Introduction à l'algorithmique et la complexité (et un peu de CAML)

Programmation Dynamique

Nicolas Nisse

Université Côte d'Azur, Inria, CNRS, I3S, France

Cours dispensés en MPSI (option Info) au CIV, depuis 2011-

http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/prepa/

Outline

Problème de rendu de Monnaie

Supposons que vous êtes un commerçant. Un client vous achête pour $127,36$ € de marchandise. Il vous tend un billet de 200 €.

Pour lui rendre la monnaie, vous disposez de pièces de $1, 5, 10, 20, 50$ centimes d'euros, de pièces de 1 et 2 € et de billets de $5, 10, 20, 50$ et 100 €.

Pour ne pas déplaire au client (en surchargeant ses poches avec 7264 pièces de 0.01 €) :

Vous voulez lui rendre le moins de pièces/billets que possible.

Que lui rendez vous ?

Problème de rendu de Monnaie (formalisation)

Formaliser (Décrire mathématiquement) un problème concret fait partie d'une des tâches les plus importantes (et souvent difficiles) des chercheurs.

Prenons l'exemple du problème de rendu de monnaie.

- **Entrée** 1 **:** Un montant $M \in \mathbb{R}$ à rendre (en euro) ($M = 200 - 127.36$ dans l'ex. précédent)
- **Entrée** 2 **:** Un système de monnaie $(x_1 \leq \cdots \leq x_r) \in \mathbb{R}^r$ (les "types" de pièces/billets dont on dispose, e.g., $1, 2, 5, 10, 20$ euros)
- **Sortie :** Décider de ce que l'on rend au client : pour chaque type i de pièces/billets ($1 \leq i \leq r$), quel est le nombre k_i de pièces/billets de type i (de valeur x_i) rendus on au client ?
- **Contrainte :** Il faut rendre au client ce qu'on lui doit : la somme des pièces/billets qu'on lui rend doit être égale à M : $\sum_{1 \leq i \leq r} k_i x_i = M$.
- **Optimisation :** On veut minimiser le nombre $\sum_{1 \leq i \leq r} k_i$ de pièces/billets rendus.

Problème de rendu de Monnaie

Étant donnés $M \in \mathbb{R}$ et $(x_1, \cdots, x_r) \in \mathbb{R}^r$
Calculer $(k_1, \cdots, k_r) \in \mathbb{N}^r$ tels que $\sum_{1 \leq i \leq r} k_i$ soit minimum sous réserve que $\sum_{1 \leq i \leq r} k_i x_i = M$.

Outline

Problèmes de Décision / d'Optimisation

Problème de rendu de Monnaie

Étant donnés $M \in \mathbb{R}^+$ et $(x_1, \cdots, x_r) \in (\mathbb{R}^+)^r$
Calculer $(k_1, \cdots, k_r) \in \mathbb{N}^r$ tels que $\sum_{1 \leq i \leq r} k_i$ soit minimum sous réserve que $\sum_{1 \leq i \leq r} k_i x_i = M$.

Exercices : trouver une (des) solutions aux problèmes suivants

- $M = 13$ et $(x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 7, x_5 = 9)$
- $M = 73$ et $(x_1 = 4, x_2 = 12)$.

Problèmes de Décision / d'Optimisation

Problème de rendu de Monnaie

Étant donnés $M \in \mathbb{R}^+$ et $(x_1, \cdots, x_r) \in (\mathbb{R}^+)^r$
Calculer $(k_1, \cdots, k_r) \in \mathbb{N}^r$ tels que $\sum_{1 \leq i \leq r} k_i$ soit minimum sous réserve que $\sum_{1 \leq i \leq r} k_i x_i = M$.

Exercices : trouver une (des) solutions aux problèmes suivants

- $M = 13$ et $(x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 7, x_5 = 9)$
  $(k_1) = (13)$ (13 pièces de 1 €) (solution *valide* mais pas *optimale*)
  $(k_3, k_4) = (2, 1)$ (2 pièces de 3 € et 1 de 7 €) (solution *optimale*, i.e., minimisant le nombre de pièces)
  $(k_2, k_5) = (2, 1)$ (2 pièces de 2 € et 1 de 9 €) **Prouvez que c'est une solution** *optimale*
- $M = 73$ et $(x_1 = 4, x_2 = 12)$.
  Toute combinaison linéaire de pièces de valeur x_1 et x_2 donne une valeur paire. Il n'y a donc pas de solution valide (satisfaisant la contrainte $\sum k_i x_i = M$).

Problèmes de Décision / d'Optimisation

Problème de rendu de Monnaie

Étant donnés $M \in \mathbb{R}^+$ et $(x_1, \cdots, x_r) \in (\mathbb{R}^+)^r$
Calculer $(k_1, \cdots, k_r) \in \mathbb{N}^r$ tels que $\sum_{1 \leq i \leq r} k_i$ soit minimum sous réserve que $\sum_{1 \leq i \leq r} k_i x_i = M$.

Exercices : trouver une (des) solutions aux problèmes suivants

- $M = 13$ et $(x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 7, x_5 = 9)$
  $(k_1) = (13)$ (13 pièces de 1 €) (solution *valide* mais pas *optimale*)
  $(k_3, k_4) = (2, 1)$ (2 pièces de 3 € et 1 de 7 €) (solution *optimale*, i.e., minimisant le nombre de pièces)
  $(k_2, k_5) = (2, 1)$ (2 pièces de 2 € et 1 de 9 €) **Prouvez que c'est une solution *optimale***
- $M = 73$ et $(x_1 = 4, x_2 = 12)$.
  Toute combinaison linéaire de pièces de valeur x_1 et x_2 donne une valeur paire. Il n'y a donc pas de solution valide (satisfaisant la contrainte $\sum k_i x_i = M$).

Contrairement aux problèmes étudiés précédement (tri d'un tableau, calcul de x^c...), où il y avait toujours une unique solution, dans les problèmes d'optimisation, il peut y avoir des solutions valides (satisfaisant les contraintes) non optimales, une ou plusieurs solutions (valides) optimales (minimisant/maximisant une certaine mesure), voire pas du tout de solution valide.

Décider si un problème admet une solution (valide) peut être "difficile" (voire "non-décidable"). Dans la suite, nous nous assurerons qu'il en existe toujours (à vous de le vérifier). Pour le problème du rendu de Monnaie, il est suffisant de supposer que $M \in \mathbb{N}^*$ et $(x_1 = 1, x_2, \cdots, x_n) \in \mathbb{N}^n$ pour qu'il y ait toujours (au moins) une solution valide (**prouvez le !**).

Outline

Rendu de Monnaie : Algorithme Glouton (1/2)

Résolvez l'exemple suivant : $M = 78$ et $(x_1, \cdots, x_5) = (1, 2, 5, 10, 50)$.

La solution que vous avez (probablement) proposée est $(k_1, \cdots, k_5) = (1, 1, 1, 2, 1)$, i.e., une piéce de $1, 2, 5$ et 50 € et 2 pièces de 10 €.

Cette solution (avec 6 pièces) est-elle optimale ?

Rendu de Monnaie : Algorithme Glouton (1/2)

Résolvez l'exemple suivant : $M = 78$ et $(x_1, \cdots, x_5) = (1, 2, 5, 10, 50)$.
La solution que vous avez (probablement) proposée est $(k_1, \cdots, k_5) = (1, 1, 1, 2, 1)$, i.e., une piéce de $1, 2, 5$ et 50 € et 2 pièces de 10 €.

Cette solution (avec 6 pièces) est-elle optimale ?

Rendu de Monnaie : Algorithme Glouton (1/2)

Résolvez l'exemple suivant : $M = 78$ et $(x_1, \cdots, x_5) = (1, 2, 5, 10, 50)$.
La solution que vous avez (probablement) proposée est $(k_1, \cdots, k_5) = (1, 1, 1, 2, 1)$, i.e., une piéce de $1, 2, 5$ et 50 € et 2 pièces de 10 €.

Cette solution (avec 6 pièces) est-elle optimale ?

Toute solution sans pièce de valeur x_5 utilise au moins $\lceil M/x_4 \rceil = 8$ pièces (**prouvez le**). Donc, toute solution utilisant moins que 8 pièces doit utiliser une pièce de type x_5. Ainsi, le nombre minimum de pièce pour rendre M vaut 1 plus le nombre minimum de pièce pour rendre $M - x_5$. En répétant récursivement cet argument, prouvez qu'il s'agit d'une solution optimale.

Rendu de Monnaie : Algorithme Glouton (1/2)

Résolvez l'exemple suivant : $M = 78$ et $(x_1, \cdots, x_5) = (1, 2, 5, 10, 50)$.
La solution que vous avez (probablement) proposée est $(k_1, \cdots, k_5) = (1, 1, 1, 2, 1)$, i.e., une piéce de $1, 2, 5$ et 50 € et 2 pièces de 10 €.

Cette solution (avec 6 pièces) est-elle optimale ?

Toute solution sans pièce de valeur x_5 utilise au moins $\lceil M/x_4 \rceil = 8$ pièces (**prouvez le**). Donc, toute solution utilisant moins que 8 pièces doit utiliser une pièce de type x_5. Ainsi, le nombre minimum de pièce pour rendre M vaut 1 plus le nombre minimum de pièce pour rendre $M - x_5$. En répétant récursivement cet argument, prouvez qu'il s'agit d'une solution optimale.

Quel algorithme avez vous (probablement) utilisé ?

Rendu de Monnaie : Algorithme Glouton (1/2)

Résolvez l'exemple suivant : $M = 78$ et $(x_1, \cdots, x_5) = (1, 2, 5, 10, 50)$.
La solution que vous avez (probablement) proposée est $(k_1, \cdots, k_5) = (1, 1, 1, 2, 1)$, i.e., une piéce de $1, 2, 5$ et 50 € et 2 pièces de 10 €.

Cette solution (avec 6 pièces) est-elle optimale ?

Toute solution sans pièce de valeur x_5 utilise au moins $\lceil M/x_4 \rceil = 8$ pièces (**prouvez le**). Donc, toute solution utilisant moins que 8 pièces doit utiliser une pièce de type x_5. Ainsi, le nombre minimum de pièce pour rendre M vaut 1 plus le nombre minimum de pièce pour rendre $M - x_5$. En répétant récursivement cet argument, prouvez qu'il s'agit d'une solution optimale.

Quel algorithme avez vous (probablement) utilisé ?

Rendre la pièce de plus grande valeur $x_i \leq M$. Il reste $M - x_i$ à rendre. Si $M - x_i = 0$ alors, c'est fini, sinon, recommencer avec la nouvelle valeur $M - x_i > 0$ à rendre.

```
#let MonnaieGreedy m systeme =
        let n = vect_length systeme in
        let rest = ref m in
        let res = make_vect n 0 in
        let i = ref (n-1) in
        while !rest>0 do
                if systeme.(!i)<= !rest then
                        begin
                        res.(!i) <- res.(!i) + 1;
                        rest := !rest - systeme.(!i);
                        end
                else i := !i -1;
        done;
        res;;
MonnaieGreedy : int -> int vect -> int vect = <fun>
#let t = [|1;2;5;10;50|];;
t : int vect = [|1; 2; 5; 10; 50|]
#MonnaieGreedy 78 t;;
- : int vect = [|1; 1; 1; 2; 1|]
```

Correction : si $m \in \mathbb{N}^*$ et $(x_1 = 1 < x_2 \cdots < x_n) \in \mathbb{N}^n$, *MonnaieGreedy* donne une solution valide !!

Complexité : $O(n + m)$

Rendu de Monnaie : Algorithme Glouton (1/2)

Résolvez l'exemple suivant : $M = 78$ et $(x_1, \cdots, x_5) = (1, 2, 5, 10, 50)$.
La solution que vous avez (probablement) proposée est $(k_1, \cdots, k_5) = (1, 1, 1, 2, 1)$, i.e., une piéce de $1, 2, 5$ et 50 € et 2 pièces de 10 €.

Cette solution (avec 6 pièces) est-elle optimale ?

Toute solution sans pièce de valeur x_5 utilise au moins $\lceil M/x_4 \rceil = 8$ pièces (**prouvez le**). Donc, toute solution utilisant moins que 8 pièces doit utiliser une pièce de type x_5. Ainsi, le nombre minimum de pièce pour rendre M vaut 1 plus le nombre minimum de pièce pour rendre $M - x_5$. En répétant récursivement cet argument, prouvez qu'il s'agit d'une solution optimale.

Quel algorithme avez vous (probablement) utilisé ?

Rendre la pièce de plus grande valeur $x_i \leq M$. Il reste $M - x_i$ à rendre. Si $M - x_i = 0$ alors, c'est fini, sinon, recommencer avec la nouvelle valeur $M - x_i > 0$ à rendre.

Il s'agit d'un algorithme glouton ("greedy" en anglais).

À chaque étape (ici, à chaque fois qu'on rend une pièce), on choisit la solution qui semble localement (à cette étape) la meilleure (ici, on rend la pièce qui diminue la somme à rendre au maximum).

L'algorithme glouton trouve t'il toujours une solution optimale pour le problème de rendu de monnaie ?

```
#let MonnaieGreedy m systeme =
        let n = vect_length systeme in
        let rest = ref m in
        let res = make_vect n 0 in
        let i = ref (n-1) in
        while !rest>0 do
                if systeme.(!i)<= !rest then
                        begin
                        res.(!i) <- res.(!i) + 1;
                        rest := !rest - systeme.(!i);
                        end
                else i := !i -1;
        done;
        res;;
MonnaieGreedy : int -> int vect -> int vect = <fun>
#let t = [|1;2;5;10;50|];;
t : int vect = [|1; 2; 5; 10; 50|]
#MonnaieGreedy 78 t;;
- : int vect = [|1; 1; 1; 2; 1|]
```

Correction : si $m \in \mathbb{N}^*$ et $(x_1 = 1 < x_2 \cdots < x_n) \in \mathbb{N}^n$, *MonnaieGreedy* donne une solution valide !!

Complexité : $O(n + m)$

Rendu de Monnaie : Algorithme Glouton (2/2)

Nouvel exemple : $M = 6$ et $(x_1, x_2, x_3) = (1, 3, 4)$.

Donnez le résultat obtenu par l'algorithme glouton (du slide précédent) :

Donnez une solution optimale :

Rendu de Monnaie : Algorithme Glouton (2/2)

Nouvel exemple : $M = 6$ et $(x_1, x_2, x_3) = (1, 3, 4)$.

Donnez le résultat obtenu par l'algorithme glouton (du slide précédent) : $(k_1, k_2, k_3) = (2, 0, 1)$

Donnez une solution optimale : $(k_1, k_2, k_3) = (0, 2, 0)$

L'algorithme glouton ne donne donc pas toujours une solution optimale !!

Rendu de Monnaie : Algorithme Glouton (2/2)

Nouvel exemple : $M = 6$ et $(x_1, x_2, x_3) = (1, 3, 4)$.

Donnez le résultat obtenu par l'algorithme glouton (du slide précédent) : $(k_1, k_2, k_3) = (2, 0, 1)$

Donnez une solution optimale : $(k_1, k_2, k_3) = (0, 2, 0)$

L'algorithme glouton ne donne donc pas toujours une solution optimale !!

Remarque : Un système $\mathscr{S} = (x_1, \cdots, x_n)$ est dit *canonique* si l'algorithme glouton produit toujours (pour tout M) une solution optimale pour $\mathscr{S}$.

Prouvez que, tout système avec 2 éléments ($n = 2$) est canonique.

Pour $n \geq 3$, on ne sait pas déterminer "simplement" (en gros, sans tester toutes les possibilités) si un système avec n éléments est canonique.

Rendu de Monnaie : Algorithme Glouton (2/2)

Nouvel exemple : $M = 6$ et $(x_1, x_2, x_3) = (1, 3, 4)$.

Donnez le résultat obtenu par l'algorithme glouton (du slide précédent) : $(k_1, k_2, k_3) = (2, 0, 1)$

Donnez une solution optimale : $(k_1, k_2, k_3) = (0, 2, 0)$

L'algorithme glouton ne donne donc pas toujours une solution optimale !!

Remarque : Un système $\mathscr{S} = (x_1, \cdots, x_n)$ est dit *canonique* si l'algorithme glouton produit toujours (pour tout M) une solution optimale pour $\mathscr{S}$.

Prouvez que, tout système avec 2 éléments ($n = 2$) est canonique.

Pour $n \geq 3$, on ne sait pas déterminer "simplement" (en gros, sans tester toutes les possibilités) si un système avec n éléments est canonique.

Puisque l'algorithme glouton n'est pas forcément optimal, nous allons décrire dans la suite 2 algorithmes, basés sur la programmation dynamique, qui donnent toujours une solution optimale pour le problème de rendu de monnaie.

Outline

Rendu de Monnaie : Algorithme Optimal 1

Soit $M, n \in \mathbb{N}^*$ et $\mathscr{S}_n = (x_1 = 1 < x_2, \cdots < x_n) \in \mathbb{N}^n$ une instance du problème.
On note $Valeur_{Opt}(M,n) = \min_{(k_1,\cdots,k_n)\in\mathbb{N}^n}\{\sum_{1\leq i\leq n} k_i \mid \sum_{1\leq i\leq n} k_i x_i = M\}$ la valeur (nombre de pièces) d'une solution optimale pour l'instance $(M, \mathscr{S}_n)$.

Théorème :

- $Valeur_{Opt}(M,1) = M$;
- $\forall n > 1$, si $x_n > M$ alors $Valeur_{Opt}(M,n) = Valeur_{Opt}(M,n-1)$, sinon
- $Valeur_{Opt}(M,n) = \min\{Valeur_{Opt}(M,n-1); 1 + Valeur_{Opt}(M-x_n,n)\}$

Preuve : Les 2 premiers points sont évidents, donc supposons que $n > 1$ et $M \leq x_n$.

Le 3^{me} point se prouve par double inégalité.

Toute solution $(k_1,\cdots,k_{n-1})$ valide pour $(M,\mathscr{S}_{n-1})$ (i.e., $\sum_{1\leq i<n} k_i x_i = M$) est aussi valide pour $(M,\mathscr{S}_n)$ (en posant $k_n = 0$).

Donc $Valeur_{Opt}(M,n) \leq Valeur_{Opt}(M,n-1)$. Par ailleurs, toute solution $(k_1,\cdots,k_n)$ valide pour $(M-x_n,\mathscr{S}_n)$ (telle que $\sum_{1\leq i\leq n} k_i x_i = M - x_n$) permet d'obtenir une solution $(k_1,\cdots,k_{n-1},k_n+1)$ valide pour $(M,\mathscr{S}_n)$ en ajoutant une pièce de valeur x_n. Donc $Valeur_{Opt}(M,n) \leq 1 + Valeur_{Opt}(M-x_n,n)$.

On en déduit que $Valeur_{Opt}(M,n) \leq \min\{Valeur_{Opt}(M,n-1); 1 + Valeur_{Opt}(M-x_n,n)\}$.

(on utilise le fait que si $(a \geq c$ et $b \geq c)$ alors $\min\{a,b\} \geq c$).

Soit $(k_1,\cdots,k_n)$ une solution optimale pour $(M,\mathscr{S}_n)$. Si $k_n = 0$, c'est une solution valide pour $(M,\mathscr{S}_{n-1})$ et donc $Valeur_{Opt}(M,n-1) \leq Valeur_{Opt}(M,n)$. Sinon (si $k_n > 0$), $(k_1,\cdots,k_{n-1},k_n-1)$ est une solution valide pour $(M-x_n,\mathscr{S}_n)$ et donc $Valeur_{Opt}(M-x_n,n) \leq Valeur_{Opt}(M,n) - 1$.

Ainsi, $\min\{Valeur_{Opt}(M,n-1); 1 + Valeur_{Opt}(M-x_n,n)\} \leq Valeur_{Opt}(M,n)$.

(pour la dernière étape, on utilise le fait que si $(a \leq c$ ou $b \leq c)$ alors $\min\{a,b\} \leq c$).

Rendu de Monnaie : Algorithme Optimal 1

Pour tout $1 \leq m \leq M$ et $1 \leq n' \leq n$, calculons récursivement $Valeur_{Opt}(m, n')$ en nous servant du théorème précédent pour calculer un terme en fonction des précédents.

```
#let RenduMonnaie m systeme =
        let n = vect_length systeme in

        let res_value = make_vect (m+1) [||] in
        for i=0 to m do
                res_value.(i) <- make_vect n 0;
        done;

        let res_vector = make_vect (m+1) [||] in
        for i=0 to m do
                res_vector.(i) <- make_vect n [||];
                for j=0 to (n-1) do
                        res_vector.(i).(j) <- make_vect n 0;
                done;
        done;

        for i=1 to m do
                res_value.(i).(0) <- i;
                res_vector.(i).(0).(0) <-i;
        done;

        for i=2 to m do
                for j=1 to (n-1) do
                        if (i < systeme.(j)) || (res_value.(i).(j-1) <= (1+res_value.(i-systeme.(j)).(j)))
                        then
                                begin
                                res_value.(i).(j) <- res_value.(i).(j-1);
                                let vecteur = res_vector.(i).(j-1) in
                                res_vector.(i).(j) <- vecteur;
                                end
                        else
                                begin
                                res_value.(i).(j) <- 1+ res_value.(i-systeme.(j)).(j);
                                let vecteur = res_vector.(i-systeme.(j)).(j) in
                                res_vector.(i).(j) <- vecteur;
                                res_vector.(i).(j).(j) <- 1+res_vector.(i).(j).(j);
                                end
                done;
        done;
        res_value.(m).(n-1), res_vector.(m).(n-1);;
RenduMonnaie : int -> int vect -> int * int vect = <fun>
#let systeme = [|1;3;4|];;
systeme : int vect = [|1; 3; 4|]
#RenduMonnaie 6 systeme;;
- : int * int vect = 2, [|0; 2; 0|]
```

Création d'un tableau res_value qui contiendra les valeurs optimales :
res_value.(m).(n-1) = Valeur_opt(m,n)
(attention au décalage d'indice pour n)

Création d'un tableau res_vector qui contiendra les solutions optimales :
res_vector.(m).(n-1) = [lk1,...knl] tel que sum(k_i)= Valeur_opt(m,n)
(attention au décalage d'indice pour n)

Initialisation :
mise à jour de Valeur_opt(i,1)=i pour tout i<=m

calcul de Valeur_opt(i,j) (et d'un vecteur correspondant) pour tout 2<=i<=m et 1<=j<=n-1
en utilisant les valeurs précédentes (dans l'ordre lexicographique) déjà calculées

Cas où Valeur_opt(i,j)=Valeur_opt(i,j-1)

Cas où Valeur_opt(i,j)=1+Valeur_opt(i-xj,j)

Terminaison : double boucles imbriquées

Correction : preuve par récurrence sur (i, j) (en se servant du théorème précédent) qu'après l'itération (i, j), $res\_value.(i).(j) = Valeur_{Opt}(i, j+1)$.

Complexité : double boucles imbriquées $O(nM)$.

Outline

Programmation dynamique

Programmation dynamique (def. de Wikipedia)

Méthode algorithmique pour résoudre des problèmes d'optimisation. Elle consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant (mémoïsation) les résultats intermédiaires.

Remarque 1 : "programmation" signifie ici "planification" ou "ordonnancement" (pas du tout "implémentation") : la programmation dynamique cherche à organiser les calculs d'une façon "efficace".

Remarque 2 : la technique de "Diviser pour régner" fait partie des méthodes par programmation dynamique. Une différence est que dans "Diviser pour régner", les sous-problèmes sont résolus indépendamment les uns des autres.
La programmation dynamique présente l'avantage de ne résoudre qu'une seule fois chaque sous-problème, ce qui n'est pas forcément le cas dans "Diviser pour régner". Par exemple, pour trier le tableau $[|4;3;5;7;13;2;4;3|]$, l'algorithme tri-fusion triera deux fois le tableau $[|4;3|]$.

Exemple : Dans le problème du Rendu de Monnaie, pour calculer $Valeur_{Opt}(M,n)$, nous avons calculé $Valeur_{Opt}(m,n')$ pour tout $m < M$ et $n' \leq n$ (et pour $m = M$ et tout $n' < n$) et déterminé $Valeur_{Opt}(M,n)$ à partir des valeurs préalablement calculées et stoquées. Notez que chaque sous-problème $Valeur_{Opt}(m,n')$ n'est calculé qu'une fois.

Outline

Rendu de Monnaie : Algorithme Optimal 2

Soit $M, n \in \mathbb{N}^*$ et $\mathscr{S}_n = (x_1 = 1 < x_2, \cdots < x_n) \in \mathbb{N}^n$ une instance du problème.
On note $Valeur_{Opt}(M,n) = \min_{(k_1,\cdots,k_n)\in\mathbb{N}^n}\{\sum_{1\leq i\leq n} k_i \mid \sum_{1\leq i\leq n} k_i x_i = M\}$ la valeur (nombre de pièces) d'une solution optimale pour l'instance $(M, \mathscr{S}_n)$.

Théorème :

- $Valeur_{Opt}(0,n) = 0$;
- $\forall M > 0,\ Valeur_{Opt}(M,n) = 1 + \min_{1\leq i\leq n,\ M-x_i\geq 0} Valeur_{Opt}(M - x_i, n)$

Preuve 1 : Application récursive de la preuve du Slide 11.

Preuve 2 : Le 1^{er} point est évident, donc supposons que $M > 0$. Le 2^{nd} point se prouve par double inégalité.

Toute solution $(k_1, \cdots, k_n)$ optimale pour (M,n) (en particulier, $\sum_{1\leq i<n} k_i x_i = M$) telle que $k_j > 0$ (pour $1 \leq j \leq n$ avec $M \geq x_j$) , donne une solution valide $(k_1, \cdots, k_{j-1}, k_j - 1, k_{j+1}, \cdots, k_{n-1})$ pour $(M - x_j, n)$. Donc $Valeur_{Opt}(M,n) \geq 1 + \min_{1\leq j\leq n,\ M-x_j\geq 0} Valeur_{Opt}(M - x_j, n)$.

Par ailleurs, pour tout $1 \leq j \leq n$ tel que $M - x_j \geq 0$, toute solution $(k_1^j, \cdots, k_n^j)$ valide pour $(M - x_j, n)$ (telle que $\sum_{1\leq i\leq n} k_i^j x_i = M - x_j$) permet d'obtenir une solution $(k_1^j, \cdots, k_{j-1}^j, k_j^j + 1, k_{j+1}^j, \cdots, k_{n-1}^j)$ valide pour (M,n) en ajoutant une pièce de valeur x_j. Donc $Valeur_{Opt}(M,n) \leq 1 + \min_{1\leq j\leq n,\ M-x_j\geq 0} Valeur_{Opt}(M - x_j, n)$.

Rendu de Monnaie : Algorithme Optimal 2

Pour tout $1 \leq m \leq M$ et $n \in \mathbb{N}$, calculons récursivement $Valeur_{Opt}(m,n)$ en nous servant du théorème précédent: $Valeur_{Opt}(M,n) = 1 + \min_{1 \leq i \leq n,\ M - x_i \geq 0} Valeur_{Opt}(M - x_i, n)$
pour calculer un terme en fonction des précédents $Valeur_{Opt}(m',n)$ pour $m' < m$.

```
#let RenduMonnaieOpt m systeme =
        let n = vect_length systeme in
        let sol = make_vect (m+1) (m+1) in
        sol.(0) <- 0;
        for k=1 to m do
                let max = ref 0 in
                while ((!max)<n) && ((k- systeme.(!max))>=0) do
                        max:= !max +1;
                done;
                for j=0 to (!max-1) do
                        sol.(k) <- min sol.(k) (1+sol.(k-systeme.(j)));
                done;
        done;
        sol.(m);;
RenduMonnaieOpt : int -> int vect -> int = <fun>
#let systeme = [|1;3;4|];;
systeme : int vect = [|1; 3; 4|]
#RenduMonnaieOpt 6 systeme;;
- : int = 2
```

Terminaison : double boucles imbriquées

Correction : preuve par récurrence sur (k,j) (en se servant du théorème précédent) qu'après l'itération (k,j), $sol.(k) = 1 + \min_{1 \leq i \leq j,\ k - x_i \geq 0} Valeur_{Opt}(k - x_i, n)$.

Complexité : double boucles imbriquées $O(nM)$.

Cet algorithme a l'avantage, par rapport à l'algorithme précédent, de n'utiliser qu'un tableau de taille $M+1$ (alors que l'algorithme précédent utilisait une matrice $(M+1)*n$). Sa complexité en espace est donc meilleure, pour une complexité en temps similaire.

(il semble plus simple à écrire que l'algorithme précédent, mais notons que nous ne gardons que le nombre minimum de pièces et non une combinaison optimale de pièces)

Outline

Problème du Sac-à-Dos avec remise

Vous arrivez dans une caverne avec votre sac-à-dos. Là vous trouvez un trésor qui contient une infinité d'objets de valeur 1 € qui pèsent 1 kilo chacun, une infinité d'objets de valeur 53 € qui pèsent 51 kilos chacun et une infinité d'objets de valeur 100 € qui pèsent 100 kilos chacun.
Votre sac-à-dos ne supporte qu'un poids de 102 **kilos**. **Quels objets prenez vous ?** (si vous voulez vendre ce que vous emportez au plus grand prix ?)

Problème du Sac-à-Dos avec remise

Vous arrivez dans une caverne avec votre sac-à-dos. Là vous trouvez un trésor qui contient une infinité d'objets de valeur 1 € qui pèsent 1 kilo chacun, une infinité d'objets de valeur 53 € qui pèsent 51 kilos chacun et une infinité d'objets de valeur 100 € qui pèsent 100 kilos chacun. **Votre sac-à-dos ne supporte qu'un poids de** 102 **kilos**. **Quels objets prenez vous ?** (si vous voulez vendre ce que vous emportez au plus grand prix ?)

Algorithme glouton : Tant qu'il "rentre" dans votre sac, vous prenez l'objet de plus grande valeur. Ce faisant, vous prenez un objet de valeur 100 € (et de poids 100 kilos) et 2 objets de valeur 1 €. Soit 102 € au total.

Une solution optimale : 2 objets de 53 € (et de poids 51 kilos chacun). Soit 106 € au total.

j'espère que cet exemple vous rappelle quelque chose...

Problème du Sac-à-Dos avec remise

Vous arrivez dans une caverne avec votre sac-à-dos. Là vous trouvez un trésor qui contient une infinité d'objets de valeur 1 € qui pèsent 1 kilo chacun, une infinité d'objets de valeur 53 € qui pèsent 51 kilos chacun et une infinité d'objets de valeur 100 € qui pèsent 100 kilos chacun. **Votre sac-à-dos ne supporte qu'un poids de** 102 **kilos**. **Quels objets prenez vous ?** (si vous voulez vendre ce que vous emportez au plus grand prix ?)

Algorithme glouton : Tant qu'il "rentre" dans votre sac, vous prenez l'objet de plus grande valeur. Ce faisant, vous prenez un objet de valeur 100 € (et de poids 100 kilos) et 2 objets de valeur 1 €. Soit 102 € au total.

Une solution optimale : 2 objets de 53 € (et de poids 51 kilos chacun). Soit 106 € au total.

j'espère que cet exemple vous rappelle quelque chose...

Généralisation du problème :
Vous arrivez dans une caverne avec votre sac-à-dos qui peut contenir un maximum de M kilos. Là vous trouvez un trésor qui contient, pour tout $1 \leq i \leq n$, une infinité d'objets de type i, de poids p_i et de valeur v_i. **Quels objets prenez vous pour maximiser votre gain tout en satisfaisant la contenance de votre sac-à-dos ?**

Problème du Sac-à-Dos avec remise

Étant donnés $M \in \mathbb{R}^+$ et $((p_1, v_1), \cdots, (p_n, v_n)) \in (\mathbb{R}^+)^n$
Calculer $(k_1, \cdots, k_n) \in \mathbb{N}^r$ tels que $\sum_{1 \leq i \leq n} k_i v_i$ soit maximum sous réserve que $\sum_{1 \leq i \leq n} k_i p_i \leq M$.

Théorème : $Valeur_{Opt}(M,n) = \max_{(k_1,\cdots,k_n) \in \mathbb{N}^n} \{ \sum_{1 \leq i \leq n} k_i v_i \mid \sum_{1 \leq i \leq n} k_i p_i \leq M \}$

- $Valeur_{Opt}(M,1) = \lfloor M/p_1 \rfloor * v_1$;
- $\forall n > 1$, si $p_n > M$ alors $Valeur_{Opt}(M,n) = Valeur_{Opt}(M,n-1)$;
- sinon $Valeur_{Opt}(M,n) = \max\{Valeur_{Opt}(M,n-1); v_n + Valeur_{Opt}(M-p_n,n)\}$

```
#let SacADosRemise m objects =
        let n = vect_length objects in

        let res_value =  make_vect (m+1) [||] in
        for i=0 to m do
                res_value.(i) <- make_vect n 0;
        done;

        let res_vector = make_vect (m+1) [||] in
        for i=0 to m do
                res_vector.(i) <- make_vect n [||];
                for j=0 to (n-1) do
                        res_vector.(i).(j) <- make_vect n 0;
                done;
        done;

        for i=1 to m do
                let k = int_of_float(float_of_int(i)/.float_of_int(objects.(0).(0))) in
                res_value.(i).(0) <- k * objects.(0).(1);
                res_vector.(i).(0).(0) <- k;
        done;

        for i=2 to m do
                for j=1 to (n-1) do
                        if (i < objects.(j).(0)) || (res_value.(i).(j-1) > objects.(j).(1) + res_value.(i-objects.(j).(0)).(j)))
                        then
                                begin
                                res_value.(i).(j) <- res_value.(i).(j-1);
                                let vecteur = res_vector.(i).(j-1) in
                                res_vector.(i).(j) <- vecteur;
                                end
                        else
                                begin
                                res_value.(i).(j) <- objects.(j).(1) + res_value.(i-objects.(j).(0)).(j);
                                let vecteur = res_vector.(i-objects.(j).(0)).(j) in
                                res_vector.(i).(j) <-(j) <- 1+ res_vector.(i).(j).(j);
                                end
                done;
        done;
        res_value.(m).(n-1), res_vector.(m).(n-1);;
SacADosRemise : int -> int vect vect -> int * int vect = <fun>
```

Nous mettons principalement en évidence les différences avec **RenduMonnaie**

objects.(j).(0)=p_{j+1} (**poids** de l'objet j+1)
objects.(j).(1)=v_{j+1} (**valeur** de l'objet j+1)

la principale différence avec le problème de Rendu de Monnaie est, qu'ici, on veut **maximiser** le gain

Exercice : Adaptez l'algorithme Optimal 2 pour le Rendu de Monnaie à ce problème.

Problème du Sac-à-Dos SANS remise

Maintenant, lorsque vous prenez un objet, il n'est plus disponible (il n'est pas "remis en jeu").
Vous ne pouvez prendre qu'AU PLUS UN objet de chaque type.

Problème du Sac-à-Dos SANS remise

Étant donnés $M \in \mathbb{R}^+$ et $((p_1, v_1), \cdots, (p_n, v_n)) \in (\mathbb{R}^+)^n$
Calculer $(k_1, \cdots, k_r) \in \{0,1\}^n$ tels que $\sum_{1 \leq i \leq n} k_i v_i$ soit maximum sous réserve que $\sum_{1 \leq i \leq n} k_i p_i \leq M$.

Théorème : $Valeur_{Opt}(M,n) = \max_{(k_1,\cdots,k_n) \in \{0,1\}^n} \{ \sum_{1 \leq i \leq n} k_i v_i \mid \sum_{1 \leq i \leq n} k_i p_i \leq M \}$

- $Valeur_{Opt}(M,1) = \min\{\lfloor M/p_1 \rfloor; 1\} * v_1$;
- $\forall n > 1$, si $p_n > M$ alors $Valeur_{Opt}(M,n) = Valeur_{Opt}(M,n-1)$;
- sinon $Valeur_{Opt}(M,n) = \max\{Valeur_{Opt}(M,n-1); v_n + Valeur_{Opt}(M-p_n, n-1)\}$

Exercice 1 : Adaptez le(s) algorithme(s) précédent(s) à ce problème.

Remarquer que l'Algorithme Optimal 2 ne convient pas

Exercice 2 : Prouvez que ce problème peut être résolu en temps $O(Mn)$.

Remarque : La "taille de l'entrée" (M et $((p_1, v_1), \cdots, (p_n, v_n)) \in (\mathbb{R}^+)^n$) est $\log M + n \cdot \max_{1 \leq i \leq n}\{\log p_i, \log v_i\}$ (codée en binaire).
Donc, les temps de résolution des algorithmes décrits précédemment $O(Mn)$ sont exponentiels en la taille de l'entrée codée en binaire.

On ne sait pas s'il existe un algorithme résolvant les problèmes précédents en temps polynomial en la taille des instances codée en binaire (ces problèmes sont faiblement NP-difficiles).

Problème du Sac-à-Dos SANS remise

Maintenant, lorsque vous prenez un objet, il n'est plus disponible (il n'est pas "remis en jeu").
Vous ne pouvez prendre qu'AU PLUS UN objet de chaque type.

Problème du Sac-à-Dos SANS remise

Étant donnés $M \in \mathbb{R}^+$ et $((p_1, v_1), \cdots, (p_n, v_n)) \in (\mathbb{R}^+)^n$
Calculer $(k_1, \cdots, k_r) \in \{0,1\}^n$ tels que $\sum_{1 \leq i \leq n} k_i v_i$ soit maximum sous réserve que $\sum_{1 \leq i \leq n} k_i p_i \leq M$.

Théorème : $Valeur_{Opt}(M,n) = \max_{(k_1,\cdots,k_n) \in \{0,1\}^n} \{ \sum_{1 \leq i \leq n} k_i v_i \mid \sum_{1 \leq i \leq n} k_i p_i \leq M \}$

- $Valeur_{Opt}(M,1) = \min\{\lfloor M/p_1 \rfloor; 1\} * v_1$;
- $\forall n > 1$, si $p_n > M$ alors $Valeur_{Opt}(M,n) = Valeur_{Opt}(M,n-1)$;
- sinon $Valeur_{Opt}(M,n) = \max\{Valeur_{Opt}(M,n-1); v_n + Valeur_{Opt}(M-p_n, n-1)\}$

Exercice 1 : Adaptez le(s) algorithme(s) précédent(s) à ce problème.
Remarquer que l'Algorithme Optimal 2 ne convient pas

Exercice 2 : Prouvez que ce problème peut être résolu en temps $O(Mn)$.

Remarque : La "taille de l'entrée" (M et $((p_1, v_1), \cdots, (p_n, v_n)) \in (\mathbb{R}^+)^n$) est $\log M + n \cdot \max_{1 \leq i \leq n}\{\log p_i, \log v_i\}$ (codée en binaire).
Donc, les temps de résolution des algorithmes décrits précédemment $O(Mn)$ sont exponentiels en la taille de l'entrée codée en binaire.

On ne sait pas s'il existe un algorithme résolvant les problèmes précédents en temps polynomial en la taille des instances codée en binaire (ces problèmes sont faiblement NP-difficiles).