



Discrete Mathematics

Assignment: Group Project

Topic: Algorithmic Paradigms

Professor: Manuele Leonelli

Max Heilingbrunner, Pablo Ostos Bollmann, Aikaterini Orlova,
Niccolo' Matteo Borgato, Wendy Quarshie, Joseph Guss

TABLE OF CONTENTS

<u>1.</u>	<u>INTRODUCTION: ALGORITHMIC PARADIGMS</u>	<u>3</u>
<u>2.</u>	<u>ALGORITHM NO. 1: BRUTE FORCE</u>	<u>3</u>
<u>3.</u>	<u>ALGORITHM NO. 2: GREEDY</u>	<u>4</u>
<u>4.</u>	<u>ALGORITHM NO. 3: DYNAMIC PROGRAMMING.....</u>	<u>5</u>
<u>5.</u>	<u>COMPARISON AND BENCHMARKING</u>	<u>7</u>
<u>6.</u>	<u>CONCLUSION.....</u>	<u>8</u>

1. Introduction: Algorithmic Paradigms

An *algorithm* is a finite sequence of detailed instructions that allow us to solve a problem or perform a computation. A *paradigm* is a step-by-step thought process which will govern a scientific apprehension for a certain period. In essence, *algorithmic paradigms* describe a pattern of how to approach various problem-solving techniques. It is a framework that underlines the creation of a group of algorithms. Similar to how an algorithm is superior to a computer program, an algorithmic paradigm is superior to the concept of an algorithm. For instance, when approaching a more complex problem it is important to outline exactly how to solve it. When it comes to computer applications there is rarely a “one-size-fits-all” approach, therefore a good programmer would know several ways (algorithmic paradigms) to solve the same issue. These paradigms would in turn outline the general framework of the proposed solution. Once there is the concept, you may focus your thinking in that specific direction. Considering a paradigm is a “thought process” there are many known and established algorithmic paradigms that allow us to think of a solution to different problems. Some examples include:

1. *Brute force* – this is the most commonly used framework, which focuses on identifying whether a solution to a problem exists or not
2. *Divide and conquer* – this approach breaks down the problem into smaller sub-problems which are not overlapping and starts solving them one by one to reach the overall solution
3. *Greedy* – through this framework we can identify the most optimal solution to solving a problem at every step of the way, to reach an optimal solution for the overall problem
4. *Recursion* – this algorithmic paradigm identifies repeating patterns in a problem and takes advantage of a terminal condition to work the way up to the top
5. *Dynamic Programming* – similar to divide and conquer, this framework looks for the optimal substructure and solves overlapping sub-problems first to find the optimal solution to the main problem

These are amongst some of the most commonly known algorithmic paradigms with each one having its advantages and disadvantages in the implementations. Below we focus more precisely on the three types of algorithmic paradigms and how they can be compared to one another.

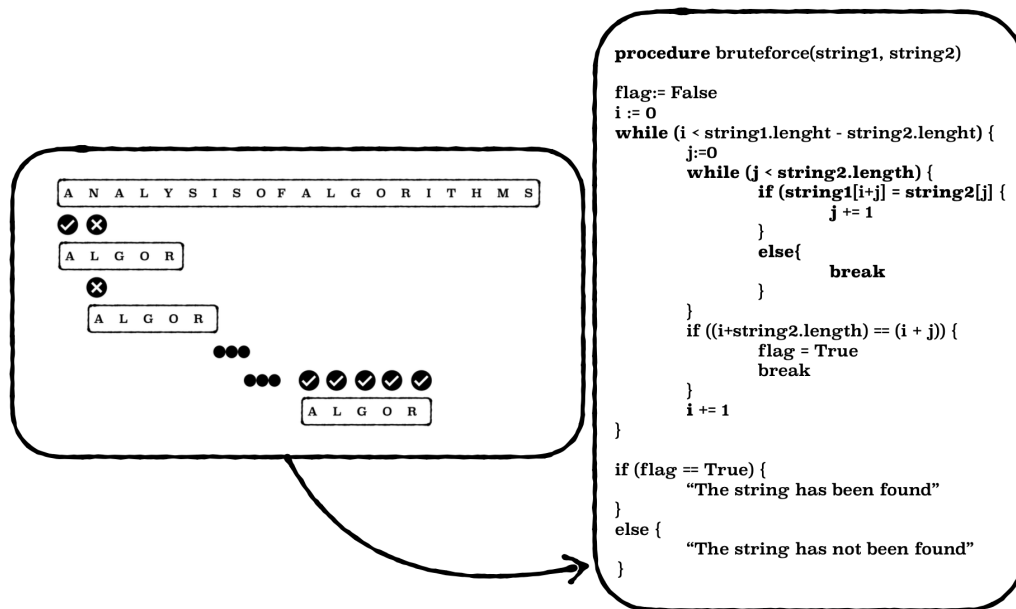
2. Algorithm No. 1: Brute Force

Brute force algorithms are often referred to as the most simple heuristic methods and can find the solution to any problem no matter the complexity. The simplicity of this paradigm comes from its main principle. The fundamental idea behind this paradigm is to exploit every combination and explore all possible paths until the conditions which are identical to the solution are identified. After going through one likely scenario and determining if it is the solution or not, the algorithm will initiate another combination until the solution is found. This algorithm neither focuses on the quality of the solution, whether it is better or worse, nor its efficiency. It aims to find the right solution by examining every possibility.

The implementation of this paradigm is as straightforward as its concept since it always finds the solution if it exists. Nevertheless, there are some drawbacks to this class of algorithms that need to be addressed. In the first place, the computational power cost is proportional to the number of candidate solutions, in other words, to the problem’s size. Furthermore, the time complexity of the algorithm is generally high. In simpler words, if the problem consists of searching for a pattern of characters of size “m” in an array of characters of size “n”, it would require the algorithm “m*n” attempts. This means that algorithms that have a time complexity of $O(\log(n))$ or $O(1)$ will be preferable in terms of efficiency.

The logic, or procedure, that this algorithm follows to solve the previously described problem implies the comparison of the pattern, character by character until it finds a mismatch. If this is the case, the next scenario is evaluated, and the process is repeated. The procedure finishes when the solution is found, or all possibilities are tested.

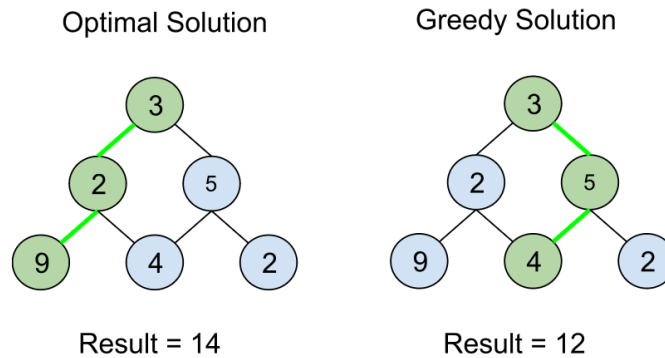
Although the brute force paradigm has its advantages and disadvantages, it is considered the starting line for more efficient algorithms, and the base for their comparison. Moreover, there are specific cases where the use of brute force algorithms may be more convenient. For example, search by brute force is usually applied when the number of potential solutions is low, or else, when this number has been previously reduced by some other heuristic method. Another scenario where this algorithm might be more convenient is when a simpler implementation is required rather than a more time-efficient one.



Graph 1: Pseudo code for brute force algorithm in action

3. Algorithm No. 2: Greedy

When it comes to **greedy algorithms**, the defining characteristic is that they make the most locally optimal choice at each stage of the decision-making process to find a global optimum. The key phrase is that the decision is made at each stage is *locally optimal*, as a greedy algorithm, many times will not be globally optimal. A very simple example to visualize a greedy algorithm in practice is to apply it to the travelling salesman problem. The travelling salesman problem asks: given a list of cities and the distances between them, what is the shortest route to visit every city one time and then finally return to the original city? If we reconsider the defining characteristic of greedy algorithms which states that the most locally optimal choice will be made, then it is clear that the greedy algorithm will simply choose the nearest unvisited city at every step in the decision-making process. While there are certain instances where using the greedy algorithm can yield globally optimal solutions or at least solutions close to being globally optimal, concerning the travelling salesman problem, this is not one of them. On average, the algorithm will produce a result that takes 25% longer than the global optimum. There are many arrangements of cities in which the greedy algorithm uniquely produces the worst possible solution available.



Graph 2: Example -for this problem the goal is to maximize the result and the greedy algorithm takes the most locally optimal route at each step of the decision process, but it does not yield the globally optimal solution.

With that being said, there are two properties which, if satisfied, make greedy algorithms optimal to use: the greedy choice and the optimal substructure properties. The greedy choice property implies that a global optimum can be reached by selecting a local optimum. The optimal substructure property says that an optimal solution to the problem contains an optimal solution to its sub-problems. In addition, while greedy algorithms may not always be globally optimal, they have practical advantages over other more advanced algorithms in the real world. The heuristic of making the most locally optimal choice is fairly straightforward, and as a result, implementing a greedy algorithm has the benefits of being easy to implement and having very reasonable runtime complexity relative to other more complex and exhaustive algorithms. Therefore in many situations when time and computational resources are not abundant, solving a problem using a greedy algorithm may be the most efficient. In essence, it is critical to understand the problem as well as the nuance of the situation when considering whether or not to employ a greedy algorithm as a solution.

4. Algorithm No. 3: Dynamic Programming

The algorithmic paradigm of **dynamic programming** is simple to understand: finding an optimal solution to a problem by solving overlapping sub-problems based on an optimal substructure. A well-known example of **dynamic programming** is the formulation of the Fibonacci sequence (1, 1, 2, 3, 5, ...). When thinking about finding the optimal solution for our given problem, a classic recursion might be a beneficial choice. Recursion happens when an algorithm calls, for example, a function repetitively, directly or indirectly. However, the implementation of a standard recursive function increases the computational complexity (time and space) depending on the size of the data. This suggests that the underlying algorithm is solving a smaller part of the problem multiple times, over and over again, leading to a Big-O of 1.6^n as $\text{Fibonacci}(n) = O(1) + \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$. How can the computational complexity be decreased?

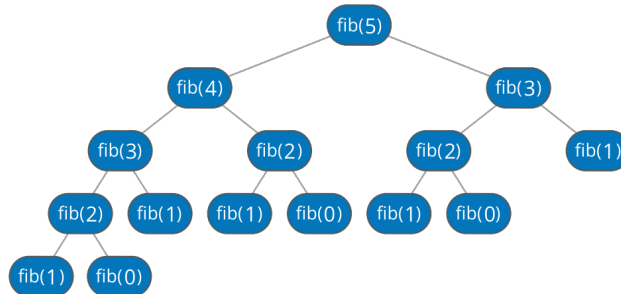
```
function Fibonacci(n):
  if n == 1 or n == 2:
    number = 1
  else:
    number = Fibonacci(n - 1) + Fibonacci(n - 2)
  return number
```

Graph 3: Adapted pseudo code for Fibonacci sequence based on recursion (Source: CS Dojo YouTube)

In this case, dynamic programming may be suitable to solve this problem more efficiently. With dynamic programming, the main problem gets separated into smaller problems, also called sub-problems. To solve problems efficiently with dynamic programming, two underlying criteria must be taken into account:

1. **Overlapping sub-problems**
2. **Optimal substructure**

For the former, the overlapping sub-problems, the solution to the main problem is based on several, redundant sub-problems. Based on the following graph, each sub-problem is based on a previous subproblem, e.g. Fibonacci(2) is based on Fibonacci(1) and Fibonacci(0).



Graph 4: Optimal substructure and overlapping sub-problems to estimate Fibonacci sequence

For the latter, the optimal substructure, the requirement is to aggregate each optimal choice of all sub-problems to identify the optimal solution for the main problem.

To implement dynamic programming, there exist two main techniques:

1. **Top-Down approach**
2. **Bottom-Up approach**

The „Top-Down“ approach can also be referred to as „memoization“. With memoization, values get stored, for example in a list, and when solving each sub-problem, the algorithm stores already solved values in the list and refers to them each time. This technique reduces the time and space complexity of the algorithm compared to a classic recursion. As a result, it leads to a time and space complexity of $O(n)$ as $\text{Fibonacci}(n) = \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$.

```

function Fibonacci(n, list):
    if list[n] != 0:
        return list[n]
    elif n == 1 or n == 2:
        number = 1
    else:
        number = Fibonacci(n - 1) + Fibonacci(n - 2)
    list[n] = number
    return number
    
```

Graph 4: Adapted pseudo code for Fibonacci sequence based on memoization (Source: CS Dojo YouTube)

The „Bottom-Up“ approach can be referred to as „tabulation“ and can be classified as an iterative technique. Hence, each sub-problem is going to be solved first and then all solutions are aggregated

to conclude with an optimal solution for the main problem. Using this technique, the efficiency of the time complexity stays the same as $O(n)$, but for the space complexity, it changes to $O(1)$. This is because only values get added to a list, similar to a stack, and the Fibonacci() function is not recursively called anymore.

```
function Fibonacci(n):
    if n == 1 or n == 2:
        return 1
    number = [n + 1]
    number [1] = 1
    number [2] = 1

    for i in range(3,n):
        number[i] = number[i - 1] + number[i - 2]
    return number
```

Graph 5: Adapted pseudo code for Fibonacci sequence based on tabulation (Source: CS Dojo YouTube)

In the real world, dynamic programming is used to estimate the optimal route in Google Maps for a user. To conclude, the various techniques of dynamic programming can have advantages in time and space complexity in contrast to a classic recursion.

5. Comparison and Benchmarking
















Among the many different types of algorithms, we have presented three examples that we think are the most significant for this paper. The brute force, greedy and dynamic programming algorithms serve the same purpose of finding the right solution but follow very different paths to achieve it. It is then interesting to compare the characteristics of the three algorithms presented and understand which one might be better suited for each specific case solution.

To compare algorithms we are going to take into consideration the complexity of the algorithm, its time complexity, its efficiency, iterations and storage needed to perform them.

Brute Force and Greedy algorithms are among the most commonly used, yet they differ in various ways. Brute Force guarantees that an optimal solution will be found while Greedy does not. Greedy, though has a lower time complexity and is more efficient and cost-effective

By comparing Brute Force algorithms and Dynamic Programming Algorithms we can notice that, while Dynamic Programming is a much more complex algorithm, it provides several advantages. First of all Dynamic Programming has a much lower Time Complexity thanks to the lower number of iterations. This makes it far more efficient and fast. Nevertheless, to have all these advantages Dynamic Programming requires more storage which could then result in higher costs when compared to Brute Force.

Similarly, comparing Greedy and Dynamic Programming Algorithms we notice differences. First of all, Dynamic Programming guarantees that an optimal solution will be generated, something that is not guaranteed in Greedy algorithms. Greedy to its advantage generally has a lower time complexity and is more efficient in terms of memory.

	Complexity	Optimal Solution	Time Complexity	Storage	Cost Effective
Brute Force					
Greedy					
Dynamic Programming					

Graph 6: Evaluation of presented algorithmic paradigms based on relevant criteria

The algorithms presented in this paper differ in many different ways and each is more specific to a single use case. For small problems, not so complex Brute Force is probably the best choice, but for problems that can be divided into smaller similar sub-problems, Dynamic Programming is better suited. Differently for optimization problems, Greedy is the best solution. In conclusion, by understanding what is the final goal of the algorithm and what are your resources it will be easy to choose the best solution.

6. Conclusion

A good algorithm should be easy to implement and efficient. It is also important to consider other properties such as having specified input and output, running efficiently and terminating but most importantly, being correct. Most often, there may be more than one technique applicable to solving a problem, however, in most cases, algorithms that are developed using one approach perform better than equivalent algorithms with alternate techniques. The performance of an algorithm is dependent on the type and structure of the data.

Designing algorithms don't have to be complicated. Even the simplest things we do, like organizing our papers or Googling questions, are significant parts of our daily lives. In this paper, we reviewed three algorithms, all of which have unique real-world applications;

1. Brute Force: A security threat to guess a password using commonly used passwords, or even guessing a 4-digit pin by trying all possible combinations.
2. Greedy Algorithm: CPU scheduling makes use of this algorithm to maximize the utilization of the CPU by scheduling the processes in the CPU.
3. Dynamic Programming: Finding the shortest route from your office to the nearest supermarket

These algorithms however all have their various limitations. Dynamic programming often works best on linearly ordered objects and cannot be reordered. Brute Force algorithms are often slow and usually go beyond the $O(n)$ order of growth. With greedy algorithms, what may be a local optimal solution, may not be a global, optimal solution because not all data is considered.

Since there is no one algorithmic solution to every problem, hence the decision on which algorithm to use would be determined by the data structure and the frequency of use. There must be always a comparison of different approaches to be able to choose one which delivers the best performance in the given scenario.