Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Laboratory session #02: Exploration of Dynamic Programming

Student: Vargas Romero Erick Efrain

Professor: Rosas Trigueros Jorge Luis

Practice completion date: November 25, 2021

Report delivery date: October 1, 2021

1 Theoretical framework

From my point of view a phrase that describes in a simple way how dynamic programming works is:

Who does not know its history is condemned to repeat it

- Napoleón Bonaparte

1.1 Dynamic Programming

Dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic is the multistage nature of the optimization procedure. More so than the optimization techniques described previously, dynamic programming provides a general framework for analyzing many problem types. Within this framework a variety of optimization techniques can be employed to solve particular aspects of a more general formulation. Usually creativity is required to solve particular aspects of a more general formulation. Usually creativity is required before we can recognize that a particular problem can becast effectively as a dynamic program; and often subtle insights are necessary to restructure the formulation so that it can be solved effectively. [1]

1.1.1 Formalizing the dynamic programming approach

Dynamic programming problems have three important characteristics:

- **Stages:** The essential feature of the dynamic programming approach is the structuring of optimization problems into multiple *stages*, which are solved sequentially one stage at a time. Although each one-stage problem is solved as an ordinary optimization problem, its solution helps to define the characteristics of the next one-stage problem in sequence. The stages represent different time periods in the problems planning horizon, although sometimes there are problems that do not have implications but more difficult to recognize as a dynamic programming problem [2].

- **States:** The specification of the states of the system is perhaps the most critical design parameter of the dynamic-programming model. There are no set rules for doing this. In fact, for the most part, this is an art often requiring creativity and subtle insight about the problem being studied [2]. The essential properties that should motivate the selection states are:
  - States should convey enough information to make future decisions without regard to how process reached the current state.
  - The number of state variables should be small, since computational effor associated with the dynamic programming approach is prohibitely expensive when there are more than two, or possible three, state variables involved in the model formulation.
- **Recursive optimization:** Finally the general characteristic of the dynamic programming approach is the development of a recursive optimization procedure, which builds a solution overall *n-stage* problem by first solving a one stage problem and sequentially including one stage at a time and solving one stage problems until the overall optimum has been found. This procedure can be based on a backward induction process where the first stage to be analyzed is the final stage of the problem and problems are solved moving back one stage at a time until all stages are included [2].

  Dynamic programming algorithms are best developed in two distict stages [2]:

  1. Formulate the problem recusively: Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hardest part. A complete recursive formulation has two parts:
     (a) **Specification:** Describe the problem that you want to solve recursively, in coherent and precise English-not how to solve that problem, but what problem are you trying to solve. Whithout this specification, it is impossible, even in principle, to determine wheter your solution is correct.
     (b) **Solution:** Give a clear recursive formula or algorithm fot the whole problem in terms of the answers to smaller instances of exactly the same problem
  2. Build solutions to your recurrence from the botton up: Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:
     - Identify subproblems.
     - Choose a memoization data structure.
     - Identify dependencies.
     - Find a good evaluation order.
     - Analyze space and running time.
     - Write down the algorithm.

Fibonacci numbers

Fibonacci sequence of numbers and associated "Golden ratio" are manifested in nature and in certain works of art. We observe that many of the natural things follow the Fibonacci sequence. It appears in biological settings such as branching in trees, phyllotaxis (the arrangement of leaves on a stem), the fruit sprouts of a pineapple, the flowering of an artichoke, an uncurling fern and the arrangement of a pine cone's bracts etc. At present Fibonacci numbers plays very important role in coding theory. Fibonacci numbers in different forms are widely applied in constructing security coding [3].

Fibonacci numbers were discovered by *Leonardo Pisano*. He was known by his nickname, Fibonacci. The Fibonacci sequence is a sequence which each term is the sum of the 2 numbers preceding it. The Fibonacci numbers are defined by the recursive relation defined by the ecuation:

$$F_n = F_{n-1} + F_{n+2} \tag{1.1}$$

For all $n \geq 3$ where $F_0 = 0$ and $F_1 = 1$ Where F_n represents the $n - th$ Fibonacci number. If we write some Fibonacci numners we get something like this:

$$\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \dots\}$$

2 Material and equipment

The necessary material for this practice is:

- A computer with the latest *Python* stable version installed
- A text editor

Or is possible to use the google site `https://colab.research.google.com/` that allows us to use a virtual machine with an *Ubuntu* operative system with *Python* installed.

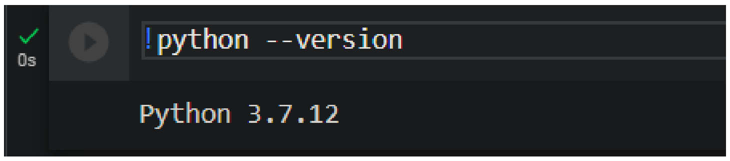
3 Practice development

3.1 Dynamic programming algorithms

To develop this practice we used the Google platform called *Colab* as this platform uses a virtual machine with linux (specifically Ubuntu) we can install some packets and of course we can verify if we have already *Python* installed. To check it we must use the command:

```
python --version
```

If we run this command in our linux terminal using *Colab* we can see the next as the result:



```
!python --version

Python 3.7.12
```

Figure 3.1: Verifying python version

Instructions: Write a script in Python to solve a problem using Dynamic Programming. One problem per team member. Include at least one example of the execution.

One clasical problem about dynamic programming is the fibonacci sequence. This is a clasical dynamic programming problem. Let's assume that you want to make some queries where the result of each of them is the $n-th$ Fibonacci number. First of all we must analize how to generate each Fibonacci number, we can easily solve this using the recurrence that we got in the theoretical framework, we can do it recursively but it will take a lot of time to process we can say that get the $n-th$ Fibonacci number will take $T(n) = T(n-1) + T(n-2)$ runtime complexity. Using dynamic programming we can reduce a lot the time complexity using some extra memory, to store the value of the $n-th$ Fibonacci number we can use an array where the $n-th$ element of the array represents the

$n-th$ Fibonacci number.

If we traslate the idea to code we can define two variables, the first one is an array that would contain the Fibonacci numbers and the other one the maximum Fibonacci number.

```
maxFibNumber = 1000 + 1
fibNumbers = [0] * maxFibNumber
```

The next step is to write in code the recurrence of the Fibonacci numbers, we can do it recursively or in an iterative way, four our case in order to avoid use more time an memory using recursive calls we use an iterative solution.

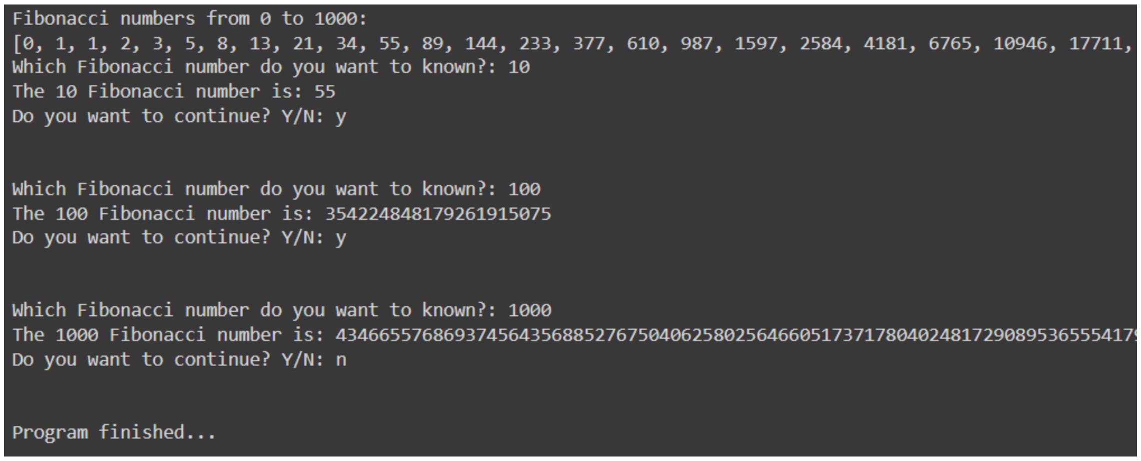
```
def calculate(fibNumbers, limit):
    for i in range(0, limit):
        #Base case when F_0 = 0 and F_1 = 1
        if i <= 1:
            fibNumbers[i] = i
        #Otherwise we known that: F_{n} = F_{n - 1} + F_{n - 2}
        else:
            fibNumbers[i] = fibNumbers[i - 1] + fibNumbers[i - 2]
```

Finally, we create a block of code to as which Fibonacci number the user wants to known its value and then if want to make another query.

```
getOther = True

while getOther:
    n = int(input("Which Fibonacci number do you want to known?: "))
    print(f"The {n} Fibonacci number is: {fibNumbers[n]}")
    ans = input("Do you want to continue? Y/N: ")
    getOther = ans == "Y" or ans == "y"
    print("\n")
```

If we ran the program and we want to discover the value of the $10th, 100th, 1000th$ Fibonacci number we see something like this:



```
Fibonacci numbers from 0 to 1000:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711,
Which Fibonacci number do you want to known?: 10
The 10 Fibonacci number is: 55
Do you want to continue? Y/N: y


Which Fibonacci number do you want to known?: 100
The 100 Fibonacci number is: 354224848179261915075
Do you want to continue? Y/N: y


Which Fibonacci number do you want to known?: 1000
The 1000 Fibonacci number is: 43466557686937456435688527675040625802564660517371780402481729089536555417
Do you want to continue? Y/N: n


Program finished...
```

Figure 3.2: Query 10th, 100th, 1000th Fibonacci number

As you can see in img. 3.2 we can make a query of different fibonacci numbers and get the answer in $O(1)$ runtime, but consider that we must precalculate from the 0 to N fibonacci numbers first, this takes $O(N)$ time and it is so much better than using the first approach that we described for this problem.

For a better understanding we add here the entire code written for this practice.

```python
def calculate(fibNumbers, limit):
    for i in range(0, limit):
        #Base case when F_0 = 0 and F_1 = 1
        if i <= 1:
            fibNumbers[i] = i
        #Otherwise we known that: F_{n} = F_{n - 1} + F_{n - 2}
        else:
            fibNumbers[i] = fibNumbers[i - 1] + fibNumbers[i - 2]


maxFibNumber = 1000 + 1
fibNumbers = [0] * maxFibNumber

calculate(fibNumbers, maxFibNumber)

print(f"Fibonacci numbers from 0 to 1000:\n{fibNumbers}")

getOther = True

while getOther:
    n = int(input("Which Fibonacci number do you want to known?: "))
    print(f"The {n} Fibonacci number is: {fibNumbers[n]}")
    ans = input("Do you want to continue? Y/N: ")
    getOther = ans == "Y" or ans == "y"
    print("\n")

print("Program finished...")
```

4 Conclusions

Dynamic programming is a very interesting technique to solve problems where we can avoid recalculate subproblems, it helps to reduce a lot the time complexity but probably we increase the space complexity of our algorithm. Additionally we can solve dynamic programming problems using two ways bottom-up or top-down in other words we can try to solve this type of problems using recursion from the entire problem and split it in subproblems or try to solve our problem from the small case to the greatest subproblem.

Bibliography

[1] *Dynamic Programming*. MIT. Accessed on November 24th, 2021. Available online: `http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf`

[2] *Dynamic Programming*. December 29th, 2018, 0th Edition (pre-publication draft). Accessed on November 24th, 2021. Available online:`http://jeffe.cs.illinois.edu/teaching/algorithms/book/03-dynprog.pdf`

[3] S. Sinha, *The Fibonacci numbers and its Amazing Applications*. Accessed on November 24th, 2021. Available online: `https://www.researchgate.net/publication/330740074_The_Fibonacci_Numbers_and_Its_Amazing_Applications/link/5d87ae48299bf1996f935716/download`