Une histoire de monnaie

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/07/

1 Approche gloutonne

Un système monétaire est une suite (finie) strictement décroissante (d_i) d'entiers (naturels non-nuls). Chaque d_i est une espèce. Par exemple, le système de l'Euro est $\langle 500, 200, 100, 50, 20, 10, 5, 2, 1\rangle$ et l'ancien système britannique est $\langle 240, 60, 30, 24, 12, 6, 3, 1\rangle$. On s'intéresse au problème du paiement exact d'un montant quelconque dans un système donné.

1. Une première question est celle de la possibilité de payer. Trouver une condition nécessaire et suffisante pour qu'un système monétaire permette de payer n'importe quel montant.
2. Une seconde question est de payer avec le minimum d'espèces (paiement optimal).
   Décrire une approche gloutonne du paiement optimal.
3. Programmer l'approche gloutonne.

   ```
   static int [] glouton(int [] d, int m) ;
   ```

   Pour un système monétaire $\langle d_0, d_1, \ldots d_{k-1}\rangle$ et un montant m, La méthode (statique) `glouton` renvoie un tableau d'entiers t tel que.

   $$\sum_{i=0}^{k-1} t_i \times d_i = m$$

   Attention, on se contraindra à atteindre une complexité en $O(k)$.
   Au passage et plus précisément, les paiements sont représentés par des tableaux t de taille k. Le cardinal d'un paiement est tout bêtement le nombre d'espèces concrètes dans un paiement, c'est-à-dire la somme des t_i. Un paiement de m est optimal quand son cardinal est minimal parmi ceux des paiements de m.
4. On va montrer que la stratégie gloutonne est optimale dans le cas du système européen.
   Soit un paiement donné $T = \langle t_0, t_1, \ldots, t_{k-1}\rangle$, on note $\mathcal{V}_i(T)$ le montant obtenu en annulant tous les t_j de 0 à $i-1$.

   $$\mathcal{V}_i(T) = \sum_{j=i}^{k-1} t_j \times d_j$$

   (On constate que si T est un paiement optimal, T_i obtenu en annulant tous les t_j de 0 à $i-1$ est également un paiement optimal.)

   (a) Montrer que, dans système européen et pour tout paiement T optimal on a :

   $$\mathcal{V}_i(T) < d_{i-1}$$

   Le plus simple me semble de compléter le tableau suivant, qui donne pour chaque entier i entre 9 et 2, le plus grand $\mathcal{V}_i(T)$ possible (noté b_i).

i	8	7	6	5	4	3	2	1
d_{i-1}	2	5	10	20	50	100	200	500
b_i	1	4						

   La borne b_8 s'explique en considérant que T optimal ne peut pas contenir plus d'une pièce d'un euro (car on peut remplacer deux pièces d'un euro par une pièce de deux euros). La borne b_7 s'explique de façon à peine plus complexe, T optimale ne peut pas contenir 3 pièces de deux euros ou plus (par le remplacement $2+2+2 \rightarrow 5+1$), et deux pièces de deux euros ne peuvent être accompagnées d'une pièce d'un euro (car $1+2+2=5$).

(b) Monter que la condition ci-dessus entraîne l'optimalité de la stratégie gloutonne.

5. Montrer que la stratégie gloutonne n'est pas optimale dans le cas du système britannique.

2 Recherche exhaustive

À cause de l'existence du système britannique, le paiement optimal ne peut pas reposer sur l'algorithme glouton. On cherche une solution générale.

1. Décrire une technique inductive pour engendrer l'ensemble des paiements d'une somme donnée m dans un système donné (d_i).
2. Écrire une méthode `tous`, qui affiche tous les paiements possibles.

```
static void tous(int [] d, int m)
```

On suppose donnée une méthode d'affichage `dump` que voici :

```
static void dump(PrintStream out, int [] system, int [] t) {
  for (int i = 0 ; i < t.length ; i++) {
    int d = system[i] ;
    for (int j = 0 ; j < t[i] ; j++)
      out.print("␣" + d) ;
  }
  out.println() ;
}
```

Dans ce type de programmes (comme par exemple dans le devoir maison numéro deux, ou le contrôle hors-classement) on essaie d'éviter d'allouer trop. Représentons un paiement par le tableau des t_i (un tableau `t`) de taille k (nombre d'espèces du système).
Il s'agit en gros de fabriquer tous les `t` possible. Une solution économique en espace utilisera une méthode `zyva`.

```
static void zyva(int [] d, int m, int i, int [] t)
```

Dans un appel à `zyva`, le tableau `t` n'est « rempli » que de 1 à $i-1$ et m est un prix à atteindre en utilisant seulement les espèces d'indice supérieur ou égal à i. Les arguments `t`, `i`, et `m` sont liés :

$$m + \sum_{j=0}^{i-1} t_j \times d_j = m_0$$

(Où m_0 est le prix initial.)
Ainsi `tous` s'écrit très simplement :

```
static void tous(int [] d, int m) {
  zyva(d, m, 0, new int[d.length]) ;
}
```

3. Transformer `tous` en `exhaustif`, qui renvoie une somme optimale.

```
static int [] exhaustif(int [] d, int m)
```

4. Il n'est pas besoin de parcourir tous les paiements possibles pour trouver un paiement optimal. En effet supposons connu un paiement de cardinal c_0. Pour un appel donné de `zyva`, on dispose d'un paiement partiel $\langle t_0, t_1, \ldots, t_{i-1}\rangle$ de cardinal c, et il reste à payer m à l'aide des espèces $\langle d_i, d_{i+1}, \ldots, d_{k-1}\rangle$. On ne peut certainement pas compléter t en un paiement optimal dans les deux cas suivants :
– Si c est déjà strictement supérieur à c_0.
– Si la somme maximale payable à l'aide de $c_0 - c$ espèces d_i est strictement inférieure à m.
On peut alors élaguer la recherche, c'est-à-dire renoncer à compléter le paiement partiel t. Modifier la recherche exhaustive pour profiter de ces remarques.

3 Paiement dynamique

La programmation dynamique revient à échanger du temps contre de l'espace. Sans plus tergiverser voici une solution au problème du paiement optimal qui emploie cette technique.

```
1  static int [] dynamiqueCard(int [] d, int m) {
2    int k = d.length ;
3    int [] card = new int [m+1] ;
4
5    card[0] = 0 ; // Inutile en Java
6    for (int n = 1 ; n <= m ; n++) {
7      int r = m+1 ;
8      int i ;
9      for (i = 0 ; d[i] > n ; i++)
10       ;
11     for (; i < k ; i++) {
12       int di = d[i] ;
13       if (card[n-di]+1 < r) {
14         r = card[n-di]+1 ;
15       }
16     }
17     card[n] = r ;
18   }
19   return card ;
20 }
```

La méthode `dynamiqueCard` renvoie un tableau d'entiers $c[0 \ldots m]$, tel que $c[n]$ est le cardinal d'un paiement optimal de n.

1. Expliquer le commentaire de la ligne 5.
2. Comment comprendre la boucle des lignes 9-10 ? Pourquoi termine-t-elle toujours ?
3. Montrer que tous les accès dans les tableaux sont légaux.
4. Donner le coût de la méthode `dynamiqueCard`.
5. Argumenter la correction de la méthode `dynamiqueCard`. On se contentera d'une définition inductive de $M(n)$, cardinal d'un paiement optimal de n.
6. Comment faire pour trouver un paiement optimal de m et non plus seulement son cardinal ?

Note

Toutes les solutions sont regroupées dans la classe `Monnaie` (disponible à partir de la version Web du cours).

Solutions des exercices

Exercice 1

1. La condition $d_{k-1} = 1$ est nécessaire et suffisante.
2. Choisir la plus grande espèce ne dépassant pas le montant (restant à) payer et recommencer jusqu'à plus soif.
3.
```java
static int [] glouton(int [] system, int m) {
  int [] r = new int[system.length] ;
  for (int i = 0 ;  m > 0 ; i++) {
    int d = system[i] ;
    r[i] = m/d ;
    m %= d ; // Pour m = m % d ;
  }
  return r ;
}
```

   On note le calcul de t_i espèce par espèce, à l'aide d'une division.
4. (a) Voici le tableau, puis les justifications.

i	8	7	6	5	4	3	2	1
d_{i-1}	2	5	10	20	50	100	200	500
b_i	1	4	9	19	49	99	199	499

   - Avec $\langle 5, 2, 1\rangle$. Un paiement T optimal contient au plus une pièce de 5 euros (par le doublement $5 + 5 = 10$). On a ensuite $\mathcal{V}_6(T) \leq 1 \times 5 + b_7 = 9$.
   - Avec $\langle 10, 5, 2, 1\rangle$ par le même argument de doublement.
   - Avec $\langle 20, 10, 5, 2, 1\rangle$, par $20+20+20 = 50+10$, T contient au plus deux billets de 20. On distingue alors trois petits sous-cas :
     - Si T contient effectivement deux billets de 20, alors T ne peut pas contenir de billet de 10 (car $20 + 20 + 10 = 50$), et donc $\mathcal{V}_4(T) \leq 40 + b_6 = 49$.
     - Si T contient un seul billet de 20, alors $\mathcal{V}_4(T) \leq 20 + b_5 = 39$.
     - Enfin, si T ne contient pas de billet de 20, alors on a évidemment $\mathcal{V}_4(T) \leq b_5$.

     Bref on a $\mathcal{V}_4(T) \leq 49$.
   - Par doublement $(50 + 50 = 100)$, on a $\mathcal{V}_3(T) \leq 50 + b_4 = 99$.
   - Par doublement $(100 + 100 = 200)$, on a $\mathcal{V}_2(T) \leq 100 + b_3 = 199$.
   - Le raisonnement est le même que pour 20 et donne la borne $\mathcal{V}_1(T) \leq 400 + b_3 = 499$.

   (b) La condition $\mathcal{V}_i(T) < d_{i-1}$ est en fait assez forte, elle entraîne l'unicité du paiement optimal qui est le paiement glouton. En effet on a, dans tous les cas, les égalités. $m = \mathcal{V}_0(T) = t_0 \times d_0 + \mathcal{V}_1(T)$, $\mathcal{V}_1(T) = t_1 \times d_1 + \mathcal{V}_2(T)$, …, $\mathcal{V}_i(T) = t_i \times d_i + \mathcal{V}_{i+1}(T)$, … Mais ici, en raison des inégalités, $\mathcal{V}_{i+1}(T) < d_i$, les t_i sont complètement déterminés, par unicité du quotient (et du reste) de la division euclidienne de $\mathcal{V}_i(T)$ par d_i. Enfin, l'unique paiement optimal est bien le paiement glouton, tel que calculé à la question précédente.
5. On recherche un contre-exemple, il s'agit de 48. En effet un paiement possible est alors $24 + 24$, tandis que le paiement glouton est $30 + 12 + 6$. L'étude du système européen conduit à considérer le sous-système $\langle 30, 24, 12, 6, 3, 1\rangle$ (30 est la première espèce qui n'est pas le double de l'espèce précédente). Autrement dit, on a le tableau partiel :

i	8	7	6	5	4	3	2	
d_{i-1}	3	6	12	24	30	60	120	240
b_i	2	5	11	23	?			

On cherche donc la petite bête dans les paiements (optimaux) qui contiennent l'espèce 24. Le contre-exemple vient dès le premier essai sérieux : $24 + 24$.

Avant de critiquer les britanniques et la base 12, remarquons qu'un problème similaire se pose si on ajoute un billet de 25 au système européen.

Exercice 2

1. Les notations sont assez lourdes alors autant décrire.
   - Si $k = 1$ alors il n'y a qu'un paiement possible, m fois l'espèce 1.
   - Sinon, un paiement qualconque contient $0, 1, \ldots$ ou m/d_1 (division euclidienne) espèces d_1, le reste des t_i étant, dans chacun de ces cas, un paiement quelconque de m, $m-d_1$, $m-2d_1$, $\ldots m-(m/d_1)d_1$ dans le système $\langle d_2, \ldots, d_l \rangle$.
2. Comme souvent dans ce genre de programmes, il faut tenter d'utiliser le moins de mémoire possible, l'idée est ici de construire un paiement progressivement de la gauche vers la droite, dans un tableau passé en argument.

```
private static void zyva (int d[], int m, int i, int [] r) {
  if (d[i] == 1) {
    r[i] = m ;
    dump(System.out, d, r) ;
  } else {
    int di = d[i] ;
    int max = m/d[i] ;
    for (int j = 0 ; j <= max ; j++) {
      r[i] = j ;
      zyva(d, m-j*di, i+1, r) ;
    }
  }
}

static void tous(int [] system, int m) {
  zyva(system, m, 0, new int[system.length]) ;
}
```

3. On peut à la place de l'impression (`dump`), comparer le paiement engendré r à un paiement minimal courant. Le paiement minimal courant est idéalement rangé dans une variable statique. Il semble assez logique de calculer le cardinal des paiements en même temps que les paiements eux-mêmes. Il vient :

```
private static int minCard ;
private static int [] minR  ;

private static void check(int card, int [] r) {
  if (card < minCard) {
    minCard = card ;
    for (int i = 0 ; i < r.length ; i++)
      minR[i] = r[i] ;
  }
}

private static void look(int [] d, int m, int i, int card, int [] r) {
  int di = d[i] ;
  if (di == 1) {
    r[i] = m ;
```

```
        check(card + m, r) ;
      } else {
        int max = m/di ;
        for (int j = 0 ; j <= max ; j++) {
          r[i] = j ;
         look(d, m-j*di, i+1, card+j,r) ;
        }
      }
    }

    static int [] exhaustif(int [] system, int m) {
      int len = system.length ;
      minR = new int [len] ;
      minCard = Integer.MAX_VALUE ;
      look(system, m, 0, 0, new int [len]) ;
      return minR ;
    }
```

Noter que l'on peut remplacer la ligne :

```
         look(d, m-j*di, i+1, card+j,r) ;
```

par :

```
         look(d, m, i+1, card+j,r) ;
         m -= di ; // Pour m = m - di
```

On fait ainsi disparaître des multiplications (coûteuses en temps calcul), mais c'est du chipotage.
Le coût exact n'est pas facile à évaluer, mais une rapide étude expérimentale montre qu'il y a problème (payer une forte somme dans le système européen par exemple).

4. En s'inspirant des remarques de l'énoncé, on transforme simplement `look` ainsi :

```
  private static void lookOpt(int [] d, int m, int i, int card, int [] r) {
      int di = d[i] ;
      if (card < minCard && di*(minCard-card) >= m) {
        if (di == 1) {
          r[i] = m ;
          check(card + m, r) ;
        } else {
          int max = m/di ;
          for (int j = max ; j >= 0 ; j--) {
            r[i] = j ;
            lookOpt(d, m-j*di, i+1, card+j,r) ;
          }
        }
      }
    }

    static int [] exhaustifOpt(int [] system, int m) {
      int len = system.length ;
      minR = glouton(system, m) ;
      minCard = card(minR) ;
      lookOpt(system, m, 0, 0, new int [len]) ;
      return minR ;
    }
```

```
static int card(int [] t) {
  int r = 0 ;
  for (int i = 0 ; i < t.length ; i++)
    r += t[i] ;
  return r ;
}
```

On remarque que `minR` et `minCard` sont initialisées avec la solution gloutonne calculée indépendamment.

L'effet de l'élagage semble assez remarquable dans le cas de systèmes monétaires raisonables, mais il est insuffisant dans le cas de systèmes comportant beaucoup d'espèces élevées dont les valeurs sont proches (genre $\langle 200, 199, 198, \ldots, 180, 1\rangle$).

Exercice 3

1. C'est le coup classique : les cases du tableau `card` sont initialisées à zéro (par la création **new int**`[m+1]`).
2. Il s'agit d'une boucle dont le corps est vide. L'effet est de calculer i_0 minimal tel que $d_{i_0} \leq n$. La boucle termine fatalement, soit parce qu'il existe un d_i inférieur ou égal à n, soit parce que i dépasse $k-1$ auquel cas il se produit une erreur.
3. Voyons accès par accès.
   - Ligne 5. On doit supposer $m \geq 0$.
   - Ligne 9. On a $d_{k-1} = 1$ et $n \geq 1$. Par conséquent $n \leq d_{k-1}$ et donc la boucle 9-10 se termine avant que i ne dépasse $k-1$.
   - Ligne 12. On a ici nécessairement $i < k$ (condition de la boucle sur la ligne du dessus).
   - Lignes 13 et 14. Les espèces du système étant rangées en ordre décroissant, on a ici $d_i \leq d_{i_0} \leq n$ et donc $0 \leq n - d_{i_0}$. Par ailleurs $n \leq m$ (ligne 6) et donc $n - d_{i_0} \leq m$ (le tableau `card` est de taille $m+1$).
   - Ligne 19. Sans commentaire.
4. Le coût est en $O(m \times k)$, m tours de la boucle 6-18 qui contient essentiellement une boucle à k tours.
5. La correction repose sur la formule :

$$M(0) = 0, \quad M(n) = 1 + \min_{d_i \leq s} M(n - d_i)$$

6. Il a deux façons de procéder, la première repose sur l'enregistrement, dans un tableau idoine, de l'espèce d_i pour laquelle on a $M(n) = M(n - d_i) + 1$.

```
  static int [] dynamiqueCard(int [] d, int m) {
    int k = d.length ;
    int [] card = new int [m+1] ;
••• int [] esp = new int [m+1] ;

    card[0] = 0 ; // Inutile en Java
    for (int n = 1 ; n <= m ; n++) {
      int r = m+1 ;
      int i ;
      for (i = 0 ; d[i] > n ; i++)
        ;
      for (; i < k ; i++) {
        int di = d[i] ;
        if (card[n-di]+1 < r) {
```

```
•••          r = card[n-di]+1 ;
             esp[n] = i ;
           }
         }
         card[n] = r ;
       }
       return card ;
     }
```

Une fois rempli le tableau `esp`, on reconstitue facilement un paiement optimal de m :

```
int r = new int[k] ;

while (m > 0) {
  int i = esp[m] ;
  r[i]++ ; // Pour r[i] = r[i]+1 ;
  m -= d[i] ;
}
```

On peut en fait, comme souvent en programmation dynamique, se passer du tableau `esp` en reconstituant les éléments intéressants à partir du tableau des cardinaux minimaux. Sur ce principe, voici une méthode qui renvoie un paiement optimal.

```
static int [] dynamique(int [] d, int m) {
  int k = d.length ;
  int [] card = dynamiqueCard(d, m) ;
  int [] r = new int[k] ;
  while (m > 0) {
    int c = card[m] ;
    int i ;
    for (i=0 ; d[i] > m ; i++)
      ;
    for ( ; ; i++) {
      int di = d[i] ;
      if (card[m-di] == c-1) {
        m -= di ; r[i]++ ;
        break ;
      }
    }
  }
  return r ;
}
```

On notera l'économie d'un tableau de taille environ m pour un prix somme toute modique en temps de calcul.

La boucle `for ( ; ; i++)...` est remarquable, car elle n'a pas de condition. Dans ce cas, la condition est vraie, de sorte que la tournure idiomatique pour la boucle « infinie » est :

```
for ( ; ; ) {
  ...
}
```

Bon tout ça est très culturel, mais en fait rien ne nous empêche de parcourir le tableau `d` de la droite vers la gauche. Le code devient alors un peu plus simple.

```
static int [] dynamique(int [] d, int m) {
  int k = d.length ;
```

```java
  int [] card = dynamiqueCard(d, m) ;
  int [] r = new int[k] ;
  while (m > 0) {
    int c = card[m] ;
    for (int i = k-1 ; ; i--) { // Pas de condition
      int di = d[i] ;
      if (card[m-di] == c-1) { // Car di tel que card[m-di] == c-1 existe
        m -= di ; r[i]++ ;
        break ;
      }
    }
  }
  return r ;
}
```