Algorithm Design and Analysis Assignment 2
Dynamic Programming

Xinmiao Zhang
202028013229129

November 19, 2020

1 Money robbing

A robber is planning to rob house along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

1. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.
2. What if all houses are arranged in a circle?

1.1 Array-like House Problem

1.1.1 Optimal Substructure and DP equation

Assuming that there is n houses on the street. We set array $M = (m_1, m_2, .., m_n)$ is the money stashed in n houses respectively, then we can transform this problem into a multistep decision problem.

Considering the very first step: decide whether the robber will rob the nth house:

1. If rob nth house, then the robber can not rob the $n-1$th house preventing from calling police. Then the problem transforms to the maximum value of rob the 1st to the $n-2$th houses plus the nth house.
2. Otherwise, apparently the problem transforms to the maximum value of rob the 1st to the $n-1$th houses.

Accoring to the analysis before, we can easily find a *Optimal Substructure*: the maximum value of rob the 1st to the ith house, we note this variable as $v[i]$. It is trival to decide that $v[0] = 0$ and $v[1] = m_1$. Then we can easily write the DP equation as below:

$$v[i] = \begin{cases} 0 & i = 0 \\ m_1 & i = 1 \\ \max(v[i-2] + m_i, v[i-1]) & else \end{cases}$$

1.1.2 Algorithm Description

First of all, we initialize the array v. We set $v[0] = 0$ and $v[1] = m_1$. And we calculate each $v[i]$ according to the DP equation. Finally we can calculate $v[n]$ which is the max value we want to solve.

We assume that the number of house is n, and the value of each house is expressed in an array $M = (m_1, m_2, .., m_n)$. The Pseudo Code is shown as below:

```
FIND-BEST-ROBBERY-METHOD(M, n)
1  Allocate v as a n-element array
2  v[0] = 0, v[1] = m_1
3  for i = 2 to n
4        v[i] = max(v[i-2] + m_i, v[i-1])
5  return v[n]
```

1.1.3 Correctness Proving

In this problem, for the ith house there are two possibilities(rob or not). If rob, then the solution to the subproblem can be calculated out in equation $v[i] = v[i-1]$.Otherwise, it can be calculated out by $v[i] = v[i-2]+m_i$. And to obtain the correctness, we take the maximum of the two values, which can guarantee that solution is the maximum, in other words, proved the correctness.

1.1.4 Algorithm Analysis

For time complexity, we just do a assignment through an array which length is n. For every element we assign a value to it, so apparently the time complexity is as follows:

$$T(n) = O(n)$$

For space complexity, according to the Pseudo-code before, we only need one data structure which is a n-element array. So the space complexity is:

$$S(n) = O(n)$$

1.2 Circle-like House Problem

1.2.1 Optimal Substructure and DP equation

Compared to the Array-like version, the houses are circle-like, which means that the 1st house and the nth house can only choose one to rob. So the problem can be seen as two subproblem:

1. If not rob the 1st house, we can call FIND-BEST-ROBBERY-METHOD($M[2:n], n-1$) to find the max value of rob other houses, mark it as V_1.

2. If not rob the nth house, we can call FIND-BEST-ROBBERY-METHOD($M[1:n-1], n-1$) to find the max value of rob other houses, mark it as V_2.

And we can finally find the solution to the problem by calculating the max value of V_1 and V_2, i.e. $\max(V_1, V_2)$.

1.2.2 Algorithm Description

We can get the solution to the problem by calling two FIND-BEST-ROBBERY-METHOD whose parameters' length is $n-1$. The Pseudo code is shown below:

FIND-BEST-ROBBERY-METHOD-CIRCLE(M, n)
1 V_1 = FIND-BEST-ROBBERY-METHOD($M[2,..,n], n-1$)
2 V_2 = FIND-BEST-ROBBERY-METHOD($M[1,..,n-1], n-1$)
3 **return** $\max\{V_1, V_2\}$

1.2.3 Correctness Proving

The only change between circle-like and array-like problem is that, when we solve the circle-like problem, when 1st house is robbed, then the nth house can not be robbed. So there is only two situation:

1. The nth house is not robbed, which means that 1st and $n-1$th house is robbed or not have no relationship with the status of nth house.

2. The 1st house is not robbed, which means that 2nd and nth house is robbed or not have no relationship with the status of 1st house.

So, the two circumstance have no relationship with the status of the other element, they are *independent* problems. We can solve the problem by our previous algorithm, and then get the maximum to get the final result.

1.2.4 Algorithm Analysis

For time complexity, it is apparently that we have to go through the FIND-BEST-ROBBERY-METHOD for twice.

$$T(n) = O(n)$$

For space complexity, considering about the real time system can allocate and free the memory dynamically, so we only need the same space complexity, which is also a n-element array. So the space complexity is:

$$S(n) = O(n)$$

2 Coin Change

You are given coins of different denominations and a total amount of money amount. Write a function to compute **the total number of ways** to make up that amount using some of those coins.

Note: You may assume that you have an infinite number of each kind of coin.

2.1 Optimal Substructure and DP equation

We denote *amount* as the totol amount of money. And we assume that there are n kinds of coins, and their denomination array is $D = D[1], D[2], \ldots, D[n]$. Then we can describe the problem as a multi-step decision problem. Considering the very first step:

If we pick k coins from the nth kind of coins, ($Note: k * D[n] \leq amount$), then we only left $n-1$ kinds of coins and $amount - k * D[n]$ money to consist of. Then the subproblem will transform to calculate the total number of ways to make up $amount - k * D[n]$ using $D[1 \ldots n-1]$.

So we find a *Optimal Substructure* $ways[i][j]$, which denotes the total number of ways to make up j amount of monet using the first i kinds of coins. It is apparently that when $i = 1$, if $j\%D[1] = 0$, then there is only one method to make up the amount. Else there is no method to make up j amount. According to the description before, we can write the DP equation as below:

$$ways[i][j] = \begin{cases} 0 & i = 0 or j = 0 \\ 0 & i = 1, j\%D[1] \neq 0 \\ 1 & i = 1, j\%D[1] = 0 \\ \sum_{k=0}^{\lfloor j/D[i] \rfloor} ways[i-1][j - k * D[i]] & 1 < i \leq n, 0 < j \leq amount \end{cases}$$

2.2 Algorithm Description

First of all, we initialize the matrix $ways[i][j]$. Let $ways[1][k * D[1]] = 1(k \leq \lfloor n/D[1] \rfloor)$. and $ways[1][t] = 0(t\%D[1] \neq 0)$. And we calculate each $ways[i][j](i > 1)$ according to the DP equation. Finally we can calculate $ways[n][amount]$ which is the max value we want to solve.

The Pseudo Code is shown as below:

```
FIND-TOTAL-COIN-CHANGE-WAYS(D, n, amount)
 1  if n = 0 and amount = 0 return 0;
 2  if amount = 0 return 1;
 3  Allocate ways as an n * amount matrix
 4  for i = 0 to n
 5      ways[i][0] = 0
 6  for i = 0 to amount
 7      ways[0][i] = 0
 8  for j = 1 to amount
 9      if j%D[1] ≠ 0
10          ways[1][j] = 0
11      else
12          ways[1][j] = 1
13  for i = 2 to n
14      for j = 1 to amount
15          for k = 0 to ⌊j/D[i]⌋
16              ways[i][j]+ = ways[i − 1][j − k * D[i]]
17  return v[n]
```

2.3 Correctness Proving

In this problem, given the rest value j, number of ith class coin k must in range $[0, \lfloor j/D[i] \rfloor]$. Once we choose k, we only left $j - D[i] * k$ value to be make up, and we only can choose from the first $i - 1$ classes of coins, i.e. $ways[i-1][j - k * D[i]]$.

By enumerate k, we can calculate the value of $ways[i][j]$, which is a sum of all the enumerations. And apparently $ways[n][amount]$ is the solution to the problem. So the correctness is proven.

2.4 Algorithm Analysis

For time complexity, the biggest time cost is the 3-loop in algorithm, which can be calculated out by multiple all the loop iterators as below:

$$T(n) = O(amount * n^2)$$

For space complexity, according to the Pseudo-code, we only need one data structure which is a $n * amount$-element matrix. So the space complexity is:

$$S(n) = O(amount * n)$$

3 The smallest difference

You are given a collection of stones, each stone has a positive integer weight. You need to divide these stones into two piles whose weights are as balanced as possible. Please give the smallest difference between the weights of the two piles.

3.1 Optimal Substructure and DP equation

We assume that there is n stones, and its weight we can recorded in the array $W = [W[1], W[2], .., W[n]]$.

And what we need to do is divide those stones into two piles, s.t. the difference of two piles are minimum.

We assume that the divided piles weight are M_1 and M_2. Without loss of generality, we assum that $M_1 \leq M_2$, and our optimization goal is:

$$\min\{d\}, d = M_2 - M_1$$

We denote that total weight of n stone as w. Then $w = \sum_{i=0}^{n} W[i]$. Then we have:

$$w = M_1 + M_2$$

Apparently, according to $M_1 \leq M_2$, we have $M_2 \leq \frac{w}{2}$. And we substract w by d, we can have:

$$d = w - 2M_2$$

As w is a constant, if we want to have the minimum of d, then we need to find the maximum of M_2. And we have the constrain that $M_2 \leq \frac{w}{2}$, so we transfer original problem into following optimization question:

We have n stones, pick the stones as heavy as possible with constraint that no more than $\frac{w}{2}$.

It is a **0-1** knapsack problem!

Considering the very first step of **0-1** knapsack problem, we have the volume of $\frac{w}{2}$, and to decide whether load with nth item or not:

1. If $W[n] \leq \frac{w}{2}$, and if we choose the nth stone, then we only left $\frac{w}{2} - W[n]$ space and $n-1$ stones.
2. If $W[n] > \frac{w}{2}$, or if we just don't choose the nth stone, then we left $\frac{w}{2}$ space and $n-1$ stones to choose

So we can denote $weight[i][j]$ as the most weight we can handle with a j space knapsack and we have only $W[1,..,i]$ stones to choose. Apparently, it is an Optimal Substructure, and we have the DP equation as follows:

$$weight[i][j] = \begin{cases} 0 & i = 0 or j = 0 \\ weight[i-1][j] & W[i] > j \\ \max\{weight[i-1][j], weight[i-1][j-W[i]] + W[i]\} & 2 \leq i \leq n, 1 \leq j \leq \frac{w}{2} \end{cases}$$

3.2 Algorithm Description

First of all, we need to calculate the limitation of our transformed problem $\frac{w}{2}$, which can be easily finished by a scan to W.

Then we initialize the $weight[i][0]$ and $weight[0][j]$ with 0.

Next, for $weight[i][j](0 < j \leq \frac{w}{2})$, we calculate the value according to DP function.

Finally we get $weight[n][\frac{w}{2}]$, and we take $c = w - 2 * weight[n][\frac{w}{2}]$ as solution to the problem. The Pseudo Code is shown as below:

FIND-SMALLEST-DIFFERENCES(W, n)

```
1  w = 0
2  for i = 0 to n
3        w+ = W[i]
4  Allocate weight as an n * w/2 matrix
5  for i = 0 to n
6        weight[i][0] = 0
7  for j = 0 to w/2
8        weight[0][j] = 0
9  for i = 1 to n
10       for j = 1 to w/2
11             if W[i] > j
12                   weight[i][j] = weight[i-1][j]
13             else
14                   weight[i][j] = max{weight[i-1][j], weight[i-1][j-W[i]] + W[i]}
15 return w - 2 * weight[n][w/2]
```

3.3 Correctness Proving

Considering the pile whose weight is less, we must make its weight closest to the half of total weight.$(w/2)$ The special case shows up when the weight is equal, then the difference is 0. So the problem can be transformed into a classical $0-1$ knapsack problem. And the result can be easily conducted according to their relationships.

3.4 Algorithm Analysis

For time complexity, the biggest time cost is the 2-loop in algorithm, which can be calculated out by multiple all the loop iterators as below:

$$T(n) = O(n * w)$$

where w=$\sum_{i=0}^{n} W[i]$

For space complexity, according to our Pseudo code, we only need one data structure which is a $n * W$-element matrix. So the space complexity is:

$$S(n) = O(n * W)$$

And if we only need to find the smallest difference, we can optimize the space complexity to $O(n)$ by only using two n element arrays. (Details can be found in Prof.Bu's slides). So the best $S'(n)$ using an optimization can be:

$$S'(n) = O(n)$$

Nonetheless utilizing the specific optimization method we can not find which stone should be loaded to get the solution. So the code shows the most traditional version of **0-1** knapsack problem algorithm.