Esiee-Paris - unité d'algorithmique - Premier projet

Janvier 2021 – R. Natowicz, I. Alame, A. Çela, D. Courivaud, D. Garrigue

Merci à Paul Courtial et Gabriel Leroux qui ont repéré une erreur dans la première version de l'énoncé :-)

1. QuickSelect.

Soit $T[0:n]$ un tableau d'entiers. Imaginons-le croissant. Alors, sa p-ème valeur, $0 \leq p < n$, est $T[p]$.

Exemple : avec $n = 4$ et $T[0:n] = [5, 8, 3, 5]$ les $p = 0, 1, 2$ et 3-èmes valeurs sont respectivement 3, 5, 5, et 8. Nous avons donc un premier algorithme de calcul de la p-ème valeur de T : " trier T puis retourner $T[p]$. " Sa complexité est en $\Theta(n \log_2 n)$. Peut-on faire mieux ? Oui !

Nous devons au très grand C.A.R. Hoare l'algorithme QuickSelect qui retourne la p-ème valeur du sous-tableau $T[i:j]$ sans trier ce sous-tableau. En pratique son temps de calcul est en $\Theta(n)$. Son pire cas, de probabilité presque nulle, est en $\Theta(n^2)$. Cet algorithme repose sur la décomposition récursive ci-dessous, dans laquelle $m = j - i$ est la taille du sous-tableau $T[i:j]$.

- Si $m = 1$, p est nécessairement égal à 0. La p-ème valeur de $T[i:j]$ est $T[i]$;
- si $m > 1$, p est nécessairement dans l'intervalle $[0:m]$.
  Calculons une permutation de $T[i:j]$ vérifiant $T[i:k] \leq T[k] < T[k+1:j]$ (voir la procédure `segmenter` du cours) Alors :
  - si $p + i \in [i:k]$, la p-ième valeur de $T[i:j]$ est ... À propos : quelle est cette p-ème valeur ?
  - si $p + i \in [k:j]$, la p-ième valeur de $T[i:j]$ est ... Quelle est cette p-ème valeur ?

– Écrire la fonction récursive `int qsel(int p, int[] T, int i, int j)`, $0 \leq p < j - i$, qui retourne la p-ème valeur de $T[i:j]$.

– Écrire une version itérative (invariant et programme commenté par l'invariant.)

Remarque : dans la vie courante, on ne parle pas de la 0-ème valeur mais plutôt de la 1ère valeur, de la 2ème valeur, etc., et de la $n$ème valeur de $T[0:n]$. Pour se conformer à cet usage, on définit la fonction `quickSelect` ci-dessous :

```
int quickSelect(int p, int[] T) { // retourne la p-ème valeur de T[0:n], avec 1 <= p <= n
   int n = T.length ;
   return qsel(p-1, T, 0, n);
}
```

La valeur médiane de $T[0:n]$ est la valeur de T qui permet de "partager" ses valeurs en deux sous-ensembles de tailles égales : autant de valeurs supérieures ou égales à la médiane que de valeurs inférieures ou égales. La médiane est une valeur statistique très importante car, contrairement à la moyenne, la médiane est insensible aux valeurs extrêmes, également appelées *outliers*[1]. La médiane est donc plus *robuste* que la moyenne.

```
int mediane(int[] T) { // retourne la valeur médiane de T
   int n = T.length ;
   return quickSelect((n+1)/2, T);
}
```

2. Remplir un sac.

La dernière partie du cours sera consacrée à la programmation dynamique (voir la description de l'unité.) Cette méthode algorithmique traite de problèmes d'optimisation. Un problème d'optimisation est le calcul de la valeur optimale (maximum ou minimum) d'un ensemble.

Exemples de problèmes que nous traiterons dans cette partie du cours :

- Vous disposez d'un certain nombre d'heures de révisions pour préparer les contrôles de fin de semestre. Comment les répartir sur les différentes unités pour obtenir une somme de notes maximum ?
  Il est ici question de l'ensemble des répartitions possibles des heures de révision sur les unités qui feront l'objet des contrôles de fin de semestre. À chaque répartition correspond une somme de notes. La somme des notes est la valeur de la répartition. On cherche une répartition de valeur maximum.

[1] `https://dictionary.cambridge.org/fr/dictionnaire/anglais/outlier`.

- Un magasin dispose d'une planche de longueur L qu'il peut débiter en tronçons de longueurs quelconques, comprises entre 1 et L (dans ce dernier cas, la planche est vendue en l'état.) Pour toute longueur l, $1 \leq l \leq L$, vous connaissez le bénéfice de la vente d'un tronçon de longueur l. On demande de calculer un débitage de la planche, de bénéfice total maximum.

  Il est ici question de tous les débitages possibles de la planche. La valeur d'un débitage est la somme des bénéfices des tronçons débités. On cherche un débitage de valeur maximum.

- Un ensemble d'objets est à disposition, chaque objet i a une taille t_i et une valeur v_i et vous disposez d'un sac d'une certaine contenance. On demande de calculer un sous-ensemble d'objets dont la somme des valeurs est maximum, sous la contrainte que ces objets puissent entrer dans le sac (la somme de leurs tailles est inférieure ou égale à la contenance du sac.)

  On parle ici de l'ensemble des sous-ensembles d'objets dont la somme des tailles est inférieure ou égale à la capacité du sac. La valeur d'un sac est la somme des valeurs des objets qu'il contient. On cherche un sac de valeur maximum.

La programmation dynamique garantit l'optimalité des solutions trouvées : la somme des notes, le bénéfice total du débitage, la valeur du sac, calculés par programmation dynamique, sont maximum.

Les méthodes "gloutonnes", quant à elles, construisent une solution en procédant par améliorations "locales." Les solutions gloutonnes ne retournent pas, dans le cas général[2], les solutions optimales, mais ce sont les premières auxquelles on pense...

Exemples de méthodes gloutonnes pour les problèmes ci-dessus :

- Pour chaque unité, ajouter une heure de révision ne fait pas baisser la note au contrôle, et parfois, cet ajout la fait même augmenter... L'augmentation éventuelle de la note d'une unité est fonction du temps que l'on consacre déjà à la réviser. En effet : ajouter une heure de révision à une unité à laquelle on consacre déjà 40 heures de révision aura moins d'influence sur la note qu'ajouter une heure de révision lorsqu'on n'y consacre que 5 heures. Ceci dépend bien sûr des unités en question.

  L'approche gloutonne alloue une heure supplémentaire à l'une des unités dont elle fera le plus augmenter la note. Les heures sont ainsi distribuées, une à une, jusqu'à épuisement des heures de révision disponibles.

- Soit l^*, $l^* \leq r = L$, la longueur du tronçon de bénéfice maximum. Couper un tronçon de cette longueur l^*. Il reste une planche de longueur $r = L - l^*$. Continuer de la sorte jusqu'à ce que $r = 0$.

- Soit i^* un objet de taille inférieure ou égale à la contenance $r = c$ de votre sac, objet qui, parmi ceux qui ne sont pas déjà dans le sac, est de valeur maximum. Le mettre dans le sac. Il reste une contenance $r = c - t_{i^*}$. Continuer de la sorte jusqu'à ce qu'aucun objet ne puisse entrer dans le sac de contenance r.

  On peut également définir la "densité de valeur" $d_i = \frac{v_i}{t_i}$ de l'objet i et procéder par densités de valeurs décroissantes (plutôt que par valeurs décroissantes.)

On demande de mettre en oeuvre la stratégie gloutonne de remplissage d'un sac par valeurs décroissantes, et par densités de valeurs décroissantes.

Un objet est la donnée de ses numéro, taille, valeur et densité de valeur. Pour le représenter on définit une classe : `class objet{int i, v, t; float d;}` (numéro, valeur, taille et densité de valeur). Les n objets sont dans un tableau $T[0:n]$ qui aura été créé par l'instruction `objet[] T = new objet[n]`.

1. Écrire une fonction `boolean[] sacGloutonParValeurs(objet[] T, int c)` de terme général $T[k]$ = " l'objet k est dans le sac de contenance c " ;
2. écrire une fonction `boolean[] sacGloutonParDensitésDeValeurs(objet[] T, int c)` de terme général $T[k]$ = " l'objet k est dans le sac de contenance c " ;
3. écrire une fonction `int valeur(boolean[] g, objet[] T)` qui retourne la valeur d'un sac glouton g construit sur l'ensemble d'objets représenté par le tableau T ;
4. écrire une fonction `objet[] aleatoires(int n, int vsup, int tsup)` qui retourne un tableau de n objets : l'objet i est en case i, sa valeur est au hasard dans l'intervalle $[0 : v_{\text{sup}}]$, sa taille est au hasard dans l'intervalle $[1 : t_{\text{sup}} + 1]$.

Vous comparerez les solutions gloutonnes par valeurs et gloutonnes par densités de valeurs à l'aide d'une validation statistique. Cette validation calcule les valeurs de sacs gloutons construits sur des ensembles d'objets aléatoires. Pour chaque ensemble d'objets aléatoires on calcule et on mémorise la valeur du sac glouton par valeurs, et de même pour le sac glouton par densités de valeur. On procède ainsi pour un grand nombre d'ensembles d'objets aléatoires. Puis on retourne la médiane des sacs gloutons par valeurs et la médiane des sacs gloutons par densités de valeurs[3].

Donner les valeurs médianes des deux approches gloutonnes pour $N = 10^6$ ensembles d'objets aléatoires, $n_{\text{sup}} = 10^3$, $C_{\text{sup}} = 10^3$, $t_{\text{sup}} = c/100$ où c est la contenance du sac choisie au hasard dans l'intervalle $[0 : C_{\text{sup}}]$, et $v_{\text{sup}} = 10^3$.

[2] Certains problèmes d'optimisation se résolvent par une méthode gloutonne, mais ces problèmes sont rares.

[3] Une validation statistique complète nous dirait, en plus, si ces deux valeurs médianes sont significativement différentes. Elle retournerait les deux valeurs médianes et la "p-value" des valeurs des sacs gloutons par valeurs et gloutons par densités de valeurs. La notion de p-value relève d'un cours de statistiques plus que d'un cours d'algorithmique. Pour en savoir plus, je vous recommande la vidéo de Josh Starmer, `https://www.youtube.com/watch?v=5Z9OIYA8He8` de la société StatQuest.

Le projet est à réaliser en binôme. Les monômes doivent être l'exception (nombre impair d'étudiants dans un groupe, etc.)

Les projets sont à envoyer à votre intervenant de TD de la semaine du 8 février. La date limite d'envoi est vendredi 12 février.

Votre courrier électronique contiendra deux fichiers : le programme java (texte source) avec vos commentaires, remarques, exemples d'exécution ; et le même fichier au format PDF.

Ne pas oublier d'indiquer les noms du binôme.

Remarque importante : les sapiens (que nous sommes) ne lisent pas les programmes non commentés ou mal commentés.