Chapter II

Introduction

The `Push_swap` project is a very simple and highly effective algorithm project: data will need to be sorted. You have at your disposal a set of int values, 2 stacks and a set of instructions to manipulate both stacks.

Your goal ? Write 2 programs in `C`:

- The first, named `checker` which takes integer arguments and reads instructions on the standard output. Once read, `checker` executes them and displays `OK` if integers are sorted. Otherwise, it will display `KO`.
- The second one called `push_swap` which calculates and displays on the standard output the smallest progam using `Push_swap` instruction language that sorts integer arguments received.

Easy?

We'll see about that...

Chapter III

Objectives

To write a sorting algorithm is always a very important step in a coder's life, because it is often the first encounter with the concept of complexity.

Sorting algorithms, and their complexities are part of the classic questions discussed during job interviews. It's probably a good time to look at these concepts because you'll have to face them at one point.

The learning objectives of this project are rigor, use of `C` and use of basic algorithms. Especially looking at the complexity of these basic algorithms.

Sorting values is simple. To sort them the fastest way possible is less simple, especially because from one integers configuration to another, the most efficient sorting algorithm can differ.

Chapter IV

General Instructions

- This project will only be corrected by actual human beings. You are therefore free to organize and name your files as you wish, although you need to respect some requirements listed below.
- The firts executable file must be named `checker` and the second `push_swap`.
- You must submit a `Makefile`. That `Makefile` needs to compile the project and must contain the usual rules. It can only recompile the program if necessary.
- If you are clever, you will use your library for this project, submit also your folder `libft` including its own `Makefile` at the root of your repository. Your `Makefile` will have to compile the library, and then compile your project.
- Global variables are forbidden.
- Your project must be written in `C` in accordance with the Norm.
- You have to handle errors in a sensitive manner. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc).
- Neither program can have any `memory leaks`.
- You'll have to submit at the root of your folder, a file called `author` containing your username followed by a '\n'

```
$>cat -e author
xlogin$
```

- Within your mandatory part you are allowed to use the following functions:
  - `write`
  - `read`
  - `malloc`
  - `free`

    - `exit`

- You can ask questions on the forum & Slack...

Chapter V

Mandatory part

V.1 Game rules

- The game is composed of 2 stacks named `a` and `b`.
- To start with:
  - `a` contains a random number of either positive or negative numbers without any duplicates.
  - `b` is empty
- The goal is to sort in ascending order numbers into stack `a`.
- To do this you have the following operations at your disposal:

sa : `swap a` - swap the first 2 elements at the top of stack `a`. Do nothing if there is only one or no elements).

sb : `swap b` - swap the first 2 elements at the top of stack `b`. Do nothing if there is only one or no elements).

ss : `sa` and `sb` at the same time.

pa : `push a` - take the first element at the top of `b` and put it at the top of `a`. Do nothing if `b` is empty.

pb : `push b` - take the first element at the top of `a` and put it at the top of `b`. Do nothing if `a` is empty.

ra : `rotate a` - shift up all elements of stack `a` by 1. The first element becomes the last one.

rb : `rotate b` - shift up all elements of stack `b` by 1. The first element becomes the last one.

rr : `ra` and `rb` at the same time.

rra : `reverse rotate a` - shift down all elements of stack `a` by 1. The flast element becomes the first one.

rrb : `reverse rotate b` - shift down all elements of stack `b` by 1. The flast element becomes the first one.

rrr : `rra` and `rrb` at the same time.

V.2 Example

To illustrate the effect of some of these instructions, let's sort a random list of integers. In this example, we'll consider that both stack are growing from the right.

```
---------------------------------------------------------------------------------------------------------
Init a and b:
2
1
3
6
8
5

- -
a b
---------------------------------------------------------------------------------------------------------
Exec sa:
1
2
3
6
8
5

- -
a b
---------------------------------------------------------------------------------------------------------
Exec pb pb pb:
6 3
5 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------
Exec ra rb (equiv. to rr):
5 2
8 1
6 3

- -
a b
---------------------------------------------------------------------------------------------------------
Exec rra rrb (equiv. to rrr):
6 3
5 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------
Exec sa:
5 3
6 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------
Exec pa pa pa:
1
2
3
5
6
8

- -
```

```
a b
-----------------------------------------------------------------------------------------------------
```

This example sort integers from `a` in 12 instructions. Can you do better ?

V.3 The "checker" program

- You have to write a program named `checker`, which will get as an argument the stack `a` formatted as a list of integers. The first argument should be at the top of the stack (be careful about the order). If no argument is given `checker` stops and displays nothing.

- `Checker` will then wait and read instructions on the standard input, each instruction will be followed by '\n'. Once all the instructions have been read, `checker` will execute them on the stack received as an argument.

- If after executing those instructions, stack `a` is actually sorted and `b` is empty, then `checker` must display `"OK"` followed by a '\n' on the standard output. In every other case, `checker` must display `"KO"` followed by a '\n' on the standard output.

- In case of error, you must display `Error` followed by a '\n' on the **standard error**. Errors include for example: some arguments are not integers, some arguments are bigger than an integer, there are duplicates, an instruction don't exist and/or is incorrectly formatted.

Thanks to the checker program, you will be able to check if the list of instructions you'll generate with the program push_swap is actually sorting the stack properly.

```
$>./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 The "push_swap" program

- You have to write a program named `push_swap` which will receive as an argument the stack `a` formatted as a list of integers. The first argument should be at the top of the stack (be careful about the order).

- The program must display the smallest list of instructions possible to sort the stack `a`, the smallest number being at the top.

- Instructions must be separaed by a '\n' and nothing else.

- The goal is to sort the stack with the minimum possible number of operations. During defence we'll compare the number of instructions your program found with a maximum number of operation tolerated. If your program either displays a list too big or if the list isn't sorted properly, you'll get no points.

- In case of error, you must display `Error` followed by a '\n' on the standard error. Errors include for example: some arguments aren't integers, some arguments are bigger than an integer, and/or there are duplicates.

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

During the defence we'll use your two programs as follow:

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

If your program `checker` displays `KO`, it means that your `push_swap` came up with a list of instructions that doesn't sort the list.

Chapter VI

Bonus part

We will look at your bonus part if and only if your mandatory part is EXCELLENT. This means that your must complete the mandatory part, beginning to end, and your error management needs to be flawless, even in cases of twisted or bad usage. If that's not the case, your bonuses will be totally IGNORED.

Find below a few ideas of interesting bonuses you could create. Some could even be useful. You can, of course, invent your own, which will then be evaluated by your correctors if they feel like it.

Because of its simplicity, creative bonuses are not the strengths of the `Push_swap` project. Here are some interesting, and useful, ideas though. You're still free to implement bonuses of your own creation which will be evaluated by your correctors.

- Debug option `-v` that can display the stack's status after each operation
- Colour option `-c` show in colours the last operation.
- As long as the mandatory part is compatible, you can add the reading and writing part of instructions from a file.