

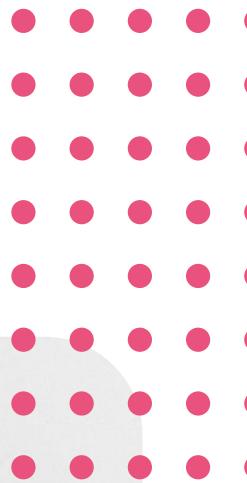
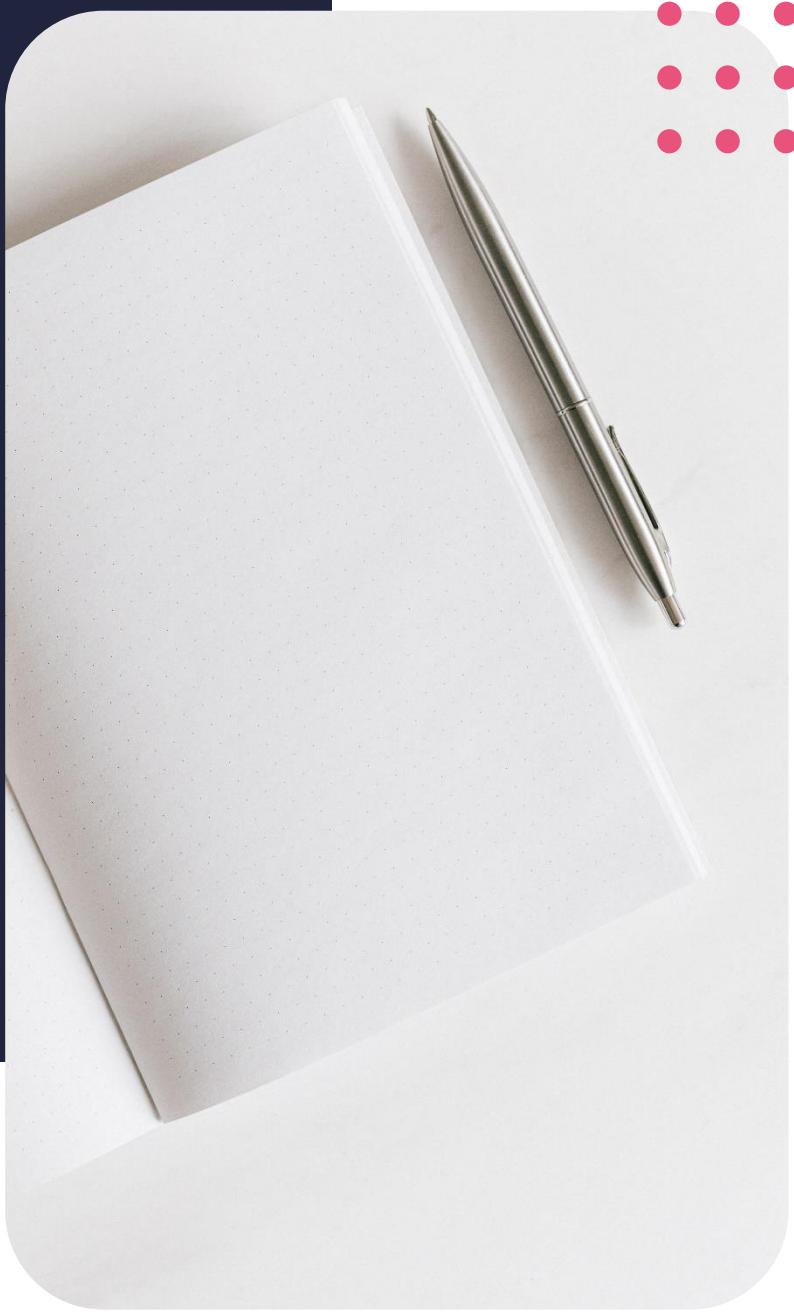
Diploma in Computer Science

Problem Definition



Contents

Formulating problems	3
Definition of a problem	3
Types of problems	3
Characteristics of computational problems	6
The world of maths	8
Mathematical modelling	9
A binary problem!	9
Common computing problems	10
Formulating problems	10
Summary	12
References	12



Lesson outcomes

By the end of the lesson, you are expected to be able to

- Understand what computing problems are
- Recognise types of computing problems and their characteristics
- Demonstrate problem formulation given a scenario
- Evaluate real world computation problems

Formulating problems

Before you write a program, you need to know why you are writing the program in the first place. This helps you to keep track of the program and make sure that the program addresses everything that it is supposed to do. This process is called problem formulation. As a computer scientist, people who want software solutions will just describe to you what they want the computer to do. It is your job to describe this problem in a way that will make it easy to formulate a solution

Definition of a problem

Computers can't do everything. There are problems that appear complex at first but can be broken down into smaller pieces then solved piece by piece. There are also problems that are complex and still remain complex and still remain complex, even after being broken down. There are also problems that seem simple but are incredibly hard to solve. For example, the question, "what is joy?" has so many different answers, scientific and non-scientific, that it is probably impossible to come to a universally accepted solution. If you dash over to google for a quick search on this topic with the intention of getting an answer, you'd probable come back angry and with more questions than answers because of the myriad of opinions that are there.

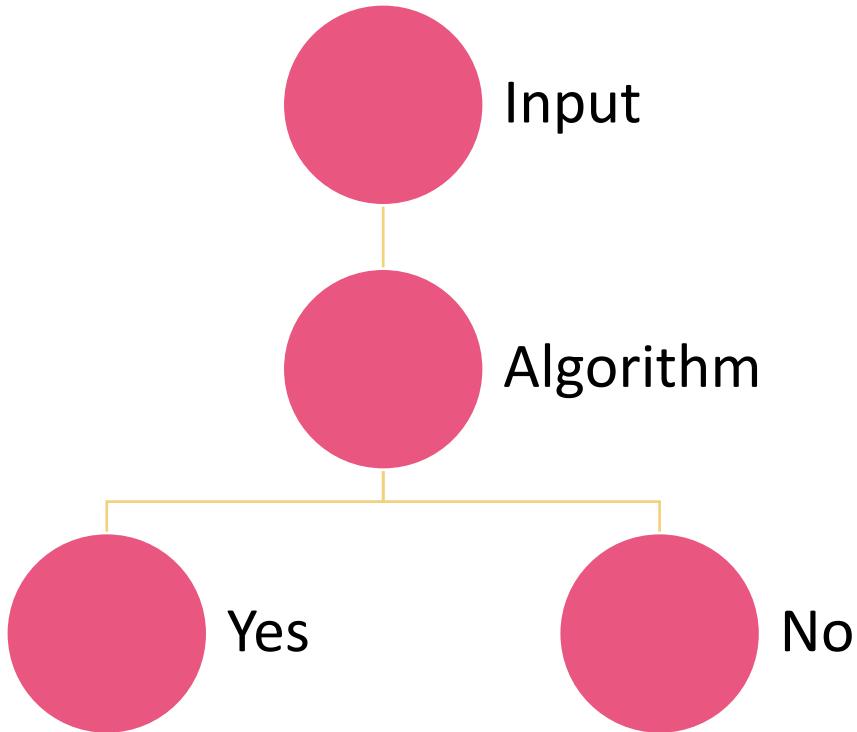
In computer science, anything that you try to achieve computationally is referred to as a problem. It can also be referred to as a set of related tasks.

Naturally, as we have observed from previous lessons, a problem should be described explicitly so that the computer knows what to do. Before we even get to the computer, a problem should be well stated so that the programmer knows what to "say" to the computer. one of the characteristics of a computer scientist is the ability to think computationally.

A computing problem is a collection of questions that a computer might be able to solve

Types of problems

There are four main types of problems, namely decision, search, counting, optimisation, and function problems.



A decision problem is a type of computational problem where the answer for every instance is either a yes or a no. A decision problem is one that seeks a solution with a certain characteristic. These types of problems are closely related to function problems; yes, those mathematical functions that you covered in school! A decision problem provides a solution for every given input. An example of this is, a program that needs to check if the sum of 2 numbers is equal to a predetermined value.

It has a set value of 1, stored as z. It then asks for 2 inputs from the user, x and y. It then adds together x and y, then compares this to z to see if the value stored as z is equal to the sum of x and y. If it is not equal, then it will ask for new inputs, otherwise, it will output z.

Let's look at this in a more concise way

Initialize z=1

Ask for x and y

Add x and y

Check if $x + y = z$

If it is equal, output z

If it is not, ask the user for new input. This program makes a decision at some point in its procedure. If it can solve a problem using this decision, it is said to be decidable. If not, it is said to be undecidable. This is a simple example, though, more complex problems are modelled using Turing machines, like we discussed in our

previous lessons. Using Turing machines, we can determine if a problem is decidable, partially decidable, or undecidable. An undecidable problem will loop forever and never provide a solution.

The finiteness of a Turing machine is an example of an undecidable problem.

Search problems are a type of problem where there exists an algorithmic way of verifying the answer. As we shall learn in lesson 8, an algorithm is a step-by-step method of solving a problem. Searching is the algorithmic way of finding an item in a collection of items. A search typically responds with a message indicating whether the search item is present or not. The algorithm is said to solve the problem if at least one corresponding structure exists, and then one of the instances of this structure is the output. Every search problem has a corresponding decision problem. A search problem is defined by

- a set of states
- a start state
- a goal state
- a Boolean function that we use to determine if a state is the goal state

A successor function, which is the method we use to jump from one state to the next.

We are going to look at state functions in lesson 8, as they are one of the most important type of algorithms. In a search problem, we not only want to know if a solution exists, but we want to find the solution as well. For example, we want to find the number 3 from the list of numbers 7, 8, 5, 9, 2, 6, 3, 10, 4. We can use a simple search algorithm that reads the numbers one by one until it gets to the value that we want. The number 3 is then set as the goal state. We then read the number 7, which becomes our current state, the current state does not match our goal state, so we discard it and go on to the next state, which is 8. It still does not match our goal state, so we discard it, and we repeat the cycle until we get to 3. This state matches our goal state, so we exit the search algorithm and output 3 as our solution.

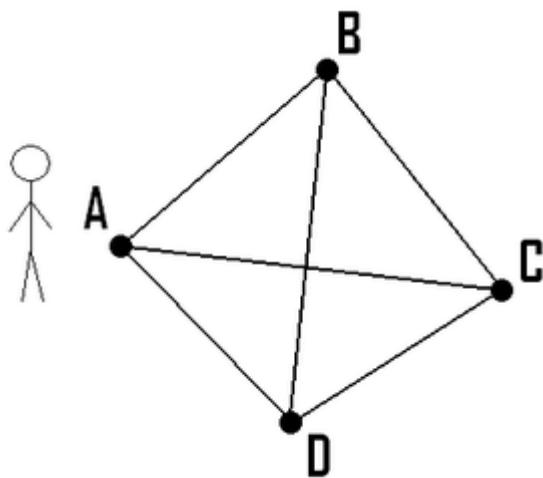
Counting problems ask for the number of solutions of a given instance, for example, given a graph, the number of instances where there are perfect matches. In a counting problem, there's typically an object that we want to check for certain characteristics. Theoretically, we could just search for all possible instances of the characteristics we are looking for. This quickly becomes impractical and can take even the fastest computers a long time to figure out. For this, clever algorithms such as Edmond's algorithm were created. That said, some types of problems can only be solved by going through the entire set and counting matches.

Optimisation problems attempt to find the best answer for a particular input. A very common example of an optimisation problem is when you search for directions on the navigation app on your phone or in your car. The map software uses an optimisation algorithm to choose the best route for you. A lot of times it will return an accurate and reasonably short route to your destination. The goal in an optimisation problem is to output the solution that has the least weight. In this context, "weight" refers to the amount of resources used to arrive at the goal. This is a term that we are going to be using when we formulate algorithms. A popular example of an optimisation problem is Dijkstra's shortest path algorithm which is used to find a minimum cost path to a solution. Cost here refers to the weight and time taken to get to the solution. Optimisation problems are in 2 distinct categories that is, the variable can either be continuous or discrete. Another example is that of a flight scheduling system. An airline will specify the departure and arrival times of aircraft, as well as the number of flights between destinations. Flights are scheduled so that a plane is idle for the shortest possible time, so the algorithm used here attempts to find the most optimal routes for an aircraft in a day, so that the waiting time for passengers is short, and an aircraft does not have to wait for a very long time for its turn to take off.

Characteristics of computational problems

A problem is characterised by task and input instances. Each of the tasks in the set are identical, the only difference being that the items to be operated on may take on different values. An instance of items to be acted upon are called an input instance. Take for example, when adding numbers, the only difference is the actual numbers to be added, otherwise the process of adding numbers is exactly the same for all numbers that will be presented.

If a problem is algorithmic, that is, if we can break it down into step-by-step instructions, and computable, that is, if it can be solved using a computer in a reasonable amount of time, then that problem is tractable. Some problems can only be solved by algorithms in which the execution time grows too quickly in relation to their input to be solved in reasonable time. These algorithms are referred to as intractable. A popular example of this is the Travelling Salesman problem. In this problem, the salesman needs to find the shortest route between a set of cities that must be visited. The salesman needs to keep both travel costs and distance travelled as low as possible



Each of the cities is connected to another by air, road, or rail. The cost of moving from one city to the other is represented by the “weight” of the route. This includes the cost of the plane/train/bus ticket, the length of the route and the time required to travel via that route. In our example above, it is fairly easy to map out A, B, D, C as the most viable sequence for the salesman to follow. As the number of cities increases, it becomes increasingly difficult to map out the most optimal route. This algorithm was defined in the 1800s, so it should be solvable by now, right. Well, powerful as computers are these days, it is currently only possible to solve the problem for a few tens of thousands of cities! The most recorded cities that this problem has been solved for is 85,900 cities in 2006! The Travelling Salesman algorithm is used in the manufacture of circuit boards to determine the order in which a laser can drill holes. It will help the robot to use as little time as possible on the drilling process, cutting production time.

There are many intractable problems that still get solved by the computer. Producing a school timetable is one such problem. The optimal solution needs to be executed in exponential time, that is, the amount of time required to solve the problem increases with the number of variables. Remember we said a problem solved in exponential time will take forever to solve, so it wouldn't be ideal here. A better approach would be to spread lessons unevenly over the timetable, allocate teachers more/ fewer lessons, have some lessons scheduled outside normal time etc. A solution of this kind is referred to as a suboptimal or approximate solution.

A heuristic program is one that can output solutions to a problem in reasonable time, although these solutions might not be optimal or formally correct but can produce good solutions to the problem. These algorithms are

designed to make decisions on how to approach the problem, typically finding a way that having to go through every possible route to find the shortest one to the solution. This can be applied to the Travelling Salesman algorithm. Heuristic algorithms are based on the value of knowledge, experience, and judgement in solving intractable problems.

If we go back to our Travelling Salesman example, the algorithm will take the cheapest route from each city.

All along, we have been referring to problems being solvable in “reasonable” time. This, however, is not the correct term, as reasonable is a very subjective word and means different things to different people. The more appropriate way to describe those problems is that they can be solved in polynomial time. You may remember from math that a polynomial is an algebraic expression that consists of variables and coefficients. We will look at this in detail when we look at algorithms in lesson 8. For now we will keep it simple and just say, a program that is solvable in polynomial time is considered fast because its time cost, in terms of the size n of the problem, is bounded by some polynomial $p(n)$.

If a problem is completely incomputable then it is referred to as unsolvable, or undecidable. For these problems, even the world’s most powerful computers, given infinite time, cannot produce a solution on all inputs. The computer either gives a wrong answer or runs forever without ever giving an answer.

The halting problem is probably the most famous of all undecidable problems. Remember Alan Turing from our first lesson? He is the one who proved that the halting problem does not have a solution. The simplest way to spell out the halting problem is as follows; given a program’s code, and some input, is it possible to tell if the program will halt or run forever on this input without running the program?

If you run the program, then it takes long to produce results, do you immediately know if it is caught in a loop or just needs more time? Short answer, no. a solution to this problem has not been found yet.

The world of maths

In lesson 1, we looked at some of the ways in which the world is made out of mathematical equations. We can even recreate a lot of natural phenomena using mathematical expressions alone! Computer science studies how we use mathematics to describe our world to computers

Relationship between maths and computational problems

Is mathematics really necessary in computer science? This is a very common question, and the answer is pretty much a yes. A lot of the coding that is done in computing is really just mathematical expressions. The bedrock that computer science sits on is comprised of the set theory, the graph theory, the probability theory, the number theory, and a lot of other stuff. Without all this, it would be really difficult to get ourselves to “think” like computers, like computer scientists should. Generally speaking, computer science is a subset of mathematical science.

In previous lessons, we talked about how computers model every single thing mathematically. This is not meant to scare you away. Mathematics is already part of our lives! Think of how you work out your budget, count the teaspoons sugar that you put in your coffee, calculate the amount of time you need to get to the office and so forth. In computer science, the same applies. We model real world problems mathematically.

Mathematics in computer science shapes your way of thinking so that you are better able to “read computers’ minds”. It is no secret that you do not necessarily need to be a math guru in order to be a programmer. However, programming is not the same as computer science. There is a limit to how far you can dive into programming without in depth knowledge of mathematics. Software engineering and development of most innovative

technologies often does require advanced math. Most of the algorithms that are running today began their lives as a bunch of complex math equations, which were proven then broken down to produce algorithm. A good example of this is the code that controls your cell phone signal. You would probably feel dizzy just looking at the code!

Math is more than just numbers; it is a logical foundation on which most of science is built upon. Studying mathematics builds analytical, critical thinking and reasoning skills, all of which will help you to write awesome code. Our approach in this course has been and will be throwing it in as a practical part of the code that you will be writing. As you progress with computer science, you will learn how to pull code from mathematical theories! Some of the code that we will write will be quite literally, a bunch of mathematical equations, so, well, you have just got to love your math! Most of the topics that we covered so far can only really be covered in greater detail using math and code!

Mathematical modelling

Mathematical modelling is the art of translating problems from an application area into tractable mathematical formulations whose theoretical and numerical analysis provides insight, answers, and guidance useful for the originating application.

Mathematical modelling aims to develop scientific understanding through quantitative expression of current knowledge of a system. By adding all the information that we know about our model, other things that we are not aware of will come up. It is easier to test the effect of changes in a system. In lesson 1 we talked about computers in the 60s being used to design wind tunnels. This had the direct effect of considerably improving vehicle aerodynamics, and cars are now so much more efficient as a result.

There are 2 types of mathematical models: stochastic models and deterministic models. Stochastic models are statistical in nature and may predict the distribution of possible outcomes. Deterministic models kick out random variation and always predict the same outcome from a given starting point.

Models can also be classified based on the level of understanding on which the model is based. A model which has loads of theoretical information generally describes what happens in one level of a hierarchy by considering all the little bits happening at lower levels is called a mechanistic model. Empirical models account for the changes that happen within the model because of changes in conditions

	empirical	mechanistic
Deterministic	Predicting cattle growth from a regression relationship with feed intake	Planetary motion, based on Newtonian mechanics (differential equations)
Stochastic	Analysis of variance of variety yields over sites and years	Genetics of small populations based on Mendelian inheritance (probabilistic equations)

A binary problem!

In lesson 2, we looked at how computers use encoding standards to represent data. This gave rise to a number of standards being produced. But then that created another problem. How does the computer know which encoding is being used?

To get a picture of how this is a real problem, let's take a closer look at how this happens.

Computers store text as a sequence of numbers where each character has a unique number according to an agreed upon "character encoding standard". The problem is that there are many standards, like we discussed in previous lessons, and each standard assigns different numbers to the same character. "ä" is for instance stored as 228 in the popular ISO-8859-1 standard but stored as the two-byte number 50084 in the UTF-8 standard (written out as c3a4 with hexadecimal notation). If a UTF-8 encoded "ä" is interpreted according to the ISO-8859-1 standard, it shows up as the character pair "Ã¤".

You may have come across this problem when a computer displays a bunch of weird characters.

To solve this problem, we need to come up with an algorithm that allows computers to "agree" on which encoding to use. A good example of where this problem has an implemented solution is the HTTP header. The first few bytes of data transmitted when you access a website sets the standards that will be used for the rest of the communication session.

Common computing problems

Merging language and mathematics

Mathematics is a very precise language that helps us formulate ideas while identifying assumptions and ambiguity. One good thing about maths is it is a very concise language; its rules are well defined and make it great for replication of models.

In order to be considered a language, a system of communication must have vocabulary, grammar, syntax, and people who use and understand it.

Mathematics, believe it or not, checks all the required boxes! Some people do not consider mathematics a language but acknowledge its use as a written rather than spoken form of communication. Maths is a universal language, especially so in computer science. The symbols and organization to form equations are the same in every country of the world.

Formulating problems

One popular problem is the Dining Philosophers problem. In this hypothetical problem, five philosophers are sitting around a circular table for a meal of spaghetti. For any philosopher to eat spaghetti, they must have two forks: one in the left hand and one in the right hand. However, they have just five forks total on the table.

To coordinate their meal, the philosophers agree to strictly follow these steps:

Think until the left fork is available, and then pick it up.

Think until the right fork is available, and then pick it up.

Eat for 30 seconds.

Put the right fork down.

Put the left fork down.

Repeat these steps.

How can we make sure that every 30 seconds, at least one philosopher gets to eat in order to keep thinking?



In case you are wondering, the Dining Philosophers problem is used to solve an important problem that is in action right now on the device that you are using to watch this lesson, multithreading. You may remember multithreading from when we covered operating systems; this is where several programs are run by the same processor, each getting a share of the processor's time while appearing to the user as though they were being run at the same time. When we went through the problem, you saw how the philosophers were all kept happy by getting a chance to eat their spaghetti. The processor uses the same approach to make sure that all running programs have their share of CPU time. This is also used when several devices want to access the same resource, such as storage. pretty cool, right?! The next works in pretty much the same way, except that this time, you have to figure it out on your own. In the next lesson, we will be looking at ways in which to write a solution given a problem such as the following one.

In our next example, we are going to look closely at the Travelling Salesman example that we talked about earlier. To get a clearer picture of what exactly needs to be done, we first state the problem as clearly as possible.

Let's now look at the attributes that our solution should have. This helps us keep track of what our algorithm needs to achieve.

- The salesman needs to follow the shortest path through all cities

- The salesman needs to come back to the first city after having gone through all cities.

Let's now look at one possible solution to this problem. It is not the only solution though; other solutions have been implemented. The brute force approach calculates all possible permutations of routes. It then draws all possible routes, calculating the total "cost" of taking each route. The total distances are then compared and the shortest one is chosen.

Here you have seen how we broke down the problem into stages to make it easier to solve. This will help us to write a clear algorithm.

Let's look at one last problem. You are given 2 jugs; one can hold 4 litres and the other can hold 3 litres. None of the jugs have measuring markers on them. how would you get exactly 2 litres of water in the 4-litre jug?

When formulating a problem, it needs to satisfy 4 conditions.

1. It needs to be precise, that is unambiguous
2. The initial state and goal state need to be clear

3. The rules need to be laid out
4. All components of the problem need to be mentioned.

For our Water Jug problem, the end goal is to get 2 litres of water into the 4-litre jug. In our next lesson, we will further analyse this problem and try to come up with a solution.

Summary

In this lesson, we learnt about the various types of problems that are encountered in computing. We also looked at how some of those problems are approached when coming up with solutions. We also looked at the relationship between mathematics and computer science. We also looked at how to formulate problems.

References

References

- Cook, K. (2018). What's The Main Importance Of Mathematics In Computer Science? [online] Houseofbots.com. Available at: <https://www.houseofbots.com/news-detail/3864-1-whats-the-main-importance-of-mathematics-in-computer-science>
- Docscopy.com. (2014). Great Algorithms that Revolutionized Computing - Docscopy. [online] Available at: <https://www.docscopy.com/en/news/interesting-facts/great-algorithms-revolutionized-computing/>
- Everythingcomputerscience.com. (2013). CS Counting. [online] Available at: https://everythingcomputerscience.com/discrete_mathematics/Counting.html
- Ma, S. (2020). Understanding the Travelling Salesman Problem (TSP). [online] Routific.com. Available at: <https://blog.routific.com/travelling-salesman-problem#>
- Marion, G., Bioinformatics and Scotland, S. (2015). An Introduction to Mathematical Modelling. [online] Available at: https://people.maths.bris.ac.uk/~madjl/course_text.pdf.
- Codeofthedamned.com. (2017). Why Does a CS Degree Require So Much Math? [online] Available at: <http://codeofthedamned.com/index.php/cs-degree-why-all-of>
- Marion, G., Bioinformatics and Scotland, S. (2015). An Introduction to Mathematical Modelling. [online] Available at: https://people.maths.bris.ac.uk/~madjl/course_text.pdf.
- MariusRangu (2007). PII: B978-0-12-226551-8.50002-8. [online] Available at: <https://www.sfu.ca/~vdabbagh/Chap1-modeling.pdf>.
- Msu.edu. (2020). Problems in Computer Science. [online] Available at: <http://www.cse.msu.edu/~torng/360Book/Problems/#LANGUAGE>
- Nist.gov. (2012). decision problem. [online] Available at: <https://xlinux.nist.gov/dads/HTML/decisionProblem.html>
- Pacha-Sucharzewski, M. (2011). Analysis of the “Travelling Salesman Problem” and an Application of Heuristic Techniques for Finding a New Solution. Undergraduate Review, [online] 7, pp.81–86. Available at: https://vc.bridgew.edu/cgi/viewcontent.cgi?article=1192&context=undergrad_rev
- Pedram Ataee, PhD (2020). How to Solve the Traveling Salesman Problem - A Comparative Analysis | Towards Data Science. [online] Medium. Available at: <https://towardsdatascience.com/how-to-solve-the-traveling-salesman-problem-a-comparative-analysis-39056a916c9f>
- Univie.ac.at. (2020). Mathematical Modelling. [online] Available at: <https://www.mat.univie.ac.at/~neum/model.html#:~:text=Mathematical%20modeling%20is%20the%20art,is%20indispensable%20in%20many%20applications>
- Wolfram.com. (2020). Polynomial Time. [online] Available at: <https://mathworld.wolfram.com/PolynomialTime.html>