UiT The Arctic University of Norway

DTE-2501 AI Methods and Applications

Basic Introduction to AI

Lecture 3/3: Dynamic Programming

Ghada Bouzidi

PhD Candidate

Department of Computer Science and Computational Engineering

Office: C4060

Email: ghada.bouzidi@uit.no

Main Algorithmic Techniques

- Brute Force
- Divide & Conquer (sort, search …)
- Greedy (fractional knapsack …)
- Dynamic Programming

Greedy: Fractional Knapsack

	Value (Nok)	Weight (Kg)	Unit Value (Nok/Kg)
2	250	5	50.0
4	50	25	2.0
3	500	20	25.0
1	300	5	60.0

Problem: Find the maximum value to fit in the knapsack without exceeding the capacity. It is allowed to take any fraction of any item.

Knapsack capacity
25 Kg

Solution pseudocode:

```
1. Sort items by unit value
2. While the picked items did not exceed the knapsack capacity
   pick an item from the remaining items that has the highest unit
   value (Greedy choice)
```

Dynamic Programming

- Widely used in different fields (Biology, Planning, Logistics …)
- Solves problems that cannot be solved using Greedy and Divide & Conquer approaches
- Some Brute Force algorithms may spend years and may not even finish in our lifetime, if a Dynamic Programming solution exists then the runtime will decrease considerably.

History of Dynamic Programming

- Developed by Richard Bellman in the 1950s

- “…An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research"...The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics...What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic...Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning.It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to...”

  Richard Bellman, Eye of the Hurricane: An Autobiography (1984, page 159)

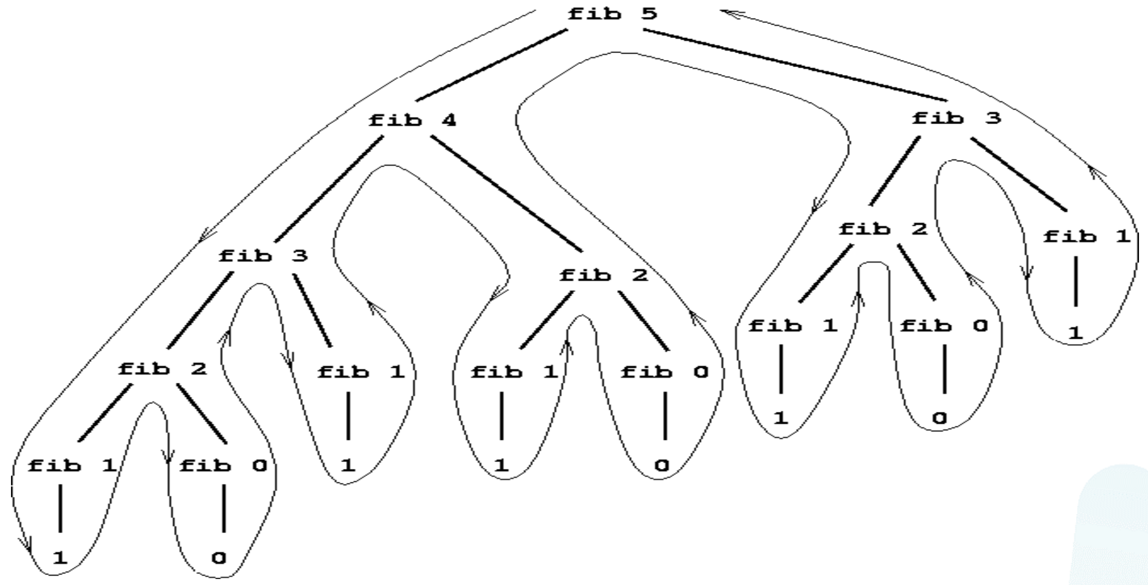
Example 1: Fibonacci Sequence

- Fib(0) = 0
- Fib(1) = 1
- Fib(n) = Fib(n-1) + Fib(n-2) for all n >= 2

The Fibonacci sequence : 0, 1, 1, 2, 3, 5, 8, 13 ...

Recursion

```
def Fib(n):
    if n <= 1:
        return n
    else:
        return Fib(n-1) + Fib(n-2)
```



Recursion tree of Fib(5)

Tree illustration source: "Structure and Interpretation of Computer Programs" (https://mitpress.mit.edu/sites/default/files/sicp/full-text/sicp/book/book.html)

Recursion + cache (Memoization)

- How many times one needs to compute Fib(2)?
- One time is sufficient, then store its value "somewhere"
- Whenever its value is needed, one can look it up

“Those who cannot remember the past are condemned to repeat it.”
Philosopher George Santayana, The Life of Reason (1905)

Memoized Fibonacci (Top-Down Approach)

```
def fib(n, memo):
    if n <= 1:
        return n
    elif memo[n] == -1:
        memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

Iterative Fibonacci: Tabular DP(Bottom-Up)

```
def fib(n):
    dp = [0] * (n+1)
    dp[1] = 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

dp:

0	1	1	2	3	5	8	13	...
0	1	2	3	4	5	6	7	...

NB: This solution can be optimized for space since you only need the last two values at each step

When to Use Dynamic Programming

- The problem cannot be solved through Greedy approach
- The problem can be divided into smaller subproblems (i.e., with smaller inputs)
- There exists a recurrent relation that defines how the subproblems can be combined to solve the problem
- The subproblems intersect

Memoization vs Tabular DP

- Sometimes it is more intuitive and easier to implement a dynamic programming algorithm recursively once the recurrent relation is identified than writing a tabular DP
- It is important to keep in mind that you may run into stackoverflow problems when the recursion depth exceeds the limited size of the function call stack
- In those cases, an iterative approach such as the Tabular DP might be the solution
- The runtime analysis of a Tabular DP can be more easily calculated than that of Memoization

Example 2: Coin Change

- Problem: Given a sum *s* and a set of coins with denominations {1, 4, 5}, write the function CoinChange that takes as input *s* and returns the minimum number of coins that sum up to *s*
- *Can Greedy solve this problem? No*
- *If s = 8 then we can first pick the coin 5 (Greedy choice, biggest coin <= s)*
  then we can pick the coin 1 three times to get 8 = 5 + 1 + 1 + 1
  output = 4 coins
  We know that we can have less coins that make up the sum 8 through picking the coin 4 two times (8 = 4 + 4). Thus, the output should be 2 coins
- *For s = 8, the minimum number of coins is 2 (4, 4)*

Recurrent Relation for Coin Change

- If we know the minimum coins to change *s-1 and s-4 and s-5, can we know the minimum coins to change s ?*
- If *s* = 8 and we know how to change 7 and 4 and 3, only adding the coins of values 1, 4 and 5 respectively will give us the sum 8. However, we need to make the choice such that the result is optimal.
- Let CoinChange(i) be the minimum coins needed to change a sum *i then:*

CoinChange(i) = 1 + min {CoinChange(i-1), CoinChange(i-4), CoinChange(i-5)}

Memoized Coin Change

```
def CoinChange(n, memo):
    if n <= 0:
        return 0
    elif n not in memo:
        memo[n] = min(CoinChange(n-1, memo),
                      CoinChange(n-4, memo),
                      CoinChange(n-5, memo)) + 1
    return memo[n]
```

Tabular Coin Change 1/2

For each index i in the table "coin_change", coin_change[i] stores the minimum number of coins to change the sum i.

0	1	2	3	1	1	2	3	2
0	1	2	3	4	5	6	7	8

How to fill this table for s = 8? coin_change[i] = min(coin_change[i-1], coin_change[i-4], coin_change[i-5]) + 1

coin_change[0] = 0

coin_change[1] = min(coin_change[1-1], ~~coin_change[1-4]~~, ~~coin_change[1-5]~~) + 1 = min(0, Inf, Inf) + 1 = 1

coin_change[2] = min(coin_change[2-1], ~~coin_change[2-4]~~, ~~coin_change[2-5]~~) + 1 = min(1, Inf, Inf) + 1 = 2

coin_change[3] = min(coin_change[3-1], ~~coin_change[3-4]~~, ~~coin_change[3-5]~~) + 1 = min(2, Inf, Inf) + 1 = 3

coin_change[4] = min(coin_change[4-1], coin_change[4-4], ~~coin_change[4-5]~~) + 1 = min(3, 0, Inf) + 1 = 1

coin_change[5] = min(coin_change[5-1], coin_change[5-4], coin_change[5-5]) + 1 = min(1, 1, 0) + 1 = 1

coin_change[6] = min(coin_change[6-1], coin_change[6-4], coin_change[6-5]) + 1 = min(1, 2, 1) + 1 = 2

coin_change[7] = min(coin_change[7-1], coin_change[7-4], coin_change[7-5]) + 1 = min(2, 3, 2) + 1 = 3

coin_change[8] = min(coin_change[8-1], coin_change[8-4], coin_change[8-5]) + 1 = min(3, 1, 3) + 1 = 2

Tabular Coin Change 2/2

- How to recover which 2 coins sum to s = 8 from the dynamic programming table?

0	1	2	3	1	1	2	3	2
0	1	2	3	4	5	6	7	8

- Recall: **coin_change[8] = min(coin_change[8-1], coin_change[8-4], coin_change[8-5]) + 1**
- Work backward from index 8 to index 0 to recover the 2 coins that sum up to 8

0	2	0
1	4	5