IN 101 - TD 05

Solution

1 Minimisation d'échange d'argent

≪Écrire un programme qui détermine les remboursements à effectuer entre des personnes se devant mutuellement de l'argent, en minimisant le nombre d'échanges. ≫

C'est un problème courant : dans un groupe chacun a payé une partie des dépenses du groupe. L'une a payé le restaurant pour d'autres, l'un a payé les courses pour une autre, etc. À la fin, chacun doit régler ses dettes envers les autres et souhaite effectuer un nombre minimum de virements à ses comparses. Un algorithme glouton existe pour déterminer qui doit *in fine* verser combien à qui.

Il consiste à commencer par calculer le **solde** net de chaque participant, c'est-à-dire la somme de ce qui lui est dû moins la somme de ce qu'il doit.

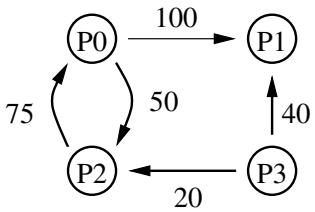
Ensuite, à chaque étape il faut trouver la personne **devant le plus** d'argent et celle **attendant le plus** d'argent. Si ces 2 montants sont nuls, plus personne ne doit rien et plus personne n'attend rien : le problème est réglé. Sinon, le montant du versement à effectuer est le minimum des deux en **valeur absolue**. On crédite et débite les comptes des 2 personnes, on affiche qui verse combien à qui.

Q 1.1. Prenons un exemple concret et nommons les personnes P_0, P_1, etc.
- P_0 doit 100 à $P1$.
- P_0 doit 50 à P_2.
- P_2 doit 75 à P_0.
- P_3 doit 20 à P_2.
- P_3 doit 40 à P_1.

Représentez ces dettes par un schéma.

Solution

Une représentation possible est de représenter les personnes par des ≪patates ≫ et de tirer des flèches entre les débiteurs et les créditeurs associés. Ainsi, une flèche signifie ≪doit à ≫.



Cette représentation est une structure bien connue, un **graphe** que vous étudierez en IN103.

Q 1.2. Imaginez, avec votre chargé de TD, comment représenter ≪informatiquement ≫ (par une structure de données) ce schéma.

Solution

Pour une personne, nous devons mémoriser combien elle doit à **chaque** personne. Autrement dit, pour une personne nous avons besoin du **tableau** de ses dettes.

Puisqu'il faut un tel tableau pour **chaque** personne, nous avons besoin d'un tableau de tableaux, c'est-à-dire un tableau à **2 dimensions** (carré). Chaque case debts[i][j] contiendra ce que la personne i doit à la personne j. Il suffira d'indicer le tableau par le ≪numéro≫ de la personne ($P_0 \to 0, P_1 \to 1$, etc).

doit à \ créd. / déb.	j = 0	1	2	3
i = 0	0	100	50	0
1	0	0	0	0
2	75	0	0	0
3	0	40	20	0

Techniquement, en `C`, nous devrons créer un tableau à 2 dimensions. Comme ces dimensions ne sont pas connues à la compilation, nous devrons faire un ≪tableau de tableaux d'entiers≫. Donc, les fonctions prenant un graphe en argument prendront un **int** ∗∗ en paramètre.

Q 1.3. À partir de la description informelle des traitements, quelles sont les 2 grandes étapes de l'algorithme ?

Solution

1. Transformer le graphe de dettes en un tableau de soldes.
2. Calculer les flux de remboursement à partir du tableau de soldes.

Q 1.4. Esquissez l'algorithme de la fonction qui implémente la **seconde** étape identifiée en Q1.3. Quels sont ses domaines d'entrée et de sortie ?

Solution

Cette fonction prendra en entrée le tableau de soldes et ne retournera rien de particulier : elle affichera les transactions à effectuer.

```
DoTheJob (soldes) :
    Chercher min_i l'indice de la personne ayant le solde le plus débiteur.
    Chercher max_i l'indice de la personne ayant le solde le plus créditeur.
    Si les soldes de min_i et max_i sont == 0 alors fini.
    Calculer le montant m de la transaction.
    Réduire de m la dette en attente du débité.
    Réduire de m le crédit en attente du crédité.
    Afficher (min_i, " donne ", m, " à ", max_i)
    Recommencer.
```

Q 1.5. Esquissez l'algorithme qui implémente la **première** étape identifiée en Q1.3. Quels sont ses domaines d'entrée et de sortie ?

Solution

La fonction correspondante prend en entrée le graphe, donc le tableau à 2 dimensions d'entiers ainsi que la taille de cette matrice (i.e. le nombre de personnes) et retourne un tableau d'entiers.

```
compute_amount (graph, nb_persons) :
    Créer un tableau de soldes de même taille que le nombre de personnes.
    Parcourir le tableau du graphe pour chaque personne i et j
       Ajouter la somme trouvée dans case du graphe[i][j] à la
           somme dans le tableau de solde de j.
       Retirer la somme trouvée dans case du graphe[i][j] de la
           somme dans le tableau de solde de i.
```

Q 1.6. Dans l'esquisse issue de Q1.4, il est question de trouver les personnes ayant le solde le plus débiteur et le solde le plus créditeur. Proposez une solution.

Solution

Ces 2 traitements reviennent à parcourir le tableau en comparant chaque solde à celui se trouvant à l'indice du solde le plus petit / grand, que l'on ait rencontré jusqu'à présent.

Plutôt que de faire 2 fonctions qui vont parcourir chacune le tableau, autant ne faire qu'une seule fonction, un seul parcours, calculant les 2 informations d'un coup. Cette fonction retournera donc un **couple** d'indices.

```
find_min_max_indices (amounts, nb_persons) =
  min_i <- 0
  max_i <- 0
  Pour i = 0 à nb_persons - 1
    Si amounts[i] < amounts[min_i] alors min_i <- i
    Si amounts[i] > amounts[max_i] alors max_i <- i
  Retourner min_i et max_i
```

Comme en `C` on ne peut retourner qu'une seule valeur, nous avons un point technique à régler pour retourner ce couple. Plutôt que de se fabriquer une structure de couple, nous allons choisir de prendre 2 arguments par adresse dans lesquels retourner chaque indice. De ce fait, son type de retour sera **void** puisque les résultats sont tous deux transmis par adresse.

2 Fraction égyptienne

≪Écrire un programme qui transforme une fraction inférieure ou égale à 1 en une fraction égyptienne. ≫

Les Égyptiens antiques n'utilisaient que des fractions dont le numérateur était 1. On qualifie une telle fraction de *unitaire*. Il est possible d'écrire n'importe quelle fraction inférieure ou égale à 1 sous forme d'additions de fractions unitaires de dénominateurs **tous différents**.

L'algorithme glouton de Fibonacci permet de déterminer les fractions à additionner pour représenter une fraction $\frac{a}{b}$.

À chaque étape il cherche la plus grande fraction unitaire $\frac{1}{n}$ **strictement inférieure** à $\frac{a}{b}$ et récurse sur $\frac{a}{b} - \frac{1}{n}$ tant que ce ≪reste ≫ n'est pas une fraction unitaire (ou ne se réduit pas trivialement à une fraction unitaire).

Les fractions $\frac{1}{n}$ trouvées ainsi que le dernier ≪reste ≫ sont celles dont la somme représente la fraction initiale.

Q 2.1. Reformulez cet algorithme de manière plus structurée.

Solution

```
algo (num, denom) :
  Si num divise le denom
    Afficher "1 /" (denom / num)
    Fin
  Trouver n tel que "1/n" soit la plus grande fraction < "num/denom"
  Afficher "1 / " n
  Calculer "x/y" égale à "num/denom - 1/n"
  algo (x, y)
```

Q 2.2. Comment trouver la plus grande fraction unitaire inférieure à $\frac{a}{b}$?

Solution

Preuve courte
Nous devons trouver une fraction unitaire, donc de la forme $\frac{1}{n}$ telle que :

$$\frac{1}{n} < \frac{a}{b}$$

Puisque $\frac{1}{n}$ doit être ≪la plus **grande** ≫, on cherche le $n \in \mathbb{N}$ le ≪plus **petit** ≫.
En réécrivant l'inégalité précédente, on recherche n tel que :

$$n > \frac{b}{a}$$

Donc n le ≪plus **petit** ≫ est la partie entière de la division b/a à laquelle on rajoute 1 :

$$n = \left\lfloor \frac{b}{a} \right\rfloor + 1$$

Preuve plus détaillée
Nous devons trouver une fraction unitaire, donc de la forme $\frac{1}{n}$ telle que :

$$\frac{1}{n} < \frac{a}{b}$$

Puisque $\frac{1}{n}$ doit être ≪la plus **grande** ≫, cela signifie que si on divisait par un n ≪juste inférieur ≫, le résultat **ne serait plus** inférieur strictement à $\frac{a}{b}$. D'où l'on recherche n tel que :

$$\frac{1}{n} < \frac{a}{b} \leq \frac{1}{n-1}$$

C'est-à-dire :

$$n > \frac{b}{a} \geq n - 1$$

Réécrit dans le sens habituel :

$$n - 1 \leq \frac{b}{a} < n$$

De par la définition de la partie entière (y est la partie entière de x si $y \leq x < y + 1$), on en déduit que $n - 1$ est la partie entière de $\frac{b}{a}$:

$$n - 1 = \left\lfloor \frac{b}{a} \right\rfloor$$

Donc le n recherché est :

$$n = \left\lfloor \frac{b}{a} \right\rfloor + 1$$

En résumé :

```
int n = (b / a) + 1 ;
```

3 Implémentation : minimisation d'échange d'argent

Q 3.1. Écrivez la fonction `compute_amounts` qui implémente la **première** étape identifiée en Q1.3 et dont vous avez esquissé l'algorithme en Q1.5.

Solution

```
#include <stdio.h>
#include <stdlib.h>


/* Convert the graph representing debts between persons into a net amount
   per person. */
int* compute_amounts (int **graph, int nb_persons) {
  int *amounts = malloc (nb_persons * sizeof (int)) ;
  if (amounts == NULL) {
    printf ("Error. compute_amounts. No memory.\n") ;
    exit (1) ;
  }
  /* Clear the array. */
  for (int i = 0; i < nb_persons; i++) amounts[i] = 0 ;
  for (int i = 0; i < nb_persons; i++) {
    for (int j = 0; j < nb_persons; j++) {
      amounts[j] = amounts[j] + graph[i][j] ;
      amounts[i] = amounts[i] - graph[i][j] ;
    }
  }
  return amounts ;
}
```

Q 3.2. Écrivez la fonction `find_min_max_indices` dont vous avez esquissé l'algorithme en Q1.6.

Solution

Comme précisé en Q1.6, notre fonction prend 2 arguments en passage par adresse (`int *min_index` et `int *max_index`) dans lesquels elle écrira, en fin de calcul, ses 2 résultats. Notons que durant la recherche, on mémorise les indices dans 2 variables locales (`min_i` et `max_i`) qui sont recopiées à la fin dans `int *min_index` et `int *max_index`. Cela a pour but d'éviter de dé-référencer ces pointeurs à chaque utilisation : question d'efficacité.

```
/* Find the 2 indices of the minimal and maximal values inside an array. */
void find_min_max_indices (int *amounts, int nb_persons,
                           int *min_index, int *max_index) {
```

```
  int min_i = 0 ;
  int max_i = 0 ;
  for (int i = 0; i < nb_persons; i++) {
    if (amounts[i] < amounts[min_i]) min_i = i ;
    if (amounts[i] > amounts[max_i]) max_i = i ;
  }
  *min_index = min_i ;
  *max_index = max_i ;
}
```

Q 3.3. Écrivez la fonction `compute_flow` qui implante l'algorithme esquissé en Q4.

Solution

Nous définissons une macro pour `min`. On pourrait également la définir sous forme d'une fonction. Mais vu ce qu'elle fait, et avec les arguments qu'elle est utilisée (pas de calculs coûteux, pas d'effets de bord) il n'est pas nécessaire de payer le coût d'un appel de fonction. Cela dit, le compilateur faisant des optimisations, il l'aurait certainement ≪inlinée≫ (remplacement de l'appel de la fonction par le code de son corps).

```
#define min(x,y) ((x) < (y) ? (x) : (y))

void compute_flow (int *amounts, int nb_persons) {
  /* Find at once the 2 persons with min and max amount, i.e. the most
     debtor and the most creditor. */
  int min_i, max_i ;

  find_min_max_indices (amounts, nb_persons, &min_i, &max_i) ;
  /* If nobody has to pay and nobody expects money, we are done. */
  if ((amounts[min_i] == 0) && (amounts[max_i] == 0)) return ;
  /* Find the smallest absolute value of amount. */
  int to_pay = min (- amounts[min_i], amounts[max_i]) ;
  /* Reduce the pending debt of the debtor. */
  amounts[min_i] = amounts[min_i] + to_pay ;
  /* Reduce the pending credit of the creditor. */
  amounts[max_i] = amounts[max_i] - to_pay ;
  printf ("%d pays %d to %d.\n", min_i, to_pay, max_i) ;
  compute_flow  (amounts, nb_persons) ;
}
```

Q 3.4. Écrivez la fonction principale `main` qui orchestre tout les traitements. Vous pourrez décrire en dur un graphe représentant un jeu de données (par exemple celui donné dans l'énoncé). Pour vous éviter l'écriture des valeurs à mettre dans chaque case de la matrice, dans le fichier `cflow_skel.c` vous sont données les quelques lignes de code le faisant. Attention il vous faut quand même allouer la matrice `debts` avant !

Attention, comme discuté en Q1.2, vous n'avez pas d'autre choix que l'allocation dynamique pour créer votre graphe puisque c'est un tableau à 2 dimensions de taille *a priori* inconnue à la compilation.

Solution

```
#define NB_PERSONS (4)

int main () {
  /* Allocate the graph.
     TODO: check success of allocations. I'm lazy today ^^ */
  int **debts = malloc (NB_PERSONS * sizeof (int*)) ;
  for (int i = 0; i < NB_PERSONS; i++)
```

```
    debts[i] = malloc (NB_PERSONS * sizeof (int)) ;
  /* Fill it. Manually it's boring!
    [ [ 0, 100, 50, 0 ],
      [ 0, 0, 0, 0 ],
      [ 75, 0, 0, 0 ],
      [ 0, 40, 20, 0 ] ] */
  debts[0][0] = 0 ; debts[0][1] = 100 ; debts[0][2] = 50 ; debts[0][3] = 0 ;
  debts[1][0] = 0 ; debts[1][1] = 0 ; debts[1][2] = 0 ; debts[1][3] = 0 ;
  debts[2][0] = 75 ; debts[2][1] = 0 ; debts[2][2] = 0 ; debts[2][3] = 0 ;
  debts[3][0] = 0 ; debts[3][1] = 40 ; debts[3][2] = 20 ; debts[3][3] = 0 ;

  /* Compute for each person what he must pay or receive. A positive value
     means that the person is creditor. A negative value means thar the
     person is debtor. */
  int *amounts = compute_amounts (debts, NB_PERSONS) ;
  printf ("Amounts:") ;
  for (int i = 0; i < NB_PERSONS; i++) printf (" %d", amounts[i]) ;
  printf ("\n") ;
  compute_flow (amounts, NB_PERSONS) ;
  free (amounts) ;
  return 0 ;
}
```

4 Implémentation : fraction égyptienne

Q 4.1. Écrivez une fonction qui prend en argument une fraction et affiche la suite de fractions unitaires composant la fraction égyptienne associée.

Attention : votre fonction devra travailler avec des **long int** et non de simples **int**.

Solution

```
void work (long int num, long int denom) {
  if (denom % num == 0) {
    printf ("1 / %ld\n", (denom / num)) ;
    return ;
  }
  long int n = (denom / num) + 1 ;
  printf ("1 / %ld\n", n) ;
  long int new_num = num * n - denom ;
  long int new_denom = denom * n ;
  work (new_num, new_denom) ;
}
```

Q 4.2. Écrivez le `main`, recevant sur la ligne de commande la fraction (numérateur et dénominateur), qui lance le calcul de la fraction égyptienne. Il faudra s'assurer que la fraction est bien **inférieure ou égale** à 1.

Pour rappel : la conversion chaîne → **long int** se fait avec la fonction `atol` qui nécessite l'inclusion de `stdlib.h`.

Solution

```
int main (int argc, char *argv[]) {
  if (argc != 3) {
    printf ("Error. Wrong number of arguments.\n") ;
    return 1 ;
  }
```

```
  long int num = atol (argv[1]) ;
  long int denom = atol (argv[2]) ;
  if (num > denom) printf ("Error : input fraction > 1.\n") ;
  else work (num, denom) ;
  return 0 ;
}
```

Q 4.3. Testez votre programme avec $\frac{7}{11}$. Calculez ce que vaut $\frac{1}{2}+\frac{1}{11}+\frac{1}{22}$. Que constatez-vous ?

Solution

Si l'algorithme que vous avez implanté correspond bien à la description donnée pour cet exercice, vous avez dû obtenir :

```
$ ./efract.x 7 11
1 / 2
1 / 8
1 / 88
```

Or, $\frac{1}{2}+\frac{1}{11}+\frac{1}{22}$ est aussi égal à $\frac{7}{11}$. Il y a donc plusieurs solutions possibles avec le même nombre de fractions unitaires.

Q 4.4. Testez votre programme avec $\frac{4}{65}$. Calculez ce que vaut $\frac{1}{26}+\frac{1}{65}+\frac{1}{130}$. Que constatez-vous ?

Solution

Si l'algorithme que vous avez implanté correspond bien à la description donnée pour cet exercice, vous avez dû obtenir :

```
$ ./exfract.x 4 65
1 / 17
1 / 369
1 / 203873
1 / 83128196385
```

Or, $\frac{1}{26}+\frac{1}{65}+\frac{1}{130}$ est aussi égal à $\frac{4}{65}$ et comporte moins de fractions. L'algorithme glouton n'est pas toujours optimal.

De plus, les dénominateurs tendent à grandir très rapidement, pouvant provoquer un « débordement ».

S'il vous reste du temps ou pour continuer après la séance.

5 Gagner un max d'or

Deux joueurs j_1 et j_2 s'affrontent dans un jeu composé de pots (en nombre **pair**) remplis de pièces d'or disposés en ligne. Chaque pot contient un certain nombre de pièces, que les deux joueurs peuvent voir. À chaque tour, un joueur doit choisir de prendre un pot, soit celui de **l'extrémité gauche**, soit celui de **l'extrémité droite** de la ligne de pots. Le but est de maximiser son gain. Ce problème comporte deux hypothèses :

- **Vous** êtes le joueur j_1, celui qui commence.
- À chaque tour, le joueur j_2 ne cherche pas à tricher et joue de manière optimale pour lui.

					j_1	j_2
7	9	5	6		6	
7	9	5				7
	9	5			9	
		5				5

Dans la partie illustrée ci-contre, j_1 prend le pot 6 et non le 7 qui lui semble plus profitable car il laisserait alors le 9 à l'adversaire et se retrouverait à son coup suivant avec le choix entre 5 et 6 : aucun de ces deux choix ne permettrait de gagner la somme maximale. La bonne stratégie permet à j_1 de gagner 15 pièces.

							j_1	j_2
8	3	6	11	10	7		8	
	3	6	11	10	7			7
	3	6	11	10			10	
	3	6	11					11
	3	6					6	
	3							3

Autre exemple de partie avec plus de pots...

Q 5.1. À chaque étape combien de possibilités a le joueur j_1 ? Combien voyez-vous de situations possibles ?

Solution

- Soit il ne reste qu'un pot, et là il n'y a pas le choix, on le prend.
- Soit il reste deux pots, et là, vu que c'est notre dernier tour, il faut prendre le pot qui maximise notre gain.
- Soit il reste plus que deux pots... et là il faut réfléchir.

Q 5.2. Dans le dernier cas, comment retomber sur un problème identique mais de taille plus petite ?

Solution

On a deux choix possibles : soit prendre le pot de l'extrémité gauche, soit celui de l'extrémité droite. Pour obtenir **le gain maximal**, il faut donc prendre le pot qui, additionné avec le **gain maximal** que l'on peut obtenir *de ce qui reste*, nous donne le meilleur résultat.

On voit donc apparaître un problème récursif : on cherche le gain maximal à partir du gain maximal des 2 sous-problèmes. Mais, on n'est pas encore arrivé à une forme récursive correcte : dans nos deux cas, notre gain maximal dépend de ce que va choisir l'adversaire, ce qui ne nous ramène pas encore à un problème où c'est le joueur j_1 (nous) qui doit choisir sur un problème plus petit.

Qu'à ce que cela ne tienne, pour retomber sur un problème où c'est à nous de choisir, il suffit de considérer ce que l'adversaire peut faire à son tour dans chacun de nos choix. Et en fonction de ses choix, ce sera de nouveau à nous de résoudre le même problème sur un sous-problème.

Nommons l l'indice de la case la plus à gauche où il est possible de prendre un pot et r celui de la case la plus à droite.

Q 5.3. Détaillez le raisonnement de la question précédente en exprimant sur quels indices se font les appels récursifs.

Solution

- Si j_1 prend le pot de gauche (l), alors j_2 a deux choix parmi les pots restants dans $[l+1, r]$:
  - s'il prend à gauche ($l+1$), on récurse sur $[l+2, r]$
  - s'il prend à droite (r), on récurse sur $[l+1, r-1]$
- Si j_1 prend le pot de droite (j), alors j_2 a deux choix parmi les pots restants dans $[l, r-1]$:
  - s'il prend à gauche (l), on récurse sur $[l+1, r-1]$
  - s'il prend à droite ($r-1$), on récurse sur $[l, r-2]$

Q 5.4. Sachant que l'adversaire j_2 joue de manière optimale pour lui, quel choix va-t-il effectuer à votre encontre et donc, comment allez-vous combiner le résultat des appels récursifs ?

Solution

Puisqu'il joue de manière optimale, dans chacun de nos 2 choix, le choix qu'il va effectuer sera celui qui **minimise** notre gain. Donc on sait qu'il choisira le `min` parmi les deux gains maximaux que l'on peut récursivement espérer.

Q 5.5. Donnez l'algorithme (naïf) issu des différents cas identifiés.

Solution

Si nous résumons, nous avons 3 cas : reste 1 pot, restent 2 pots, restent n pots. Les deux premiers cas sont simples. Dans le dernier cas, nous avons 2 choix et pour chacun, le joueur adverse j_2 en a 2 pour lesquels il choisira le résultat minimal.

```
max_loot (pots, l, r) =
  Si l = r alors retourner pots[l]
  Si l + 1 = r alors retourner max (pots[l], pots[r])
  Sinon
    gain_gauche <-
      pots[l] + min (max_loot (pots, l + 2, r), max_loot (pots, l + 1, r - 1))
    gain_droite <-
      pots[r] + min (max_loot (pots, l + 1, r - 1), max_loot (pots, l, r - 2))
    Retourner max (gain_gauche, gain_droite)
```

Q 5.6. Pourquoi cet algorithme effectue-t-il plusieurs fois les mêmes calculs ?

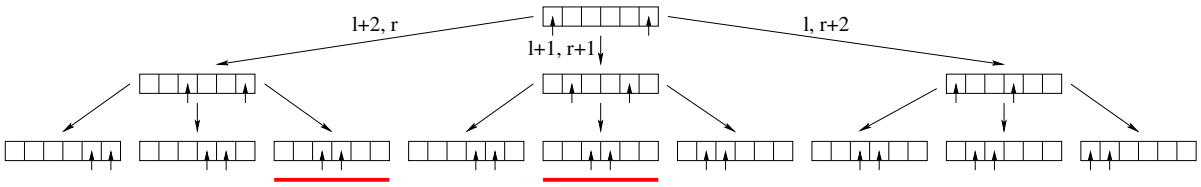
Solution

Sans chercher bien loin, on voit déjà que `max_loot (pots, l + 1, r - 1)` est appelé 2 fois. Donc on va effectuer la descente récursive 2 fois. Cela dit, on pourrait ne le calculer qu'une seule fois et le mémoriser dans une variable pour calculer `gain_gauche` et `gain_droite`.

Mais le problème est plus profond. Supposons deux possibilités de deux itérations successives :

	Cas 1	Cas 2
Itération 1	$l' \leftarrow l+2,\ r' \leftarrow r$	$l' \leftarrow +1, \vert r' \leftarrow r-1$
Itération 2	$l'' \leftarrow l',\ r'' \leftarrow r'-2$	$l'' \leftarrow +1,\ r'' \leftarrow r'+1$

Dans les deux cas, à la fin de la seconde itération on a $l'' = l+2$ et $r'' = r-2$, ce qui correspond aux cas soulignés en rouge dans l'arbre des possibilités suivant, illustrant une ligne de 6 pots. Bien entendu, on constate bien d'autres cas de recouvrements.



Q 5.7. Par quelle technique pourrait-on éviter cette redondance de calculs ?

Solution

On voit que l'algorithme procède en explorant l'arbre des possibilités « en profondeur » (quel que soit l'ordre que l'on a choisi, gauche-droite ou droite-gauche). Donc, à un moment il va arriver en bas de l'arbre des possibilités avant de remonter, puis redescendre. Lorsque l'on a résolu un problème à un certain niveau, on peut mémoriser le résultat en remontant. Ainsi, lorsque l'on redescendra explorer une autre branche, si l'on retombe sur le même problème, on n'aura qu'à aller piocher dans notre mémoire.

On peut se faire un tableau à 2 dimensions, indexé sur les valeurs de l et de r dans lequel on mémorisera le gain maximal pour chaque valeur rencontrée de (l, r). Quand on tombera sur un problème pour la première fois, une fois que l'on l'aura résolu, on mémorisera son résultat dans notre tableau pour les éventuels coups suivants.

Le tableau de mémoïsation sera initialisé avec une valeur « impossible » pour signaler que l'on n'a encore trouvé la solution à aucun sous-problème. La valeur -1 fera parfaitement l'affaire. Notre algorithme devient donc le suivant, dans lequel les parties en fond gris restent inchangées.

```
max_loot (pots, l, r, memo) =
  Si memo[l][r] <> -1 alors retourner memo[l][r]
  Sinon
    Si l = r alors
      memo[l][r] <- pots[l]
      Retourner pots[l]
    Si l + 1 = r alors
      best <- max (pots[l], pots[r])
      memo[l][r] <- best
      Retourner best
    Sinon
      gain_gauche <-
        pots[l] + min (max_loot (pots, l + 2, r, memo),
                       max_loot (pots, l + 1, r - 1), memo)
      gain_droite <-
        pots[r] + min (max_loot (pots, l + 1, r - 1, memo),
                       max_loot (pots, l, r - 2), memo)
      best <- max (gain_gauche, gain_droite)
      memo[l][r] <- best
      Retourner best
```

Ah oui, quand même, si l'on cherche la solution pour les pots `8 3 6 11 10 7 5 6 10 10 4 5`, la version naïve fait 1365 appels contre 101 pour la version optimisée. Et pour `8 3 6 11 10 7 5 6 10 10 4 5 8 3 6 11 10 7 5 6 10 10 4 5`, on passe de 5592405 à 485 !

Le code complet de la solution n'est pas inclu dans ce PDF mais il peut être consulté dans l'archive de correction qui vous est fournie (fichier `gold-opt.c`).

On notera toutefois que ce nouvel algorithme requiert $O(n^2)$ espace mémoire, pour n étant le nombre de pots. Il est possible, comme pour l'exemple vu en cours, de faire une version qui procède « de bas en haut », permettant ainsi de réduire ce besoin mémoire à $O(n)$ et surtout de faire disparaître la récursion au profit d'une simple boucle.