Dynamic Programming - Fibonacci, Rod Cutting

CS313E - Elements of Software Design

Kia Teymourian

11/01/2023

Agenda

1. Dynamic Programming
2. Fibonacci Sequence
3. Rod Cutting Problem

Dynamic Programming

- Dynamic programing is invented by Richard E. Bellman (1920-1984).
- "Bellman, (1984) p. 159 explained that he invented the name "dynamic programming" to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who "had a pathological fear and hatred of the term, research." [1]

The name "Dynamic Programming" means simply optimization by using careful brute-force search of solutions.

[1] Dynamic Programming, entry for consideration by the New Palgrave Dictionary of Economics John Rust, University of Maryland April 5, 2006
https://editorialexpress.com/jrust/research/papers/dp.pdf

Dynamic Programming

- The name **"Dynamic Programming"** means simply solving an optimization problem in a very special form. It has nothing to do with "computer programming".
- If you have a problem that have multiple cases and one of them is the solution case (the best optimal case), for example In Single-Source shortest path, we can maybe brute force and find the solution.
- Dynamic programing can be summarized as careful brute force algorithm of a recursive algorithm that can reuse sub-trees of the recursion tree (memorize it - named Memoization) so that it basically does not need to recompute it in each sub-tree. We will learn these techniques by using a couple of examples.

DP

Dynamic Programming can be seen as a careful brute force algorithm with memorization (named Memoization).

Example Case - Fibonacci Sequence

- Fibonacci numbers or aka Fibonacci sequence is a series of numbers such that each number is the sum of the two preceding ones, starting from 0 and 1.
- The beginning of the Fibonacci sequence is thus: $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots \rangle$.

fibonacci(n)

We want to develop an algorithm that can generate the n_{th} element of the Fibonacci sequence. The following code is a python implementation of such algorithm. For example, fibonacci(6) = 8, and fibonacci(7) = 13 .

fibonacci(n) - Top-Down

```
def fibonacci(n) :
if (n <=2 ) :
    # First Fibonacci number is 0
    return 1
else:
    return fibonacci(n-1) + fibonacci(n-2)
```

We can analyze the above recursive implementation and write the running time as a recurrence.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + O(1) \\ &\geq 2T(n-2) + O(1) \\ &\geq 2^{n/2} \end{aligned}$$

The running time of the above algorithm is exponential $O(2^n)$ which is bad.

Top-Down Solution

This solution is called a Top-Down Solution.

fibonacci(n) - Top-Down

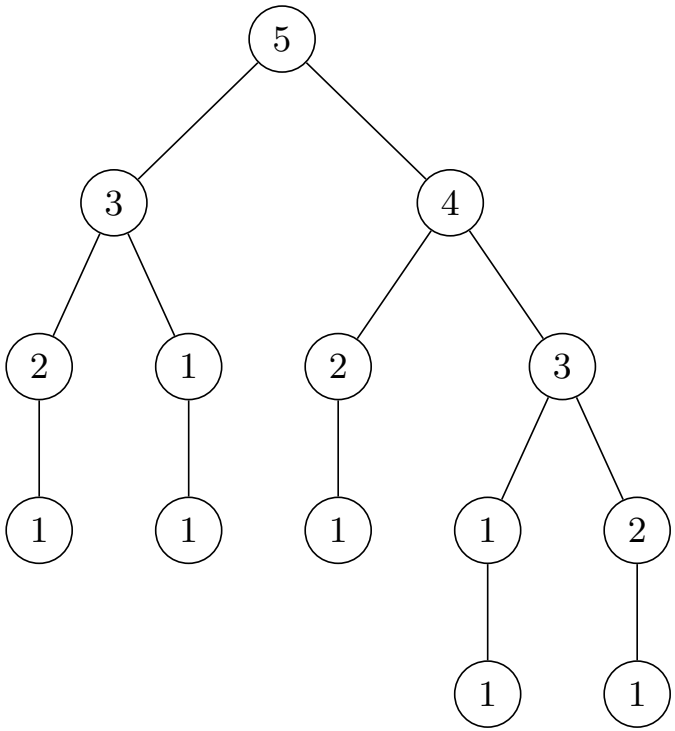


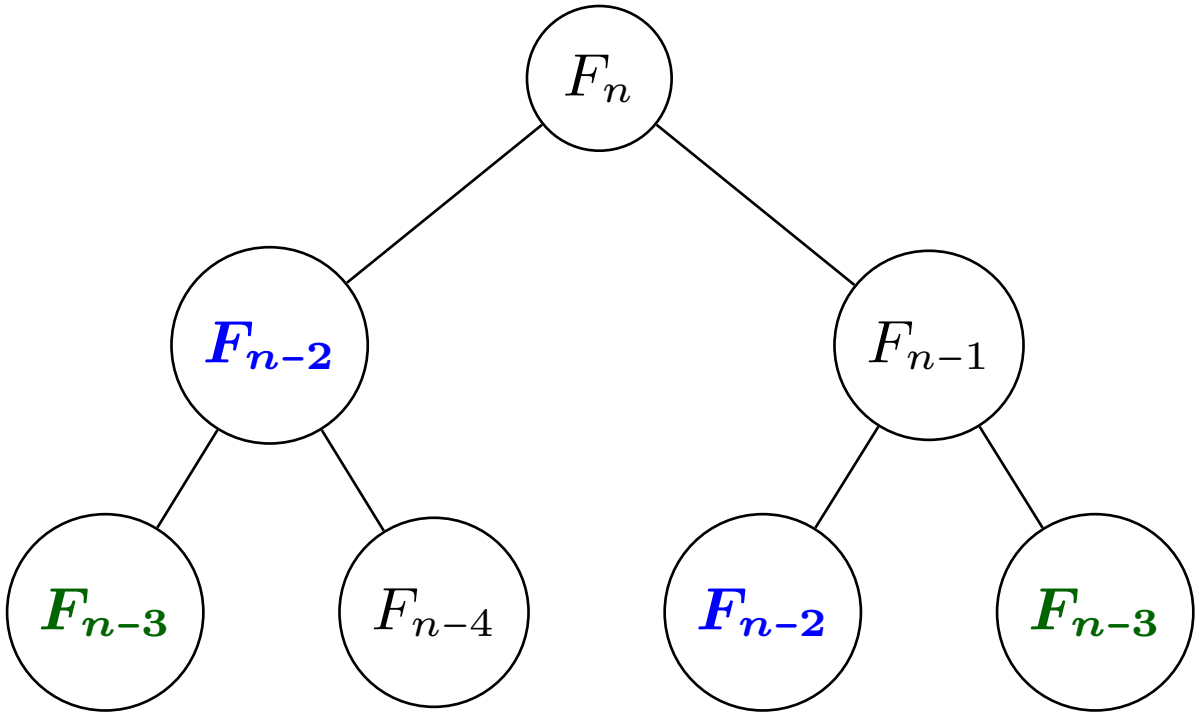
Figure: Fibonacci Recursive Tree for Computing 5.

fibonacci(5) = 5 , count up the leaves here.

Top-Down Solution

This solution is called a Top-Down Solution because we open the problem from top to bottom.

Memoization of Fibonacci



Memoization

One of the main techniques that "Dynamic Programing" is reuse of the recursive tree. Memorize where we have visited and reuse it.

An implementation of Fibonacci Sequence with Memoization

```
# Fibonacci Series using Dynamic Programming
#  Memoized it in a dictionary or list
mem={}

def fibonacciMem(n):
    if n <= 1:
        return n
    else:
        if (n-1) not in mem:
            mem[n - 1] = fibonacciMem(n - 1)
        if (n-2) not in mem:
            mem[n - 2] = fibonacciMem(n - 2)
    mem[n] = mem[n - 2] + mem[n - 1]
    return mem[n]
```

- We use a dictionary in python to record previous results, and reuse it rather than recompute it each time.

What is the running time of this recursive algorithm?

```
# Fibonacci Series using Dynamic Programming
mem={}

def fibonacciMem(n):
    #  Memoized it

    if n <= 1:
        return n
    else:
        if (n-1) not in mem:
            mem[n - 1] = fibonacciMem(n - 1)
        if (n-2) not in mem:
            mem[n - 2] = fibonacciMem(n - 2)
    mem[n] = mem[n - 2] + mem[n - 1]
    return mem[n]
```

- If F_i is computed, it is stored in the array and never recomputed, so the algorithm basically traces a path from root to rightmost leaf of the tree, adding up all the results at each level in one addition.
- Thus the algorithm runs in time **$\Theta(n)$. This is much better than exponential.**

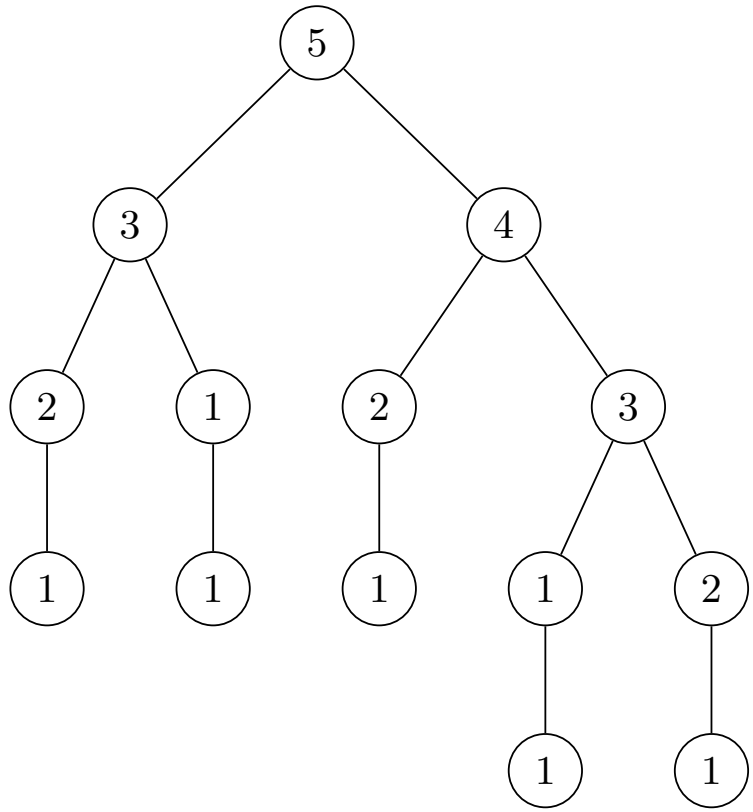
Memoization

This memoization technique can be applied to many recursive algorithms similarly to speed up your programs.

Fibonacci Series using Dynamic Programming - Bottom Up Solution.

- One other important dynamic programing technique is to open up the recursive tree (which is actually a DAG (Directed Acyclic Graph)) from leaves to the root.
- This technique is named **"Botton Up"** because we look at the recursive tree from the leaves to the root and modify the algorithm to open up the solution set from solution result to the instantiation recursive root.

```
# Fibonacci Series using Dynamic
    Programming
# Bottom Up Solution
def fibonacciBUS(n):
a = 0
b = 1
if n <= 1:
    return n
else:
    for i in range(2,n+1):
        c = a + b
        a = b
        b = c
    return b
```



Bottom Up Solution

```
# Fibonacci Series using Dynamic Programming
# Bottom Up Solution
def fibonacciBUS(n):
a = 0
b = 1
if n <= 1:
    return n
else:
    for i in range(2,n+1):
        c = a + b
        a = b
        b = c
    return b
```

- The bottom up solution has exactly the same computation running time as the memoized solution of dynamic programing.
- This is just unrolling the recursive tree.
- In practical it is much faster than the recursive implementation.
- There is no memoization and no storage, also there is no rolling of recursive tree on memory stack. Both are $O(n)$ memory usage. We can save memory space.
- Analysis of such bottom up solution is more clear and simpler.

Bottom Up Solution - Topological Sort

We can think that the bottom-up solution is a kind of **"Topological Sort"** (One of the application of Depth-First Algorithm) approach of a recursive tree DAG structure.

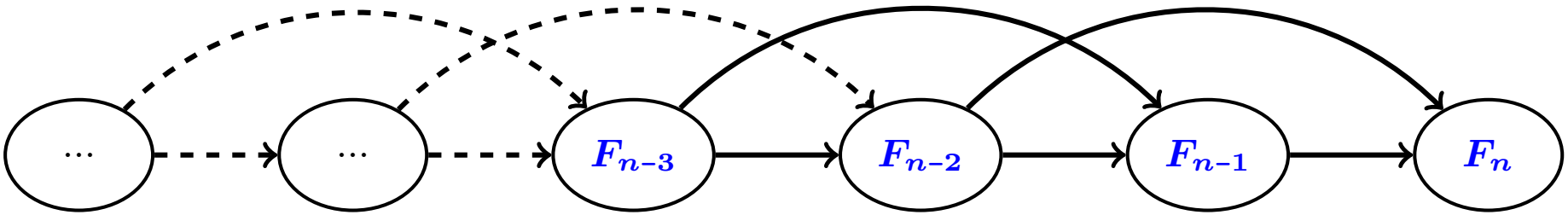


Figure: Fibonacci Recursive Tree as a Topological Sort of a DAG.

Rod Cutting Problem

The rod-cutting problem can be defined as follows:

- Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$
- Find the the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
- If the price p_n for a rod of length n is large enough, no cut is needed for the optimal solution.

Table: An example of Rod Cutting Problem Lengths and Prices

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Rod Cutting - Example

Table: An example of Rod Cutting Problem Lengths and Prices

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Example

Cut a rod with size $n = 4$
8 possible ways of cuts:

1. 0 cut, Revenue $r = 9$
2. 1 cut (1, 3), Revenue $r = 1 + 8 = 9$
3. 1 cut (2, 2), Revenue $r = 5 + 5 = 10$
4. 1 cut (3, 1), Revenue $r = 1 + 8 = 9$
5. 2 cuts (1, 1, 2), Revenue $r = 1 + 1 + 5 = 7$
6. 2 cuts (1, 2, 1), Revenue $r = 1 + 5 + 1 = 7$
7. 2 cuts (2, 1, 1), Revenue $r = 5 + 1 + 1 = 7$
8. 3 cuts (1, 1, 1, 1),Revenue $r = 1 + 1 + 1 + 1 = 4$

The Optimal Revenues from shorter rods is 1 cut (2, 2), $r = 5 + 5 = 10$

Recursion Tree for the above example of Rod Cutting

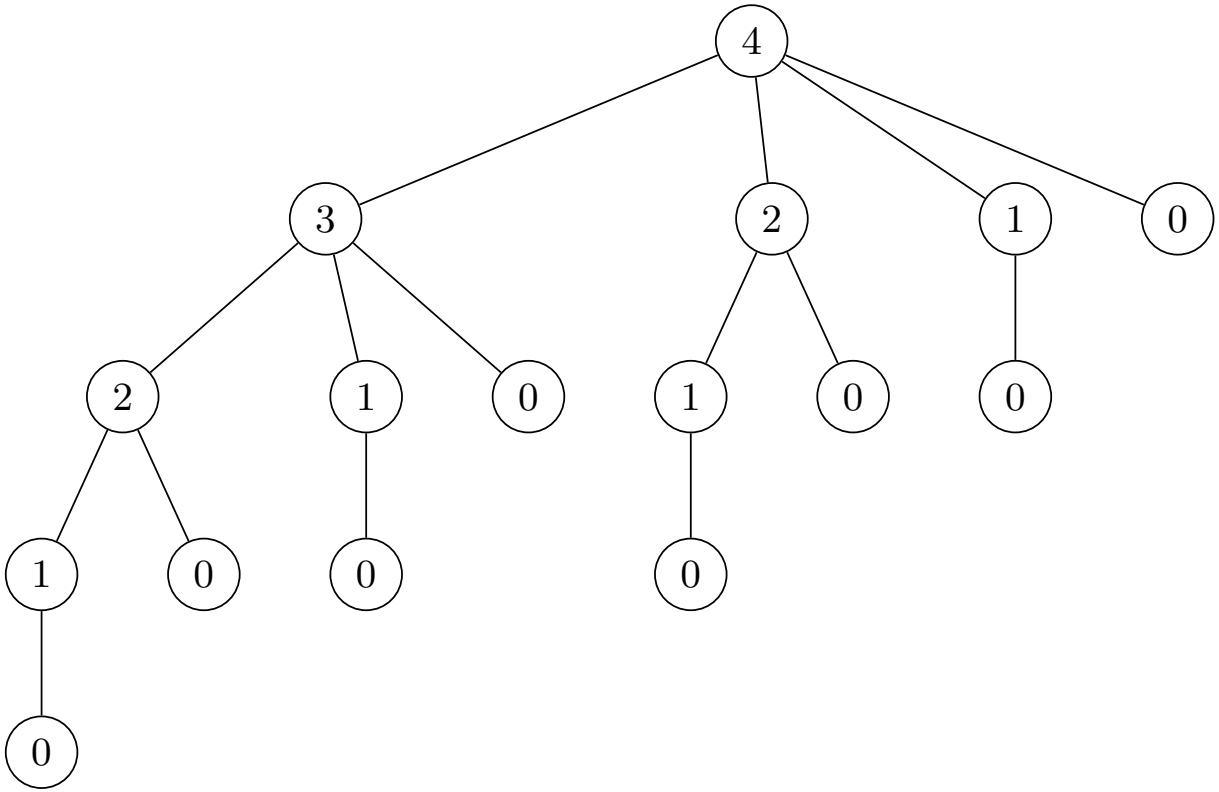


Figure: Recursion Tree for the $n = 4$ example

- A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n.
- This recursion tree has 2^n nodes and 2^{n-1} leaves.

Rod Cutting - Top-Down

Algorithm CUT_ROD(p, n)

```
1: if n == 0 then
2:     return 0
3: end if
4: let q = −∞
5: for i = 1 to n do
6:     q = max(q, p[i] + CUT_ROD(p, n-i))
7: end for
8: return q
```

- Input input an array $p[1 \ldots n]$ of prices and an integer n
- Returns the maximum revenue possible for a rod of length n.
- If $n = 0$, no revenue is possible
- We loop through 1 to n, and find max recursively.

Rod Cutting - Top-Down

Algorithm CUT_ROD(p, n)

```
1: if n == 0 then
2:     return 0
3: end if
4: let q = −∞
5: for i = 1 to n do
6:     q = max(q, p[i] + CUT_ROD(p, n-i))
7: end for
8: return q
```

$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$
$T(n) = \Theta(2^n)$
The running time of the CUT-ROD is exponential in n.

Memoized Solution of Rod Cutting Problem

Algorithm MEMOIZED-CUT-ROD(p, n)

1: let $r[0, \ldots, n]$ be new array
2: **for** $i = 1$ to n **do**
3: $\quad r[i] = -\infty$
4: **end for**
5: **return** MEMOIZED-CUT-ROD-AUX(p, n, r)

Memoized Solution of Rod Cutting Problem

Algorithm MEMOIZED-CUT-ROD-AUX(p, n, r)

```
1:  if r[n] ≥ 0 then                                   ▷ If it is already known return it
2:      return r[n]
3:  end if
4:  if n == 0 then                                     ▷ If it is base.
5:      q = 0
6:  else q == −∞                                       ▷ If we have not visited that subgraph before.
7:      for i = 1 to n do
8:          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n-i, r))
9:      end for
10: end if
11: r[n] = q                                           ▷ We keep track of it and memoize it
12: return q
```

BOTTOM-UP-CUT-ROD(p, n)

Algorithm BOTTOM-UP-CUT-ROD(p, n)

```
1:  let r[1...n] be a new array
2:  r[0] = 0                                          ▷ Base case
3:  for j = 1 to n do
4:      q = −∞
5:      for i = 1 to j do                             ▷ Open it up from botton to the up
6:          q = max(q, p[i] + r[j − 1])
7:      end for
8:      r[j] = q
9:  end for
10: return r[n]
```

Apply Dynamic Programming

Dynamic programing in 5 steps:

1. **Define your Sub-problems.** After the initial choice we would then have one or more sub-problems to work on to solve it. (Think of count of possible sub-problems)
2. **Guessing is part of the solution.** For a given problem, you are given a set of the choices that leads to an optimal solution.
   You do not concern yourself yet with how to determine this choice. You can just guess it! (Think of number of possible guesses that you have)
   Characterize the structure of an optimal solution.
3. **Related the given sub-problems to the optimal solution.**
   Recursively define the value of an optimal solution. (Based on number of sub-problems)
4. **Compute the value of an optimal solution by using recursion and memoization.**
   Alternatively, you can build dynamic programming bottom up approach.
   Check subproblems using topological order.
5. **Solve the original problem and construct an optimal solution by combining sub-problem solutions.**

Next Lecture - More dynamic programing examples

We learn about more examples of Dynamic Programing:

- 1-0 Knapsack Problem
- Matrix-chain Multiplication, aka Parenthesizations $(A_1(A_2A_3))$ or $((A_1A_2)A_3)$

Readings from CLRS Book (Introduction to Algorithms, 3rd Edition)

- Section 15 Dynamic Programming
- Section 15.1 Rod Cutting
- Section 15.2 Matrix-chain multiplication
- Section 15.3 Elements of dynamic programming
- Section 15.5 Optimal binary search trees