Homework #4
Introduction to Algorithms/Algorithms 1
601.433/633
Spring 2020

Mou Zhang
Due on: Tuesday, March 10th, 12pm

Where to submit: On Gradescope.
Please type your answers; handwritten assignments will not be accepted.

1 Problem 1 (15 points)

Suppose you are managing the construction of billboards on an east-west highway that extends in a straight line. The possible sites for billboards are given by reals $x_1, x_2, \ldots, x_n$ with $0 \leq x_1 < x_2 < \cdots < x_n$, specifying their distance in miles from the west end of the highway. If you place a billboard at location x_i, you receive payment $p_i > 0$.

Regulations imposed by the Baltimore County's Highway Department require that any pair of billboards be more than 5 miles apart. You'd like to place billboards at a subset of the sites so as to maximize your total revenue, subject to that placement restriction.

For example, suppose n = 4, with

$$\langle x_1, x_2, x_3, x_4 \rangle = \langle 6, 7, 12, 14 \rangle,$$

and

$$\langle p_1, p_2, p_3, p_4 \rangle = \langle 5, 6, 5, 1 \rangle.$$

The optimal solution would be to place billboards at x_1 and x_3, for a total revenue of $p_1 + p_3 = \$10$.

Give an $O(n)$ time dynamic-programming algorithm that takes as input an instance (locations $\{x_i\}$ given in sorted order and their prices $\{p_i\}$) and returns the maximum revenue obtainable. As usual, prove correctness and running time of your algorithm.

EDIT: We will give full credit for an $O(n \log(n))$ algorithm too.

Solution:

Algorithm 1 Maximize total revenue

```
1:  procedure MAXIMIZE TOTAL REVENUE(X, P)
2:      function CALCULATE PREV(X)
3:          i ← 0
4:          for j ← 1, X.length do
5:              while x_{i+1} < x_j − 5 do
6:                  i = i + 1
7:              end while
8:              prev[j] ← i
9:          end for
10:     end function
11:     DP[0] = 0
12:     DO calculate prev(X)
13:     for j ← 1, X.length do
14:         DP[j] ← max{DP[j − 1], DP[prev[j]] + P[j]}
15:     end for
16:     return DP[X.length]
17: end procedure
```

1. Define the subproblems (and write down what it means). If not for anything but to help you define the algorithm.
   - DP[i] = the maximum total revenue you get from the first i sites
2. Write down a recurrence showing how the larger subproblem can be solved in terms of the smaller ones.

- $$DP[i] = \begin{cases} 0 & i = 0 \\ \max\{DP[i-1], DP[prev[i]] + p_i\} & i \geq 0 \end{cases}$$
- In which prev[i] is the latest position such that $x_{prev[i]} < x_i - 5$, which can be defined as $\text{prev}[i] = \max\{j < i | x_j < x_i - 5\}$. The way to calculate prev[] is in the pseudo code.

3. Write down which subproblem gives the answer to the original question.
   - Output DP[n]
4. Prove correctness by proving the "optimal substructure property". I.e. The recurrence relation defined by 2. is the solution for the subproblem as defined by 1.
   - We want to prove that the recurrence relation for DP[i] in 2. is the the maximum total revenue you get from the first i sites
   - To do so, we will induct on i.
     (a) Base Case (i=0), the recurrence relation is correct as the revenue of none billboard is 0.
     (b) IH : for i $\leq$ k, DP[i] is the maximum revenues you get from the first i sites
     (c) Induction Step: For i = k + 1, there are two possible cases —
       – Case 1 : don't place billboard at site i, then the maximum revenue for the first k+1 sites is equal to the maximum revenue for the first k sites, which can be presented as $DP[i] = DP[i-1]$
       – Case 2 : place billboard at site i. Since you place a billboard at site i, you can not place any billboard in the range [$x_i - 5$, x_i]. The latest place before site i that you can place a billboard is prev[i]. So the formula can be calculated as the maximum revenue for the first site prev[i] + the revenue we get for billboard at site i,which can be presented as $DP[i] = DP[prev[i]] + p_i$
       – Since there are only 2 cases, the maximum revenue must be the bigger one of them. So $DP[i] = \max\{DP[i-1], DP[prev[i]] + p_i\}$. True.
5. Prove runtime by counting the number of subproblems.

- For the non-DP part, the runtime of calcularing array prev is obviously O(n) accoring to the pseudocode. For DP part, since we can use memoization to store the results of each sub problem, the runtime of the problem depends on the the number of times each sub problem gets called. As we can see from the equation in 2, each iteration only calls 2 result of subproblems. Because there are O(n) subproblmes $DP[0]...DP[n]$ and we do O(1) in each subproblems, the runtime of the algorithm is O(n). To sum up, the runtime is O(n).

2 Problem 2 (20 points)

You are given two numbers, n and k, such that $n \in \mathbb{N}$ and $k \in \{1, \ldots, 9\}$. Use dynamic programming to devise an algorithm which will find the number of $2n$-digit integers for which the sum of the first n digits is equal to the sum of the last n digits and each digit takes a value from 0 to k.

For example, when $k = 2$ and $n = 1$: you have only 3 such numbers 00, 11, 22. For example, when $k = 1$ and $n = 2$: you have only 6 such numbers 0000, 0101, 0110, 1001 , 1010, 1111.

Your algorithm should work in time polynomial of n and k. Prove correctness and provide running time analysis.

Solution:

Let's analysis this problem at first.As we can see, the number of permutations that satisfy our condition for a 2n-digit number is the square of the number of permutations for that sum using n digits. It is because that using the the number of 2n-digit with a certain sum is the number of permutation of the first half times the permutation of the second half and these two numbers are the same. One example is for n = 2 and k = 2. The way to get sum of 0 is 1, sum of 1 is 2 and sum of 2 is 1.Therefore, there are only $1^2 + 2^2 + 1^2 = 6$ ways to construct the 2n-digit number. So now, what we need to do is to calculate how many ways to construct a i-digit number with the sum of all digits is j.

1. Define the subproblems (and write down what it means). If not for anything but to help you define the algorithm.

Algorithm 2 count number of 2n-digit integers

```
1: procedure COUNT NUMBER OF 2N-DIGIT INTEGERS(X, P)
2:     Initialization case j = 0 or i = 0
3:     for i ← 1, n do
4:         for j ← 1, kn do
5:             for s ← 0, min{k, j} do
6:                 DP[i, j] ← DP[i, j] + DP[i − 1, j − s]
7:             end for
8:         end for
9:     end for
10:    ans ← 0
11:    for i ← 0, kn do
12:        ans ← ans + DP[n, i]
13:    end for
14:    return ans
15: end procedure
```

- DP[i, j] = the number of permutations to construct a i-digit number and the sum of all digits is equal to j.

2. Write down a recurrence showing how the larger subproblem can be solved in terms of the smaller ones.

   - $$DP[i,j] = \begin{cases} 1 & j = 0 \\ 0 & i = 0 \; and \; j \neq 0 \\ \sum_{s \in [0, \min\{k,j\}]} DP[i-1, j-s] & 1 \geq 0 \; and \; j \geq 0 \end{cases}$$

3. Write down which subproblem gives the answer to the original question.

   - Output $\sum_{i \in [0, k \times n]} DP[n,i]^2$

4. Prove correctness by proving the "optimal substructure property". I.e. The recurrence relation defined by 2. is the solution for the subproblem as defined by 1.

   - We want to prove that the recurrence relation for DP[i, j] in 2. is the number of permutations to construct a i-digit number and the sum of all digits is equal to j
   - To do so, we will induct on (i, j).

(a) Base Case (i=0 or j = 0), the recurrence relation is correct. It is because that when i = 0, there are 0 digits and the sum must be 0. When j = 0, there is always 1 way to get sum of zero - all digits are equal to 0. True.

(b) IH : for i $\leq$ k1 and j $\leq$ k2, DP[i, j] is the number of permutations to construct a i-digit number and the sum of all digits is equal to j.

(c) Induction Step: For i = k1 + 1 and j = k2 + 1, there are at most k cases —

- There are at most k cases, which are the cases that the number at position i is equal to [0...k]. If the number at position i = s. Then all permutations that use i - 1 digits and the sum of i-1 digits are equal to j - s (j -s must be bigger than 0) can contribute to the permutation numbers of i digits with sum of j. And these cases are all distinct. So DP[i - 1, j - s] must be calculated into DP[i, j], which is DP[i, j] += DP[i - 1, j - s].
- Suming up all k cases above, we have $DP[i, j] = \sum_{s \in [0, \min\{k,j\}]} DP[i - 1, j - s]$

5. Prove runtime by counting the number of subproblems.

- There are at most (n * (n * k)) unique subproblems. Using memoization, and by the definition of the algorithm in 2., calculating each subproblem only calls other subproblems at most k times(the $\sum$ part in the formula in 2) and since we do O(1) in each subproblem, the runtime of the algorithm is O(n * (n * k) * k) = O(n^2k^2). The runtime of calaulating answer in line 14 is O(n). So the total time complexity is O(n^2k^2) + O(n) = O(n^2k^2).

3 Problem 3 (15 points)

Alice and Bob found a treasure chest with different golden coins, jewelry and various old and expensive goods. After evaluating the price of each object they created a list $P = \{p_1, \ldots, p_n\}$ for all n objects, where $p_i \in \{1, \ldots, K\}$ is the price of the object i. Help Alice and Bob to check if the treasure can be divided equally, i.e. if it is possible to break the set of all abjects P into two parts P_A and P_B such that $P_A \cup P_B = P$, $P_A \cap P_B = \emptyset$ and $\sum_{i \in P_A} p_i = \sum_{i \in P_B} p_i$?

Your algorithm should run in time polynomial in n and K. As usual, prove correctness and running time of your algorithm.

Solution:

Algorithm 3 Split golden coins

```
1: procedure SPLIT GOLDEN COINS(X, P)
2:     Initialization case j = 0 or i = 0, calculating T
3:     for i ← 1, n do
4:         for j ← 1, T/2 do
5:             DP[i, j] ← DP[i − 1, j]
6:             if g ≥ p_i then
7:                 DP[i, j] ← DP[i, j] OR DP[i − 1, j]
8:             end if
9:         end for
10:    end for
11:    return DP[n][T/2]
12: end procedure
```

Assume $T = \sum P_i$. The original problem is equal to the following statement: If such P_A, P_B exist, such that $\sum_{i \in P_A} p_i = \sum_{i \in P_B} p_i = \frac{T}{2}$, which is the same as $\exists P_A$, s.t. $\sum_{i \in P_A} p_i = \frac{T}{2}$.

1. Define the subproblems (and write down what it means). If not for anything but to help you define the algorithm.
   - DP[i, j] = If using the first i goods to get j total values is possible, which is the same as $\exists P_A$, s.t. $\sum_{i \in P_A} p_i = j$.(Presented by True/False)
2. Write down a recurrence showing how the larger subproblem can be solved in terms of the smaller ones.
   - $$DP[i,j] = \begin{cases} True & j = 0 \\ False & i = 0 \; and \; j \neq 0 \\ DP[i-1,j] & i \geq 0 \; and \; j < p_i \\ DP[i-1,j] \; OR \; DP[i-1, j-p_i] & i \geq 0 \; and \; j \geq p_i \end{cases}$$
3. Write down which subproblem gives the answer to the original question.

- Output DP[n, $\frac{T}{2}$]

4. Prove correctness by proving the "optimal substructure property". I.e. The recurrence relation defined by 2. is the solution for the subproblem as defined by 1.

   - We want to prove that the recurrence relation for DP[i, j] in 2. is the correct True/False of whether using some of the first i elements to get the total value j is possible.
   - To do so, we will induct on (i, j).
     (a) Base Case (i=0 or j = 0), the recurrence relation is correct. It is because when j = 0, there is always a way to get 0 value - to take nothing. When i = 0 and j $\neq$ 0, since there's no good, getting value of j is impossible.
     (b) IH : for i $\leq$ k1 and j $\leq$ k2, DP[i, j] is the the True/False of whether using some of the first i elements to get the total value j is possible.
     (c) Induction Step: For i = k1 + 1 and j = k2 + 1, there are two possible cases —
     - Case 1 : take the i_th good in P_A. In this case, the True/False state of DP[i,j] is equal to the True/False state of using the first i - 1 goods to get a total value of j - p_i, which can be presented as DP[i, j] = DP[i - 1, j - p_i]. (This case only happens when j $> p_i$ because if j ¡ p_i, the sum of the value when taking p_i must be bigger than j)
     - Case 2 : don't take the i_th good in P_A. In this case, since we don't use the i_th good, the problem is equal to use the first i - 1 goods to get total value j. This can be presented as DP[i, j] = DP[i - 1, j]
     - Since there are only 2 cases and either case above is true will make DP[i, j] becomes True, the maximum revenue must be the bigger one of them. So

$$DP[i,j] = \begin{cases} DP[i-1,j] & i \geq 0 \; and \; j < p_i \\ DP[i-1,j] \; OR \; DP[i-1, j-p_i] & i \geq 0 \; and \; j \geq p_i \end{cases}$$

     is the correct True/False state of whether using some of the first i elements to get the total value j is possible. True.

5. Prove runtime by counting the number of subproblems.

- There are at most (n * $\frac{T}{2}$) unique subproblems. Using memoization, and by the definition of the algorithm in 2., each subproblem only calls other subproblems at most twice and since we do O(1) in each subproblem, the runtime of the algorithm is O(n * $\frac{T}{2}$) = O(n * n * k / 2) = O(n^2k). The initialization part takes O(n) time. So the total time complexity is O(n^2k) + O(n) = O(n^2k).