CH 01 : INTRODUCTION TO ALGORITHMS

By Dr. LOUNNAS Bilal

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 CONTENTS OF THIS COURSE

Advanced Algorithms

This course is a first-year of Master degree (SIGL). Emphasis is placed on fundamental algorithms and advanced methods of algorithmic design, analysis, and implementation. Subjects to be covered include basices of algorithms, analysing algorithms which include how to calculate complexity and classification of problems, data structures, recursions methods, and other advanceds algorithms such as sorting, graphs, hashage, text...etc.

The course will be divided into four (4) chapters

Ch 01 Introduction of algorithms: This chapter contains some definitions and basics of algorithms in computer science
- This chapter will take between 2 and 4 sessions.
Ch 02 Algorithms analysis: This chapter contains some definitions and basics of algorithms analysis, it also talk about algorithm complexity, and different class of problems... ext
- This chapter will take between 2 and 4 sessions.
Ch 03 Review of algorithmic techniques and data structures: This chapter will recall some techniques such as recursion, and we will talk about basic data structures such as pointer, table, list, ...ext
- This chapter will take between 2 and 4 sessions.
Ch 04 Advanced techniques in algorithms: This chapter will focus on the advenced techniques of some well known algorithms in differents areas such sorting, hashage, text, graphs
- This chapter will take between 2 and 4 sessions.

2 BASICS OF ALGORITHMS

2.1 What is algorithms?

To make a computer do anything, you have to write a computer program. To write a computer program, you have to tell the computer, step by step, exactly what you want it to do. The computer then "executes" the program, following each step mechanically, to accomplish the end goal.

When you are telling the computer what to do, you also get to choose how it's going to do it. That's where computer algorithms come in. The algorithm is the basic technique used to get the job done.

Let's say that you have a friend arriving at the airport, and your friend needs to get from the airport to your house. Here are four different algorithms that you might give your friend for getting to your home:
The taxi algorithm:

- Go to the taxi stand.
- Get in a taxi.
- Give the driver my address.

The call-me algorithm:

- When your plane arrives, call my cell phone.
- Meet me outside baggage claim.

The rent-a-car algorithm:

- Take the shuttle to the rental car place.
- Rent a car.
- Follow the directions to get to my house.

The bus algorithm:

- Outside baggage claim, catch bus number 70.
- Transfer to bus 14 on Main Street.
- Get off on Elm street.
- Walk two blocks north to my house.

All four of these algorithms accomplish exactly the same goal, but each algorithm does it in completely different way. Each algorithm also has a different cost and a different travel time. Taking a taxi, for example, is probably the fastest way, but also the most expensive. Taking the bus is definitely less expensive, but a whole lot slower. You choose the algorithm based on the circumstances.

In computer programming, there are often many different ways – algorithms – to accomplish any given task. Each algorithm has advantages and disadvantages in different situations.

2.2 Why should we care?

Imagine for a second you have two Rubik's Cubes (Figure 1 on the following page) sitting out in front of you. One of them you are allowed to use algorithms (like how many times or which direction to turn a face), and the other you have to find your own way.

Which way will be faster?

Almost certainly the the first way. There are already hundreds of well-established algorithms to solve a Rubik's cube, and trying to invent your own method to solve it will take much longer than utilizing the current ones. Worse than that, it is most likely that the method you choose will be less efficient and slower than the methods that already exist. But imagine that there are no well-established algorithms to solve Rubik's cube problem yet, however, your algorithm with its bad performance will be counted as the first well-established algorithm for solving cube problem no matter how good this algorithm is.

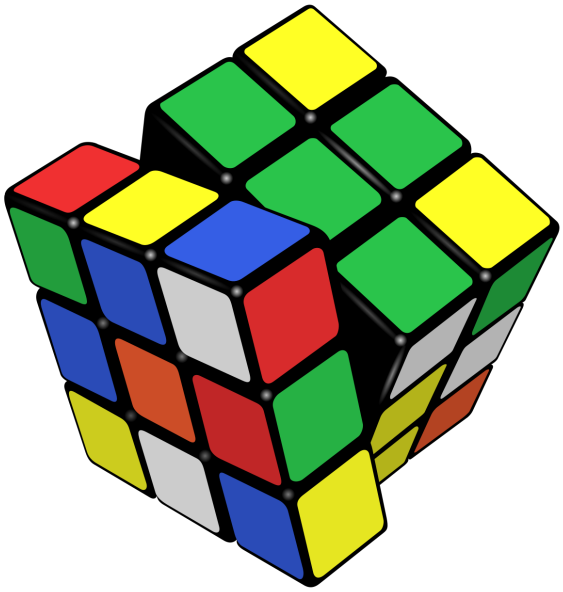

Figure 1: Rubik's Cubes

2.3 History of algorithmic

Etymologically, the word "algorithm" is a combination of the Latin word algorismus, named after Al-Khwarizmi, a 9th-century Persian mathematician, and the Greek word arithmos, meaning "number". In English, it was first used in about 1230 and then by Chaucer in 1391. English adopted the French term, but it wasn't until the late 19th century that "algorithm" took on the meaning that it has in modern English.

Another early use of the word is from 1240, in a manual titled Carmen de Algorismo composed by Alexandre de Villedieu.

2.4 Definition of algorithmic

An informal definition could be "a set of rules that precisely defines a sequence of operations." which would include all computer programs, including programs that do not perform numeric calculations. Generally, a program is only an algorithm if it stops eventually.

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming

language.

A prototypical example of an algorithm is the Euclidean algorithm to determine the maximum common divisor of two integers; an example (there are others) is described by the flow chart (Figure 2)

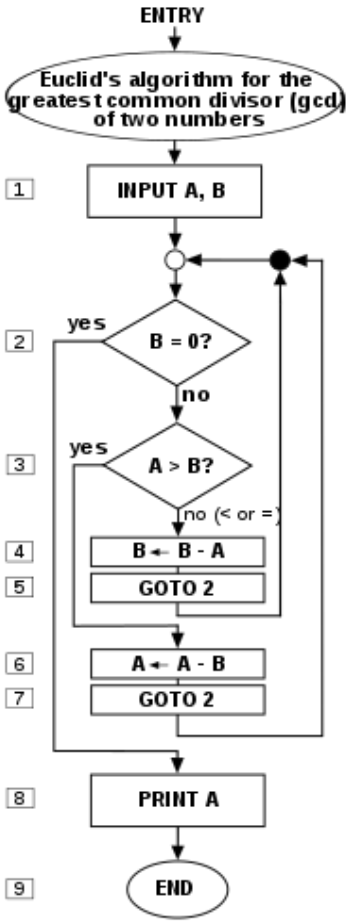

Figure 2: Flow chart of an algorithm (Euclid's algorithm)

3 MOVING FORWARD WITH ALGORITHMICS

3.1 How to write an algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These com-

mon constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example
Let's try to learn algorithm-writing by using an example. Problem - Design an algorithm to add two numbers and display the result.

```
step 1 - START
step 2 - declare three integers a, b & c
step 3 - define values of a & b
step 4 - add values of a & b
step 5 - store output of step 4 to c
step 6 - print c
step 7 - STOP
```

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as :

```
step 1 - START ADD
step 2 - get values of a & b
step 3 - c ← a + b
step 4 - display c
step 5 - STOP
```

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways (Figure **??** on the following page) .

3.2 Become a good algorithmic designer

Algorithms are not a special type of operation, necessarily. They are conceptual, a set of steps that you take in code to reach a specific goal.

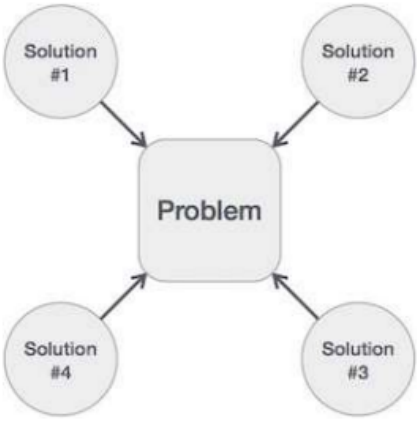

Figure 3: Multiple solution for one problem

For that : There is no better way to become a better algorithm designer than to have a deep understanding and appreciation for algorithms..

So, how can you become a good algorithmic design?

1. **Know your input:** One of the main principles of algorithmic design is to, if possible, build your algorithm in such a way that the input itself does some of the work for you. For instance, if you know that your input is always going to be numbers, you do not need to have exceptions/checks for strings.

2. **Understand your tools:** Understanding your tools means that you understand what each line of code does, both immediately (the return value of a function or the effect of a method) and implicitly (how much overhead is associated with running a library function, or which is the most efficient method for concatenating a string). To write great algorithms, it is important to know the performance of lower-level functions or utilities, not just the name and implementation of them. Understand the Environment

3. **Understand the environment:** Designing efficient algorithms is a full-engagement undertaking. Beyond understanding your tools as a standalone piece, you must also understand the way that they interact with the larger system at hand.

4. **Reducing the workload:** In general, the goal of algorithm design is to complete a job in fewer steps. When you write your code, take into consideration all of the simple operations the computer is taking to reach the goal.

5. **Study Advanced Techniques:** There is no better way to become a better algorithm designer than to have a deep understanding and appreciation for algorithms.

3.3 Characteristics of algorithms

Not all procedures can be called an algorithm. An algorithm should have the following characteristics :

1. **Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
2. **Input:** An algorithm should have 0 or more well-defined inputs.
3. **Output:** An algorithm should have 1 or more well-defined outputs, and should match the desired output.
4. **Finiteness:** Algorithms must terminate after a finite number of steps.
5. **Feasibility:** Should be feasible with the available resources.
6. **Independent:** An algorithm should have step-by-step directions, which should be independent of any programming code.

4 CLASSIFICATION

There are various ways to classify algorithms, each with its own merits.

4.1 By implementation

One way to classify algorithms is by implementation means.

4.1.1 *Recursion*

A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition (also known as termination condition) matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other. For example, towers of Hanoi (Figure **??**) is well understood using recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.

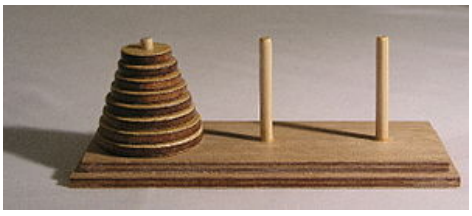

Figure 4: A model set of the Tower of Hanoi (with 8 disks)

4.1.2 *Logical*

An algorithm may be viewed as controlled logical deduction. This notion may be expressed as: Algorithm = logic + control. The logic component expresses the axioms that may be used in the computation and the control component determines the way in which deduction is applied to the axioms. This is the basis for the logic programming paradigm. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant semantics: a change in the axioms has a well-defined change in the algorithm.

4.1.3 *Serial, parallel or distributed*

Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms or distributed algorithms. Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilize multiple machines connected with a network. Parallel or distributed algorithms divide the problem into more symmetrical or asymmetrical subproblems and collect the results back together. The resource consumption in such algorithms is not only processor cycles on each processor but also the communication overhead between the processors. Some sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems.

4.1.4 *Deterministic or non-deterministic*

Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithms solve problems via guessing although typical guesses are made more accurate through the use of heuristics.

4.1.5 *Exact or approximate*

While many algorithms reach an exact solution, approximation algorithms seek an approximation that is closer to the true solution. Approximation can be reached by either using a deterministic or a random strategy. Such algorithms have practical value for many hard problems. One of the examples of an approximate algorithm is the Knapsack problem. The Knapsack problem is a problem where there is a set of given items. The goal of the problem is to pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number X. So, we must consider weights of items as well as their value.

4.1.6 *Quantum algorithm*

They run on a realistic model of quantum computation. The term is usually used for those algorithms which seem inherently quantum, or use some essential feature of quantum computation such as quantum superposition or quantum entanglement.

4.2 By design paradigm

Another way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories include many different types of algorithms. Some common paradigms are:

4.2.1 *Brute-force or exhaustive search*

This is the naive method of trying every possible solution to see which is best.

4.2.2 *Divide and conquer*

A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in the conquer phase by merging the segments. A simpler variant of divide and conquer is called a decrease and conquer algorithm, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple subproblems and so the conquer stage is more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is the binary search algorithm.

4.2.3 *Search and enumeration*

Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes search algorithms, branch and bound enumeration and backtracking.

4.2.4 *Randomized algorithm*

Such algorithms make some choices randomly (or pseudo-randomly). They can be very useful in finding approximate solutions for problems where finding exact solutions can be impractical (see heuristic method below). For some of these problems, it is known that the fastest approximations must involve some randomness. Whether randomized algorithms with polynomial time complexity can be the fastest algorithms for some problems is an open question known as the P versus NP problem. There are two large classes of such algorithms:

4.2.5 *Reduction of complexity*

This technique involves solving a difficult problem by transforming it into a better known problem for which we have (hopefully) asymptotically optimal algorithms. The goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithm's. For example, one selection algorithm for finding the median in an unsorted list involves first sorting the list (the expensive portion) and then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as transform and conquer.

4.3 Optimization problems

For optimization problems there is a more specific classification of algorithms; an algorithm for such problems may fall into one or more of the general categories described above as well as into one of the following:

4.3.1 *Linear programming*

When searching for optimal solutions to a linear function bound to linear equality and inequality constraints, the constraints of the problem can be used directly in producing the optimal solutions. There are algorithms that can solve any problem in this category, such as the popular simplex algorithm. Problems that can be solved with linear programming include the maximum flow problem for directed graphs. If a problem additionally requires that one or more of the unknowns must be an integer then it is classified in integer programming. A linear programming algorithm can solve such a problem if it can be proved that all restrictions for integer values are superficial, i.e., the solutions satisfy these restrictions anyway. In the general case, a specialized algorithm or an algorithm that finds approximate solutions is used, depending on the difficulty of the problem.

4.3.2 *Dynamic programming*

When a problem shows optimal substructures meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, a quicker approach called dynamic programming avoids recomputing solutions that have already been computed. For example, FloydWarshall algorithm, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is that subproblems are more or less independent in divide and conquer, whereas subproblems overlap in dynamic programming. The difference between dynamic programming and straightforward recursion is in caching or memoization of recursive calls. When subproblems are independent and there is no repetition, memoization does not help; hence dynamic programming is not a solution for all complex prob-

lems. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

4.3.3 *The greedy method*

A greedy algorithm is similar to a dynamic programming algorithm in that it works by examining substructures, in this case not of the problem but of a given solution. Such algorithms start with some solution, which may be given or have been constructed in some way, and improve it by making small modifications. For some problems they can find the optimal solution while for others they stop at local optima, that is, at solutions that cannot be improved by the algorithm but are not optimum. The most popular use of greedy algorithms is for finding the minimal spanning tree where finding the optimal solution is possible with this method. Huffman Tree, Kruskal, Prim, Sollin are greedy algorithms that can solve this optimization problem.

4.3.4 *The heuristic method*

In optimization problems, heuristic algorithms can be used to find a solution close to the optimal solution in cases where finding the optimal solution is impractical. These algorithms work by getting closer and closer to the optimal solution as they progress. In principle, if run for an infinite amount of time, they will find the optimal solution. Their merit is that they can find a solution very close to the optimal solution in a relatively short time. Such algorithms include local search, tabu search, simulated annealing, and genetic algorithms. Some of them, like simulated annealing, are non-deterministic algorithms while others, like tabu search, are deterministic. When a bound on the error of the non-optimal solution is known, the algorithm is further categorized as an approximation algorithm.

4.4 By field of study

Every field of science has its own problems and needs efficient algorithms. Related problems in one field are often studied together. Some example classes are search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, computational geometric algorithms, combinatorial algorithms, medical algorithms, machine learning, cryptography, data compression algorithms and parsing techniques.

Fields tend to overlap with each other, and algorithm advances in one field may improve those of other, sometimes completely unrelated, fields. For example, dynamic programming was invented for optimization of resource consumption in industry, but is now used in solving a broad range of problems in many fields.

4.5 By complexity

Algorithms can be classified by the amount of time they need to complete compared to their input size:

1. Constant time: if the time needed by the algorithm is the same, regardless of the input size. E.g. an access to an array element.
2. Linear time: if the time is proportional to the input size. E.g. the traverse of a list.
3. Logarithmic time: if the time is a logarithmic function of the input size. E.g. binary search algorithm.
4. Polynomial time: if the time is a power of the input size. E.g. the bubble sort algorithm has quadratic time complexity.
5. Exponential time: if the time is an exponential function of the input size. E.g. Brute-force search.

Some problems may have multiple algorithms of differing complexity, while other problems might have no algorithms or no known efficient algorithms. There are also mappings from some problems to other problems. Owing to this, it was found to be more suitable to classify the problems themselves instead of the algorithms into equivalence classes based on the complexity of the best possible algorithms for them.