# Sac à dos

## 1 Présentation du problème

On considère $n$ objets numérotés de 0 à $n-1$ ayant chacun un poids $p_i \in \mathbb{N}$ et une valeur $v_i \in \mathbb{N}$. On dispose d'un sac à dos pouvant contenir un nombre quelconque d'objets tant que la somme de leurs poids ne dépasse pas une constante $p$ connue. On souhaite choisir un sous-ensemble des objets de valeur totale maximale parmi ceux rentrant dans le sac à dos.

Ce problème, dont le nom usuel est 0, 1-KNAPSACK, est un exemple célèbre de problème *np*-complet, ce qui signifie en particulier que l'on ne connaît pas de solution en temps polynomial en $n$. Cependant, nous allons voir qu'il est possible de trouver une solution par programmation dynamique en temps polynomial en $n$ **et** $\boldsymbol{p}$.

Dans tout le problème, les poids et les valeurs seront donnés sous forme de deux `int array` que l'on appellera `p` et `v`. On notera en ocaml `pmax` le poids autorisé $p$. Comme d'habitude, on commencera par tenter de calculer la valeur optimale et l'on ne s'intéressera qu'ensuite au problème de la recherche d'une solution optimale.

1. *Force brute* : Si l'on souhaite procéder en énumérant toutes les solutions possibles :
   (a) Combien faut-il en considérer ?
   (b) Quelle complexité peut-on raisonnablement attendre ?

## 2 Une relation de récurrence

On définit $f(k, d)$ la valeur maximale que l'on peut obtenir si le poids disponible est $d$ et que l'on se limite à choisir des objets dont l'indice est strictement inférieur à $k$.

1. Quelle valeur de $f$ veut-on calculer ?
2. Exprimer $f(1, d)$ en fonction de $d$, $p_0$ et $v_0$.
3. Pour $k \geqslant 1$, donner une relation entre $f(k+1, d)$ et les $f(k, d')$ (faisant bien sûr intervenir les poids et les valeurs des objets).
4. Quelle valeur peut-on raisonnablement donner à $f(0, d)$ (avec $d \geqslant 0$) ? Quel intérêt cela présente-t-il ?
5. Peut-on de même définir $f(k, d)$ quand $d < 0$ ?

1. *solution récursive naïve* ! : Écrire une fonction `sac p v pmax` répondant au problème posé. on utilisera une fonction auxiliaire récursive `f k d` correspondant à la définition vue plus haut.

## 3 Programmation dynamique ascendante et descendante

1. *Chevauchement de sous-problèmes* : Comme nous l'avons vu en cours, deux conditions doivent être réunies pour que l'approche par programmation dynamique ait un intérêt : la présence de sous-structures optimales, que nous avons mise en évidence à la partie précédente, et le chevauchement de sous-problemes. Contrairement aux exemples que nous avons vus jusqu'à présent, il n'est pas évident ici de déterminer *a priori* combien de fois chaque appel à `f n d` sera effectué. nous allons donc utiliser une approche expérimentale.
   (a) Pour les `p_ex` et `v_ex` fournis et en prenant $p = 100$, combien y a-t-il au maximum d'appels **distincts** à `f n d` ? *on veut juste une majoration, le nombre exact d'appels distincts dépend du contenu de* `p_ex` *et de* `v_ex`.
   (b) Écrire une fonction `sac_instrumente p v pmax` effectuant le même travail que `sac` mais comptant en parallèle le nombre total d'appels à `f` effectués. Le plus simple est de définir une variable mutable que l'on incrémentera à chaque appel.
   (c) Dans le cas du 1.a, que peut-on dire du nombre moyen de fois que l'on effectue chaque appel (distinct) ?
2. Écrire deux fonctions `sac_mem p v pmax` et `sac_dyn p v pmax` implémentant respectivement les versions descendantes et ascendantes de la solution par programmation dynamique du problème. Pour créer un tableau de tableaux (c'est-à-dire une matrice) de dimensions $n \times p$ initialisée avec la valeur $x$, on pourra utiliser `array.make_matrix n p x`.
   *Comme souvent, on prendra une table de type* `int array array` *dans le cas ascendant et* `int option array array` *dans le cas descendant.*
3. Déterminer (ou majorer de manière pas trop grossière) les complexités en temps et en espace des deux fonctions de la question précédente.

# 4 Reconstruction de la solution

1. Reconstruire une solution optimale demande de retrouver la série de choix qui a permis d'obtenir la valeur optimale. Souvent, le plus simple est de stocker le choix fait (on prend l'objet ou pas) dans chaque case de la table.
   — Dans le cas descendant, on peut stocker dans chaque case un couple [1] $(f(k,d), s)$ où $s$ est une liste correspondant à une solution optimale pour les objets 0 à $k$ avec un poids maximal de $d$. Il n'y a alors pas de phase de reconstruction à proprement parler : on construit la solution au fur et à mesure des appels récursifs.
   — Dans le cas ascendant, il est plus naturel d'utiliser un `(int * bool) array array`, où le booléen indique si l'objet a été choisi. Il faut alors reconstruire la liste solution dans un deuxième temps.

   Implémenter l'une de ces approches (au choix). On renverra un couple $(valeur_{opt}, solution_{opt})$, où la solution est donnée sous forme d'une liste d'entiers.
2. En réalité, on peut ici assez facilement retrouver la série de choix effectués sans stocker d'information supplémentaire (c'est-à-dire en travaillant avec les mêmes tables qu'à la partie précédente). Expliquer comment on procéderait.

# 5 De l'intérêt des deux approches

1. *Intérêt de la solution mémoïsée* : Déterminer le nombre de valeurs de $f(k,d)$ que la fonction `sac_mem` calcule pour $p = 100$ avec les $p$ et $v$ donnés en exemple. Aue constate-t-on ?
2. *Intérêt de la solution bottom-up* : Expliquer comment diminuer la complexité en espace de la fonction `sac_dyn` (et le faire si vous avez le temps). Auel problème cela pose-t-il ?

1. ou plutôt une option sur un tel couple