



## Push\_swap

Because Swap\_push doesn't feel as natural

### *Summary:*

*This activity 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 optimized data sorting.*

***This is a group activity to be completed by exactly 2 learners.***

*Version: 1.0*

# Contents

<b>I</b>	<b>Foreword</b>	<b>2</b>
<b>II</b>	<b>Common Instructions</b>	<b>3</b>
<b>III</b>	<b>AI Instructions</b>	<b>5</b>
<b>IV</b>	<b>Introduction</b>	<b>7</b>
<b>V</b>	<b>Objectives</b>	<b>8</b>
<b>VI</b>	<b>Mandatory part</b>	<b>9</b>
VI.1	Group project requirements . . . . .	9
VI.2	The rules . . . . .	9
VI.3	Algorithm requirements . . . . .	10
VI.3.1	Complexity model and operation constraints . . . . .	10
VI.3.2	Disorder metric (mandatory) . . . . .	10
VI.3.3	Required strategies . . . . .	12
VI.4	Example . . . . .	13
VI.5	The "push_swap" program . . . . .	14
VI.5.1	Usage examples . . . . .	15
VI.6	Performance Benchmark . . . . .	17
<b>VII</b>	<b>Readme Requirements</b>	<b>18</b>
<b>VIII</b>	<b>Bonus part</b>	<b>19</b>
VIII.1	The "checker" program . . . . .	19
<b>IX</b>	<b>Submission and peer-evaluation</b>	<b>21</b>

# Chapter I

## Foreword

Once upon a time in the mysterious lands of **Computer Science**, a certain [Donald Knuth](#)<sup>1</sup> popularized the *Big-O* notation to help us talk about how algorithms scale. Big-O is basically a polite way of saying: *"Your code will either run fast forever... or get so slow that you'll have enough time to make coffee, drink it, and grow old while waiting."*

Legend has it that in the early days, programmers didn't have Big-O. They just ran their code and, if it took too long, they blamed the *hardware*. Then Big-O arrived and ruined the fun by proving mathematically that sometimes your algorithm is just **bad** — no matter how many hamsters you put in the CPU wheel.

Why does this matter for `push_swap`? Because here you are, armed with two stacks and a bunch of awkwardly limited moves, trying to sort numbers faster than a sleep-deprived insertion sort. Big-O will help you face the brutal truth: a brilliant  $\mathcal{O}(n \log n)$  strategy will always outlive your clumsy  $\mathcal{O}(n^2)$  one when the input grows... unless, of course, you choose to ignore the math and watch your operation count skyrocket.

So as you design your algorithms, remember: Big-O is not here to scare you — it's here to prevent you from becoming the person who writes:

`pb; pa; pb; pa; pb; pa; ...`

10,000 times in a row and then wonders why the checker hates them. **Sort smart, not slow.**

---

*Big-O is not a popcorn size rating... but if it were,  $\mathcal{O}(n \log n)$  would still be the sweet spot.*

---

<sup>1</sup>Yes, the guy with the glorious beard and infinite patience.

# Chapter II

## 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 III

## AI Instructions

### ● Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

### ● Main message

- 👉 Use AI to reduce repetitive or tedious tasks.
- 👉 Develop prompting skills — both coding and non-coding — that will benefit your future career.
- 👉 Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- 👉 Continue building both technical and power skills by working with your peers.
- 👉 Only use AI-generated content that you fully understand and can take responsibility for.

### ● Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

## ● Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

## ● Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

### ✓ Good practice:

I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

### ✗ Bad practice:

I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

### ✓ Good practice:

I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

### ✗ Bad practice:

I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

# Chapter IV

## Introduction

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

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

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

Easy?

We'll see...

# Chapter V

## Objectives

The goal of this project is to make you discover [algorithmic complexity](#) in a very concrete way.

Sorting numbers is easy; sorting them *fast* with only two stacks and a handful of allowed moves is another story. Sorting a fully random list and sorting an almost sorted list are also two extremely different things.

You will quickly see how the choice of algorithm can make the difference between a quick win and an endless scroll of operations.

# Chapter VI

## Mandatory part

### VI.1 Group project requirements

- This project must be completed by exactly 2 learners working together.
- Both learners must contribute meaningfully to the project and understand all implemented algorithms.
- The repository must clearly indicate both learners' contributions in the README.md file.
- During the defense, both learners must be present and able to explain any part of the code.
- The project submission should include both learners' logins in the repository.

### VI.2 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 without any duplicate.
  - 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 two elements at the top of stack **a**.  
Do nothing if there is only one or no elements.

**sb** (**swap b**): Swap the first two 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 one.  
The first element becomes the last one.

**rb** (`rotate b`): Shift up all elements of stack `b` by one.  
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 one.  
The last element becomes the first one.

**rrb** (`reverse rotate b`): Shift down all elements of stack `b` by one.  
The last element becomes the first one.

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

## VI.3 Algorithm requirements

To enforce a rigorous understanding of algorithmic complexity (*time* and *space*), you **must implement four distinct sorting strategies** and integrate them into your `push_swap` program. Your program must be able to select a strategy at runtime based on the input configuration.

### VI.3.1 Complexity model and operation constraints

All strategies are implemented in C and must generate sequences of `Push_swap` operations to perform the sorting. This means:

- Your C algorithms analyze the input and generate the appropriate sequence of operations: `sa`, `sb`, `ss`, `pa`, `pb`, `ra`, `rb`, `rr`, `rra`, `rrb`, `rrr`.
- The output of your strategy is the sequence of these operations that sorts the stack.
- When you state a complexity class, it must reflect the cost **measured in number of Push\_swap operations generated**, not the theoretical complexity of a classical array-based algorithm.

### VI.3.2 Disorder metric (mandatory)

In this subject, **disorder** is a number between 0 and 1 that tells how far your initial stack `a` is from being sorted.

If the numbers are already in the right order, the disorder is 0. If they are in the worst possible order, the disorder is 1. Anything in between means your stack is partly sorted, but still messy.

To calculate it, you can think of looking at all the possible pairs of numbers in the stack. Each time a bigger number appears before a smaller one, that pair counts as a *mistake*. The more mistakes you have, the closer the disorder is to 1.

```
function compute_disorder(stack a):
    mistakes = 0
    total_pairs = 0
    for i from 0 to size(a)-1:
        for j from i+1 to size(a)-1:
            total_pairs += 1
            if a[i] > a[j]:
                mistakes += 1
    return mistakes / total_pairs
```

You must measure the disorder **before** doing any moves.

### VI.3.3 Required strategies

1. **Simple algorithm ( $O(n^2)$ ):**

Implement at least **one** baseline algorithm in the  $O(n^2)$  class. Examples include:

- Insertion sort adaptation
- Selection sort adaptation
- Bubble sort adaptation
- Simple min/max extraction methods

2. **Medium algorithm ( $O(n\sqrt{n})$ ):**

Implement at least **one** algorithm in the  $O(n\sqrt{n})$  class. Examples include:

- Chunk-based sorting (divide into  $\sqrt{n}$  chunks)
- Block-based partitioning methods
- Bucket sort adaptations with  $\sqrt{n}$  buckets
- Range-based sorting strategies

3. **Complex algorithm ( $O(n \log n)$ ):**

Implement at least **one** algorithm in the  $O(n \log n)$  class. Examples include:

- Radix sort adaptation (LSD or MSD)
- Merge sort adaptation using two stacks
- Quick sort adaptation with stack partitioning
- Heap sort adaptation
- Binary indexed tree approaches

4. **Custom adaptive algorithm (learner's design):** Design an **adaptive** strategy that selects different internal methods depending on the measured **disorder**. You are **not** constrained to any specific named algorithm; the internal techniques are entirely up to you. However, your design must respect the following **complexity targets** per regime (in the Push\_swap operation model):

**Low disorder:** if disorder  $< 0.2$ , your chosen method must run in  $O(n)$  time.

**Medium disorder:** if  $0.2 \leq$  disorder  $< 0.5$ , your chosen method must run in  $O(n\sqrt{n})$  time.

**High disorder:** if disorder  $\geq 0.5$ , your chosen method must run in  $O(n \log n)$  time.

You must document in your repository (e.g., `README.md`) the rationale for your thresholds, the internal techniques used in each regime, and a brief complexity argument (upper bounds) for time and space within the Push\_swap model.

## VI.4 Example

To illustrate the effect of some of these operations, 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 operations. Can you do better?

## VI.5 The "push\_swap" program

<b>Program Name</b>	push_swap
<b>Files to Submit</b>	Makefile, *.h, *.c
<b>Makefile</b>	NAME, all, clean, fclean, re
<b>Arguments</b>	stack a: A list of integers
<b>External Function</b>	<ul style="list-style-type: none"> <li>• read, write, malloc, free, exit</li> <li>• ft_printf or any equivalent YOU coded</li> </ul>
<b>Libft authorized</b>	Yes
<b>Description</b>	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 must write a program named **push\_swap** that takes as arguments:
  - The stack **a** formatted as a list of integers (the first argument is the top of the stack).
  - An optional **strategy selector**:
    - simple Forces the use of your  $O(n^2)$  algorithm.
    - medium Forces the use of your  $O(n\sqrt{n})$  algorithm.
    - complex Forces the use of your  $O(n \log n)$  algorithm.
    - adaptive Forces the use of your adaptive algorithm based on **disorder**. This is the default behavior if no selector is given.
- The program must display the smallest list of **Push\_swap** operations possible to sort stack **a**, the smallest number being at the top.
- Operations must be separated by a `\n` and nothing else.
- The complexity class claimed for each algorithm must be valid in this model.
- The strategy selection must work for **all valid inputs**. Any selector flag should work regardless of input size or disorder.
- 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: arguments that are not integers, integers outside the valid range, or duplicate values.
- Your binary must embed all four strategies (Simple  $O(n^2)$ , Medium  $O(n\sqrt{n})$ , Complex  $O(n \log n)$ , and Adaptive). The selected strategy name and complexity class must be available in --bench mode.
- The optional **benchmark mode** (--) must display, after sorting:
  - The computed **disorder** (% with two decimals).
  - The name of the strategy used and its theoretical complexity class.
  - The total number of operations.
  - The count of each operation type (**sa**, **sb**, **ss**, **pa**, **pb**, **ra**, **rb**, **rr**, **rra**, **rrb**, **rrr**).

The benchmark output must be sent to **stderr** and only appear when the flag is present.

### VI.5.1 Usage examples

*Notation: lines prefixed with [bench] represent messages printed by the optional benchmark mode (to **stderr**). The operation stream remains on **stdout**.*

```
$>./push_swap 2 1 3 6 5 8
ra
pb
rra
pb
pb
ra
pb
ra
pb
pb
pb
pa
pa
pa
pa
pa
pa
```

Default selection (--) and operation count:

```
$> ARG="4 67 3 87 23"; ./push_swap --adaptive $ARG | wc -l
13
```

Force the simple ( $O(n^2)$ ) strategy:

```
$> ./push_swap --simple 5 4 3 2 1
rra
pb
rra
pb
rra
pb
ra
pb
pb
pa
pa
pa
pa
pa
```

Force the complex ( $O(n \log n)$ ) strategy and verify with the checker:

```
$> ARG="4 67 3 87 23"; ./push_swap --complex $ARG | ./checker_linux $ARG
OK
```

push\_swap with a large input:

```
$> shuf -i 0-9999 -n 500 > args.txt ; ./push_swap $(cat args.txt) | wc -l
6784
$>
```

Run with benchmark enabled; hide operations and show only metrics:

```
$>shuf -i 0-9999 -n 500 > args.txt ; ./push_swap --bench $(cat args.txt) 2> bench.txt | ./checker_linux $(cat args.txt)
OK
$> cat bench.txt
[bench] disorder: 49.93%
[bench] strategy: Adaptive / O(n $\sqrt{n}$ )
[bench] total_ops: 7997
[bench] sa: 0 sb: 0 ss: 0 pa: 500 pb: 500
[bench] ra: 4840 rb: 1098 rr: 0 rra: 0 rrb: 1059 rrr: 0
```

Pipe operations to the checker while saving benchmark to a file:

```
$> ARG="4 67 3 87 23"; ./push_swap --bench --adaptive $ARG 2>
bench.txt | ./checker_linux $ARG
OK
$> cat bench.txt
[bench] disorder: 40.00%
[bench] strategy: Adaptive / O(n $\sqrt{n}$ )
[bench] total_ops: 13
[bench] sa: 0 sb: 0 ss: 0 pa: 5 pb: 5
[bench] ra: 2 rb: 1 rr: 0 rra: 0 rrb: 0 rrr: 0
```

Error management examples:

```
$> ./push_swap --adaptive 0 one 2 3
Error
$> ./push_swap --simple 3 2 3
Error
```

## VI.6 Performance Benchmark

To validate this project, you must achieve certain performance targets with a minimal number of operations:

- For **100 random numbers**, your program should use:
  - Less than **2000 operations** to pass (minimum requirement)
  - Less than **1500 operations** for good performance
  - Less than **700 operations** for excellent performance
- For **500 random numbers**, your program should use:
  - Less than **12000 operations** to pass (minimum requirement)
  - Less than **8000 operations** for good performance
  - Less than **5500 operations** for excellent performance

All of this will be verified during your evaluation using the provided checker.

# Chapter VII

## Readme Requirements

A `README.md` file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the project (peers, staff, recruiters, etc.) to quickly understand what the project is about, how to run it, and where to find more information on the topic.

The `README.md` must include at least:

- The very first line must be italicized and read: *This project has been created as part of the 42 curriculum by <login1>/, <login2>/, <login3>[...]]*.
  - A “**Description**” section that clearly presents the project, including its goal and a brief overview.
  - An “**Instructions**” section containing any relevant information about compilation, installation, and/or execution.
  - A “**Resources**” section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the project.
- ➡ Additional sections may be required depending on the project (e.g., usage examples, feature list, technical choices, etc.).

*Any required additions will be explicitly listed below.*

- A detailed explanation and justification of the algorithms selected for this project must also be included.



The choice of language is at your discretion. It is recommended to write in English, but it is not mandatory.

# Chapter VIII

## 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 operations 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, your bonus part will not be evaluated at all.

### VIII.1 The "checker" program

Program Name	checker
Files to Submit	*.h, *.c
Makefile	bonus
Arguments	stack a: A list of integers
External Function	<ul style="list-style-type: none"><li>• read, write, malloc, free, exit</li><li>• ft_printf or any equivalent YOU coded</li></ul>
Libft authorized	Yes
Description	Execute the sorting operations

- 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 operations 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 IX

## 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.

### Group project submission requirements:

- Both learners must be listed as contributors in the repository.
- The README.md must clearly document both learners' contributions.
- Both learners must be present during the peer-review defense.
- Each learner must be able to explain and defend any part of the code.

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behaviour change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific time frame is defined as part of the evaluation.

You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.