**The Joker**

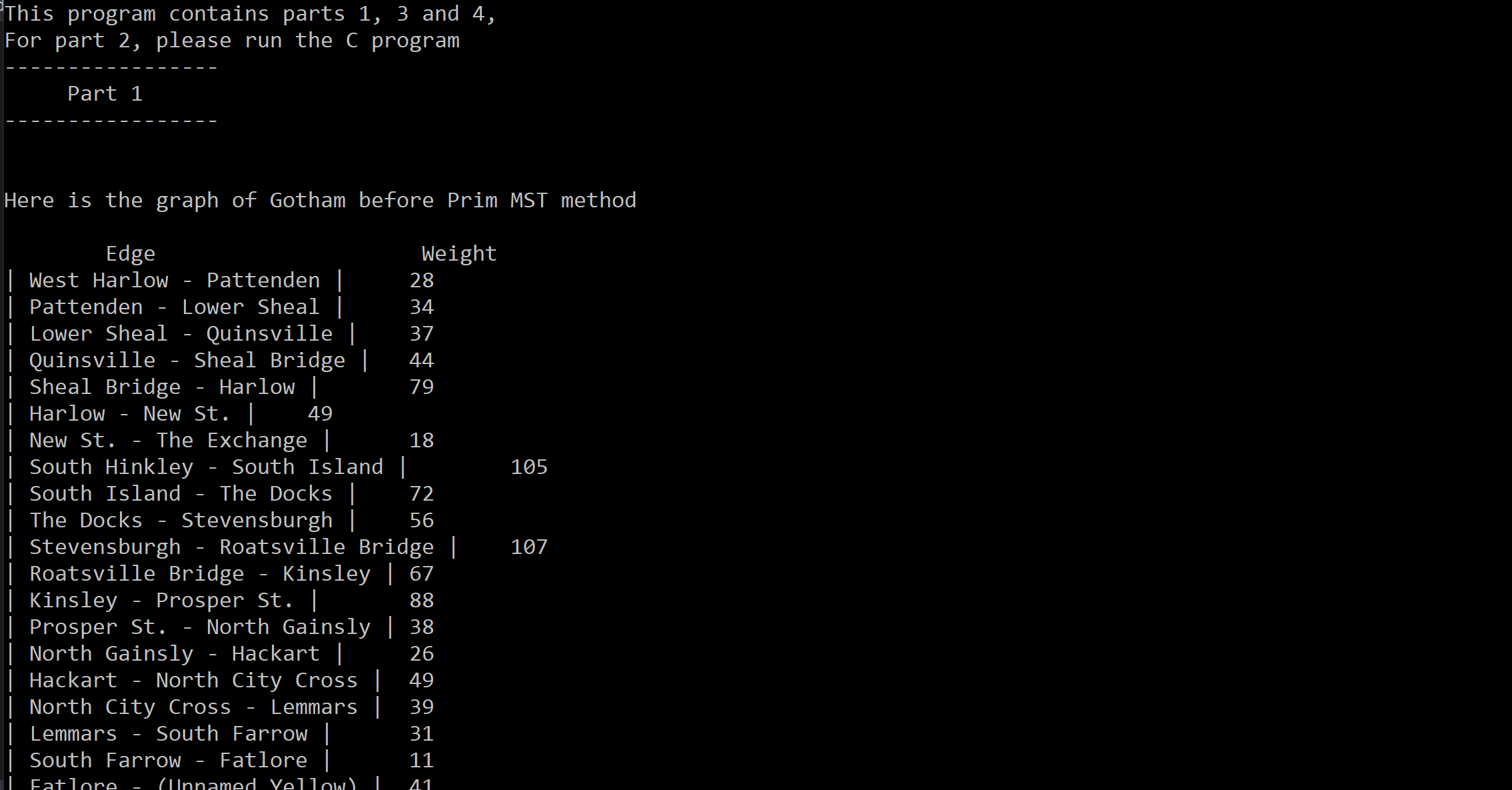


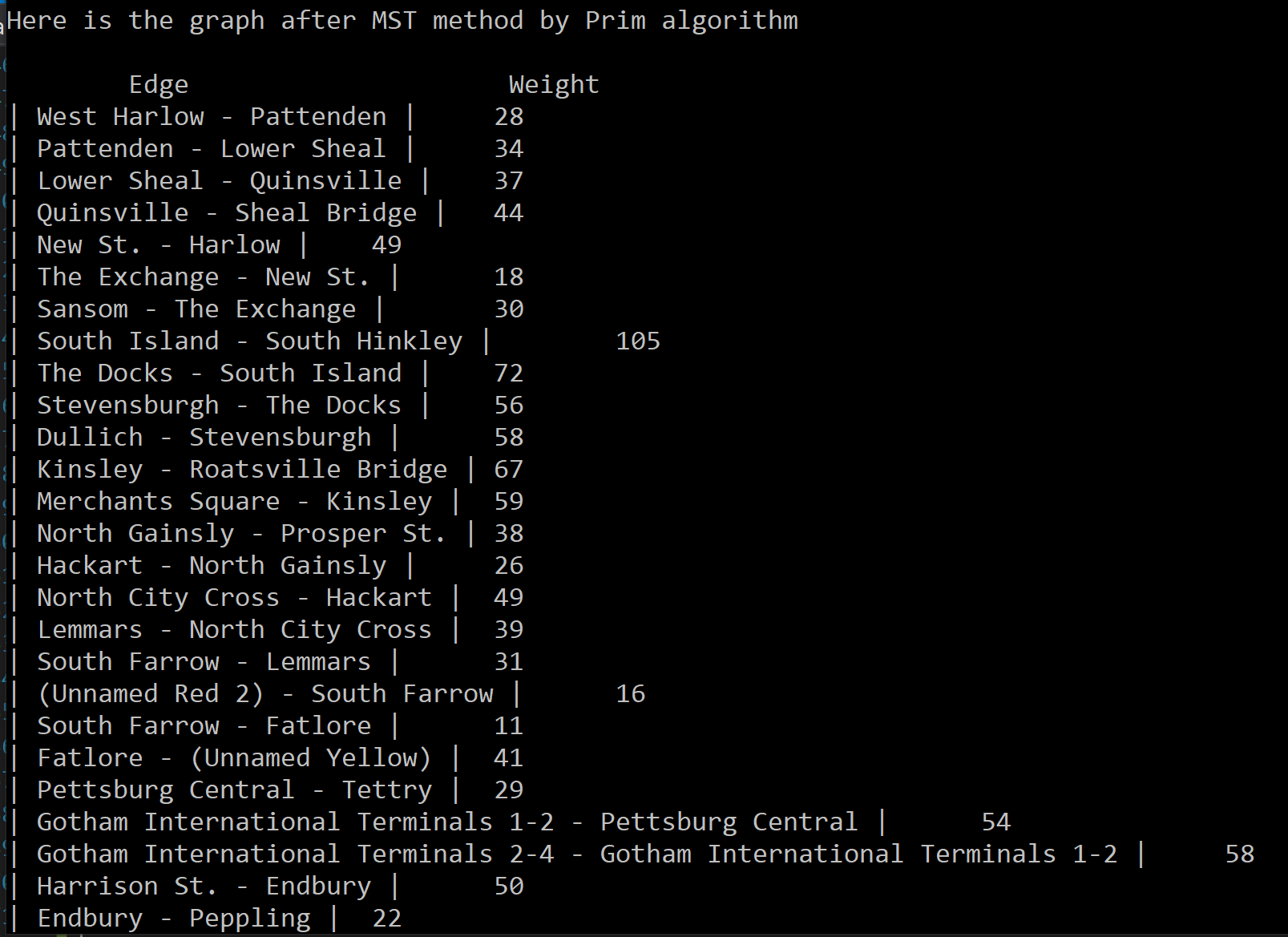
I did part 2 using C all by myself, without tutorial. It’s a language close to the computer memory and I found useful to get this part optimized to it’s maximum following C rigor and ISO99 norm. C allows really fast execution if things are done well and that’s why it’s useful for pathfinding. I did the other parts in c# with some internet tutorials (stackoverflow, openclassroom).

**Part 1 :**

2)

The problem we need to solve is a minimal spanning tree. I’m using Prim method which complexity is O(n²) for arrays. I measured the plan and put the distances in a csv, the programs reads it. Datas are put in arrays.





**Part 2 :**

1)

In the subject, we are asked to find the shortest path from g1 to another city. I am using Djikstra’s algorithm.

This algorithm is of polynomial complexity. More precisely, for n nodes and a arcs, the time is in O ((a + n) \* log n)

In this algorithm, too long paths are not are not taken in account. I’ve focused on the shortest and the fastest path. User can user gotham dataset and French citys dataset. If two paths have the same length then they’ll be both displayed. I also have an optional random path fiding.

To avoid duplicating datas, I have connections that allow us to make the link between departure and arrival. Connections carry informations for user (time and length between the 2 citys)

Thanks to these connections, it is therefore easily understandable that going to g1 from g2 is exactly the same as leaving g2 for g1, there is a certain reciprocity.

The first question I asked myself was: "damn but how to build these connections?" ".

For my structure, I therefore first create a chained list of our cities, where each city points to another city and to a chained list of connection pointers (which we will call nodes).

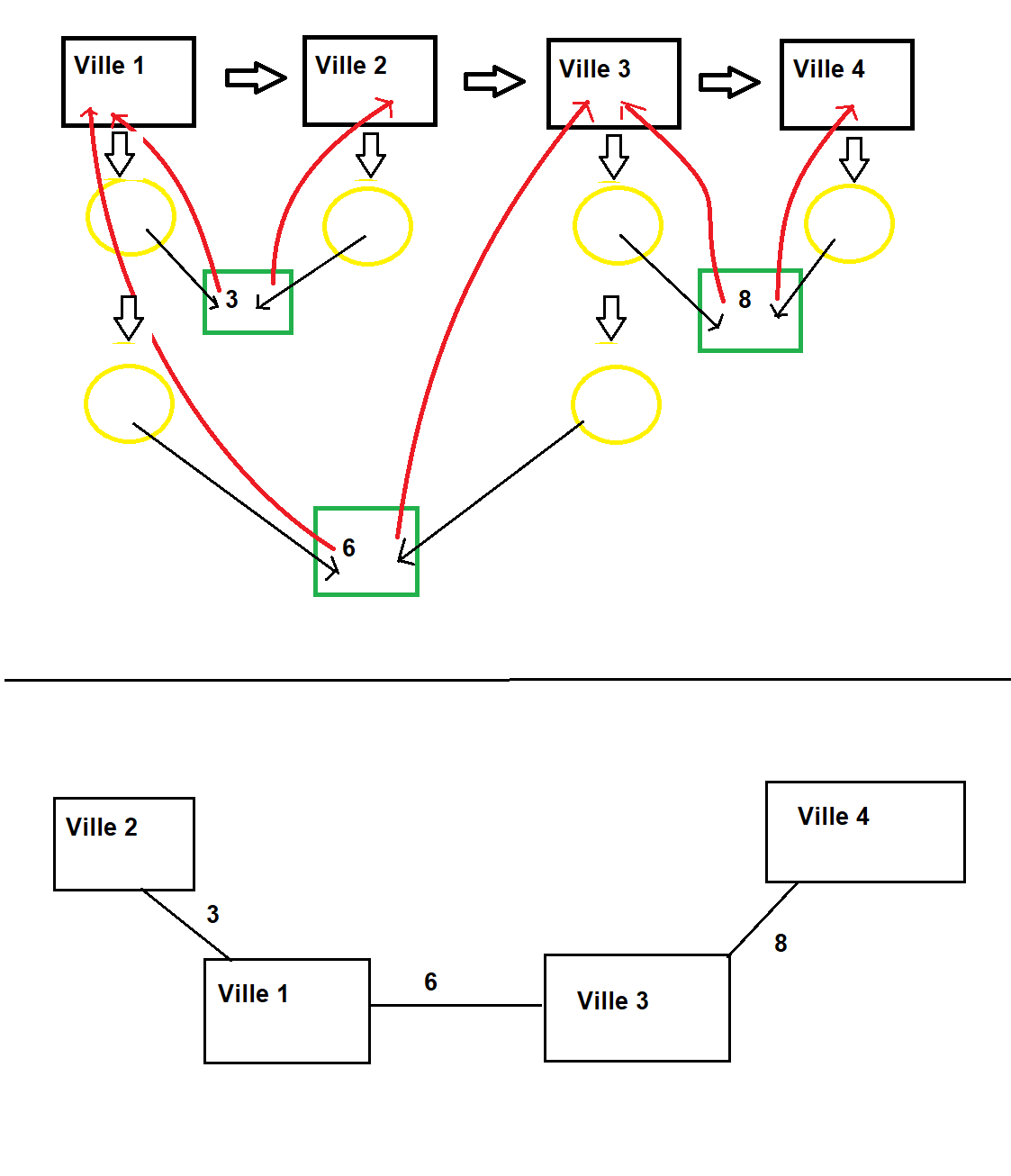
Each node in turn points to another node (until the last node points to null) and a connection.

I have also added a boolean in our connection, allowing the passage of only once through this connection, during the naive search of the path.

I also added a cell structure in the city structure which contains a chained list of pointer to a city and a weight.

Tables need to know their maximum size during creation. Which is not the case here, unless you read the files twice. Today they only weigh a few KB, but let's imagine that tomorrow they will be 10GB, we don't want to read them twice. In addition, we preferred to read the file only once but use linked lists rather than using tables (more practical) but read the files twice. In addition, thanks to the linked list, we can modify the list of cities in runtime. Which seemed to us the wisest choice in terms of future prospects.

This data structure allowed us to easily apply the algorithm which allows THE quickest and shortest path in the most optimized way possible: thanks to the dijkstra algorithm, here is the explanatory link of the site on which we we are based to design our algorithm http://serge.mehl.free.fr/anx/algo\_dij.html. All my work comes from this site, I have not taken any explicit code or even algorithm on the internet. Everything has been recoded. I thus considered the connection file given as input as an undirected graph.We will see its usefulness in the rest of our explanation for better understanding.



Here is an illustration of how I build the data structure from the graph below.

As I said above, the primary goal of our algorithm is to have a correct result (the right path), a fast result (optimal algorithm complexity). Which seems good resolutions given the title of the module.

For this purpose, we realized that there could be a repetition when we load the data of the graphs because A to B also implies B to A.

A city is created if and only if it has not already been called before.

Similarly, the connection takes the attributes (distance and time).

In this way, we read the file ONLY once, which allows us to save a huge amount of time, compared to an algorithm which should read 2 times each file to initialize the size of its tables and the second time for them. fill.

Concerning the rigor of the code, we followed at best some regulations specific to C ISO99 :

- One instruction per line (So no int i = 0; but int i; i = 0;)

- A line break before each hug

- No more than 25 lines per function (scope excluded)

- No more than 5 functions per file

- Each function name must contain a verb

- Each function should only do one thing at a time

- The only line break allowed is after the definition of the variables

- Functions not containing parameters take the void parameter

- Coherence of variables

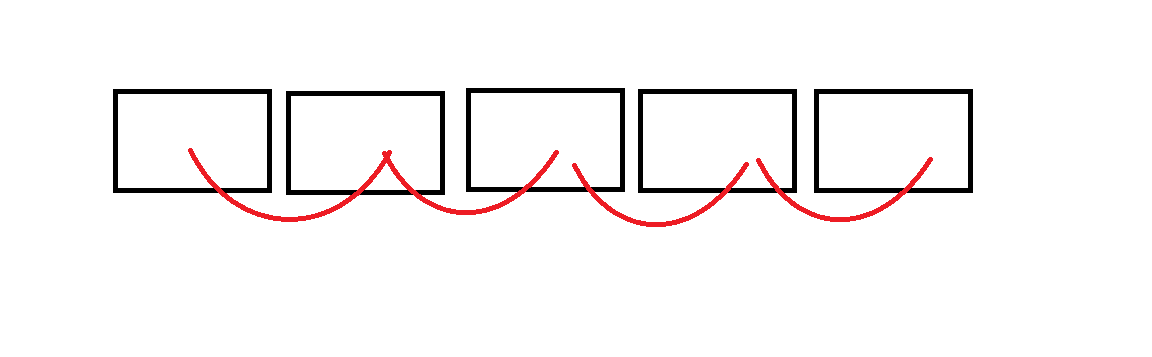
- 0 warning at compilation

**Part 3 :**

Each database element is a Joker member. They are anonymous. A joker member is characterized by a letter and an Id.

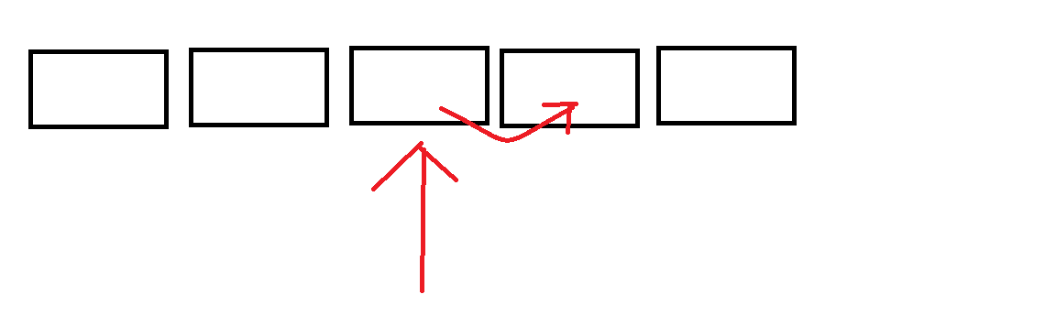
1)

A greedy method to find an Id is to linearly look for all id until we find the good one. The complexity is O(n).



2)

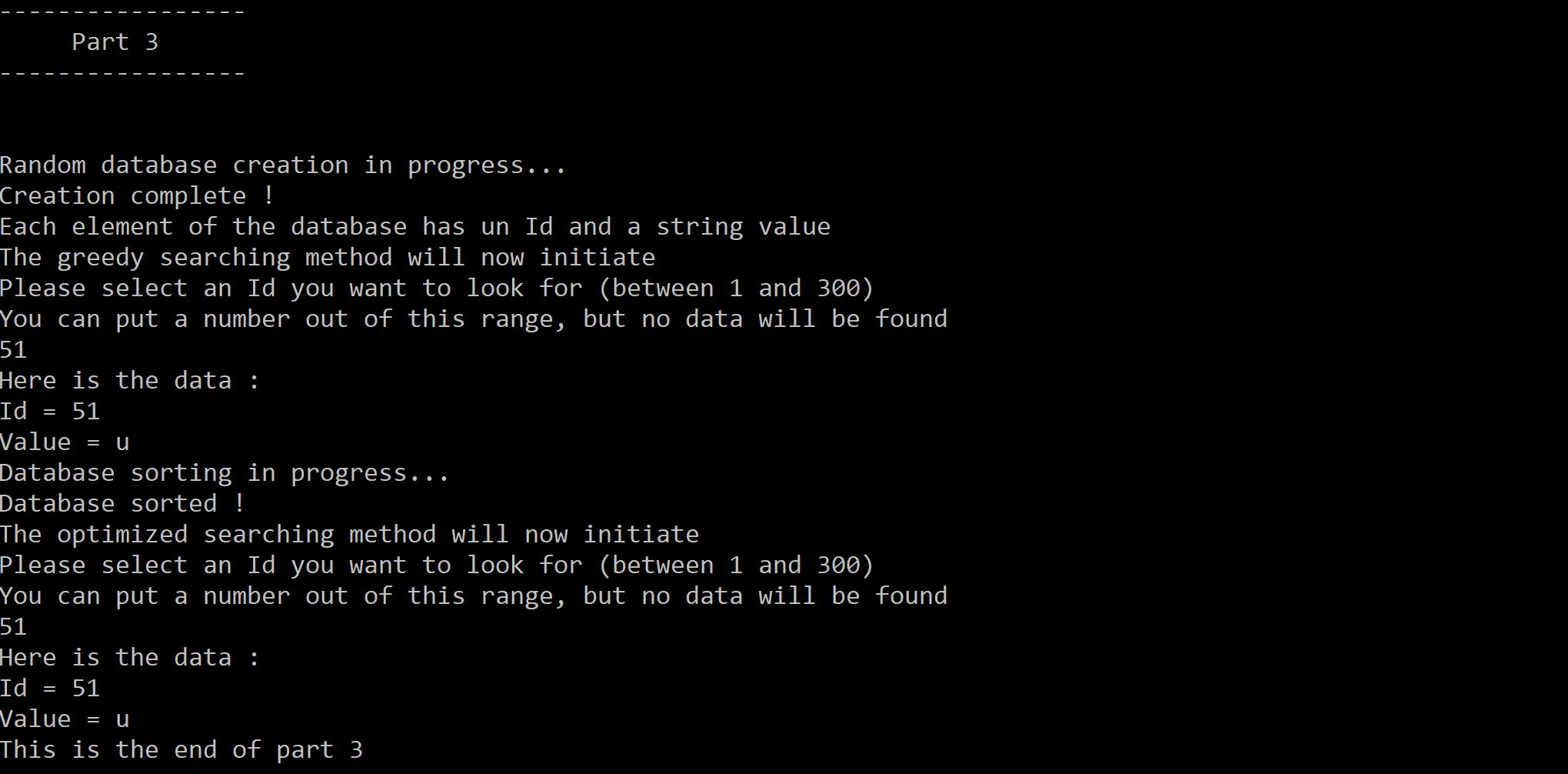
A better method to find an id in the database is dichotomic search. The database must be sorted. The complexity is O(log(n)).



3)

I’m using binary insertion sort. The complexity is O(n\*log(n))

A new blank list is created and for each element in the original list (O(n)), the element is inserted in the new list with dichotomic insertion (O(log(n))).



**Part 4 :**

