Anash Mehta
USC ID: 6498-8280-04

CSCI 570 - Fall 2022 - HW 6

1. From the lecture, you know how to use dynamic programming to solve the 0-1 knapsack problem where each item is unique and only one of each kind is available. Now let us consider knapsack problem where you have infinitely many items of each kind. Namely, there are *n* different types of items. All the items of the same type *i* have equal size w_i and value v_i. You are offered with infinitely many items of each type. Design a dynamic programming algorithm to compute the optimal value you can get from a knapsack with capacity *W*.

   Answer:

   Let there be two arrays, weights[w_0, w_1, …., w_n] and values[v_0, v_1, …, v_n]. These will contain the weights and respective values of the different items in the set.

   The total given capacity is *W* and *n* are the number of elements.

   The optimal value at (i, j) is equal to the value of the optimal solution using the subset [1…i] with the maximum allowed weight *j*.

   The formula will be:

   If *i* belongs to optimal solution -> opt(i, j) = value[i-1] + opt(i, j – weights[i-1])

   If *i* does not belong to the solution -> opt(i, j) = opt (i-1, j)

   Let M[n][W] be a 2-D array which stores the optimal solution for each block for memoization where first row and first column are initialized as 0 and rest of them with -1.

   Algorithm:

```
Infinite-Knapsack(weights, values, n, W)
    If M[n][w] != -1
        Return M[n][w]
    Else if
        Return M[n][w] = Max(value[n-1] + Infinite-Knapsack(weights, values, n, W-
        weights[n-1], Infinite-Knapsack(weight, value, n-1, W))
    Else
        Return M[n][w] = Infinite-Knapsack(weight, value, n-1, W)
    End If
End Function
```

   The optimal solution will be stored at M[n+1][w+1].

2. Given a non-empty string *s* and a dictionary containing a list of unique words, design a dynamic programming algorithm to determine if *s* can be segmented into a space-separated sequence of one or more dictionary words. If *s* = *"algorithmdesign"* and your dictionary contains *"algorithm"* and *"design"*. Your algorithm should answer Yes as *s* can be segmented as *"algorithmdesign"*.

Answer:

We parse the string and check with the dictionary. For this, let us create an array S[1….n] where n is the size of the string s. Let OPT(n) show whether the string can be segmented or not. If yes, we denote it as 1 or else we will denote it to 0. We can say that the string can be segmented if the last character can also be segmented.

The algorithm would be:

```
checkStr(0, n):
  for i=0 to n:
          if n <= 2:
                  return s
          end if
          for j=0 to n:
                  if(word[i,j] in dictionary):
                          checkStr(j+1, n)
                  end if
          end for
  end for
end function
```

The initialization will be OPT(0) = 0, and the time complexity of the algorithm would be O(n^2).

3. Given *n* balloons, indexed from 0 to $n - 1$. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If you burst balloon *i* you will get $nums[left] * nums[i] * nums[right]$ coins. Here left and right are adjacent indices of *i*. After the bursting the balloon, the left and right then becomes adjacent. You may assume $nums[-1] = nums[n] = 1$ and they are not real therefore you can not burst them. Design a dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.

   Here is an example. If you have the nums arrays equals [3,1,5,8]. The optimal solution would be 167, where you burst balloons in the order of 1, 5 3 and 8. The left balloons after each step is:

   $$[3,1,5,8] \rightarrow [3,5,8] \rightarrow [3,8] \rightarrow [8] \rightarrow []$$

   And the coins you get equals:

   $$(3 * 1 * 5) + (3 * 5 * 8) + (1 * 3 * 8) + (1 * 8 * 1) = 167$$

Answer:

Here, we define an array A which contains the number of balloons. For the edge cases, we will add the first and last value as 1. Thus, the array A looks like {1, nums[0], nums[1] …, nums[n-1], 1}. The size of A would be n+2.

We will try bursting the edge balloons at the end because bursting them before could reduce the value of the optimal solution.

We define an array M where M[i, j] represents the maximum coins collected by bursting the balloons [(i+1), (j+1)] and optimal solution for the subset nums[0…i]. We define M[n+2][n+2] such that it represents the maximum coins collected by bursting balloons [(i+1)…(j-1)]

The first case would be 0 because we have no balloons to burst.

The formula would be: M[i][j] = Max(M[i][k] + M[k][j] + nums[i] * nums[k] * nums[j])

where $(i+1) \le k \le (j+1)$

The final answer is M[0, n+1].

Algorithm:

```
for s=2 to n+2
  for i=0 to n+2-s+1
      // Size 2 sub-arrays correspond to the base case (0 coins collected)
      if(s == 2)
          M[i][i+s-1] = 0;
      else
          // Select the one which maximises the number of coins collected
       for j=i+1 to (i+s-1)
         M[i][i+s-1] = Max(M[i][i+s-1], M[i][j] + M[j][i+s-1] + nums[i]*nums[j]*nums[i+s-1]);
       end for
      end if
  end for
end for
```

Our algorithm takes O(n) time for each pass, and there are O(n^2) passes. Hence, the total time complexity is O(n^3).

4. Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth pi dollars. Devise a Dynamic Programming algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces.

   Answer:

   This problem is a similar problem to the Infinite Knapsack problem.

   Let there be two arrays, lengths[l_0, l_1, …., l_n] and prices[p_0, p_1, …, p_n]. These will contain the lengths and respective prices of the each subsize of the rod.

   The optimal value at (i, j) is equal to the value of the optimal solution using the subset [1…i] with the maximum allowed length *j*.

   The formula will be:

   If *i* belongs to optimal solution -> opt(i, j) = prices[i-1] + opt(i, j – lengths[i-1])

   If *i* does not belong to the solution -> opt(i, j) = opt (i-1, j)

   Let M[n][N] be a 2-D array which stores the optimal solution for each block for memoization where first row and first column are initialized as 0 and rest of them with -1.

   Algorithm:

   MaxValueOfRod(lengths, prices, n, N)

```
        If M[n][N] != -1
```

```
            Return M[n][N]
        Else if
            Return M[n][N] = Max(value[n-1] + MaxValueOfRod(lengths, prices, n, N- lengths
            [n-1], MaxValueOfRod(lengths, prices, n-1, N))
        Else
            Return M[n][N] = MaxValueOfRod(lengths, prices n-1, N)
        End If
    End Function
```

The optimal solution will be stored at M[n+1][N+1].

5. Solve Kleinberg and Tardos, Chapter 6, Exercise 6.

Answer:

We observe that the last line ends with word w_n and then starts with some word w_j. Breaking off words w_j to w_n, we have a recursive sub problem on $w_1 \ldots w_{j-1}$.

Thus, we define OPT(i) as the value of optimal solution on the set of words $W_i = (w_1, \ldots., w_i)$. For any $i <= j$, let S_{ij} denote the slack of a line containing words w_i to w_j. For each fixed *i*, we can compute S_{ij} in linear time by considering values of j in non-decreasing order. Thus, we compute the entire string in $O(n^2)$ time.

The optimal solution must begin at the last line somewhere (at word w_j) and we solve the subproblem on the earlier lines in an optimal fashion. Thus, we find the formula as:

$$OPT(n) = Min\ (S^2_{ij} + OPT(j-1))\ \text{where}\ 1<= j <= n$$

And the line of words $w_j, \ldots, w_n$ is used in the optimal solution if the minimum is obtained with index *j*.

The loop to get the value would be:

```
Compute all values $S_{ij}$ as described above.
Set OPT(0) = 0
for k=1 to n:
    OPT(k) = Min ($S^2_{ij}$ + OPT(j-1)) where 1<= j <= n
end for
return OPT(n)
```

6. Solve Kleinberg and Tardos, Chapter 6, Exercise 10

Answer:
(a)

A	15	5	5	75	5
B	10	2	12	3	6

The mentioned algorithm will first select A1 (15). Now, since we selected A1, we will compare B3 (12) with A2+A3 (5+5=10). Since it is greater, we skip minute 2 and then proceed to process B3. Now, we compare if A5 is greater than B4+B5 which it is not. So, it carries forward with B4 and then moves to B5. The total value hence here is 15 + 12 + 3 + 6 = 36.

But the optimal solution is 15+5+5+75=90. Hence, the mentioned algorithm does not give us the optimal solution always.

(b)
We have two matrices, A and B which are of size n and contain the values for a and b.

Here, each element represents the optimal solution for that position. So $OPT(A_i)$ represents the optimal solution for Matrix A at position i.

The optimal solution's formula would be:
$OPT(A_i) = P(A_3) + Max(OPT(A_i – 1), OPT(B_i – 2))$
$OPT(B_i) = P(B_3) + Max(OPT(B_i – 1), OPT(A_i – 2))$

We initialize values 0-2, since the most backwards the equation can go is two steps. For case 0, we will select from whichever is the greater at 0th position. For 1st and 2nd values, we calculate them before going forward with the loop.

<u>Algorithm:</u>

```
for i=3 to n:
        $OPT(A_i) = P(A_3) + Max(OPT(A_i – 1), OPT(B_i – 2))$
        $OPT(B_i) = P(B_3) + Max(OPT(B_i – 1), OPT(A_i – 2))$
end for
```

Once the algorithm's execution ends, we go to A[n] and B[n] and pick the greater value. This selected value contains the optimal solution.