Projet : Le compte est bon

Notions : Fichiers, structures, récursivité, développement logiciel

1 Introduction

Le *compte est bon* est à l'origine un jeu télévisé créé par Armand Jammot, diffusé pour la première fois le 4 janvier 1972 dans l'émission Des Chiffres et des Lettres. Voir la vidéo des archives INA pour un exemple.

Le but est d'atteindre un nombre cible entre 100 et 999 à l'aide de six nombres tirés au hasard sans remise parmi 24 nombres possibles et des quatre opérations élémentaires : addition, soustraction, multiplication, division entière, en respectant les règles suivantes :

- Seuls les nombres entiers strictement positifs sont acceptés
- Les 24 nombres possibles sont les nombres 1 à 10 en 2 exemplaires et les nombres 25, 50, 75 et 100 en 1 seul exemplaire.
- Chaque nombre tiré peut être utilisé une seule fois ou pas du tout. Un résultat intermédiaire peut être bien sûr réutilisé.

L'exemple de la vidéo est de trouver 595 avec le tirage suivant de 6 nombres : 10 1 25 9 3 6.

Et le compte est bon en utilisant les nombres 9,6,3,10,25 et en réalisant les opérations suivantes :

1. 9 * 6 = 54
2. 54 + 3 = 57
3. 57 * 10 = 570
4. 570 + 25 = 595

Les 2 problèmes que nous allons résoudre sont les suivants, pour N nombres tirés au hasard [1] et pour un nombre cible inférieur à 100000 [2] :

1. problème 1 : trouver si le nombre cible peut être atteint et afficher la suite d'opérations correspondantes
2. problème 2 : trouver le nombre le plus proche du nombre cible qui peut être atteint et afficher la suite d'opérations correspondantes

2 Algorithme

Pour trouver les solutions d'un tirage au compte est bon, la seule méthode est de réaliser tous les calculs possibles.

Pour le problème 1, on s'arrête dès qu'une solution exacte est trouvée. Pour le problème 2, on s'arrête aussi dès qu'une solution exacte est trouvée. Mais si la cible n'est pas atteignable, il faut tester toutes les solutions possibles pour conserver la meilleure.

1. Extension de 6 nombres à un nombre N quelconque de valeurs tirées
2. Extension de la cible maximale

Le principe général est de partir du tirage des N nombres, d'en choisir 2 et d'essayer toutes les opérations (+,-,*,/) avec ces 2 nombres. Si le résultat d'une opération est la cible attendue, c'est gagné. Sinon, on remplace les 2 nombres par le résultat de l'opération et on recommence avec les N-1 nombres du tirage modifié, et ainsi de suite.

2.1 Complexité

En premier lieu, avec beaucoup de chance, le nombre cible à trouver est parmi les N nombres du tirage, soit N possibilités.

Ensuite, en utilisant le principe général exposé ci dessus, on choisit donc 2 nombres parmi N à la première étape, soit C_N^2 possibilités. Comme il y a 4 opérations possibles, il y a donc $4 * C_N^2$ possibilités.

Si la solution n'est pas parmi ces possibilités, on passe à l'étape suivante, en choisissant 2 nombres parmi les N-1 restants et 4 opérations. Il y a alors $4 * C_{N-1}^2$ possibilités pour cette étape, soit $4 * C_N^2 * (1 + 4 * C_{N-1}^2)$ possibilités au total.

On montre que le nombre de solutions à tester dans le cas général de N valeurs tirées vaut :

$$S = N + \sum_{i=1}^{N-1} 4^i * \prod_{j=N-i+1}^{N} \binom{j}{2} \tag{1}$$

La croissance de ce nombre est très rapide, comme le montre la table **??**. Lorsqu'un nombre cible n'est pas atteignable avec un tirage de N nombres, le coût de la recherche, qui effectue tous les tests, peut donc s'avérer prohibitif.

Table 1 – Nombre de tests à effectuer

Nombre de valeurs tirées	Nombre de tests
3	63
4	1468
5	58605
6	3516066
7	295349131
8	33079102008
9	4763390688153
10	857410323866110
11	188630271250542231
12	12904903462724043124

Sur nos machines modernes, le compte est bon avec 6 valeurs est résolu quasiment instantanément dans le pire des cas. Pour 7 valeurs, il faut moins d'une seconde lorsque la solution n'existe pas. Pour 8 valeurs, c'est de l'ordre de 30 à 60 secondes. Au delà, avec la méthode récursive proposée, le temps n'est plus raisonnable lorsque la solution n'existe pas, ce qui est assez rare.

2.2 Algorithme général

Le tirage est représenté par un tableau alloué dynamiquement de N entiers qui contiennent les valeurs du tirage.

L'algorithme est simplement un algorithme récursif qui teste les solutions l'une après l'autre.

1. On commence par sauvegarder l'état de notre jeu.
2. On prend le premier nombre et on en choisit un deuxième ainsi qu'une opération.
3. On remplace le premier nombre par le résultat de notre opération.
4. Si le résultat de l'opération coïncide avec la cible, on a gagné. On stocke l'opération réalisée et on quitte la fonction en retournant la valeur **0** (on a trouvé la solution)

5. sinon,
   (a) On remplace le deuxième nombre utilisé par le dernier nombre du tableau
   (b) On recherche la solution avec ce nouveau tableau et N-1 nombres
   (c) Si la solution est trouvée,
      i. On a gagné,
      ii. On stocke l'opération réalisée,
      iii. On restaure l'état du tableau avant les changements effectués,
      iv. On quitte la fonction en retournant la valeur 0 (on a trouvé la solution)
   (d) sinon, on restaure l'état du tableau avant les changements effectués et on itére sur le choix d'un nombre et d'une opération de l'étape 2.
6. Si on arrive ici, pas de solution : on retourne 1

L'algorithme correspondant, très simple à mettre en œuvre, est donné par l'Algorithme **??**

Algorithm 1 int resolution(int* tirage, int nb, int cible, char** operations)

Entrées: tirage, le tirage de départ
Entrées: nb, le nombre de valeurs utiles
Entrées: cible, la valeur à atteindre
Entrées: operations, le tableau des opérations réalisées pour trouver la cible
Sorties: 0 si valeur cible est atteinte, 1 sinon

```
1: Pour tout les nombres tirage[i] Faire
2:   Pour tout les nombres tirage[j] avec i! = j Faire
3:     Pour tout les 4 opérations Faire
4:       Sauvegarder l'état du tableau tirage
5:       Effectuer le calcul tirage[i] operation tirage[j]
6:       Mettre le résultat dans tirage[i]
7:       //tirage[j] est utilisé et maintenant interdit
8:       Mettre le dernier nombre non utilisé tirage[nb − 1] à la place de tirage[j]
9:       Si le résultat est la cible OU resolution(tirage, nb-1, cible, operations) == 0 Alors
10:        //On a trouvé une solution, soit immédiatement (resultat=cible)
11:        //soit en continuant les opérations avec les nb-1 nombres restants
12:        //On met l'operation et les nombres dans le tableau operations
13:        Mettre les nombres, l'opération utilisés et le résultat dans le tableau operations
14:        Restaurer les valeurs du tableau tirage
15:        return 0 ;
16:      Fin Si
17:      // tirage[i] operation tirage[j] ne mêne pas à une solution.
18:      // On essaye une autre opération ou un autre couple i,j
19:      Restaurer les valeurs du tableau tirage
20:    Fin Pour
21:  Fin Pour
22: Fin Pour
23: //Aucune solution possible si on arrive ici
24: return 1 ;
```

2.3 Affichage des résultats

Pour pouvoir afficher les opérations menant aux résultats, il est possible de supprimer le tableau `operations` et de remplacer la ligne 13 par un affichage simple des nombres utilisés et de l'opération réalisée avec un `printf` [3].

Cette version simplifiée de l'affichage présente 2 inconvénients :

3. Cette version simplifiée peut cependant vous être utile pour la mise au point de votre code

— l'ordre des opérations affichées est l'ordre inverse des opérations qui doivent être effectuées.

— il se peut que des opérations inutiles soient affichées.

Pour répondre au premier problème, le tableau *operations* est un tableau de chaînes de caractères qui contient les opérations effectuées. Une fois l'algorithme exécuté, ce tableau est facilement affiché dans l'ordre inverse de son remplissage et l'ordre normal des opérations est restauré.

Quand une solution est trouvée, on stocke l'opération qui a permis d'obtenir cette solution dans ce tableau sous forme d'une chaîne de la forme "10 + 4 = 14" en utilisant la fonction `sprintf` qui remplit une chaîne de caractères de la même manière que la fonction `printf` affiche. Une instruction du genre `sprintf(operations[k],"%d %c %d = %d",val1,op,val2,res)` pourra être utilisée : l'entier `val1` est le premier opérande, l'entier `val2` est le deuxième opérande, `op` est le caractère correspondant à l'opération réalisée et `res` est le résultat obtenu par cette opération.

A vous de voir dans quelle ligne `k` du tableau `operations` cette chaîne doit se trouver.

Le type du tableau *operations* est `char**` car le nombre de lignes dépend du nombre N de valeurs du tirage. Il doit donc être alloué dynamiquement au début du programme et libéré à la fin. On allouera un tableau de N lignes de 256 caractères.

2.4 Tirage des nombres

Vous travaillerez avec des tirages qui seront obtenus soit par un tirage aléatoire, soit par lecture d'un fichier que nous fournissons dans le répertoire `jeux`.

2.4.1 Format des fichiers de données

Les fichiers du répertoire `jeux` contiennent le tirage et la valeur cible dans le format suivant (exemple figure **??**) :

— première ligne : le nombre N de valeurs tirées

— deuxième ligne : les N valeurs tirées

— troisième ligne : la valeur cible à atteindre

9 3 8 10 100 50 10 6 6 1 123456	6 10 10 25 50 75 100 653

FIGURE 1 – Exemple de fichiers avec tirage à 9 et à 6 valeurs.

Les fichiers `jeux/jeu6-x.txt` à `jeux/jeu10-x.txt` sont des tirages à 6,7,8,9 chiffres qui ont une solution exacte. Les fichiers `jeux/jeu106-x.txt` à `jeux/jeu109-x.txt` sont des tirages à 4,6,7,8,9 chiffres qui n'ont pas de solution exacte[4].

2.4.2 Tirage aléatoire

Dans un premier temps, il faut d'abord créer un tableau *plaques* contenant les 24 plaques (2 exemplaires de 1 à 10, 1 exemplaire des valeurs 25, 50, 75, 100). Pour réaliser un tirage aléatoire de N valeurs à partir de ce tableau, le principe est le suivant :

1. tirer au hasard **l'indice** `i` de la valeur (et non la valeur elle-même), i.e un nombre entre 0 et 23 (indice de la dernière plaque).
2. mettre la **valeur** choisie ($plaques[i]$) dans le tirage.
3. remplacer la valeur à l'indice `i` de *plaques* par la dernière valeur du tableau *plaques* non utilisée (soit la valeur à l'indice 22 à la première itération).
4. recommencer l'étape 1 avec les plaques restantes (23 et non plus 24 à la première itération) jusqu'à obtenir les N valeurs.

4. Tous les fichiers proposés ont des résolutions aux problèmes 1 et 2 en moins de 1 à 2 minutes

3 Les 2 problèmes

3.1 Problème 1

L'algorithme proposé donne la solution du premier problème directement. S'il existe une solution, vous pouvez facilement augmenter le nombre N de valeurs tirées.

Pour une cible donnée, plus le nombre de valeurs tirées est grand, plus la résolution est facile. Le seul cas problématique est celui de l'absence de solutions pour un tirage et une cible donnée, qui devient prohibitif à partir de 9 valeurs.

3.2 Problème 2

S'il n'existe pas de solutions, l'algorithme précédent va parcourir tous les cas possibles. Au cours de ce parcours, il existe un (ou plusieurs) cas qui sont les plus proches de la cible.

Pour la détecter et l'afficher, il faut modifier l'algorithme précédent pour garder en mémoire la meilleure solution trouvée à un instant, ainsi que la valeur de cette solution. On ne garde qu'une seule solution. Il faut donc la mettre à jour à chaque essai qui est meilleur que les précédents. L'algorithme **??** ci-dessous est une modification du précédent.

Algorithm 2 int resolutionplusproche(int* tirage, int nb, int cible, int distancemin, char** operations)

Entrées: tirage, le tirage de départ ; nb, le nombre de valeurs utiles ; cible, la valeur à atteindre
Entrées: distancemin, la distance a la cible la plus petite atteinte jusqu'a présent
Entrées: operations, le tableau des opérations réalisées
Sorties: La distance entre la valeur cible et la meilleure valeur atteinte

```
1: Pour tout les nombres tirage[i] Faire
2:   Pour tout les nombres tirage[j] avec i! = j Faire
3:     Pour tout les 4 opérations Faire
4:       Sauvegarder l'état du tableau tirage
5:       Effectuer le calcul tirage[i] operation tirage[j]
6:       Mettre le résultat dans tirage[i]
7:       Mettre le dernier nombre tirage[nb − 1] à la place de tirage[j]
8:       Si Le resultat est plus proche de l'ancien plus proche Alors
9:         distancemin = cible-resultat
10:        Mettre les nombres utilises, l'operation realisee et le resultat dans le tableau operations
11:      Fin Si
12:      Si le résultat est la cible OU val=resolutionplusproche(tirage, nb-1, cible, distancemin, operations) est nul Alors
13:        //On a trouvé une solution, on met l'opération et les nombres dans le tableau operations
14:        Mettre les nombres utilisés, l'opération réalisée et le résultat dans le tableau operations
15:        Restaurer les valeurs du tableau tirage
16:        return 0 ;
17:      Fin Si
18:      //On a trouvé une solution plus proche mais pas exacte en utilisant ces operations
19:      Si abs(val) < abs(distancemin) Alors
20:        distancemin = resultat
21:        Mettre les nombres utilisés, l'opération réalisée et le résultat dans le tableau operations
22:      Fin Si
23:      Restaurer les valeurs du tableau tirage
24:    Fin Pour
25:  Fin Pour
26: Fin Pour
27: //Aucune solution possible si on arrive ici
28: return distancemin ;
```

4 Optimisations

Sur le principe, il n'existe pas de méthodes n'examinant pas toutes les solutions possibles. La complexité restera celle définie par l'équation **??**.

Mais il possible de diminuer le nombre de tests réalisés et donc le temps de calcul avec les remarques suivantes :

— les opérateurs + et $*$ sont commutatifs : $a + b$ et $b + a$ sont identiques ;

— la soustraction n'est pas possible si le résultat est négatif ou nul

— la division est une division entière. a/b suppose que a est plus grand que b et b doit diviser a

— l'addition ou la multiplication par une valeur nulle n'ont pas d'intérêt

— la multiplication et la division par 1 n'ont pas d'intérêt

Pour être exact, il existe aussi la possibilité de construire les résultats en réalisant d'abord toutes les opérations entre 2 nombres initiaux et en les conservant. Puis il faut chercher les résultats possibles avec 3 nombres initiaux, c'est à dire les résultats combinant 2 nombres avec un nombre. Puis il faut chercher les résultats possibles avec 4 nombres, c'est à dire les résultats à 3 nombres combinés avec un nombre ou les résultats à 2 nombres combinés entre eux. Puis il faut chercher les résultats possibles avec 4 nombres, 5 nombres, N nombres.

C'est la notion de programmation dynamique qui demande de conserver les essais intermédiaires (à 2 nombres, à 3 nombres, etc.). Cette solution prend donc beaucoup plus de mémoire et est sensiblement plus rapide que la version récursive proposée, surtout en cas d'échec, mais elle fait appel à des notions qui seront vues au semestre 2. Pour vous donner une idée de performances, elle est implantée par la commande `/users/prog1A/C/librairie/projetS12023/lcbfast.exe` des machines de Phelma.

5 Améliorations possibles

5.1 Affichage

Une version encore plus aboutie de l'affichage permet de supprimer les opérations inutiles.

Avec le tirage, 1000 7 50 6 2 3 5000 et la valeur 60 à trouver, il est probable d'obtenir la séquence d'opérations suivantes avec l'algorithme proposé :

1. 1000 + 7 = 1007
2. 1007 + 5000 = 6007
3. 3 + 2 = 5
4. 50 / 5 = 10
5. 10 * 6 = 60

Il est clair que les 2 premières opérations ne servent pas à obtenir le résultat, mais elles ont été générées obligatoirement pour arriver à la solution.

Dans ce cas, la solution consiste à partir de la dernière opération (10 * 6 = 60) donnant le résultat attendu (60) et à rechercher si une opération précédente sert à calculer un des opérandes (6 ou 10). Dans ce cas, l'opération qui obtient cet opérande l'opération 4 (50 / 5 =10) est utile pour calculer 10, tandis qu'aucune opération n'est utilisée pour calculer 6 qui est un des nombres du tirage initial. Il faut recommencer avec les opérations utiles i.e. l'opération 4 dans cet exemple. Pour calculer 10, on vérifie quelle opération sert à calculer les opérandes 50 et 5. L'opération 3 est utile pour calculer 5. On itère ce procédé, ce qui permet de savoir que les opérations utiles sont les opérations 5,4 et 3. On affichera uniquement ces opérations.

Pour faciliter les recherches, il est alors préférable de stocker les opérandes, opérations et résultats dans 4 chaînes différentes au moment de la recherche des solutions et non dans une seule chaîne.

6 Le projet

Le projet clôture le premier semestre. Il met en œuvre l'ensemble des concepts que vous avez vus tout au long du semestre : tableau, fichier, structure, allocation dynamique. Il ajoute une dimension : travailler en équipe, en utilisant bien sûr le gestionnaire de version git.

Ce projet se réalise donc en binôme, que vous devez former avant le début du projet. Lisez le sujet et préparez vos questions pour la première séance qui est encadrée par un enseignant. Vous gérez ensuite votre projet à 2 en autonomie et posez vos questions, de préférence sur l'outil `riot.ensimag.fr` aux enseignants. Vous déposerez sur `gitlab.ensimag.fr` votre code. Les 2 séances prévues dans l'emploi du temps ne sont pas suffisantes pour réaliser ce projet, qui nécessite du travail supplémentaire. Il n'y a pas de rapport à rendre, uniquement votre code, avec un fichier README qui indique comment exécuter le programme, ainsi que ce qui fonctionne ou non.

Ce projet permet de revoir la plupart des notions que vous avez abordées pendant le semestre. Une partie de l'examen écrit sera inspirée de ce projet.

6.1 Travail à réaliser

Vous devez donc réaliser 2 programmes qui résolvent les problèmes posés. Une comparaison des solutions est possible avec les exécutables que nous fournissons sur les machines de Phelma.

Chacun de ses programmes effectue les actions suivantes :

1. lire un fichier contenant un jeu ou effectuer un tirage aléatoire de n valeurs [5]
2. afficher le tirage et la cible à atteindre ,
3. résoudre le problème,
4. afficher la solution si elle existe,

Faites la première version et testez la, avant d'essayer les optimisations proposées. Il s'exécutera avec une commande du type : `./probleme1.exe fichier_jeu`

6.2 Fichiers de données et exécutables donnés

Nous fournissons quelques fichiers contenant des problèmes (tirage et valeur cible) dans le répertoire `jeux` (voir § **??**)

Pour chaque problème, une version exécutable `probleme1.exe, probleme2.exe` résolvant le problème posé est disponible sur les machines de Phelma dans le répertoire `/users/prog1a/C/librairie/projetS12023`. Ces programmes s'utilisent de la manière suivante :

— Sans paramètres, il demande le nombre de valeurs à tirer et la cible à atteindre au clavier.

— Avec 1 paramètre, il considère que c'est un fichier contenant un tirage et une cible.

— Avec 2 paramètres, le premier paramètre est le nombre de valeurs à tirer, le deuxième la cible à atteindre. Il effectue un tirage aléatoire, affiche ce tirage et donne la solution.

6.3 Gestion de projet

Lors de la gestion de projet, il est important de définir des jalons afin d'en suivre l'avancement. Ces jalons permettent de définir le travail que chaque membre de l'équipe doit réaliser et pour quelle date.

Voici quelques jalons ou étapes dont vous pourriez vous inspirer pour travailler :

1. Jalon 1 : définir les structures de données et les entêtes de fonctions,
2. Jalon 2 : allocation dynamique et libération mémoire, fonction de lecture dans un fichier et d'affichage à l'écran,
3. Jalon 3 : résolution récursive simple du problème 1

5. Limiter la valeur de n à 10

4. Jalon 4 : affichage ordonné,
5. Jalon 5 : résolution récursive du problème 2
6. Jalon 6 : optimisations proposées.

6.4 Rendu du projet

Le code sera déposé sur le dépôt gitlab que nous vous avons créé. Il comportera les fichiers sources `*.c` et d'entête `*.h`, ainsi qu'un fichier `Makefile` permettant de compiler le programme. Il comportera aussi un fichier README indiquant ce qui fonctionne et ce qui ne fonctionne pas dans ce projet, comment compiler et comment exécuter les programmes réalisés.

Tout plagiat, qu'il soit interne ou externe à Phelma, est bien sûr interdit et sera sanctionné.

6.5 Travailler en binôme

Quelques conseils pour travailler en binôme

— Commencez par analyser le problème posé, définissez les structures de données dont vous avez besoin et/ou complétez celles que nous vous suggérons, puis définissez les fonctions, leur rôle et leur prototype ensemble. Ensuite, répartissez vous les fonctions à écrire en indiquant dans quels fichiers elles se trouvent.

— Travaillez de manière incrémentale : écrivez une première fonction, compilez puis testez cette fonction avec un programme principal `main` spécifique à cette fonction. Déposez le fichier sur `gitlab.ensimag.fr`. Passez ensuite à une autre fonction.

— Utilisez les outils de debug pour traiter les cas de `segmentation fault`, dûs à des accès mémoire inadéquats : `gdb`, `ddd` et `valgrind` dont vous trouverez une initiation sur le site : http://tdinfo.phelma.grenoble-inp.fr/1AS1/site.

— Evitez de travailler à 2 sur les mêmes fichiers. Bien que cela soit possible, cela peut générer des conflits au moment de déposer votre code sur git, conflits qu'il faut ensuite gérer correctement. Cela demande un peu d'habitude avec `git`.

— Déposer régulièrement votre code sur gitlab, et faites aussi régulièrement un `git pull` pour récupérer le travail de votre binôme.