Le compte est bon - Mini projet de NF04 du semestre P22

Lien du projet sur GitHub : https://github.com/TeddyRoncin/NF04-LeCompteEstBon

Sommaire

1 Rappel du sujet

1.1 Le sujet

Le but du jeu est de trouver tous les nombres pouvant être créés à partir de 3 nombres saisis par l'utilisateur et des 4 opérations élémentaires : l'addition, la soustraction, la multiplication et la division.

Notons que la soustraction `a - b` ne peut être effectuée que lorsque `a >= b`. La division ne peut être faite que quand le résultat est entier

On ne peut pas utiliser plusieurs fois le même nombre, et on n'est pas obligés d'utiliser tous les nombres

1.2 Ce qu'il faut faire

Il faut créer ce programme dans les langages C et Python.

Nous ne pouvons pas créer de fonctions (sauf fonctions simples)

2 Notions utilisées

L'algorithme implémenté en Python se base sur la représentation des nombres en binaire.

On lira les bits de la droite vers la gauche. Ainsi, dans le nombre `01100101`, le 1er bit vaudra 1 et le 8ème vaudra 0

Il existe quelques opérateurs importants :

- `a << b` : le décalage vers la gauche. Décale `a` de `b` bits. Les bits rajoutés à droite valent 0. Les bits qui ne peuvent pas être stockés à gauche (par exemple, on ne peut pas stocker de 9ème bit dans un octet = 8 bits) sont effacés. Cela est équivalent à `(a x 2^b) % N`, où `N` est la valeur maximale codable (256 pour un octet)

  Exemple : `01011101 << 3 = 11101000`

- `a >> b` : le décalage vers la droite. Décale `a` de `b` bits. Les bits rajoutés à gauche valent 0. Les bits qui ne peuvent pas être stockés à droite (par exemple, on ne peut pas stocker de 0ème bit) sont effacés. Cela est équivalent à `E(a / 2^b)`, où `E` est la fonction partie entière

  Exemple : `01011101 >> 3 = 00001011`

- `a & b` : ET binaire. Agit comme un opérateur ET pour chaque bit des 2 nombres

  Exemple : `01011001 & 10011101 = 00011001` : seuls les bits 1, 4 et 5 valent 1 dans les 2 nombres

Cela nous permet de faire des opérations très utiles

- Récupérer le `n`-ème bit d'un nombre `a` : Il faut isoler ce bit. Pour cela, on place ce bit en première position (`b = a >> (n - 1)` : on soustrait 1 à n, car on commence à compter à partir de 1, mais la valeur nulle de l'opérateur est 0). Il faut ensuite faire en sorte que tous les bits soient égaux à 0, sauf le premier, de façon à isoler le bit (`b & 1`). Pour récupérer le `n`-ème bit du nombre `a`, on fera donc `(a << (n - 1)) & 1`

Nous appellerons un *swap* un échange de 2 valeurs dans un tableau

3. Programme en C

3.1 Approche

Le but est de lire un fichier, contenant toutes les façons de combiner les nombres entre eux.

3.2 Syntaxe du fichier de combinaisons

Le fichier s'appelle `combinations`. Il est localisé dans le dossier `/c/` La syntaxe est très simple :

- Les caractères `a`, `b` et `c` représentent respectivement les première, deuxième et troisième valeurs entrées par l'utilisateur.
- Les caractères `+`, `-`, `*` et `/` représentent respectivement une addition, une soustraction, une multiplication et une division.
- Les 7 caractères décrits dans les 2 premiers points ne doivent pas être séparés par des espaces
- Une ligne commençant par `;` est un commentaire et sera ignorée
- Le fichier doit se finir par une ligne vide

3.3 Lecture du fichier

Pour lire le fichier, on ne peut pas stocker tous les caractères dans un tableau, car on ne connaît pas encore la taille du fichier.

On peut parser le fichier au fur et à mesure que l'on le lit. Il faudra pour cela stocker le résultat courant du calcul ainsi que la dernière opération lue. Par exemple, si les 3 nombres choisis par l'utilisateur sont 1, 2 et 3, et que le calcul est `b/a*c`, on fera :

- Par défaut, le résultat vaut 0
- Par défaut, la dernière opération est une addition
- On lit le premier caractère, c'est un `b`. On ajoute donc le deuxième terme de la liste de nombre (2) au résultat, puisque la dernière opération était une addition. Le résultat vaut donc désormais 0 + 2 = 2
- On lit le deuxième caractère, c'est un `/`. La dernière opération est donc désormais une division
- On lit le troisième caractère, c'est un `a`. On divise donc le résultat par le premier terme de la liste de nombre (1), puisque la dernière opération était une division. Le résultat vaut donc désormais 2 / 1 = 2
- On lit le quatrième caractère, c'est un `*` . La dernière opération est donc désormais une multiplication
- On lit le cinquième caractère, c'est un `c` . On divise donc le résultat par le troisième terme de la liste de nombre (3), puisque la dernière opération était une multiplication. Le résultat vaut donc désormais 2 * 3 = 6
- On lit le caractère suivant. C'est un retour à la ligne, la ligne est donc finie. Le résultat final est donc 6

4. Programme en Python

4.1 Approche

Le but est de parcourir les nombres et de trouver quelle combinaison de nombres et d'opérations chaque nombre représente On fera 2 boucles principales, imbriquées les unes dans les autres. La première boucle comptera le nombre d'opérations que l'on fera (entre 0 et 2). La seconde sera la boucle qui permettra de parcourir les nombres

4.2 Représentation binaire des opérations

Une opération est définie sur 2 bits. Voici le tableau de représentation des opérations :

00	**01**	**10**	**11**
Addition	Soustraction	Multiplication	Division

4.3 Représentation binaire d'un arrangement de 3 nombres

Un arrangement de 3 nombres est défini sur 3 bits.

Tout d'abord, on stocke ces 3 nombres dans un tableau. L'ordre est important, il doit toujours être le même à chaque décoder d'un nombre

Si le premier bit vaut 1, alors on échange le premier et le dernier nombre du tableau de nombres

Si le deuxième bit vaut 1, alors on échange les 2 premiers nombres du tableau de nombres

Si le troisième bit vaut 1, alors on échange les 2 derniers nombres du tableau de nombres

Exemple :

Prenons le nombre 3 dont la représentation binaire est `011`. La liste de nombre est : `1, 2, 3`

Le premier bit vaut 1, on fait donc un *swap* entre la première et la dernière valeur de la liste. On obtient ainsi

la liste `3, 2, 1`

Le deuxième bit vaut 1, on fait donc un *swap* entre la première et la deuxième valeur de la liste. On obtient ainsi la liste `2, 3, 1`

Le troisième bit vaut 0, on ne fait donc pas de *swap*.

L'arrangement représenté par le nombre 3 est donc `2, 3, 1`

4.4 Décodage complet d'un nombre

On appelle `n` le nombre d'opérations

- Les `n`x2 derniers bits permettent de décrire les opérations
- Les 3 derniers bits permettent de décrire l'arrangement des nombres

Exemple :

Prenons `n` = 2, et essayons de décoder le nombre 105, dont la représentation binaire est `1101001`. Supposons que notre liste de nombre soit `1, 2, 3`

Puisque `n` = 2, on a 2 opérateurs, notre calcul sera donc de la forme : `(a <opérateur> b) <opérateur> c`

- Premier opérateur : on prend les 2 derniers bits, qui sont `01`. Ce code représente la soustraction, donc le calcul est de la forme `(a - b) <opérateur> c`
- Deuxième opérateur : on prend les 2 bits précédents, qui sont `10`. Ce code représente la multiplication, donc le calcul est de la forme `(a - b) x c`
- *swap* des première et dernière valeurs : le cinquième bit vaut 0, on ne fait donc pas de swap.
- *swap* des 2 premières valeurs : le sixième bit vaut 1, on fait donc un *swap* des 2 premières valeurs de notre liste. Celle-ci vaut désormais `2, 1, 3`
- *swap* des 2 dernières valeurs : le septième bit vaut 1, on fait donc un *swap* des 2 dernières valeurs de notre liste. Celle-ci vaut désormais `2, 3, 1`
- Calcul : en remplaçant `a`, `b` et `c` par les valeurs de la liste, on obtient le calcul `(2 - 3) x 1`. On doit d'abord calculer `2 - 3`. 2 étant inférieur à 3, on ne peut pas effectuer ce calcul, le calcul représenté par le nombre 105 n'est donc pas possible dans les règles que nous nous sommes fixées.

4.5 Redondance

La façon de stocker les arrangements contient de la redondance : il existe 3! = 6 arrangements possibles de 3 éléments. Nous stockons ces 6 arrangements sur 3 bits, c'est-à-dire 8 valeurs possibles. Il y a donc plusieurs valeurs qui donneront les mêmes résultats (2 valeurs redondantes). Celles-ci sont :

- `010` et `111`
- `011` et `110`

Nous pouvons remarquer qu'il est inutile de parcourir les nombres de `000` à `101`. On peut se contenter de les parcourir entre `000` et `101` (de 0 à 5 inclus).

5 Algorithme

Voici l'algorithme que j'ai utilisé en C. J'utilise 4 fonctions, 1 article et 1 syntaxe que je suppose définis :

- `Fichier` : un article représentant un fichier
- `OuvrirFichier` : un sous-algorithme permettant d'ouvrir un fichier. Il prend en paramètre le

chemin vers le fichier (relatif ou absolu. Dans cet algorithme, j'utilise un chemin relatif) et renvoie un fichier. Si le fichier n'existe pas, le programme crashe
- `FermerFichier` : un sous-algorithme permettant de fermer un fichier.
- Je suppose que l'on peut utiliser la fonction `Lire` de la manière suivante : `Lire(fichier!c)`, où `fichier` est une variable de type `Fichier` et `c` est une variable de type caractère. La fonction lit alors un caractère dans le fichier et avance le curseur de lecture de 1. Ainsi, après un deuxième appel de la fonction, le caractère suivant sera lu.
- `char2int` : retourne le code ASCII du caractère passé en paramètre
- `int2char` : retourne le caractère ASCII associé au nombre passé en paramètre

```
Programme LeCompteEstBon
Variables:
file : Fichier  // Le fichier que l'on lit
values : entiers[1...3]  // Les 3 valeurs de l'utilisateur
currChar : caractère  // Le caractère courant
fileEnd : booléen  // Si on a fini de lire le fichier. Si currChar == '\0', al
stop : booléen  // Calcul impossible ?
possibleResults : tableau[1...74] d'entiers  // Le stockage des différents rés
                                             // (par exemple, avec 1, 2 et 3,
                                             // 74 est le nombre de combinaiso
possibleResultsCount : entier  // Variable de compteur permet de savoir où on
result : entier  // Résultat du calcul
operation : entier  // La dernière addition lue
computation : tableau[1...5] d'entiers  // La trace de notre calcul. Les 5 ent
computationSize : entier  // La taille de notre calcul. C'est la somme du nomb
i : entier  // Variable compteur
value : entier  // Une valeur des valeurs entrées par l'utilisateur. Elle repr
found : booléen  // Variable booléenne qui dit si on a trouvé ou non le résult
Algorithme:
// On ouvre le fichier "combinations" en lecture
OuvrirFichier("c/combinations"!file)
// Entrée des 3 valeurs de l'utilisateur
Pour j allant de 1 à 3 par pas de 1 faire
  Ecrire("Entrez le caractère ", j!)
  Lire(clavier!values[j])
Fin Pour
fileEnd <- faux
possibleResultsCount <- 0
// Boucle principal : lit une ligne et calcule le résultat associé à cette lig
Tant que NON fileEnd faire
  stop <- faux  // Par défaut, on suppose que le calcul est possible
  result <- 0  // La valeur est modifiée au fur et à mesure que l'on lit la li
  operation <- 0  // Par défaut, c'est l'addition : quand on traitera le premi
  computationSize <- 0
  // Boucle qui lit un calcul. S'arrête quand le calcul est impossible ou quan
  Lire(file!currChar)  // On lit un premier caractère
  TantQue NON stop ET currChar != '\n' faire
    // Les caractères représentent les nombres du tableau "values"
    Si 'a' <= currChar ET currChar <= 'c' faire
      value <- values[char2int(currChar) - char2int('a')]  // "c - 'a'" retour
      // On regarde ce qu'il faut faire avec ce nombre. Par exemple, si c'est
      Si operation = 0 faire  // Addition
        result <- result + value
      Sinon si operation = 1 faire  // Soustraction
        // Il faut gérer le cas où on ne peut pas faire la soustraction
        // Si on ne peut pas la faire, le résultat ne sera pas utilisé, on peu
        result <- result - value
        Si result < 0 faire
          stop <- vrai
        FinSi
      Sinon si operation = 2 faire  // Multiplication
```

```
      result <- result * value
    Sinon si operation = 3 faire  // Division
      // Il faut gérer le cas où on ne peut pas faire la division
      // Si on ne peut pas la faire, le résultat ne sera pas utilisé, on peu
      Si result % value != 0 faire
        stop <- vrai
      FinSi
      result <- result / value
    FinSi
    // On se souvient de cette étape
    computation[computationSize] <- value
  Sinon  // Si ce n'était pas 'a', 'b' ou 'c', alors on interprète directeme
    Si currChar = '+' faire
      operation <- ADDITION
    Sinon si currChar = '-' faire
      operation <- SOUSTRACTION
    Sinon si currChar = '*' faire
      operation <- MULTIPLICATION
    Sinon si currChar = '/' faire
      operation <- DIVISION
    Sinon si currChar = '\EOF' faire
      // On est arrivés à la fin du fichier.
      // On arrête la boucle de lecture de ligne ainsi que celle de lecture
      fileEnd <- vrai
      stop <- vrai
    Sinon si currChar = ';' faire
      // C'est un commentaire. On arrête donc la ligne
      stop <- vrai
    Sinon faire:
      // Le symbol n'est pas reconnu
      Ecrire("Fichier corrompu : symbole inconnu : ", c!)
      fileEnd <- vrai
      stop <- vrai
    FinSi
    // On se souvient de cette étape
    computation[computationSize] <- char2int(c!)
  FinSi
  // Dans tous les cas, on a fait une étape en plus
  computationSize++
  Lire(file!currChar)  // On lit le caractère suivant
FinTantQue
// Si on a forcé la fin de lecture mais qu'on n'est pas à la fin du fichier,
// On lit donc jusqu'à la fin de la ligne
Si stop ET NON fileEnd faire
  TantQue currChar != '\n' faire
    Lire(file!currChar)
  FinTantQue
Sinon faire  // Si l'arrêt n'a pas été forcé (et donc que l'on n'est pas à l
  // On cherche "result" dans "possibleResults", pour savoir s'il faut l'aff
  found <- vrai
  Pour i allant de 1 à possibleResultsCount par pas de 1 faire
    Si possibleResults[i] = result faire
      found <- vrai
    FinSi
  FinPour
  // Si on n'avait encore jamais trouvé ce résultat, alors on stocke ce résu
  Si NON found faire
    possibleResults[possibleResultsCount] <- result
    possibleResultsCount <- possibleResultsCount + 1
    Ecrire(result, " ["!)
    // S'il y avait 2 calculs, on met des parenthèses, pour éviter d'écrire
    Si computationSize = 5 faire
      Ecrire("("!)
```

```
      FinSi
      Pour i allant de 1 à computationSize par pas de 1 faire
        // Si on est au 4ème caractère, alors "computationSize" vaut 5
        // (elle est forcément impaire, sauf si le fichier est corrompu. Nous
        // On est alors après le deuxième nombre, il faut donc fermer la paren
        Si i = 4 faire
          Ecrire(")"!)
        FinSi
        Si i % 2 faire  // C'est un caractère, il faut le re-transformer en ca
          Ecrire(int2char(computation[i]!)!)
        Sinon faire  // C'est un nombre, on l'affiche directement
          Ecrire(computation[i]!)
        FinSi
      FinPour
      Ecrire(" = ", result, "]\n", result!)
    FinSi
  FinSi
Fin TantQue
// On n'oublie pas de fermer le fichier
FermerFichier(file!)
Fin LeCompteEstBon
```