TP de Python numéro 7 : Algorithmes gloutons

*Semaine du **jeudi 8 janvier**.*

Le but de ce TP est de découvrir la notion d'algorithme glouton, permettant de proposer des solutions à des problèmes de décisions, et d'appliquer ces algorithmes dans certaines situations classiques.

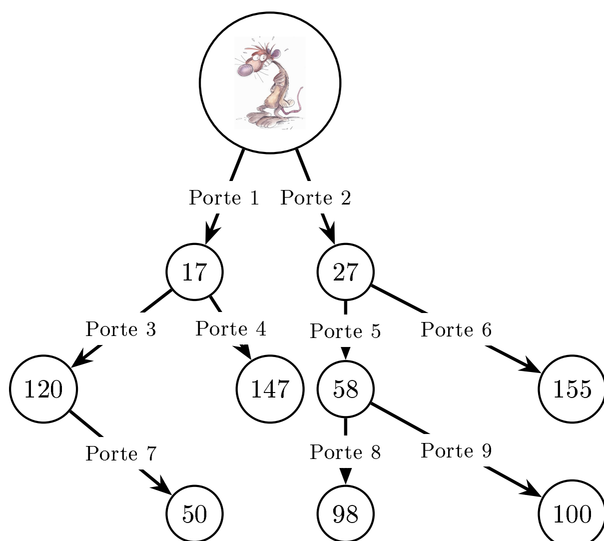
I. Algorithmes gloutons : principe général et premier exemple

1. Des problèmes de choix

Dans ce TP, nous allons nous intéresser à des problèmes que nous pouvons décrire comme étant **des problèmes de choix**. D'une manière générale, ce sont des situations qui nous confrontent, pour être résolues :

- à des choix successifs,
- avec la nécessité qu'une fois cette succession de choix effectués, nos choix soient les meilleurs possibles (**optimalité** des choix).

Exemple 1. Un rat mathématicien se trouve au début d'un parcours constitué d'une série de portes. Derrière chaque porte se trouve un tas de riz, et éventuellement d'autres portes. Lorsque notre rat choisit une porte, celle-ci se referme derrière lui. Il doit choisir, une à une, les portes à emprunter et veut naturellement avoir récolté le plus de riz possible lors de son parcours.
Les disposition des portes et les quantités de riz (en nombre de grains) sont représentées par le graphe suivant.



Ce plan est dans la première salle où est le rat, et celui-ci étant mathématicien, il est ici capable de considérer toutes les possibilités pour choisir la meilleure. Quelle suite de portes le rat emprunte-t-il pour maximiser son nombre de grains de riz ?

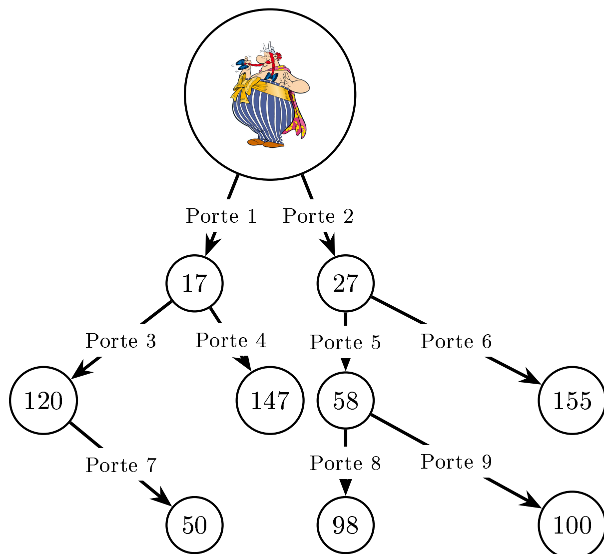
Dans ce cas simple, on est capable de considérer toutes les possibilités pour choisir la meilleure.

Le problème est que, si le nombre de possibilités est beaucoup plus grand (imaginez la situation précédente avec 1001 portes) ou si les nombres en jeux sont plus pénibles à calculer (imaginez une situation avec des nombres de grains de riz à 14 chiffres...), ce temps de calcul (et la mémoire nécessaire pour retenir tous les résultats) peut devenir beaucoup trop important : on construit facilement des problèmes où les ordinateurs actuels prendraient des milliards d'années pour considérer toutes les possibilités. Par exemple, il n'est pas, à ce jour, envisageable de faire calculer toutes les parties d'échecs possibles à un ordinateur.

2. Algorithmes gloutons

On appelle **stratégie gloutonne** une stratégie permettant de proposer une solution à un problème de choix, construite sur le principe suivant : à chaque fois qu'un choix est à faire, on fait le choix qui nous semble immédiatement le plus profitable.

Exemple 2.



Cette fois-ci, c'est Obélix qui est confronté à ce problème (avec des sangliers). Il ne sait pas très bien compter, et il a très faim, ce glouton. Alors, il adopte la stratégie suivante : à chaque choix qu'il a à faire, il ouvre la porte derrière laquelle il trouvera le plus de sangliers.
Quelle suite de portes va-t-il ouvrir, et combien de sangliers pourra-t-il ainsi ingurgiter ? Cette stratégie est-elle optimale ?

Un **algorithme glouton** est tout simplement un algorithme (qu'on implémentera en Python) proposant une solution à un problème de choix, en suivant une stratégie gloutonne.

3. Optimalité

Comme on l'a vu, une stratégie gloutonne permet de proposer une solution à un problème de choix, mais cette solution n'est pas forcément optimale : il peut exister de meilleures stratégies, et c'est souvent le cas.

Dans le cadre du programme :

- vous devez être conscient de cette problématique,
- vous devez pouvoir éventuellement constater qu'une stratégie gloutonne n'est pas optimale,
- et vous devez connaitre quelques situations simples pour lesquelles une stratégie gloutonne bien choisie est optimale.

Il existe bien d'autres stratégies de résolution existantes pour les problèmes de choix, qui sont souvent plus couteuses en calcul mais qui mènent aussi plus souvent à un résultat optimal.

4. Un premier exemple : le problème du rendu de monnaie

Un premier exemple

On va s'intéresser au problème suivant : comment rendre la monnaie à un client en donnant le moins de pièces ou billets possibles ?

Pour simplifier, dans la suite :

- le mot "pièce" désignera aussi bien des billets que des pièces.
- On considérera que les magasins ont un stock infini de pièces de chaque valeur.
- On considérera que, dans la zone euro, les échanges se font avec des pièces de valeurs 1€, 2€, 5€, 10€, 20€, 50€, 100€, et 200€ (pas de centime dans les pièces ou dans les prix, ni de billet de 500 €).

Exercice 3. Dans un magasin, un client règle un achat de 41€ avec une pièce de 50€.

1. Dresser la liste complète des manières dont le vendeur dispose pour lui rendre la monnaie.
2. Parmi celles-ci, laquelle utilise le moins de pièces possibles ?

On remarque alors que la stratégie suivante permet, au moins dans ce cas, de rendre le moins de pièces possibles:

- Déterminer la valeur de la plus grande pièce qu'on peut rendre - c'est-à-dire inférieure à la somme à rendre,
- Rendre cette pièce tant que la somme à rendre lui est supérieure,
- Recommencer de la sorte jusqu'à avoir rendu toute la monnaie.

Exercice 4. 1. En quoi cette stratégie est-elle une stratégie gloutonne ?

2. Pour quelle raison simple cette stratégie permet-elle de toujours proposer un rendu de monnaie dans la zone euro ?

Exercice 5. Une autre zone monétaire utilise les mêmes pièces que la zone euro, à l'exception des pièces de 2€ et 5€ remplacées par des pièces de 3€ et 4€.

Un client règle un achat de 14€ avec un billet de 20€.

1. Dresser la liste complète des manières dont le vendeur dispose pour lui rendre la monnaie.
2. Parmi celles-ci, laquelle utilise le moins de pièces possibles ?
3. Parmi celles-ci, laquelle utilise la stratégie gloutonne ci-dessus ? Que remarquez vous ?

On dit qu'un système monétaire (c'est-à-dire, en ensemble de pièces et de billets choisi pour faire des échanges au sein d'une population) est **canonique** si la stratégie gloutonne décrite ci-dessus fournit toujours la solution optimale au problème de rendu de monnaie, c'est-à-dire le rendu de monnaie utilisant le moins de pièces ou billets possibles.

On peut démontrer que le système monétaire utilisé dans la zone euro est canonique, et l'exercice 3 fournit un exemple de système monétaire non canonique. La valeur des pièces et des billets utilisés dans la zone euro n'est pas due au hasard...

Généralisation et implémentation

Pour modéliser le problème de rendu de monnaie en général, on considère que :

- Un système monétaire V est donné par les valeurs $v_1, ..., v_n$ des pièces et billets qui le constituent. Pour l'implémentation en Python, un tel système monétaire sera donné par la liste
$$V = [v_1, ..., v_n]$$
où les valeurs sont rangées par ordre décroissant ($v_1 > v_2 > ... > v_n$). On considérera pour simplifier que ces valeurs sont entières. L'entier n est le nombre de pièces ou billets.
- Une somme à rendre est simplement un entier S, donc une variable de type `int` en Python.
- Un rendu de monnaie X est la donnée, pour tout $i \in [\![1, n]\!]$, d'un entier naturel x_i donnant le nombre de fois que la pièce de valeur v_i a été rendue. En Python, un tel rendu de monnaie sera donné par la liste
$$X = [x_1, ..., x_n].$$
- On dit, avec ces notations, donc que le rendu de monnaie X rend la somme S (dans le système monétaire V) si $S = \sum_{i=1}^{n} x_i v_i$.

Enfin, on considérera que tous nos systèmes monétaires contiennent une pièce de valeur 1 (sans quoi, beaucoup de problèmes se posent).

On rappelle que le système monétaire utilisé dans la zone euro est canonique.

Exercice 6. Écrire un code Python permettant de construire le rendu de monnaie `[`x_1`,...,`x_8`]` optimal rendant la somme $S = 98$€ dans la zone euro, à l'aide d'un algorithme glouton.

Exercice 7. 1. Écrire le code d'une fonction Python d'entête `def Rendu_Glouton(V,S):` prenant en entrée un système monétaire `V` et une somme à rendre `S`, et renvoyant en sortie le rendu de monnaie `X` rendant la somme `S` (dans le système monétaire `V`) obtenu en appliquant la stratégie gloutonne.

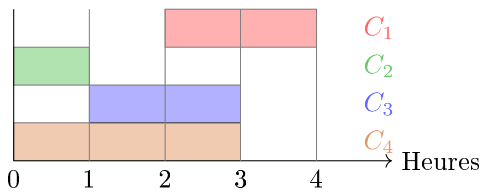
2. Tester cette fonction pour le système monétaire de la zone euro pour rendre 49€.
3. Tester cette fonction pour le système monétaire `V=[30,24,12,6,3,1]`, en vigueur au Royaume-Uni jusqu'en 1971, pour rendre la somme $S = 49$. Ce rendu de monnaie est-il optimal ? Que dire de ce système monétaire ?

II. Problème de réservation d'une salle

Le contexte est le suivant : vous êtes gestionnaire d'un centre de conférences. Vous recevez des demandes de réservation pour votre salle principale, de la part de conférenciers. Chaque conférencier vous fait une demande pour un créneau horaire, et votre but est de **maximiser le nombre de conférenciers** qui interviendront dans votre salle principale.

Commençons par un premier exemple. Vous traitez les demandes pour une matinée de quatre heures, de l'heure notée 0 (pour 8h) à l'heure notée 4 (pour 12h).

Vous recevez les demandes de 4 conférenciers C_1, C_2, C_3 et C_4. Les créneaux horaires demandés par ces conférenciers sont représentés ci-dessous.



Tenant compte des incompatibilités entre ces demandes, vous dressez la liste des conférenciers que vous pourriez retenir. Vous observez que vos possibilités sont les suivantes :

- Retenir C_2 et C_1,
- Retenir C_2 et C_3,
- Retenir C_4

(les autres possibilités étant immédiatement exclues).

Dans le but de maximiser le nombre de conférenciers, les deux premières options sont bonnes mais la troisième est à exclure.

1. Quelques stratégies gloutonnes

On va voir quelques stratégies gloutonnes permettant de proposer une solution à ce problème.

Le principe commun à ces stratégies est le suivant :

- On classe les conférenciers selon un ordre de préférence,
- puis on remplit notre planning en prenant un à un le conférencier préféré parmi les conférenciers dont le créneau n'entre pas en conflit avec ceux déjà choisi.

Les stratégies diffèrent uniquement selon la manière de classer les conférenciers. Voici les classements auxquels on pense.

- Stratégie (S_1) : On classe les conférenciers par ordre croissant d'heure de début de conférence (plus un conférencier commence tôt, plus on veut le choisir).
- Stratégie (S_2) : On classe les conférenciers par durée croissante de conférence.
- Stratégie (S_3) : On classe les conférenciers par ordre croissante d'incompatibilité avec les autres conférenciers.
- Stratégie (S_4) : On classe les conférenciers par ordre croissant d'heure de fin de conférence.

En cas d'égalité entre des conférenciers, ces stratégies ne décrivent pas l'ordre dans lequel les placer. On décrira un planning par une liste donnant les conférenciers gardés.

Exemple 8. Reprenons l'exemple précédent et appliquons la stratégie (S_1).
- Le classement des conférenciers donne $C_2 = C_4 < C_3 < C_1$. L'ordre entre C_2 et C_4 n'est pas précisé par la stratégie (S_1), admettons que notre algorithme place C_2 en premier.
- On remplit notre planning avec C_2, notre conférencier préféré.
- Les conférenciers restants sont dans l'ordre $C_4 < C_3 < C_1$. C_4 étant incompatible avec C_2, on regarde C_3. Celui-ci étant compatible avec C_2, on ajoute C_3 à notre planning. Le planning devient $[C_2, C_3]$.
- On regarde les conférenciers restants : il ne reste que C_1, qui est incompatible avec C_3. L'algorithme s'arrête et propose le planning $[C_2, C_3]$.

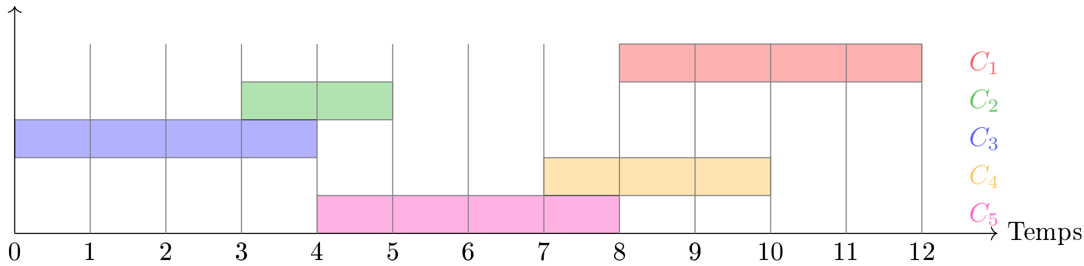
Exercice 9. Appliquer de même (à la main) les stratégies (S_2), (S_3) et (S_4) aux demandes de l'exemple précédent.

On souhaite maintenant départager ces stratégies : l'une est-elle meilleure que les autres ? On va pour cela appliquer ces stratégies sur un autre exemple.

Exercice 10. Avec le tableau des demandes ci-dessous,

- Classer les conférenciers par ordre (croissant) de préférence,
- en déduire un planning obtenu par une stratégie gloutonne,

et ce pour chacune des stratégies proposées (S_1), (S_2), (S_3) et (S_4).



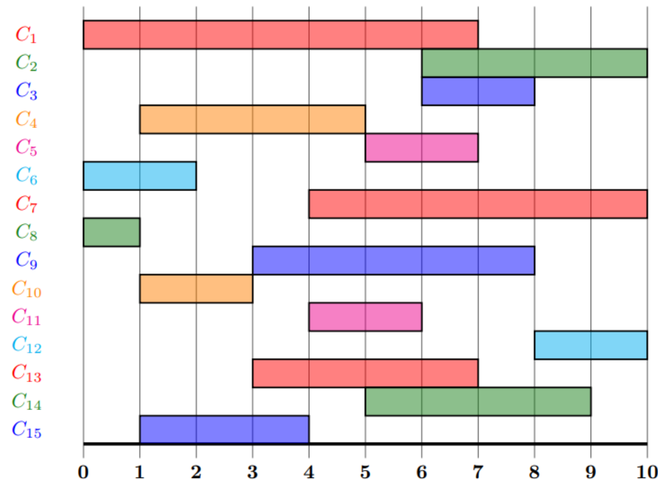
Quelle stratégie gloutonne semble la meilleure ?

2. Une stratégie gloutonne optimale

On peut démontrer que la stratégie gloutonne (S_4), consistant à classer les conférenciers par **heure de fin de conférence** (dans l'ordre croissant) puis à appliquer la méthode gloutonne, **fournit toujours un planning optimal**.

Dans la suite, on ne s'intéresse donc plus qu'à cette stratégie.

Exercice 11. Déterminer à la main un planning optimal pour les demandes suivantes.



Programmation en Python

On représentera les données ainsi en Python :

- Une demande de conférencier est donnée par une liste de longueur 3 de la forme $["C", d, f]$ où $"C"$ (type float) est le nom du conférencier, d (type int) l'heure de début du créneau demandé, et f (type int) son heure de fin. Par exemple, dans l'exercice 11, la demande du conférencier C_1 sera représentée par la liste `["C1", 0, 7]`.
- La liste des demandes à traiter est donnée par une liste de ces demandes, elles-mêmes sous forme de liste (c'est une liste de listes). Les listes des demandes des exercices 10 et 11 sont données en annexe, en guise d'exemple.
- Un planning sera donné par la liste des noms des conférenciers retenus. Par exemple, un planning formé par les conférenciers C_1 et C_2 sera représenté par la liste `["C1","C2"]`.

Exercice 12. Écrire une fonction `TriDemandes` prenant en entrée une liste de demandes, et triant cette liste par ordre croissant de préférence selon la stratégie (S_4). On adaptera pour cela le tri à bulles (voir exercice 4 du TP 6 si vous ne vous en souvenez pas !).

Tester cette fonction pour les demandes `P = [["C1",2,4],["C2",0,1],["C3",1,3],["C4",0,3]]`.

Exercice 13. Écrire une fonction `MaximiseConference` prenant en entrée une liste de demandes, et renvoyant en sortie le planning obtenu en appliquant la stratégie gloutonne (S_4) à ces demandes.

Tester cette fonction pour les demandes `P = [["C1",2,4],["C2",0,1],["C3",1,3],["C4",0,3]]`.

Le planning renvoyé devra être `["C2","C3"]`.

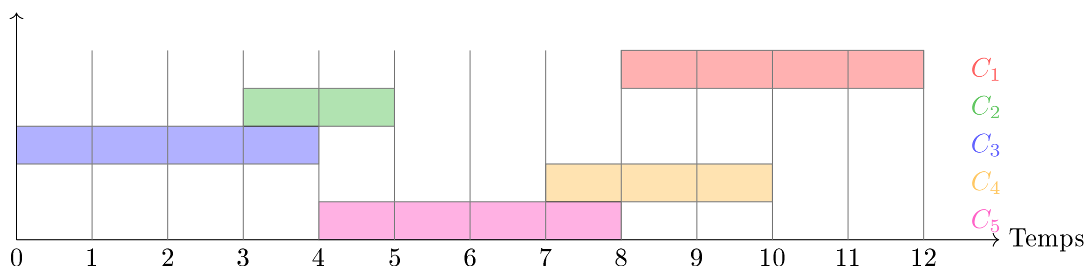
Tester cette fonction sur les exemples des exercices 10 et 11 (vous pouvez copier-coller les listes de demandes données en annexe).

III. Allocation de salles de cours

Maintenant, vous êtes à la direction d'un établissement scolaire et il est l'heure d'attribuer des salles de cours aux différents cours prévus.

Vous recevez toujours un planning de demandes de créneaux (les heures de cours d'une journée) mais ici, vous devez installer tous ces cours dans des salles de classe. Ce que vous souhaitez **optimiser**, c'est le nombre de salles nécessaires. Vous voulez utiliser le moins de salles possibles pour que les cours aient lieu.

Exemple 14. Reprenons les demandes de l'exercice 10.



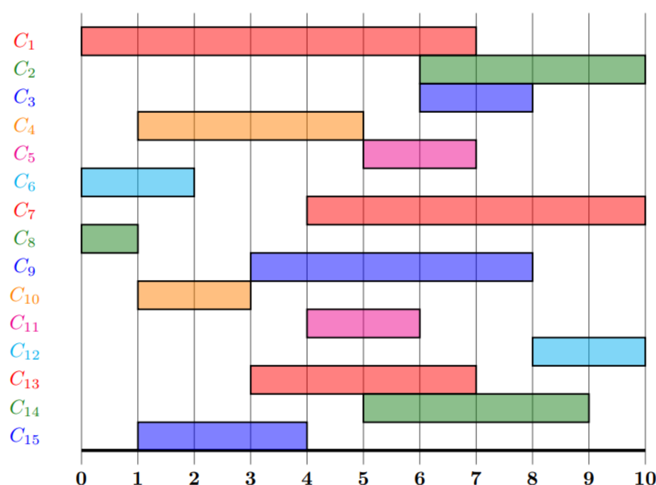
Déterminer une répartition des cours sur un nombre minimal de salles. Justifier que ce nombre est minimal.

La stratégie gloutonne optimale (S_4) de la partie précédente peut être enrichie pour fournir une stratégie gloutonne optimale à ce problème. Voici l'algorithme que cela donne :

- On ordonne les cours par heure croissante de fin de cours.
- On place le premier cours (pour cet ordre établi) dans la première salle.
- On traite les cours suivants dans cet ordre établi en première étape. Pour chaque cours :
  - Si ce cours peut être placé dans une salle déjà utilisée, on le place dans la première salle possible (dans l'ordre d'ouverture de celle-ci).
  - Sinon, on ouvre une salle et on y place ce cours.

On admet que cette méthode fournit une répartition des cours dans des salles de manière à utiliser un nombre minimal de salles.

Exercice 15. Déterminer à la main une répartition optimale pour les cours suivants (mêmes demandes que pour l'exercice 11).



Programmation en Python

En Python, la liste des cours est représentée de la même manière que dans la partie précédente : un cours nommé C_1 commençant à l'heure d et terminant à l'heure f sera représenté par la liste Python ["C1", d, f].

Les salles à ouvrir seront numérotées à partir de 1. Une attribution de cours à une salle sera donnée par une liste selon l'exemple suivant : la liste `["S1","C1","C3"]` représentera le fait de placer les cours C_1 et C_3 dans la salle S_1.

Un planning est donné par une liste de salles attribuées. Par exemple, dans le cas de l'exemple 14, le planning suivant :

- Salle 1 : C_3, C_5, C_1
- Salle 2 : C_2, C_4

sera représenté par la liste `[ ["S1", "C3", "C5", "C1"], ["S2", "C2", "C4"] ]`.

Exercice 16. Écrire le code d'une fonction `AffecteSalles` prenant en entrée une liste de cours, et renvoyant en sortie le planning obtenu en appliquant la stratégie ci-dessus.

On pourra utiliser la fonction `TriDemandes` de la partie précédente.

On rappelle que si une variable `i` vaut, par exemple, `3`, alors la commande `"S"+str(i)` renvoie "S3" (concaténation du texte `"S"` et du texte associé à l'entier `3`).

Tester cette fonction sur l'exemple de l'exercice 15 (liste des demandes en annexe).

IV. Quelques exercices

Exercice 17. Un automobiliste veut effectuer un long trajet en minimisant le nombre d'arrêts à des stations services pour faire le plein d'essence. Son véhicule a un (petit) réservoir d'une capacité de 250 km. Par ailleurs, l'automobiliste dispose de la liste `L = [120,142,90,70,130,150,84,25,110,50]` formée des distances consécutives entre deux stations-service situées le long de son trajet (et les distances au départ et à l'arrivée), stations que l'on suppose dans cet exemple numérotées de 1 à 9. Ici, la station 1 est à 120 kilomètres de son point de départ, puis 142 kilomètres séparent la station 1 de la station 2, etc. La dernière entrée (50) indique la distance de la station 9 à son point d'arrivée.

À l'aide d'une stratégie gloutonne, déterminer à la main le nombre minimal d'arrêts possibles ainsi que la liste des numéros des stations-service auxquelles s'arrêter.

Écrire le code d'une fonction Python, nommée `planifie_trajet`, prenant en entrée une liste `L` donnant ces distances consécutives entre deux stations-service le long d'un trajet (selon le format précédent) ainsi qu'une capacité de réservoir `c` strictement supérieure au minimum des valeurs contenues dans `L` et renvoyant en sortie la liste formée des numéros des stations-service auxquelles il faut s'arrêter, de sorte à minimiser le nombre d'arrêts.

Exercice 18. Le **théorème de Zeckendorf** affirme que tout entier naturel N non nul, il existe une unique suite d'entiers $c_0, \ldots, c_k$ avec $c_0 \geq 2$ et $c_{i+1} > c_i + 1$ (ie des entiers non consécutifs deux à deux) tels que

$$N = \sum_{i=0}^{k} F_{c_i}$$

où F_n est le nombre de Fibonacci d'indice n.

Cette écriture est appelée la **décomposition de Zeckendorf** de N.

On rappelle que la suite de Fibonacci est la suite $(F_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} F_0 = F_1 = 1 \\ \forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n \end{cases}.$$

1. Écrire le code d'une fonction Python `Fibonacci` prenant en entrée un entier naturel n et renvoyant en sortie le n-ième terme F_n de la suite de Fibonacci.
2. Utiliser cette fonction pour afficher à l'écran les 12 premiers termes de la suite de Fibonacci.
3. Vérifier que $17 = F_0 + F_1 + F_2 + F_4 + F_5$. Est-ce la décomposition de Zeckendorf de 17 ?
4. Déterminer la décomposition de Zeckendorf de 17, puis de 130.
5. Soit $N \in \mathbb{N}^*$. Soit k le plus grand entier tel que $F_k \leq N$.
   (a) Justifier que k est correctement défini.
   (b) Montrer que si la décomposition de Zeckendorf de $N - F_k$ s'écrit $N - F_k = \sum_{j=1}^{n} F_{x_j}$, pour un certain entier naturel $n \geq 1$ et des entiers $x_1, ..., x_n$ deux à deux distincts et non consécutifs deux à deux, alors la décomposition de Zeckendorf de N est donnée par l'égalité : $N = F_k + \sum_{j=1}^{n} F_{x_j}$.
6. En déduire le code d'une fonction `Zeckendorf` prenant en entrée un entier $N \geq 1$ et renvoyant en sortie la liste des termes de la suite de Fibonacci apparaissant dans la décomposition de Zeckendorf de N.
7. En quoi votre fonction utilise-t-elle un algorithme glouton ?

V. Annexe

Liste des demandes de l'exercice 10 :

```
P=[["C1",8,12],["C2",3,5],["C3",0,4],["C4",7,10],["C5",4,8]]
```

Liste des demandes de l'exercice 11 :

```
P=[["C1",0,7],["C2",6,10],["C3",6,8],["C4",1,5],["C5",5,7],["C6",0,2],["C7",4,10],["C8",0,1],
["C9",3,8],["C10",1,3],["C11",4,6],["C12",8,10],["C13",3,7],["C14",5,9],["C15",1,4]]
```