Introduction à la programmation en Brainfuck

Astremy (Lucie)

Table des matières

Introduction

Bonjour,

Si vous lisez ce cours, vous êtes sûrement curieux, avides de choses improbables, voir peut-être même un peu maso. Ça tombe bien, moi je le suis totalement.

Je suis ici pour vous apprendre les joies de ce langage ésotérique qu'est le brainfuck, et les possibilités qu'il propose, qui sont bien au-delà de ce que l'on pourrait penser à première vue. Selon vous, peut-on calculer des divisions, des puissances ou encore des racines carrées, en ne pouvant faire que additions et soustractions, et ne pouvant aller que de 0 à 255 ? Voire même un interpréteur du langage avec ce même langage ?!

C'est ce que je vais essayer de vous montrer dans cette petite introduction aux éléments de langage du brainfuck.

Qu'est-ce que le brainfuck

On peut tout d'abord, avant de commencer les programmes incompréhensibles, se demander ce qu'est ce langage.

Le brainfuck est un langage de programmation qualifié d'exotique qui a été créé par Urban Müller en 1993.

C'est un langage qui se veut le plus simple possible tout en restant turing-complet (c'est à dire qu'en théorie, on peut faire avec ce langage tout ce que l'on peut faire avec n'importe quel autre langage).

Je suppose que je n'ai pas besoin de vous expliquer sa signification. Contrairement à beaucoup de langages exotiques, le brainfuck n'a pas vocation à seulement être un langage "troll", mais de proposer réellement une structure cohérente et d'être utilisé dans certains domaines, comme la recherche scientifique par exemple.

Il y a quoi dans le brainfuck

On y vient !

Pour ceux qui n'aiment pas apprendre des trucs par coeur, vous êtes servis ! Ce langage est fait pour vous (à condition que vous ayez un niveau de logique hors du commun).

Pas de for, if, else, while, int, str, char ou n'importe quoi d'autre, il faudra **en théorie** n'apprendre que... 10 choses. 8 caractères et 2 spécificités. Voici donc le "TOP 10 DES CHOSES A SAVOIR EN BRAINFUCK (la 2 va vous étonner)" :

1. En brainfuck, tout est mémoire (OMG), ainsi, la mémoire est constituée d'un minimum de 30 000 cases (selon l'implémentation de base, cela peut évoluer selon les implémentations).

2. Une case contient une valeur entière qui ne peut aller que de 0 à 255 (sérieux ?!). Si l'on ajoute 1 à une case qui vaut 255, elle passera à 0 et inversement. On peut se sentir limité, mais on peut faire un programme qui va "grouper" les cases pour que l'on puisse sans problème atteindre des nombres bien plus haut (par exemple en groupant seulement 2 cases, on peut aller jusqu'à 65 535 !).
3. Le caractère "+" permet d'ajouter 1 à la case que l'on regarde.
4. Le caractère "-" permet d'enlever 1 à la case que l'on regarde.
5. Le caractère ">" permet de passer à la case suivante du tableau.
6. Le caractère "<" permet de passer à la case précédente du tableau.
7. Le caractère "[" permet de commencer une boucle. Cela ne lancera la boucle et n'exécutera le code dedans seulement si la valeur de la case que nous regardons est positive (pas égale à 0).
8. Le caractère "]" permet de finir une boucle. La boucle se relancera si la case qu'elle regarde est positive, sinon le code continuera normalement et ne recommencera pas au "[".
9. Le caractère "." (Waaaa, le point ! Vous vous rendez compte ?!) permet d'afficher dans la sortie le caractère ascii correspondant au nombre sur la case actuelle (par exemple une valeur de 65 sur la case vaudra "A" en ascii).
10. Le caractère "," permet de mettre dans la case courante le code ascii du premier caractère du stdin (l'entrée utilisateur).

Voilà, c'est tout ce que vous avez à savoir du langage, tous les autres caractères sont considérés comme des commentaires, on peut donc écrire librement au beau milieu d'un code brainfuck sans aucun risque.

Si vous n'avez pas tout compris au top, je vais les réexpliquer en montrant en même temps quand nous commencerons le développement.

Annexe avant de commencer le cours

Présentation du code

Pour que vous compreniez ce que je fais et comment je présente, il faut que je montre comment sera mis en forme le code.

Voici un exemple de code que je pourrais vous montrer :

```
+++++ [
    ->+++<
] >
```

Il sera parfois séparé comme ça en partie et indenté pour que vous puissiez vous y retrouver. Je mettrais souvent à la fin de l'exercice le code en une ligne sans commentaires si besoin.

Pour vous expliquer comment évoluera l'état de la mémoire, je le ferais sûrement par "stade" de cette façon :

```
[*0, 0, 0...]
[*5, 0, 0...]
[*4, 0, 0...]
[4, *0, 0...]
[4, *3, 0...]
.....
```

Une ligne équivaudra à une "étape" d'exécution de la mémoire. Des fois je sauterais des étapes et les ferais d'un coup (exemple : Ajouter 5 d'un coup quand on a 5 "+" de suite).

L'étoile quant à elle représente la case où l'on est actuellement.

Débug

Le débug en brainfuck est quelque-chose d'extrêmement compliqué, bien plus que dans beaucoup d'autres langages de programmation. Il faut pouvoir regarder à quel état en est la mémoire à certains instants, et en déduire la cause. Pour cela, je vous conseille ce site qui permet de mettre un programme et si besoin une entrée utilisateur, et de le simuler à une certaine vitesse, l'arrêter ou le reprendre.

Je vous conseille fortement de parfois mettre de côté ce cours et essayer par vous-même de faire un petit projet et du code, pour constater qu'une erreur est très vite arrivée et apprendre à résoudre vos problèmes dès le début de votre phase d'apprentissage, et de ne pas vous retrouver avec un code de 50 lignes à débugger.

Sandbox

Ici, vous pouvez tester vos programmes brainfuck. Cet outil ne permet pas de voir la mémoire pour débugger, mais permet au moins de pouvoir créer et tester les programmes de ce pdf.

Je parie que vous avez jamais vu de pdf qui permet d'exécuter du code :p

Petit disclaimer :

Cela ne fonctionne pas avec tout les lecteurs de pdf. Firefox par exemple n'y arrive pas.

Cela fonctionne cependant avec les navigateur chromium-based.

Entrée utilisateur :

Exécuter

Résultat :

Chapitre 1 : Utilisation basique des symboles

Commençons par voir les symboles du brainfuck et leur utilisation. Vu que chaque caractère du brainfuck a à peu près son "inverse", nous les verrons deux par deux.

Le plus et le moins

Le plus et le moins (`+` et `-`) sont les caractères principaux du brainfuck. Sans eux, rien ne serait possible, car ce n'est qu'à l'aide de la modification des valeurs de la mémoire que l'on peut faire de réelles choses. Sans cela, on peut uniquement se déplacer dans la mémoire.

Utilisation :

```
+++--
```

Regardons ensemble ce que fait ce programme basique :

```
[*0, 0, 0...]
[*1, 0, 0...]
[*2, 0, 0...]
[*3, 0, 0...]
[*2, 0, 0...]
[*1, 0, 0...]
```

Nous pouvons voir qu'il commence avec toutes les cases à 0 (nomal). Ensuite un `+`, donc il met la valeur 1 à la case actuelle, la première, encore un `+`, donc la case passe à 2, et un dernier `+` pour la mettre à 3. Ensuite un `-` qui la met à 2 et un autre `-` qui la remet à 1.

Notre premier programme commence donc par ajouter 3 et enlever 2. Si vous êtes malin, vous me direz "Hé Lucie, sinon on fait juste `+` et ça la met à 1, c'est pareil !" et je vous répondrais que vous avez raison. A eux seuls, le `+` et le `-` ne servent pas à grand chose.

Le supérieur et le inférieur

Les symboles supérieur (`>`) et inférieur (`<`) permettent de se déplacer dans la mémoire. Nous ne sommes plus limités une seule case. Voyons comment nous pouvons les combiner aux deux précédents symboles :

```
+++>
->
++<<
-
```

```
[*0, 0, 0, 0...]
[*3, 0, 0, 0...]
[3, *0, 0, 0...]
[3, *255, 0, 0...]
```

```
[3, 255, *0, 0...]
[3, 255, *2, 0...]
[*3, 255, 2, 0...]
[*2, 255, 2, 0...]
```

Le programme commence donc par mettre la valeur 3 sur la première case, puis se déplacer sur la 2eme qui enlève 1 donc la passe à 255 (parce que 0-1 = 255, on rappelle que une case n'a que une valeur stockée sur un octet), puis nous allons sur la 3eme case et mettons 2. Pour finir nous retournons à la première case et enlevons 1 ce qui la fait retomber à 2.

Arrivez-vous à suivre le fonctionnement du programme jusque-là ? Si vous avez du mal, essayez de le relire jusqu'à comprendre, j'ai essayée de le faire aller étape par étape le plus possible au début pour que ce soit logique et facile.

Les crochets

Les deux crochets permettent de faire des boucles. Les boucles sont un peu comme un "while case", pour ceux qui ont fait un peu de programmation. Elle exécute en boucle les instruction à l'interieur tant que à la fin de la boucle, le pointeur n'est pas sur une case avec la valeur 0.

Voici un peu comment l'utiliser :

```
+>
++>
+++<<
[
    >
]+
```

```
[*0, 0, 0, 0, 0...]
[*1, 0, 0, 0, 0...]
[1, *0, 0, 0, 0...]
[1, *2, 0, 0, 0...]
[1, 2, *0, 0, 0...]
[1, 2, *3, 0, 0...]
[*1, 2, 3, 0, 0...]
[1, *2, 3, 0, 0...]
[1, 2, *3, 0, 0...]
[1, 2, 3, *0, 0...]
[1, 2, 3, *1, 0...]
```

Le programme met les valeurs 1, 2 et 3 respectivement sur les 3 premières cases, puis retourne au début.

Arrive la boucle qui nous intéresse. Celle-ci est sur une valeur positive, donc s'active et se décale d'une case à droite. Vu qu'elle reste sur une valeur positive, elle se réactive et se décale à nouveau jusqu'à arriver à la 4eme case.

Cette dernière portant la valeur 0, la boucle se fini. La fin du programme consiste à mettre la valeur 1 à cette case.

Le but de la boucle est donc de se déplacer jusqu'à trouver une case vide ou elle met la valeur 1.

Le point et la virgule

Le point et la virgule servent à interragir avec l'entrée et la sortie utilisateur. Le point sert à afficher un caractère dans la sortie (La correspondance ascii de la valeur de la case).

Quand à elle, la virgule s'occupe de mettre dans la case courante la valeur ascii du premier caractère de l'entrée utilisateur. Ainsi, faire par exemple `,>,` va mettre dans la première et la deuxième case respectivement les deux premiers caractères de l'entrée utilisateur.

Imaginons l'exécution du code suivant avec l'entrée utilisateur `"ABD"`

```
,[
    +.>,
]
```

```
[*0, 0, 0, 0, 0...]
[*65, 0, 0, 0, 0...]
[*66, 0, 0, 0, 0...] > "B"
[66, *0, 0, 0, 0...]
[66, *66, 0, 0, 0...]
[66, *67, 0, 0, 0...] > "C"
[66, 67, *0, 0, 0...]
[66, 67, *68, 0, 0...]
[66, 67, *69, 0, 0...] > "E"
[66, 67, 69, *0, 0...]
```

La sortie du programme est donc `"BCE"`.

Vous pourrez aisément le deviner, le but de ce programme est de renvoyer la chaine initiale mais modifiée en remplaçant chaque caractère par le suivant dans la table ascii. Pour des lettres de l'alphabet, ça reviens à les remplacer par la lettre suivante dans l'alphabet (sauf pour le Z).

Ce programme commence par prendre le premier caractère entré par l'utilisateur, puis lance la boucle. La raison pour laquelle on ne lance pas directement la boucle est que si on le fait le programme va directement s'arreter sans lancer la boucle vu que le pointeur est sur une case avec 0 au démarrage.

Ensuite, il ajoute 1 à la case et affiche dans la sortie utilisateur le caractère ascii équivalent. Enfin, il se déplace d'une case et reprend un caractère de l'entrée utilisateur. Cela continuera ainsi jusqu'à ce que l'entrée utilisateur soit vide et que la virgule mette donc 0 dans la case et termine le programme.

Chapitre 2 : Les schémas classiques

Certains bouts de codes reviennent dans la plupart des programmes. Nous y retrouvons la structure du "Si", le code pour vider une case et pleins d'autres petites choses. Un code brainfuck est souvent constitué, comme n'importe quel programme dans un autre langage, simplement d'association de ces schémas.

Oui je vous ai dit que il ne fallait quasiment rien apprendre en brainfuck, je vous ai peut-être un peu menti... oups :p

Non en vrai l'essentiel est de les comprendre, car la structure peut être reproduite facilement une fois que l'on comprend le fonctionnement, et souvent les structures ne sont pas exactement les mêmes, on y apporte des modifications selon la structure de la mémoire ou les besoin d'un test.

Si à un moment vous ne comprennez donc pas une des structures, essayez de vous pencher dessus, de le comprendre ligne par ligne, et d'utiliser le site donné en annexe 1 pour visualiser la mémoire.

Les boucles mono-instruction

Nous allons commencer par nous focaliser sur des schémas très facile, de par le fait qu'ils ne contiennent que 3 caractères dont une boucle (donc deux caractères déjà).

Il y en as 3 : Deux à but de suppression, et un de déplacement.

- `[-]` : C'est sans doute le mono-instruction qui vous sera le plus utile. Si vous cherchez à comprendre son fonctionnement vous allez assez vite comprendre que son utilité est de purement et simplement vider une case. Elle peut permettre par exemple de vider une case de mémoire parce que on va faire d'autres opérations à cet endroit ensuite et on ne veut pas que les "restes" du précédent calcul puissent poser des problèmes.
- `[,]` : C'est surement celui que vous utiliserez le moins. Ce programme sert à vider l'entrée utilisateur de tout ce qu'il contient. Remarque : Il faut être sur une case avec une valeur positive pour lancer ce schéma, sinon il ne marchera pas. Sinon, vous pourrez utiliser sa variante : `+[,]`.
- `[>]` ou `[<]` : Ce schéma permet de se déplacer à droite ou à gauche jusqu'à trouver une case vide (avec 0). Cela peut vous permettre de, si vous avez une mémoire organisée, trouver par exemple la première case de libre, ou trouver la fin de votre mémoire. Cela peut être aussi utilisé avec deux déplacement voir plus au lieu d'un seul, avec `[>>]` par exemple pour chercher toutes les deux cases. Vous les utiliserez selon l'organisation de votre mémoire.

Les mouvements de cases

Le déplacement

Voici une partie très intéressante et qui est souvent mise de coté : Bouger une valeur d'une case à l'autre.

On part du principe que votre mémoire est de la forme `[*x, 0, 0...]`. Vous devez mettre la valeur de x (première case) sur la deuxième case.

Pour cela nous utiliserons une boucle. Le but sera de, à chaque tour de boucle, enlever 1 à la première case, et ajouter 1 à la deuxième, et cela tant que la première case n'est pas à 0.

Essayez un peu de chercher comment faire pour chaque structure de base que je vous apprendrais, parce que je vais salement tout vous spoiler et juste lire ce pdf ne vous rendra pas pro en brainfuck, pour progresser il faut expérimenter.

Voici le code que la logique exprimée ci-dessus donnerais (on va partir pour l'exemple de la mémoire que x vaut 3):

```
[->+<]
```

```
[*3, 0, 0...]
[*2, 0, 0...]
[2, *0, 0...]
[2, *1, 0...]
[*2, 1, 0...]
[*1, 1, 0...]
[1, *1, 0...]
[1, *2, 0...]
[*1, 2, 0...]
[*0, 2, 0...]
[0, *2, 0...]
[0, *3, 0...]
[*0, 3, 0...]
```

Ainsi, la valeur de x qui était en case 1, se retrouve en case 2.

La duplication

Maintenant, nous voulons changer de stratégie. Au lieu de vouloir simplement déplacer la valeur, nous voulons la dupliquer. Ainsi, nous voulons que la valeur de la case 1 se retrouve en case 1 ET 2.

Nous allons devoir cette fois décomposer ce problème en deux parties.

Vu que l'on ne peut pas déplacer une valeur sans la supprimer, on pourrait la déplacer sur la case 2 ET 3, puis déplacer la valeur de la case 3 vers la case 1. Le problème serait résolu.

Alors, comment on code ça ?

```
[
    -> On enlève 1 à la première case
    +> On met 1 à la deuxième case
    +<< On met aussi 1 à la troisième case
] Jusqu'à ce que la case 1 soit à 0
>> Ensuite on va à la case 3
[
    -<< On enlève 1 à la case 3
    +>> Et on remet 1 à la case 1
] Jusqu'à ce que la case 3 soit vide
```

Au final, cela donne "`[->+>+<<]>>[-<<+>>]`". Cette structure peut paraitre complexe, mais je vous promet qu'elle est simple par rapport à ce que nous allons faire après. Petit warning : La duplication de case nécessite donc 3 cases ! Pas seulement 2. Faites attention à avoir l'espace mémoire suffisant pour ne pas empiéter sur d'autres valeurs.

Les calculs

Il y a divers formes de codes pour les calculs. Nous allons voir les deux formes principales : L'addition, et la multiplication.

Imaginons que vous ayez une mémoire de la forme `[*x, y, 0, 0...]`, et que votre but soit d'additionner x et y. Comment vous y prendriez-vous ?

Avec un peu de logique, nous pouvons trouver la réponse. Il suffit en soit de "déplacer" la valeur de la case 1 sur la case 2. Elle s'aditionnera alors avec la valeur déjà présente sur la case, et on trouvera x+y. Ce calcul n'est donc qu'un déplacement, et nous trouvons assez facilement `[->+<]`.

Pour la multiplication, cela est plus compliqué. Pour multiplier x par une constante k, cela est facile.

Imaginons que l'on doive calculer x * k avec k = 3.

Cela va fonctionner à peu près pareil, avec un déplacement. Par contre, au lieu d'ajouter 1 à la deuxième case quand on enlève 1 à la première, on va ajouter k (dans notre cas 3) à la deuxième.

Ainsi, le code donne cela :

```
[->+++<]
```

Le code ne reste pas très compliqué. La difficulté surviens quand on essaye par contre de multiplier x par une autre variable y et non une constante que l'on peut hardcoder.

Imaginons une mémoire sous la forme `[x, y, 0, 0, 0...]`

Il faut mettre sur n'importe quelle case le résultat de la multiplication de x par y.

En toute logique et en réfléchissant un peu, on pourrait se dire que x * y n'est rien d'autre que x * (y * k) avec k = 1.

En effet, x*y*1 = x*y. Nous pouvons donc essayer d'implémenter cela. Pour ce faire, nous allons commencer une multiplication normale par une constante pour x, puis lorsque nous arriverons sur la case y, au lieu d'ajouter par une constante, nous allons remettre le même code de multiplication.

Voyons voir si cela marche. Prenons x = 2, y = 3.

```
[
    -> (on commence le code de la multiplication)
    [
        ->+< On refait le code de la multiplication, mais par 1
        (équivalent à une déplacement de case)
    ]
<]
```

```
[*2, 3, 0, 0, 0...]
[*1, 3, 0, 0, 0...]
[1, *3, 0, 0, 0...]
[1, *2, 0, 0, 0...]
[1, 2, *1, 0, 0...]
[1, *2, 1, 0, 0...]
[1, *1, 1, 0, 0...]
[1, 1, *2, 0, 0...]
[1, *1, 2, 0, 0...]
[1, *0, 2, 0, 0...]
[1, 0, *3, 0, 0...]
[*1, 0, 3, 0, 0...]
[*0, 0, 3, 0, 0...]
[0, *0, 3, 0, 0...]
[*0, 0, 3, 0, 0...]
```

On peut voir que cela n'a pas fait le comportement attendu. En effet, après le premier tour de boucle, la valeur de y de la deuxième case a été mise à 0, ce qui fais que l'on ne peut plus l'utiliser.

La solution (je vous invite à y réfléchir de votre coté avant de lire la fin de cette phrase), est d'utiliser la duplication de case pour "sauvegarder" la valeur de y tout en l'ajoutant à la case. Au lieu de dupliquer la valeur, puis de l'ajouter à la 5eme case par exemple, on va utiliser la propriété qui fait que bouger ajoute, et ne pas utiliser la valeur restante sur la 3eme case. A chaque tour de boucle, la valeur sera donc ajoutée à la case en la dupliquant.

Voici donc une version corrigée du code précédent :

```
[
    -> (on commence le code de la multiplication)
    [
        ->+>+<< On copie la valeur sur la 3eme et 4eme case
```

```
    ]
    >>
    [
        -<<+>> On remet la valeur de la 4eme case en 2eme case
    ]
    <<< On reviens à la première case
]
```

Le code n'est pas aussi évident que ça à faire, mais en s'y penchant un peu, on peut trouver.

Ce code étant plus compliqué que ceux vu au-dessus, je ne vous demande pas de vous y pencher et de le comprendre par coeur. Le principal est de comprendre son fonctionnement. On en reparlera dans un chapitre plus avancé.

Hé, vous êtes encore là ? Votre cerveau fonctionne toujours pleinement ? C'est pas facile hein.

Allez boire de l'eau, manger un petit truc, c'est la pause, avant de passer au chapitre suivant.

Pour vous féliciter d'être arrivés jusqu'ici, vous avez le droit à une petite anecdote :

Actuellement, je fait un peu moins de brainfuck. Ce n'est pas que j'ai perdu le gout pour le langage, c'est juste que je me suis un peu calmée. (Les emploi du temps, tout ça..)

Mais à un des moments ou j'en faisait activement, je n'hésitait pas à embeter certains amis pour qu'ils fassent du brainfuck avec moi.

Certains ont acceptés, et j'ai passée des nuits folles avec eux. Bon malheureusement, ils arrivaient souvent pas à suivre ma vitesse, mais je les adore quand même.

J'ai fait beaucoup de choses. J'ai commencée un interpréteur brainfuck **en brainfuck**, mais je n'ai jamais terminée les boucles. Et sinon je voulais faire un système de mémoire avec adressage, ça serait super !

En vrai, je vais surement le finir grâce au cours, et vous le faire faire en exo, si vous survivez jusque-là !

A l'heure ou j'écrit ces mots, nous sommes le 03/03/21, il est 1h30 du matin, et ça fait genre 3-4 jours que j'ai commencée le pdf, et j'en suis déjà au début du chapitre 5, c'est cool.

En tout cas merci de m'avoir écoutée, je vous retrouve peut-être plus tard pour d'autres anecdotes.

Chapitre 3 : Début des projets

Nous n'avons pas encore vu le si, mais je ne vais pas en parler dans les schéma classiques (même si je devrait), car ça commencerais à faire beaucoup durer. Nous le verrons donc en cours de route.

En attendant, il est tant de nous lancer dans la vraie, la dure, la programmation de projets n

Nous allons commencer par essayer d'afficher un caractère dans la console, pour voir le fonctionnement. Ensuite, nous essayerons de faire une addition entrée par l'utilisateur :p.

Affichage d'un caractère dans la console

Bon. Cela ne va pas être bien complexe, bien moins que la multiplication de x par y que nous avons vu au-dessus.

Pour commencer, il va falloir se renseigner sur le caractère que l'on souhaite afficher. Imaginons que l'on souhaite afficher le caractère "A".

On se souvient que le . permet d'afficher la correspondance ascii de la vlaeur de la case. Ainsi, pour afficher notre "A", il va falloir se renseigner sur la valeur ascii du "A". Bon, ça c'est simple, c'est 65.

Il faut donc mettre le nombre 65 sur la case que l'on veut faire afficher. On pourrait mettre en soit 65 +, mais ce serait un peu moche.

Cherchons plutot au niveau de la multiplication. On pourrait faire 13*5, mais on peut surement faire mieux. Vu que 65 est très proche de 64, on peut à la place faire 8*8 + 1.

Alors, comment ça se code ça ?

```
++++++++ On met 8 sur la case actuelle
[
    -> On enlève 1 et on se déplace
    ++++++++ On ajoute 8 (donc 8*8)
    < On retourne à la précédente case
]
>+. On reviens à la case avec 64 on ajoute 1 et on l'affiche
```

```
[*0, 0, 0, 0...]
[*8, 0, 0, 0...]
[7, *8, 0, 0...]
[*7, 8, 0, 0...]
[6, *16, 0, 0...]
[*6, 16, 0, 0...]
[5, *24, 0, 0...]
...
[*1, 56, 0, 0...]
```

```
[0, *64, 0, 0...]
[*0, 64, 0, 0...]
[0, *64, 0, 0...]
[0, *65, 0, 0...] > "A"
```

Si on veut par exemple mettre un "s" derrière, on récupère le code du s minuscule : 115.

On va essayer d'utiliser la case actuelle pour coder le s. Comment faire ?

On peut se dire que 115 c'est 58*2-1. Il suffit donc de baisser la valeur de 65 jusqu'à atteindre 58, puis multiplier par 2 et enlever 1.

Cela donne :

```
Précédent Code
++++++++[->++++++++<]>+.

------- On enlève 7
[
    -<++> On multiplie par 2 sur la case précédente (case 1)
]
<-. (on enlève 1 et on affiche)
```

```
[0, *65, 0, 0...] > "A"
[0, *58, 0, 0...]
[*2, 57, 0, 0...]
[2, *57, 0, 0...]
[*4, 56, 0, 0...]
...
[114, *1, 0, 0...]
[*116, 0, 0, 0...]
[116, *0, 0, 0...]
[*116, 0, 0, 0...]
[*115, 0, 0, 0...] > "s"
```

Ainsi, le code final met la chaine "As" dans la sortie utilisateur. En soit on pourrais écrire le texte que l'on veut de cette manière.

Si vous voulez, vous pouvez vous amuser à faire un "Hello World !" en brainfuck de cette manière.

Calcul avec l'entrée utilisateur

Bon, ça c'est un peu plus dur qu'un hello world.

Le but est que l'utilisateur entre deux chiffres (par exemple 3 et 5) donc "35", et que le programme lise ces deux nombres et renvoie la somme.

Pour des raisons de simplicités, on va supposer que l'on vie dans un monde parfait (ce qui est faux, un gars à la cheveulure orange en témoigne), et que

ce qui sera entré par l'utilisateur sera un chiffre compris entre 0 et 9, et que le résultat de l'addition sera inférieur à 10.

Bon, maintenant, comment le faire ? Déjà la première étape est de se demander comment convertir un nombre entré en nombre pour la case.

Pour cela, nous nous en referons encore à la table ascii, qui nous dit que "0" = 48 en ascii et que l'on passe par tout les chiffres pour avoir "9" = 57. Ainsi, pour passer du caractère au nombre, il suffit d'enlever 48.

Il faut enlever aux deux caractère pour avoir les deux chiffres correspondants.

le début de notre code sera donc :

```
,>, (on met les deux caractères dans la case 1 et 2)
>>
++++++++ On met 8
[-<+++++++>] On met 8*6 = 48 dans la 3eme case
<[-<-<->>] On enlève 48 aux deux cases
<< On reviens sur la première case
```

Ensuite, il suffit de faire l'addition(ça, on sais le faire). Pour terminer, il faut retransformer le résultat en chiffre.

Pour cela il suffit d'ajouter 48, que l'on aura par exemple sauvegardé quelquepart pour éviter de devoir recréer de 0 le nombre à chaque fois. On modifiera donc un peu le début du code pour ajouter une sauvegarde du 48.

Après cela, on affiche le chiffre dans la console et c'est terminé.

Le code final donne donc :

```
>,>, (on met les deux caractères dans la case 2 et 3)
>>
++++++++ On met 8
[-<+++++++>] On met 8*6 = 48 dans la 4eme case
<[-<-<-<+>>>] On enlève 48 aux deux cases et sauvegarde le 48
< On reviens sur la troisième case
[-<+>] On ajoute la 3eme case à la deuxième
<[-<+>] On ajoute la 2eme à 48
<. On affiche finalement le retour
```

Mettre le code dans la sandbox Y aller

Au final, ce code est assez long, mais n'a aucune difficulté particulière, il a suffit de décomposer le problème pour arriver à obtenir une solution viable.

Calcul avec l'entrée utilisateur version modifiée

Maintenant, au lieu de recevoir deux chiffres, il faut que l'on puisse mettre le nombre de chiffres que l'on veut. Pour les besoins du programme, on va imaginer que l'on ne peut recevoir que des chiffres entre 1 et 9 (plus 0). Par

exemple, avec "1132" il faut que le programme renvoie le chiffre "7" (1 + 1 + 3 + 2).

Ce sera votre premier exercice, je vous invite à essayer de le faire de votre coté.

Solution :

Déjà, il faudra utiliser une variante de la fonction pour vider l'entrée utilisateur. Au lieu de réécrire par-dessus l'entrée précédente, on va se décaler à chaque fois pour avoir en ligne tout les chiffres.

Ensuite, il faudra faire un peu la même chose, quelque-chose pour enlever 48 à toutes les cases. Il suffit de passer sur toutes les cases tant que l'on n'a pas la valeur 0 sur la case actuelle. A un endroit, il atteindra une case vide qui signifie la fin de l'entrée utilisateur (C'est pour ça que l'on ne peut utiliser "0" comme entrée, vu que la valeur de "0" - 48 fait 0, ça pourrait poser des problemes avec le programme).

Il suffit ainsi de les additionner toutes ensemble en chaine, puis d'y ajouter 48.

Je vais montrer la solution au programme, donc évitez de le regarder sans chercher de votre coté.

```
>> On commence par se décaler de deux cases
pour laisser la place pour la sauvegarde du 48

,[>,] On met les entrées utilisateurs jusqu'à ce qu'il n'y en ait plus

++++++ On ajoute 6 à la case

[->++++<] On met 6*4 = 24 sur la case suivante
On devra donc enlever 2 à chaque fois à la case pour enlever 48 en tout

>[  Tant que la case est positive (boucle de 24 tours)
    -<<[ On enlève 1 et va sur la première entrée utilisateur
        --< Tant que on est pas sur une case à 0 on enlève 2
            (au final cela fera 48)
    ]
    <++ On se décale sur la première case et on sauvegarde le 48
    >>[>] On reviens à la fin de l'entrée utilisateur
    > Et on reviens pour recommencer la boucle
]
```

Cela va permettre de mettre toutes les cases à leur bonne valeur, et la première à 48 pour la sauvegarde.

Maintenant, il suffit de faire l'addition, ce qui est facile.

```
<< On retourne à la fin de l'entrée utilisateur
<[  Tant que la case d'avant n'est pas à 0
```

```
    >[-<+>] On ajoute la case d'après
    << Et on recommence
]
La boucle va s'arreter quand on atteindra la case avant la sauvegarde du 48
>[-<<+>>] On as juste à déplacer la valeur obtenue sur le 48
<<. Et on peut afficher le résultat
```

Mettre le code dans la sandbox Y aller

Vous pouvez essayer d'exécuter le code, et vous verrez que cela marche. Nous l'avons fait !

Alors, est-ce que le brainfuck vous parait toujours aussi vide de possibilités.

Nous avons réussi sans aucun problème à faire un calcul en ne sachant pas le nombre de chiffres à additionner... Pourquoi on ne pourrais pas faire plus ?

Il est largement possible de faire des divisions, même à virgule ! Bon, avant ça, il faudra au moins voir les conditions.

Chapitre 4 : Les conditions

Et justement, quand on parle du loup ! Nous y voilà !

Vous voyez souvent des cours ou une condition est emmenée comme quelque-chose de compliqué ? Non ? Et bah le brainfuck ne déroge pas à la rêgle.

Faire une condition en brainfuck n'est pas excessivement compliqué, et le schéma n'est pas non plus énorme.

Bon par contre cette fois c'est vrai que vous le voyez pas souvent, un tuto qui commence par vous présenter le "n'est pas égal" (différent de) avant le égal.

if x != y

Bon. Pourquoi est-ce que on commence par le != et non pas par le == ?

La réponse est simple. Détecter une valeur positive est facile, une valeur nulle, un peu moins. Pour comparer les valeurs, on va soustraire l'une à l'autre et regarder le résultat. Si la case est nulle, x est égal à y, sinon, x est différent de y.

Faisons le :

```
[->-<]>[code[-]]
```

Donc, on commence par soustraire la première case à la deuxième, puis on va sur la deuxième. Si elle est positive, alors on fait "code" (code que vous voulez faire dans le cas ou x != y. Il doit forcément prendre en compte qu'il doit partir **et revenir** sur la case, puis on la met à 0 pour ne pas faire plus de tour de boucle).

Ainsi, que la case soit égale ou non, il finira sur la deuxième case, avec la valeur 0. La seule chose qui aura changé sera l'exécution ou non du code en question.

if x == y

Bon, pour cela, c'est un peu plus difficile. Comment peut-on détecter que le résultat de la soustraction **est** égal à 0 ?

Alors, pour cela, il y a plusieurs techniques. Deux principalement.

Je vais vous les présenter toutes les deux mais sachez que par défaut je préfère la deuxième car elle permet de garder le résultat original.

La première technique est de partir du !=.

Elle consiste juste à mettre 1 à la case qui suit, et l'enlever si la case est positive. Elle se fait donc comme ça :

```
[->-<]>>+<[>-<[-]]>[code-]
```

Ce code à l'avantage d'être court et de n'utiliser que peu de cases (3) tandis que la deuxième en utilise plus (5).

Quand je dit 5, c'est 5 cases "qui doivent être à une valeur". Mais par exemple 2 cases de la 2eme doivent être à 0, ce qui fait que en pratique on consomme 5 cases mais n'en utilise que 3.

Le principe d'avoir une case qui vaut 1 et une case à 0. Ensuite, quand la valeur est positive, on se déplace d'une case.

Ensuite, on se redéplace. Si on atteris sur une case valant 0 la valeur était positive donc on ne fait rien, et si la case vaut 1 elle était nulle, donc on exécute du code.

```
[->-<]>>>+<<[>]>>[code>]
```

Elle à l'avantage de ne supprimer ni le résultat, ni les cases réservés pour le test de l'égalité, on peut donc la réutiliser directement.

Une autre version existe, qui est une fusion entre la première et la deuxième, ne prenant que 4 cases (la dernière devant contenir 0), mais qui cependant efface le bit validateur à chaque fois (il faut donc juste garder le `>+<` dans le code, alors que dans le deuxième nous ne sommes pas obligé de le garder):

```
[->-<]>>+<[>-]>[code->]
```

Bon, on fait une petite pause. C'est fatiguant tout ça, non ? Avant de passer au plus dur de ce chapitre, on se calme l'esprit.

Vous n'avez pas encore le cerveau en compote ? Parce qu'il faut le garder frais, vous n'êtes pas au bout de ce pdf :p

Attendez de voir le chapitre 5, qui est sur la gestion de la mémoire. C'est là qu'il va falloir vous poser 10 minutes avant de coder et faire un schéma, parce que faut prévoir les choses.

Bon, en attendant, petite lucienecdote :

J'aime beaucoup l'algorithmie, et j'ai un très bon esprit logique.

J'ai même fait plusieurs concours ou j'ai eu des places pas dégeulasses.

Par contre, je n'ai jamais pris de cours d'algo ou quoi que ce soit du genre. Je trouve toujours cela "logique".

Je sais que sans mon esprit, je serait certainement pas aussi forte que je le suis aujourd'hui (même si je suis pas vraiment super forte, syndrome de l'imposteur tu connais).

Quoi ? Faire des petites pauses pour vous reposer l'esprit serait une excuse pour parler de ma vie ? Je vois pas de quoi vous parlez *tousse tousse* bon on disait `x >= y`. Pour la peine vous aurez pas le strictement supérieur.

if x >= y

Bon, là, cela deviens compliqué.

Comment peut-on tester si une valeur est supérieur à une autre ?

Toute tentative de le tester s'avère assez infructueuse, on ne peut pas s'y prendre de la même manière que on teste l'égalité ou l'inégalité. A moins que...

Si x >= y, alors à un moment de la soustraction y - x, y prendra la valeur 0..

Il faut donc, au lieu de tester la valeur 0 à la fin, le faire **pendant** la boucle.

Nous avons deux possiblités. Soit faire toute la soustraction, soit arreter la boucle dés que l'on sais si x >= y.

Codons ensemble la première version.

```
>>>+<<< On met la valeur du bit de test

[
    ->- On commence la soustraction
    [>] On se décale si ce n'est pas égal à 0
    >> [ Donc si la valeur est égale à 0
        On met ici le code à exécuter
    >] On reviens sur la case commune
    <<<< On reviens au début
]
```

Pour la deuxième, il ne faut changer que la fin.

```
>>>+<<< On met la valeur du bit de test

[
    ->- On commence la soustraction
    [>] On se décale si ce n'est pas égal à 0
    >> [ Donc si la valeur est égale à 0
        On met ici le code à exécuter
    <<<[-]>>>>] On va vider le reste du test
    <<<<
]
```

Mettre le code dans la sandbox Y aller

Vous pouvez utiliser la méthode que vous préférez, la deuxième méthode est plus longue mais la première permet de savoir de combien x est plus grand que y.

Si vous souhaitez faire y >= x, il suffit d'inverser les cases x et y, et je vous laisse trouver par vous-même comment faire le > strict.

Chapitre 5 : La gestion de la mémoire

Voici quelque-chose que nous n'avons pas vu jusqu'alors. La gestion de la mémoire.

Explication du fonctionnement et de la mise en oeuvre

Il y a quelques temps nous avions essayés de faire par exemple un programme pour additionner. Le probleme rencontré est que dans la version ou l'on pouvais mettre autant de nombre que l'on veut, mettre un 0 aurait par la suite cassé le programme.

Bon... Comment régler ça ?

Vous vous souvenez quand je vous ait parlé de bit validateur ? On peut en soit les réutiliser. Mettre des valeurs dans la mémoire qui seront définies comme étant positive (ou nulles), pour éviter tout malentendu dans le traitement des autres nombres.

Ainsi, au lieu de se déplacer tant que le nombre sur lequel on est est positif, on se déplacera tant que le bit qui dit "la il y a de la mémoire" sera positif. Cela n'a pas l'air de changer énormément, mais cela fait toute la différence.

Avec cette technique, on peut faire toutes les opérations que l'on veut sur les nombres, cela n'aura pas d'impact sur la bonne exécution ou non du code.

Regardons par exemple le code suivant :

```
>>,[
    >+>, On met l'input jusqu'à tromber sur du vide
]
<[
    <[-]< On vide tout
]
>>[
    <[
        ->>+<<  On déplace la valeur de la case à la case suivante
    ]>>+> On ajoute 1 et on va sur la case suivante
]
```

Dans ce programme, je supprime totalement les valeurs, je met tout à 0, et pourtant, je peut continuer à faire des opérations dessus.

Comme je le montre, je peut même compter le nombre de caractères qu'il avait alors que tout le contenu pur et dur de l'input à été supprimé.

Et ça, c'est grâce aux bits qui sont à 1 quand il y à une valeur.

Mieux ! Si on doit s'arreter à une valeur, pour x ou y raison, on peut mettre le bit validateur à 0. Ensuite, le code ira à droite tant qu'il ne tombe pas sur 0, et saura que c'est la valeur à laquelle on s'est arreté. Les possibilités sont bien plus grandes.

Projet : compteur du nombre d'occurence d'une lettre dans un texte

Bon, cette fois-ci c'est un gros morceau. Vous allez devoir mettre en oeuvre toutes vos connaissances, tout ce que vous avez appris jusqu'ici, pour pouvoir répondre à cet exercice.

```
Énoncé :

Vous devez, sans utiliser vos pieds de préférence,
développer un programme permettant de compter le nombre
d'occurence d'une lettre dans un texte.

L'entrée sera sous forme d'une chaine de caractère.
Le premier caractère sera le caractère à rechercher,
et le reste de la chaine sera le texte dans lequel rechercher le caractère

Exemple :
ssalut à tous les amis

Le caractère à chercher est le s, dans "salut à tous les amis"
Ainsi, le retour doit être le chiffre 4.

Nous imaginerons que le retour est un chiffre entre 0 et 9.

Good Luck have fun.
```

Bon, ne lisez pas cela si vous n'allez pas trouver ou n'êtes pas en vrai galère,je vais ici donner des pistes qui vont nous aider à résoudre notre problème.

Je suppose que récupérer les entrées utilisateur n'est pas trop difficile si vous avez bien lu le pdf jusqu'à maintenant, j'en ai déjà parlé.

Bon, maintenant, en rapport avec le titre du chapitre, nous devons chercher comment gérer la mémoire.

Pour cela, nous avons deux solutions.

Soit nous faisons une opération d'ensemble de x == y avec y chaque case de la mémoire en soustrayant x à toutes les cases, puis en testant quelles cases sont égales à 0, ce qui demande pas mal de place mémoire pour chaque valeur (4 cases), soit on déplace une par une chaque valeur dans une zone mémoire de "traitement" et teste chacune séparément l'égalité qui prend moins de cases mémoire pour chaque valeur (2 cases).

Avec ça, vous devriez être en mesure de réussir à faire le programme. Ce n'est pas si complexe. Ça n'est pas **facile** à faire, mais cela reste tout à fait abordable.

Dans la prochaine page, j'explique la résolution de l'exercice, alors ne passez pas si vous n'êtes pas sur d'avoir réussi !

Solutions : Compteur d'occurences

Donc, il parait que vous voulez voir la solution.. Est-ce que je vous fait confiance, ou je part du principe que vous êtes, comme ça va être le cas de 99% des gens ici, passé à cette page sans avoir fait l'exercice (je vous vois ! Retournez le faire, aller, pan pan cul cul).

Bon, je suppose que je n'ai pas le choix..

Voici la solution consistant à soustraire x à toute les cases puis tester celles-égales à 0 :

```
,>>> On met la valeur à compter
,[>>+>>,] On crée la zone mémoire pour chaque valeur
<< [<<<<] On retourne au début
<[
    ->>>>> On enlève 1 à x
    [<<->>>>>>] On enlève 1 à tout les autres
    <<<<[<<<<] On reviens au début
<]
>>>>>[ Pour chaque case rencontrée
    <<[>]>>[ Si la valeur dessus est 0
        -<<<<[<<<<]>+>>>[>>>>]+> On reviens au début et on ajoute 1 au compteur
    ]>>>] Et on passe à la case suivante
<<<<[<<<<] On remet le pointeur au début
++++++++[->++++++<] On ajoute 48 au compteur ("0")
>. On l'affiche
```

Suivie de la version qui utilise moins de cases mais déplace les valeurs pour les traiter :

[compteur, 0, valeur, 0, test, 0, char1, validateur1, char2...]

```
>>,>>>> On met la valeur à chercher
,[>+>,] On met tout les caractères
<[-<[-<[<<]<+>>>[>>]<]<[<<]<<<[->+<<+>]<[->+<]>>[->-<]>[<]<<[<<+>]>>>[-]>>>[>>]+<
<<<<++++++++[-<++++++>]<. a commenter
```

On peut aussi faire une solution qui combine les deux pour être plus efficace :

```
,>>,[>+>,]<[<<]<[->>>[<->>>]<<[<<]<]>>>[>>]<<[<[>-]>[-<<[<<]<+>>>[>>]>]<+<[-]<]++
```

Mettre le code dans la sandbox Y aller

Chapitre 6 : Exercice

Voici un petit exercice si vous vérifier les connaissances basiques :

QCM

1. Le brainfuck est :
Un langage de traitement de mémoire ◯
Un langage de gestion de base de données ◯
Un langage de programmation ◯

2. Avec on ne peut pas :
Rien ◯
Faire des fonctions ◯
Faire des sauts inconditionnels ◯

3. Comment fait-on une addition :
`[->+<]` ◯

`[->-<]` ◯

`[+>+<]` ◯

4. Le symbole "." permet de :
Mettre la valeur de la case - 48 dans la sortie ◯
Mettre l'ascii de la valeur de la case dans la sortie ◯
Mettre la valeur de la case dans la sortie ◯

5. On utilise la "gestion de la mémoire" principalement pour :
Pouvoir s'y retrouver globalement dans la mémoire ◯
Pouvoir gérer les nombres traités quoi qu'il arrive ◯
Accélérer la vitesse du programme ◯

6. Dupliquer une case consiste à :
Bouger la valeur sur les deux cases adjacentes avant d'en remettre une sur la première ◯
Mettre la valeur sur la case d'à coté sans changer la valeur sur la case actuelle ◯
Ne rien faire. Dupliquer une case est impossible en brainfuck ◯

Cliquez pour voir le résultat

Score : Non Calculé

Chapitre 7 : Les nombres

Comment ça les nombres ? On as pas tout vu déjà ?!

Et bah non, faut trimer, même si on pense maitriser un sujet, on est souvent loin en fait.

Récupérer un nombre de l'utilisateur

Bon, ça c'est pour du beurre, pour vous mettre en bouche de ce chapitre.

Vous vous souvenez de comment récupérer l'entrée utilisateur et l'exploiter quand il est sous forme de chiffres ? On l'a vu au chapitre 3 n

Bon au cas ou, faisons un rappel. Vu que le code ascii de 0 est 48, de 1 49...

Donc, il faut "enlever" 48 à la valeur pour avoir sa valeur réelle sur la case et non pas sa valeur ascii.

```
,>++++++++[-<------>]<
```

Ça, c'est fait. Maintenant ? Hmmm... C'est bien d'avoir la valeur réelle, mais ce n'est pas vraiment suffisant. Imaginons l'utilisateur entre une valeur qui n'est pas un chiffre !? Il faudrait le vérifier...

Dans ce cas, faisons le !

Il faut commencer par récupérer l'entrée utilisateur et enlever 48. Ensuite, il suffit d'enlever 10 en vérifiant avant à chaque fois que le nombre sur la case n'est pas égal à 0.

Si à un moment, le nombre sur la case est égal à 0, l'utilisateur à bien entré un nombre entre 0 et 9.

```
>,>>+< On met le chiffre en input
++++++++[-<------<+>>] On enlève 48
<<++[ Pour une boucle de 10
    ->[>]>>[ Si le nombre est égal à 0
        >>+< On incrémente une case plus loin
    ]<<<-< Quoi qu'il arrive on enlève 1 à la case et on continu la boucle
]
>>>++++++++[->++++++<]>. Affiche un booleen 0 ou 1
selon que le nombre soit un chiffre ou non
```

Nous avons donc un moyen de vérifier si un caractère est ou non un chiffre..

Nous ne l'utiliserons pas tant que ça, mais cela peut toujours servir un jour si vous ne souhaitez pas faire confiance à l'utilisateur (ne faites jamais confiance à l'utilisateur).

Mettre le code dans la sandbox Y aller

Utiliser des valeurs sur plusieurs cases

Le brainfuck, ne pouvant aller sur jusqu'à 255, on peut être limité assez vite.

C'est pour cela que on se dit, pourquoi ne pas stocker un nombre sur plusieurs cases ?!

En effet, rien que sur 2 cases, on passe la valeur maximale de 255 à $(256^2)-1$, à savoir 65 535, déjà beaucoup plus !

Et pareil, en soit nous pourrions stocker la valeur sur 3, voir 4, pour atteindre la valeur maximale d'un "int" dans les langages de programmation classiques, à savoir 4 294 967 295 (en non signé).

Par exemple, une mémoire comme ça : `[2, 34, 0...]` pourrait signifier "2 fois 256 + 34 = 546".

Bon, ça, c'est la théorie. Dans la pratique on peut se demander comment interagir avec des nombres comme ça.

On va partir sur une mémoire de 2 octets, comment faire des choses comme additions ou soustractions avec ?

Je vais vous montrer comment faire M

Imaginons que l'on a une addition de deux nombres à deux octets à faire, dans une base 256. Comment peut-on s'y prendre ?

Tout d'abord, nous commencerons par additionner les octets "majorants" (ceux les plus imporants) totalement normalement.

Ensuite, nous allons additionner les octets minorants. Seulement, si lors de l'addition la valeur 0 arrive sur l'octet, alors nous avons dépassé 255 et il faut ajouter 1 à l'octet majorant.

Cela donne ceci :

```
,>,>,>,>>+< On met les 2 doubles nombres
++++++++[-<[------<] On enlève 48 à chaque case (pour avoir la valeur du nombre)
    >>>>> Et on revient
]
<<<< On va sur l'octet majorant du premier nombre
[->>+<<] On l'additionne avec l'octet majorant du deuxième nombre
> On va sur l'octet minorant du premier nombre
[->>+ On ajoute 1 à l'octet minorant du deuxième nombre
    [>]>>[ Si la case est à 0
        <<<+>>>> On ajoute 1 à l'octet majorant du deuxième nombre
    ]<<<<< Et on revient
]
>>>>+++++++[-<++++++>]<[-<+<+>>]<<.>.
à la fin on affiche les valeurs des octets de résultat
```

Comment tester si le fait de dépasser la valeur de l'octet marche bien ? Bah en soit vous pouvez utiliser des valeurs qui sont juste en-dessous de 0 dans

le tableau ascii, par exemple "/", ".", "-"...

Par exemple `99//` va donner 98 (+255 255 = -1)

Pour la soustraction, c'est exactement pareil, sauf que au lieu de tester le 0 après avoir fait la soustraction pour l'octet minoritaire, il faut faire le test avant. Je vous laisse le soin de faire le code pour cela :p

Aller, on s'arrête quelques minutes ?

Le dernier chapitre n'est en soit pas très très compliqué si vous êtes arrivés jusque-là, mais il fallait bien aborder ce point quelque-part n

A l'heure ou vous lisez ce pdf il y aura sûrement la suite, mais personnellement, je ne sais pas encore. Est-ce que je vous parle enfin des divisions ? Est-ce que je vous fais la version avancée du chapitre sur la gestion de mémoire ? Ou peut-être encore autre chose ?

Bon, je suppose que vous savez ce qui suit... le moment où je raconte ma vie pour vous divertir xD

Alors... Mon premier prénom, vous devez le connaitre, c'est *Lucie*, mais est-ce que vous connaissez mon deuxième ? *Miline*. C'est pas du tout un prénom fréquent, n'est-ce pas ? Pourtant je l'aime beaucoup, je le trouve très joli. Il y a même une amie qui lui a trouvé un diminutif, *Mili* hihi n.

Ce qui est bien, c'est que j'ai repris un peu le brainfuck depuis que je rédige ce pdf, j'essaye de l'avancer petit à petit pour qu'il soit complet un jour. Actuellement, nous sommes le 24/03/2021, mais ça fait quand même pas mal de temps que je n'ai pas énormément bossé sur le pdf.. Quand je m'y mets ça avance rapidement, je vous promets monsieur le juge UwU.

Hé les gens... Je vous aime bordel ♡.

Bon, je m'égare, reprenons le cours..

Chapitre 8 : Les divisions

Enfin, nous y arrivons !

Normalement, si vous êtes arrivés jusqu'ici, vous devriez avoir aquis les compétences nécessaires pour ça !

En soit, ce n'est pas si dur, mais je ne voulais pas précipiter les choses. Il y à tellement de choses de base à voir avant, que je ne savais pas forcément comment le caser..

Bon, on disait, les divisions !

Explications

Comment allons-nous nous y prendre ? C'est simple (ou presque) !

Si on a par exemple n le nominateur, à diviser par d le dénominateur, comment pouvons-nous nous y prendre ?

En soit, il suffit d'enlever d à n jusqu'à ce que n soit inférieur à d !

Pour cela, on va sauvegarder d, puis l'enlever à n. Si dans la soustraction, n passe à la valeur 0, on a fini. Sinon, on continue jusqu'à ce que n ou d passe à 0. Si d passe à 0, on "charge la sauvegarde de d", et on ajoute 1 au compteur, qui dit a combien est égal n/d.

Pour l'instant, on ne va pas calculer la valeur avec une virgule, on va juste calculer la partie entière.

Bon le problème va être surtout de détecter la fin. Pour cela, on va tester après avoir enlevé 1 si n est à 0, et si oui, enlever la valeur de d pour finir vite la boucle.

Essayez sérieusement de le faire vous-même... s'il vous plait

Donc..

Code

Si on met en place ce que l'on a dit plus haut, cela donne ce code :

```
>>>+>>,>,> Met les valeurs
++++++++[<[------<]>[>]<-]
Convertis leurs valeurs ascii en décimales
>>-<< On démarre le compteur à moins 1
<[->>+<<] Déplace la valeur de d
<[ Tant que n n'est pas à 0
    >>>[-<+<+>>] On copie la sauvegarde
    >+ On ajoute à au compteur
    <<[->+<]< On remet la save
    [ Tant que d
        <[- On enlève 1 à n
```

```
            [<]<<[ Si n = 0
                >>>[-]+<<<<
                On va mettre d à 1 pour qu'il passe à 0 à la fin du tour
            ]>>
        ]>>[>]<- On reviens à d quoi qu'il arrive et enlève 1
    ]< Quand c'est fini on va vérifier la case n
]
<[-]>>>>[-]++++++++[->++++++<]>.
On ajoute 48 et on affiche
```

Ce code marche... presque !

Il y a juste un problème qui fait que quand l'on a une division parfaite (exemple : 9/3), cela donne 1 en moins...

Pour corriger cela, il suffit de tester si d est à 0 quand on voit que n est à 0.

Après débug, cela donne cela :

```
>>>+>>,>,> Met les valeurs
++++++++[<[------<]>[>]<-]
Convertis leurs valeurs ascii en décimales
>>-<< On démarre le compteur à moins 1
<[->>+<<] Déplace la valeur de d
<[ Tant que n n'est pas à 0
    >>>[-<+<+>>] On copie la sauvegarde
    >+ On ajoute à au compteur
    <<[->+<]< On remet la save
    [ Tant que d
        <[- On enlève 1 à n
            [<]<<[ Si n = 0
                >>>>>>+<<< On va ajouter 1 au compteur
                -[ Si d != 1
                    [-]>>>-<<<
                    On le vide et va enlever ce que l'on avait ajouté au compteur
                ]+<<<<
                On va mettre d à 1 pour qu'il passe à 0 à la fin du tour
            ]>>
        ]>>[>]<- On reviens à d quoi qu'il arrive et enlève 1
    ]< Quand c'est fini on va vérifier la case n
]
<[-]>>>>[-]++++++++[->++++++<]>.
On ajoute 48 et on affiche
```

Mettre le code dans la sandbox Y aller

Et voilà ! Bravo à vous si vous avez essayé/réussi de votre coté.

NB : Il est possible de le faire bien plus simplement, juste j'ai essayée de faire une version logique/didactique, mais bon, je sais pas si vous pourrez

comprendre facilement en lisant le code. Des versions beaucoup plus courtes existent et j'en parlerais peut-être plus loin dans les chapitres.

Sans tout les commentaires, il est quand même bien moins conséquent, je vous le dit '(même si il le reste).

Si vous voulez le modulo, vous pourrez simplement récupérer la valeur de d, la soustraire à la valeur de la sauvagarde de d, et de l'afficher... Bonne chance pour le faire **fui très vite** !

Chapitre 9 : Fin des bases, merci

Je tiens à vous dire bravo, si vous avez réussi à suivre et comprendre tout ce cours.

Vous avez aquis la plupart des bases nécessaires à créer et organiser un programme brainfuck.

Avec cela, vous êtes capables de vous débrouiller pour faire beaucoup de programmes différents, vous avez appris ce qui était **nécessaire**.

J'ai encore des choses à vous apprendre, bien sûr, mais je ne suis pas sûre que faire durer ce pdf sur des dizaines de pages de plus soit une bonne idée... imaginez des nouveaux avec 50 pages de pdf à lire sur un langage exotique, ils perdraient vite leur motivation..

En tout cas, merci de l'avoir lu, merci de vous être intéressé à ce langage, qui à mon sens est passionnant et très éduquant sur les principes de bases

Voici quelques projets faciles que vous pouvez faire pour vous entrainer :

- Le modulo
- Le carré, cube, la puissance x
- La factorielle
- Un jeu des batons
- Une calculatrice (avec que + et - par exemple, ou on peut genre rentrer "5-3+21" et ça fait le calcul)

J'ai deux choix :

Soit, à partir de maintenant, la suite du pdf est consacré à du "bonus", ce que je vais surement faire.

Ou sinon, si je veut continuer, je le fait dans un autre pdf spécialisé, pour ne pas faire peur aux débutants avec un pdf de 50 ou 100 pages alors que au final seulement 30 seront vraiment ce que je considère comme "nécessaire".

Quoi qu'il arrive, merci à tout les gens qui m'ont soutenu, qui m'ont aidés. Et aussi à vous, de m'avoir lu, et appréciée j'espère.

Merci,

Lucie