TP07 : Algorithmes gloutons

L'objectif de ce TP est d'étudier une méthode algorithmique permettant de résoudre des problèmes concrets soulevés par des thématiques de la vie courante.

L'approche explorée ci-dessous est itérative et propose de construire une solution au problème posé par une succession d'approximations, en imposant à chaque étape d'effectuer un choix optimal qui rapproche le plus possible de la solution cherchée, les choix locaux effectués à chaque étape n'étant jamais remis en questions ultérieurement.

Les algorithmes répondant à ce cahier des charges sont appelés **algorithmes gloutons**, ils peuvent parfois aboutir à des solutions globalement optimales des problèmes posés, mais ce n'est pas garanti en général.

Problème du rendu de monnaie

Le problème le plus classique pour lequel l'utilisation d'algorithmes gloutons est envisageable est motivé par une situation de la vie courante : la minimisation du nombre de pièces utilisées pour rendre la monnaie.

Études d'exemples concrets

Les transactions en espèces dans la zone euro sont effectuées par un échange de pièces et de billets, dont on supposera pour simplifier que les valeurs sont : 1€, 2€, 5€, 10€, 20€, 50€, 100€ et 200€ (en laissant de côté les centimes d'une part et les billets de 500€ que personne n'utilise d'autre part).

On supposera généreusement pour tous les problèmes de rendu de monnaie ci-dessous que l'on dispose toujours d'un stock illimité de chaque pièce ou billet.

Exercice 1 Dans un magasin, un client règle un achat d'un montant de 41€ avec un billet de 50€.

(a) Dresser la liste complète de toutes les manières dont le vendeur peut lui rendre la monnaie.

(b) Parmi toutes ces manières, laquelle utilise le moins de pièces possibles ?

(c) Quelle stratégie le vendeur peut-il suivre en rendant la monnaie pour aboutir à cette solution optimale ?

En pratique, dans les échanges de la vie courante, tout le monde ou presque adopte la stratégie permettant de rendre la monnaie en minimisant le nombre de pièces utilisées.

On peut décrire cette stratégie de la façon suivante :

- Déterminer la plus grande valeur v de pièce ou billet inférieure ou égale à la somme à rendre.
  - Tant que la somme à rendre reste plus grande que v, rendre une pièce ou un billet de valeur v et soustraire la valeur v à la somme à rendre.
- Recommencer ainsi de suite jusqu'à ce que la somme à rendre soit nulle.

On qualifie la stratégie présentée ci-dessus de **gloutonne**.

On peut se demander si le fait d'adopter cette stratégie permet toujours d'aboutir à un rendu de monnaie minimisant le nombre de pièces utilisées.

Exercice 2 Imaginons que dans la zone euro d'un univers parallèle, les valeurs 2€ et 5€ sont remplacées respectivement par 3€ et 4€, les autres valeurs étant inchangées.

Dans cet univers parallèle, un client règle un achat d'un montant de 14€ avec un billet de 20€.

(a) Dresser la liste complète de toutes les manières dont le vendeur peut lui rendre la monnaie.

(b) Parmi toutes ces manières, laquelle utilise le moins de pièces possibles ?

(c) Parmi toutes ces manières, laquelle utilise la stratégie gloutonne ?

Un système monétaire peut donc tout à fait ne pas être adaptée à la stratégie gloutonne pour le problème de l'optimisation du rendu de monnaie.

On dit qu'un système monétaire est **canonique** si pour toute somme à rendre, la stratégie gloutonne fournit la solution optimale permettant de minimiser le nombre de pièces ou billets rendus.

Il est possible de prouver que le système monétaire dont les valeurs sont {1€,2€,5€,10€,20€,50€,100€,200€} est canonique, ce qui justifie a posteriori le caractère raisonnable de la stratégie gloutonne que tout le monde ou presque adopte pour rendre la monnaie en zone euro.

En revanche, le système monétaire dont les valeurs sont {1€,3€,4€} n'est pas canonique, en vertu de l'exemple étudié dans l'exercice 2 où la stratégie gloutonne ne donne pas un résultat optimal.

 CC BY-NC

Formalisation du problème

Pour modéliser de façon générale le problème du rendu de monnaie, on peut introduire le système monétaire sous forme d'un n-uplet :

$$V = (v_1, \ldots, v_n)$$

tel que $v_1 > v_2 > \cdots > v_n$ où les valeurs v_i désignent les montants des pièces et billets disponibles.

Par exemple, pour le système en euros décrit ci-dessus, $V = (200, 100, 50, 20, 10, 5, 2, 1)$.

Pour garantir l'existence de solutions au problème de rendu de monnaie, on supposera que $V \in (\mathbb{N}^*)^n$, que $v_n = 1$ et que toutes les sommes à rendre ont des valeurs entières non-nulles. Dans ce cas, il est toujours possible de rendre la somme souhaitée, par exemple en utilisant exclusivement des pièces de 1€.

Soit alors $S \in \mathbb{N}^*$ une somme à rendre. Un rendu de monnaie associé à S est la donnée d'un n-uplet $X = (x_1, \ldots, x_n) \in (\mathbb{N})^n$ tel que $S = \displaystyle\sum_{i=1}^{n} x_i v_i$. Chaque valeur x_i du n-uplet s'interprète comme le nombre de fois que la pièce ou le billet de valeur v_i est utilisé pour le rendu de la somme S.

Dans ce cas, le nombre de pièces ou billets utilisés pour le rendu de la somme S est égal à $\displaystyle\sum_{i=1}^{n} x_i$.

Ainsi, le problème de minimisation du nombre de pièces ou billets pour le rendu de la somme S est équivalent à la détermination, parmi tous les n-uplets $X = (x_1, \ldots, x_n) \in (\mathbb{N})^n$ tels que $\displaystyle\sum_{i=1}^{n} x_i v_i = S$, de celui pour lequel la somme $\displaystyle\sum_{i=1}^{n} x_i$ est la plus petite possible.

En général, on dit d'une telle situation qu'il s'agit d'un problème d'optimisation sous contrainte : il s'agit de **minimiser** la valeur de $\displaystyle\sum_{i=1}^{n} x_i$ tout en satisfaisant la **contrainte** $\displaystyle\sum_{i=1}^{n} x_i v_i = S$.

Exercice 3 Soit $V = (v_1, \ldots, v_n)$ un système monétaire canonique et $S \in \mathbb{N}^*$ une somme à rendre.

Écrire le principe d'un algorithme de recherche de la solution minimisant le nombre de pièces ou billets pour le rendu de la somme S, en utilisant la stratégie gloutonne (ce principe devra être indépendant de tout langage de programmation). Cet algorithme devra renvoyer l'unique n-uplet $X = (x_1, \ldots, x_n)$ tel que la somme $\displaystyle\sum_{i=1}^{n} x_i$ soit minimale sous la contrainte $\displaystyle\sum_{i=1}^{n} x_i v_i = S$.

Programmation en Python

On choisira d'utiliser la structure de liste pour implémenter en Python la notion de système monétaire.

Une somme à rendre sera représentée par une variable de type `int` et une solution au problème de rendu de monnaie sera également représentée sous forme de liste, ordonnée de manière cohérente avec la liste définissant le système monétaire.

Le système $V = (200, 100, 50, 20, 10, 5, 2, 1)$ sera représenté par la liste `[200,100,50,20,10,5,2,1]` et pour la somme `S = 34`, la solution optimale du problème de rendu de monnaie obtenue grâce à la stratégie gloutonne est donnée par la liste `[0,0,0,1,1,0,2,0]` qui indique que l'on rend un billet de 20€, un billet de 10€ et deux pièces de 2€.

Exercice 4 On considère le système monétaire représenté par la liste `V = [200,100,50,20,10,5,2,1]`.

On suppose que la somme à rendre est `S = 98`.

Écrire un code Python affichant sous forme de liste l'unique 8-uplet $X = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ solution du problème d'optimisation du rendu de monnaie pour la somme S dans le système V (bien entendu, on ne déterminera pas ce 8-uplet à la main).

Exercice 5 Écrire le code d'une fonction Python, nommée `rendu_glouton`, prenant en paramètres une liste `V` représentant un système monétaire et une somme à rendre `S` et renvoyant une liste `X` indiquant les nombres de pièces et billets à rendre en suivant la stratégie gloutonne.

Tester cette fonction pour les arguments :

(a) `V = [200,100,50,20,10,5,2,1]` et `S = 49`

(b) `V = [30,24,12,6,3,1]` et `S = 49` (ce système monétaire était celui du Royaume-Uni jusqu'en 1971).

CC BY-NC

Problème de réservation d'une salle

Un autre problème d'organisation venant de situations de la vie courante peut amener à développer une approche de résolution gloutonne : celui de la réservation de salle, dans le but de maximiser le nombre d'activités.

Nous considérerons la situation concrète d'un centre de conférence, disposant d'un amphithéâtre que plusieurs conférenciers souhaitent réserver sur des créneaux horaires qui ne sont pas forcément compatibles.

Le secrétariat du centre de conférence doit alors choisir une stratégie permettant de **maximiser le nombre de conférenciers** qui pourront intervenir (ce choix permettant d'optimiser l'attractivité de la conférence sur la base du prestige des conférenciers).

Études d'exemples concrets

Commençons par considérer la situation où les réservations se font pour une matinée dont l'heure de début est numérotée 0 et l'heure de fin est numérotée 4.

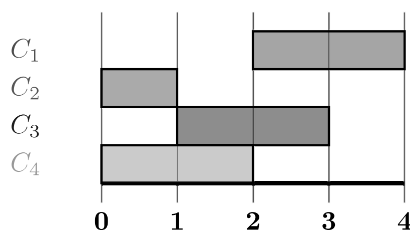
Les créneaux horaires disponibles sont donc : $[0,1[$, $[1,2[$, $[2,3[$ et $[3,4[$.

On peut placer deux conférenciers consécutivement sur de tels créneaux (les intervalles sont ouverts à droite, donc les créneaux sont deux à deux disjoints) et un conférencier peut demander à intervenir pendant plusieurs heures, à condition qu'elles soient consécutives.

Considérons tout d'abord la situation où quatre conférenciers C_1, C_2, C_3 et C_4 souhaitent intervenir avec les créneaux horaires suivants :

- C_1 : $[2,4[$
- C_2 : $[0,1[$
- C_3 : $[1,3[$
- C_4 : $[0,2[$

On peut représenter cette liste de demandes par la figure ci-dessous :



En tenant compte des incompatibilités de créneaux horaires, on peut proposer la liste complète des plannings possibles en donnant la liste des conférenciers retenus classés par ordre de passage :

- $[C_2, C_3]$
- $[C_2, C_1]$
- $[C_4, C_1]$
- $[C_3]$

Dans le but de maximiser le nombre de conférenciers, on pourra retenir n'importe lequel des trois premiers plannings, mais il conviendra d'écarter le quatrième.

Afin de définir une stratégie permettant de construire, en toute généralité, un planning permettant de maximiser le nombre de conférenciers, il faut choisir un critère permettant de sélectionner, un par un, les conférenciers les plus intéressants, en fonction de leurs créneaux horaires.

Plusieurs critères de choix naturels peuvent être proposés :

- Stratégie S_1 : classer les conférenciers par ordre croissant d'heure de début de conférence
- Stratégie S_2 : classer les conférenciers par ordre croissant de durée de conférence
- Stratégie S_3 : classer les conférenciers par ordre croissant d'incompatibilités avec les autres conférenciers
- Stratégie S_4 : classer les conférenciers par ordre croissant d'heure de fin de conférence

Dans le cas de l'exemple ci-dessus, on peut appliquer chacune de ces quatre stratégies pour ordonner les conférenciers par ordre croissant (mais pas forcément strictement croissant) :

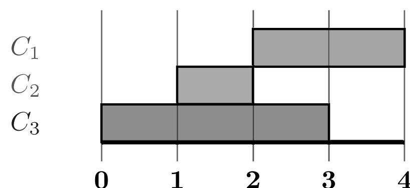
- Stratégie S_1 : $C_4 = C_2 < C_3 < C_1$, ce qui nous donne par exemple le planning : $[C_4, C_1]$
- Stratégie S_2 : $C_2 < C_4 = C_3 = C_1$, ce qui nous donne par exemple le planning : $[C_2, C_3]$
- Stratégie S_3 : $C_2 = C_1 < C_4 = C_3$, ce qui nous donne par exemple le planning : $[C_2, C_1]$
- Stratégie S_4 : $C_2 < C_4 < C_3 < C_1$, ce qui nous donne le planning : $[C_2, C_3]$

CC BY-NC

Chaque stratégie peut donner un planning différent et on retrouve en particulier les trois solutions optimales de notre premier exemple. Rien ne suggère donc à ce stade de privilégier une stratégie plutôt qu'une autre.

Afin de les départager, étudions d'autres exemples à travers les quelques exercices qui suivent.

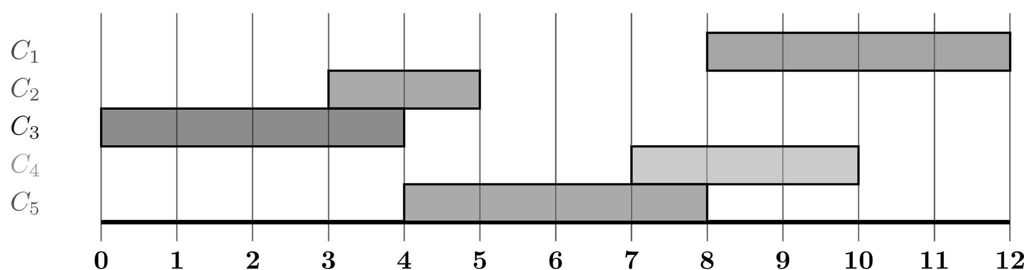
Exercice 6 On considère le planning des demandes ci-dessous :



(a) Appliquer chacune des quatre stratégies pour ordonner les conférenciers.

(b) Déterminer un planning possible pour chacune des quatre stratégies. Lesquels sont optimaux ?

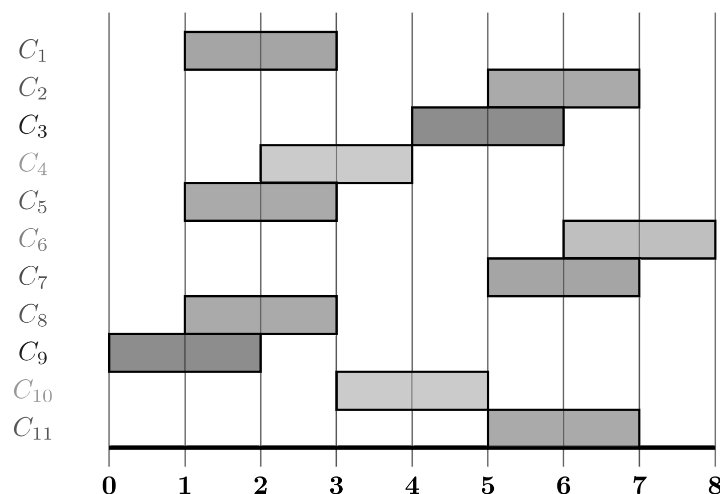
Exercice 7 On considère le planning des demandes ci-dessous :



(a) Appliquer chacune des quatre stratégies pour ordonner les conférenciers.

(b) Déterminer un planning possible pour chacune des quatre stratégies. Lesquels sont optimaux ?

Exercice 8 On considère le planning des demandes ci-dessous :



(a) Appliquer chacune des quatre stratégies pour ordonner les conférenciers.

(b) Déterminer un planning possible pour chacune des quatre stratégies. Lesquels sont optimaux ?

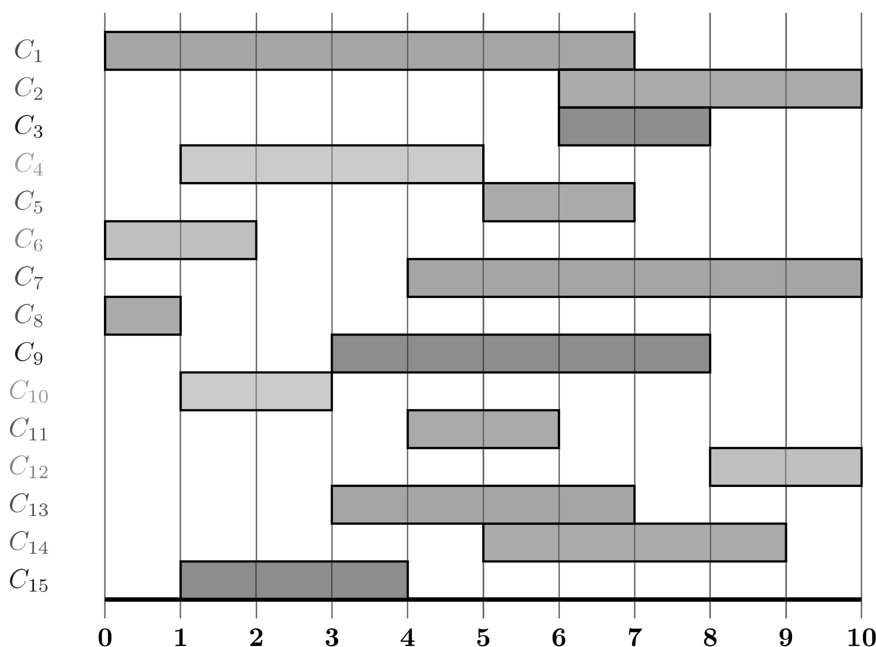
On peut retenir de l'étude de ces exemples qu'il semble plus pertinent d'ordonner les conférenciers par ordre croissant d'heure de fin de conférence pour construire le planning : en effet, dans tous les exemples ci-dessus, il s'agit de la seule stratégie qui donne dans tous les cas une organisation optimale.

Une telle stratégie est **gloutonne** au sens où à chaque étape, le choix d'un conférencier est fait pour **maximiser le temps restant**. Il est possible de démontrer que cette stratégie S_4 donne toujours une solution optimale au problème posé, contrairement aux stratégies S_1, S_2 et S_3.

Par la suite, on retiendra donc uniquement la stratégie S_4 consistant à classer les conférenciers par ordre croissant d'heure de fin de conférence pour construire des planning d'occupation de salle.

BY-NC

Exercice 9 On considère le planning des demandes ci-dessous :



Déterminer à la main un planning optimal d'occupation de la salle.

Programmation en Python

On choisira d'utiliser la structure de liste de listes pour implémenter en Python un planning de demandes de réservations. Une demande individuelle sera représentée par une variable de type `list` de longueur 3 dont les éléments seront, dans l'ordre, une chaîne de caractères identifiant le conférencier, puis l'heure de début et l'heure de fin données sous forme de nombres entiers naturels.

Par exemple, les quatre demandes des conférenciers du premier exemple étudié sur la page 3 sont représentées par les listes : `["C1",2,4]`, `["C2",0,1]`, `["C3",1,3]` et `["C4",0,2]`. Ainsi, le planning complet des demandes est représenté par la liste de listes : `[["C1",2,4],["C2",0,1],["C3",1,3],["C4",0,2]]`.

On rappelle que l'on peut trier une liste numérique par ordre croissant en effectuant suffisamment de permutations de termes consécutifs lorsqu'ils sont rangés dans le mauvais ordre. La fonction ci-dessous prend en argument une liste numérique `L` et renvoie en sortie une liste trié par ordre croissant contenant exactement les mêmes valeurs que `L` (voir TP04 : Introduction aux listes).

```
def tri_liste(L):
    for n in range(0,len(L)-1):
        for i in range(0,len(L)-1):
            if L[i]>L[i+1]:
                L[i],L[i+1] = L[i+1],L[i]
    return L
```

Exercice 10 En s'inspirant du code de la fonction `tri_liste` ci-dessus, écrire le code d'une fonction Python, nommée `tri_planning`, prenant en paramètre un planning `P` sous forme de liste de listes comme décrit ci-dessus, et renvoyant en sortie un planning équivalent dans lequel les conférenciers sont triés par ordre croissant des heures de fin de conférence.

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

Exercice 11 Écrire le code d'une fonction Python, nommée `maximise_conferences`, prenant en paramètre un planning `P` sous forme de liste de listes comme décrit ci-dessus, et renvoyant en sortie une liste contenant, dans l'ordre de passage, les conférenciers retenus par la méthode gloutonne de la stratégie S_4.

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

Le résultat renvoyé devra être la liste `["C2","C3"]`.

Tester ensuite cette fonction pour chacun des plannings de demandes étudiés dans les exercices 6 à 9 (on peut trouver les plannings correspondant dans l'annexe page 7 que l'on pourra copier-coller directement depuis le fichier pdf en ligne).

CC BY-NC

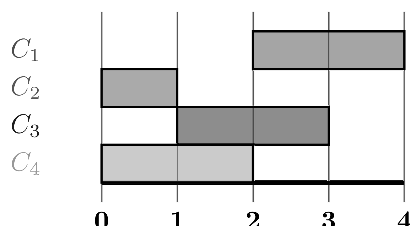
Problème d'allocation de salles de cours

Dans cette partie, on considère la situation concrète d'un établissement scolaire disposant d'un nombre limité de salles de cours. Dans cet établissement, de nombreux cours doivent se dérouler lors d'une même journée et il faut attribuer une salle pour chaque cours.

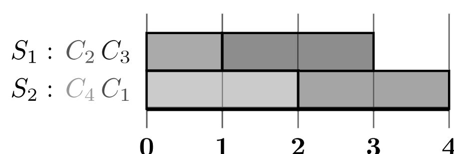
Le proviseur adjoint reçoit un planning de demandes, sous une forme similaire à celle étudiée pour le problème de réservation de salles dans la partie précédente. Son rôle consiste alors à affecter chaque demande à une salle de cours de manière à établir un emploi du temps sans incompatibilité, tout en **minimisant le nombre de salles** utilisées (ce choix permet d'optimiser la consommation électrique, les besoins en équipement matériel, le travail quotidien des agents d'entretien, la gestion de la sécurité, etc).

Études d'exemples concrets

On conserve les notations de la partie précédente. Considérons le planning de demandes de cours suivant :



Dans cette situation, on peut proposer de répartir les cours sur deux salles, en affectant les cours C_2 et C_3 dans une première salle S_1, puis C_4 et C_1 dans une seconde salle S_2. Il n'est pas possible de faire mieux, puisque certains cours sont incompatibles. On peut donc proposer de renvoyer le planning définitif ci-dessous :

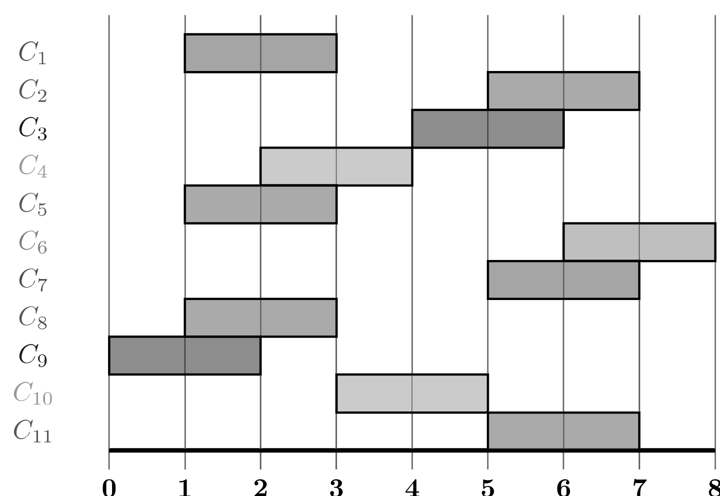


Pour construire un planning optimal, on peut chercher à maximiser le nombre d'utilisateurs par salle avec la même stratégie que précédemment et décider de n'ouvrir une nouvelle salle qu'en cas d'incompatibilité.

La démarche algorithmique gloutonne décrite ci-dessous permet de minimiser le nombre de salles de cours :

- Ordonner les cours par heure croissante de fin
- Placer le premier cours dans une première salle
- Pour chacun des cours suivants, tester si ce cours peut être placé dans l'une des salles déjà utilisées :
  - si oui le mettre dans la première salle disponible dans le planning
  - si non, affecter ce cours à une nouvelle salle dans le planning

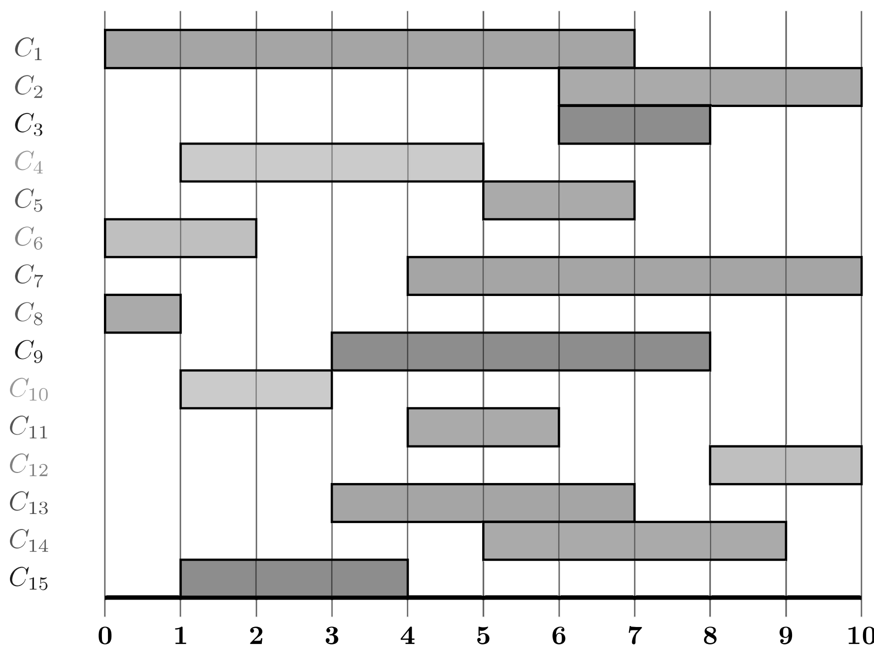
Exercice 12 On considère le planning des demandes ci-dessous :



Déterminer à la main un planning de cours utilisant un nombre minimal de salles de cours.

BY-NC

Exercice 13 On considère le planning des demandes ci-dessous :



Déterminer à la main un planning de cours utilisant un nombre minimal de salles de cours.

Programmation en Python

Un planning de demandes de cours est représenté exactement comme l'étaient les plannings de demandes de conférences dans la partie précédente. Un planning définitif d'affectation de salles fourni par le proviseur adjoint sera constitué d'une liste de listes. Chaque liste individuelle correspondra à une salle de cours et sera constitué d'un premier élément qui sera un identifiant de la salle sous forme de chaîne de caractère, par exemple `"S1"`, suivi des identifiants des cours qui auront été affectés à cette salle, dans l'ordre croissant de leurs horaires.

Par exemple, le planning définitif d'affectation de salles obtenu dans l'étude du premier exemple en page 6 est représenté par la liste de listes `[["S1","C2","C3"],["S2","C4","C1"]]`.

Exercice 14 Écrire le code d'une fonction Python, nommée `affecte_salles`, prenant en paramètre un planning de demandes de cours `P`, et renvoyant en sortie une liste de listes représentant le planning d'affectation des salles.

Indication : si `i` vaut `3`, alors la commande `"S"+str(i)` renvoie la chaîne de caractères `"S3"`.

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

Le résultat renvoyé devra être la liste de listes `[["S1","C2","C3"],["S2","C4","C1"]]`.

Tester ensuite cette fonction pour chacun des plannings de demandes étudiés dans les exercices 6 à 9.

Annexe : plannings des demandes des exercices 6 à 9

On trouve ci-dessous la représentation informatique des plannings des exercices 6 à 9, pouvant être copiés-collés directement dans Pyzo pour éviter d'avoir à les saisir à la main (le concepteur de ce sujet de TP les a généreusement saisis à la main !).

Exercice 6

```
P=[["C1",2,4],["C2",1,2],["C3",0,3]]
```

Exercice 7

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

Exercice 8

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

Exercice 9

```
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]]
```


CC BY-NC

Exercices

Exercice 15 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]` formée des distances consécutives entre deux stations-service situées le long de son trajet, stations que l'on suppose dans cet exemple numérotées de 1 à 9.

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 paramètres une liste `L` donnant les distances consécutives entre deux stations-service le long d'un trajet 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 16 Un employé d'une entreprise est responsable de la réalisation de plusieurs **tâches**, chacune demandant un certain **temps de travail** et étant assortie d'une **date limite** à laquelle il est impératif qu'elle soit réalisée (en anglais, on dirait une deadline). On considérera un référentiel de temps qui commence à $t = 0$.

L'objectif de l'employé est de trouver un ordre dans lequel effectuer les tâches de manière à ne jamais dépasser de date limite, mais dans la vie réelle ceci n'est pas toujours possible et il est alors souhaitable de chercher une solution qui minimise le retard maximal pouvant être constaté à la fin de l'exécution d'une tâche (en considérant qu'il est préférable d'avoir 5 fois une heure de retard sur 5 tâches différentes plutôt que d'avoir une seule fois un retard de 4 heures).

Considérons par exemple l'ensemble de tâches dont les caractéristiques sont définies ci-dessous :

Numéro de tâche	1	2	3	4	5	6
Temps de travail	3	2	1	4	3	2
Date limite	6	8	9	9	14	15

Proposer une stratégie gloutonne permettant de choisir un ordre dans lequel réaliser les tâches de manière à minimiser le retard maximal et mettre en œuvre cette stratégie sur l'exemple ci-dessus.

En supposant que chaque tâche est représentée informatique par une liste de type `["T1",3,6]` et que l'ensemble des tâches est représenté par une liste de listes, écrire le code d'une fonction Python, nommée `ordre_taches`, prenant en paramètre un ensemble de tâches et renvoyant en sortie une liste contenant une permutation des numéros des tâches permettant de minimiser le retard maximal.

Exercice 17 Le **théorème de Zeckendorf** affirme que tout nombre entier naturel non-nul peut s'écrire, de façon unique, sous forme d'une somme de termes **distincts** et **non-consécutifs** de la suite de Fibonacci.

On rappelle que la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ est définie par $\begin{cases} F_0 = 1 \\ F_1 = 1 \\ \forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n \end{cases}$

En particulier, les premiers termes de cette suite sont donnés par : 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.

Par exemple, on a $17 = 1 + 3 + 13 = F_1 + F_3 + F_6$. Ceci est la **décomposition de Zeckendorf** de 17.

On pourrait aussi écrire que $17 = 1 + 1 + 2 + 5 + 8 = F_0 + F_1 + F_2 + F_4 + F_5$ qui est bien une somme de termes distincts de la suite de Fibonacci, mais certains de ces termes sont consécutifs, donc cela ne satisfait pas la contrainte d'une décomposition de Zeckendorf.

(*a*) Déterminer la décomposition de Zeckendorf de 130.

(*b*) Soit $N \in \mathbb{N}^*$. Soit $k \in \mathbb{N}$ tel que F_k soit le plus grand terme de la suite de Fibonacci inférieur ou égal à N. Montrer que si $\sum_{i=1}^{n} F_{x_i}$ est la décomposition de Zeckendorf de $N - F_k$, pour un certain n-uplet $(x_1, \dots, x_n)$, alors $\sum_{i=1}^{n} F_{x_i} + F_k$ est la décomposition de Zeckendorf de N. En déduire une stratégie gloutonne permettant de déterminer la décomposition de Zeckendorf de N.

(*c*) Écrire le code d'une fonction Python, nommée `Fibonacci`, prenant en argument un nombre entier naturel non-nul `N` et renvoyant en sortie la liste de tous les termes de la suite de Fibonacci inférieurs ou égaux à `N`.

(*d*) Écrire le code d'une fonction Python, nommée `Zeckendorf`, prenant en paramètre un nombre entier naturel non-nul `N` et renvoyant en sortie la liste des termes de la suite de Fibonacci apparaissant dans la décomposition de Zeckendorf de `N`.

cc BY-NC