CSCI 570 - Fall 2022 - HW 6

Due Oct. 12th

Note: This homework assignment covers dynamic programming. It is recommended that you read chapter 6.1 to 6.4 from Kleinberg and Tardos.

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.

   Similar to what is taught in the lecture, let $OPT(k, w)$ be the maximum value achievable using a knapsack of capacity $0 \leq w \leq W$ and with k types of items $1 \leq k \leq n$. We find the recurrence relation of $OPT(k, w)$ as follows. Since we have infinitely many items of each type, we choose between the following two cases:

   - We include another item of type k and solve the sub-problem $OPT(k, w - w_k)$.
   - We do not include any item of type k and move to consider next type of item this solving the sub-problem $OPT(k - 1, w)$.

   Therefore, we have

   $$OPT(k, w) = \max\{OPT(k - 1, w), OPT(k, w - w_k) + v_k\}.$$

   Moreover, we have the initial condition $OPT(0, 0) = 0$.

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 *"algorithm design"*.

   Let $s_{i,k}$ denote the substring $s_i s_{i+1} \ldots s_k$. Let $Opt(k)$ denote whether the substring $s_{1,k}$ can be segmented using the words in the dictionary, namely $OPT(k) = 1$ if the segmentation is possible and 0 otherwise. A segmentation of this substring $s_{1,k}$ is possible if only the last word (say $s_i \ldots s_k$) is in the dictionary the remaining substring $s_{1,i}$ can be segmented. Therefore,we have equation:

   $$Opt(k) = \max_{0<i<k \text{ and } s_{i+1,k} \text{ is a word in the dictionary}} Opt(i)$$

We can begin solving the above recurrence with the initial condition that $Opt(0) = 1$ and then go on to compute $Opt(k)$ for $k = 1, 2, \ldots, n$. The answer corresponding to $Opt(n)$ is the solution and can be computed in $\Theta(n^2)$ time.

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

   Let $OPT(l, r)$ be the maximum number of coins you can obtain from balloons $l, l+1, \ldots, r-1, r$. The key observation is that to obtain the optimal number of coins for balloon from l to r, we choose which balloon is the last one to burst. Assume that balloon k is the last one you burst, then you must first burst all balloons from l to $k-1$ and all the balloons from $k+1$ to r which are two sub problems. Therefore, we have the following recurrence relation:
$$OPT(l, r) = \max_{l \le k \le r}\{OPT(l, k-1) + OPT(k+1, r) + nums[k] * nums[l-1] * nums[r+1]\}$$
   We have initial condition $OPT(l, r) = 0$ if $r < l$. For running time analysis, we in total have $O(n^2)$ and computation of each state takes $O(n)$ time. Therefore, the total time 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.

   At each remaining length of the rod, we can choose to cut the rod at any point, and obtain points for one of the cut pieces and recursively compute the maximum points we can get for the other piece. It is also possible to recursively attempt to obtain the maximum value for both pieces after the cut, but that requires adding an extra check to see if the value obtained by selling a rod of this length is more than recursively cutting it up. The bottom-up pseudo-code to obtain the maximum amount of money is:

Algorithm 1 Bottom-Up-Cut-Rod(p, n)

```
Let $r[0, ..., n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$ do
    $q = -\infty$
    for $i = 1$ to $j$ do
        $q = \max(q, p[i] + r[j - i])$
    end for
    $r[j] = q$
end for
return $r[n]$
```

The time complexity of this algorithm is $\theta(n^2)$ because of the double nested for loop.

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

   Let $W = \{w_1, w_2, \dots, w_n\}$ be the set of ordered words which we wish to print. In the optimal solution, if the first line contains k words, then the rest of the lines constitute an optimal solution for the sub problem with the set $\{w_{k+1}, \dots, w_n\}$. Otherwise, by replacing with an optimal solution for the rest of the lines, we would get a solution that contradicts the optimality of the solution for the set $\{w_1, w_2, \dots, w_n\}$.

   Let $Opt(i)$ denote the sum of squares of slacks for the optimal solution with the words $\{w_i, \dots, w_n\}$. Say we can put at most the first p words from w_i to w_n in a line, that is, $\sum_{t=i}^{p+i-1} c_t + p - 1 \le L$ and $\sum_{t=1}^{p+i} c_t + p > L$. Suppose the first k words are put in the first line, then the number of extra space characters is

$$s(i,k) := L - k + 1 - \sum_{t=i}^{i+k-1} c_t$$

   So we have the recurrence

$$Opt(i) = \begin{cases} 0 & \text{if } p \ge n - i + 1 \\ \min_{1 \le k \le p}\{(s(i,k))^2 + Opt(i+k)\} & \text{if } p < n - i + 1 \end{cases}$$

   Trace back the value of k for which $Opt(i)$ is minimized to get the number of words to be printed on each line. We need to compute $Opt(i)$ for n different values of i. At each step p may be asymptotically as big as L. Thus the total running time is $O(nL)$.

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

   (a) Consider the following example: there are totally 4 minutes, the numbers of steps that can be done respectively on the two machines in the 4 minutes are listed as follows (in time order):

   - Machine A: 2, 1, 1, 200

- Machine B: 1, 1, 20, 100

The given algorithm will choose A then move, then stay on B for the final two steps. The optimal solution will stay on A for the four steps.

(b) An observation is that, in the optimal solution for the time interval from minute 1 to minute i, you should not move in minute i, because otherwise, you can keep staying on the machine where you are and get a better solution ($a_i > 0$ and $b_i > 0$). For the time interval from minute 1 to minute i, consider that if you are on machine A in minute i, you either (i) stay on machine A in minute $i-1$ or (ii) are in the process of moving from machine B to A in minute $i-1$. Now let $OPT_A(i)$ represent the maximum value of a plan in minute 1 through i that ends on machine A, and define $OPT_B(i)$ analogously for B. If case (i) is the best action to make for minute $i1$, we have $OPT_A(i) = a_i + OPT_A(i-1)$; otherwise, we have $OPT_A(i) = a_i + OPT_B(i-2)$. In sum, we have

$$OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\} :$$

Similarly, we get the recursive relation for $OPT_B(i)$:

$$OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\} :$$

The algorithm initializes $OPT_A(0) = 0, OPT_B(0) = 0, OPT_A(1) = a_1$ and $OPT_B(1) = b_1$. Then the algorithm can be written as follows:

```
$OPT_A(0) = 0$; $OPT_B(0) = 0$;
$OPT_A(1) = a_1$; $OPT_B(1) = b_1$;
for $i = 2, \cdots, n$ do
    $OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\}$;
    Record the action (either stay or move) in minute $i-1$ that achieves the maximum.
    $OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\}$;
    Record the action in minute $i-1$ that achieves the maximum.
end for
Return $\max\{OPT_A(n), OPT_B(n)\}$;
Track back through the arrays $OPT_A$ and $OPT_B$ by checking the action records from minute $n-1$ to minute 1 to recover the optimal solution.
```

It takes $O(1)$ time to complete the operations in each iteration; there are $O(n)$ iterations; the tracing backs takes $O(n)$ time. Thus, the overall complexity is $O(n)$.