



# The Huffman Project

Using GitHub

Introduction To  
Software  
Engineering

- By -

Baptiste BALBI  
Flavien CHANDON  
Thomas CHANE-  
WAYE  
Myriam JOST  
Victor VALET

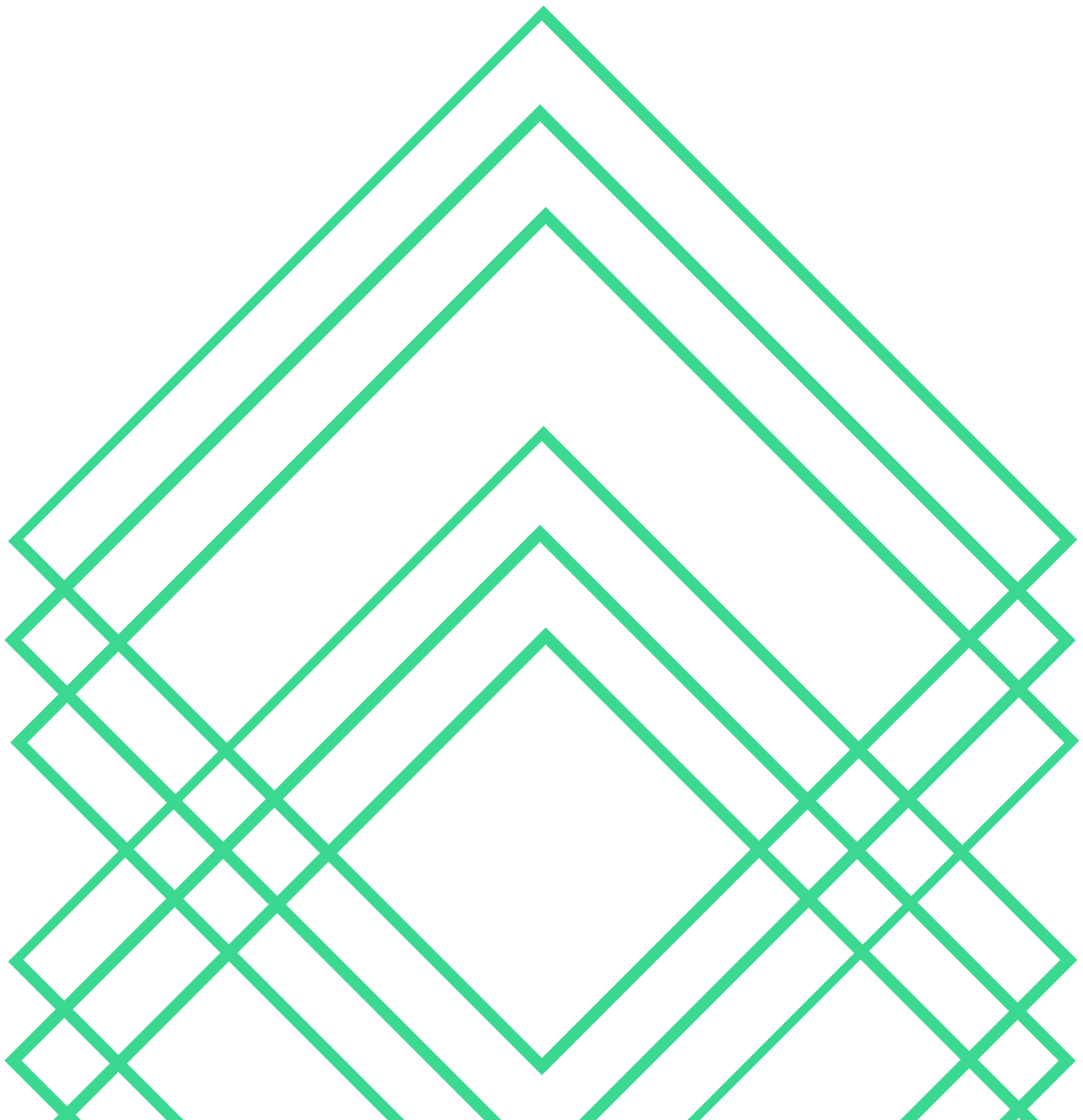
# Outline

1. Introduction
2. Gantt diagram and assignment of tasks
3. Code skeleton
4. Unit tests
5. Summary of our work
  - a. The use of GitHub and Doxygen
  - b. Main difficulties encountered
  - c. New perspectives
6. Conclusion
7. Bibliographical references
8. Appendix (User Guide)

# 1. Introduction

We have been given the subject of our “Algorithmics 3” project right before November holidays. The name of that project is the “Huffman Project” and its goal is the following : implement a compression algorithm, which means that we have, by groups of five, to code a program that must reduce the space taken by any information stored in a text file, and this without losing any data.

Moreover, we have had to learn in class and by ourselves how to use the platform GitHub in order to share and efficiently modify our program among the group. We have not been taught the method to use GitHub at the beginning of our project, so we started by sharing each other’s program on another platform, not at all as optimized as GitHub : Discord. In this report we will explain our work and how we organized ourselves.



## 2. Gantt diagram and assignment of tasks

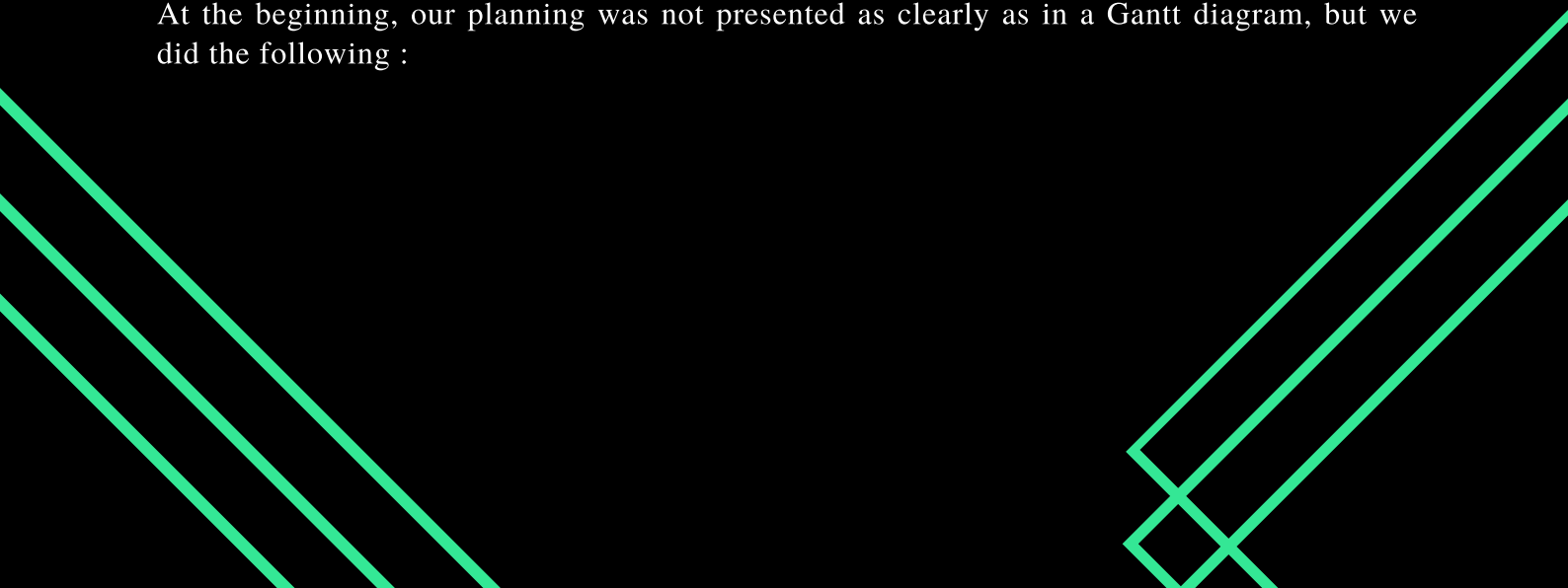
As we founded our group for the project, we realized there were differences in coding skills. Because of that we decided to try to all work together, some gave ideas and others would then program it. After reading thoroughly the project's required work, we organized ourselves by trying to plan what we should do and when it should be done. For that, we looked at the number of weeks we had, and we estimated the time every part should take for us to finish on time.

At first, we read the project and defined all the functions that would be necessary. As every part of the project depends on another, we could not really assign one group on part one and one group on part two. Although, we had time, so we decided to try and code the first part during at least a week. Victor and Myriam did it, and we then decided to work all together for the beginning of the second part. Thomas coded the list with the letters and occurrences, and then all of the group tried to work together in order to code the tree.

Flavien and Baptiste helped each other for that part, and then after implementing it in the current code they realized that there was a problem. We all tried to look at it in order to solve the issue, but with no success. Thomas then decided to modify his program in order to make it easier for us to find a solution for the tree, but as he did that, he found the solution and coded it.

Victor and Myriam tried to code the dictionary, but as Thomas' tree was hard to understand, they explained him what they tried to do, and he solved the issue. After that, Baptiste and Flavien created the encoding function, but were unsuccessful in coding the optional decoding function even though they put a lot of time in it. As we were late in our schedule and the function was optional, we decided to stop and only explain how we tried to program it.

At the beginning, our planning was not presented as clearly as in a Gantt diagram, but we did the following :



# Planning

## 3. Code skeleton

This program's objective is to compress information. As the input we have the information written in a text file called Texte.txt, that is read by the program. As the output we have multiple .txt files that are the following :

- Input.txt : the output is the number of characters that are inside the Texte.txt file.
- Bit.txt : the output is the content of the Texte.txt file but translated into its binary equivalent.
- Output.txt : the output is the number of characters (in this case integers) inside the Bin.txt file.
- Dictionary.txt : it contains each letter implemented in the tree, with their new binary code deducted from the Huffman tree.

Theme : Huffman project partie 1, 2 et 3

Module : one .h file named Structure.h and one .c file named Function.c

We have three files that are the following :

- Structure.h : contains our structures and the signature of our functions
- Function.c : contains our defined functions
- Main.c :

Our two .c files depend on Structure.h. In this program we also defined two new data types which are Node and Element.

```
typedef struct Node
{
    char letter ;
    int occ ;
    char bin[100];
    struct Node* left ;
    struct Node* right;
}Node;
```

```
typedef struct Element
{
    Node* El_letter ;
    struct Element* next ;
}Element;
```

## 4. Unit tests

As we programmed our different functions for our project, we have had to test them repeatedly in order to know if :

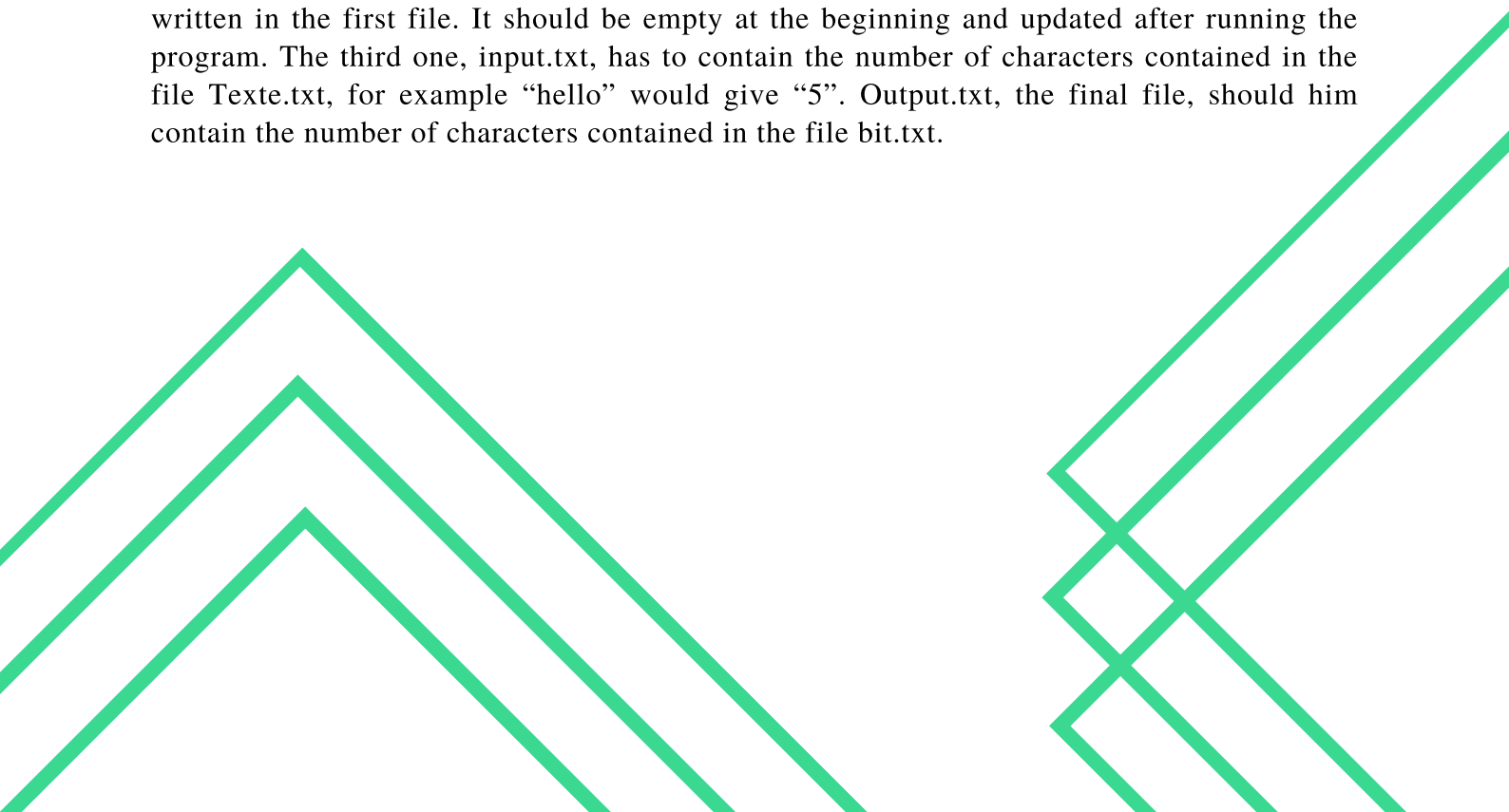
- There are syntax errors
- There are semantic errors that prevent the program from running
- There are semantic errors that prevent a part of the program from running entirely
- The program runs without any issue and the function can be implemented in the source code

We have had to program two main parts that are here going to be referred as the first part and the second part.

The first part consists of the two main following functions :



- `DecimalToBinary()` : The principle of this function is to take one letter from the text and turn it into its corresponding binary code. This binary code is stored in `char* octet` and we do that using the ASCII table.
- `TextToBinary()` : This function modifies the values of `char* binary` (location of the stored text translated in binary, first it is empty and after the function is compiled in the main, it has some spaces allocated where all of the binary translation of the text is stored, not for just one character), therefore when we call this function in the main, we want to print the variable `binary` which has changed and contains all the text translated in binary.

In order to test the above functions, we had to create four separated text files. The first is, of course, the one in which we write our input such as “hello” or any type of sentence and it is named `Texte.txt`. The second one, `bit.txt`, must contain the binary equivalent of the text written in the first file. It should be empty at the beginning and updated after running the program. The third one, `input.txt`, has to contain the number of characters contained in the file `Texte.txt`, for example “hello” would give “5”. `Output.txt`, the final file, should contain the number of characters contained in the file `bit.txt`.



However, as every computer programmer, we have encountered issues regarding these functions. First of all, when testing the function that writes the number of characters contained in the file `Texte.txt`, we noticed that not all characters were counted properly. We tested it for sentences in English and in French, and the French letters that possess accents such as “é” were counted twice instead of once. That test showed us that not all characters were supported by our `TextToBinary()` function. After struggling to find ways to solve this issue, we decided not to change our function.

Indeed, the solution for our problem was actually long and would have made the modification of the function too complicated, all that for only one type of letter that is not even a part of the English alphabet. Because of that last reason, we thought that it was actually not a problem as we are making a project in the English language. Then we tried to make sure that each binary number would be readable, so we added a space between them. Although, it revealed itself useless as we need to know the length of the text and its conversion to binary for the next part. Adding a space actually messed up everything.



When we finished the part 1 we moved on without changing anything. Afterwards, when we arrived at the end of part 2, we tested texts that were more complicated than a simple “hello” on one line. We found out that the `bit.txt` file showed only the first line written in `Texte.txt` when the text was one more than one line. In fact, the “`fscanf`” was picking the text until a certain point, corresponding to the “Enter” button equivalent.

That was a mistake, because the sample text we had (Alice in Wonderlands, chapter 1) contained a lot of “back to the line”. Since we discovered this issue the day before the defence, we didn’t have the time to make sure that it worked until the end of the file. But as it doesn’t prevent the main parts of the program from working, we implemented it anyway.

Now for the part 2 we had no real trouble with the list of occurrences, but more with the tree. We had to deal mainly with recursive functions leading to infinite loops. We had to agree on the structures used and how the function would run. Firstly, we created 3 data structures, but it was meaningless and time consuming to use them all. Instead, 2 out of 3 structures were merged and we had to rewrite parts of the code once or twice to adapt it to the new structures.



Secondly, after agreeing on the structures, we had to create two functions to sort a list of 'nodes' before the Huffman tree creation. One of the most difficult challenge was to agree on how to create the actual tree : some people wanted to create a tree using Queues / Stacks, while the others wanted to use a list of nodes. Finally, we used the following technique :

```
// Creation of the tree from a sorted list of node, return the last Node = Huffman tree//
Node* create_huffmanTree (Element **list) {
    if ((*list) == NULL) {
        printf ("Your list is empty \n") ;
        return NULL ;
    }
    else {
        Element* temp ; // Var that take list -> next

        while ((*list)->next != NULL) {
            temp = (*list)->next ;
            Node* new_node = Thrid_Node ((*list) ->El_letter, temp-> El_letter) ;
            (*list) ->El_letter = new_node ;
            (*list) -> next = temp ->next ;
            free (temp) ; // free the Element but not the Node
            sorted_list (list) ; // To know what happen in the funtion
            //display_list((*list));
        }

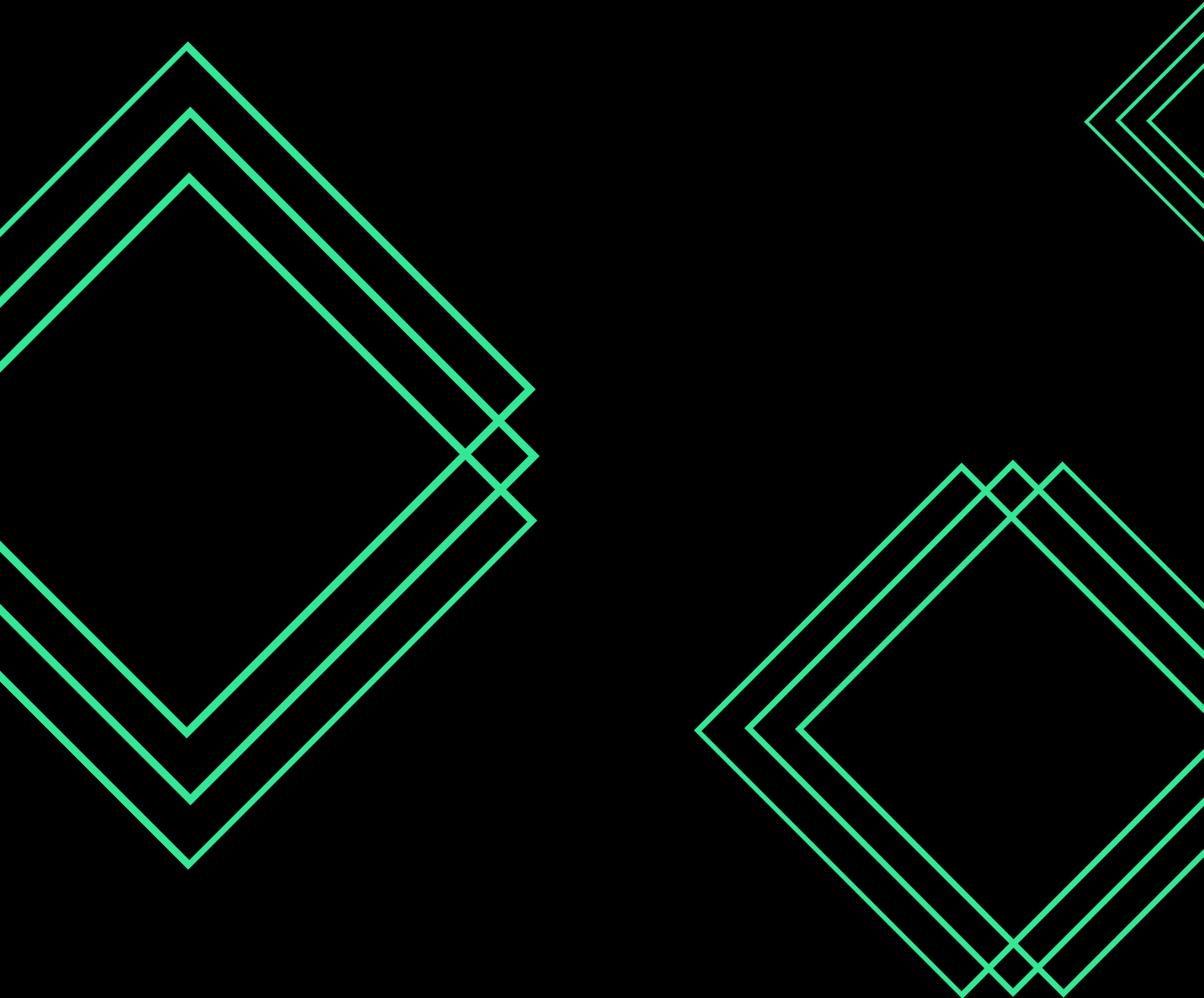
        Node* save = (*list) ->El_letter ; // to save the node in the Element
        free (*list) ; // free the last Element
        return save ; // return the Node in the last Element
    }
}
```

After that we were able to code the dictionary, however we encountered some issued when doing so. Before having the dictionary, we have right now we tested some functions with several methods. One of them was creating a new structure (AVL) using some premade functions like `memset()` or `memmove()`.

This method didn't work because obviously we had a function `sorted_list()` beforehand, and it worked the same has having an AVL. Also, these premade functions were really hard to understand and pretty new to us, so we didn't risk using them. In fact, we moved on to something else, simpler, which is described below.

The main difficulty resided in how we should travel in the tree and at the same time write in a new file. The trick here was to first establish a function that display the leaves of a tree, which is the Huffman tree, and then to establish a way to write in file with a recursive function. This part was done by using the mode “a+” (starts writing at the end of a file / creates a file if the file does not already exist) of file function. The problem with this method is was that we didn’t clean the already existing file. So, in the main we added a line that opens the file with “w” and then “cleans” the file or creates a new one.

Finally, we had some difficulties with the encoding function at first. For this function, we had to take what is inside our Texte.txt file by going through it with fgetc(). We took a character after another and translate each one of them into their respective compressed binary code. Simultaneously, we wrote them in our output.txt file. Although, one of the biggest problems was the fact that our list of letters did not exist anymore, as it was turned into our Huffman tree. To face this issue, we made a function to copy this list before its transformation.



# 5. Summary of our work

## a. The use of GitHub and Doxygen

As the project was given to us before we attended any “Introduction to Software Engineering” TD session, we began the project without using GitHub. Then when we had the first TD sessions, we all tried to understand how it works and how to efficiently put our programs in it. We learned the git commands and then we saw how to create and manage a repository, create branches, and make pull requests.

Some of us encountered issues with the TD session 2 and got a bit late, so we decided to not use a GitHub repository right after the session 2. We helped each other to understand how it works, and then we began using it.

Victor created the repository and named it “Huffman\_Project”, he then invited the four remaining group members. As the first part of the project was already finished, we uploaded it in the “main” branch. The following programs had all been uploaded in separate branches in order for us to check them before making a pull request.

As we did not make them technically “mergeable” at first, we had two branches with correct code that had to be merged with the first part. After checking that the different coded could be merged without causing any issue in the source code, we merged them and replaced the file from the main branch.



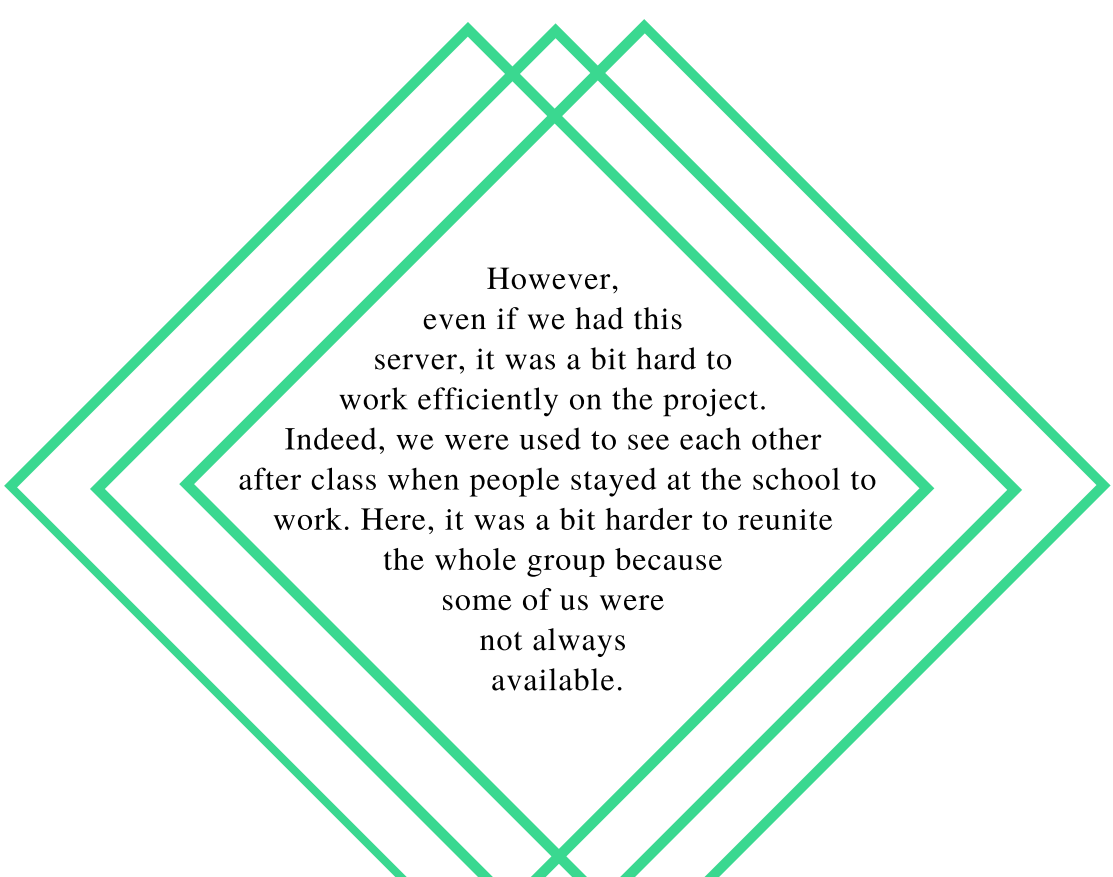
## b. Main difficulties encountered

The first problem we encountered was that we thought we would have time. We took a bit of time understanding exactly what was asked from us, and we inevitably got late on our initial planning. We already knew that we would not have enough time to finish the optimization of the program. As we were aware of the coding skills differences, we had to organize ourselves carefully so that everyone would participate. Those with coding difficulties would help the programmers on theoretical aspects for the hard functions.

Furthermore, we have had the misfortune of having to live a second containment, and even though we had one when we did the “project transverse” last year, it was still hard to be efficient. Because of this, we were forced to find a way, convenient for everyone, to share our documents such as working documents or codes in progress. We decided to share our files via the platform Discord, which we all use almost every day.

We created a server dedicated to the “Software Engineering” course, and then different “channels” to organize the server :

- General : to discuss about the project
- Notes and resources : to share documentation such as instructions, ideas on how to program a specific thing, and our programs
- Session planning : to ask group members if they are available on a particular date, in order to meet and progress on the project



However,  
even if we had this  
server, it was a bit hard to  
work efficiently on the project.  
Indeed, we were used to see each other  
after class when people stayed at the school to  
work. Here, it was a bit harder to reunite  
the whole group because  
some of us were  
not always  
available.

## c. New perspectives

It was difficult for us to use GitHub at first because we did not know this platform before this semester. We had to learn how to use it little by little which resulted in us beginning to be more efficient with it. We still have trouble using it as a group because we are not used to it, but we will obviously use it for all our next projects, such as our transverse project of next semester.

Before using that, we were using Discord to share our programs and meet as a group. Sharing codes is now done a more efficient way, but we all agree on the fact that Discord is really helpful to meet. We have also used Trello in order to stay organized, explain things to each other and make our report.

Now, regarding the planned organization of this project (cf. Gantt diagram), we did not manage to stick to it for many reasons :

- Firstly, the project was given to us a long time ago and sometimes we would think that we were ahead on time when we were not. By thinking that, we became late in our project and had to try and quickly do the part 3.
- Also, this point is more external to the project, but the fact that all the mid-term exams were postponed to the same week of the project delivery did not help us at all. We were supposed to have them spread over almost a month and it also made us think that we would have time for the project.

## 6. Conclusion

This project was really interesting to work on. Indeed, we learned more about project management and we have understood the process behind file compression. Using GitHub gave us a great example of group programming and we learned how to use it. We used tools that were new for us and we developed skills such as team cohesion, we also separated tasks between us to optimize our work in terms of spend time and quality. Now, for future projects, we will be more productive and efficient in our way to code and share the result between group members.

# 7. Bibliographical references

The use of GitHub :

<https://www.edureka.co/blog/how-to-use-github/>

The software architecture :

[https://en.wikipedia.org/wiki/Software\\_architecture](https://en.wikipedia.org/wiki/Software_architecture)

<https://www.techopedia.com/definition/3843/module>

The unit tests :

[https://en.wikipedia.org/wiki/Unit\\_testing](https://en.wikipedia.org/wiki/Unit_testing)

[https://en.wikipedia.org/wiki/Test-driven\\_development](https://en.wikipedia.org/wiki/Test-driven_development)

[https://en.wikipedia.org/wiki/Regression\\_testing](https://en.wikipedia.org/wiki/Regression_testing)

The Huffman tree :

[https://en.wikipedia.org/wiki/Bubble\\_sort](https://en.wikipedia.org/wiki/Bubble_sort)

[https://en.wikipedia.org/wiki/Insertion\\_sort](https://en.wikipedia.org/wiki/Insertion_sort)



# Appendix (User Guide)

In order to use our program, the first step is simple : write any sentence you want in the Texte.txt file. Then, after opening the folder in your API you must run the program. To do so, in Visual Studio Code, you must write two sentences in your terminal. Those are the following :

```
gcc -Wall -o toto *.c
.\toto.exe
```

After doing so you will see many information that is not relevant in your terminal. There will be our sorted list, our copied list and our Huffman tree. Those were only useful when we coded our program. You will have to look at four text files to see the result of the program. Here are the result with the sentence "Hello world".

```
≡ Texte.txt
1 Hello world
```

This one corresponds to the text you entered before running the program.

```
≡ bit.txt
1 01001000011001010110110001101100011011110010000001101110110111100100110110001100100
```

This one corresponds to the binary equivalent of the text you previously entered.

```
≡ dictionary.txt
1 'o' : 00
2 'r' : 010
3 ' '
4 ' : 0110
5 'd' : 0111
6 'l' : 10
7 ' ' : 1100
8 'w' : 1101
9 'H' : 1110
10 'e' : 1111
```

This one corresponds to the "dictionary". Here, we have compressed the 8-bit binary equivalent of each letter from the text. This will be used for the following file.

```
≡ output.txt
1 1110111110100011001101000101001110110
```

This one corresponds to the compressed binary code of the text you entered.