Assignment 2
Algorithm Design and Analysis

bitjoy.net

October 23, 2015

I choose problem 1,2,5,6.

1 Money robbing

1.1 Algorithm description

Suppose there are n houses, A[i] is the money stashed in i^{th} house. For each house, we have two choices, rob it or not. Let S[i][j] be the sum of money robbed from first i houses, while the i^{th} house's status is j(1 for being robbed, 0 for not).

If we rob the i^{th} house, the $(i-1)^{th}$ house can't be robbed, so S[i][1]=S[i-1][0]+A[i]. Otherwise, we don't rob the i^{th} house, the $(i-1)^{th}$ house can either be robbed or not, so S[i][0]=max(S[i-1][0],S[i-1][1]).

At last, we choose the maximum of S[n][0] and S[n][1]. The DP equation shows below:

$$\begin{cases} S[i][0] = 0 & \text{if } i = 1 \\ S[i][1] = A[i] & \text{if } i = 1 \\ S[i][0] = max(S[i-1][0], S[i-1][1]) & \text{if } i > 1 \\ S[i][1] = S[i-1][0] + A[i] & \text{if } i > 1 \end{cases}$$

To summarize, we have the following algorithm.

```
MONEY-ROBBING(A)
1  S[1][0]=0; S[1][1]=A[1];
2  for i = 2 to n
3        S[i][0]=max(S[i-1][0],S[i-1][1])
4        S[i][1]=S[i-1][0]+A[i]
5  return max(S[n][0],S[n][1])
```

1.2 Correctness of the algorithm

We can use ***loop invariant*** to prove it.

Initialization: When i=1, there is only one house, if we rob it, money is S[1][1]=A[1], if not, S[1][0]=0, so the maximum amount of money we can rob is max(S[1][0],S[1][1])=A[1].

Maintenance: Given the maximum amount of money robbed from the first i-1 houses, say S[i-1][0] and S[i-1][1]. For the i^{th} house, if we rob it, the $(i-1)^{th}$ house

can't be robbed, so the money robbed is S[n-1][1]+A[i]; if we don't rob it, the $(i-1)^{th}$ house can either be robbed or not, so the money robbed is max(S[i-1][0],S[i-1][1]). The maximum amount of money robbed so far is max(S[i][0],S[i][1]), so the algorithm holds for i.

Termination: When i=n, S[n][0] means the maximum amount of money robbed from n houses and n^{th} house wasn't robbed; S[n][1] means the maximum amount of money robbed from n houses and n^{th} house was robbed. So, the global maximum is max(S[n][0],S[n][1]).

1.3 Complexity of the algorithm

According to the pseudo-code, there is only one ***for*** loop, so the time complexity is O(n).

The size of array S is $n \times 2$, so the space complexity is O(n). As S[i][j] only needs S[i-1][j], only four variables(S[i-1][0],S[i-1][1],S[i][0],S[i][1]) are necessary, so the space complexity can be optimized to O(1).

1.4 What if all houses are arranged in a circle?

If all houses are arranged in a circle, the first and the last house can't be robbed at the same night, so we rob either from 1 to n-1 or from 2 to n.

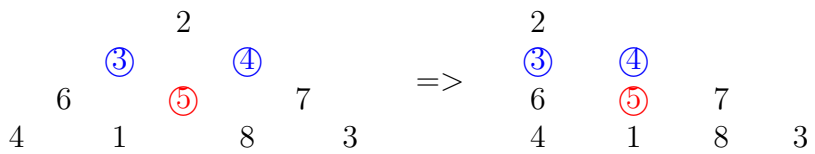
CIRCLE-ROBBING(A)
1 B=A[1,...,n-1]; C=A[2,...,n];
2 max1=MONEY-ROBBING(B)
3 max2=MONEY-ROBBING(C)
4 **return** max(max1,max2)

The complexity is the same as MONEY-ROBBING's.

2 Minimum path sum

2.1 Algorithm description

In order to simplify the problem, we change the triangle's shape like this:



As we can see, each number can only be reached from its left or right parent. For example, number 5 can only be reached from number 3 or 4.

If we change the shape to the right picture, let A[i][j] be the number on i^{th} row and j^{th} column, S[i][j] be the minimum path sum from top to number A[i][j].

If we reach A[i][j] from its left parent, S[i][j]=S[i-1][j-1]+A[i][j]; if we reach A[i][j] from its right parent, S[i][j]=S[i-1][j]+A[i][j]. So the minimum path sum from top to A[i][j] should be $min(S[i-1][j-1], S[i-1][j]) + A[i][j]$.

At last, we choose the minimum path sum in the bottom row. The DP equation shows below:

$$S[i][j] = min(S[i-1][j-1], S[i-1][j]) + A[i][j]$$

Suppose the number triangle has r rows, i^{th} row has i numbers. To summarize, we have the following algorithm.

```
MIN-PATH-SUM(A)
 1  S[1][1] = A[1][1]
 2  for i = 2 to r
 3       for j = 1 to i
 4            S[i][j] = +∞
 5            if A[i][j] has left parent
 6                 S[i][j] = min(S[i][j], S[i-1][j-1] + A[i][j])
 7            if A[i][j] has right parent
 8                 S[i][j]=min(S[i][j], S[i-1][j] + A[i][j])
 9  ans = +∞
10  for j = 1 to r
11       ans = min(ans, S[r][j])
12  return ans
```

2.2 Correctness of the algorithm

We can use ***loop invariant*** to prove it.

Initialization: When i=1, there is only one number, so the minimum path sum is itself A[1][1].

Maintenance: Given the minimum path sum in $(i-1)^{th}$ row, for A[i][j] in i^{th} row, we can only reach it from its left or right parent, so the minimum path sum from top to A[i][j] is min(S[i-1][j-1],S[i-1][j])+A[i][j]. Algorithm holds for i^{th} row.

Termination: When we reach the bottom row, say i=r, we have work out all minimum path sum from top to numbers in bottom row, so the global minimum path sum is $min_{j=1}^{j=r} S[r][j]$.

2.3 Complexity of the algorithm

According to the pseudo-code, from line 2 to 8, we scan all numbers once, from line 10 to 11, we scan the bottom row once. If the size of triangle is n, then the time complexity is O(n).

The size of array S equals to the size of triangle, so the space complexity is O(n).

5 Decoding

5.1 Algorithm description

Let array A be the encoded message, each digit A[i] can either be decoded alone or be decoded together with its former digit A[i-1] except that A[i] is zero or A[i-1]A[i] > 26. Let B[i] be the number of ways to decode A[1,...,i], so there are 2 cases:

- if $A[i] \in [1, 9]$, A[i] can be decoded alone, B[i] = B[i] + B[i-1];
- if $A[i-1]A[i] \in [10, 26]$, A[i-1] and A[i] can be decoded together, B[i] = B[i] + B[i-2].

Note that '01' can't be decoded to '1'. The DP equation shows below:

$$\begin{cases} B[i] = B[i] + B[i-1] & \text{if } A[i] \in [1,9] \\ B[i] = B[i] + B[i-2] & \text{if } A[i-1][A[i] \in [10,26] \end{cases}$$

To summarize, we have the following algorithm.

```
DECODING(A)
1  B[0] = B[1] = 1
2  for i = 2 to n
3        B[i] = 0
4        if A[i] ∈ [1,9]
5              B[i] = B[i] + B[i-1]
6        if A[i-1]A[i] ∈ [10,26]
7              B[i] = B[i] + B[i-2]
8  return B[n]
```

5.2 Correctness of the algorithm

We can use ***loop invariant*** to prove it.

Initialization: When i=1, as A is a valid encoded message, $A[1] \neq 0$, so there is only one decoding way, so B[1]=1.

Maintenance: Given the number of decoding ways in A[1,...,i-2] and A[1,...,i-1], say B[i-2] and B[i-1]. As for A[i], if $A[i] \in [1,9]$, A[i] can be decoded alone, B[i] = B[i-1]; if $A[i-1]A[i] \in [10,26]$, A[i-1]A[i] can be decoded together, B[i] = B[i-2]. So we get the number of decoding ways in A[1,...,i], algorithm holds for i.

Termination: B[n] is the number of decoding ways in A.

5.3 Complexity of the algorithm

According to the pseudo-code, we scan the array A once, so the time complexity is O(n).

The size of array B equals to the size of array A, so the space complexity is O(n). As B[i] only needs B[i-2] and B[i-1], so the space complexity can be optimized to O(1).

6 Maximum profit of transactions

6.1 Problem description

Let array A store the daily price of the stock, we can complete at most two transactions to get the maximum profit.

Note, two transactions can't overlap, you must buy first, sell first and buy second, sell second.

Let B[i] be the maximum profit of one transaction among A[1,...,i], C[i] be the maximum profit of another transaction among A[i,...,n], the maximum profit of two transactions should be $\max_{i=1}^{n}\{B[i] + C[i]\}$.

So, here is the problem, how to work out B[i] and C[i].

As for B[i], we scan array A from 1 to n, let variable $min\_price\_so\_far$ be the minimum price so far and $max\_profit\_so\_far$ be the maximum profit of one transaction so far. Obviously, we have $max\_profit\_so\_far = max\{max\_profit\_so\_far, A[i] - min\_price\_so\_far\}$. $max\_profit\_so\_far$ is exactly B[i].

As for C[i], we scan array A from n to 1, similarly, we have $C[i] = max\{max\_profit\_so\_far, max\_price\_so\_far - A[i]\}$.

To summarize, we have the following algorithm.

FIND-MAX-PROFIT(A)

```
1  min_price_so_far = A[1]
2  B[1] = 0
3  for i = 2 to n
4       min_price_so_far = min{min_price_so_far, A[i]}
5       B[i] = max{B[i − 1], A[i] − min_price_so_far}
6  max_price_so_far = A[n]
7  C[n] = 0
8  for i = n-1 downto 1
9       max_price_so_far = max{max_price_so_far, A[i]}
10      C[i] = max{C[i + 1], max_price_so_far − A[i]}
11 global_max_profit = 0
12 for i = 1 to n
13      global_max_profit = max{global_max_profit, B[i] + C[i]}
14 return global_max_profit
```

6.2 Implementation in C++

```cpp
#include<iostream>
#include<fstream>
#include<algorithm>
using namespace std;

const int MAXN = 100; // max records
int A[MAXN]; // daily price of the stock
int B[MAXN]; // max profit among A[1,...,i]
int C[MAXN]; // max profit among A[i,...,n]

int main()
{
    freopen("input.txt", "r", stdin);

    int n = 1;
    while (~scanf("%d", &A[n]))
        n++;

    int min_price_so_far = A[1];
    B[1] = 0;
    for (int i = 2; i < n; i++)
    {
        min_price_so_far = min(min_price_so_far, A[i]);
        B[i] = max(B[i - 1], A[i] - min_price_so_far);
    }

    int max_price_so_far = A[n - 1];
    C[n - 1] = 0;
```

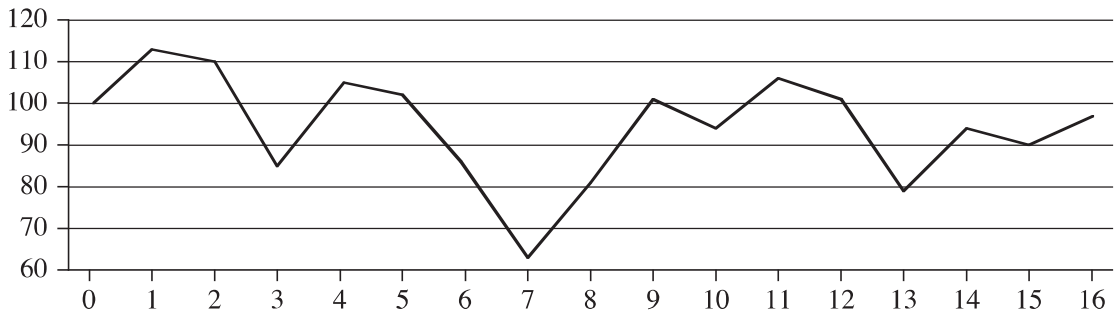
```
    for (int i = n - 2; i >= 1; i--)
    {
        max_price_so_far = max(max_price_so_far, A[i]);
        C[i] = max(C[i + 1], max_price_so_far - A[i]);
    }

    int global_max_profit = 0;
    for (int i = 1; i < n; i++)
        global_max_profit = max(global_max_profit, B[i] + C[i]);
    printf("%d\n", global_max_profit);

    return 0;
}
```

Sample input:
113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97

Sample output:
63



The figure of the sample data is showed above, we buy stock on day 3 and sell it on day 4 and buy it on day 7 and sell it on day 11, earning a profit of (105-85)+(106-63)=63.

According to the pseudo-code, there are three ***for*** loops, so the time complexity is O(n). As we need extra arrays B and C, so the space complexity is O(n) too.