091M4041H - Assignment Two
Dynamic Programing

张 帅
201828018670119
网络空间安全学院, UCAS

December 14, 2018

1 Money Robbing

1.1 Description

A robber is planning to rob houses 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.

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.

What if all houses are arranged in a circle?

1.2 Defining the general form of sub-problems

It's not easy to solve the problem with n houses, let's examine whether it is possible to reduce the problem into smaller sub-problems.

Solution: a subset of houses. let's describe the solving process as a process of multistage decisions. At i-th decision state, we decide whether the house i should be robbed.

Suppose we have already worked out the optimal solution. Consider the first decision, i.e. wheather the optimal solution contains house n or not (here we assume an order of houses and consider the houses from end to beginning). This decision has two options:

- ***ROB :*** since we cannot rob the neighboring house, then we should calculate optimal rob solution for region $1 \dots n-2$.
- ***ABANDON :*** it suffices to rob houses as "rich" as possible from region $1 \dots n-1$.

In each cases, the origin problem is reduced into smaller sub-problems.

If all houses are arranged in a circle, the problem is similar to that above. Due to the first house and the last house are adjacent, we cannot rob them at the same time. So we can divide this problem into two sub-problems, the one's region is [1,n-1](including the first house, but not include the last one), the other's region is [2,n](including the last house, but not include the first one). Then we calculate the results of the sub-problems and return the larger result.

1.3 Optimal Sub-structure Property

Summarizing these two cases, we can set the general form of sub-problems as: to rob houses as "rich" as possible from region $1 \dots i$. Denote the optimal solutions value as $OPT[i]$.

Then we can prove the optimal sub-structure property:

$$OPT[n] = max \begin{cases} OPT[n-2] + v_n & (a) \ \ if \ house_n \ was \ robbed \\ OPT[n-1] & (b) \ \ if \ house_n \ wasn't \ robbed \end{cases}$$

1.4 Pseudo-Code

PROBLEM 1 house robber with or without circle

INPUT: Given a list of non-negative integers representing the amount of money of each house, V. The house number, n.

OUTPUT: Determine the maximum amount of money you can rob tonight without alerting the police.

```
1: function ROB_WITHOUT_CIRCLE(V, n)
2:     Create a 1 * (n + 1) array OPT
3:     Initialize OPT with 0
4:     for i = 1 → n do
5:         OPT[i] ← MAX(OPT[i], OPT[i − 1])
6:         if i − 2 >= 0 then
7:             OPT[i] ← MAX(OPT[i], OPT[i − 2] + V_i)
8:         else
9:             OPT[i] ← MAX(OPT[i], V_i)
10:        end if
11:    end for
12:    return OPT[n]
13: end function
14:
15: function ROB_WITH_CIRCLE(V, n)
16:    Opt_Without_1 ← ROB_WITHOUT_CIRCLE(V[1, n − 1], n − 1)
17:    Opt_Without_N ← ROB_WITHOUT_CIRCLE(V[2, n], n − 1)
18:    return MAX( Opt_Without_1, Opt_Without_N)
19: end function
```

1.5 Correctness Proof

1.5.1 House robber without Circle

Consider the first case: house n was robbed, so why the optimal solution was define as $(a)OPT[n-2] + v_n$?

Suppose that there are three houses between house A and house B (define A as house $n-4$, while define B as house n). If we define the optimal solution as $OPT[n] = OPT[n-4] + v_n$, then there must be another robbery scheme to make the solution better, $OPT^{'}[n] = OPT[n-4] + v_n + v_{n-2}$. Obviously, $OPT^{'}[n]$ is bigger than $OPT[n]$. Of course, if there are more houses that was not be robbed between house A and B, we can always find an example that is better than the optimal solution we define.

But what if there are two houses between A and B? Why we define Why do we not define the optimal solution is to take the maximum from these three values($OPT[n-1], OPT[n-2]+v_n, OPT[n-3]+v_n$)? let's prove it: for region $1 \dots n$,

$$OPT[n] = max\{OPT[n-1], OPT[n-2]+v_n\} \tag{1}$$

Expand $OPT[n-2]$,

$$OPT[n-2] = max\{OPT[n-3], OPT[n-4]+v_{n-2}\} \tag{2}$$

Substituting equation (2) for equation (1), we can get the following equation:

$$OPT[n] = max\{OPT[n-1], OPT[n-3]+v_n, OPT[n-4]+v_{n-2}+v_n\} \tag{3}$$

Obviously, the third value ($OPT[n-3]+v_n$) is included in the second value ($OPT[n-2]+v_n$).

Then, let's proof the correctness of this algorithm. Suppose for $house_i$, there is another rob solution $OPT^{'}[i]$ better than $OPT[i]$, Then the $OPT^{'}[i]$ may lead to a new solution with higher value than $OPT[i+1]$(if $house_{i+1}$ isn't robbed) and $OPT[i+2]$(if $house_{i+2}$ is robbed): a contradiction for the definition of optimal function.

1.5.2 House robber with Circle

The problem is been divided into two sub-problems, we can seen that it just the above problem without difference, so it can't influent the correctness, just one more comparison operation.

Then why we divide them into two sub-problems(region $[1, n-1]$ and region $[2, n]$)? We mainly want to guarantee that the first house and the last house won ' t be robbed at the same time.

1.6 Complexity Analysis

1.6.1 House robber without Circle

- **Time Complexity:**
  Each iteration only cost several operations.

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

- **Space Complexity:**
  We use a 1*n array to store the optimal solution of the subproblem. so the space complexity is $O(n)$.

1.6.2 House robber with Circle

- **Time Complexity:**

$$T(n) = 2 * c * (n-1) + c = O(n)$$

- **Space Complexity:**
  We use two 1*(n) arrays to store the optimal solution of the subproblem. so the space complexity is $O(n)$.

2 Decoding

2.1 Description

A message containing letters from A-Z is being encoded to numbers using the following mapping:

$$A : 1$$
$$B : 2$$
$$...$$
$$Z : 26$$

Given an encoded message containing digits, determine the total number of ways to decode it.

For example, given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12). The number of ways decoding "12" is 2.

2.2 Defining the general form of sub-problems

It's not easy to solve the problem with n which is the length of given non-empty string S containing only digits, let's examine whether it is possible to reduce the problem into smaller sub-problems.

Firstly, we define S_i as the i-th digit, and S_iS_{i+1} as the combination of the i-th and the $i+1$-th digit.

Solution: a subset of string. let's describe the solving process as a process of multistage decisions. At i decision state, we calculate the decoding results starting with S_i or S_iS_{i+1} respectively.

Suppose we have already worked out the optimal solution. Consider the first decision, i.e. due to S_i and S_iS_{i+1} has mutiple situation, the decision has three options.

- $\boldsymbol{S_i = 0}$ **:** Since no encoding starts with 0, so the number of possible encoding is zero.
- $\boldsymbol{S_iS_{i+1} \in [10, 26]}$ **:** We use S_iS_{i+1} as the beginning of encoding when S_iS_{i+1} is between 10 and 26, Because the range we can encode is $[1, 26]$, Obviously, [10,26] is included in it.Then we just to calculate the optimal decode solution for region $i+2 \dots n$.
- $\boldsymbol{S_i \in [1, 9]}$ **and** $\boldsymbol{S_iS_{i+1} \notin [10, 26]}$ **:** We use S_i as the beginning of encoding when S_i is between 1 and 9, Because the range we can encode is $[1, 26]$, Obviously, $[1, 9]$ is included in it. Then we just to calculate the optimal decode solution for region $i+1 \dots n$.

At last, we combine the three options, and the optimal decode solution for region $i \dots n$ will be got.

In each cases, the origin problem is reduced into smaller sub-problems.

2.3 Optimal Sub-structure Property

Summarizing these two cases, we can set the general form of sub-problems as: to rob houses as "rich" as possible from region $i \dots n$. Denote the optimal solutions value as $OPT([i])$.

Then we can prove the optimal sub-structure property:

$$OPT([i]) = \begin{cases} 0 & if\ S_i = 0 \\ OPT([i+1]) + OPT([i+2]) & if\ S_iS_{i+1} \in [10, 26] \\ OPT([i+1]) & if\ S_i \in [1, 9] \textbf{ and } S_iS_{i+1} \notin [10, 26] \end{cases}$$

2.4 Pseudo-Code

PROBLEM 2 house robber with or without circle

INPUT: Given a non-empty string containing only digits, S. The length of S, n

OUTPUT: Determine the total number of ways to decode it.

```
1: function DECODE(S, n)
2:     Create a 1 * (n + 2) array OPT
3:     Initialize DP with 0
4:     OPT[n + 1] ← 1
5:     OPT[n] ← S_n == 0?0 : 1
6:     for i = n − 1 → 1 do
7:         if S_i == 0 then
8:             continue
9:         else if S_i S_{i+1} ∈ [10, 26] then
10:            OPT[i] ← OPT[i − 1] + DP[i + 2]
11:        else
12:            OPT[i] ← OPT[i − 1]
13:        end if
14:    end for
15:    return OPT[1]
16: end function
17:
```

2.5 Correctness

As described in 2.2, if $S_i = 0$, due to there is no letter's encoding start with 0, so the decoding ways for region $[i, n]$ is zero.

Suppose for the region $[i, n]$, we have a optimal solution $OPT[i]$, if we have another decoding way $OPT^{'}[i]$ better than $OPT[i]$, then the decoding way may $OPT^{'}[i]$ lead to a new solution with more decoding ways than $OPT[i-1]$ {if $S_{i-1} \in [1, 9]$ and $S_{i-1}S_i \notin [10, 26]$} and $OPT[i-2]${if $S_{i-2}S_{i-1} \in [10, 26]$}: a controdiction.

2.6 Complexity Analysis

- **Time Complexity:**
  Each iteration only has several operations.

$$\begin{aligned} T(n) &= c * n \\ &= O(n) \end{aligned}$$

- **Space Complexity:**
  We use a 1*(n+2) array to store the optimal solution of the subproblem. so the space complexity is $O(n)$.

3 Maximum Profit of Transactions

3.1 Description

You have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm and implement it to find the maximum profit. You may complete at most two transactions.

Note : You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again), and one transaction includes buying and selling.

3.2 Defining the general form of sub-problems

- ***General Form:*** Firstly, let's consider the general form of the problem—assume that we can complete at most $\boldsymbol{k}$ transactions with an array P for which the P_i is the price of a given stock on day i, $i \in [1, n]$.

It's not easy to solve the problem with n which is the number of a given price sequense, let's examine whether it is possible to reduce the problem into smaller sub-problems.

Solution: a sub-sequence of the price sequence. Let ' s describe the solving process as a process of multistage decisions; at each decision stage, At the i-th decision stage, we decide whether the i-th day should be involved in the transaction (*NOTE:* if the stock is sold at i-th day, we think the day is involved in the transaction).

Suppose we have already worked out the optimal solution. Consider the first decision, i.e. whether the optimal solution contains item n or not (here we assume an order of the items and consider the items from end to beginning). This decision has two options:

- ***SELL :*** Then it suffices to make profit as more as possible from day 1 to day n with at most k transactions. And day n take part in one of these transactions(the stock was sold on day n).
- ***ABANDON :*** Otherwise, we should make profit as more as possible from day 1 to day $n-1$ with at most k transactions.

In both cases, the original problem is reduced into smaller sub-problems.

3.3 Optimal Sub-structure Property

Summarizing these two cases, we can set the general form of sub-problems as: to make profit as more as possible from day 1 to day n with at most k transactions. Denote the optimal solution value as $OPT(n, k)$.

Then we can prove the optimal sub-structure property:

$$OPT(n,k) = max \begin{cases} OPT(n-1,k) \\ max_{1 \le i < j \le n-1}\{OPT(i,k-1) + P_n - P_j\} \end{cases}$$

3.4 Pseudo-Code

Here, we give two pseudo-codes that implement the same functionality, but with different complexity, one is the general form and the other is the form of state compression.

The following is the general form:

PROBLEM 3 Maximum Profit of Transactions

INPUT: The price sequence of stock, P. The number of days, n. The maximum number of transaction, k.

OUTPUT: Determine the maximum profit of transactions.

```
1: function MaxProfit(P, n, k)
2:     Create a (n + 1) * (k + 1) array OPT
3:     Initialize OPT with 0
4:     for i = 1 → n do
5:         for l = 1 → k do
6:             for j = 1 → i − 2 do
7:                 OPT[i, l] ← max{OPT[i − 1, l], OPT[j, l − 1] + P[i] − P[j + 1]}
8:             end for
9:         end for
10:    end for
11:    return OPT[n, k]
12: end function
```

The following algorithm uses state compression techniques. The array PRO record the intermediate results. Assume today is the i-th day, then we define $OPT[k]$ as the maximum profit that we make with k transaction on the first $i-1$ days. So, $OPT[k]-P[i]$ represent the rest of profit after buying stocks, then we assign The opposite of $OPT[k]-P[i]$ to $PRO[k]$. In the next few days, if we want to sell the stock, we can get the final profit by subtracting the intermediate result($PRO[k]$) from the day's stock price($p[j]\{j \in [i+1, n]\}$).

PROBLEM 4 Maximum Profit of Transactions

INPUT: The price sequence of stock, P. The number of days, n. The maximum number of transaction, k.

OUTPUT: Determine the maximum profit of transactions.

```
1: function MaxProfitCompr(P, n, k)
2:     Create two 1 * (k + 1) arrays OPT, PRO
3:     Initialize OPT with 0
4:     Initialize PRO with P[1]
5:     for i = 1 → n do
6:         for l = 1 → k do
7:             PRO[l] ← min{PRO[l], P[i] − OPT[l − 1]}
8:             OPT[l] ← max{OPT[l], P[i] − PRO[l]}
9:         end for
10:    end for
11:    return OPT[k]
12: end function
```

3.5 Correctness

Suppose for day i,there is another transaction strategy $OPT'[i,k]$ better than $OPT[i,k]$, it may lead to a new transaction way with higher profit than $OPT[i+1,k]$: a contradiction for the definition of optimal function.

3.6 Complexity Analysis

3.6.1 General Form

- **Time Complexity:**

$$\begin{aligned} T(n) &= c * k * (1 + 2 + 3 + \cdots + n - 2) \\ &= O(kn^2) \end{aligned}$$

- **Space Complexity:**

  We use a $(n + 1) * (k + 1)$ array to store the optimal solution of the subproblem. so the space complexity is $O(kn)$.

3.6.2 DP with State Compression

- **Time Complexity:**

$$\begin{aligned} T(n) &= c * n * k \\ &= O(kn) \end{aligned}$$

- **Space Complexity:**

  We use two $1 * (k + 1)$ arrays to store the optimal solution of the problem and the intermediate results. so the space complexity is $O(k)$.

3.6.3 In This Problem($k = 2$)

- ***General Form:***

  Time Complexity: $O(n^2)$

  Space Complexity: $O(n)$

- ***State Compression:***

  Time Complexity: $O(n)$

  Space Complexity: $O(1)$