Ch25-DynamicProgramming

October 30, 2020

1 Dynamic Programming (DP)

- https://www.cs.cmu.edu/~avrim/451f09/lectures/lect1001.pdf
- https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/
- powerful technique that allows one to solve many different types of problemns in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time
- two main properties of a problem that warrents DP solution:
  1. Overlapping Subproblems
  2. Optimal Substructures

1.1 Overlapping Subproblems

- problem combines solutions from many overlapping sub-problems
- DP is not useful when there are no common (overlapping) subproblems
- computed solutions to sub-problems are stored in a look-up table to avoid recomputation
- slighlty different from **Divide and Conquer** technqiue
  - divide the problems into smaller non-overlapping subproblems and solve them independently
  - e.g.: merge sort and quick sort

1.2 Optimal Substructures

- optimal solution of the given problem can be obtained by using optimal solutions of its subproblems

1.3 2 Types of DP solutions

1.4 1. Top-Down (Memoization)

- based on the Latin word memorandum, meaning “to be remembered”
- similar to word memorization, its a technique used in coding to improve program runtime by memorizing intermediate solutions
- using dict type lookup data structure, one can memorize intermediate results of subproblems
- tpically recursion use top-down approach

1.4.1 Process

- start solving the given problem by breaking it down
- first check to see if the given problem has been solved already

- if so, return the saved answer
- if not, solve it and save the answer

1.5 2. Bottom-Up (Tabulation)

- start solving from the trivial subproblem
- store the results in a table/list/array
- move up towards the given problem by using the results of subproblems
- typically iterative solutions uses bottom-up approach

1.5.1 simple recursive fib function

- recall, fibonacci definition is recursive and has many common/overlapping subproblems

[1]:
```
count = 0
def fib(n):
    global count
    count += 1
    if n <= 1:
        return 1
    f = fib(n-1) + fib(n-2)
    return f

n=30 #40, 50? takes a while
print("fib at {}th position = {}".format(n, fib(n)))
print("fib function count = {}".format(count))
```

```
fib at 30th position = 1346269
fib function count = 2692537
```

1.5.2 theoritical computational complexity

- Time Complexity: $T(n)$ = time to calculate Fib(n-1) + Fib(n-2) + time to add them: O(1)
- using Big-Oh (O) notation for upper-bound:
  - $T(n) = T(n-1) + T(n-2) + O(1)$
  - $T(n) = O(2^{n-1}) + O(2^{n-2}) + O(1)$
  - $T(n) = O(2^n)$

  precisely
  - $T(n) = O(1.6)^n$
    - 1.6… is called golden ratio - https://www.mathsisfun.com/numbers/golden-ratio.html
- Space Complexity = $O(n)$ due to call stack

[8]:
```
#print(globals())
import timeit
print(timeit.timeit('fib(30)', number=1, globals=globals()))
# big difference between 30 and 40
```

```
0.35596761399983734
```

1.5.3 memoized recursive fib function

[9]:
```
count = 0
def MemoizedFib(memo, n):
    global count
    count += 1
    if n <= 1:
        return 1
    if n in memo:
        return memo[n]
    memo[n] = MemoizedFib(memo, n-1) + MemoizedFib(memo, n-2)
    return memo[n]
```

[11]:
```
memo = {}
n=1000 #try 40, 50, 60, 100, 500, 10000, ...
print("fib at {}th position = {}".format(n, MemoizedFib(memo, n)))
print("fib function called {} times.".format(count))
```

```
fib at 1000th position = 703303677114228158218352548771835497701812698363587327426049050871545371181969335797422494945626117334877504492417659910881863632654502236471060120533741212738673391111981393731255987676900919022452453234035 01
fib function called 2118 times.
```

[21]:
```
import timeit
memo = {}
n=1000
print(timeit.timeit('MemoizedFib(memo, n)', number=1, globals=globals()))
```

```
0.0009976609999284847
```

1.6 using function decorator @cache

- no need to write our own caching mechanism

[1]:
```
# cache is new in Python 3.9
from functools import cache

count = 0
@cache
def cachedFib(n):
    global count
    count += 1
    if n <= 1:
        return 1
    f = fib(n-1) + fib(n-2)
    return f
```

```
---------------------------------------------------------------------------
```

```
ImportError                               Traceback (most recent call last)
<ipython-input-1-c5a0643615f6> in <module>
----> 1 from functools import cache
      2
      3 count = 0
      4 @cache
      5 def fib(n):

ImportError: cannot import name 'cache' from 'functools' (/Users/rbasnet/
↪miniconda3/lib/python3.8/functools.py)
```

[ ]:
```
import timeit
memo = {}
n=1000
print(timeit.timeit('CachedFib(n)', number=1, globals=globals()))
```

[1]:
```
%%bash
conda update python
```

```
Collecting package metadata (current_repodata.json): …working… done
Solving environment: …working… done

# All requested packages already installed.
```

[2]:
```
! python --version
```

```
Python 3.8.5
```

1.6.1 computational complexity of memoized fib

- Time Complexity - $O(n)$
- Space Complexity - $O(n)$

1.6.2 normally large integer answers are reported in mod

- mod of a farily large prime number e.g. $(10^9 + 7)$
- need to know some modular arithmetic: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-addition-and-subtraction
- $(A + B)\%C = (A\%C + B\%C)\%C$
- $(A - B)\%C = (A\%C - B\%C)\%C$

[15]:
```
mod = 1000000007
def MemoizedModFib(memo, n):
    if n <= 1:
        return 1
    if n in memo:
        return memo[n]
```

```
    memo[n] = (MemoizedFib(memo, n-1)%mod + MemoizedFib(memo, n-2)%mod)%mod
    return memo[n]
```

[17]:
```
memo = {}
n=1000 #try 40, 50, 60, 100, 500, 10000, ...
print("fib at {}th position = {}".format(n, MemoizedModFib(memo, n)))
```

```
fib at 1000th position = 107579939
```

1.6.3 bottom-up (iterative) fibonacci solution

- first calculate fib(0) then fib(1), then fib(2), fib(3), and so on

[22]:
```
def iterativeFib(n):
    # fib array/list
    fib = [1]*(n+1) # initialize 0..n list with 1
    for i in range(2, n+1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib[i]
```

[24]:
```
n=1000
print(timeit.timeit('iterativeFib(n)', number=1, globals=globals()))
# is faster than recursive counterpart
```

```
0.0001866370002971962
```

1.7 Coin Change Problem

- https://www.geeksforgeeks.org/understanding-the-coin-change-problem-with-dynamic-programming/
- essential to understanding the paradigm of DP
- a variation of problem definition:
  - Given an infinite number of coins of various denominations such as 1 cent (penny), 5 cents (nickel), and 10 cents (dime), can you determine the total number of combinations (order doesn't matter) of the coins in the given list to make up some amount N?
- Example 1:
  - Input: coins $= [1, 5, 10]$, $N = 8$
  - Output: 2
  - Combinations:
    1. $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8$
       * $1 + 1 + 1 + 5 = 8 $

- Example 2:
  - Input: coins $= [1, 5, 10], N = 10$
  - Output: 4
  - Combinations:
    1. $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 10$
       * $ 1+1+1+1+1+5 = 10$

        * \$ 5+5 = 10\$
        * 10 = 10
- Implementation:
    - we use tabulation/list/array to store the number of ways for outcome $N = 0$ to 12
    - values of list represent the number of ways; indices represent the outcome/sum N
    - so ways = [0, 0, 0, 0, 0, 0....] initialized with 12 0s
    - base case:
        * ways[0] = 1; there's 1 way to make sum N=0 using 0 coin
    - for each coin:
        * if the value of coin is less than the outcome/index N,
            · update the ways[n] = ways[n-coin] + ways[n]

[3]:
```python
def countWays(coins, N):
    # use ways table to store the results
    # ways[i] will store the number of solutions for value i
    ways = [0]*(N+1) # initialize all values 0-12 as 0
    # base case
    ways[0] = 1
    # pick all coins one by one
    # update the ways starting from the value of the picked coin
    print('values:', list(range(N+1)))
    for coin in coins:
        for i in range(coin, N+1):
            ways[i] += ways[i-coin]
        print('ways:  ', ways, coin)
    return ways[N]
```

[4]:
```python
coins = [1, 5, 10]
N = 8
print('Number of Ways to get {} = {}'.format(N, countWays(coins, N)))
```

```
values: [0, 1, 2, 3, 4, 5, 6, 7, 8]
ways:   [1, 1, 1, 1, 1, 1, 1, 1, 1] 1
ways:   [1, 1, 1, 1, 1, 2, 2, 2, 2] 5
ways:   [1, 1, 1, 1, 1, 2, 2, 2, 2] 10
Number of Ways to get 8 = 2
```

[5]:
```python
N = 10
print('Number of Ways to get {} = {}'.format(N, countWays(coins, N)))
```

```
values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
ways:   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] 1
ways:   [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3] 5
ways:   [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 4] 10
Number of Ways to get 10 = 4
```

```
[6]: N = 12
print('Number of Ways to get {} = {}'.format(N, countWays(coins, N)))
```

```
values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
ways:   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] 1
ways:   [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3] 5
ways:   [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 4, 4, 4] 10
Number of Ways to get 12 = 4
```

1.8 find minimum number of coins that make a given value/change

- Problem:
  - Input: $coins = [5, 10, 25], N = 30$
  - Output: 2
  - Combinations: $25 + 5 = 30$

```
[50]: import math

# DP solution for min coin count to make the change N
def minCoins(coins, N):
    # count list stores the minimum number of coins required for i value
    # all values 0-N are initialized to infinity
    count = [math.inf]*(N+1)
    # base case
    # no. of coin required to make 0 value is 0
    count[0] = 0
    # computer min coins for all values from 1 to N
    for i in range(1, N+1):
        for coin in coins:
            # for every coin smaller than value i
            if coin <= i:
                if count[i-coin]+1 < count[i]:
                    count[i] = count[i-coin]+1
    return count[N]
```

```
[51]: coins = [1, 3, 4]
N = 6
print('min coins required to give total of {} change = {}'.format(N,␣
 ↪minCoins(coins, N)))
```

```
min coins required to give total of 6 change = 2
```

1.9 Exercises

1. Ocean's Anti-11 - https://open.kattis.com/problems/anti11
   - Hint: count all possible n length binary integers (without 11) for the first few (2,3,4) positive integers and you'll see a Fibonaccii like pattern that gives the total number of possible binaries without 11 in them

- Write a program that finds factorials of a bunch of positive integer numbers. Would memoization improve time complexity of the program?

[ ]:
```

```