UNIVERSITY OF WATERLOO

CS Fundamentals

Contents

1 Dynamic Programming

- Dynamic programming is a technique to solve a subset of problems in an efficient way.
- There are problems which take exponential time to solve if DP is not used.
- DP is enhanced recursion.
- DP is an improved approach based on recursive solution. DP is recursion with Storage.
- Recursion might calculate a same value over an over again within its nested calls. DP uses a storage mechanism to avoid making inefficient recursive calls.
- Recursion + Table = Memoization
- Top-Down approach is when we only use Storage of table, to intelligently calculate further sub-problems.

1.1 How to check a problem for DP?

1. There is a choice to add or remove an item.
2. An optimal solution is required.

1.2 Approach new DP Problem

- Write a recursive solution to the problem, EASY.
- Memoize the recursive function.
- Then go for TOP-DOWN approach.

1.3 Problem Pattern

There are following 10 problem patterns which apply DP:

1. 0-1 Knapsack
    - Subset sum
    - Equal sum partition
    - Count of subset
    - Minimum subset sum difference
    - Count the number of subset with a given difference
    - Target sum
2. Unbounded Knapsack
3. Fibonacci
4. LCS: Longest Common Subsequence
5. LIS: Longest Increasing Subsequence
6. Kadaneś Algorithm
7. Matrix Chain Multiplication*
8. DP on Trees
9. DP on Grid
10. Others

1.4 Detailed Knapsack Visit

Problem Statelemt

Knapsack problem is when we are given a bag of weight W, a weight array Wt and a value array Val.

We have to output the maximum value output with choices of input in bag. The condition is sum of weights is less than or equal to W.

Knapsack has following types:

- 0-1 Knapsack: value is added or not.
- Fractional Knapsack: we can add fraction of an item.
- Unbounded Knapsack: Item can be repeated!

1.4.1 Knapsack: Recursive

```
def recursive_knapsack(wt, val, W):

    # Smallest possible input
    if len(wt) == 0 or W == 0:
        return 0

    # Get rid of last item because weight of item is greater
    # than the W itself.
    if wt[-1] > W:
        # Advance in recursion without last item.
        return recursive_knapsack(wt[:-1], val[:-1], W)
    else:
        # Weight of last item is within W, thus, 2 possibilities are there:
        # max(consider item, not consider item)
        return max(val[-1] + recursive_knapsack(wt[:-1], val[:-1], W-wt[-1])
                                                  ,
                   recursive_knapsack(wt[:-1], val[:-1], W))
```

- Method calls itself with a smaller sub problem.
- Repetitive calls to same method utilizes call stack.
- Recursive solutions are simple but inefficient as system stacks may overflow.

Whenever there is a single recursive call, DP might not be best to use. DP is helpful when 2 or more recursive calls are present within the same method.

To come up with recursive solution:

- Think of the smallest possible input in the problem (BASE CONDITION)
- Think about choice diagram. If there is an item, make a small condition diagram of effects or including or not including that item.

1.4.2 Knapsack: Memoize

```
def knapsack_memoize(wt, val, W):
    t = []

    n = len(wt)

    # Memoization table.
    for i in range(n + 1):
        t.append([-1] * (W + 1))

    def helper_memoize(wt, val, W, n):

        # Smallest possible input
        if n == 0 or W == 0:
            return 0

        if t[n][W] != -1:
            # If memory table already has that entry calculated, use it.
            return t[n][W]

        # Get rid of last item because weight of item is greater
        # than the W itself.
        if wt[n - 1] > W:
            # Advance in recursion without last item.
            # but store the results in the memory table.
            t[n][W] = helper_memoize(wt, val, W, n - 1)
            return t[n][W]
        else:
            # Weight of last item is within W, thus, 2 possibilities are
                                                   there:
            # max(consider item, not consider item)
            # store the results in memory table.
            t[n][W] = max(val[n - 1] + helper_memoize(wt, val, W - wt[n - 1]
                                                   , n - 1),
                          helper_memoize(wt, val, W, n - 1))

            return t[n][W]

    return helper_memoize(wt, val, W, n)
```

- Memoization is a combination of recursive solution and storage.
- Some extra storage is used to avoid extra recursive calls. This storage stores values that a recursive solution tends to calculate over and over again.
- Recursion is still involves, therefore, there is a chance for the stack to overflow.
- Therefore, memiozation is useful where number of recursive calls are less than system stack.

1.4.3 Knapsack: Top Down DP

```
    n = len(wt)

    # DP One line initialization.
    t = [[0 if i == 0 or j == 0 else -1 for j in range(W+1)] for i in range(
                                       n+1)]

    # Because initialization is already done for n = 0, W = 0.
    for i in range(1, n+1):
        for j in range(1, W+1):
            if wt[i-1] <= j:
                # Convert n to i and W to j from recursive solution.
                t[i][j] = max(val[i-1]+t[i-1][j-wt[i-1]], t[i-1][j])
            else:
                t[i][j] = t[i - 1][j]

            pprint(t)

    return t[n][W]

# The final DP table becomes:
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
 [0, 3, 3, 3, 4, 7, 7, 7, 7, 7, 7],
 [0, 3, 3, 3, 4, 7, 8, 8, 8, 9, 12],
 [0, 3, 3, 3, 4, 7, 8, 8, 10, 10, 12]]
```

1.5 Problems

1.5.1 Subset Sum Problem

Input: [2 3 7 8 10]

Sum: 11

Is there a subset of array which has sum 11?

Output: True/False

Similarity: Sum is Weight of Knapsack. If there is only 1 array consider it Weight array in Knapsack. There is also choice weather to add an element in subset or not.