Algorithm Homework 2

Jingwei Zhang 201528013229095

2015-10-13

1 Problem 1

1.1 Sequence

1.1.1 Optimal Substructure

Suppose the houses are placed in a line from left to right labeled with integer from 1 to n, each storing money $h_i(i = 1, \ldots, n)$. The optimal substructure is the maximum amount of money m_i the the robber can get from house 1 to $i([1, i])$, the DP equation is:

$$m_i = \begin{cases} 0\ , & \text{if } i = 0 \\ h_i\ , & \text{if } i = 1 \\ max(m_{i-1}, h_i + m_{i-2})\ , & \text{otherwise} \end{cases}$$

The answer to this problem is m_n.

1.1.2 Algorithm

```
MAXIMIM-ROBBED-MONEY-SEQUENCE(H)
n = H.length
m[0] = 0
m[1] = H[1]
for i = 2 to n
    m[i] = max(m[i − 1], H[i] + m[i − 2])
return m[n]
```

1.1.3 Correctness

Proof. $m[i]$ maintains the maximum amount of money the robber can get in house $[1, i]$. For $i = 0, 1$, the correctness of $m[i]$ is obvious. Suppose $m[i]$ is correct $\forall i = 1, \ldots, k - 1$. For $i = k$:

If the robber steals house k, he will get $H[k]$ in it and have not steal house $k-1$; before house $k-1$ he gets at most $m[k - 2]$, according to the optimal structure, $m[k] = H[i] + m[k - 2]$.

The correctness of this optimal structure: Suppose we have a smaller $m'[k-2](< m[k-2])$(not the optimal solution of sub-problem with size $k - 2$) that generates the optimal solution $m'[k](> m[k])$ of larger problem with size k. Then we substitute its previous $k-2$ part with our optimal solution for sub-problem since house $k-1$ is not stolen, we get a new solution $m'[k] - m'[k - 2] + m[k - 2]$ for larger problem, which is larger than $m'[k]$, contradictory.

If he does not steal house k, no constrain exits on previous houses. Thus, $m[i] = m[i-1]$, according to the optimal structure.

The correctness of this optimal structure: Suppose we have a smaller $m'[k-1](< m[k-1])$(not the optimal solution of sub-problem with size $k - 1$) that generates the optimal solution $m'[k](> m[k])$ of larger problem with size k. Then we substitute its previous $k - 1$ part with our optimal solution for sub-problem since house k is not stolen, we get a new solution $m'[k] - m'[k - 1] + m[k - 1]$ for larger problem, which is larger than $m'[k]$, contradictory.

Then, picking the maximum value of these two(steal or not) gets maximum amount of money the robber can get in house $[1, k]$, which means $m[i]$ is correct for $i = k$ and obviously when $i > n$ MAXIMIM-ROBBED-MONEY-SEQUENCE stops. Thus the correctness of this algorithm is proven. ■

1.1.4 Complexity

The size of this problem is the number of houses n. Thus, sequence m has $n+1$ elements with $O(1)$ computing each. Thus the total time complexity is $O(n)$ and space complexity is $O(n)$ for storing array m.

1.2 Circle

1.2.1 Algorithm

Suppose the houses are placed in a circle, and we arbitrary label one with integer 1, then label others with integer from 2 to n, clockwise, each storing money $h_i (i = 1, \dots, n)$. Then, we enumerate all possible conditions of house 1, stolen or not stolen, then, the problem left is a sequence problem we have solved in previous part.

MAXIMIM-ROBBED-MONEY-CIRECLE(H)

$n = H.length$
// If house 1 is stolen, house 2 and house n must not be stolen.
$Hs = H[3, \dots, n-1] + H[1]$
$ms =$ MAXIMIM-ROBBED-MONEY-SEQUENCE(Hs)
// If house 1 is not stolen.
$Hns = H[2, \dots, n]$
$mns =$ MAXIMIM-ROBBED-MONEY-SEQUENCE(Hns)
return $max(ms, mns)$

1.2.2 Correctness

Proof. If we enumerate two possible states of house 1, the left part ($[3, \dots, n-1]$ for stolen and $[2, n]$ for not stolen) have no connection directly between the head and tail house, thus, they are sequence problem. The correctness of sequence problem has been proven in previous section. ∎

1.2.3 Complexity

We call function MAXIMIM-ROBBED-MONEY-SEQUENCE twice, both with size $O(n)$. This function costs $O(n)$ of time and $O(n)$ of space. Thus, the total time complexity is $O(n)$, total space complexity is $O(n)$.

2 Problem 2

2.1 Optimal Substructure

The optimal substructure is the minimum path sum $s_{i,j}$ from current place(row i, column j, $j \leq i$) to bottom, the DP equation is ($a_{i,j}$ denotes the number in row i , column j):

$$s_{i,j} = \begin{cases} a_{i,j} \ , & \text{if row i is the bottom row} \\ a_{ij} + min(s_{i+1,j}, s_{i+1,j+1}) \ , & \text{otherwise} \end{cases}$$

The answer to this problem is $s_{1,1}$.

2.2 Algorithm

Pseudo-code: A is the matrix storing the number. r is the number of rows(columns) in A. S is the matrix storing the minimum path sum $S[i][j]$ from current place(row i, column j, $j \leq i$) to bottom.

MINIMUM-PATH-SUM(A, r)

for $j = 1$ **to** r
    $S[r][j] = A[r][j]$
for $i = n-1$ **to** 1
    for $j = 1$ **to** i
        $S[i][j] = A[i][j] + min(S[i+1][j], S[i+1][j+1])$
return $S[1][1]$

2.3 Correctness

Proof of Optimal Substructure: Suppose there exists a smaller path sum s_l'(what we get is s_l in MINIMIM-PATH-SUM and $s_l' > s_l$) in an arbitrary sub-problem(the min sum from a not-top place to bottom) which leads to the minimum global path sum s_g'(what we get is s_g in MINIMUM-PATH-SUM and $s_g' < s_g$). Due to the path from top to this place above and the path from this place to bottom are independent, we can always substitute s_l' part with s_l part, that leads a larger global path sum $s_g'' = s_g' + s_l - s_l' < s_g'$. However, this contradicts to the assumption that s_g' is the minimum global path sum. ■

2.4 Complexity

Let the size of this Problem be the total numbers in this triangle. S matrix have totally $O(n^2)$ numbers with $O(1)$ for computing each. Thus, the total time complexity is $O(n)$ and space complexity is $O(n^2)$ for the storage of S.

3 Problem 5

3.1 Optimal Substructure

The optimal substructure is the number of ways (w_j) to decode the sequence $S[1, i]$ (suppose the original sequence is $S[1, n]$), the DP equation is:

$$w_j = \begin{cases} 0 \ , & \text{if } j = 0 \\ 1 \ , & \text{if } j = 1 \\ w_{j-1} \ , & \text{if } j \geq 2 \text{ and } 10 * S[j-1] + S[j] > 26 \\ w_{j-1} + w_{j-2} \ , & \text{if } j \geq 2 \text{ and } 10 * S[j-1] + S[j] \leq 26 \end{cases}$$

The answer to this problem is w_n.

3.2 Algorithm

Pseudo-code: S represents the message containing n digits. w stores the number of ways decoding the prefix of message sequence.

NUMBER-OF-WAYS-DECODING-MESSAGE(S)

```
n = S.length
w[0] = 0
w[1] = 1
for j = 2 to n
    if 10 * S[j-1] + S[j] > 26
        w[j] = w[j-1]
    else w[j] = w[j-1] + w[j-2]
return w[n]
```

3.3 Correctness

Proof. $w[i]$ maintains the number of ways to decode the sequence $S[1, i]$. For $i = 0, 1$, the correctness of $w[i]$ is obvious. Suppose $w[i]$ is correct $\forall i = 1, \ldots, k-1$. For $i = k$:

If $10 * S[k-1] + S[k] \leq 26$, which means $S[j-1]S[j]$ can be decoded as a single character. In this case the number of ways is $w[k-2]$ according to the optimal structure. Also, $S[k]$ can be decoded as a single character. In this case, the number is $w[k-1]$. Thus, $m[k] = w[k-1] + w[k-2]$.

The correctness of optimal substructure: Suppose there exists a larger number of ways decoding the prefix sequence $S[1, k-1]$ with number w_{k-1}'(what we get is w_{k-1} in NUMBER-OF-WAYS-DECODING-MESSAGE and $w_{k-1}' \leq w_{k-1}$) which leads to the optimal global solution w_k'(what we get is w_k in NUMBER-OF-WAYS-DECODING-MESSAGE and $w_k' \geq w_k$). If we substitute w_{k-1}' with w_k' we will get a larger optimal global solution. Contradictory. For sub-problem $k-2$, it is similar.

If $10 * S[k-1] + S[k] > 26$, which means $S[j-1]S[j]$ can not be decoded as a single character. The only possibility for $S[j]$ is decoding it singly to a character. This leads to $w[i] = w[i-1]$ according to optimal structure mentioned above.

Thus, $w[i]$ is correct for $i = k$. Moreover, this procedure will stop after $w[n]$. Thus, the correctness of this algorithm is proven ■

3.4 Complexity

The size of this problem is the length of message sequence, n, the same as w. For each $w[j]$, computing costs only $O(1)$. So the total time complexity is $O(n)$ and space complexity is $O(n)$ for storing w.

4 Problem 6

4.1 Algorithm

Since there are two transactions(if there is only one, we can sell and buy the stock in a single day within this transaction), we can divide the problem into two independent sub-problems:the max profit p_1 of the first transaction within day $[0, i)$ and max profit p_2 of the second transaction within day $[i, n)$. We can enumerate all possible i and find the max $p_1 + p_2$. This costs $O(n)$.

To find the max profit in first transaction, we compute the max profit ps_i we can get if we sell the stock in day i. The optimal substructure is the minimum price min_i in day $[0, i]$, $min_{i+1} = min(p_i, min_i)$. Then $ps_i = p_i - min_i$, which costs $O(n)$ to enumerate all i in $[0, n)$. Then the max profit pm_i within day $[0, i)$ will be $pm_0 = 0, pm_i = max(pm_{i-1}, ps_i)$, that costs $O(n)$.The second transaction is similar.

From what have mentioned above, the total time complexity is $O(n)$.

4.2 C++ Code

```cpp
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <algorithm>
#include <vector>

using namespace std;

void fill_left(vector<int> &p, const vector<int> &d){
    p.resize(d.size());
    if(p.size() > 0){
        // Firstly, p[i] represents the max profile you can get
        // when you sell the stock in day i

        // price_min means the minimum price in [0,i]
        // when i iterates in array d
        int price_min = d[0];
        p[0] = 0;

        for(int i = 1; i < d.size();i++){
            price_min = min(price_min, d[i]);
            p[i] = d[i] - price_min;
        }

        // Now compute the max profile you can get during [0,i]
```

```
        // Store it in p[i]

        // profile_max maintains the max in p[0,i]
        int profile_max = 0;
        for(int i = 0;i < p.size();i++){
            profile_max = max(profile_max, p[i]);
            p[i] = profile_max;
        }
    }
}

void fill_right(vector<int> &p, const vector<int> &d){
    p.resize(d.size());
    if(p.size() > 0){
        // Firstly, p[i] represents the max profile you can get
        // if you buy the stock in day i

        // price_min means the maximum price in [0,i]
        // when i iterates reversely in array d
        int price_max = p[p.size()-1];
        p[p.size()-1] = 0;

        for(int i = p.size()-1;i >= 0;i--){
            price_max = max(price_max, d[i]);
            p[i] = price_max - d[i];
        }

        // Now compute the max profile you can get during [i,end]
        // Store it in p[i]
        int profile_max = p[p.size()-1];
        for(int i = p.size()-1;i >= 0;i--){
            profile_max = max(profile_max, p[i]);
            p[i] = profile_max;
        }

    }
}

int main()
{
    freopen("stocks.in","r",stdin);
    //freopen(".out","w",stdout);
    vector<int> d;
    int t;
    while(cin>>t){
        d.push_back(t);
    }

    // pre[i] stores the max profit you get during day [0, i]
    // in a single transaction
    vector<int> left;


    // last[i] stores the max profit you get during day from i to last
    // in a single transaction
    vector<int> right;
```

```
    fill_left(left,d);
    fill_right(right,d);

    int sum_max = 0;
    for(int i = 0;i < left.size();i++){
        sum_max = max(sum_max, left[i] + right[i]);
    }
    cout<<sum_max<<endl;

    return 0;
}
```