Dynamic Programming Ideas & Examples

Dr. Samuel Cho, PhD

NKU

1. Dynamic Programming: what, why, and how
2. Simple Example of Dynamic Programming

Dynamic Programming What, Why, and How

Dynamic Programming: what

- When an algorithm has a name programming in it, it is most likely an optimization algorithm.
- For example, integer programming is an optimization algorithm to find the mathematical solution with integer variables constraints.
- Dynamic in 'dynamic programming' means that we can find the optimal solution dynamically from the sub-solutions.

- Let's start with a simple example.
- Fibonacci sequence is defined mathematically as follows: $F(n) = F(n-1) + F(n-2)$ if $n > 2$ where $F(0) = 0, F(1) = 1$.
- It means n^{th} number is the sum of the previous two numbers.
- Thus, the sequence goes like this: *0, 1, 1, 2, 3, 8, 13* ...

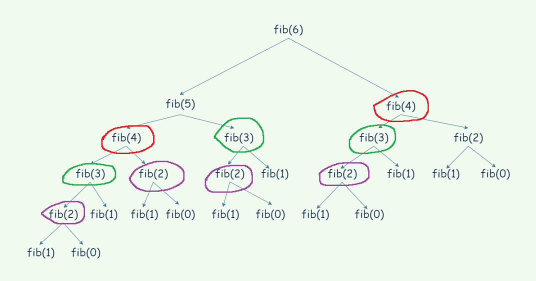
Solution with a Brutal Force Method

- We can make Python code to get n^{th} Fibonacci number easily using recursion as follows:

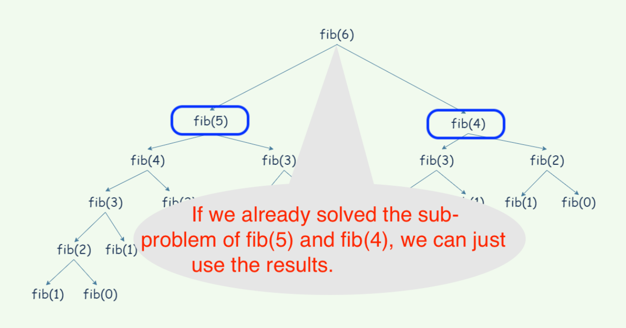
Code #

```
def fib(n):
  if n < 0: return -1 # Input error
  if n == 0: return 0
  if n == 1: return 1
  return fib(n-1) + fib(n-2)
```

- However, we soon realize that this approach using recursion is not a good solution, as it takes so much time to get the result when *n* is a large number.
- It is because this algorithm should recompute multiple times to get the results that are already computed.
- This diagram shows the duplications to get the result of fib(6).



- In this example, if we already have the solution of the subproblem, in this case, fib(5) and fib(4), we can easily get the result.



Better solution with DP 1

- To apply this idea, we can iteratively get all the sub-solutions in a list.
- To get the solution of f(6), we can just retrieve solutions from the list and add them.

Code #

```
def fib(n):
    subsolutions = [0] * n  # make a list of size n

    subsolutions[0] = 0
    subsolutions[1] = 1
    for i in range(2, n):  # get subsolutions from 2 to (n-1)
        subsolutions[i] = subsolutions[i-1] + subsolutions[i-2]
    return subsolutions[n-1] + subsolutions[n-2]
```

Better solution with DP 2

- We also can get the solution recursively by caching (memoizing) the sub-solutions in a dictionary.

Code #

```
def fib(n):
    if n in subsolutions:  # we already have the result
        return subsolutions[n]
    else:
        result = fib(n-1) + fib(n-2)
        subsolutions[n] = result # memoization
        return subsolutions[n]
```

Why DP?

- $T(n) = O(2^n)$
- $T(n) = O(n)$

	bf	dp1	dp2
n = 15	0.029	0.031	0.030
n = 35	3.438	0.028	0.028
n = 100	N/A	0.030	0.029

- This is theoretical time to compute the Fibonacci numbers.
- We can see that we can get results only with DP when n becomes large.
- It is no wonder that we couldn't get the results when n is 100.

N	TIME (MSEC)	
	RECURSIVE	DP
10	0	0
20	1	0
30	8	0
40	922	0
50	113770	0

Decorator Pattern in Python

- Decorator pattern is one of the design patterns to enhance the functionality of an original method without modifying it.
- Python uses @ to use the decorator method.
- Python has a decorator method (@functools.lru_cache(None)) for memoizing.

Code #

```python
import functools

@functools.lru_cache(None)
def fib(n):
    if n == 0: return 0
    elif n == 1: return 1
    else:
        return fib(n-1) + fib(n-2)
```

Simple Example of Dynamic Programming

Question

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed; the only constraint stopping you from robbing each of them is that **adjacent houses have security systems connected, and it will automatically contact the police if two adjacent houses were broken into** on the same night.

Given an integer array nums representing the amount of money of each house, **return the maximum amount of money** you can rob tonight without alerting the police.

Input/Output 1

- Input: nums = [1,2,3,1]
- Output: 4
- Constraints: 1 <= nums.length <= 100 and 0 <= nums[i] <= 400
- Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
- Total amount you can rob = 1 + 3 = 4.

With brutal force algorithm, we get (1 + 3), (1 + 1), and (2 + 1). We can check that 1 + 3 is the maximum value.

Input/Output 2

- Input: nums = [2,7,9,3,1]
- Output: 12
- Constraints: same as before
- Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
- Total amount you can rob = $2 + 9 + 1 = 12$.

With brutal force algorithm, we get $(2 + 9 + 1)$, $(2 + 3)$, $(2 + 1)$, and $(7 + 3)$. We can check that $(2 + 9 + 1)$ is the maximum value.

How to solve DP Problems

Step 1: Define what are variables and functions.

Step 2: Find what is the recursion formula.

Step 3: Find what are the initial values.

Step 4: Select top down (recursion) or bottom up (table).

Step 1: Let's say f(n) is the Largest amount that you can rob from first house to the n^{th} indexed house.

Step 1: Let's say A_n is the amount of maximum money at the n^{th} index house.

Step 2: At n^{th} house, we can select the last (previous) house f(n) or current house (A_n) and f(n-2). So the recursion formula is $max(A_n + f(n-2), f(n-1))$.

Step 3: $f(0) = A_0$ and $f(1) = max(A_0, A_1)$.

Step 4: We can use both approaches.

Bottom up (Iteration)

Code #

```python
1 def f(nums):
2     size = len(nums)
3     if size == 0: return 0
4     if size == 1: return nums[0]
5 
6     A = [0] * size
7     for (index, value) in enumerate(nums):
8         if index == 0: A[0] = value
9         elif index == 1: A[1] = max(A[0], value)
10        else: A[index] = max(value + A[index - 2], A[index - 1])
11    return A[-1]
```

- This implementation computes the A_n iteratively.
- Notice that we should check the special cases when the input size is 0 or 1 (lines 3–4).

Top down (Recursion)

Code #

```
def f(nums):
    size = len(nums)
    if size == 0: return 0
    if size == 1: return nums[0]
    return f0(nums, len(nums)-1)

cache = {}
def f0(nums, n):
    if n in cache: return cache[n]
    else:
        if n == 0: result = nums[0]
        elif n == 1: result = max(nums[0], nums[1])
        else: result = max(nums[n] + f0(nums, n - 2), f0(nums, n
            ↪ - 1))
    cache[n] = result
    return result
```

- This is one of the easy DP problems, as it is relatively easy to find the relationship between f(n) and f(n-1).
- Also, we see a pattern max(A + f(n-1), f(n)) which we will see over and over again in DP problems.
- Hard DP problems are nothing more than the problems that are hard to identify the pattern.

- For some questions, it is easier to understand and solve DP problems using the bottom-up method with a table and iteration.
- But for other questions, it is easier to understand and solve DP problems using the top-down method with a cache and recursion.
- So, it is important to solve as many questions as possible to understand the relationship between f(n) and f(n-1).

- You can find more DP questions in the dp_questions.pdf file. When you send your answers to me using an email, I will send you the answers with some explanations.