中国科学院大学

University of Chinese Academy of Sciences

计算机算法设计与分析
第二次作业

201818013229056 苗天昊

2018 年 10 月 30 日

1 Money robbing

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.

问题分析与证明：先考虑不成环的情况。可以将这N个屋子用一个数组w[N]表示，屋子顺序即为下标，每一个数组元素的值代表该屋子所含财富值。用一个数组OPT[n] 来表示n 个房子可以抢得财富的最大值，其中OPT[i] 表示到第i 个房子能抢到的最大值。先考虑一个最简单情况n=2。两间屋子，不能抢相邻的屋子，则OPT[0] = w[0]，OPT[1] = max{OPT[0], w[1]}。当n=3时，OPT[3] = max{OPT[2], OPT[1] + 3}。

现考虑一般情况，该问题可以抽象为给定一个非负数组w[N]，每个元素的值为权重，求任何不相邻元素的权重和最大值。从最后一个元素考虑，最后一个元素可以选则加或不加。如果抢了最后一个屋子，那么倒数第二个屋子就一定不能抢，因此只能考虑从第一间屋子开始，抢到倒数第三间屋子的财产在加上最后一个屋子的财产，将其作为总财富值。如果不抢最后一间屋子，那么倒数第二间屋子就可抢可不抢，其财富总值就是最后一间屋的总财富值。将这两种情况进行比较可得较大值，将最大情况作为抢劫方案。

参考图1，对于一个从1，2,⋯，i的子问题，其递归表达式OPT[i - 1] = max{OPT[i - 2], OPT[i - 3] + w[i - 1]}

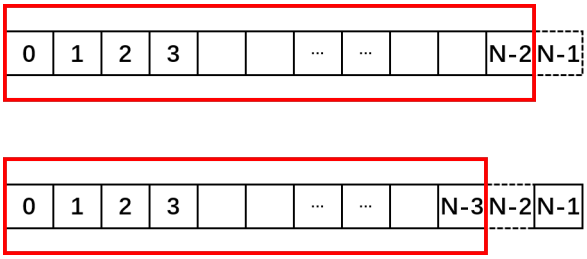

图 1: 直线情况

Dynamic Programming 1 Money robbing

```
MAXMONEY(OPT, n)
OPT[0] = w[0]
if n == 1 then
    return OPT[0]
end if
OPT[1] = max {w[0], w[1]}
if n == 2 then
    return OPT[1]
end if
for i = 0 to n − 1 do
    OPT[i] = max {OPT[i - 1], OPT[i - 2] + w[i]}
end for
return OPT[n - 1]
```

时间复杂度分析：该算法有n个子问题，每个子问题需要常数个时间复杂度，因此总的算法时间复杂度为O(n)。

现考虑成环情况。仍然可以将该街道进行编号，用一个数组w[n]去存储其权重，同样用OPT[n]来存储其抢劫所得财富最大值。如果最后一个屋子不抢，能否抢劫其他屋子不受最后一个屋子的影响，那么其问题就可以等价成n-1个屋子的直线排列情况。如果抢第n 个屋子，则第一个屋子不能抢，也就等价于在直线情况下从第2 个屋子开始，抢到第n个屋子这种情况，并且第n个屋子是一定要抢的。

参考图2，对于一个从1，2,⋯，i的子问题，其递归表达式OPT[i] = max{MAXMONEY(OPT, i - 1), MAXMONEY(OPT_new, i - 3) + w[i - 1]}

该算法用到了前面的直线情况下的函数。调用了两次MAXMONEY，而该函数时间复杂度为O(n)，因此环形情况下的时间复杂度也为O(n)。

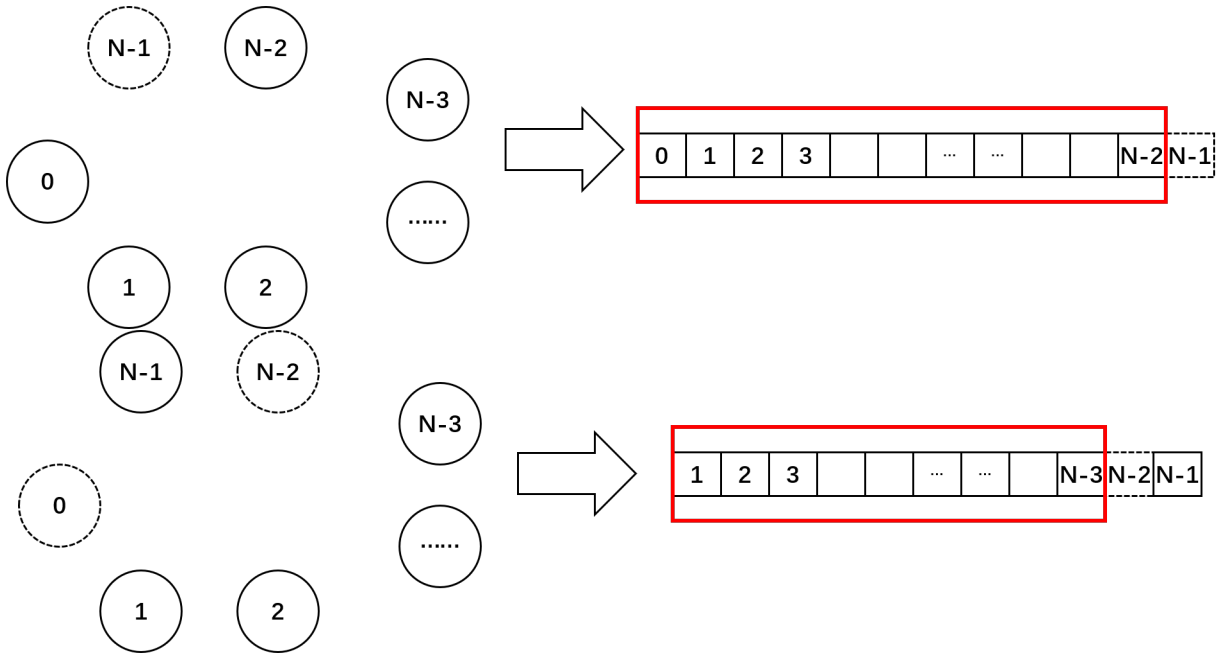

图 2: 环形情况

Dynamic Programming 2 Money robbing

```
MAXMONEY(OPT, n)
OPT[0] = w[0]
if n == 1 then
    return OPT[0]
end if
OPT[1] = max {w[0], w[1]}
if n == 2 then
    return OPT[1]
end if
for i = 0 to n − 1 do
    OPT[i] = max {OPT[i - 1], OPT[i - 2] + w[i]}
end for
return OPT[n - 1]

MAXMONEY_CYCLE(OPT, n)
M1 = MAXMONEY(OPT, n - 1)
update the point of the array OPT and w // 将OPT和w的指针首地址指针后移一位。
M2 = MAXMONEY(OPT_new, n - 3) + w_new[n - 2]
return max{M1, M2}
```

2 Node selection

You are given a binary tree, and each node in the tree has a positive integer weight. If you select a node, then its children and parent nodes cannot be selected. Your task is to find a set of nodes, which has the maximum sum of weight.

问题分析与证明：二叉树每个节点都有一个权重，如果选了其中一个节点，那么该节点的子节点和父节点都不能选，求权重和的最大值。

现考虑最小的情况，参考图3。该二叉树只有一个节点，则最大权重只有一种选法，根节点的权重就是最大值。考虑一个拥有三个节点的二叉树的情况。从根节点开始考虑，根节点可以选或不选，如果选择根节点，则根节点的左右子节点都不能选，如果不选根节点，则左右子节点可以选。所以根节点的最大权重为max{visit(root), visit(root->left) + visit(root->right)}。

现考虑一般情况，我们假设OPT(root)是求以root为根的二叉树最大权重总和函数，则OPT(root) = max{visit(root) + OPT(root->left->left) + OPT(root->left->right) + OPT(root->right->left) + OPT(root->right->right), OPT(root->left) + OPT(root->right)}

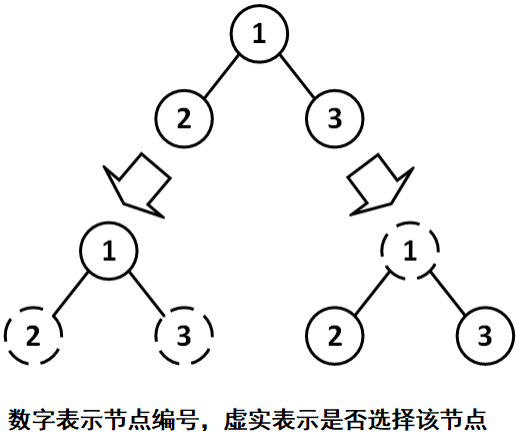

图 3: 最小情况

时间复杂度分析：对于该算法，相当于逐层遍历所有节点，所以时间复杂度为O(n)。

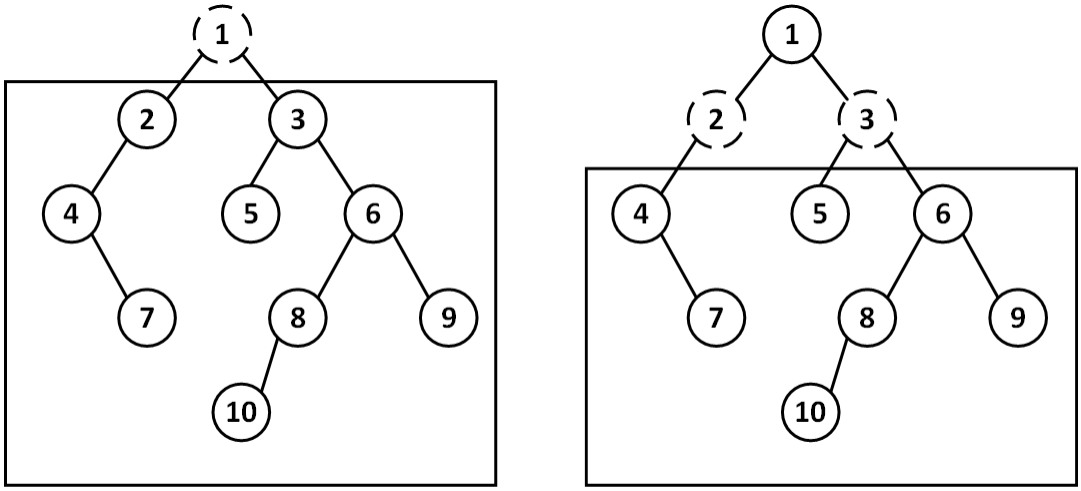

图 4: 一般情况

Dynamic Programming 3 Node selection

```
OPT(root)
if root == NULL then return 0
end if
if root->left == NULL then
    if root->right == NULL then
        node.append(root) //将节点加入队列
        return visit(root) //访问该节点求值
    else
        son_sum = OPT(root->right)
        grand_sum = visit(root) + OPT(root->right->left) + OPT(root->right->right)
        if son_sum < grand_sum then
            node.append(root)
        end if
        return max{son_sum, grandson_sum}
    end if
else
    if root->right == NULL then
        son_sum = OPT(root->left)
        grand_sum = visit(root) + OPT(root->left->left) + OPT(root->left->right)
        if son_sum < grand_sum then
            node.append(root)
        end if
        return max{son_sum, grand_sum}
    else
        son_sum = OPT(root->left) + OPT(root->right)
        grand_sum = visit(root) + OPT(root->left->left) + OPT(root->left->right) +
OPT(root->right->left) + OPT(root->right->right)
        if son_sum < grand_sum then
            node.append(root)
        end if
        return max{son_sum, grand_sum}
    end if
end if
```

3 Decoding

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.

问题分析与证明：本题求解的问题是给定一个数字字符串，按照对应规则转化为字母编码。先考虑一个简单情况，给出一个编码“1”，则其编码就是“A”。给出一个稍复杂的情况，“15320”，仍然从最后一个元素开始考虑。此时出现了一个特殊情况0，输入经过格式化处理，则0一定是前面的数的个位，即10或者20这种情况，因此取前一位元素，将这两个元素取出，进行译码，然后考虑“153”这个数组，最后一个数为3，不为0，可以对应一个编码，再取一个5，判断其是否可以组成另外一种编码。这里无法组成编码，因此3只能独立译码。此时字符串为“15”，最后一个是5，可以单独译码，再取前一位，组成15，可以组成另外一种译码，因此只有两种译码。

考虑一般情况，对于一个从1，2,…，i的子问题，其数字编码存储在num[i]数组里，用OPT[i]存储其最大字母编码种类，如果a[i] 为0，则OPT[i] = OPT[i - 2]。如果a[i - 1]*10 + a[i]小于10或者大于26，则OPT[i] = OPT[i - 1]。对于其他情况，其递归表达式为OPT[i] = OPT[i - 1] + OPT[i - 2]。

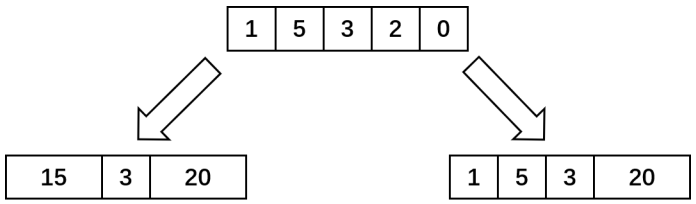

图 5: 特殊情况

时间复杂度分析：该算法最多为斐波那契递归的时间复杂度，为O(2^n)，因此可以对算法进行改进，从开始到结尾进行遍历。

改进后算法只需要遍历一次数组，因此其之间复杂度为O(n)。

Dynamic Programming 4 Decoding

```
SumDecoding(i):
if i == 0 then return 0
end if
if i == 1 then return 1
end if
if a[i] == 0 then
    OPT[i] = OPT[i - 2]
end if
if a[i - 1]*10 + a[i] < 10 or a[i - 1]*10 + a[i] > 26 then
    OPT[i] = OPT[i - 1]
end if
OPT[i] = OPT[i - 1] + OPT[i - 2]
SumDecoding(n)
```

Dynamic Programming 5 ImprovedDecoding

```
SumDecoding:
if n == 0 then return 0
end if
OPT[0] = 1
for i = 1 to n - 1 do
    if a[i] == 0 then
        OPT[i] = OPT[i - 2]
        continue
    end if
    if a[i - 1]*10 + a[i] < 10 or a[i - 1]*10 + a[i] > 26 then
        OPT[i] = OPT[i - 1]
        continue
    end if
    OPT[i] = OPT[i - 1] + OPT[i - 2]
end for
```