Homework 6

Jiahao Liu February 24, 2024

1. 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".

Define S[i] to be true if the first i^{th} characters of s can be segmented into a space-separated sequence of one or more dictionary words. Otherwise, S[i] is false.
Initialization: S[0] = true, S[i] = false, for all $i \in [1, s.length]$
Recurrence Formula: S[i] = true iff there exists a j < i such that S[j] is true and the substring of s in range $[j+1, i]$ is in the dictionary.

```
Buttom Up Pass:
for(i = 1, i <= s.length, i++):
    for(j = 0, j < i, j++):
        If S[j] is true and s.substring(j+1, i) is in the dictionary:
            S[i] = true;
            break;
return S[s.length];
```

Time Complexity: If the length of s is n, the time complexity of the algorithm is $\Theta(n^2)$, this is polynomial time.

2. 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 bursting the balloon, the left and right then become 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.

To think dynamically, if we know the index of last burst balloon in one sub range balloons, we can easily get the maximum coins of this range by sum the maximum coins from the left and right subsequence and the coins from burst the last balloon.
Define f[i][j]: the maximum coins we can collect by bursting the balloons between i and j, not including i,j.
Initialization: $\forall i \geq j-1, f[i][j] = 0$
Recurrence Formula:

$$f[i][j] = \max_{i<k<j} \{f[i][k] + nums[i] * nums[k] * nums[j] + f[k][j]\}$$

```
Buttom Up Pass:
for(len = 2, len <= n, len++):
    for(i = -1, i <= n-len, i++):
        j = i + len;
        for(k = i+1, k < j, k++):
            f[i][j] = max{f[i][j], f[i][k] + nums[i] * nums[k] * nums[j] + f[k][j]};
return f[-1][n];
```

Time Complexity: $\Theta(n^3)$, this is polynomial time.

3. You are in Downtown of a city where all the streets are one-way streets. At any point, you may go right one block, down one block, or diagonally down and right one block. However, at each city block (i, j) you have to pay the entrance fees fee(i, j). The fees are arranged on a grid as shown below. Your objective is to travel from the starting point at the city's entrance, located at block (0,0), to a specific destination block (n, n). The city is laid out in a grid, and at each intersection or block (i, j), you might either incur a cost (pay an entrance fee) or receive a reward (get a payback) for passing through. These transactions are captured in a grid, with positive values representing fees and negative values representing paybacks You would like to get to your destination with the least possible cost. Formulate the solution to this problem using dynamic programming
a. Define (in plain English) subproblems to be solved.
b. Write the recurrence relation for subproblems.

	0	1	2	3	⋯	n
0	$fee_{(0,0)}$	$fee_{(0,1)}$	$fee_{(0,2)}$	$fee_{(0,3)}$	⋯	$fee_{(0,n)}$
1	$fee_{(1,0)}$	$fee_{(1,1)}$	$fee_{(1,2)}$	$fee_{(1,3)}$	⋯	$fee_{(1,n)}$
2	$fee_{(2,0)}$	$fee_{(2,1)}$	$fee_{(2,2)}$	$fee_{(2,3)}$	⋯	$fee_{(2,n)}$
3	$fee_{(3,0)}$	$fee_{(3,1)}$	$fee_{(3,2)}$	$fee_{(3,3)}$	⋯	$fee_{(3,n)}$
⋮	⋮	⋮	⋮	⋮	⋱	⋮
n	$fee_{(n,0)}$	$fee_{(n,1)}$	$fee_{(n,2)}$	$fee_{(n,3)}$	⋯	$fee_{(n,n)}$

Define f[i][j] to be the minimum cost to travel from the starting point at the city's entrance, located at block (0,0), to a specific destination block (i, j).
Initialization: $f[0][0] = fee(0,0), \forall i \in [1,n] : f[i][0] = f[i-1][0] + fee(i,0), f[0][i] = f[0][i-1] + fee(0,i)$
Recurrence Formula: $\forall i,j \in [1,n] : f[i][j] = \min\{f[i-1][j], f[i][j-1], f[i-1][j-1]\} + fee(i,j)$

Buttom Up Pass:

```
for(i = 1, i <= n, i++):
    for(j = 1, j <= n, j++):
        f[i][j] = min{f[i − 1][j], f[i][j − 1], f[i − 1][j − 1]} + fee(i, j);
return f[n][n];
```

Time Complexity: $\Theta(n^2)$, this is polynomial time.

4. Assume we have N workers. Each worker is assigned to work at one of M factories. For each of the M factories, they will produce a different profit depending on how many workers are assigned to that factory. We will denote the profits of factory i with j workers by P(i,j). Develop a dynamic programming solution to find the assignment of workers to factories that produce the maximum profit.(Mention the pseudocode)

Define f[i][j]: the maximum profit of assigning the first i factories with j workers.
Define p[i][j]: the number of workers assigned to the i^{th} factory when considering the best profits for j workers.
Initialization: Suppose $P(i,0) \geq 0$, then $\forall i \in [1,M] : f[i,0] = \sum_{k=1}^{i} P(k,0) \quad \forall j \in [0,N] : f[0,j] = 0$
Recurrence Formula:

$$f[i][j] = \max_{0 \leq k \leq j} \{f[i-1][j-k] + P(i,k)\} \qquad p[i][j] = k$$

Buttom Up Pass:

```
for(i = 1, i <= M, i++):
    for(j = 1, j <= N, j++):
        for(k = 0, k <= j, k++):
            // Update the maximum profit and the number of workers assigned to the i^th factory.
            If f[i][j] < f[i-1][j-k] + P(i, k):
                f[i][j] = f[i-1][j-k] + P(i, k);
                p[i][j] = k; // Store the number of workers assigned to factory m
return f[M][N];
```

Top Down Pass: Initialize an empty list or array A[M] to store the number of workers assigned to each factory.

```
j = N // Start with the total number of workers.
for(i = M, i >= 1, i–): // Have to start from top right to bottom left.
    A[i] = p[i][j];
```

j = j - p[i][j]; // Update the remaining number of workers for the next iteration
End of the loop.
Time Complexity: $\Theta(MN^2)$, this is polynomial time because input P(i,j) array is of size M*N.

5. You have two rooms to rent out. There are n customers interested in renting the rooms. The i^{th} customer wishes to rent one room (either room you have) for d[i] days and is willing to pay bid[i] for the entire stay. Customer requests are non-negotiable in that they would not be willing to rent for a shorter or longer duration. Devise a dynamic programming algorithm to determine the maximum profit that you can make from the customers over a period of D days.
a) Define (in plain English) subproblems to be solved. b) Write the recurrence relation for subproblems.

Intuitively, we need to keep track of the available days of two rooms for different number of customers.
Define f[i][j][k] to be the maximum profit we can make from the first i customers over a period of j days for the first room and k days for the second room.
Initialization: $\forall i \in [0, n] : f[i][0][0] = 0$
Recurrence Formula:

$$f[i][j][k] = \max\{f[i-1][j-d[i]][k] + bid[i], f[i-1][j][k-d[i]] + bid[i], f[i-1][j][k]\}$$

Bottom Up Pass:

for(i = 1, i <= n, i++):
  for(j = 0, j <= D, j++):
    for(k = 0, k <= D, k++):
      if(j >= d[i] and k >= d[i]):
        $f[i][j][k] = \max\{f[i-1][j-d[i]][k]+bid[i], f[i-1][j][k-d[i]]+bid[i], f[i-1][j][k]\}$;
      else if(j >= d[i]):
        $f[i][j][k] = \max\{f[i-1][j-d[i]][k] + bid[i], f[i-1][j][k]\}$;
      else if(k >= d[i]):
        $f[i][j][k] = \max\{f[i-1][j][k-d[i]] + bid[i], f[i-1][j][k]\}$;
      else:
        $f[i][j][k] = f[i-1][j][k]$;
return f[n][D][D];

Time Complexity: $\Theta(nD^2) = \Theta(n2^{2log_2 D})$, this is pseudo-polynomial time because input array is of size n and an integer D for total days.

6. You are given a sequence of n numbers (positive or negative) $x_1, x_2, \ldots, x_n$. Your job is to select a subset of these numbers of maximum total sum, subject to the constraint that you can't select two elements that are adjacent (that is, if you pick x_i then you cannot pick either x_{i-1} or x_{i+1}. On the boundaries, if x_1 is chosen, x_2 cannot be chosen; if x_n is chosen, then x_{n-1} cannot be chosen. Give a dynamic programming solution to find, in time polynomial in n, the subset of maximum total sum.

Define f[i] to be the maximum total sum of the subset of $x_1, x_2, \ldots, x_i$
Initialization: $f[0] = 0, f[1] = x_1$
Recurrence Formula:

$$f[i] = \max\{f[i-1], f[i-2] + x_i\}$$

Bottom Up Pass:

for(i = 2, i <= n, i++):
  $f[i] = \max\{f[i-1], f[i-2] + x_i\}$;
return f[n];

Then we need the Top Down Pass to reconstruct the subset:
Initialize an empty list or array A to store the subset, int j = n;
while(j >= 1):
  if(f[j] = f[j-1]):
    j = j - 1;
  else:
    A.append(x[j]);

j = j - 2;
End of the loop, and Return S.
Time Complexity: $\Theta(n)$, this is polynomial time.

7. 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 p_i 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.

Define f[i] to be the maximum amount of money you can get by cutting the rod of length i
Initialization: $f[0] = 0$
Recurrence Formula:

$$f[i] = \max_{1 \le j \le i} \{p_j + f[i-j]\}$$

```
Buttom Up Pass:
for(i = 1, i <= N, i++):
    for(j = 1, j <= i, j++):
        f[i] = max{f[i], p[j] + f[i − j]};
return f[N];
```

Time Complexity: $\Theta(N^2)$, this is polynomial time because input p[] array is of size N.