Kristin Schaefer

HW3 : Dynamic Programming

Sources:

Problem 1

1. https://www.youtube.com/watch?v=5o-kdjv7FD0&list=PLBZBJbE_rGRVnpitdvpdY9952IsKMDuev&index=2

1. Block Puzzle

a. description of approach to solve block puzzle using Dynamic Programming paradigm

N = 0	=>	{ [0] }	=> arguably 0 or 1 combinations
N = 1	=>	{ [1] }	=> 1 combinations
N = 2	=>	{ [1+1], [2] }	=> 2 combinations
N = 3	=>	{ [1+1+1], [1+2], [2+1] }	=> 3 combinations
N = 4	=>	{ [1+1+1+1], [1+1+2], [1+2+1], [2+1+1], [2+2] }	=> 5 combinations

N	# combinations
0	1
1	1
2	2
3	3
4	5

1+1 = 2 => N(2)

1 + 2 = 3 => N(3)

2 + 3 = 5 => N(4)

5 + 3 = 8 => ...

It is apparent from analyzing the table of results that the number of combinations possible given N total length is equal to the sum of combinations from the previous two solutions, so:

blockpuzzle(N) = blockpuzzle(N-1) + blockpuzzle(N-2)

This is the same approach as solving for Fibonacci sequence, so function definitions and time complexity analysis will be modeled on this approach (following examples given in class lectures).

Kristin Schaefer

HW3 : Dynamic Programming

b. bottom-up approach

```
blockpuzzle_dp(N)
        # base case
        if N = 0 or N = 1:
                return 1
        # make an array of size N + 1 to hold solutions
        combinations = [0] * (N+1)
        # for loop to fill the combinations solutions array in bottom-up approach
        for i in range 0 to N:
                if i = 0 or i = 1:
                        combinations[i] = 1
                else:
                        combinations[i] = combinations[i-1] + combinations[i-2]
        # return only the last result of the combinations array, as this contains the final solution
        return combinations[N]
```

c. brute force approach

```
blockpuzzle_bf(N)
        # base case
        if N = 0 or N = 1:
                return 1
        # recursive case
        else:
                return blockpuzzle_bf(N-1) + blockpuzzle_bf(N-2)
```

d. time complexity analysis comparison

Bottom-up approach: We are executing a for loop for range n. Hence our time complexity will be $\Theta(n)$

Brute force approach: $T(n) = T(n-1) + T(n-2) + 1 = \Theta(2^{n/2})$

Kristin Schaefer

HW3 : Dynamic Programming

e. recurrence relation

Although a recursive solution was not used to write the bottom-up function of blockpuzzle_dp(N), the approach can be formulated as:

$$\text{blockpuzzle}(N) = \begin{cases} 0 & N = 0 \\ 1 & N = 1 \\ \text{blockpuzzle}(N-1) + \text{blockpuzzle}(N-2) & N > 1 \end{cases}$$

2. Game

c. description of approach to solve Game puzzle using Dynamic Programming paradigm

N = 1	=>	Arguably False
N = 2	=>	A wins / True
N = 3	=>	B wins / False
N = 4	=>	A wins / True
N = 5	=>	B wins / False
N = 6	=>	A wins / True
N = 7	=>	B wins / False
N = 8	=>	A wins / True
N = 9	=>	B wins / False
N = 10	=>	A wins / True
N = 11	=>	B wins / False
N = 12	=>	A wins / True
N = 13	=>	B wins / False
...		

N	result
1	0
2	1
3	0
4	1
5	0
6	1
7	0
8	1
9	0
10	1
11	0
12	1
13	0
...	

!(N-1)

!(N-1)

I approached solving the problem by creating a table of the results of N from N(1), N(2),..., N(13).

The results show that when N is odd the result is False, and when N is even the result is True.

Thus, you can say Game(N) = !Game(N-1).

For the top-down approach I made a recursive function which first checks if the solution in the topdown_memo array already exists. If the solution has not been calculated, then a recursive call is made to fill the topdown_memo array in a top-down fashion.

For the bottom-up approach, I started by initializing an empty array to hold the results. I then use a for loop to calculate each result and store it in the array, starting from the base case.

Kristin Schaefer

HW3 : Dynamic Programming

d. time and space complexity of top-down approach

$T(n) = T(n-1) + \Theta(1) \Rightarrow \Theta(n)$

We need space of size n for the memo array which requires O(n) space complexity.

e. time and space complexity of bottom-up approach

We are execute a for loop for range n, so: $n * \Theta(1) \Rightarrow \Theta(n)$

We need space of size n for the memo array which requires O(n) space complexity.

f. subproblem and recurrence formula

$$\text{Game}(N) = \begin{cases} \text{False} & N = 1 \\ !\text{Game}(N-1) & N > 1 \end{cases}$$

Thus the subproblems of Game(N) are then Game(N-1), Game(N-2), ..., Game(1)