Résolution du jeu "Le Compte est bon"

Arthur Barjot

Janvier 2022

Contents

1 Introduction, règles du jeu

J'ai produit un programme de résolution du jeu "Le Compte est bon", issu de l'émission "Des Chiffres et des Lettres" ceci en m'appuyant sur les cours que j'ai pu suivre en MP* option informatique au lycée Blaise Pascal ainsi que mes autres projets précédents. J'atteste avoir écrit, créé et eu l'idée seul de tout ce projet.

Le jeu se joue avec des "plaques" (des entiers naturels) et un résultat (aussi un entier naturel, le principe est de combiner nos plaques (en les utilisant qu'une seule fois maximum chacune) avec des opérations arithmétiques, en faisait attention que chaque résultat intermédiaire soit un entier naturel (problème avec la soustraction et la division). Plus précisément, on fournit au joueur (ici au programme):

-un résultat : r, un entier avec r appartenant à [100,999], exemple : 493.

-une liste d'entiers (les plaques) : l contenant 6 nombres, dont des éléments de [1,10], qui peuvent chacun intervenir 2 fois maximum chacun ; et des éléments de {25;50;75;100}, qui peuvent chacun intervenir 1 fois maximum chacun, exemple : [1;5;100;7;1;25].

2 Résolution du jeu

2.1 Principe de la résolution

- Le principe est récursif : Durant une partie le joueur combine les plaques pour obtenir des résultats qu'il pourra réutiliser, on peut donc considérer tout ces résultats intermédiaires comme faisant partie de nouvelles plaques du joueur, auxquelles on aurait retiré les plaques déjà utilisées. On obtient donc que chaque étape du jeu peut être considéré comme le début d'une partie, avec le même résultat, les plaques étant celle que nous avons définie plus haut comme des "nouvelles plaques". Il y en a évidemment strictement moins de "nouvelles plaques" car deux anciennes plaques sont nécessaires à la création d'une nouvelle, nous en perdons au moins une.

- Définissons la méthode d'étude d'une configuration possible des plaques : On note encore r le résultat et $l = [x_1; x_2; x_3; ...; x_n]$ la liste des plaques données dans un certain ordre. Tout d'abord si r = 0 on a le compte bon. Sinon, si l n'a qu'un seul élément x_1, on regarde si r est plus proche de 0 ou de x_1, et nous renvoyons donc 0 ou x_1 en fonction de qui sera la meilleure approche. Sinon l contient au moins deux éléments, on commence par vérifier que l'on a pas le compte bon en un coup, donc on regarde si un des x_i vaut r, si c'est le cas on a fini.

  Principe récursif : Dans l'autre cas, on teste de résoudre de jeu (avec la méthode que je suis en train de présenter) avec le même résultat r et les nouvelles listes $[x_1+x_2; x_3; ...; x_n]$; $[x_1-x_2; x_3; ...; x_n]$; $[x_2-x_1; x_3; ...; x_n]$; $[x_1 * x_2; x_3; ...; x_n]$; $[x_1/x_2; x_3; ...; x_n]$; $[x_2/x_1; x_3; ...; x_n]$. On écarte rapidement les cas x_1/x_2 non entier ou $x_1 - x_2$ négatif... Puis on s'arrange pour

récupérer le meilleur des 6 résultats obtenus. Cette méthode termine car la taille de l réduit de 1 à chaque appel récursif et nous avons vu plus haut les cas limites.

- Nous avons donc une méthode pour trouver le meilleur résultat possible pour une certaine "configuration de l", c'est à dire un certain ordre de ses éléments. Il ne reste plus qu'à regarder les différents résultats pour chaque configuration possible de l. En fait on remarque que seuls les deux premiers éléments sont utilisés dans la boucle récursive, on regarde donc seulement les différentes configurations pour les deux premiers éléments (il y en a $\frac{n\times(n+1)}{2}$). Ensuite on peut choisir une solution optimale en les regardant toutes et en les comparant pour ne garder que la meilleure. Ou une solution "rapide" qui renvoie un résultat des qu'une configuration conduit à un compte bon.

- Pour ce qui est de l'affichage de la solution : il faut renvoyer la façon d'obtenir le résultat. Pour cela nous allons utiliser une liste s de chaînes de caractères vérifiant une bijection permanente avec la liste l des plaques : en place i de l se trouve l'élément x_i et en place i de s se trouve la représentation de x_i. Par exemple si x_i est l'une des plaque renseignée en début de jeu, s contient "x_i" en place i. Si au contraire x_i est le fruit d'opération faites durant la partie : par exemple au début de la partie l vaut $[2; 3; 10; 50; 4; 9]$ disons que l'on arrive en cours de partie à la liste l dont l'élément numéro i, x_i est trouvé avec les opérations $x_i = \frac{2+10}{4} = 3$ alors l contient 3 en place i et s contient "$(2 + 10)/4$". Ainsi lorsque l'on trouve un résultat du jeu, nous n'avons plus qu'à regarder à la place correspondante dans s la façon de l'obtenir. On comprend qu'avec cette méthode, nous devons conserver la bijection à tout moment de l'algorithme et donc à chaque modification de l nous modifions en conséquence s.

2.2 Solution optimale

J'ai produit un programme capable de trouver une solution dite "optimale" à ce jeu, c'est à dire :

- Lors ce que le jeu a une solution (on peut arriver au résultat exacte), on parle de "compte bon", solution utilisera le moins de plaques possible pour atteindre cette solution et donc le moins d'opérations possible. Il renverra également un booléen "true", signifiant que le programme a trouvé le compte bon.

  Exemple de résolution lorsqu'il y a compte bon :

  ```
  solution_chiffre_dcdl [301;9;8;1;9;5;1];;
  ```

  Le programme renvoie en 1.1 secondes :

  ```
  sol = S (301, "(((1+1)*((9+8)*9))-5)", true)
  ```

- Lors ce qu'il n'y a pas de solution exacte, le programme renverra la solution la plus proche du résultat qu'il trouvera et celle utilisant le moins de plaque, il renverra également le résultat auquel il arrive et un booléen "false", signifiant que le compte n'est pas bon.

  Exemple de meilleure approche lorsqu'il n'y a pas compte bon :

  ```
  solution_chiffre_dcdl [857;2;50;3;1;1;7];;
  ```

  Le programme renvoie en 0.9 secondes :

  ```
  sol = S (856, "(7+1)*(((50+3)*2)+1)", false)
  ```

2.3 Solution rapide

J'ai également produit un programme "rapide", ou du moins, beaucoup plus rapide que le précédent. Celui-ci à pour but lors de comptes compliqués, dépassant les règles du jeu (ceci est expliqué dans la partie suivante) de trouver une solution sans trop attendre. En fait le point négatif est qu'il ne renvoie pas une solution optimale, il renvoie la première solution qu'il trouve, celle-ci peut ne pas être visuellement très belle (trop longue alors qu'un humain trouverai rapidement une solution bien plus courte). De plus si le compte n'est pas bon elle ne renvoie pas de résultat approché.

Exemple de résolution rapide :

```
resol_rapide [801;10;50;3;1;1;75];;
```

Le programme renvoie en 0.3 secondes :

```
sol = S (801, "(((1+3)*75)+((10*50)+1))", true)
```

3 Extension de la résolution

Mon programme résout aussi ce jeu avec moins de contraintes :

-On peut lui fournir une listes l d'entiers, de taille quelconque, et d'entiers quelconques avec un nombre d'occurrence de chacun quelconque.

-On peut lui fournir un résultat quelconque avec pour seule condition que ce résultat soit différent de 0.

Attention tout de même, si l'on augmente la taille de l, les temps de calcul sont décuplés, je vous conseille alors d'utiliser le programme "resol_rapide" qui pourra résoudre dans un temps encore acceptable si l n'est pas trop grande, mais il ne renverra pas la solution optimale, seulement la première solution qu'il trouvera.

Exemple de la puissance, l'utilité que peut avoir ce programme, avec cette entrée :

```
resol_rapide [91372; 172;96;268;64;4;22;78];;
```

Le programme renvoie en 17 secondes :

```
sol = S (91372, "((((22*96)-((172+64)*4))*78)+268)", true)
```

Ainsi on voit que mon programme résout dans un temps raisonnable un problème qui aurait était très dur pour un être humain.