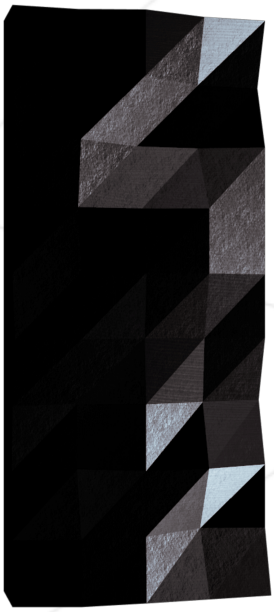

Push_swap

Parce que Swap_push c'est moins naturel

Staff pedago pedago@42.fr

Résumé: Ce projet vous demande de trier des données sur une pile, avec un set d'instructions limité, en moins de coups possibles. Pour le réussir, vous devrez manipuler différents algorithmes de tri et choisir la (ou les ?) solution la plus appropriée pour un classement optimisé des données.

Table des matières

Chapitre I

Préambule

- C

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

- ASM

```
cseg segment
assume cs:cseg, ds:cseg
org 100h
main proc
jmp debut
mess db 'Hello world!$'
debut:
mov dx, offset mess
mov ah, 9
int 21h
ret
main endp
cseg ends
end main
```

- LOLCODE

```
HAI
CAN HAS STDIO?
VISIBLE "HELLO WORLD!"
KTHXBYE
```

- PHP

```php
<?php
 echo "Hello world!";
?>
```

- BrainFuck

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

- C#

```
using System;

 public class HelloWorld {
    public static void Main () {
       Console.WriteLine("Hello world!");
    }
 }
```

- HTML5

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Hello world !</title>
    </head>
    <body>
        <p>Hello World !</p>
    </body>
</html>
```

- YASL

```
"Hello world!"
print
```

- OCaml

```
let main () =
    print_endline "Hello world !"

let _ = main ()
```

Chapitre II

Introduction

Le projet `Push_swap` est un projet d'algo simple et efficace : il faut trier. Vous avez à votre disposition un ensemble d'entiers, deux piles, et un ensemble d'instructions pour manipuler les piles.

Votre but ? Ecrire deux programmes en `C` :

- Un premier nommé `checker` qui prend des entiers en paramètres et qui lit des instructions sur l'entrée standard. Une fois ces instructions lues, `checker` les exécute puis affiche `OK` si les entiers sont triés, ou `KO` sinon.
- Un second nommé `push_swap` qui calcule et affiche sur la sortie standard le plus petit programme dans le langage des instructions de `Push_swap` qui trie les entiers passés en paramètre.

Facile ?

Et bien on va voir ça...

Chapitre III

Objectifs

Ecrire un algorithme de tri est toujours une étape importante de la vie d'un programmeur débutant car c'est souvent la première rencontre avec la notion de complexité.

Les algorithmes de tri, et leur complexité, font parti des grands classiques des entretients d'embauche. C'est donc l'occasion ou jamais de vous pencher sérieusement sur la question car soyez certains que cela vous sera demandé.

Les objectifs de ce projet sont rigueur, pratique du `C` et pratique d'algorithmes élémentaire. En particulier, la complexité de ces algorithmes élémentaire.

Trier des valeurs c'est simple. Les trier le plus vite possible, c'est moins simple vu que d'une configuration des entiers à trier à l'autre, c'est plus le même algo de tri qui est le plus efficace...

Chapitre IV

Consignes générales

- Ce projet ne sera corrigé que par des humains. Vous êtes donc libres d'organiser et de nommer vos fichiers comme vous le désirez, en respectant néanmoins les contraintes listées ici.
- Le premier exécutable doit s'appeller `checker` et le second `push_swap`.
- Vous devez rendre un `Makefile`. Ce `Makefile` doit compiler le projet, et doit contenir les règles habituelles. Il ne doit recompiler et relinker les programmes qu'en cas de nécessité.
- Si vous êtes malin et que vous utilisez votre biliothèque `libft` pour ce projet, vous devez en copier les sources et le `Makefile` associé dans un dossier nommé `libft` qui devra être à la racine de votre dépôt de rendu. Votre `Makefile` devra compiler la bibliothèque, en appelant son `Makefile`, puis compiler votre projet.
- Les variables globales sont interdites.
- Votre projet doit être en `C` et à la Norme. La norminette fait foi.
- Vous devez gérer les erreurs de façon raisonnée. En aucun cas vos programmes ne doivent quitter de façon inattendue (segmentation fault, bus error, double free, etc...).
- Vos programmes ne doivent pas avoir de `fuites mémoire`.
- Vous devez rendre, à la racine de votre dépôt de rendu, un fichier `auteur` contenant votre login suivi d'un '\n' :

```
$>cat -e auteur
xlogin$
```

- Dans le cadre de votre partie obligatoire, vous avez le droit d'utiliser les fonctions suivantes de la libc :
  - `write`
  - `read`
  - `malloc`
  - `free`
  - `exit`

- Vous pouvez poser vos questions sur le forum, sur slack, ...

Chapitre V

Partie obligatoire

V.1 Règles du jeu

- Le jeu est constitué de 2 piles nommées `a` et `b`.
- Au départ :
  - `a` contient un nombre arbitraire d'entiers positifs ou négatifs, sans doublons.
  - `b` est vide
- Le but du jeu est de trier `a` dans l'ordre croissant.
- Pour ce faire, on ne dispose que des opérations suivantes :

sa : `swap a` - intervertit les 2 premiers éléments au sommet de la pile `a`. Ne fait rien s'il n'y en a qu'un ou aucun.

sb : `swap b` - intervertit les 2 premiers éléments au sommet de la pile `b`. Ne fait rien s'il n'y en a qu'un ou aucun.

ss : `sa` et `sb` en même temps.

pa : `push a` - prend le premier élément au sommet de `b` et le met sur `a`. Ne fait rien si `b` est vide.

pb : `push b` - prend le premier élément au sommet de `a` et le met sur `b`. Ne fait rien si `a` est vide.

ra : `rotate a` - décale d'une position vers le haut tous les élements de la pile `a`. Le premier élément devient le dernier.

rb : `rotate b` - décale d'une position vers le haut tous les élements de la pile `b`. Le premier élément devient le dernier.

rr : `ra` et `rb` en même temps.

rra : `reverse rotate a` - décale d'une position vers le bas tous les élements de la pile `a`. Le dernier élément devient le premier.

rrb : `reverse rotate b` - décale d'une position vers le bas tous les élements de la pile `b`. Le dernier élément devient le premier.

rrr : `rra` et `rrb` en même temps.

V.2 Exemple

Pour illustration, trions une liste arbitraire pour constater l'effet de quelques instructions.

`0.` *Init a and b :*

a	b
2	
1	
3	
6	
5	
8	

`1.` *Exec sa :*

a	b
1	
2	
3	
6	
5	
8	

`2.` *Exec pb pb pb :*

a	b
6	3
5	2
8	1

`3.` *Exec ra rb : (equiv. to rr)*

a	b
5	2
8	1
6	3

`4.` *Exec rra rrb : (equiv. to rrr)*

a	b
6	3
5	2
8	1

`5.` *Exec sa :*

a	b
5	3
6	2
8	1

`6.` *Exec pa pa pa :*

a	b
1	
2	
3	
5	
6	
8	

Cet exemple trie les entiers de `a` en 12 instructions. Pouvez-vous faire mieux ?

V.3 Le programme "checker"

- Vous devez écrire un programme nommé `checker` qui prend en paramètre la pile `a` sous la forme d'une liste d'entiers. Le premier paramètre est au sommet de la pile (attention donc à l'ordre).
- `Checker` doit ensuite attendre de lire des instructions sur l'entrée standard, chaque instruction suivie par un '\n'. Une fois toutes les instructions lues, le checker va exécuter ces instructions sur la pile d'entiers passée en paramètres.
- Si après exécution la pile `a` est bien triée et la pile `b` est vide, alors `checker` doit afficher `"OK"` suivi par un '\n' sur la sortie standard. Dans tous les autres cas, `checker` doit afficher `"KO"` suivi par un '\n' sur la sortie standard.
- En cas d'erreur, vous devez afficher `Error` suivi d'un '\n' sur la **sortie d'erreur**. Par exemple si certains paramètres ne sont pas des nombres, certains paramètres ne tiennent pas dans un `int`, s'il y a des doublons, ou bien sûr si une instruction n'existe pas ou est mal formatée.

Grâce au programme checker, vous allez pouvoir vérifier que le programme que vous allez générer avec le programme push_swap trie bien la pile passée en paramètre.

```
$>./checker 3 2 1 0
rra
pb
sa
rra
pa
OK
$>./checker 3 2 1 0
sa
rra
pb
KO
$>./checker 3 2 one 0
Error
$>
```

V.4 Le programme "push_swap"

- Vous devez écrire un programme nommé `push_swap` qui prend en paramètre la pile `a` sous la forme d'une liste d'entiers. Le premier paramètre est au sommet de la pile (attention donc à l'ordre). Si aucun paramètre n'est passé, `checker` termine immédiatement et n'affiche rien.
- Le programme doit afficher le programme composé de la plus petite suite d'instructions possible qui permet de trier la pile `a`, le plus petit nombre étant au sommet.
- Les instructions doivent être affichées separées par un '\n' et rien d'autre.
- Le but est de trier les entiers avec le moins d'opérations possibles. En soutenance, nous comparerons le nombre d'instructions que votre programme a calculé avec un nombre d'opération maximum toléré. Si votre programme affiche un programme trop long, ou si bien sûr ce programme ne trie pas la liste, vous n'aurez pas de points pour ce test.
- En cas d'erreur, vous devez afficher `Error` suivi d'un '\n' sur la **sortie d'erreur**. Par exemple si certains paramètres ne sont pas des nombres, certains paramètres ne tiennent pas dans un `int`, ou encore s'il y a des doublons.

```
$>./push_swap 2 1 3 6 5 8
sa
pb
pb
pb
sa
pa
pa
pa
$>./push_swap 0 one 2 3
Error
$>
```

Pendant la soutenance, nous utiliserons vos deux programmes de la manière suivante :

```
$>ARG="4 67 3 87 23"; ./push_swap $ARG | wc -l
       6
$>ARG="4 67 3 87 23"; ./push_swap $ARG | ./checker $ARG
OK
$>
```

Si votre programme `checker` affiche `KO`, cela signifie que votre programme `push_swap` calcule un programme qui ne trie pas la liste et qui est donc faux.

Chapitre VI

Partie bonus

Les bonus ne seront évalués que si votre partie obligatoire est PARFAITE. Par PARFAITE, on entend bien évidemment qu'elle est entièrement réalisée, et qu'il n'est pas possible de mettre son comportement en défaut, même en cas d'erreur aussi vicieuse soit-elle, de mauvaise utilisation, etc. Concrètement, cela signifie que si votre partie obligatoire n'obtient pas TOUS les points à la notation, vos bonus seront intégralement IGNORÉS.

Le projet `Push_swap` se prête peu à la création de bonus de par sa simplicité. Voici tout de même quelques idées de bonus. Vous pouvez évidemment ajouter des bonus de votre invention, qui seront évalués à la discrétion de vos correcteurs.

- L'option `-v` peut afficher l'etat des piles à chaque coup.
- L'option `-c` peut faire afficher en couleur la dernière action.
- Tant que la partie obligatoire reste compatible, ajouter l'écriture et la lecture des insctructions depuis un fichier.

Chapitre VII

Rendu et peer-évaluation

Rendez-votre travail sur votre dépôt `GiT` comme d'habitude. Seul le travail présent sur votre dépot sera évalué.

Bon courage à tous et n'oubliez pas votre fichier auteur !