3	4
2	3	4	0
3	4	0	1 (B)

   Donner le nombre maximal de fleurs cueillies par la petite fille.

2. On note $n(i,j)$ le nombre maximum de fleurs que la petite fille peut récolter en se déplaçant de A à la case (i,j). Exprimer $n(i,j)$ en fonction de $n(i-1,j)$ et $n(i,j-1)$. En déduire une fonction récursive de prototype `int recolte(int champ[m][n], int i, int j)` qui, étant données les coordonnées `i`, `j` d'une case, calcule le nombre maximum de fleurs cueillies par la petite fille de A à la case (`i`, `j`).

3. On suppose $m = n = 4$ et on effectue donc un appel à `recolte(champ,3,3)` pour résoudre le problème posé. Donner le nombre de fois où votre fonction calcule le nombre de fleurs maximum cueillies dans la case $(1,1)$ (deuxième case de la diagonale).

D'une manière générale, le nombre d'appels à la fonction récursive est important. On a donc intérêt à transformer l'algorithme récursif en algorithme dynamique. On propose de déclarer dans le programme principal un tableau `fleurs` dont la case (i,j) est destinée à contenir la récolte maximale que la petite fille peut obtenir en cheminant de A vers la case (i,j).

4. Dans quel ordre remplir le tableau `fleurs` de sorte à éviter de recalculer une valeur ?

5. Écrire une fonction de prototype `int recolte_iterative(int champ[m][n], int i, int j, int fleurs[m][n])` qui calcule, stocke dans `fleurs[i][j]` et retourne la cueillette maximale obtenue en parcourant le champ de A à la case (`i`, `j`).

La fonction `recolte_iterative` permet de déterminer la cueillette maximale en (`i`, `j`) mais ne précise pas le chemin parcouru pour l'obtenir.

6. Écrire la fonction de prototype `void deplacements(int fleurs[m][n], int i, int j)` qui affiche la suite des déplacements effectués par la petite fille sur un chemin permettant de récolter

le nombre maximum de fleurs entre (0,0) et (`i`, `j`).

7. Insérer un appel de `deplacements` dans la fonction `recolte_iterative` pour afficher le chemin parcouru.

Extrait du code compagnon, non présent dans l'énoncé original.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*
La directive #define est utilisée pour définir des valeurs pour
des constantes qui serviront à déclarer des tableaux de taille fixe.
*/
#define m 5
#define n 3

/* Macro de calcul du maximum entre i et j */
int max(int i,int j){
    if (i<j)
        return j;
    else
        return i;
}
```

```c
int main(){
  int champ[m][n],fleurs[m][n];
  int i,j;

  /* Exemple du champ de fleurs : le nombre de fleurs par case est un
  entier aléatoire entre 0 et 10. On utilise la fonction int rand()de
  stdlib. Le générateur de nombre pseudo-aléatoires est tout d'abord
  initialisé.*/
   srand(time(NULL));
   for (i=0;i<m;i++) for(j=0;j<n;j++)
        champ[i][j] = rand() % 11;

   printf("\n Nombre de fleurs maximum cueillies :
    ↪ %d\n",recolte_iterative(champ,3,3,fleurs));

   return 0;
}
```

Exercice 7 *Chemins simples sans issue (type B)*

Consignes : Cet exercice est à traiter en OCaml. Le fichier chemins_simples.ml *est fourni avec ce sujet. Il est à compléter en y implémentant les fonctions demandées.*

L'objectif de cet exercice est de programmer une fonction générant la liste des chemins simples sans issue d'un graphe. On rappelle les définitions d'un graphe, d'un chemin, et on donne leur représentation en OCaml.

Un *graphe orienté* est un couple (V, E) où V est un ensemble fini (ensemble des sommets), E est un sous-ensemble de $V \times V$ où tout élément $(v_1, v_2) \in E$ vérifie $v_1 \neq v_2$ (ensemble des arcs).

Étant donné un graphe $G = (V, E)$ un *chemin non vide* de G est une suite finie $s_0, \cdots, s_n$ de sommets de V avec $n \geq 0$ et vérifiant $\forall i \in \{0, \cdots, n-1\}, (s_i, s_{i+1}) \in E$. On dit que ce chemin est *simple* si $s_0, \cdots, s_n$ sont distincts deux à deux. On dit qu'il est *sans issue* si pour tout s_{n+1} sommet tel que $(s_n, s_{n+1}) \in E$, s_{n+1} appartient à $\{s_0, \cdots, s_n\}$.

Dans la suite, les graphes considérés sont définis sur un ensemble de sommets de la forme $\{0, 1, \cdots, n-1\}$. Pour représenter un graphe en OCaml, on utilise le type suivant :

```
type graphe = int list array
```

qui correspond à un encodage par un tableau de listes d'adjacence. Par exemple, le graphe

$$G_1 = (\{0,1,2,3\}, \{(0,1),(0,3),(2,0),(2,1),(2,3),(3,1)\})$$

est représenté par le tableau `[|[1;3];[];[0;1;3];[1]|]`. L'ordre dans lequel sont écrits les éléments dans les listes importe peu. Par contre, l'emplacement des listes dans le tableau est important. Par exemple, `[|[];[0];[0;3;1];[1]|]` représente le graphe

$$G_2 = (\{0,1,2,3\}, \{(1,0),(2,0),(2,1),(2,3),(3,1)\}\})$$

On rappelle différentes fonctions pouvant être utiles :

- `List.filter : ('a -> bool) -> 'a list -> 'a list` où l'expression `List.filter f l` est la liste obtenue en gardant uniquement les éléments `x` de `l` vérifiant `f`.
- `List.iter :('a -> unit) -> 'a list -> unit` où `List.iter f l` correspond à `(f a0);(f a1);...;(f an)` dans le cas où on a `l = a0::a1::...::an::[]`.
- `List.rev : 'a list -> 'a list` est une fonction qui renvoie le retourné d'une liste. Par exemple, `List.rev [3;1;2;2;4]` est égal à `[4;2;2;1;3]`.
- `Array.length : 'a array -> int` est une fonction qui renvoie la longueur d'un tableau.

Les questions de programmation sont à traiter dans le fichier chemins_simples.ml. L'utilisation d'autres fonctions de la bibliothèque que celles mentionnées sont à reprogrammer.

1. Écrire une fonction `est_sommet : graphe -> int -> bool` où `est_sommet g a` est égal à `true` si `a` est un sommet du graphe `g` et `false` sinon.
2. Écrire une fonction `appartient : 'a list -> 'a -> bool` où `appartient l x` est égal à `true` si `x` est un élément de `l` et `false` sinon.
3. Écrire une fonction `est_chemin : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin de `g` et `false` sinon. On suppose que la liste vide représente le chemin vide, qui est bien un chemin et que les éléments de `l` sont bien des sommets du graphe `g`.
4. Compléter la fonction `est_chemin_simple_sans_issue : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin simple sans issue de `g` et `false` sinon. On supposera que les éléments de `l` sont des sommets du graphe `g` et que le chemin vide n'est pas simple sans issue.

5. On cherche à écrire une fonction qui construit la liste des chemins simples sans issue d'un graphe. Pour cela, on procède à l'aide de parcours en profondeur et d'un algorithme de retour sur trace. Compléter le code de la fonction `genere_chemins_simples_sans_issue` présent dans le fichier chemins_simples.ml et qui permet de générer la liste des chemins simples sans issue d'un graphe.
6. Écrire des expressions donnant les listes des chemins simples pour les deux graphes G_1 et G_2.
7. Expliciter la complexité des fonctions `appartient` et `est_chemin_simple_sans_issue`.

Extrait du code compagnon, non présent dans l'énoncé original.

```
10 type graphe = int list array
11
12 (*graphes donnés en exemple*)
13 let g1 = [|[1;3];[];[0;1;3];[1]|]
14
15 let g2 = [|[];[0];[0;3;1];[1]|]
```

```
40 (*Question 4*)
41 let est_chemin_simple_sans_issue g liste =
42   let n = Array.length g in
43   let visites = Array.make false (n - 1)
44   (* tableau qui enregistre les sommets déjà vus*)
45   in
46   let rec test_aux liste =
47     (* parcourt la liste en vérifiant au fur et à mesure que tous
48        les critères sont vérifiés*)
49     match liste with
50     | [] -> (* à completer : cas du chemin vide*)
51     | [a] ->(* à compléter : vrai dans le cas où le dernier sommet n'a
52                jamais été visité et que tous ses voisins l'ont été *)
53     | a::b::suite ->
54         begin
55           (* à compléter *)
56         end
57     in
58     test_aux liste
```

```
(*Question 5*)
let genere_chemins_simples_sans_issue (g:graphe) =
  let taille = Array.length g in
  let liste_chemins = ref [] in
  (*mémorise les chemins déjà trouvés en sens inverse *)
  let visites = Array.make taille false in
  (*garde en mémoire les sommets en cours de visite *)
  let chemin_courant_envers = ref [] in
  (*garde en mémoire le début d'un chemin*)
  let rec profondeur s =
  (*trouve tous les chemins simples sans issue commençant par s*)
    if not visites.(s) then
    begin
      visites.(s) <- true ;
      chemin_courant_envers := s::(!chemin_courant_envers) ;
      let voisins_libres =
      (*à compléter : calcule la liste des vois. de s non visités*)
      in if voisins_libres = [] then
        begin
        (*à compléter : cas où on a trouvé un ch. simple sans issue*)
        end
      else
        begin
        (*à compléter: cas où on continue pour trouver d'autres chem.*)
        end ; (*fin du if en fonction de voisins_libres *)
      visites.(s) <- false ; (*pour revenir en arrière *)
      chemin_courant_envers := List.tl !chemin_courant_envers ;
      (*pour revenir en arrière,
      List.tl calcule la liste privée de son premier elem *)
    end
  in
    for i = 0 to (taille-1) do
      profondeur i
    done ; !liste_chemins
```

Exercice 8 *Calculs avec les flottants (type B)*

Consignes : Cet énoncé est accompagné d'un code compagnon en C, flottants.c *fournissant les structures de données et certaines fonctions mentionnées dans l'énoncé. Il est à compléter en y implémentant les fonctions demandées. La ligne de compilation* gcc -o main.exe *.c -lm *vous permet de créer un exécutable* main.exe*. Vous pouvez également utiliser l'utilitaire* make*. En ligne de commande, il suffit de taper* make*. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande* ./main.exe*. Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant* make clean *et relancer* make*.*

Un nombre réel x est représenté en machine en base 2 par un flottant qui a un signe s, une mantisse m et un exposant e tel que $x = s \times m \times 2^e$. Dans la norme IEEE 754, en convention normalisé la partie entière de la mantisse est 1 qui est un bit caché. En simple précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 23 bits et l'exposant sur 8 bits. En double précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 52 bits et l'exposant sur 11 bits.

Dans cet exercice, on observe le résultat de calculs obtenus par un programme. On pourra utiliser la fonction de signature : `double pow(double v, double p)` qui calcule v^p.

1. Dans la fonction principale `main`, on a défini 3 variables a, b, c de type `double`. Compléter le code pour calculer et afficher le résultat des opérations $(a+b)+c$ et $a+(b+c)$. Que constatez-vous ?

2. Compte tenu des approximations faites lors du codage, on peut trouver plusieurs nombres x tels que $1+x=1$ après un calcul fait par la machine. Le plus petit nombre représentable exactement en machine et supérieur à 1 s'écrit $1+\epsilon$, avec ϵ un réel appelé ϵ machine. On admet que ϵ s'écrit sous la forme 2^{-n} avec n un entier naturel strictement positif. Écrire une fonction de signature `double epsilon()` qui renvoie la valeur de n. Justifier cette valeur.

3. On considère une suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_n &= 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}\times u_{n-2}} \text{ si } n \geq 2 \end{cases}$$

   Écrire une fonction de signature `double u(int n)` qui renvoie la valeur du terme u_n.

4. La limite théorique de la suite $(u_n)_{n\in\mathbb{N}}$ est 6. Compléter la fonction `main` afin d'afficher les 22 premiers termes de la suite. Vers quelle valeur semble tendre la suite ?

5. On définit une liste chaînée de nombres à l'aide d'une structure `nb` comportant un double et un pointeur vers une structure `nb` définie ci-dessous

```
struct nb {double x; struct nb* suivant;};
```

   Écrire une fonction de signature `double somme(struct nb* tab)` qui calcule la somme des éléments de la liste `tab`.

6. L'algorithme suivant permet d'augmenter la précision du calcul lors du calcul d'une somme.

> **Entrée :** Une liste l de réels triée dans l'ordre croissant de taille au moins 2.
> **Sortie :** La somme des réels contenus dans la liste l.
> **tant que** la liste l contient strictement plus d'un élément
> | Calculer la somme $s = x + y$ des deux premiers éléments x et y de l
> | Supprimer x et y de l
> | Insérer s dans l de sorte à ce que l reste triée
> **renvoyer** l'unique élément de l

   - Compléter la fonction `somme2` qui implémente cet algorithme.
   - La fonction proposée ne prend pas en compte un cas d'insertion. Illustrer ce propos.

Extrait du code compagnon, non présent dans l'énoncé original.

```c
#include<stdio.h>
#include<math.h>
#include<stdlib.h>

struct nb{
    double x;
    struct nb* suivant;
};

double somme2(struct nb* tab){
    //on suppose que tab est triée par ordre
    //croissant et est non vide

    struct nb* temp=tab;

    while(temp!=NULL) {
        if (temp->suivant !=NULL) {
            struct nb* s;
            //Création du maillon s contenant la somme des 2 premiers éléments
            // A compléter
            //temp pointe vers le 3e élément de la liste
            temp=(temp->suivant)->suivant;
            //recherche de l'emplacement de s

            struct nb* t=temp;
            struct nb* t2=temp;

            while(t!=NULL){
                /*si la valeur pointée par t est strict. supérieure à
                celle de s, t pointe sur l'élément suivant,
                sinon on quitte la boucle */
                //A compléter
            }
            //insertion de s
            if (t==NULL & t2==NULL) {
                temp=s;
            } else {
                if (t==NULL & t2!=NULL){
                    t2->suivant=s;
                } else {
                    t2->suivant=s;
                    s->suivant=t;
                }
            }
        } else {
            return temp->x;
        }
    }
    return temp->x;
}
```