TP 5 : Algorithmes gloutons

1 Allocation de salles pour des cours

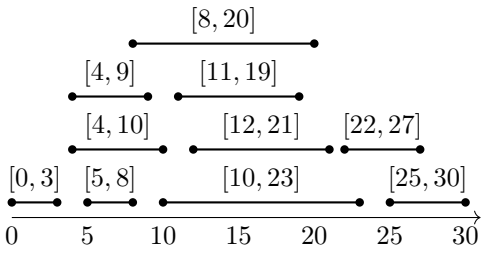
Dans le cours, on a étudié le problème de la maximisation d'activités : on se donne un ensemble d'activités qui doivent chacune se dérouler pendant un intervalle de temps $[g, d[$, et on veut pouvoir en choisir le plus possible qui se déroulent pendant des intervalles de temps disjoints.

On a vu que pour répondre à cette question, il suffisait de trier les intervalles par date de fin croissante, et d'appliquer l'algorithme glouton suivant :

Algorithme 1 : Algorithme glouton pour la sélection d'activités

Entrée : Un ensemble $\mathcal{I}$ d'intervalles triés par date de fin croissante
Sortie : Une partie de $\mathcal{I}$ constituée d'intervalles disjoints, de cardinal maximal
$\mathcal{S} \leftarrow \emptyset$;
pour *tout intervalle* I *de* $\mathcal{I}$ **faire**
  si I *n'intersecte aucun des éléments de* $\mathcal{S}$ **alors**
    Rajouter I à $\mathcal{S}$
Renvoyer $\mathcal{S}$

Le problème considéré dans cette section est semblable : les intervalles représentent des cours qui doivent avoir lieu au lycée, chaque cours a lieu dans une salle, et évidemment deux cours dont les intervalles s'intersectent ne peuvent se dérouler dans la même salle. La figure ci-dessous montre un exemple de différents cours pouvant se répartir sur 4 salles (et pas moins, puisqu'au temps 14 par exemple, il y a 4 cours en même temps !)



Combien de salles sont nécessaires au minimum, et quels cours attribuer à chaque salle ? Nous allons voir que l'algorithme suivant répond au problème, et l'implémenter.

Algorithme 2 : Algorithme glouton pour l'allocation de salles

Entrée : Un ensemble $\mathcal{I}$ d'intervalles triés par date de fin croissante, représentant des cours
Sortie : Un ensemble minimal de salles pour accueillir les cours
$\mathcal{S} \leftarrow \emptyset$; `#ensemble vide de salles`
pour *tout intervalle* $I = [g, d[$ *de* $\mathcal{I}$ **faire**
  si *il existe une salle de* S *inoccupée au moment* g **alors**
    L'ajouter à cette salle
  sinon
    Créer une autre salle contenant uniquement I
Renvoyer $\mathcal{S}$

Il est clair que l'algorithme affecte les cours dans des salles de sorte que deux cours n'ont jamais lieu en même temps dans la même salle.

Question 1. Montrer par récurrence sur le nombre d'intervalles de $\mathcal{I}$ que l'algorithme précédent renvoie bien un ensemble de salles de cardinal minimal.

Maintenant, on implémente l'algorithme. Pour faciliter l'implémentation, outre l'ensemble de salles en construction, il est commode de stocker également pour chaque salle le prochain instant où celle-ci sera libre pour accueillir de nouveaux cours (l'instant où termine le dernier cours affecté à la salle).

Question 2. Écrire une fonction `est_libre(T,g)` prenant en entrée une liste de nombres `T` (dans l'idée, les instants où les salles sont libres), et un nombre g (dans l'idée, l'heure où commence le prochain cours à affecter), et qui renvoie l'indice i d'une salle pouvant accueillir le cours s'il en existe au moins une, et -1 sinon.

```
>>> est_libre([],0)
-1
>>> est_libre([1, 5],4)
0
>>> est_libre([19, 9, 10],8)
-1
>>> est_libre([19, 9, 10],12)
1
```

Question 3. En déduire une implémentation `minimisation(I)` de l'algorithme, prenant en entrée une liste de couples (les intervalles, supposés déja triés dans l'ordre croissant de leur deuxième composante) et renvoyant une liste de listes de couples, chaque liste représentant une salle contenant les cours l'occupant.

```
>>> L
[(0, 3), (5, 8), (4, 9), (4, 10), (11, 19), (8, 20), (12, 21), (10, 23), (22, 27), (25, 30)]
>>> minimisation(L)
[[(0, 3), (5, 8), (11, 19), (22, 27)], [(4, 9), (12, 21), (25, 30)], [(4, 10), (10, 23)], [(8, 20)]]
```

Pour l'implémentation, on utilisera une liste `S` initialement vide pour les salles (qui contient donc des listes de couples), et une autre liste `T` pour les instants auxquels ces salles sont libres. Pour chaque cours $[g, d[$ de I (représenté par le couple `(g,d)`) :

— si une salle peut l'accueillir, rajouter l'intervalle à la salle, et modifier l'élément de `T` correspondant en d ;

— sinon, rajouter à `S` la liste `[(g,d)]` et à `T` la valeur d.

Question 4. Quelle est la complexité de l'algorithme ?

2 Rendu de monnaie : introduction à la programmation dynamique

On rappelle le problème du rendu de monnaie : soit $(p_0, \dots, p_{n-1})$ une séquence croissante d'entiers (les valeurs des pièces, avec $p_0 = 1$), et une somme $S \in \mathbb{N}$ à rendre.

$$\text{Problème : trouver } (\alpha_0, \dots, \alpha_{n-1}) \in \mathbb{N}^n \text{ tel que } S = \sum_{i=0}^{n-1} \alpha_i p_i \text{ et } \sum_{i=0}^{n-1} \alpha_i \text{ minmal.}$$

Question 5. *Rendre la monnaie gloutonnement*. On a vu en cours que l'algorithme glouton consistant à rendre au maximum la pièce p_{n-1}, puis la pièce p_{n-2}, etc... était optimal pour certains problèmes de pièces. Écrire l'algorithme `rendu_glouton(P,S)` correspondant, où `P` est la liste contenant $p_0, \dots, p_{n-1}$. Le résultat sera une liste de taille n contenant les α_i.

```
>>> rendu_glouton([1, 2, 5, 10, 20, 50], 666)
[1, 0, 1, 1, 0, 13]
>>> rendu_glouton([1, 3, 4], 6)
[2, 0, 1]
```

Quelle est la complexité de `rendu_glouton(P,S)` ?

Comme on le voit sur ce dernier exemple, l'algorithme glouton ne fournit pas une réponse optimale dans ce cas : le triplet $(0, 2, 0)$ convient et permet d'utiliser moins de pièces. Une méthode plus coûteuse, mais qui donne toujours une réponse optimale, est la suivante.

On définit $(N_{k,s})_{0 \leq k < n, 0 \leq s \leq S}$ comme le nombre de pièces nécessaires pour obtenir la somme s, avec seulement les $(k+1)$ premières valeurs de pièces $p_0, \dots, p_k$. Puisque $p_0 = 1$, il est toujours possible de construire la somme s, et on aura toujours $N_{k,s} \leq s$.

Question 6. Que vaut $N_{0,s}$ pour tout s ?

La méthode consiste à calculer d'abord *tous* les $N_{k,s}$ pour en déduire $N_{n,S}$, qui est la quantité minimale de pièces, solution de notre problème. Dans un second temps, on calcule les α_i.

Question 7. 1. Justifier que $N_{k,s} = \begin{cases} N_{k-1,s} & \text{si } p_k > s \\ \min(N_{k-1,s}, 1 + N_{k,s-p_k}) & \text{si } p_k \leq s \end{cases}$
(On pourra dans un premier temps comprendre d'où vient cette relation, et réfléchir à comment la prouver proprement plus tard).

2. Ainsi, comme on connaît les $N_{0,s}$, on peut facilement calculer les autres $N_{k,s}$, en procédant par k croissant puis par s croissant à k fixé. En déduire une fonction `calcule_N(P,S)` renvoyant une matrice (liste de listes) `N`, telle que `N[k][s]` contienne $N_{k,s}$ pour tous $0 \leq k < n$ et $0 \leq s \leq S$. Pour créer une liste de n listes de taille $(S+1)$ remplies de zéros, utiliser `[[0]*(S+1) for i in range(n)]`.

```
>>> calcule_N([1, 3, 4], 6)
[[0, 1, 2, 3, 4, 5, 6], [0, 1, 2, 1, 2, 3, 2], [0, 1, 2, 1, 1, 2, 2]]
```

Quelle est la complexité de `calcule_N(P,S)` ?

À partir de la matrice précédente, on sait le nombre de pièces cherché : `N[n-1][S]`. Dans l'exemple précédent, on retrouve bien qu'il suffit de 2 pièces avec la somme $S = 6$ et le système $P = (1, 3, 4)$. Il reste à savoir comment retrouver les bonnes pièces à utiliser (dans notre exemple, deux pièces de valeurs 3). Pour cela, on part du coefficient $(n-1, S)$ dans la matrice. Deux cas se présentent :

— Si $N_{n-1,S} = N_{n-2,S}$ (c'est le cas ici), on n'utilise pas du tout la pièce p_n : on se déplace en case $(n-2, S)$;
— Sinon, c'est que $N_{n-1,S} = 1 + N_{n-1,S-p_n}$, et on utilise au moins une fois la pièce p_n. On ajoute notre pièce p_n et on se déplace en case $(n-1, S-p_n)$.

Le même processus se répète en chaque case (k, s), jusqu'à aboutir à une case où $k = 0$ (on sait qu'il faut utiliser alors s pièces de valeur $p_0 = 1$).

Question 8. Écrire une fonction `rendu_optimal(P,S)` renvoyant une liste de même longueur que `P`, indiquant comment décomposer optimalement la somme `S` avec les pièces de `P`.

```
>>> rendu_optimal([1, 3, 4], 6)
[0, 2, 0]
>>> rendu_optimal([1, 2, 5, 10, 20, 50], 666) #le glouton est optimal sur cet ensemble de pièces
[1, 0, 1, 1, 0, 13]
```

Question 9. Quelle est la complexité de `rendu_optimal(P,S)`, en fonction de S et de n, la taille de `P` ?

3 Pour aller plus loin

Cette section présente quelques problèmes d'optimisation où les techniques du TP s'appliquent.

— Voyageur de commerce : on se donne n points du plan, on cherche une tournée (chemin démarrant du premier point, passant une fois et une seule par tous les autres, et revenant au point de départ) de longueur la plus courte possible. Une approche gloutonne consiste à partir du point d'indice 0, et choisir à chaque fois comme prochain point à visiter le plus proche *non déja exploré* (sans oublier de revenir au point de départ à la fin). Implémenter l'approche, en supposant que le nuage de points est fourni par une liste de couples. Montrer par un exemple que cette approche ne fournit pas une solution optimale en général.
— Somme de sous-ensembles : on se donne une liste d'entiers `L`, et un entier cible C. On cherche à savoir si on peut écrire C comme somme d'éléments de `L`. Plutôt qu'une technique gloutonne, chercher une technique par programmation dynamique comme en section 2 : on construira une matrice de booléens $(b_{k,c})$, avec $b_{k,c}$ indiquant si l'entier c de $[\![0, C]\!]$ peut s'écrire comme somme d'éléments de `L[:k]`.