Push_swap

Because Swap_push doesn't feel as natural

Summary:
In this project, you will sort data in a stack using a limited set of instructions, aiming to achieve the lowest possible number of actions. To succeed, you will need to work with various algorithms and choose the most appropriate one for optimized data sorting.

Version: 8.4

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 simple yet highly structured algorithmic challenge: you need to sort data.

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 C program called `push_swap` that calculates and displays the shortest sequence of Push_swap instructions needed to sort the given integers.

Easy?

We'll see...

Chapter III

Objectives

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

Sorting algorithms and their complexities are common topics in job interviews. Now is a great time to explore these concepts, as you will likely encounter them in the future.
The learning objectives of this project are rigor, proficiency in `C`, and the application of basic algorithms, with a particular focus on their complexity.

Sorting values is straightforward, but finding the fastest way to sort them is more challenging. The most efficient sorting method can vary depending on the arrangement of integers.

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.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc.) except for undefined behavior. If this occurs, your project will be considered non-functional and will receive a `0` during the evaluation.
- All heap-allocated memory must be properly freed when necessary. Memory leaks will not be tolerated.
- If the subject requires it, you must submit a `Makefile` that compiles your source files to the required output with the flags `-Wall`, `-Wextra`, and `-Werror`, using `cc`. Additionally, your `Makefile` must not perform unnecessary relinking.
- Your `Makefile` must contain at least the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To submit bonuses for your project, you must include a `bonus` rule in your `Makefile`, which will add all the various headers, libraries, or functions that are not allowed in the main part of the project. Bonuses must be placed in `_bonus.{c/h}` files, unless the subject specifies otherwise. The evaluation of mandatory and bonus parts is conducted separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` into a `libft` folder. 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 **does not need to be submitted and will not be graded**. It will give you an opportunity to easily test your work and your peers' work. You will find these 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 the assigned Git repository. Only the work in the Git repository will be graded. If Deepthought is assigned to grade your work, it will occur

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 number of unique negative and/or positive integers.
  - The stack `b` is empty.
- The goal is to sort the numbers in stack `a` in ascending order. To achieve 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 element or none.

sb (`swap b`): Swap the first 2 elements at the top of stack `b`.
Do nothing if there is only one element or none.

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
-----------------------------------------------------------------------------------------------------
```

The integers in stack `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 or 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 shortest sequence of instructions needed to sort stack `a` with the smallest number 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 should return to the prompt.
- In case of error, it must display `"Error"` followed by an '\n' on the standard error. Errors include, for example: some arguments not being integers, some arguments exceeding the integer limits, and/or the presence of 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.

V.4 Benchmark

To validate this project, you must perform certain sorts with a minimal number of operations:

- For **maximum project validation** (100%) and eligibility for bonuses, you must:
  - Sort **100 random numbers in fewer than 700 operations**.
  - Sort **500 random numbers in no more than 5500 operations**.
- For **minimal project validation** (which implies a minimum grade of 80%), you can succeed with different averages:
  - **100 numbers in under 1100 operations** and **500 numbers in under 8500 operations**
  - **100 numbers in under 700 operations** and **500 numbers in under 11500 operations**
  - **100 numbers in under 1300 operations** and **500 numbers in under 5500 operations**

  ...

All of this will be verified during your evaluation.

If you wish to complete the bonus part, you must thoroughly validate the project with each benchmark step achieving the highest possible score.

Chapter VI

Bonus part

Due to its simplicity, this project offers limited opportunities for additional features. However, why not create 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.

The bonus part will only be assessed if the mandatory part is perfect. Perfect means the mandatory part has been fully completed and functions without errors. In this project, this entails validating all benchmarks without exception. If you have not passed ALL the mandatory requirements,b your bonus part will not be evaluated at all.

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 or 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 from the standard input, with each instruction 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.

Chapter VII

Submission and peer-evaluation

Submit 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.

file.bfe:VABB7yO9xm7xWXROeASsmsgnYOoOsDMJev7zFHhwQS8mvM8V5xQQp
Lc6cDCFXDWTiFzZ2H9skYkiJ/DpQtnM/uZO