Algorithmique I
Année 2007-2008

A. Benoit, B. Depardon
JF. Pineau, C. Rezvoy

TD n°5 - Partiel 2006/2007

1 Un problème de mémoire

On souhaite enregistrer sur une mémoire de taille L un groupe de fichiers $P = (P_1, \ldots, P_n)$. Chaque fichier P_i nécessite une place a_i. Supposons que $\sum a_i > L$: on ne peut pas enregistrer tous les fichiers. Il s'agit donc de choisir le sous ensemble Q des fichiers à enregistrer.

On pourrait souhaiter le sous-ensemble qui contient le plus grand nombre de fichiers. Un algorithme glouton pour ce problème pourrait par exemple ranger les fichiers par ordre croissant des a_i.

Supposons que les P_i soient ordonnés par taille ($a_1 \leq \ldots \leq a_n$).

Question 1.1 Écrivez un algorithme (en pseudo-code) pour la stratégie présentée ci-dessus.

Cet algorithme doit renvoyer un tableau booléen S tel que $S[i] = 1$ si P_i est dans Q et $S[i] = 0$ sinon.

Quelle est sa complexité en nombre de comparaisons et en nombre d'opérations arithmétiques ?

Solution : *On suppose que les P_i sont ordonnés par taille croissante ($a_1 \leq \cdots \leq a_n$).*

Algorithme 1 Plus grand nombre de fichiers

```
ENTRÉES: a_1, ..., a_n, L
SORTIES: S
  S ← []
  taille ← 0
  pour i de 1 à n faire
    taille_tmp ← taille + a_i
    si taille_tmp ≤ L alors
      taille ← taille_tmp
      S[i] ← 1
    sinon
      S[i] ← 0
  return S
```

Il nous faut comparer à chaque fois la nouvelle taille ($taille + a_i$) avec L pour vérifier si on peut ajouter le fichier, il faut donc n comparaisons. Il nous faut au plus n additions (calcul de $taille_{tmp}$). □

Question 1.2 Montrer que cette stratégie donne toujours un sous-ensemble Q maximal tel que

$$\sum_{P_i \in Q} a_i \leq L$$

Solution : *Soit un sous ensemble Q quelconque de cardinal maximal. Comparons le à la solution gloutonne G. Soit a_j le plus petit élément présent dans G et non dans Q, et a_k un élément dans Q n'étant pas dans G (comme Q est de cardinalité maximal, on est sur de trouver un tel a_k pour chaque a_j à considérer). L'ensemble Q' obtenu en échangeant dans Q a_k par a_j est toujours un ensemble valide (puisque la taille mémoire est inférieure à celle de Q), et de même cardinalité que Q. Par construction, on peut ainsi construire un ensemble G' à partir de Q, optimal. G' ne peut enfin pas avoir plus d'élément que G, par construction de G, donc la solution glouton est optimale.* □

Question 1.3 Soit Q le sous-ensemble obtenu. À quel point le quotient d'utilisation $\frac{\sum_{P_i \in Q} a_i}{L}$ peut-il être petit ?

Solution : *Si les a_i sont quelconques, ils peuvent tous être plus grand que L. Le quotient est alors 0. Dans le cas plus intéressant où $a_i \leq L$, le quotient est au pire $\frac{a_1}{L}$ (exemple avec $a_1 = 1$ et $a_2 = L$.* □

Supposons maintenant que l'on souhaite enregistrer le sous-ensemble Q de P qui maximise ce quotient d'utilisation, c'est-à-dire celui qui remplit le plus de disque. Une approche *gloutonne* consisterait à considérer les fichiers dans l'ordre décroissant des a_i et, s'il reste assez d'espace pour P_i, on l'ajoute à Q.

Question 1.4 On suppose toujours les P_i ordonnés par tailles croissantes. Écrivez un algorithme pour cette nouvelle stratégie.

Solution : *C'est le même que précédemment sauf qu'on part de la fin de la liste*

Algorithme 2 Plus grand nombre de fichiers

Entrées: $a_1, \ldots, a_n, L$
Sorties: S

```
S ← []
taille ← 0
pour i de n à 1 faire
    taille_tmp ← taille + a_i
    si taille_tmp ≤ L alors
        taille ← taille_tmp
        S[i] ← 1
    sinon
        S[i] ← 0
return S
```

□

Question 1.5 Montrer que cette nouvelle stratégie ne donne pas nécessairement un sous-ensemble qui maximise le quotient d'utilisation. À quel point ce quotient peut-il être petit ? Prouvez-le.

Solution : *Contre-exemple : $a_1 = \frac{L}{2} + 1$, et $a_2 = a_3 = L$. L'algorithme glouton retournera une solution ayant comme quotient $1/2$, alors que le quotient de la solution optimale est 1.*

Le minimum du coefficient d'utilisation est $1/2$ *si* $a_i \leq L$ *(0 sinon). En effet, soit* P *le poids de la liste de l'algorithme glouton. Par définition, comme la somme des poids est plus grande que* L*,* P *est strictement plus grand que* $L/2$ *(soit* $a_n > L/2$*, soit la somme des éléments de plus grand poids dépasse* $L/2$*). Donc le quotient est plus grand que* $\frac{\frac{L}{2}+1}{L} = \frac{1}{2} + \frac{1}{L} \simeq \frac{1}{2}$*. Ce quotient est atteint pour l'exemple précédent.* □

2 Partitionnement d'un carré

On se donne n réels positifs $s_1, \ldots, s_n$ de somme 1. On veut partitionner le carré unité en n rectangles de surfaces $s_1, \ldots, s_n$. On cherche un partitionnement en colonnes comme illustré sur la figure 1. On veut minimiser la longueur des traits dessinés (donc à une constante près la somme des demi-périmètres des n rectangles.

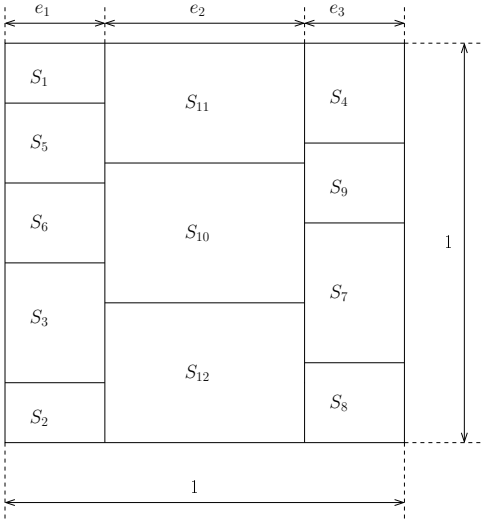

FIG. 1 – Partitionnement du carré unité en colonnes de rectangles.

Question 2.1 Résoudre le problème dans le cas $n = 4$ avec $s_1 = s_2 = s_3 = s_4 = 0,25$.

Solution : *Il suffit de diviser le carré en 4 sous carrés de côté* $0,5$*. La valeur obtenue est de* 2*. Voir Figure 2.* □

Question 2.2 Résoudre le problème dans le cas $n = 6$ avec $s_1 = 0,5$ et $s_2 = s_3 = s_4 = s_5 = s_6 = 0,1$.

Solution : *Il faut mettre le plus grand (*S_1*) avec un de plus petits dans une colonne et le reste dans une deuxième colonne. On obtient* $2,8$*. Voir Figure 3.* □

Question 2.3 Dans le cas général, on ne connaît pas le nombre de colonnes qu'on va utiliser. Mais :

2.3.1 Montrer que l'on peut se ramener à placer les s_i dans l'ordre croissant.

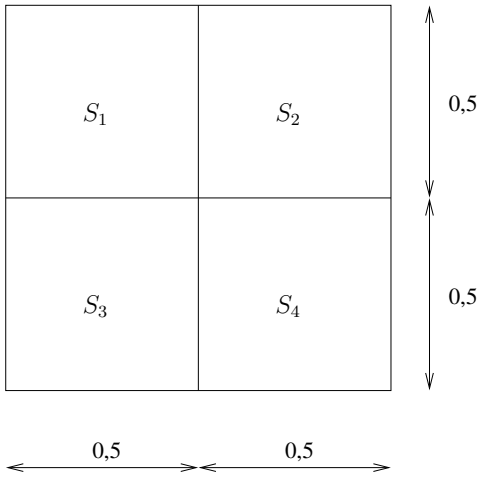

FIG. 2 – Solution pour $n = 4$, $obj = 2$.

Solution : *Montrer qu'on peut se ramener à placer les s_i dans l'ordre croissant, c'est-à-dire tous ceux de la première colonne sont inférieurs à tous ceux de la deuxième, tous ceux de la deuxième sont inférieurs à tous ceux de la troisième, et ainsi de suite.*

Considérons un partitionnement P de somme S (la fonction à minimiser). Cette somme ne change pas si nous réarrangeons les colonnes de manière différente, par exemple par nombre décroissant de rectangles. Donc si S est minimale nous pouvons supposer que pour le partitionnement P, la première colonne a plus de rectangles que la deuxième, qui en a plus que la troisième... Supposons maintenant que les rectangles ne soient pas dans l'ordre croissant, c'est-à-dire qu'il existe un rectangle s dans la colonne i et un rectangle s' dans la colonne j avec $s > s'$ et $i < j$. Soit r_i (respectivement r_j) le nombre de rectangles de la colonne i (respectivement j). Nous avons donc $r_i \geq r_j$. En inversant s et s' nous obtenons un nouveau partitionnement de somme $S' = S - r_i(s - s') + r_j(s - s')$ (coût d'une colonne i : $r_i \cdot \sum_{P_k \in i} s_k$). Si $r_i = r_j$ alors $S = S'$ et si $r_i > r_j$ alors $S' < S$, ce qui contredit la minimalité de S. Cela démontre la propriété. □

2.3.2 Résoudre le problème par programmation dynamique.

Solution :

Pour trouver la récurrence, il suffit de remarquer que si S est minimal pour le partitionnement de $s_1, \ldots, s_n$ en c colonnes avec r rectangles dans la dernière colonne alors en enlevant la dernière colonne on obtient un partitionnement optimal de $s_1, \ldots, s_{n-k}$ en $c - 1$ colonnes.

On commence par trier les s_i par ordre croissant. On crée un (demi) tableau T dont la case d'abscisse c (le nombre de colonne de la solution) et d'ordonnée $k \geq c$ (le nombre de s_i pris en compte) contient $T[c, k] = (S, r)$ où S est la somme minimale des demi-périmètres d'un partitionnement par colonne d'un rectangle de hauteur 1 et de largeur $\sum_{1 \leq i \leq k} s_i$ par $s_1, \ldots, s_k$ et r le nombre de rectangles de la dernière colonne. Le remplissage du tableau s'effectue comme indiqué dans l'Algorithme 3.

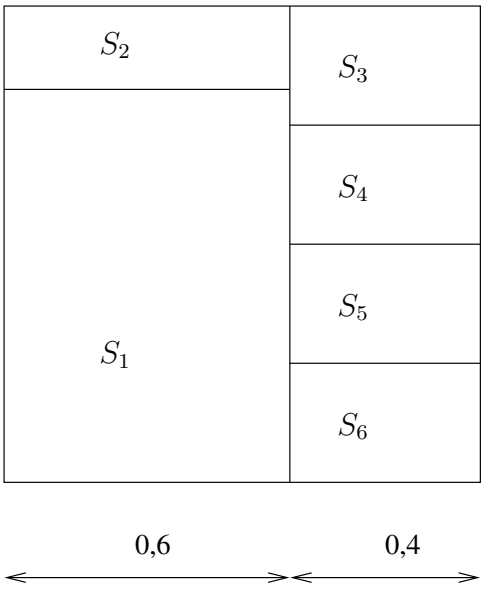

FIG. 3 – Solution pour $n = 6$, $obj = 2,8$.

Algorithme 3 Remplissage du carré

pour k **de** 1 **à** n **faire**
  $T[1,k] = (1 + k \cdot \sum_{1 \leq i \leq k} s_i, k)$
pour c **de** 2 **à** n **faire**
  $T[c,c] \leftarrow (c + \sum_{1 \leq i \leq c} s_i, 1)$
  pour k **de** $(c+1)$ **à** n **faire**
    $i_{min} \leftarrow 1$
    $S_{min} \leftarrow T[c-1,k-1] + 1 + s_k$
    pour i **de** 2 **à** $k - c + 1$ **faire**
      $S \leftarrow T[c-1,k-i]$
      $S_i \leftarrow S + 1 + i \cdot \sum_{j=k-i+1}^{k} s_j$ // Périmètre si la dernière colonne contient i rectangles
      si $S_i < S_{min}$ **alors**
        $S_{min} \leftarrow S_i$
        $i_{min} \leftarrow i$
    $T[c,k] \leftarrow (S_{min}, i_{min})$

Pour calculer le plus petit S pour $s_1, \ldots, s_n$ il faut donc regarder le contenu des cases $T[1,n] \ldots T[n,n]$ et choisir le nombre de colonnes c tel que S soit minimal dans $T[c,n]$. Pour retrouver le partitionnement correspondant il suffit alors de lire le nombre de rectangles r de la dernière colonne dans $T[c,n]$, puis le nombre de rectangles de l'avant dernière colonne dans $T[c-1,n-r]$ et ainsi de suite.

□

3 Le tri idiot...

On se donne un ensemble de n cartes numérotées de 1 à n. On veut trier le paquet de cartes de la manière (stupide) suivante :

- On parcourt le paquet jusqu'à trouver la première carte i qui n'est pas à la place i ;
- On insère cette carte à sa place, c'est-à-dire que si la carte que l'on range est la carte i on la met en i-ème position, en décalant alors toutes celles qui sont au-dessus d'elle ;
- On recommence jusqu'à ce que le paquet soit trié.

Par exemple, si le paquet a 4 cartes et est initialement dans l'ordre $(2, 1, 4, 3)$, on prend la carte 2 qui n'est pas à sa place et on la met en deuxième position : $(1, 2, 4, 3)$, puis comme les cartes 1 et 2 sont à leur place on prend la carte 4 qui est la première qui n'est pas à sa place et on la met en quatrième position : $(1, 2, 3, 4)$ et l'on s'arrête car le paquet est trié.

Question 3.1 Donner la suite des étapes successives du tri d'un paquet de 5 cartes initialement dans l'ordre $(3, 2, 5, 4, 1)$.

Solution : *Paquet initial :* $(3, 2, 5, 4, 1)$.

Étapes du tri : $(2, 5, 3, 4, 1)$ *;* $(5, 2, 3, 4, 1)$ *;* $(2, 3, 4, 1, 5)$ *;* $(3, 2, 4, 1, 5)$ *;* $(2, 4, 3, 1, 5)$ *;* $(4, 2, 3, 1, 5)$ *;* $(2, 3, 1, 4, 5)$ *;* $(3, 2, 1, 4, 5)$ *;* $(2, 1, 3, 4, 5)$ *;* $(1, 2, 3, 4, 5)$ *Ouf !* □

Question 3.2 Quelle est la complexité de l'algorithme (en nombre de déplacements de cartes) dans le cas où le paquet est initialement trié dans l'ordre inverse $(n, n-1, \ldots, 2, 1)$?

Solution : *Si le paquet est trié dans l'ordre inverse* $(n, n-1, \ldots, 2, 1)$*, on aura* $n-1$ *déplacements (1 pour chaque carte sauf la 1 qui sera inversée avec la 2).* □

On veut montrer que cet algorithme termine dans tous les cas, et calculer sa complexité dans le pire des cas. Pour cela, on associe à chaque ordre possible du paquet un poids $p = \sum_{i=1}^{n} 2^{i-1} d_i$ où $d_i = 1$ si la carte i est à sa place dans le paquet et 0 sinon. Par exemple $p(1, 2, 4, 3, 5) = 1 + 2 + 16 = 19$ car 1, 2 et 5 sont à leur place mais pas 3 et 4.

Question 3.3 Montrer qu'à chaque fois qu'on déplace une carte selon l'algorithme considéré le poids de l'ordre du paquet augmente strictement.

Solution : *Le pire cas qu'on puisse avoir est que la première carte ne soit pas à sa place* j*, et que toutes les cartes entre la première place et la* j^e *soient à leur place (celles qui sont derrière la* j^e *place ne modifieront pas le poids après déplacement), voir Figure 4. Dans ce cas, toutes les cartes entre la première place et la* j^e *ne seront plus à leur place après le déplacement. Par contre on remettre bien la carte qui était en première place à sa place véritable* j*.*

La différence entre les deux états est donc au pire :

$$\underbrace{2^{j-1}}_{\text{carte } j \text{ remise sa place}} - \underbrace{\sum_{i=2}^{j-1} 2^{i-1}}_{\text{cartes qui ne sont plus leur place}}$$

Montrons que : $2^{j-1} > \sum_{i=2}^{j-1} 2^{i-1}$.

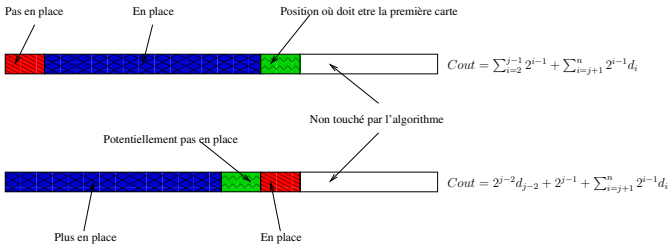

FIG. 4 – Position des cartes avant et après déplacement.

On a $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
donc

$$\begin{array}{rcl} \sum_{i=2}^{j-1} 2^{i-1} & = & \sum_{i=1}^{j-2} 2^i \\ & = & \sum_{i=0}^{j-2} 2^i - 1 \\ & = & 2^{j-1} - 1 - 1 \end{array}$$

Donc $2^{j-1} > \sum_{i=2}^{j-1} 2^{i-1}$*, et le poids augmente bien strictement.* □

Question 3.4 En déduire une preuve que l'algorithme termine toujours et donner une majoration de sa complexité dans le pire des cas (en nombre de déplacements de cartes).

Solution : *On a un poids maximum à atteindre* $P_{max} = \sum_{i=1}^{n} 2^{i-1} = 2^n - 1$*, et l'algorithme fait augmenter le poids strictement à chaque itération. Avec suffisamment d'itérations on voit bien qu'on atteindra la borne supérieure.*

Si au pire on augmente de 1 à chaque itérations il nous faudra $2^n - 1$ *itérations pour atteindre l'état final.* □

On veut maintenant affiner cette borne et montrer qu'elle est atteinte.

Question 3.5 Justifier que le poids augmente d'au moins 2 à chaque déplacement de carte.

Solution : *En reprenant le pire cas exposé précédemment :*

$$\begin{array}{rcl} \sum_{i=2}^{j-1} 2^{i-1} & = & \sum_{i=1}^{j-2} 2^i \\ & = & \sum_{i=0}^{j-2} 2^i - 1 \\ & = & 2^{j-1} - 2 \end{array}$$

Donc $2^{j-1} - \sum_{i=2}^{j-1} 2^{i-1} = 2$*, on augmente bien d'au moins 2 à chaque déplacement de carte.* □

Question 3.6 En déduire que cet algorithme effectue au plus $(2^{n-1} - 1)$ déplacements de cartes sur un paquet de n cartes, quel que soit l'ordre initial du paquet.

Solution : *Vu qu'on augmente d'au moins* 2 *à chaque déplacement de carte, on gagne un facteur* 2 *par rapport à la complexité présentée précédemment. On effectue donc au plus* $2^{n-1} - 1$ *déplacements.* □

Question 3.7 Donner une configuration initiale qui atteint cette borne (en le prouvant si possible).

Solution : *Considérons l'exemple suivant constitué de 5 cartes :* $(2, 3, 4, 5, 1)$. *Le tri est réalisé en* $2^{5-1} - 1 = 15$ *changements.*

Étapes du tri : $(3, 2, 4, 5, 1)$ *;* $(2, 4, 3, 5, 1)$ *;* $(4, 2, 3, 5, 1)$ *;* $(2, 3, 5, 4, 1)$ *;* $(3, 2, 5, 4, 1)$ *;* $(2, 5, 3, 4, 1)$ *;* $(5, 2, 3, 4, 1)$ *;* $(2, 3, 4, 1, 5)$ *;* $(3, 2, 4, 1, 5)$ *;* $(2, 4, 3, 1, 5)$ *;* $(4, 2, 3, 1, 5)$ *;* $(2, 3, 1, 4, 5)$ *;* $(3, 2, 1, 4, 5)$ *;* $(2, 1, 3, 4, 5)$ *;* $(1, 2, 3, 4, 5)$ □