Sélection de sujets posés lors de la session 2026

Exercices de type A

Exercice 1 Langages réguliers

Soit V un alphabet. Étant donnés deux mots $x, z \in V^\star$, on dit que x est conjugué à z s'il existe un mot $y \in V^\star$ tel que $xy = yz$.

1. Pour $V = \{0, 1\}$, montrer que $x = 001$ est conjugué avec $z_1 = 010$ mais pas avec $z_2 = 01$.
2. Expliciter l'ensemble des mots z qui sont conjugués avec $x = \varepsilon$.
3. Montrer que s'il existe deux mots $u, v \in V^\star$ tels que $x = uv$ et $z = vu$ alors x est conjugué à z.
4. On suppose que $x \neq \varepsilon$ est conjugué à z. Soit alors y **de longueur minimale** tel que $xy = yz$. Montrer qu'on a nécessairement $|x| \geq |y|$.
5. Montrer que x est conjugué à z si et seulement s'il existe deux mots $u, v \in V^\star$ tels que :

$$x = uv \text{ et } z = vu$$

6. Montrer que la relation de conjuguaison est une relation d'équivalence.
7. Soit $C(x)$ l'ensemble des mots de $V^\star$ conjugués au mot x. $C(x)$ est-il un langage régulier ?

Pour un langage L, on note $C(L) = \bigcup_{x \in L} C(x)$ l'ensemble des mots conjugués aux mots de L.

8. Donner un exemple de langage L tel que $C(L)$ n'est pas régulier. On pourra considérer par exemple le cas d'un alphabet V à une seule lettre.
9. Si L est régulier, $C(L)$ est-il nécessairement régulier ?

Proposition de corrigé

1. On a $x0 = 0010 = 0z_1$ donc x et z_1 sont conjugués. De plus, si deux mots x et z sont conjugués alors il existe y tel que $xy = yz$ et donc en particulier x et z doivent avoir la même longueur. Ce n'est pas le cas de x et z_2 donc ces mots ne sont pas conjugués.
2. Si z est conjugué à ε alors il existe y tel que $\varepsilon y = yz$ donc nécessairement $z = \varepsilon$. Réciproquement, ε est bien conjugué à ε, donc l'ensemble des mots conjugués à ε est $\{\varepsilon\}$.
3. Dans ces conditions, $xu = (uv)u = u(vu) = uz$ donc x et z sont conjugués via u.
4. Supposons par l'absurde que $|x| < |y|$. Comme on a $(\star)$: $xy = yz$, cette inégalité entre les longueurs indique que x est préfixe (strict) de y donc il existe un mot u tel que $y = xu$. Comme x contient au moins une lettre $|u| < |y|$. En injectant l'égalité $y = xu$ dans $(\star)$, on obtient $xxu = xuz$ et donc $xu = uz$ ce qui signifie que x et z sont conjugués via u qui est de longueur strictement inférieure à y, pourtant supposé de longueur minimale.
5. La réciproque a été montrée en question 3. Pour le sens direct :
   - Soit $x = \varepsilon$ et dans ce cas $z = \varepsilon$ d'après la question 2. Prendre $u = v = \varepsilon$ convient.

CC BY NC SA

- Soit $x \neq \varepsilon$. Dans ce cas il existe un y de longueur minimale tel que $xy = yz$ et d'après la question 4, $|x| \geq |y|$. On en déduit que y est préfixe de x donc qu'il existe $w \in V^\star$ tel que $x = yw$. On a alors $ywy = yz$ donc $z = wy$. On prend donc $u = y$ et $v = w$.

6. La relation de conjugaison est :
   - Réflexive puisque $x = x\varepsilon = \varepsilon x$.
   - Symétrique d'après la question 5. En effet, si x est conjugué à z, il existe u, v tels que $x = uv$ et $z = vu$ (question 5 sens direct) donc z est conjugué à x (question 5 sens réciproque).
   - Transitive en utilisant la définition. Si x est conjugué à z et z est conjugué à w alors il existe u, v tels que $xu = uz$ et $zv = vw$. On en déduit que $x(uv) = uzv = (uv)w$ et donc x est conjugué à w via le mot uv.

7. D'après la question 5, x a au plus $|x|$ conjugués (autant que de décomposition de la forme $x = uv$). On en déduit que $C(x)$ est fini donc qu'il est immédiatement régulier.

8. Dans le cas d'un alphabet à une lettre, on a $C(L) = L$ donc il suffit de choisir un langage L sur un alphabet unaire qui n'est pas régulier, par exemple $\{a^p \mid p \in \mathbb{P}\}$ ou $\{a^{(n^2)} \mid n \in \mathbb{N}\}$.

   Une autre solution sur un alphabet à deux lettres est de considérer $L = \{a^n b^n \mid n \in \mathbb{N}\}$. La caractérisation de la question 5 montre que tous les conjugués de $a^n b^n$ sont des permutations circulaires de ce mot (en particulier : ont autant de a que de b) donc soit contiennent un bloc de n lettres a, soit contiennent un bloc de n lettres b. En itérant dans ce facteur on peut donc construire des mots dont le nombre de a et de b ne coïncide plus et donc sortent de $C(L)$. Le lemme de l'étoile assure donc que $C(L)$ ne peut pas être régulier.

9. Si L est régulier, $C(L)$ l'est aussi. Voici deux preuves possibles.

   D'après la caractérisation de la question 5, $C(L) = \{z = vu \mid uv \in L\}$ Il est donc reconnu par un automate qui découpe (de manière non déterministe) l'entrée z en un préfixe v et le suffixe u correspondant et vérifie que $uv \in L$ avec l'automate pour L.

   Une réponse plus précise (dans le style de la construction pour $\sqrt{L}$). Soit $A = (V, Q, q_0, F, \delta)$ un automate déterministe et complet reconnaissant L. Un mot $z = vu$ est dans $C(L)$ si et seulement s'il existe $q \in Q$ tel que la lecture de u dans A mène de q_0 à q et celle de v dans A mène de q à un état final. En dupliquant A en $A' = (V, Q', q_0', F', \delta')$ de telle sorte que $q' \in Q$ corresponde à $q \in Q$, on a alors z qui est reconnu par $A_q = (V, Q \cup Q', q, \{q'\}, \delta \cup \delta' \cup \{(f, q_0', \varepsilon) \mid f \in F\})$. $C(L)$ est donc reconnu par un automate fini non déterministe constitué de la juxtaposition des A_q (pour tout $q \in Q$, or cet ensemble est bien fini).

CC BY NC SA

Exercice 2 Contrôle du flot

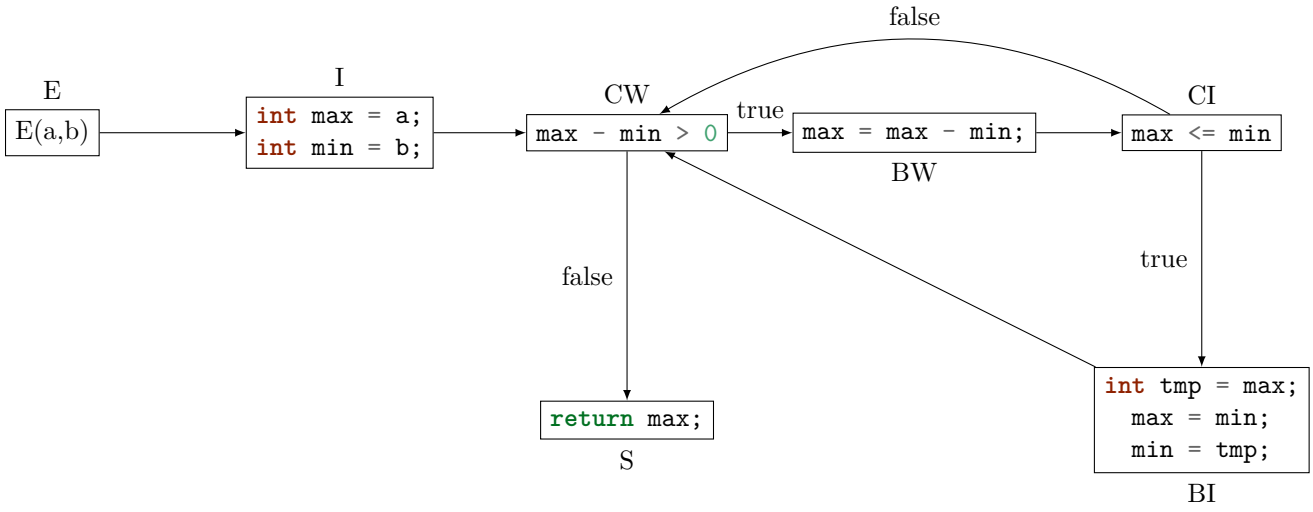
On considère le programme C suivant :

```
int f(int a,int b){
    int max = a;
    int min = b;
    while(max - min > 0){
        max = max - min;
        if(max <= min){
            int tmp = max;
            max = min;
            min = tmp;
        }
    }
    return max;
}
```

1. Donnez le graphe de flot de contrôle de ce programme.
2. Donnez un ensemble de chemins couvrant tous les sommets mais pas tous les arcs.
3. Donnez un ensemble de chemins couvrant tous les arcs. Est-ce possible d'en trouver un couvrant tous les arcs mais pas tous les sommets ?
4. À l'aide d'une exécution symbolique, déterminez une entrée réalisant un chemin de l'ensemble de la question 2.
5. Proposez une spécification pour la fonction `f`.
6. Donnez l'ensemble de tous les chemins réalisant au plus un tour de la boucle `while`.
7. Le programme est-il correct vis-à-vis de la spécification proposée ? Si oui, prouvez-le en vous appuyant sur un invariant bien choisi, sinon exhibez une entrée posant problème et corrigez le programme.

Proposition de corrigé

1. Ici E signifie entrée, S sortie, C condition, W while, B body et I if.



Les notations ne sont pas standardisées mais les points importants sont :

- Regrouper les instructions non séparées par une conditionnelle.
- Les instructions sont bien celles du programme de l'énoncé.

- Indiquer les résultats des tests sur les arcs sortant des sommets conditionnels pour prévenir toute confusion.
- La structure du graphe est bien celle du flot de contrôle du programme.
- Ne pas oublier un sommet de sortie contenant le `return`, un sommet d'entrée est appréciable.

2. E, I, CW, BW, CI, BI, CW, S convient, tous les sommets sont couverts mais pas tous les arcs (il manque CI, CW).
3. On rajoute E, I, CW, BW, CI, CW, S au chemin précédent. Tous les arcs et tous les sommets sont couverts. Si tous les arcs sont couverts, tous les sommets non isolés le sont aussi puisque pour couvrir un arc, il faut couvrir ses deux extrémités.
4. Pour l'ensemble proposé dans cette correction, on n'a pas le choix puisque l'ensemble ne contient qu'un chemin.
   - On part de a et b
   - en entrant dans CW, on a une première condition de chemin : $a - b > 0$
   - puis on change max en $a - b$
   - seconde condition de chemin : $a - b \leq b$
   - puis max devient b et min devient $a - b$
   - puis dernière condition de chemin avant la sortie : $b - (a - b) \leq 0$

   On doit donc résoudre $a - b > 0$, $a - b \leq b$ et $2b - a \leq 0$.

   On en déduit que a et b doivent vérifier $2b \geq a$ et $2b \leq a$, c'est-à-dire $a = 2b$. Pour $a = 2 > b = 1$, on a bien les conditions vérifiées, et l'algorithme renvoie alors $a - b = 1$.
5. Cet algorithme calcule le pgcd de a et b si ces deux arguments sont positifs ou nuls.
6. Vu les chemins précédents qui couvrent toutes les manières de faire exactement un tour de boucle, il ne reste qu'à ajouter le chemin qui ne rentre jamais dans la boucle : E, I, CW, S.
7. Le dernier chemin ajouté met en lumière un bug si on sort pour max - min < 0 avec une inégalité stricte due au fait que a n'est pas forcément le max au début et b pas forcément le min. Exemple avec $a = 6, b = 8$, on renvoie 6 mais le pgcd est 2.

   Pour le corriger, il suffit de dupliquer le if d'échange avant d'entrer dans la boucle.

   On pourrait alors prouver la correction avec l'invariant de boucle selon lequel min $\leq$ max et pgcd(max, min) = pgcd(a,b). En sortant de boucle on aurait alors max - min ≤ 0 et puisque max $\geq$ min, max = min. La valeur renvoyée serait donc bien le pgcd de max et min (puisqu'on a égalité) et donc de a et b. La modification apportée rend vrai l'invariant avant l'entrée dans la boucle.

BY NC SA

Exercices de type B

Exercice 3 Langford

Cet énoncé est accompagné d'un ou plusieurs codes compagnons en C fournissant certaines des fonctions mentionnées dans l'énoncé : il sont à compléter en y implémentant les fonctions demandées.

La ligne de compilation `gcc -o main.exe -Wall *.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 d'écrire* `make`*. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande* `./main.exe`*.*

Il est possible d'activer davantage d'avertissements et un outil d'analyse de la gestion de la mémoire avec la ligne de compilation `gcc -o main.exe -g -Wall -Wextra -fsanitize=address *.c -lm` *ou en écrivant* `make safe`*. L'examinateur pourra vous demander de compiler avec ces options.*

Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant `make clean` *et relancer une compilation.*

Dans ce sujet, on s'intéresse aux *permutations de Langford*. Pour $n \in \mathbb{N}^*$, une permutation de Langford d'ordre n est une permutation du multi-ensemble $\{0, 0, 1, 1, 2, 2, \ldots, n-1, n-1\}$ telle que pour tout $k \in [\![0, n-1]\!]$, il y a exactement k nombres entre les deux occurrences de k.

Par exemple $(3, 1, 2, 1, 3, 2, 0, 0)$ est une permutation de Langford d'ordre 4. Entre les deux occurrences de 0 il y a 0 nombres, entre les deux occurrences de 1 il y a 1 nombre, entre les deux occurrences de 2 il y a 2 nombres et entre les deux occurrences de 3 il y a 3 nombres.

1. Donner une autre permutation de Langford d'ordre 4, différente de $(3, 1, 2, 1, 3, 2, 0, 0)$.
2. Parmi les deux permutations suivantes, laquelle est une permutation de Langford d'ordre 5 ?
   (a) $(3, 4, 0, 0, 3, 2, 4, 1, 2, 1)$
   (b) $(3, 1, 4, 1, 3, 2, 0, 0, 2, 4)$
3. Écrire une fonction de prototype `int* initialise_tableau(int n, int v)` qui alloue un tableau d'entiers de taille $n \geq 1$ dont toutes les cases sont initialisées avec la valeur `v`. Que doit-on faire pour libérer un tableau alloué avec cette fonction ?

On représente une permutation du multi-ensemble $\{0, 0, 1, 1, 2, 2, \ldots, n-1, n-1\}$ par un tableau d'entiers de taille $2n$. On se propose d'écrire un programme vérifiant qu'un tableau d'entiers de taille $2n$ est une permutation de Langford. On procède en trois étapes : on vérifie d'abord que les valeurs du tableau sont entre 0 et $n-1$; puis qu'il s'agit bien d'une permutation du multi-ensemble ; et enfin qu'il s'agit bien d'une permutation de Langford. Des tests pour ces étapes sont proposés dans le code compagnon.

4. Écrire une fonction de prototype `bool valeurs_valables(int n, int* t)` qui vérifie si un tableau `t` d'entiers ne contient que des valeurs dans $[\![0, 1, 2, ..., n-1]\!]$. On suppose que `t` est un pointeur vers la première case d'un tableau d'entiers de taille $2n$. Quelle est la complexité dans le meilleur et dans le pire cas de cette fonction ?
5. Vérifier que votre programme passe le premier test.
6. Écrire une fonction de prototype `bool est_permutation(int n, int* t)` qui vérifie si un tableau `t` d'entiers est une permutation du multi-ensemble $\{0, 0, 1, 1, 2, 2, ..., n-1, n-1\}$. On suppose uniquement que `t` est un pointeur vers la première case d'un tableau d'entiers de taille $2n$. La complexité dans le pire cas de cette fonction doit être linéaire en n ce que l'on justifiera rapidement. *On ne demande pas ici de vérifier qu'il s'agit d'une permutation de Langford.*
7. Vérifier que votre programme passe le deuxième test.

CC BY NC SA

8. Écrire une fonction de prototype `bool est_langford(int n, int* t)` qui vérifie si un tableau `t` d'entiers est une permutation de Langford multi-ensemble $\{0, 0, 1, 1, 2, 2, ..., n-1, n-1\}$. On suppose uniquement que `t` est un pointeur vers la première case d'un tableau d'entiers de taille $2n$. La complexité dans le pire cas de cette fonction doit être linéaire en n ce que l'on justifiera rapidement.
9. Vérifier que votre programme passe le troisième test.
10. Compiler votre programme avec la ligne de compilation complète ou bien avec `make safe` et vérifier que vous n'obtenez aucune erreur de gestion de la mémoire à l'exécution.
11. Détailler comment on pourrait mettre en œuvre une méthode de retour sur trace (*backtracking*) pour afficher toutes les permutations de Langford d'ordre n. Donner le squelette de ce programme sous forme de pseudo-code. *On ne demande pas d'implémenter cette méthode.*
12. Montrer qu'une condition nécessaire pour avoir une permutation de Langford d'ordre $n \in \mathbb{N}^*$ est d'avoir n congru à 0 ou 1 modulo 4. *Indication : on pourra étudier la parité des indices où doivent se placer les deux occurrrences des nombres de* $[\![0, n-1]\!]$ *dans la permutation.*

Proposition de corrigé

1. Il suffit de mettre les 0 en premier et l'on obtient $(0, 0, 3, 1, 2, 1, 3, 2)$. On peut aussi prendre le miroir et obtenir $(0, 0, 2, 3, 1, 2, 1, 3)$, où l'on peut aussi placer les 0 à la fin et obtenir $(2, 3, 1, 2, 1, 3, 0, 0)$. Les deux dernières possibilités sont $(1, 2, 1, 3, 2, 0, 0, 3)$ et son miroir.
2. Seule la permutation (a) est une permutation de Langford. Dans la permutation (b) il y a 6 nombres entre les deux occurrences de 4 au lieu de 4.
3.
```c
int* initialise_tableau(int n, int v) {
  int* t = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++) {
    t[i] = v;
  }
  return t;
}
```
   On libère un tableau `t` alloué avec cette fonction avec `free(t)`.
4.
```c
bool valeurs_valables(int n, int* t) {
  for (int i = 0; i < 2 * n; i++) {
    if (t[i] < 0 || t[i] >= n) {
      return false;
    }
  }
  return true;
}
```
   Dans le meilleur cas cette fonction est en $\mathcal{O}(1)$ et dans le pire cas elle est en $\mathcal{O}(n)$.
5. RAS
6.
```c
bool est_permutation(int n, int* t) {
  if (!valeurs_valables(n, t)) {
```

```
    return false;
  }
  // On initialise un tableau d'occurrences à 0
  int* occ = initialise_tableau(n, 0);
  // On compte le nombre d'occurrences des éléments de {0, 1, ..., n -
  // 1}. Si on en trouve strictement plus que 2 on peut s'arrêter.
  for (int i = 0; i < 2 * n; i++) {
    if (occ[t[i]] > 1) {
      free(occ); // Ne pas oublier
      return false;
    }
    occ[t[i]]++;
  }
  // Si on arrive ici, on a réussi à placer les `n` valeurs au plus
  // deux fois dans `2n` cases donc exactement deux fois chacune
  free(occ);
  return true;
}
```

La complexité est bien linéaire puisque l'on effectue un appel à la fonction `valeurs_valables` en $\mathcal{O}(n)$, une initialisation d'un tableau en $\mathcal{O}(n)$ et une passe sur le tableau avec une complexité constante pour le corps de la boucle.

7. RAS

8.

```
bool est_langford(int n, int* t) {
  if (!est_permutation(n, t)) {
    return false;
  }
  // On ne veut tester que pour la première occurrence, on utilise un
  // tableau de marques.
  int* vu = initialise_tableau(n, 0);
  for (int i = 0; i < 2 * n; i++) {
    // On vérifie la condition pour les éléments non vu : on doit
    // trouver le même t[i] + 1 cases plus loin.
    if (vu[t[i]] == 0
        && (i + t[i] + 1 >= 2 * n || t[i] != t[i + t[i] + 1])) {
      free(vu); // Ne pas oublier
      return false;
    }
    vu[t[i]] = 1;
  }
  free(vu);
  return true;
}
```

Encore une fois, la complexité est bien linéaire puisque l'on effectue un appel à la fonction `est_permutation` en $\mathcal{O}(n)$, une initialisation d'un tableau en $\mathcal{O}(n)$ et une passe sur le tableau avec une complexité constante pour le corps de la boucle.

9. RAS

10. RAS

11. Seul un pseudo code est attendu mais les idées doivent être précises. Le code suivant est une

solution qui procède par retour sur trace.

```c
// Affiche toutes les permutations de Langford d'ordre `n` qui
// complètent la solution partielle `sol` sachant que l'on en est à la
// position `i` en ayant encore `r` entiers à considérer qui sont ceux
// marqués dans `vu`.
void parcours(int n, int i, int r, int* vu, int* sol) {
  if (r == 0) {
    // On a trouvé une solution
    print_tab(n, sol);
    return;
  }
  // On recherche la prochaine position à partir de laquelle remplir
  while (sol[i] != -1) {
    i++;
  }
  // On essaie de compléter cette position avec ce qui est possible.
  for (int j = 0; j < n; j++) {
    if (vu[j] == 0 && i + j + 1 < 2 * n && sol[i + j + 1] == -1) {
      vu[j] = 1;
      sol[i] = j;
      sol[i + j + 1] = j;
      parcours(n, i + 1, r - 1, vu, sol);
      // On n'oublie pas de défaire ce que l'on a fait
      sol[i + j + 1] = -1;
      sol[i] = -1;
      vu[j] = 0;
    }
  }
}

// Affiche toutes les permutations de Langford d'ordre `n`
void affiche_toutes_langford(int n) {
  int* sol = initialise_tableau(2 * n, -1);
  int* vu = initialise_tableau(n, 0);
  parcours(n, 0, n, vu, sol);
  free(vu);
  free(sol);
}
```

12. Dans la permutation, les deux occurrences d'un nombre pair de $[\![0, n-1]\!]$ sont placées à des indices de parité différente. En revanche, les deux occurrences d'un nombre impair sont placées à des indices de même parité. Il doit donc y avoir un nombre pair de nombre impairs dans $[\![0, n-1]\!]$, ce qui n'est possible que si n est congru à 0 ou 1 modulo 4.

CC BY NC SA

Exercice 4 Rendu de monnaie

Cet énoncé est accompagné d'un ou plusieurs codes compagnons en OCaml fournissant certaines des fonctions mentionnées dans l'énoncé : ils sont à compléter en y implémentant les fonctions demandées.

On s'intéresse au problème $\mathcal{P}$ suivant : étant donné un système monétaire, caractérisé par la valeur entière de ses pièces et une valeur $v \in \mathbb{N}$, déterminer le nombre minimal de pièces du système à utiliser pour rendre le montant v (on dispose d'autant de pièces qu'on veut de chaque valeur). Par exemple, pour le système monétaire $\mathcal{M}_0 = \{1, 2, 5, 10, 20, 50, 100, 200\}$, le nombre minimal de pièces permettant de rendre $v = 29$ est de 4 (une pièce de 20, une de 5 et deux de 2).

Les systèmes monétaires utilisés dans la suite disposeront toujours d'une pièce de valeur 1 de façon à ce que pour tout $v \in \mathbb{N}$ on puisse rendre le montant v (en utilisant v fois la pièce 1). Pour l'implémentation, on prendra garde au fait que la façon de représenter un système monétaire varie selon les questions.

1. À l'aide des informations mises à disposition dans le code compagnon, compléter la spécification de la fonction `verification`. Cette dernière contient des erreurs : les corriger.

Pour résoudre $\mathcal{P}$, on propose tout d'abord une approche gloutonne : tant qu'il reste quelque chose à rendre, choisir la plus grosse pièce qu'on peut rendre sans rendre trop.

2. Dans cette question, un système monétaire est représenté par la liste des valeurs possibles des pièces triée dans l'ordre décroissant des valeurs. Écrire une fonction récursive `rendu_glouton : int list -> int -> int` résolvant $\mathcal{P}$ selon cette stratégie étant donné un système et une valeur.

3. La fonction `rendu_glouton` renvoie-t-elle la valeur attendue pour le système $\mathcal{M}_0$ et $v = 352$? Qu'en est-il avec le système $\{1, 7, 8\}$ et $v = 14$? Commenter.

Dans la suite, un système monétaire à n pièces est représenté par un tableau trié dans l'ordre croissant à n cases $[|v_0; ...; v_{n-1}|]$ où v_i est la valeur de la pièce d'indice i. On note pour tout $w \in [\![0, v]\!]$ et tout $k \in [\![0, n-1]\!]$, $r(w, k)$ le nombre minimal de pièces permettant d'obtenir la valeur w en n'utilisant que les pièces ayant un indice dans $[\![0, k]\!]$. On admet (pour le moment) que pour tout $w \in [\![1, v]\!]$ et tout $k \in [\![1, n-1]\!]$:

$$(\star) \quad r(w,k) = \begin{cases} \min(r(w,k-1), 1 + r(w - v_k, k)) & \text{si } w \geq v_k \\ r(w,k-1) & \text{sinon} \end{cases}$$

On cherche à résoudre le problème $\mathcal{P}$ par programmation dynamique (d'abord avec une méthode descendante puis avec une méthode ascendante) en exploitant l'égalité $(\star)$.

4. Pour $k \in [\![0, n-1]\!]$, que vaut $r(0, k)$? Pour $w \in [\![0, v]\!]$ que vaut $r(w, 0)$? Compléter la fonction `rendu_descendant` en conséquence. Les fonctions du module `Hashtbl` sont décrites en annexe.

5. Écrire une fonction `rendu_ascendant : int array -> int -> int` résolvant le problème $\mathcal{P}$ par programmation dynamique ascendante. On rappelle que la création d'une matrice se fait via `Array.make_matrix`.

6. Quelle est la complexité temporelle de `rendu_ascendant` ? Et sa complexité spatiale ? Expliquer comment diminuer cette complexité spatiale à moindres frais. On ne demande pas l'implémentation correspondante et on pourra s'aider d'un dessin.

7. Écrire une fonction `repartition : int array -> int -> int array array -> int array` prenant en entrée un système monétaire $\mathcal{M}$, une valeur v et une matrice contenant en case (w, k) la valeur de $r(w, k)$ selon le système $\mathcal{M}$ et par rapport à la valeur v et qui renvoie un tableau contenant en case i le nombre de pièces d'indice i à utiliser dans un rendu optimal.

8. Expliquer pourquoi la formule $(\star)$ est correcte.

CC BY NC SA

Annexe

- `Hashtbl.create n` créé un dictionnaire de taille initiale n. La taille est automatiquement réajustée si on essaie d'insérer plus de n éléments dans le dictionnaire ainsi créé.
- `Hashtbl.mem d c` renvoie `true` si et seulement s'il existe une association dans le dictionnaire `d` de clé `c`.
- `Hashtbl.find d c` renvoie la valeur liée à la clé `c` dans le dictionnaire `d` s'il existe une association de clé `c`. L'exception `Not_found` est levée autrement.
- `Hashtbl.add d c v` ajoute une association dans le dictionnaire `d` entre la clé `c` et la valeur `v`.

Proposition de corrigé

1. Cette fonction vérifie que la somme atteinte en suivant les instructions du tableau `repartition` (c'est-à-dire en rendant `repartition.(i)` pièces d'indice `i`) permet effectivement de rendre le montant `v`.

   Il y a deux problèmes dans le code proposé :

   - `v_courante` est une référence, donc : accès avec `!v_courante` et modification avec `:=`.
   - La borne supérieure de la boucle pour est erronée.

   Voici une version corrigée :

   ```
   let verification (pieces:int array) (v:int) (repartition:int array) =
     let v_courante = ref 0 in
     let n = Array.length repartition in
     for i = 0 to n-1 do
       v_courante := !v_courante + repartition.(i) * pieces.(i)
     done;
     !v_courante = v
   ```

2. On suit la stratégie gloutonne. Le deuxième cas filtre le cas où il reste de la monnaie à rendre mais plus de pièces permettant de le faire : d'après l'énoncé il n'arrive jamais car la pièce de valeur 1 est en dernière position dans la liste `pieces` et permet de rendre n'importe quel montant. On pourrait être plus malin dans le troisième cas de filtrage et rendre immédiatement le maximum de pièces de valeur `t` possible.

   ```
   let rec rendu_glouton (pieces:int list) (v:int) :int =
     match pieces, v with
     |_, 0 -> 0
     |[], _ -> failwith "il devrait y avoir une pièce de 1 empêchant ce cas"
     |t::q, x when x >= t -> (rendu_glouton (t::q) (x-t)) + 1
     |t::q, x -> rendu_glouton q x
   ```

3. Le rendu est correct dans le premier cas : $352 = 200 + 100 + 50 + 2$. On ne peut pas faire mieux : pour dépasser 350 sans pièce de 200, il faut au moins 4 pièces. Ensuite pour dépasser 150 sans pièce de 100, il faut au moins 3 pièces, et pour dépasser 50 sans pièce de 50 il faut au moins 2 pièces.

   Le rendu n'est pas correct dans le second cas et c'est normal : la stratégie gloutonne propose d'utiliser une pièce de 8 puis est contrainte d'utiliser 6 pièces de 1 pour amener à 14 alors que 2 pièces de 7 suffiraient. On constate donc que la stratégie gloutonne ne renvoie pas toujours une solution optimale.

4. On a :

- Pour tout $k \in [\![0, n-1]\!]$, $r(0,k) = 0$ car le nombre minimal de pièces à rendre pour rendre 0 est 0 quelles que soient les pièces autorisées.
- Pour tout $w \in [\![0, v]\!]$, $r(w,0) = w$ car $r(w,0)$ est nombre minimal de pièces à utiliser pour rendre la valeur w en n'utilisant que la pièce 0 c'est à dire la pièce de valeur 1 (puisque d'après l'énoncé cette dernière est toujours présente et qu'on trie les valeurs des pièces dans l'ordre croissant).

On complète donc le code proposé à l'aide de `Hashtbl.add` :

```
...
  match w, k with
  |0, _ -> Hashtbl.add val_r (w,k) 0
  |_, 0 -> Hashtbl.add val_r (w,k) w
...
```

5. Pour calculer les $r(w,k)$ de manière ascendante, il suffit de déterminer dans quel ordre les calculer. On peut pour ce faire dessiner la matrice contenant les $r(w,k)$ et identifier quelles sont les cases de cette matrice qui doivent être remplies avant que la case (w,k) ne soit remplie :

$$\begin{pmatrix} r(0,0) & \cdots & r(0,k-1) & r(0,k) & \cdots & r(0,n-1) \\ \vdots & & \vdots & \vdots & & \vdots \\ r(w,0) & \cdots & r(w,k-1) & r(w,k) & \cdots & r(w,n-1) \\ \vdots & & \vdots & \vdots & & \vdots \\ r(v,0) & \cdots & r(v,k-1) & r(v,k) & \cdots & r(v,n-1) \end{pmatrix}$$

La valeur qu'on souhaite calculer est celle en bas à droite, encadrée en gras. Pour calculer la valeur $r(w,k)$, on a besoin d'après la formule $(\star)$ de connaître la valeur de la case juste à gauche et celle d'une des cases au-dessus puisque $w - v_k < w$, c'est à dire des valeurs qui sont forcément dans les cases grisées. On peut donc : soit calculer les valeurs dans cette matrice ligne par ligne de gauche à droite ; soit colonne par colonne de haut en bas.

C'est la première option qui est adoptée dans le code suivant :

```
let rendu_ascendant (pieces:int array) (v:int) =
  let n = Array.length pieces in
  let r = Array.make_matrix (v+1) n 0 in
  for w = 1 to v do
    r.(w).(0) <- w;
    for k = 1 to (n-1) do
      r.(w).(k) <- if w-pieces.(k) >= 0
                   then min r.(w).(k-1) (1 + r.(w-pieces.(k)).(k))
                   else r.(w).(k-1)
    done
  done;
  r.(v).(n-1)
```

L'initialisation de la matrice `r` initialise automatiquement les bonnes valeurs dans la première ligne car pour tout k, $r(0,k) = 0$ d'après la question 5. Attention, la matrice `r` a $v+1$ lignes...

6. La complexité temporelle comme spatiale de `rendu_ascendant` est en $O(nv)$. On peut réduire la complexité spatiale à un $O(v)$ en calculant les $r(w,k)$ colonne par colonne et en ne conservant que deux colonnes (ce qui est possible vu le dessin de la question 5).

CC BY NC SA

7. Pour savoir quelles pièces rendre en connaissant la matrice des $r(w,k)$, on "remonte" soit vers le haut soit vers la gauche dans cette matrice à partir du coin en bas à droite. En effet, un $r(w,k)$ est égal soit à $r(w,k-1)$ — auquel cas la pièce k n'est pas utilisée et on remonte à partir de $r(w,k-1)$ — soit à $r(w-v_k,k)$ — auquel cas on compte que la pièce k est utilisée une fois — et on poursuit la remontée à partir de $r(w-v_k,k)$. On poursuit jusqu'à ce qu'il ne reste plus rien à rendre ou qu'il n'y ait plus que la pièce de valeur 1 pour compléter le rendu :

```
let repartition (pieces:int array) (v:int) (r:int array array) :int array =
  let n = Array.length pieces in
  let rendu = Array.make n 0 in
  let rec remplir_repartition (w:int) (k:int) :int array = match w, k with
    |0, _ -> rendu
    |_, 0 -> rendu.(0) <- w; rendu
    |w, k when w-pieces.(k) >= 0 -> if r.(w).(k) = r.(w).(k-1)
                                    then remplir_repartition w (k-1)
                                    else begin
                                        rendu.(k) <- rendu.(k) + 1;
                                        remplir_repartition (w-pieces.(k)) k
                                      end
    |_, _ -> remplir_repartition w (k-1)
  in remplir_repartition v (n-1)
```

On obtient évidemment les mêmes cas de filtrage que dans l'algorithme descendant.

8. La formule $(\star)$ est vraie car, si $w < v_k$ alors on ne peut pas utiliser la pièce k pour rendre la monnaie sur w. Sinon, on a deux possibilités :

   - La solution optimale n'utilise pas la pièce k auquel cas on est ramené à trouver le minimum de pièces pour rendre w en utilisant les pièces jusque $k-1$.
   - Soit la solution optimale utilise une pièce k, auquel cas il nous reste à rendre $w-v_k$ avec les pièces jusqu'à k (au cas où la pièce k soit utilisée plusieurs fois).