z5135897

Mingkai Ma

1.

(a)

For example, the array is: 10, 10, 10, 11, 8, 2.

If we always pick the largest item with the constraint, we will pick: 11, 10, 2, the result is 23, however there is a better solution if we pick 10, 10, 8, whose result is 28.

(b)

For example: 1, 12, 3, -13, 5,

If we only select odd numbered elements,

We will get 12, -13, the result is obviously not optimal.

If we only select even numbered elements,

We will get 1, 3, 5 the result is also not optimal.

For both above example, we need to select a combination of odd and even

numbered elements. For this example, we need to select 12, 5, 12 is the

second number, 5 is the fifth number.

(c)

Dynamic Programming, implement in C++:

int MaxTotalSum(vector<int>& A)

{

vector<int> res(A.size() + 1);

res[A.size()] = 0;

res[A.size() - 1] = A.back();

for(int i = A.size() – 2; i >= 0; i--)

{

res[i] = max(A[i] + res[i + 2], res[i + 1]);

}

return res[0];

}

2.

We represent each employee with a node in the tree. The root node is CEO.

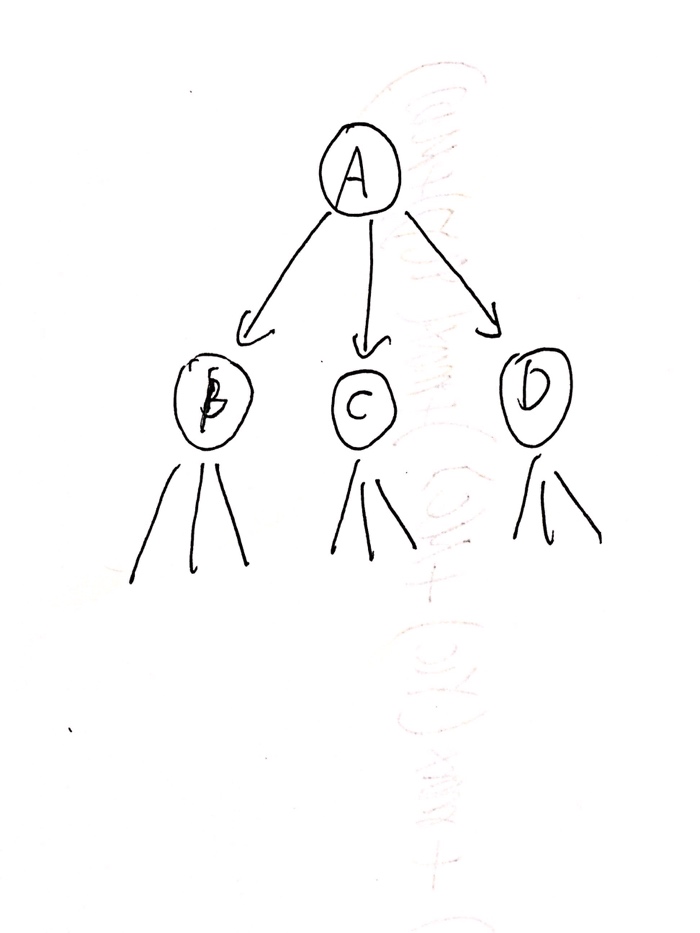
For each node x, let cost(x) denote the cost of inviting this employee. For each

node x in the tree

we compute the values of Y(x) and N(x), which represent the minimum total

cost of hosting the retreat where x is invited or not invited respectively.

For example:



If A is invited to attend the final optimal retreat with minimum total cost, then

B, C, D could be either invited or not. Therefore

Y(A) = cost(A) + min{Y(B), N(B) + min{Y(C), N(C)} + min{Y(D), N(D)}.

If A is not invited to attend the final optimal retreat with minimum total cost, then B, C, D must be invited to attend the final optimal retreat. Therefore

N(A) = Y(B) + Y(C) + Y(D)

We get sub problems, we can compute the sub-problems recursively, or we can use Dynamic Programming to store the computed sub-problems, and whenever we need the results of some sub problems, we will look up it in a data structure.

Pseudo code:

Retreat(u):

Y(u) = cost(u)

N(u) = 0

For all children v of u:

Retreat(v)

Y(u) = Y(u) + min{Y(v), N(v)}

N(u) = N(u) + Y(v)

Once we have computed Y(x) and N(x) for all node x in the tree, we will get the minimum total cost by compare Y(root) and N(root)

If Y(root) > N(root), the minimum total cost is N(root)

Otherwise, the minimum total cost is Y(root).

3.

Dynamic programming.

We use a 2-D matrix to store subproblems results, therefore we don’t need to recursively compute same sub problems.

C++ code:

**int** count(string B, string A)  
{  
 **int** m = A.size();  
 **int** n = B.size();  
  
 **if**(m > n)  
 **return** 0;  
  
 *// here we use the 2D matrix to store sub problem results* **int** matrix[m + 1][n + 1];  
  
 *// If string B's length is 0, there is no subsequence, we set 0* **for**(**int** i = 1; i <= m; i++)  
 matrix[i][0] = 0;  
  
 *// An empty subsequence is subsequence of all sequence, we set 1* **for**(**int** j = 0; j <= n; j++)  
 matrix[0][j] = 1;  
  
  
 *// In buttom up to fill the matrix* **for**(**int** i = 1; i < m + 1; i++)  
 {  
 **for**(**int** j = 1; j < n + 1; j++)  
 {  
 *// If current B[j - 1] does not match with A[i - 1],  
 // then the value of matrix[i][j] is same as the value matrix[i][j - 1]  
 // which means without the B[j - 1] character* **if**(A[i - 1] != B[j - 1])  
 matrix[i][j] = matrix[i][j - 1];  
  
  
 *// If current B[j - 1] does match with A[i - 1]  
 // The value of matrix[i][j] composes of two parts  
 // 1) value of matrix[i - 1][j - 1]  
 // 2) value of matrix[i][j - 1]* **else** matrix[i][j] = matrix[i][j - 1] + matrix[i - 1][j - 1];  
 }  
 }  
  
 *// return the number of different occurrences* **return** matrix[m][n];  
}

4.

(a)

Dynamic Programming

The grid in the below coe in the two-dimensional array A

C++ code:

**int** maxPathSum(vector<vector<**int**>>& grid)  
{  
 **int** row = grid.size();  
 **int** col = grid[0].size();  
  
 **for**(**int** i = 0; i < row; i++)  
 {  
 **for**(**int** j = 0; j < col; j++)  
 {  
 *// start position* **if**(i == 0 && j == 0)  
 ;  
  
 *// no choice, get this position only from left* **else if**(i == 0)  
 grid[i][j] += grid[i][j - 1];  
  
 *// no chiose, get this position only from top* **else if**(j == 0)  
 grid[i][j] += grid[i - 1][j];  
  
 *// choose the max value between top and left direction* **else** grid[i][j] += max(grid[i - 1][j], grid[i][j - 1]);  
 }  
 }

**return** grid[row - 1][col -1];  
}

(b)

Use the matrix to get the path information.

Start from bottom right position, find the path using matrix to top left position,

Using a stack to store the path information

*// 0: down  
// 1: right*stack<**int**> path;  
**while**(i >= 0 && j >= 0)  
{  
  
 **if**(i == 0 && j == 0)  
 **break**;  
  
 **if**(j == 0)  
 {  
 path.push(0);  
 i--;  
 **continue**;  
 }  
  
 **if**(i == 0)  
 {  
 path.push(1);  
 j--;  
 **continue**;  
 }  
  
 **if**(grid[i - 1][j] >= grid[i][j - 1])  
 {  
 path.push(0);  
 i--;  
 }  
  
 **else** {  
 path.push(1);  
 j--;  
 }  
  
  
  
}  
  
  
**while**(!path.empty())  
{  
 **if**(path.top() == 0)  
 cout << **"D"**;  
 **else** cout << **"R"**;  
  
 path.pop();  
}

(c).

Since the input array A is read only,

We need a vector whose size is n to finish the task.

C++ code:

**int** maxPathSum(vector<vector<**int**>>& grid)  
{  
 **int** row = grid.size();  
 **int** col = grid[0].size();  
  
 *// here is the extra space we use* vector<**int**> v(row, grid[0][0]);  
  
 **for**(**int** i = 1; i < row; i++)  
 v[i] = v[i - 1] + grid[i][0];  
  
 **for**(**auto**& i: v)  
 cout << i << **", "**;  
 cout << endl;  
 **for**(**int** j = 1; j < col ; j++)  
 {  
 v[0] += grid[0][j];  
 **for**(**int** i = 1; i < row; i++)  
 v[i] = max(v[i - 1], v[i]) + grid[i][j];  
  
 }  
  
 **return** v[row - 1];  
  
}