Assignment 1: Sorting in Assembly In this assignment we will emulate a small research project on efficient implementation of a program to solve a computational problem. There are therefore 3 parts:

1. Programming, where you after deciding on an algorithm will implement it along with the supporting code for input and output.

2. Implementation evaluation, where you will perform systematic benchmarks of your implementation. You will also need to make small code modifications to obtain statistics of your program performance.

3. Report, where you will document the whole project. Importantly this should be written as if this project description does not exist and was all your own idea. (This part, I can do my own 😊)

1. Programming

Your task is to write a program that when given a list of (x, y)-coordinates it sorts them on just the y-dimension and then prints the result. The program must be implemented in x86-64 Linux assembly. This means that you may not link to any libraries or use a higher-level language to generate the assembly code. Additionally, some code snippets especially for this project are provided, which you may use and modify as you much as you like. You are free to select which sorting algorithm(s) you implement, but you are recommended to start with an a relatively simple in-place algorithm (e.g., insertion sort, bubblesort) to quickly obtain a working solution. Afterwards you can experiment with better algorithms if time permits.

1.1 Input/Output Specification:

Your program should be callable with a single command-line argument which is a filename for a file with coordinates to sort. The indicated file will contain 1 coordinate per line with the format:

```
⟨x⟩\t⟨y⟩\n
```

Where

- \t is a tab character (ASCII code 9),
- \n is a newline character (ASCII code 10), and
- ⟨x⟩ and ⟨y⟩ are integers in the range 0 to 32767, i.e., the non-negative numbers representable in 16-bit twos complement format.

Note that the line endings are the customary ones on Linux, but that Windows and macOS usually uses different characters for line endings. Your program must read the numbers in the file, sort them on the y-coordinate (i.e., the second number), and then print them to stdout in the same format as the input file. Nothing else may be written to stdout, but you are free (and encouraged) to print error messages to stderr. This means that:

```
> ./yourProgram numbers.txt
```

should do the same as

```
> sort -n -k 2 numbers.txt
```

(Though, if two coordinates have the same y-value, their order may be different than what sort produces.)

1.2 Hints:

Iteratively Obtaining a Working Solution:

It may be tempting to write the program in the order it should work in the end, but it is difficult to test it as you go along. Instead, consider breaking the program into smaller bits and work “outside-in”. For example, if you implement something that reads input, implement output functionality for it to check that it works. The last major part should be the sorting algorithm itself.

Creating Random Numbers

For creating a test file with, say 42, coordinates you can use the $RANDOM variable in Bash:

```
> for i in {1..42}; do echo -e "$RANDOM\t$RANDOM"; done > numbers.txt
```

1.2.1: Checking the Output:

You can use the check.py Python program to check if your program output is sorted:

```
> ./yourProgram numbers.txt > output.txt

> python3 check.py numbers.txt output.txt
```

It will check 1) whether the second file ("output.txt" in the example above) is sorted, and 2) whether the lines in the two files are permutations of each other. If the first check, for sorted order, fails it will print the line numbers of the two first lines that are not in order, along with the lines, e.g.:

```
python3 check.py numbers.txt output.txt

Disorder between line 42 and 43. Lines are 26263 31002 15812 11169
```

If the second check, for permutation, fails it will print the first lines it cannot match up. The second check can also be performed manually with ordinary command line tools:

```
> ./yourProgram numbers.txt > output.txt

> sort numbers.txt > sorted_numbers.txt

> sort output.txt > sorted_output.txt

> diff -u sorted_numbers sorted_output.txt
```

1.2.2 Code Snippets:

They all follow the ordinary calling conventions for x86-64 on Linux. You are additionally allowed and encouraged to use snippets.

allocate (allocate.s):

- This function asks the OS to change the size of the data segment in the running program, thereby getting more space. As argument you simply pass the number of bytes needed in %rdi. A pointer to the allocated memory is returned in %rax.
- Note that memory is never freed, so there really is not much manage�ment of memory, and you should be careful if you allocate memory in a recursive function.

getFileSize (fileHandling.s):

- When you have opened a file you can get its size (in bytes) with this function. Pass the file descriptor for the file as argument.

getLineCount (parsing.s):

- When you have read the contents of a file into a buffer, this function can count how many coordinates are in it (by counting the number of newline characters). This will allow you to allocate the appropriate amount of memory for the parsed coordinates.
- The first argument must be the address of the memory with the data.
- The second argument must be the length of that piece of memory

parseData (parsing.s): This function converts the ASCII representation of the coordinates into pairs of numbers.

- The first argument must be the address of the text buffer to parse.
- The second argument must be the length of that text buffer.
- The third argument must be a pointer to a piece of memory to write the parsed coordinates to.

Let n be the number of coordinates in the input data, and let (xi , yi) denote the ith coordinate. The parsed coordinates will then be writ�ten into the the given memory block as an array with the content x0, y0, x1, y1, x2, y2, . . . , xn−1, yn−1. Each individual number will be taking up 8 bytes (make sure you have allocated enough memory).

Part 2: Program Evaluation:

After you have finished your sorting program, you should evaluate its performance, e.g., to compare it with the theoretical complexity of your chosen algorithm(s). The goal is to perform benchmarks to convince the reader of your report that your implementation performs as expected. As a default suggestion for this assignment:

1. Generate test instances with each of the following amount of uniformly random coordinates:
   - 10,000
   - 50,000
   - 100,000
   - 500,000
   - 1,000,000
   - 5,000,000 (if the runtime is not unreasonably high)

For each size you should create 3 instances.

2. Measure the runtime of your program using the “time” command. You should record the “real”/ “wall” time. If it differs dramatically from the “user” time, it would proper to discuss it in your report. Note, the displaying of output from the program may slow it down. To be sure this is not the case, you can redirect the output to the null device:

```
> time ./yourProgram numbers.txt > /dev/null
```

3. Display your runtime evaluation in an appropriate manner suitable for evaluating the performance

4. Obtain statistics to compare the performance with the theoretical complexity of your sorting algorithm: Count the number of compare operations you use during sorting, where elements from the input participate. How is the number of comparisons related to the overall runtime? Hints: In order to calculate the number of comparisons, you can simply increment a variable after each compare instruction and output the result in the end. Make sure you make your time measurements without this counting overhead. The submitted version of your program sources should not contain this counting or additional output either. If your chosen sorting algorithm is not based on comparisons, you may skip the part about counting compare instructions (and the related tasks). You must instead illustrate how the execution time relates to the theoretical complexity of your chosen algorithm in an appropriate manner.

5. Related the number of compare instructions to the execution time. Calculate the number of comparisons per second as a performance metric for each of your input files. How does it change with different input sizes?

6. Assuming the sorting took no time (e.g., temporarily comment the call to your sorting algorithm out), the remaining runtime is the overhead of the program. How is the overhead time related to the file size?