Le Compte Est Bon

Lucas Nussbaum

Table des matières

1 Introduction

Ce problème est inspiré d'un jeu télévisé du même nom, faisant partie de l'émission *Des chiffres et des lettres*, diffusée sur *France 2*.

1.1 Règles

Six nombres (appelés *nombres de base* par la suite) sont tirés au sort parmi les nombres de 1 à 10, 25, 50, 75 et 100. Un septième nombre, appelé *but* par la suite, est tiré au sort entre 100 et 999. Le joueur doit combiner les 6 nombres de base à l'aide des opérations mathématiques addition, soustraction, multiplication et division entière pour obtenir le but ou, à défaut, un nombre s'en approchant le plus possible.

1.2 Exemples

Voici quelques exemples afin d'y voir plus clair :

- Nombres de base : 4, 3, 5, 100, 1 et 6. But : 886
  Il est possible d'atteindre le but à l'aide des opérations suivantes :
  100 - 1 = 99
  6 + 3 = 9
  99 * 9 = 891
  891 - 5 = 886
- Nombres de base : 8, 2, 5, 6, 1, 3. But : 927
  A l'aide des algorithmes exposés plus loin, nous pouvons déterminer qu'il n'est pas possible d'atteindre le but. Toutefois, il est possible d'atteindre 930, et c'est le nombre le plus proche du but qu'il soit possible d'atteindre. C'est donc la meilleure solution.
  6 * 5 = 30
  30 + 1 = 31
  8 + 2 = 10
  10 * 3 = 30
  31 * 30 = 930

1.3 Intérêt algorithmique

Ce problème est très intéressant d'un point de vue algorithmique. Contrairement à ce qui est annoncé sur un site web, la seule solution possible n'est pas une recherche aléatoire. Plusieurs solutions permettent d'obtenir de manière performante des résultats, et sont suffisamment différentes pour qu'il soit intéressant de les comparer. Je vais étudier 2 solutions permettant d'obtenir de bons résultats dans la suite de ce texte.

2 Solution récursive

Cette solution est la plus facile à comprendre.

2.1 Principe

- On part d'un tableau de 6 éléments contenant les 6 nombres de base.
- On combine deux de ces éléments afin d'en obtenir un nouveau.
- On obtient donc un tableau de 5 éléments, le nouvel élément remplaçant les 2 éléments nécessaires pour le former.
- On continue à créer de nouveaux éléments ainsi pour chaque tableau créé, pour chaque couple d'éléments possible, et pour chaque opérateur.
- On s'arrête lorsque le nouvel élément généré est égal au but.
- On garde en mémoire une trace de l'élément le plus proche du but, afin de pouvoir le retrouver si on n'arrive pas à atteindre le but.

2.2 Exemple

On part de la situation suivante : Nombres de base : 4, 3, 5, 100, 1 et 6. But : 886.

On a donc un tableau de 6 éléments contenant 4 ;3 ;5 ;100 ;1 ;6. Le premier couple qu'on peut créer à partir de deux éléments de ce tableau est le couple (4,3). On le combine à l'aide de l'addition, et on obtient 7.

On obtient donc le tableau de 5 éléments 7 ;5 ;100 ;1 ;6. On remplace le couple (7,5) par 12, obtenu avec l'addition.

On continue l'exploration descendante puis, si cette exploration ne permet pas d'atteindre le but, on essaie les autres opérateurs, et les autres couples.

2.3 Avantages et inconvénients

2.3.1 Avantages

Le principal avantage de cet algorithme est bien sûr sa facilité de compréhension.

De plus, il est rapide, et facile à implémenter.

2.3.2 Inconvénients

Le principal inconvénient de cet algorithme est qu'il ne permet pas de conserver en mémoire l'ensemble des solutions trouvées, ou des résultats intermédiaires. Cela peut-être gênant lors de l'analyse des résultats, par exemple pour rechercher un résultat particulier.

D'autre part, la solution trouvée par l'algorithme n'est pas forcément la plus simple (celle obtenue en effectuant le moins d'opérations, ou en utilisant le moins de nombres de base). L'algorithme dynamique décrit plus loin permet d'éviter cet inconvénient.

2.4 Implémentation

Une implémentation en C est disponible dans le fichier **lceb_rec.c**.

2.4.1 Utilisation

Après l'avoir compilé en tapant `make`, lancer le programme en tapant `./lceb_rec` *but suivi des 6 nombres de base*. Par exemple :

```
$ ./lceb_rec 886 4 3 5 100 1 6
Processing 4 3 5 100 1 6 . Goal : 886
FOUND
100 - 1 = 99
6 + 3 = 9
99 * 9 = 891
891 - 5 = 886
```

`FOUND` signifie que le but a été atteint. Dans le cas contraire, `NOTFOUND` est suivi du résultat atteint, ainsi que de la distance par rapport au but :

```
$ ./lceb_rec 749 4 1 2 3 8 6
Processing 4 1 2 3 8 6 . Goal : 749
NOTFOUND 750 1
8 * 6 = 48
48 + 2 = 50
4 + 1 = 5
5 * 3 = 15
50 * 15 = 750
```

2.4.2 Détails de l'implémentation

Les valeurs contenues dans les tableaux sont stockées sous la forme d'une structure :

```
struct result {
   int val; /* value */
   operation op; /* operator */
   struct result * l; /* left operand */
   struct result * r; /* right operand */
};
```

En plus de contenir la valeur, elle contient les 2 résultats nécessaires pour l'obtenir, ainsi que l'opérateur utilisé. Si par exemple on stocke 4, comme la somme de 2 autres nombres 1 et 3, `val` vaudra 4, `op` vaudra ADD, et `l` et `r` pointeront respectivement sur les structures contenant 1 et 3.

La fonction `dispres` permet de remonter l'arbre des opérations, et ainsi, d'écrire les opérations nécessaires pour arriver à un résultat sous une forme lisible, comme dans les exemples ci-dessus.

La fonction `add` retourne, pour 2 nombres et un opérateur donnés, la structure du résultat, si l'opération est valide.

La fonction `resultest` permet de vérifier si un résultat intermédiaire est égal au but, et arrête l'exploration si c'est le cas.

La fonction `compute` est la fonction appelée récursivement pour traiter les tableaux de nombres.

Il faut noter que la mémoire n'est jamais libérée. Tous les résultats intermédiaires sont conservés, car il serait trop compliqué de déterminer lesquels sont nécessaires à l'affichage ultérieur éventuel d'une série d'opérations par dispres.

3 Solution dynamique

Cette solution est bien plus satisfaisante d'un point de vue logique.

3.1 Principe

Comme dans tout algorithme dynamique, la recherche de la solution passe par la recherche de sous-solutions.

Le point de départ est les 6 nombres de base. On va rechercher tous les résultats intermédiaires composés par 2 de ces 6 nombres de base, en combinant tous les couples possibles à l'aide de tous les opérateurs disponibles. Ensuite, les résultats intermédiaires composés de 3 nombres de base sont recherchés, puis par 4, 5, et 6 nombres de base.

Dès qu'un résultat intermédiaire est égal au but, on peut s'arrêter : on a trouvé une solution, et c'est une des plus propres, puisqu'elle utilise un minimum de nombres de base.

Si, à la fin de la recherche de tous les résultats intermédiaires, on n'a pas trouvé de résultat égal au but, il suffit de les parcourir à nouveau en recherchant le résultat qui s'en approche le plus.

3.2 Avantages et inconvénients

3.2.1 Avantages

Cet algorithme peut être facilement adapté à la recherche de solutions spécifiques. Par exemple, il est facile de rechercher la solution utilisant le plus de nombres de base, ou passant par le résultat intermédiaire le plus grand. La solution récursive exposée précédemment ne permet pas de faire facilement de telles recherches.

De plus, cet algorithme est rapide. Lors de la recherche de la première solution, il se montre un peu plus rapide que la solution récursive.

3.2.2 Inconvénients

Par contre, cet algorithme a le désavantage de nécessiter le stockage de tous les résultats intermédiaires. C'est aussi le cas dans la solution récursive avec l'implémentation choisie, mais d'autres implémentations de la solution récursive peuvent éviter ce problème. L'utilisation de la mémoire par l'implémentation de l'algorithme qui est présentée plus loin peut être très importante.

3.3 Implémentation

Une implémentation en C est disponible dans le fichier **lceb_dyn.c**.

3.3.1 Détails de l'implémentation

De nombreux points de l'implémentation sont communs avec celle de la solution récursive.

Deux champs sont ajoutés à la structure `result` :
- `used` décrit quels sont les nombres de base utilisés par un résultat intermédiaire donné.
- `next` est un pointeur vers une structure result, permettant de stocker ces structures dans une liste chaînée.

Le tableau `results`, déclaré dans la fonction `main`, contient tous les résultats intermédiaires. `results[0]` contient les nombres de base, `results[1]` contient les résultats composés par 2 nombres de base, `results[2]` contient les résultats composés par 3 nombres de base, etc ...

Dans l'implémentation actuelle, l'exploration est arrêtée dans `resultest` dès que le but est atteint. Ensuite, si le but n'est pas atteint, tous les résultats sont repassés en revue à la fin de la fonction `main` pour déterminer celui s'approchant le plus du but.

3.4 Optimisations éventuelles

3.4.1 Mémorisation du résultat le plus proche en cours de calcul

Il est possible de mémoriser le résultat le plus proche du but en calculant la proximité du résultat courant dans `resultest`, et en positionnant `best` et `min`. Mais ce surplus de calcul ralentit l'algorithme. En effet, le but est atteint dans 98% des cas, et il est donc préférable de rechercher le résultat le plus proche séparément.

3.4.2 Utilisation d'un tableau de fonctions pour les opérateurs

Une étude avec GNU prof tend à montrer que le `switch` dans la fonction `add` est un point faible de l'algorithme. Pour le contourner, un tableau de fonctions est ajouté. L'implémentation avec tableau de fonction est contenue dans **lceb_dynopt.c**.

4 Comparaison des performances des différentes implémentations

Ces valeurs ont été obtenues en compilant avec GNU cc v. 2.95.3, avec l'option -O6, et en utilisant une liste de 1000 valeurs générées aléatoirement avec le programme random.c. Les durées ont été mesurées avec l'utilitaire UNIX `time` et le script `lcebusedata.sh` fourni.

Version	real	user	sys
Récursif	1m13.324s	1m5.239s	0m20.903s
Dynamique	0m47.694s	0m40.080s	0m20.804s
Dynamique optimisé	0m47.665s	0m40.230s	0m20.693s

La très faible différence entre les versions dynamiques et dynamiques optimisé laissent penser que le compilateur effectue lui-même l'optimisation consistant à utiliser un tableau de fonctions.

5 Rechercher autre chose

5.1 Plus grande somme des résultats intermédiaires

Le programme **lceb_bigsum.c** montre avec quelle facilité il est possible de modifier la version dynamique pour rechercher autre chose. Ce programme recherche la solution dont la somme des résultats intermédiaires est la plus grande.

Exemple :

- Avec recherche de la plus grande somme :

```
$ ./lceb_bigsum 374 50 3 5 10 8 75
Processing 50 3 5 10 8 75 . Goal : 374
FOUND
```

```
75 * 50 = 3750
3750 + 3 = 3753
3753 - 5 = 3748
3748 - 8 = 3740
3740 / 10 = 374
```

– Version dynamique classique :

```
$ ./lceb_dyn 374 50 3 5 10 8 75
Processing 50 3 5 10 8 75 . Goal : 374
FOUND
75 * 5 = 375
10 - 3 = 7
375 + 7 = 382
382 - 8 = 374
```

6 Recherche de la solution dont l'arbre est le moins profond

L'algorithme dynamique permet aussi, avec quelques modifications, de rechercher la solution dont l'arbre des solutions est le moins profond. Par exemple, si on cherche à faire 4 avec 1, 1, 1, 1, il sera plus élégant de faire $(1 + 1) + (1 + 1)$, que $((1 + 1) + 1) + 1$.

Le fichier **lceb_easiest.c** recherche la solution dont l'arbre est le moins profond.

Exemples :

– Version dynamique classique :

```
$ ./lceb_dyn 444 100 1 3 75 8 10
Processing 100 1 3 75 8 10 . Goal : 444
FOUND
75 - 1 = 74
10 + 8 = 18
74 * 18 = 1332
1332 / 3 = 444
```

– Version avec recherche de l'arbre le moins profond : En affichant tous les résultats intermédiaires, on constate que la solution proposée par l'algorithme dynamique classique a une complexité de 4221, alors qu'une autre solution, utilisant le même nombre de nombres de base, a une complexité inférieure :

```
*** 2421 ***
75 - 1 = 74
10 + 8 = 18
18 / 3 = 6
74 * 6 = 444
```

7 Composition de l'archive

Outre les exemples déjà présentés, l'archive contient aussi plusieurs scripts permettant de faciliter les tests des différents algorithmes.

– **Makefile** : fichier permettant d'automatiser la compilation des programmes en tapant `make`.

– **getnotfound.sh** : ce fichier permet, à partir d'une liste de problèmes, d'afficher ceux pour lesquels une solution n'a pas pu être trouvée. Exemple : `./random 100 | ./getnotfound.sh lceb_dyn`

- **lceb_bigsum.c**, **lceb_dyn.c**, **lceb_dynopt.c**, **lceb_easiest.c**, **lceb_rec.c** sont les fichiers sources des algorithmes présentés précédemment. Ils sont sous licence GPL.
- **lceb.tex** est le fichier LaTeX utilisé pour générer ce que vous êtes en train de lire. Il est placé sous licence FDL.
- **lcebusedata.sh** permet d'utiliser une liste de problèmes avec un algorithme. Exemple : `./lcebusedata.sh lceb_dyn liste`.
- **random.c** permet de générer aléatoirement une liste de problèmes. Exemple : `./random 100` (pour générer 100 problèmes).

8 Conclusion

J'espère que la lecture de ces quelques idées sur ce problème très ludique vous aura intéréssée. Si vous pensez que des améliorations sont à apporter à ce document ou aux programmes d'exemple, vous pouvez bien sûr me contacter à l'adresse lucas@lucas-nussbaum.net.

9 Licence