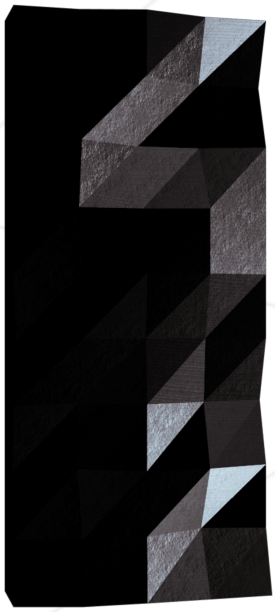

Push_swap

Because Swap_push isn't as natural

Summary:
This project will make you sort data on a stack, with a limited set of instructions, using the lowest possible number of actions. To succeed you'll have to manipulate various types of algorithms and choose the most appropriate solution (out of many) for an optimized data sorting.

Version: 6

Contents

Chapter I

Foreword

- 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
 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 ()
```

Chapter II

Introduction

The **Push swap** project is a very simple and a highly straightforward algorithm project: data must be sorted.

You have at your disposal a set of integer values, 2 stacks, and a set of instructions to manipulate both stacks.

Your goal? Write a program in C called `push_swap` which calculates and displays on the standard output the smallest program, made of *Push swap language* instructions, that sorts the integers received as arguments.

Easy?

We'll see...

Chapter III

Objectives

Writing a sorting algorithm is always a very important step in a developer's journey. It is often the first encounter with the concept of complexity.

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

The learning objectives of this project are rigor, use of `C`, and use of basic algorithms. Especially focusing on their complexity.

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 solution can differ.

Chapter IV

Common Instructions

- Your project must be written in C.
- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check and you will receive a `0` if there is a norm error inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`, use cc, and your Makefile must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To turn in bonuses to your project, you must include a rule `bonus` to your Makefile, which will add all the various headers, librairies or functions that are forbidden on the main part of the project. Bonuses must be in a different file `_bonus.{c/h}` if the subject does not specify anything else. Mandatory and bonus part evaluation is done separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` in a `libft` folder with its associated Makefile. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done

after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter V

Mandatory part

V.1 The rules

- You have 2 stacks named `a` and `b`.
- At the beginning:
  - The stack `a` contains a random amount of negative and/or positive numbers which cannot be duplicated.
  - The stack `b` is empty.
- The goal is to sort in ascending order numbers into stack `a`. To do so 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 last element becomes the first one.

rrb (`reverse rotate b`): Shift down all elements of stack `b` by 1.
The last 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 stacks grow from the right.

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

Integers from `a` get sorted in 12 instructions. Can you do better?

V.3 The "push_swap" program

Program name	`push_swap`
Turn in files	`Makefile, *.h, *.c`
Makefile	`NAME, all, clean, fclean, re`
Arguments	`stack a: A list of integers`
External functs.	• `read, write, malloc, free, exit` • `ft_printf and any equivalent YOU coded`
Libft authorized	`Yes`
Description	`Sort stacks`

Your project must comply with the following rules:

- You have to turn in a `Makefile` which will compile your source files. It must not relink.
- Global variables are forbidden.
- You have to write a program named `push_swap` that takes 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 separated by a '\n' and nothing else.
- The goal is to sort the stack with the lowest possible number of operations. During the evaluation process, the number of instructions found by your program will be compared against a limit: the maximum number of operations tolerated. If your program either displays a longer list or if the numbers aren't sorted properly, your grade will be 0.
- If no parameters are specified, the program must not display anything and give the prompt back.
- In case of error, it 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 evaluation process, a binary will be provided in order to properly check your program.

It will work as follows:

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

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

The checker_OS program is available in the resources of the project in the intranet.
You can find a description of how it works in the Bonus Part of this document.

Chapter VI

Bonus part

This project leaves little room for adding extra features due to its simplicity. However, how about creating your own checker?

Thanks to the checker program, you will be able to check whether the list of instructions generated by the push_swap program actually sorts the stack properly.

VI.1 The "checker" program

Program name	`checker`
Turn in files	`*.h, *.c`
Makefile	`bonus`
Arguments	`stack a: A list of integers`
External functs.	• `read, write, malloc, free, exit` • `ft_printf and any equivalent YOU coded`
Libft authorized	`Yes`
Description	`Execute the sorting instructions`

- Write a program named `checker` that takes 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, it stops and displays nothing.
- It will then wait and read instructions on the standard input, each instruction will be followed by '\n'. Once all the instructions have been read, the program has to execute them on the stack received as an argument.

- If after executing those instructions, the stack `a` is actually sorted and the stack `b` is empty, then the program must display `"OK"` followed by a '\n' on the standard output.
- In every other case, it 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 doesn't exist and/or is incorrectly formatted.

```
$>./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
$>./checker "" 1
Error
$>
```

You DO NOT have to reproduce the exact same behavior as the provided binary. It is mandatory to manage errors but it is up to you to decide how you want to parse the arguments.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter VII

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.

As these assignments are not verified by a program, feel free to organize your files as you wish, as long as you turn in the mandatory files and comply with the requirements.