# Algorithm Design and Analysis: Assignment 2

钟赟 202028013229148

2020 年 11 月 18 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.

1. 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.

2. What if all houses are arranged in a circle?

## Answer

### 1.1 Optimal Substructure

我们从街上的第一个房子抢起，设 $OPT(i)$ 为抢到第 $i$ 个房子时获得的最大值，设 $money[i]$ 表示第 $i$ 个房子的钱数。则如果我们可以抢劫第 $i$ 个房子，则没有抢劫第 $i-1$ 个房子；如果我们抢劫了第 $i-1$ 个房子，则不能抢劫 $money[i]$。于是有如下 DP 递推表达式：

$$OPT(i) = max \begin{cases} money[i] + OPT(i-2), \\ OPT(i-1) \end{cases}$$

若房子的分布为环形，则第一间房子和最后一间房子相邻，不能同时抢劫。在上面算法的基础上，假设共有 $n$ 个房子，分别计算前 $n-1$ 间房子和后 $n-1$ 间房子所能抢劫的最大值，再将两者取最大值，即为环形房子能抢劫到的最大值。

### 1.2 Algorithm Description

伪代码见 Algorithm 1 。

### 1.3 Correctness Proof

当 $len = 2$ 时，只能抢劫其中一个房子，算法返回两个房子中钱数较大的数额，显然正确。

**Algorithm 1** MONEY_ROBBING algorithm

```
function MAX_MONEY_ROBBING_LINE(money, len)
    if len <= 2 then
        return max(money);
    end if
    OPT[0] = money[0]; OPT[1] = max (money[0], money[1]);
    for i = 2 to len do
        OPT[i] = max (OPT[i − 1], OPT[i − 2] + money[i]);
    end for
    return OPT[len − 1];
end function
function MAX_MONEY_ROBBING_CIRCLE(money, len)
    if len <= 2 then
        return max(money);
    end if
    return max(MAX_MONEY_ROBBING_LINE(money[0, len − 1], len − 1),
    MAX_MONEY_ROBBING_LINE(money[1, len], len − 1));
end function
```

当 $len \geq 2$ 时，我们只需证明 DP 表达式的正确性即可。对于第 $i$ 个房子，如果我们不抢劫它的钱，显然目前能够抢劫的最大值为 $OP(i-1)$；如果我们抢劫它的钱，意味着第 $i-1$ 个房子的钱不能抢，否则会触发报警器，故此时能抢劫的最大值为 $OPT(i-2)+money[i]$。每步都选择两者的最大值作为抢劫到房子 $i$ 的最优方案，则最终可以得到抢劫所有房子的最大值。

同理，当房子为环形时，我们需要注意的是第一间房子与最后一间房子不能同时抢劫，其余情况与之前相同。因此，分别掐头去尾调用两次原来的抢劫算法即可保证不会同时抢劫第一间和最后一间房子。

因此，算法正确。

### 1.4 Complexity

算法 MAX_MONEY_ROBBING_LINE 和 MAX_MONEY_ROBBING_CIRCLE 均遍历了一次数组，复杂度为 $\mathcal{O}(n)$。

## 2 Largest Divisible Subset

Given a set of distinct positive integers, find the largest subset such that every pair $(S_i, S_j)$ of elements in this subset satisfies: $S_i\%S_j = 0$ or $S_j\%S_i = 0$ Please return the largest size of the subset.

Note: $S_i\%S_j = 0$ means that $S_i$ is divisible by $S_j$

# Answer

## 2.1 Optimal Substructure

首先，我们将数组按升序排列。对于有序列表 $x_1, x_2, \ldots, x_n$ ，我们定义 $OPT(x_i)$ 为到达 $x_i$ 为止的最大的可整除集合大小。对于元素 $x_i, x_j, x_k$fi$i > j > k$ ，已知有 $x_i \% x_j == 0$ 且 $x_j, x_k$ 在子集中，则若有 $x_i \% x_j == 0$ ，则有 $x_i \% x_k == 0$ ，可将 $x_i$ 加入该子集中。故该问题具有最优子结构。

$OPT(x_i) = \max(OPT(j) + 1), j < i, array[i] \,\%\, array[j] == 0$

## 2.2 Algorithm Description

伪代码见 Algorithm 2。

**Algorithm 2** MONEY_ROBBING algorithm

```
1:  function LARGEST_DIVISIBLE_SUBSET(array, len)
2:      Sort array in ascending order;
3:      max_size = 1;
4:      for i = 1 to len − 1 do
5:          for j = 0 to i do
6:              if array[i] % array[j] == 0 and OPT[j] + 1 > OPT[i] then
7:                  OPT[i] = OPT[j] + 1;
8:              end if
9:          end for
10:         if OPT[i] > max_size then
11:             max_size = OPT[i];
12:         end if
13:     end for
14:     return max_size;
15: end function
```

## 2.3 Correctness Proof

根据 2.1 中的算法描述，我们已知下式成立：

$x\%y == 0 \wedge y\%z == 0 \Rightarrow x\%z == 0, where\ x > y > z.$

因此，在一个满足题设关系的子集中，上式可以保证新加入子集的元素依然满足题设中元素之间的关系条件。最优子结构正确，故动态规划的地推表达式通过枚举小于 $x$ 的所有元素来更新最大可整除子集可保证整个算法的正确性。如果需要求出最大可整除子集合的元素，只需进行回溯即可。

## 2.4 Complexity

在算法的主循环中，我们需要遍历每个元素，并对每个元素 $x_i$ ，枚举所有小于 $x_i$ 的元素，故时间复杂度为 $\mathcal{O}(n^2)$ 。

# 3 Coin Change

You are given coins of di erent denominations and a total amount of money amount. Write a function to compute the total number of ways to make up that amount using some of those coins.

Note : You may assume that you have an in nite number of each kind of coin.

# Answer

## 3.1 Optimal Substructure

我们定义 $OPT(i)$ 为组成金额 $i$ 所需要的最少硬币数量。假设计算 $OPT(i)$ 之前，我们已经知道 $OPT(0), OPT(1), \ldots, OPT(i-1)$，那么我们只需要求出用不同硬币凑成金额 $i$ 所使用的硬币数的最小值即可。设硬币集合为 $x_1, x_2, \ldots, x_n$ ，则动态规划方程为：

$OPT(i) = F(i) = \min_{j=1\ldots n} OPT\,(i - x_j) + 1$

## 3.2 Algorithm Description

伪代码见 Algorithm 3。

**Algorithm 3** COIN_CHANGE algorithm

```
1: function COIN_CHANGE(total, coins[0, ..., n − 1])
2:     OPT[0] = 0;
3:     for i = 1 to total do
4:         for j = 0 to n − 1 do
5:             OPT[i] = min (OPT[i], OPT[i − coins[j]] + 1);
6:         end for
7:     end for
8:     return OPT[total];
9: end function
```

## 3.3 Correctness Proof

已知该问题具有最优子结构，在计算 $OPT(i)$ 时，我们枚举所有硬币。对于硬币 $x_j$ ，我们需要从 $i - x_j$ 这个金额的状态 $OPT(i - x_j)$ 转移过来，再算上枚举这枚硬币数量 1 的贡献。由于要求硬币数量最少，所以 $OPT(i)$ 为前面能转移过来的状态的最小值加上枚举的硬币数量 1 。

因此可以得知， $OPT(total)$ 为组成金额为 $total$ 所需要的最少硬币数。算法正确。

## 3.4 Complexity

设总金额是 $S$ ，硬币种类为 $n$ ，根据算法 3 中的二重循环，得知时间复杂度为 $\mathcal{O}(Sn)$ 。