DM 1

À rendre le 8 novembre en TD.

- Exercice 1 - Découpage électoral. Le parti des programmeurs dynamiques (PPD) se présente aux élections sur deux circonscriptions qui regroupent à elles deux un total de n zones (avec n pair). Chaque zone a m électeurs. Le nombre de voix du PPD dans une zone $i \in [n]$ (on note $[n] := \{1, \dots, n\}$) est estimé à v_i, avec $0 \leq v_i \leq m$. Le PPD peut choisir à sa guise le regroupement des zones en deux circonscriptions de tailles égales. Un *découpage gagnant* est une partition de $[n]$ en deux parties sur lesquelles le PPD à la majorité absolue.

Précisément, c'est une partie $I \subseteq [n]$ de taille $n/2$ telle que $\sum_{i \in I} v_i > m.n/4$ et $\sum_{i \in [n] \setminus I} v_i > m.n/4$.

Par exemple pour $n = 4$ et $m = 100$, si les estimations de vote sont $46, 60, 55, 44$, un découpage gagnant est 46+55 et 60+44 (regroupant donc les zones 1 et 3 et les zones 2 et 4).

1. Proposer une instance ayant 6 zones telle qu'il existe un unique découpage gagnant (on choisira ici $m = 100$).
   Correction: On a 6 zones avec 100 électeurs dans chacune. Prenons :

   $$(v_i)_{1 \leq i \leq 6} = (43, 57, 45, 54, 51, 52)$$

   L'unique découpage gagnant est donné par $I = \{1, 2, 5\}$. Une preuve n'est pas attendue mais prouvons le, histoire de fixer les notations.

   Soit $I \subset [6]$ un découpage (i.e. $|I| = 3$). On note

   $$J = \overline{I};\ S_I = \sum_{i \in I} v_i;\ S_J = \sum_{j \in J} v_j$$

   Quitte à échanger les rôles de I et J, supposons $1 \in I$. Si $2 \notin I$, alors,

   $$S_I \leq v_1 + v_4 + v_6 = 149$$

   impliquant que I ne soit pas gagnant. Ansi, si I est gagnant, il est nécessaire que $2 \in I$. Alors, $3 \notin I$, sans quoi $S_I = 145$. Donc, $3 \in J$. De manière similaire cela implique $4 \in J$ (sans quoi $S_J = 148$). On voit facilement que pour que le découpage soit gagnant il faut encore $5 \in I$ et $6 \in J$.
   Cette instance admet donc un unique découpage gagnant (à symétrie entre les deux circonscriptions près) donné par $I = \{1, 2, 5\}$.

2. Proposer un algorithme en $O(n^3 m)$ qui calcule tous les nombres possibles de voix des sous-ensembles de $n/2$ zones.
   Correction: Reprenons les notations J, S_I et S_J, et étendons les à

   $$S_K = \sum_{k \in K} v_K$$

   pour tout $K \subseteq [n]$. On note

   $$t(i, j, k) = \begin{cases} 1 \text{ si on peut partitionner } [i+j] \text{ en } K_1, K_2 \text{ de sorte à ce que } S_{K_1} = k \text{ et } |K_1| = i \\ 0 \text{ sinon} \end{cases}$$

   En particulier, $t(n/2, n/2, k)$ vaut 1 si et seulement s'il existe un découpage I tel que S_I la somme des voix des zones de I vaille k. Calculer $t(n/2, n/2, k)$ pour tout $0 \leq k \leq nm$ répond à la question.
   On a la formule récurrente suivante :

   $$\begin{cases} t(i, 0, k) = \begin{cases} 1 \text{ si } k = S_{[i]} \\ 0 \text{ sinon} \end{cases} \\ t(0, j, k) = \begin{cases} 1 \text{ si } k = 0 \\ 0 \text{ sinon} \end{cases} \\ t(i+1, j+1, k) = t(i, j+1, k - v_{i+j+2}) \vee t(i+1, j, k) \end{cases}$$

   Cette formule décrit un algorithme de programmation dynamique permettant de calculer $t(i, j, k)$ pour tout $0 \leq i, j \leq n/2$ et $0 \leq k \leq nm$ en $O(n^2 nm) = O(n^3 m)$.

3. Comment décider s'il existe un découpage gagnant en temps polynomial ?

Correction: Soit $I \subset [n]$ un découpage gagnant, c'est-à-dire une partie de $[n]$ de taille $n/2$ telle que :

$$S_I > \frac{nm}{4}$$
$$S_{\overline{I}} > \frac{nm}{4}$$

Notons $V = \sum_{i \in [n]} v_i$. Alors, $S_{\overline{I}} = V - S_I$.

Un découpage gagnant est donc exactement un découpage $I \subset [n]$ tel que $V - mn/4 > S_I > mn/4$.

Afin de déterminer l'existence d'un tel découpage, on peut commencer par appliquer l'algorithme de la question 2 en $O(n^3 m)$, avant de parcourir $t(n/2, n/2, k)$ pour k allant de $mn/4 + 1$ à $V - mn/4 - 1$. Ce parcours se fait en $O(mn)$. Il existe un découpage gagnant si et seulement si l'une des valeurs parcourue est à 1.

Cela décrit un algorithme en $O(n^3 m)$, c'est-à-dire un algorithme polynomial, décidant l'existence d'un découpage gagnant.

Sur la complexité de cette algorithme. Si l'on cherche à définir la complexité de ce problème, il faut commencer par l'encoder d'une manière correcte. Ici, le nombre m doit être encodé en binaire. En revanche n peut être encodé en unaire, car le problème vient avec n nombres v_i. Chaque v_i est encodé sur au plus $\log(m)$ bits. Une instance a donc une taille en $O(n \log m)$. Or $O(n^3 m)$ n'est pas polynomial en cette taille. L'algorithme proposé ici n'est polynomial en la taille de l'instance que si tous les nombres que cette dernière contient sont encodés en unaire. On a enfait décrit un algorithme *pseudopolynomial*.

- Exercice 2 - Indépendant. On se donne un graphe $G = (V, E)$ ainsi qu'une fonction de poids ω positive définie sur les arêtes. Un sous-ensemble d'arêtes F est *indépendant* si chaque composante connexe de F possède au plus un cycle (i.e. est un arbre ou un arbre plus une arête).

1. Montrer que si G a n sommets, alors un indépendant a au plus n arêtes.

Correction: Un arbre sur n sommets admet $n - 1$ arêtes exactement. C'est une propriété basique que l'on peut utiliser sans démonstration.

Soit F un indépendant de G. On peut voir F comme l'union disjointe de ses composantes connexes $C_1, \ldots, C_n$. Les sommets touchés par F sont donc ceux touchés par les C_is et les arêtes de F sont données par l'unions disjointe des arêtes des C_is. Si l'on considère que C_i touche n_i sommets, on a donc

$$\sum_i n_i \leq n$$

Par ailleurs, C_i, par définition d'indépendant, et par la propriété sur les arbres énoncée ci-dessus, a soit $n_i - 1$ soit n_i arêtes. Si l'on note a_F le nombre d'arêtes de F, et a_i celui de C_i, on a finalement,

$$a_F = \sum_i a_i \leq \sum_i n_i \leq n$$

2. Soit F un indépendant. On dit qu'un sommet $v \in V$ est *saturé dans* F s'il appartient à une composante de F qui possède un cycle. Montrer que si e est une arête de $E \setminus F$ qui ne relie pas deux sommets saturés, alors $F \cup \{e\}$ est indépendant.

Correction: À partir de maintenant, si F est un indépendant, on considérera un sommet isolé, i.e. non touché par F, comme une composante connexe de F.

Soit $(u, v) = e \in E \setminus F$ une arête ne reliant pas deux sommets saturés dans F. Notons F_u et F_v les composantes connexes de F dans lesquelles se trouvent u et v respectivement. Quitte à échanger u et v, supposons que u ne soit pas saturé : F_u n'admet pas de cycle, c'est un arbre.

Si $F_u = F_v$, alors v est dans une composante n'admettant pas de cycle non plus. L'arête e crée un cycle dans un arbre. La composante devient saturée dans $F \cup \{e\}$, mais ce dernier ensemble reste un indépendant.

Sinon, $F_u \neq F_v$. Deux cas sont possibles : soit F_v n'admet pas de cycle, soit elle en admet un unique.

Dans les deux cas, la nouvelle composante $F_u \cup F_v \cup \{e\}$ ne contient au plus cycle. Si ce n'était pas le cas, l'un des cycles passerait nécessairement par e, et il serait le témoin d'un chemin de u vers v dans F qui ne passerait pas par e. Mais u et v n'étant pas dans les mêmes composantes connexes dans F, ce n'est pas possible.

Donc $F \cup \{e\}$ reste indépendant dans ce cas aussi.

3. Montrer que l'ensemble des indépendants forme un matroide.

Correction: L'hérédité est assez évidente : on ne peut pas créer plus de cycles qu'il n'en existe en prenant un sous-graphe. Tout sous-ensemble d'un indépendant, décrivant un sous-graphe du premier, donne un ensemble de composantes connexes de G au sein desquelles soit il n'y a pas de cycle, soit il y en a un seul, nécessairement existant dans l'indépendant original.

On rappelle que l'on élargit la notion de composante connexe d'un indépendant en définissant un sommet touché par aucune arête de l'indépendant comme composante connexe.

Montrons la propriété d'échange : soient X et Y deux indépendants tels que $|X| < |Y|$. Supposons, par l'absurde, que pour tout $e \in Y \setminus X$, $X \cup \{e\}$ ne soit pas un indépendant. On notera $e = (u, v)$.

Regardons les composantes connexes saturées de X. On les note $C_1, \ldots, C_k$ et on note N les sommets qu'elles touchent.

Par hypothèse et par la question 2, pour tout $e \in Y \setminus X$, e relie des sommets des C_is. D'après la question 1, si l'on note n_i et a_i le nombre de sommets et d'arêtes de C_i, on a $a_i = n_i$. Considérons une arête e de Y. Soit e touche deux sommets de N, soit non. Dans le premier cas, par la question 1, il y a au plus $\sum_i n_i$ telles arêtes. Dans le second cas, $e \in X \cap Y$.

On peut donc diviser les arêtes de Y entre celle qui touche les sommets de N, et qui sont moins de $\sum_i n_i$ (or dans X il y a exactement $\sum_i n_i = \sum_i a_i$ telles arêtes) et les autres qui sont partagées par X et Y. Cela prouve donc $|X| \geq |Y|$. Absurde.

Donc il existe $e \in Y \setminus X$ telle que $X \cup \{e\}$ soit un indépendant.

Finalement $\emptyset$ est bien sûr un indépendant, assurant que l'ensemble des indépendants forme un matroïde.

4. Proposer un algorithme qui calcule un indépendant de poids maximum (on indiquera surtout comment les composantes connexes sont mises à jour).

   Correction: Le fait que l'ensemble des indépendants forme un matroïde nous permet, dans le cas où les arêtes sont pondérées, de définir l'algorithme glouton associé à la structure de matroïde et donnant un indépendant de poids maximal : on tri les arêtes, on part de l'ensemble vide, et on ajoute, tant que c'est possible, les arêtes de plus gros poids à l'ensemble courant ne changeant pas son caractère indépendant.

   Afin de décider si une arête peut ou non être ajoutée à l'ensemble courant, on pourra conserver une liste courante des composantes connexes ainsi que de celles d'entre elles qui sont saturées. On utilisera alors la question 2 : si les deux bouts d'une arête sont chacuns dans une composante saturée, alors on n'ajoutera pas l'arête.

- Exercice 3 - Arbre binaire. On se donne un arbre binaire A de racine r ayant n noeuds tel que chaque noeud i (interne ou feuille) possède une valeur $v_i > 0$. On note p la *hauteur* de A, c'est à dire la distance maximale de la racine à une feuille. Pour les problèmes suivants, décrire un algorithme de résolution aussi performant que possible, montrer sa validité, et calculer sa complexité.

1. Calculer un sous-ensemble X de noeuds de valeur totale maximale sans relation de parenté directe (i.e. pour toute paire de noeuds x, y de X le parent de y n'est pas x).

   Correction: Si $r \in X$ alors aucun des fils de la racine ne peut être dans X. Si $r \notin X$ alors au moins un fils de r est dans X, sans quoi on pourrait ajouter r à X et obtenir une meilleure solution. Soit $i \in A$ un sommet quelconque. Le même raisonnement s'applique si l'on cherche à résoudre le problème sur le sous-arbre de A enraciné en i.

   Notons i_1 et i_2 les fils de i, et i_{11}, i_{12} et i_{21}, i_{22} leur fils respectifs. Si l'on note S_i la solution optimale pour i, alors le raisonnement ci-dessus donne :

   $$S_i = \max\left(v_i + S_{i_{11}} + S_{i_{12}} + S_{i_{21}} + S_{i_{22}}, S_{i_1} + S_{i_2}\right)$$

   On définira $S_i = 0$ si i n'est pas bien défini (par exemple i_1 quand i est une feuille).

   Un algorithme de programmation dynamique calcul S_r en $O(n)$. Afin d'obtenir X il suffit de noter pour chaque nœud interne i si S_i prend le sommet i ou non durant l'algorithme. Un parcours de l'arbre permet alors de reconstituer X en $O(n)$ (notez que ce n'est pas non plus trivial, il ne faut pas prendre dans X tout sommet étiqueté 1).

2. Calculer un sous-ensemble X de noeuds de valeur totale maximale sans relation de descendance directe (i.e. tout chemin de la racine à une feuille contient au plus un élément de X).

   Correction: Si $r \in X$, alors $X = \{r\}$ car tout chemin de la racine à une feuille débute en r. Sinon, alors X contient au moins un élément dans chacun des sous-arbres associés aux fils de r. En effet si pour l'un de ses fils r_i aucun élément du sous-arbre n'était dans X, alors on pourrait ajouter r_i à X et obtenir une meilleure solution. Dire que tout chemin de r aux feuilles de l'un de ses fils r_i contient au plus un élément de X sans que ce soit r, c'est encore pareil que dire que tout chemin de r_i à ses feuilles contient au plus un élément de X.

   De tout ce paragraphe, notant S_i la valeur associée à la solution optimale pour le sous-arbre enraciné en $i \in A$ et i_1, i_2 ses deux fils, on déduit la formule récurrente suivante :

   $$S_i = \max\left(v_i, S_{i_1} + S_{i_2}\right)$$

   (encore une fois, $S_j = 0$ si j n'est pas bien défini).

   Ici pas besoin de programmation dynamique. On ne gagne rien sur un algorithme diviser-pour-régner. Malgré tout, on n'appliquera pas le théorème maître. À moins que l'arbre ne soit équilibré on n'obtient pas une formule utilisable. On se contente de remarquer qu'étant donnés S_{i_1} et S_{i_2} on calcule S_i en temps constant, que l'on calcule S_i pour i une feuille en temps constant, et qu'il faut faire cela pour chaque noeud de l'arbre une unique fois : on a donc une complexité en $O(n)$.

3. Calculer un minimum local de A (dont la valeur est inférieure ou égale à celles de son enfant gauche, son enfant droit, et de son parent, si applicable).

   Correction: Soient r_1 et r_2 les fils de la racine. Si $v_r \leq \min(v_{r_1}, v_{r_2})$, alors r est un minimum local et on a fini. Ce test se fait en temps constant. Sinon, c'est que pour au moins l'un des fils, disons r_1, on a $v_r \geq v_{r_1}$.

   Montrons alors que le sous-arbre enraciné en r_1 admet un minimum local dans A. Soit i un minimum local de A_{r_1} le sous-arbre de A enraciné en r_1. Si $i \neq r_1$, alors i est encore local dans A puisque ses voisins sont les mêmes dans A et dans A_{r_1}. Sinon, $i = r_1$. Puisque $v_{r_1} \geq v_r$, r_1 reste un minimum local dans A.

   Ce paragraphe nous donne un algorithme glouton : on part d'un sommet ; soit c'est un minimum local et on a fini, soit l'un de ses fils admet une valeur inférieur à la sienne, et alors on le choisi pour continuer.

   Si l'algorithme s'arête avant d'avoir atteint une feuille, c'est qu'il a trouvé un minimum local. Sinon on atteint une feuille. Dans ce cas c'est que l'algorithme a refusé le parent de la feuille, indiquant que la valeur de la feuille est plus faible que celle de son parent, et donc qu'il s'agit d'un minimum local.

   Dans le pire des cas on parcourt l'arbre dans sa profondeur, chaque étape se faisant en temps constant. Cela donne donc un algorithme en $O(p)$.

4. (Bonus) Calculer un minimum local dans la grille $n \times n$ en temps $O(n)$.

   Correction: On commence par considérer la méthode naïve : on part d'une cellule de la grille quelconque puis on itère la procédure suivante :

— regarder si la cellule courante est un mimimum local.
— Si oui, terminer.
— Si non, remplacer la cellule courante par sa voisine de plus petite valeur.

La valeur de la cellule courante diminuant strictement à chaque itération parmis un nombre finis de valeurs, cette procédure termine et donne un minimum local.

Le problème est que sa complexité est en $O(n^2)$.

Pour obtenir un meilleur algorithme, on considère la ligne et la colonne du milieu (d'indice $\lfloor n/2 \rfloor$). On cherche, parmis les valeurs qu'elles comprennent, le minimum (ce que l'on fait en $O(n)$).

Étant donné ce minimum on regarde ses voisins. S'il s'agit d'un minimum local, on termine. C'est le cas si on tombe sur la cellule commune à la colonne et à la ligne. Sinon il admet un plus petit voisin ne se trouvant ni sur la ligne du milieu, ni sur la colonne du milieu. On choisit le quart de la grille déterminé par la colonne et la ligne du milieu contenant ce plus petit voisin, et on itère.

La correction de l'algorithme vient de la propriété suivante : le quart de la grille choisi contient un minimum local. En effet, si on lui appliquait l'algorithme naïf ci-dessus en partant du plus petit voisin, le chemin décrit par la procédure ne pourrait passer ni par la colonne du milieu ni par la ligne du milieu. Le minimum obtenu serait donc bien dans le quart choisi.

Après $O(\log(n))$ étapes il ne reste qu'une grille 1×1 décrivant nécessairement un minimum local. La i-ème étape nécessite la recherche d'un minimum sur une union colonne/ligne de taille $O(n/2^i)$.

Or la somme $\sum_{i=1}^{N} \frac{1}{2^i}$ est majorée par $\sum_{i=1}^{\infty} \frac{1}{2^i} = 2$, de sorte que notre complexité est en $O(n)$.

- Exercice 4 - Rendu de la monnaie. On considère le problème du rendu de la monnaie. Étant donne une somme à rendre $S \in \mathbb{N}$ et un système monétaire $P := \{p_1 > p_2 > \ldots > p_n\}$ avec $\forall i \in [n], p_i \in \mathbb{N}$ et $p_n := 1$, on cherche à minimiser le nombre de pièces pour rendre la somme S en utilisant autant de fois que l'on souhaite chaque pièce. Formellement, cela revient à minimiser $\sum_{i=1}^{n} \lambda_i$ sous la contrainte $\sum_{i=1}^{n} \lambda_i p_i = S$ et $\forall i \in [n], \lambda_i \in \mathbb{N}$. On appellera $N_P(S)$ ce minimum.

1. Calculer $N_P(S)$ à la main pour :
   (a) $S = 9$ et $P = \{5, 2, 1\}$.
   Correction: $N_{\{5,2,1\}}(9) = 3$.
   (b) $S = 6$ et $P = \{4, 3, 1\}$.
   Correction: $N_{\{4,3,1\}}(6) = 2$.
2. Exprimer $N_P(S)$ en fonction des $N_P(S - p_i)$. En déduire un algorithme de programmation dynamique qui résoud le problème en temps $O(Sn)$.
   Correction: Si l'on prend une pièce de valeur p_i dans la solution, il reste encore à obtenir la somme $S - p_i$ en choisissant d'autres pièces. Notons que p_i ne peut être utilisée que si $S - p_i \geq 0$. Si $S \neq 0$, il faut toujours ajouter au moins une pièce. On en déduit la formule suivante :

$$N_P(S) = \begin{cases} 0 \text{ si } S = 0 \\ \min_{i=1, p_i \leq S}^{n} \{1 + N_P(S - p_i)\} \end{cases}$$

   On peut alors calculer $N_P(T)$ pour tout $0 \leq T \leq S$ par un algorithme de programmation dynamique, qui utilise à chaque étape la formule précédante pour calculer $N_P(T)$ sachant $N_P(T')$ pour tout $T' < T$. Chacune de ces étapes nécessite le calcul d'un minimum en $O(n)$. On a S valeurs à calculer. Cet algorithme est donc en $O(nS)$.

L'algorithme précédent est malheureusement seulement *pseudo-polynomial*, car l'entier S, codé en binaire, est de taille $\log(S)$. L'objectif à présent est de trouver une solution polynomiale en n et la taille de S (la taille des p_i étant bornée) dans certains cas. Nous allons nous intéresser à l'approche gloutonne qui consiste à prendre la plus grande pièce tant que c'est possible.

1. Prouver que si $\{\lambda_i^G\}_{i \in [m]}$ est la solution donnée par l'algorithme glouton, alors $\forall i \in [n-1], \lambda_{i+1}^G < \frac{p_i}{p_{i+1}}$.
   Correction: Puisqu'on a une solution, $\sum_{i \in [m]} \lambda_i^G p_i = S$. Le fait que la solution soit celle de l'algorithme glouton donne que pour tout $i \in [m-1]$, $\sum_{j \geq i+1} \lambda_j p_j < p_i$. En particulier, $\lambda_{i+1} p_{i+1} < p_i$ et donc :

$$\lambda_{i+1} < \frac{p_i}{p_{i+1}}$$

   pour tout $i \in [m-1]$.
2. Pour les systèmes monétaires P suivants, prouver que l'algorithme glouton est correct ou exhiber un contre-exemple S (les méthodes numériques pour trouver un tel contre-exemple sontt autorisées) :

(a) $P = \{5, 2, 1\}$.

Correction: L'algorithme glouton, par la question 1, renvoie au plus une pièce de 1, et deux pièces de 2. Mieux, une solution qui renvoie au plus une pièce de 1 et au plus deux pièces de 2 est la solution gloutonne si et seulement s'il n'y a pas à la fois deux pièces de 2 et une pièce de 1 (sans quoi on peut les remplacer par une pièce de 5).

Soit une solution qui n'est pas gloutonne. Dans le cas mentionné ci-dessus on voit que la solution n'est pas optimale car trois pièces peuvent être remplacées par une unique pièce de 5. Sinon, notre solution tombe dans l'un des deux cas suivants.

i. Elle utilise au moins deux pièces de 1 ; ces deux pièces peuvent être remplacées par une unique pièce de 2 de sorte que la solution ne soit pas optimale.

ii. Elle utilise au moins trois pièces de 2 ; ces trois pièces peuvent être remplacées par une pièce de 5 et une pièce de 1 de sorte que la solution ne soit pas optimale non plus.

Toute solution qui n'est pas gloutonne n'est pas optimale. Donc l'algorithme glouton fournie une solution optimale.

(b) $P = \{2^{m-i}\}_{i\in[m]}$.

Correction: On raisonne de manière similaire. Par la question 1, l'algorithme glouton renvoie au plus une pièce de valeur 2^{m-i} pour $i > 1$ (car $p_{i-1}/p_i = 2$ ici).

Inversement, une solution renvoyant au plus une pièce de valeur 2^{m-i} pour $i > 1$ est nécessairement la solution gloutonne.

En effet, cette propriété implique que $S = \lambda_1 2^{m-1} + S'$ où $S' \leq \sum_{i=2}^{m} 2^{m-i} < 2^{m-1}$. La division euclidienne indique qu'une telle écriture existe et est unique. Ainsi les deux solutions partagent λ_1 comme nombre de pièces de valeurs 2^{m-1}. Ce raisonnement s'appliquant pour tout m et S, on recommence avec $m-1$ et S' et ainsi de suite. In fine les deux solutions coïncident et toute solution telle que $\lambda_i \leq 1$ pour tout $i > 1$ est la solution gloutonne.

Ceci étant dit, considérons une solution qui n'est pas la gloutonne. Alors, pour un certains $i > 1$, cette solution prend $\lambda_i \geq 2$ pièces de valeur p_i. Or, deux pièces de valeur 2^{m-i} peuvent toujours être remplacées par une unique pièce de valeur 2^{m-i+1}. Une telle solution n'est pas optimale.

L'algorithme glouton est donc optimal encore une fois.

(c) $P = \{200, 149, 33, 1\}$.

Correction: Comme contre-exemple prenons 298. La solution optimale et bien sûr de prendre deux pièces de 149.

En revanche, l'algorithme glouton prend une pièce de 200, deux de 33, et trente-deux de 1.

(d) (Bonus) $P = \{F_{m-i+1}\}_{i\in[m]}$ pour $F_1 = 1, F_2 = 2, F_{k+2} = F_{k+1} + F_k$.

Correction: On a $F_{k+2}/F_{k+1} = 1 + F_k/F_{k+1}$. La suite F_k étant strictement croissante, on obtient par la question 1 que l'algorithme glouton renvoie au plus une pièce de valeur F_{m-i+1} pour $i > 1$. De plus, supposons que λ_i et λ_{i+1} valent 1 dans la solution donnée par glouton pour $i > 1$. Par définition de F_k, $\lambda_i p_i + \lambda_{i+1} p_{i+1} = F_i + F_{i+1} = F_{i+2}$. Glouton aurait renvoyé une pièce de valeur F_{i+2} au lieu des pièces de valeurs F_i et F_{i+1}.

Ainsi glouton renvoie au plus une pièce de valeurs F_i pour $i > 1$ et il ne peut en renvoyer qu'au plus une pour deux valeurs consécutives.

Soient λ_i pour $i > 1$ des valeurs valant au plus 1 telles qu'il n'y en ait pas deux consécutives valant 1. Alors, $\sum_{i=2}^{m} \lambda_i F_i < F_m$. On montre cela par récurrence sur m (ce que l'on laisse au lecteur).

Supposons que l'on ait une autre solution vérifiant les mêmes propriétés que glouton : $\lambda_i \leq 1$ pour $i > 1$ et pas deux λ_i consécutifs valant 1 pour $i > 1$. On peut écrire $S = qF_m + r$ avec $r < F_m$ cette écriture étant unique par division euclidienne. La propriété ci-dessus assure que les deux solutions prennent chacune q pièces de valeurs F_m. En itérant on montre qu'elles sont égales et que la solution gloutonne est entièrement caractérisée par ses propriétés montrées ci-dessus.

Soit une solution ne vérifiant pas les propriétés ci-dessus. On peut lui appliquer l'algorithme suivant :

- si un λ_i supérieur ou égal à 2 existe pour $i > 1$,
  - soit $i = m$ et on remplace λ_m par $\lambda_m - 2$ et λ_{m-1} par $\lambda_{m-1} + 1$, obtenant une meilleure solution ($p_m = 1$ et $p_{m-1} = 2$),
  - soit $i = m-1$ et on remplace λ_{m-1} par $\lambda_{m-1} - 2$, λ_{m-2} par $\lambda_{m-2} + 1$ et λ_m par $\lambda_m + 1$, obtenant le même nombre de pièces dans la nouvelle solution ($p_{m-2} = 3$),
  - soit $i < m-1$ et on remplace λ_i par $\lambda_i - 2$, λ_{i-2} par $\lambda_{i-2} + 1$, et λ_{i+1} par $\lambda_{i+1} + 1$, obtenant une solution équivalente ;
- sinon s'il existe $\lambda_i = \lambda_{i+1} = 1$ pour $i > 1$, on remplace les deux pièces par une pièce de la valeur juste supérieure

et on répète. On laisse au lecteur le soin de vérifier que cette procédure termine.

Partant d'une solution on finit par tomber sur une solution au moins aussi bonne vérifiant les propriétés de la solution gloutonne. On a vu qu'il s'agit nécessairement de cette dernière : la solution gloutonne est donc optimale.

3. On appelle les systèmes monétaires pour lesquels l'algorithme glouton est correct des systèmes *canoniques*. Prouver que si P n'est pas canonique, alors il existe un contre-exemple S de taille inférieure à $p_1 \sum_{i=2}^{n} p_i$.

Correction: Soient S de taille supérieure à $p_1 \sum_{i=1}^{n} p_i$ et $\lambda_1, \ldots, \lambda_n$ une solution telle que

$$S = \sum_{i=2}^{n} \lambda_i p_i \geq p_1 \sum_{i=2}^{n} p_i.$$

J'affirme qu'il existe $i \geq 2$ tel que $\lambda_i \geq p_1$. En effet, si j est tel que $\lambda_j = \max_{n \geq i \geq 2} \lambda_i$, alors

$$p_1 \sum_{i=2}^{n} p_i \leq \sum_{i=2}^{n} \lambda_i p_i \leq \lambda_j \sum_{i=2}^{n} p_i$$

prouvant $\lambda_j \geq p_1$.

Or, si $\lambda_i \geq p_1$, on peut remplacer p_1 pièces de valeur p_i, par $p_i < p_1$ pièces de valeur p_1.

Donc, pour tout $S \geq p_1 \sum_{i=2}^{n} p_i$ admettant une solution optimale

$$S = \sum_{i=1}^{n} \lambda_i p_i$$

on a

$$\sum_{i=2}^{n} \lambda_i p_i < p_1 \sum_{i=2}^{n} p_i$$

Supposons que l'algorithme glouton soit optimal sur toute valeur de taille strictement inférieure à $p_1 \sum_{i=2}^{n} p_i$. Soient S un nombre et $\lambda_1, \ldots, \lambda_n$ sa solution gloutonne associée. Alors, la division euclidienne de S par $p_1(\sum_{i=2}^{n} p_i) = N$ donne

$$S = qN + r$$

avec $r < N$.

La solution étant la gloutonne, $\lambda_1 \geq q(\sum_{i=2}^{n} p_i)$ de sorte que $\sum_{i=2}^{n} \lambda_i p_i \leq r < p_1(\sum_{i=2}^{n} p_i)$.

Donc, glouton commence par prendre au moins un certain nombre de pièces de valeur p_1 avant de résoudre le problème sur $S' = \sum_{i=2}^{n} \lambda_i p_i$. Or, la solution optimale λ'_i fait de même puisque $\sum_{i \geq 2} \lambda'_i p_i < p_1 \sum_{i \geq 2} p_i$.

Mais par hypothèse, glouton renvoie la solution optimale sur S'.

Donc glouton est optimal sur tout S.

Par contraposée, si glouton n'est pas canonique, alors il existe un contre-exemple de taille strictement inférieure à $p_1 \sum_{i=2}^{n} p_i$.

4. Conclure : donner un algorithme en temps polynomial en n et $\log(S)$ (la taille des p_i étant bornée par une constante C) qui vérifie que le système monétaire P est canonique et qui, le cas échéant, résoud le problème du rendu de la monnaie sur S.

   Correction: Par la question 3 il suffit de regarder si glouton est optimal ou non sur les $p_1(\sum_{i=2}^{n} p_i) \leq nC$ exemples les plus petits pour déterminer si glouton est canonique ou non.

   Étant donnée une de ces instances, on sait que sa taille est limitée par un $O(nC) = O(n)$. L'algorithme de programmation dynamique de la question 2 permet de vérifier si glouton est optimal en $O(n^2C) = O(n^2)$. Puisqu'on a $O(n)$ vérifications à faire, on peut toutes les faire en $O(n^3)$ et vérifier si le système est canonique en temps polynomial.

   On peut utilise alors, le cas échéant, l'algorithme glouton pour résoudre le problème en temps polynomial en n et $\log(S)$.