DIU « Enseigner l'Informatique au Lycee »

Algorithmiques gloutons

Un algorithme glouton fait toujours le choix qui semble le meilleur sur le moment. Il fait un choix localement optimal dans l'espoir que ce choix mènera à la solution globalement optimale. Ils ne conduisent donc pas toujours à la solution optimale.

Problème du rendu de pièces de monnaie

Ce problème est un grand classique de l'algorithmique. Lorsque vous passez à la caisse d'un magasin quelconque, il n'est pas rare que le caissier doive vous rendre de l'argent car le montant que vous lui avez donné est supérieur à celui que vous devez payer.

Exemple. Supposons qu'on doive vous rendre la somme de 2,63€. Il y a plusieurs façons de procéder. On peut par exemple vous rendre 263 pièces de 1 centime, 125 pièces de 2 centimes et 13 pièces de 1 centime ou encore 5 pièces de 50 centimes, une de 10 centimes, une de 2 centimes et une de 1 centime.

Il y a bien évidemment énormément de possibilités pour vous rendre la dite somme. Mais il y a fort à parier que les solutions du type « 263 pièces de 1 cent » ne vous conviennent pas... Le problème qui se pose est donc de minimiser le nombre de pièces rendues pour un montant fixé.

Une méthode à laquelle on pense immédiatement est d'énumérer toutes les combinaisons possibles, et de sélectionner celles qui impliquent un minimum de pièces. Cette solution, dite de force brute ou naïve, fonctionnera toujours mais est très loin d'être efficace. En effet, si elle est simple dans certains cas, elle implique en général un nombre très important de combinaisons différentes, ce qui nuit grandement à son efficacité.

La méthode gloutonne vise à optimiser la résolution d'un problème en partant du principe suivant : des choix locaux optimaux, étape après étape, devraient produire un résultat global optimal. Dans notre cas, on va répéter le choix de la pièce de plus grande valeur qui ne dépasse pas la somme restante.

Reprenons notre exemple. On doit donc rendre la somme de 2,63€ et on dispose du système de monnaie européen, à savoir les pièces de : 1, 2, 5, 10, 20, 50 centimes et 1 et 2 euros. La méthode gloutonne va d'abord choisir de rendre une pièce de 2 euros, il reste donc 63 centimes. Puis on choisira une pièce de 50 centimes etc. jusqu'à ce qu'il ne reste plus rien à rendre.

Supposons donnée la liste Python suivante : `systemeMonetaire = [200,100,50,20,10,5,2,1]` représentant les valeurs des pièces disponibles (en centimes).

Ecrire la fonction `renduMonnaie` prenant en argument le montant à rendre, et retournant la liste des pièces à rendre.

Que va retourner cet algorithme si l'on demande de rendre 6 centimes avec un système monétaire mettant à disposition uniquement des pièces de 1, 3 et 4 centimes ? Qu'en pensez-vous ?

Problème du choix d'activités

On considère un ensemble d'activités, chacune possédant une date de début et une date de fin. Pour des raisons logistiques, on ne peut pas programmer des activités se déroulant en même temps. La question est la suivante : quel est le nombre maximal d'activités que l'on va pouvoir planifier ?

Formalisons tout d'abord ce problème avec quelques notations mathématiques :

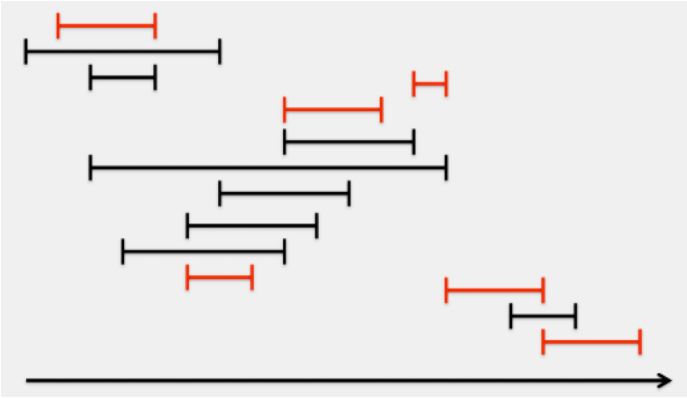
- L'ensemble des n activités peut être modélisé par un n-uplet de couples $S = ((b_1, e_1), (b_2, e_2), \cdots (b_n, e_n))$ où b_i représente la date de début de l'activité i et e_i sa date de fin.
- On suppose que $\forall i, 1 \leq i \leq n$, $b_i < e_i$ et que l'activité i se déroule pendant l'intervalle $[b_i; e_i[$
- Deux activités i et j sont dites compatibles si $b_i \geq e_j$ ou $b_j \geq e_i$. On peut dans ce cas les programmer toutes les deux.

Notre question peut alors être reformulée comme suit : on cherche un sous-ensemble d'activités compatibles dont le cardinal est le plus grand possible.

Considérons l'ensemble d'activités S :

$$S = ((1,4), (0,6), (3,5), (12,13), (8,11), (8,12), (2,13), (6,10), (5,9), (3,8), (5,7), (13,16), (15,17), (16,19))$$

Il est relativement facile de voir visuellement que le nombre maximal d'activités que l'on peut planifier est 6, et que celles-ci sont celles marquées en rouge dans le schéma suivant :



Dans cet exercie, nous allons imaginer différentes approches gloutonnes pour tenter de résoudre ce problème de manière automatique. On se posera bien sûr la question de leur optimalité. Pour ce faire, il nous faut trier les données, ici les activités, selon un critère spécifique. C'est le choix de ce critère qui différentiera une approche d'une autre.

Nous représenterons en Python l'ensemble S des activités par une liste de couples. En effet, nous serons amenés à effectuer des tris et même d'autres opérations sur S, il nous faut donc une structure de données muable.

Nous désignerons par C la solution de notre problème, c'est-à-dire la liste des activités choisies.

Tri par durée

Une première méthode consiste à prendre les activités par ordre croissant de leur durée. On peut en effet raisonnablement penser que planifier d'abord les activités les plus courtes laissera plus de temps pour les autres.

Cela donne cette résolution :

1. Trier les activités par ordre croissant de leur durée
2. Initialiser la liste C des activités choisies avec l'ensemble vide
3. Parcourir dans l'ordre la liste S des activités, et rajouter l'activité courante à C si elle n'est pas incompatible avec celles déjà présentes dans C

Cette approche est bien gloutonne. A chaque étape on considère que le meilleur choix à effectuer est de prendre l'activité la plus courte compatible avec toutes celles déjà planifiées.

Ecrire la fonction `choixActivitesDuree` qui prend en argument la liste de toutes les activités S et retourne la liste des activités choisies C.

Que donne l'exécution de cet algorithme sur l'exemple ? Est-ce la solution optimale ?

Tri par date de début

Une autre approche est de considérer les activités par ordre croissant de leur date de début. L'idée sous-jacente étant de perdre le moins de temps possible en commençant au plus vite une nouvelle activité. Ici, le choix glouton consiste à prendre à chaque étape l'activité commençant le plus tôt, à condition qu'elle soit compatible avec celles déjà planifiées.

Ecrire la fonction `choixActiviteDateDebut` correspondante.

Que donne l'exécution de cet algorithme sur l'exemple ? Est-ce la solution optimale ?

Tri par nombre d'incompatibilités

Une troisième possibilité revient à prendre les activités par ordre croissant de leurs incompatibilités. Procéder ainsi permet de privilégier les activités qui "gènent" le moins, et donc avoir moins de contraintes pour planifier les autres.

Ici le critère de tri (et donc le choix glouton) est moins simple que dans les deux autres méthodes. Il va nous falloir d'abord une procédure nous permettant d'évaluer le nombre d'activités incompatibles de chaque activité.

Ecrire une procédure `ajoutIncompatible` qui prend en argument une liste d'activité S et qui transforme les couples (début, fin) de cette liste en triplets en rajoutant le nombre d'activités incompatibles (début, fin, nombre d'activités incompatibles).

Ecrire la fonction `choixActivitesNbIncompatible` qui retourne la liste des activités choisies par cette méthode (sous la forme de triplets).

Que donne l'exécution de cet algorithme sur l'exemple ? Est-ce la solution optimale ?

Tri par date de fin

Les tentatives précédentes avaient certes du sens, mais elles avaient comme défaut commun de ne pas être optimales. Il est donc temps de présenter un algorithme qui le soit. Pour cela nous allons envisager les activités par ordre croissant de leur date de fin. On va ainsi chercher à laisser le plus de temps disponible après la planification d'une activité.

Ecrire la fonction `choixActivitesDateFin` qui choisie les activités en fonction de leur date de fin.

Que donne l'exécution de cet algorithme sur l'exemple ? Démontrer l'optimalité de cette stratégie.

On obtiendrait un résultat équivalent en classant les activités par ordre décroissant de leur date de début. Mais cette façon de voir la chose est moins intuitive.

Quelle est la complexité de ce dernier algorithme ?