Kordel France
Lab 1
Class 605.621 Section 83
13 July 2021

Lab 1 Analysis Document

FranceLab1 compares the performance and asymptotic costs of four Deterministic Turing Machines (DTMs). *FranceLab1* is a software module written in Python 3.7 that constructs DTMs to facilitate arithmetic between binary operands.

DTM Design

Four DTMs are constructed for this program:

a) a pattern recognition DTM leveraged from the class material called that detects a pattern of multiple zeros in binary strings (referred to as *DTM_demo*)

b) an addition DTM that adds two binary strings (referred to as *DTM_add*)

c) a subtractive DTM that subtracts one binary string from the other (referred to as *DTM_sub*)

d) a multiplicative DTM that multiplies two binary strings together (referred to as *DTM_mul*)

The state machine diagram for each DTM is discussed below along with some insight into the DTM's operation. Eventually, I realized that a common set of rules that facilitate states of increment and decrement among the operands significantly helped simplify the three arithmetic DTMs.

Pattern Recognition DTM

The pattern recognition DTM proved to require the least amount of rules, logic and operational cost. This DTM detects attempts to detect a pattern of two zeros following a 1. If it finds this pattern, the state is "complete" and the DTM ends in $q_{completion\_state}$. Otherwise, the tape is checked until the DTM enters the state "terminate" denoted by $q_{terminate\_state}$. Only four states are needed to facilitate its operation. One state was added for error handling, but the algorithm can perform more simply on three states. Personally, seeing the DTM step through states really opened up my eyes to the power of DTMs. The fact that such a simple machine can detect patterns gave me a much more appreciation for the complexity of more sophisticated pattern recognition algorithms in artificial intelligence such as neural networks.

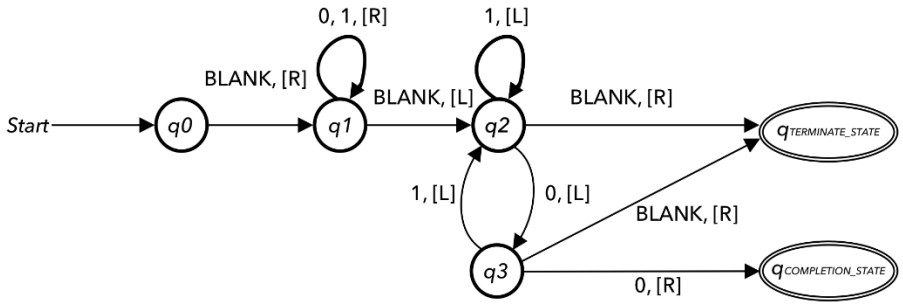

Figure 1 - State diagram for the Demo DTM that recognizes a pattern of zeros in a binary string.

Addition DTM

The Addition DTM proved to be more complex than the previous one. The DTM required eight states in order to execute successfully, though some of those states facilitate error handling along the tape. This DTM served as the foundation for the Subtraction and Multiplication DTMs. At one point, I consolidated the Subtraction and Multiplication DTMs into the Addition DTM to serve as one giant DTM capable of performing multiple mathematical operations.

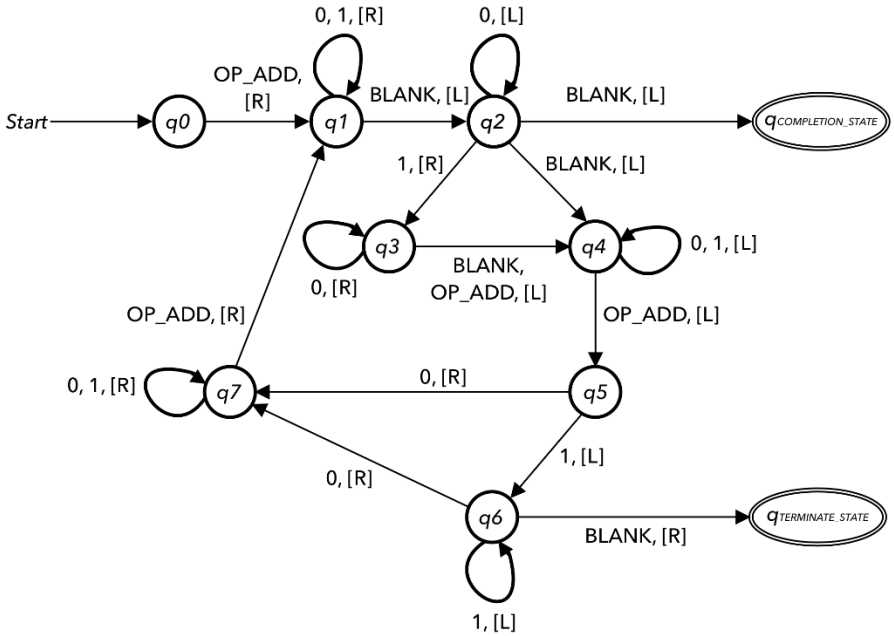

Figure 2 - State diagram for the Add DTM that adds two binary strings together.

Subtraction DTM

The subtraction DTM seemed intimidating at first, but ultimately proved to be just a slight tweak on the Addition. Essentially, by reversing the direction of integer carrying and swapping the decrement-increment procedure on either side of the operand, the Addition DTM turns into a Subtraction DTM. It was at this point I realized that having a common set of rules that facilitate states of "decrement" and "increment" across the three arithmetic DTMs was necessary to keep state space low.

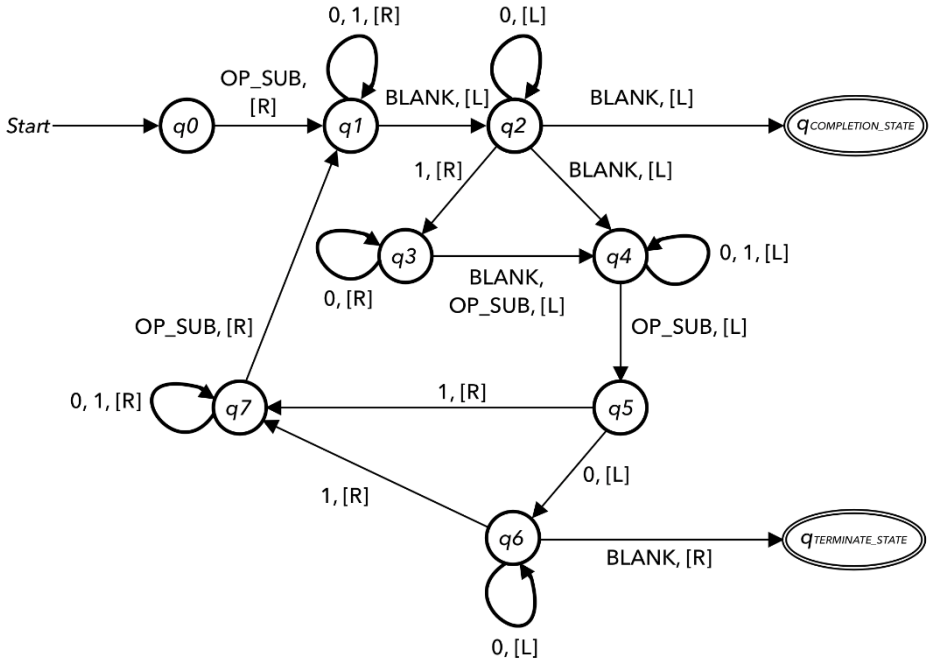

Figure 3 - State diagram for the Sub DTM that subtracts one binary string from another.

Multiplication DTM

The multiplication DTM was the most difficult one to construct. This primarily had to do with more back-and-forth runs from each side of the operand in order to properly distribute the multiplication. On the Addition and Subtraction DTMs, their operation seemed to take less *O(n)* runs than the Multiplication DTM needed, where *n* is the number of digits within the binary string on one side. Interestingly enough, the DTM still required less rules than the other two arithmetic DTMs, though less error handling was handled within the state space here.

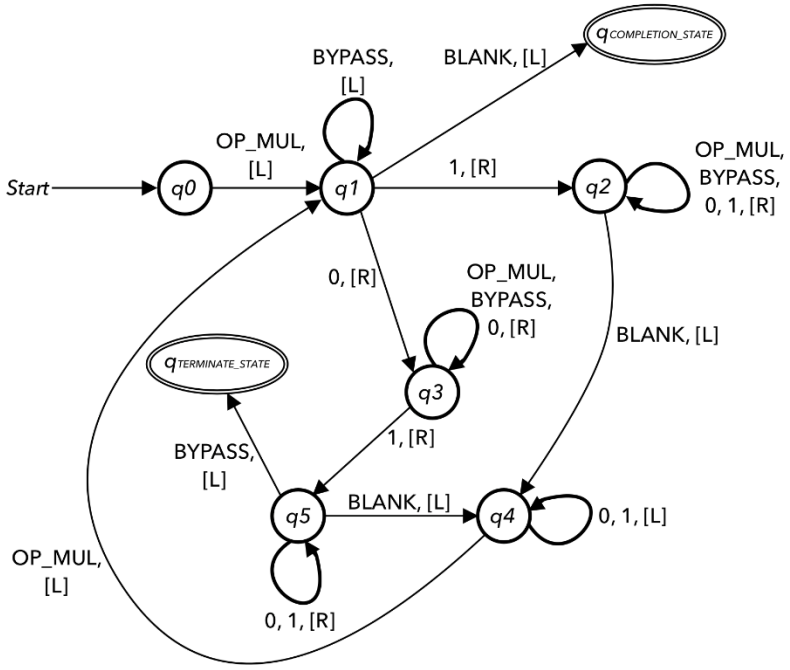

Figure 4 - State diagram for Mul DTM that multiplies two binary strings together.

Kordel France
Lab 1
Class 605.621 Section 83
13 July 2021

Cost Analysis for Both Algorithms

Please refer to the table below for acquired runtime costs for each DTM in the *FranceLab1* module. Please note that when *n* is mentioned, it refers to the length of a binary string, such that *10* represents a binary string of *n=2*, and *1001* represents a binary string of *n = 4*. Obviously larger values of *n* will result in more algorithmic operations by the DTM, so this was the metric scaled to determine the associated runtime complexities for each DTM.

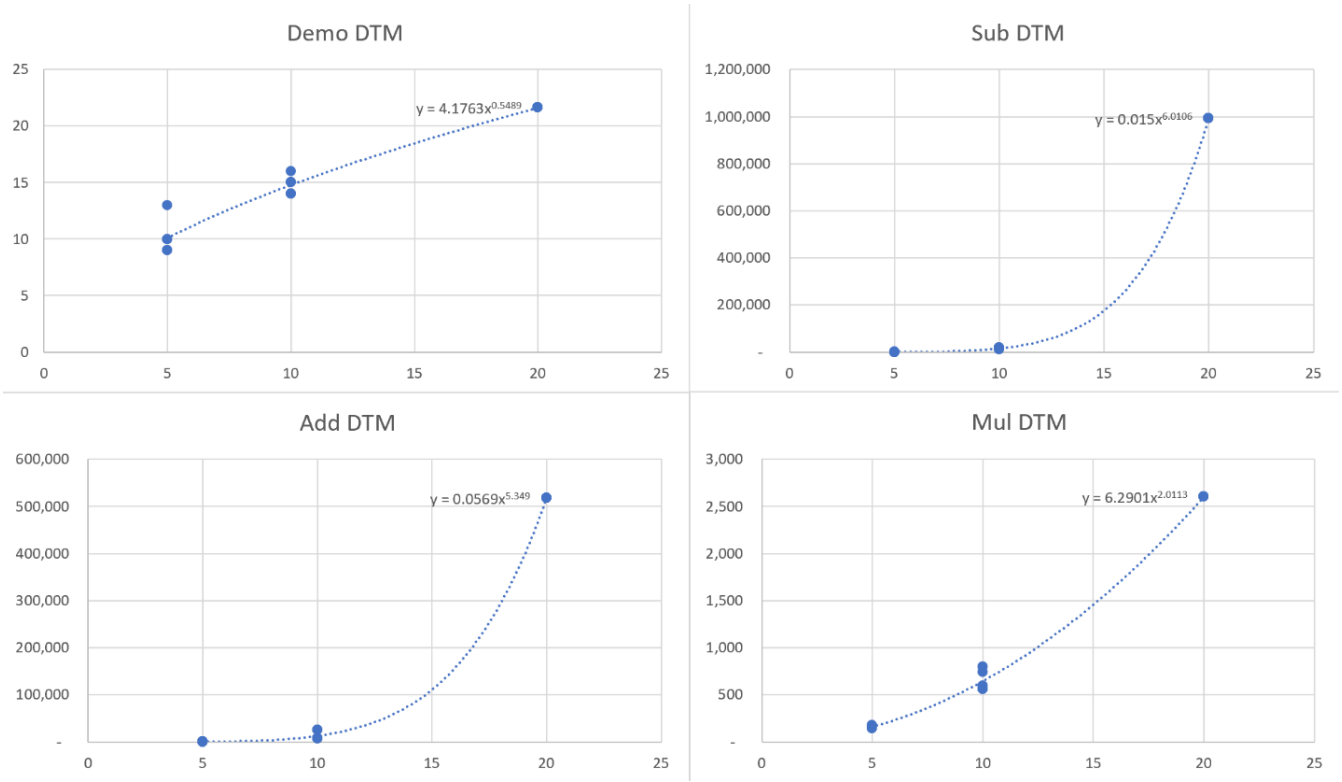

Table 1 - Cost trajectories using exponential regression for cost runs on each DTM.

A series of cost runs with binary strings of 5, 10, and 20 integers long for each operand was extrapolated out using exponential regression to get an idea of what the true cost may be for larger binary strings on their respective DTMs. Due to the time it took to perform cost runs on larger binary strings (i.e. *n = 100, 1000*, etc), these statistical methods were used to get an estimate.

The pattern recognition DTM (a.k.a. "Demo DTM") provided the best runtime of all the DTMs with a time complexity of *O(n)*. Costs get much more severe for the Addition and Subtraction DTMs with time complexities of nearly *O(n^6)* each. This is especially concerning given that the multiplication algorithm, which intuitively seems like a more complex operation, attained a time complexity of *O(n^2)*. I presume the reason behind this is due to the fact that the state space is larger in the Addition and Subtraction DTMs because more error handling is incorporated. Not much error handling occurs in the

Multiplication DTM which is why the state space is smaller and, consequently, the runtime is significantly smaller.

It seems that the runtime is directly dependent on the size of the state space. This can be optimized by better design of DTM rules and less "error handling".

A graph with another series of cost runs for all four DTMs that is a direct result of the output from *Lab1* is displayed below.

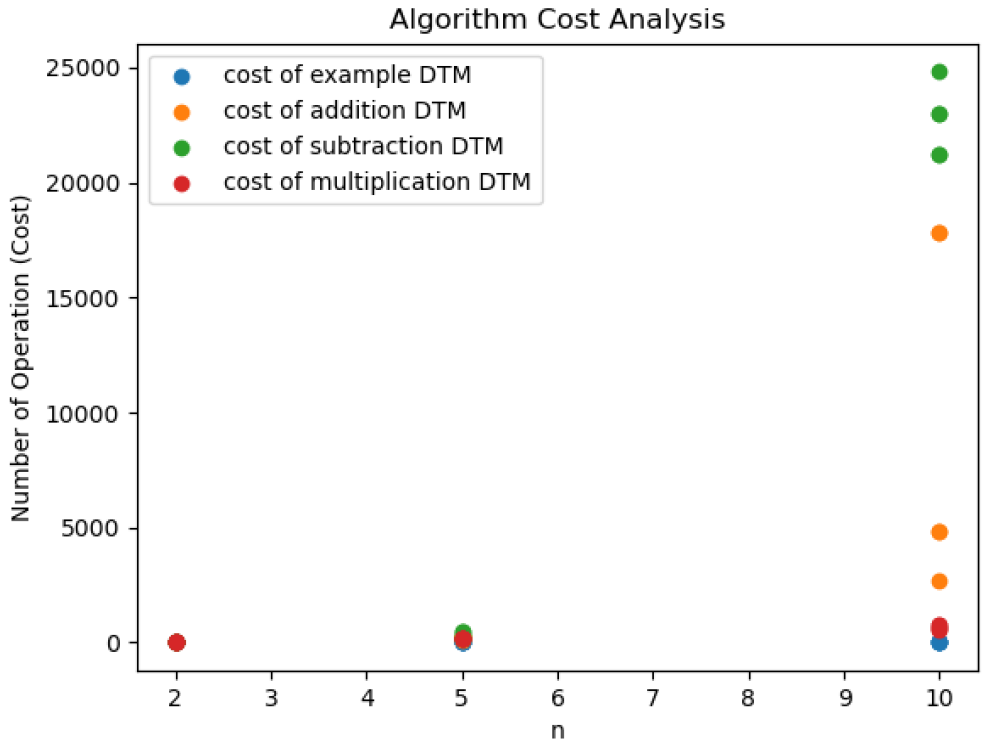

Figure 5 - Series of cost runs for all four DTMs together.

A table showing direct output from a series of cost runs for binary strings of length 2, 5, 10, and 20 is shown below. These numbers were used to derive the regression equations above as well as the time and space complexities discussed below.

#	DTM	n	Cost	DTM	n	Cost	DTM	n	Cost	DTM	n	Cost
1	Demo	2	7	Demo	5	9	Demo	10	14	Demo	20	25
2	Demo	2	7	Demo	5	10	Demo	10	15	Demo	20	26
3	Demo	2	7	Demo	5	9	Demo	10	14	Demo	20	24
4	Demo	2	7	Demo	5	10	Demo	10	15			
5	Demo	2	7	Demo	5	13	Demo	10	16			
6	Add	2	17	Add	5	185	Add	10	8,297	Add	20	26,968,127
7	Add	2	7	Add	5	367	Add	10	25,757	Add	20	40,888,311
8	Add	2	12	Add	5	205	Add	10	8,304	Add	20	34,783,029
9	Add	2	16	Add	5	349	Add	10	25,701			
10	Add	2	20	Add	5	605	Add	10	7,243			
11	Sub	2	43	Sub	5	225	Sub	10	17,989	Sub	20	10,993,717
12	Sub	2	17	Sub	5	193	Sub	10	10,865	Sub	20	9,340,213
13	Sub	2	39	Sub	5	220	Sub	10	17,604	Sub	20	10,570,173
14	Sub	2	21	Sub	5	199	Sub	10	18,209			
15	Sub	2	31	Sub	5	407	Sub	10	13,667			
16	Mul	2	44	Mul	5	169	Mul	10	740	Mul	20	2,622
17	Mul	2	32	Mul	5	149	Mul	10	564	Mul	20	2,334
18	Mul	2	40	Mul	5	181	Mul	10	801	Mul	20	2,714
19	Mul	2	36	Mul	5	140	Mul	10	559			
20	Mul	2	32	Mul	5	165	Mul	10	600			

Figure 6 - Actual values from trace runs on the four DTMs.

A beautiful thing about DTMs is that the space complexity will always be constant and very low at *O(1).*The only thing that the DTM must keep in memory is its current state and the state space in general. In this sense it sort of exhibits the Markov property. This emphasizes the compromise between speed and memory in that we have a very simple machine that requires very little memory yet is very slow in computation. Nevertheless, the machine works and gets the job done.

As a final note, I designed the program to regenerate the trace run files at runtime so that the program experienced a more uniformly distributed dataset. Because of this, the numbers illustrated in Table 1 show costs that are averaged multiple times over these trace runs, not just one set of trace runs in general. This randomization of output seemed to provide a well-rounded dataset.

Lessons Learned

Above all, the principles I took home from *Project 1* revolved around an engineering perspective: DTMs provide a great opportunity in scenarios where processing power is severely constrained and memory availability is low. Complex computation can be performed on very inexpensive, meager processors through carefully designed DTMs which can minimize mechanical packaging of the computer and dramatically affect program budgets. Above all, *Project 1,* allowed me to see just how powerful simple algorithms are when implemented with meticulous, effective rules. It is obvious now that a Turing machine can solve nearly any complex problem with enough rules and allowed states. DTMs also give good insight into how computers work at a hardware level by varying voltages as 1's and 0's. The pattern recognition DTM, though the most simple of the four, was quite remarkable after I finished its

development; it provided deeper understanding (and appreciation) into the execution of complex neural networks.

Generating the input data, building and running the DTM on that data, storing the results for output, and then logging that output to the console and text files facilitated a few functions with nested for-loops. Upon further review of the code, I am very confident I could reduce these nested loops to multiple sequential for-loops to bring the runtime of some helper methods from $O(n^2)$ down to $O(n)$. These inefficiencies are due to memory management of the output, but I was able to bring one $O(n^3)$ function down to $O(n^2)$ which I actually noticed in the runtime.

Given that this project is working with binary strings, many helper functions can take advantage of $O(lgn)$ or $O(n\ lgn)$ runtimes. These is especially true with helper methods needing to sort or prepare the output data in any way and I noticed this in my program design.

Enhancements

The program provides 5 major enhancements (along with many other minor enhancements) on top of the original requirements.

1) The first major enhancement involves the automatic regeneration of trace runs at runtime. Each time the program is started, trace runs are erased and regenerated to facilitate an unbiased estimation of the asymptotic cost of each DTM. I originally incorporated this only as a form of quality control on each DTM to test the correctness of its output. However, I kept the feature in there for the final assignment because it helps facilitate data follows the *IID* rule by ensuring an identical and independently distributed data population.

2) The second major enhancement incorporated time delays. When the program is conventionally executed, the command prompt flashes a plethora of data and finishes executing in under a second. The user is then forced to go back and read through the steps executed. *FranceLab1* incorporates time delays to allow the user to read information in the command prompt as processing steps are completed. For example, when the program starts up, a time delay of 3 seconds allows the user to know whether or not the input files were detected. Additionally, when the program successfully imports the I/O files, this information is logged to the command prompt and a time delay of 3 seconds halts processing so that the user can take in this information accordingly. After the data for each trace run is configured, processing is briefly paused to allow the user to visualize the operators, operands, and result. Finally, after each cost and correctness run finishes evaluation, a time delay of 1.5 seconds briefly holds processing for each evaluated set of points as well to allow the user to see whether or not that step failed.

3) The next major enhancement revolves around error handling. Total error handling is difficult with DTMs, but I was able to incorporate some methods to handle most cases of bad input. If an input file contains illegal characters (non-numeric numbers and non-decimals), the program

notifies the user through both the console and the output file. However, instead of simply letting the user know that the expression contained invalid characters, they are notified of exactly which characters were illegal in their expression. Processing then continues with the points that were successfully read in, if any.

4) The fourth major enhancement builds a summary metric for the user through a graph. At the end of the program, a summary of the runtime costs evaluated for each trace run are consolidated and displayed in a graph. This makes it exceptionally simple for the user to visualize how well the DTMs performed in comparison to each other as the of the input grew.

5) The final major enhancement involves aesthetics. This is more good programming practice than anything but something that I feel took considerable effort to implement. It is one thing to satisfy the project requirements by dumping all of the information to the console and/or output file, but I felt that it was important to keep the user in mind by making things easier to follow and interpret as the program executed. I tried to discretely separate cost runs from correctness runs and intentionally segregate trace run data for different levels of *n*. I also tried to communicate data more effectively by displaying output like *"original tape for DTM: 0 1 1 0 + 0 0 1, final tape for DTM: 0 1 1 1."* These principles were followed throughout both the console output and output file, the console output being a duplicate of what was written to the file.

Future Optimizations

In a future version, it would be nice to provide some error handling around negative numbers. Automatically detecting whether the arithmetic operation was being performed on positive or negative numbers was the most difficult aspect for me. Additionally, I would have liked to implement the division operation between two binary strings given more time. The rules around the multiplication DTM could also be optimized. While my multiplication DTM works, I feel there is a rule or two I could relieve from the design through better programming.

This project really sets up the algorithm for base-6 and base-10 numbers. I would like to build a program that could handle these numbers as well as ASCII codes. Watching the tape run on the DTM gave me a lot more insight into how some encryption algorithms may work in real-time. In a future version, I would build a simple encryption algorithm that can be handled by the DTM.

Finally, this project effectively displays how a DTM runs along input represented as a 1D strip of tape. It would be really interesting to construct a DTM that could handle 2D input such as matrices. It's easy to extrapolate how a DTM would operate in this fashion and easier to see how complex the rule set and state space could become over a 2D input tape. Nevertheless, it is possible.

Conclusion

The *FranceLab1* is a Python module performs mathematical operations on and between binary strings through the use of Deterministic Turing Machines. The program defines, constructs, and facilitates the execution of four different DTMs on both explicit input from the course and input automatically generated at runtime. Costs for execution on each of the machines for each trace run are acquired throughout the program and communicated to the user through both written output files and console display. The state space and action space for each DTM's execution are presented to the user in this format as well. Finally, a graph comparing the asymptotic costs of each DTM as the number of digits within the binary string operand scales is presented to the user.

References

The following items were used as references for the construction of this project.

1) Cormen, T. H., & Leiserson, C. E. (2009). Introduction to Algorithms, 3rd edition.

2) Miller, B. N., & Ranum, D. L. (2014). Problem solving with algorithms and data structures using Python (2nd ed.). Decorah, IA: Brad Miller, David Ranum.

3) Kleinberg, John & Tardos, Eva. (2014). Algorithm Design. Dorling Kindersley.